Multitask Learning

Rich Caruana

23 September 1997

CMU-CS-97-203

School of Computer Science

Carnegie Mellon University

Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Thesis Committee:

Tom Mitchell, Chair

Herb Simon

Dean Pomerleau

Tom Dietterich, Oregon State

This work was supported by NSF Grant BES-9402439, by Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and DARPA Grant F33615-93-1-1330, by the Agency for Health Care Policy and Research grant HS06468, and by the Justsystem Pittsburgh Research Center. The U.S. Government is authorized to reproduce reprints for government use. The views contained herein are those of the authors and do not necessarily represent Wright Laboratory, the National Science Foundation, or the U.S. Government.

Keywords: machine learning, neural networks, k-nearest neighbor, multitask learning, inductive bias, medical decision making, pneumonia, ALVINN, autonomous vehicle navigation, pattern recognition, inductive transfer, learning-to-learn

Dedicated to my parents, for fostering my interest in science,
to Herb Simon, for teaching me to ask bigger questions, and
to Diane, for being willing to come to Pittsburgh.

Acknowledgements

Thanks go first to my advisors, Tom Mitchell and Herb Simon, and to the other two members of my thesis committee, Dean Pomerleau and Tom Dietterich. They did a great job pushing, prodding, questioning, interpreting, and suggesting as the research progressed.

I'd also like to thank Greg Cooper, Michael Fine, Constantin Alifers, Tom Mitchell, and other members of the Pitt/CMU Cost-Effective Health Care group for help with the Pneumonia Databases; Dean Pomerleau for the use of his road simulator; Tom Mitchell, Reid Simmons, Joseph O'Sullivan, and other members of the Xavier Robot Project for help with Xavier the robot; and Tom Mitchell, Dayne Freitag, David Zabowski, and other members of the Calendar Apprentice Project for help using the CAP data.

Thanks also go to the Mitre Group for the Aspirin/Migraines Neural Net Simulator and to Geoff Hinton's group at the University of Toronto for the Xerion Neural Net Simulator.

Rankprop was developed with Shumeet Baluja and Tom Mitchell. The work on input features that are more useful as extra output tasks is joint work with Virginia de Sa.

This research benefited from discussions with many people, most notably Shumeet Baluja, Justin Boyan, Tom Dietterich, Virginia de Sa, Dayne Frietag, Scott Fahlman, Ken Lang, Tom Mitchell, Andrew Moore, Dean Pomerleau, Herb Simon, Sebastian Thrun, Dave Touretzky, and Raul Valdes-Perez. It also benefited from the feedback I received from many friends who sat through nearly a dozen pizza seminars on this and related topics.

Finally, I'd like to thank Tom Mitchell again, who served both as a doubting Thomas in the early days, and as an unwavering source of support and encouragement in the later days. Thanks Tom.

Abstract

Multitask Learning is an approach to inductive transfer that improves learning for one task by using the information contained in the training signals of other *related* tasks. It does this by learning tasks in parallel while using a shared representation; what is learned for each task can help other tasks be learned better. In this thesis we demonstrate multitask learning for a dozen problems. We explain how multitask learning works and show that there are many opportunities for multitask learning in real domains. We show that in some cases features that would normally be used as inputs work better if used as multitask outputs instead. We present suggestions for how to get the most out of multitask learning in artificial neural nets, present an algorithm for multitask learning with case-based methods like k-nearest neighbor and kernel regression, and sketch an algorithm for multitask learning in decision trees. Multitask learning improves generalization performance, can be applied in many different kinds of domains, and can be used with different learning algorithms. We conjecture there will be many opportunities for its use on real-world problems.

Contents

"Your ability to juggle many tasks will take you far."

– *Fortune Cookie*

Chapter 1

Introduction

Multitask Learning is an approach to inductive transfer that improves learning for one task by using the information contained in the training signals of other *related* tasks. It does this by learning tasks in parallel while using a shared representation; what is learned for each task can help other tasks be learned better. In this thesis we demonstrate multitask learning for a dozen problems. We explain how multitask learning works and show that there are many opportunities for multitask learning in real domains. We show that in some cases features that would normally be used as inputs work better if used as multitask outputs instead. We present suggestions for how to get the most out of multitask learning in artificial neural nets, present an algorithm for multitask learning with case-based methods like k-nearest neighbor and kernel regression, and sketch an algorithm for multitask learning in decision trees. Multitask learning improves generalization performance, can be applied in many different kinds of domains, and can be used with different learning algorithms. We conjecture there will be many opportunities for its use on real-world problems.

1.1 Motivation

The world we live in requires us to learn many things. These things obey the same physical laws, derive from the same human culture, are preprocessed by the same sensory hardware.... Perhaps it is the *similarity* of the many tasks we learn that enables us to learn so much with so little experience.

You learn to play tennis in a world that asks you to learn many other things. You also learn to walk, to run, to jump, to exercise, to grasp, to throw, to swing, to recognize objects, to predict trajectories, to rest, to talk, to read, to study, to practice, etc. These tasks are not the same—running in tennis is different from running on a track—yet they are related. Perhaps the similarities between the thousands of tasks you learn are what enable you to learn any one of them, including tennis, given so little training data.

An artificial neural network (or a decision tree, ...) trained tabula rasa on a single, isolated, difficult task is unlikely to learn it well. For example, a net with a 1000x1000 pixel input retina is unlikely to learn to recognize complex objects in real-world scenes given the number of training patterns likely to be available. Might it be better to require the learner to learn many things simultaneously? If the tasks can share what they learn, the learner may find it is easier to learn them together than in isolation. Thus, if we simultaneously train a net to recognize object outlines, shapes, edges, regions, subregions, textures, reflections, highlights, shadows, text, orientation, size, distance, etc., it may learn better to recognize complex objects in the real world. We call this approach to learning Multitask Learning (MTL).

The thesis of this research is that a task will be learned better if we can leverage the information contained in the training signals of other related tasks during learning. We use the term "task" to refer to a target function that will be learned from a sample of points (called the training set) drawn from the target function. A training set consists of a finite number of data points defined on a vector of k attributes (the input features),

$$\mathbf{X}_n = (X_{1,n}, X_{2,n}, X_{3,n}, \ldots, X_{k,n}), \tag{1.1}$$

each with an associated target value y_n (or training signal):

$$\mathbf{TrainingSet} = \{(y_n, \mathbf{X}_n)\ n = 1, \ldots, N\} \tag{1.2}$$

In supervised learning the goal is to learn models that accurately predict new values y for future instances of $\mathbf{X}$ (the test set). We call the task we wish to learn better the **main task**. The related tasks whose training signals are used by multitask learning to learn the main task better are the **extra tasks**.[1] Usually, we do not care how well extra tasks are learned; their sole purpose is to help the main task be learned better. We call the union of the main task and all extra tasks a **domain**. In this thesis, we usually restrict ourselves to domains where tasks are defined on a common set of k input features $(X_{1,n}, X_{2,n}, \ldots, X_{k,n})$ (though some extra tasks may be functions of only a subset of the k features). Not all tasks in a domain will necessarily be beneficial to the main task. One goal of this thesis is to develop learning methods that are robust to interference between tasks, i.e., methods that do not allow the performance on the main task to be made worse by extra tasks that are not helpful to it. Another goal is to develop heuristics to help us select tasks from the domain that are likely to be helpful.

1.2 MTL with Backprop Nets

Hinton proposed that generalization in artificial neural networks improves if networks learn to represent underlying regularities of the domain [Hinton 1986]. A learner that learns many related tasks *at the same time* can use these tasks as inductive bias for each other and thus better learn the domain's regularities. This can make learning more accurate and may allow hard tasks to be learned that could not be learned in isolation.

Figure 1.1 shows four separate artificial neural nets. Each net is a function of the same eight inputs and has one output. (An "output" in a backprop net is where the training signals y for the target function are fed into the net during training, and where predictions of y are read from the net during testing. In this thesis we often use the term "output" as a synonym for task or target function.) Backpropagation is applied to these nets by training each net in isolation. Because the four nets are not connected what is learned by one net does not affect the other nets. We call this Single Task Learning (STL).

Figure 1.2 shows a single net with the same eight inputs as the four nets in Figure 1.1, but which has four outputs, one from each of the nets in Figure 1.1. Note that the four outputs are fully connected to a hidden layer that they share.[2] Backpropagation is done in

[1] As is common in machine learning, we make no assumption that the k features are necessary, nor sufficient, inputs for learning accurate models of the tasks, including the main task.

[2] More complex architectures than a fully connected hidden layer sometimes work better. See Section 6.3.2

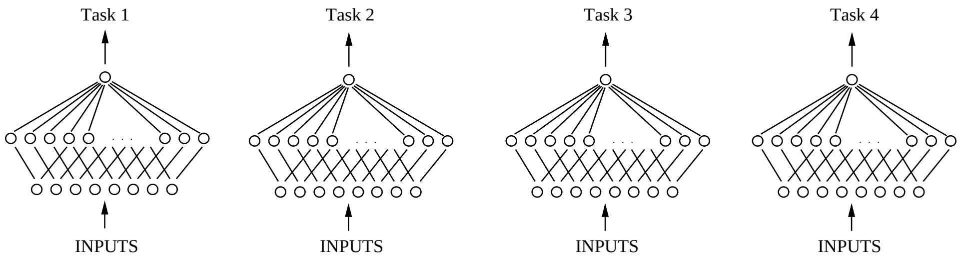


Figure 1.1: Single Task Backprop (STL) of four tasks with the same inputs.

parallel on the four outputs in this MTL net. Because the four outputs share a common hidden layer, it is possible for internal representations that develop in the hidden layer for one task to be used by other tasks. Sharing what is learned by different tasks while tasks are trained in parallel is the central idea in multitask learning [Suddarth & Kergosien 1990; Dietterich, Hild & Bakiri 1990, 1995; Suddarth & Holden 1991; Caruana 1993a, 1993b, 1994, 1995; Baxter 1994, 1995, 1996; Caruana & de Sa 1996].

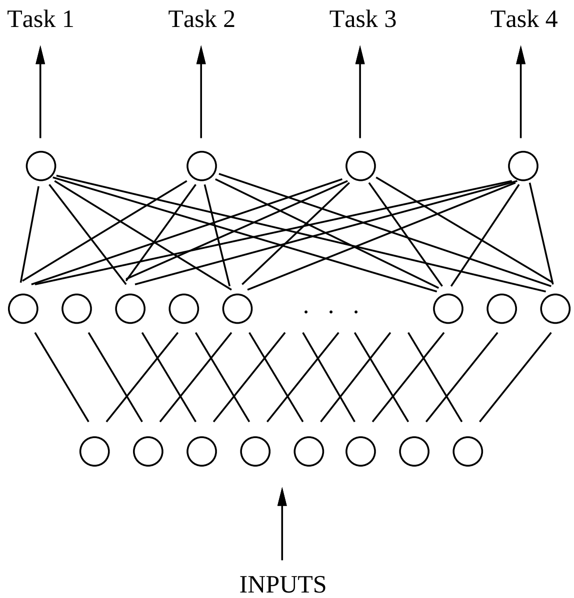


Figure 1.2: Multitask Backprop (MTL) of four tasks with the same inputs.

Multitask learning is a collection of learning algorithms, analysis methods, and heuristics, not a single learning algorithm. It is an approach to inductive transfer (using what is learned for one problem to help another problem) that emphasizes learning multiple tasks in **parallel** while using a shared representation so that what is learned by all tasks is available to the main task. By learning multiple tasks in parallel, MTL is able to use the extra information about the domain contained in the training signals of the related tasks. In backpropagation, MTL allows features developed in the hidden layer for one task to be used by other tasks. It also allows features to be developed to support several tasks that would not have been developed in any STL net trained on the tasks in isolation. Importantly, MTL-backprop also allows some hidden units to become specialized for just one or a few tasks; other tasks can ignore hidden units they do not find useful by keeping the weights connected to them small.

1.3 A Motivating Example

Consider the following boolean functions defined on eight bits, $B_1 \cdots B_8$:

$$\begin{aligned}
Task1 &= B_1 \vee Parity(B_2 \cdots B_6) \\
Task2 &= \neg B_1 \vee Parity(B_2 \cdots B_6) \\
Task3 &= B_1 \wedge Parity(B_2 \cdots B_6) \\
Task4 &= \neg B_1 \wedge Parity(B_2 \cdots B_6)
\end{aligned}$$

where "B_i" represents the ith bit, "$\neg$" is logical negation, "$\vee$" is disjunction, "$\wedge$" is conjunction, and "$Parity(B_2 \cdots B_6)$" is the parity of bits 2–6. Bits B_7 and B_8 are not used by the functions.

These tasks are related in several ways:

- they are all defined on the same inputs, bits $B_1 \cdots B_8$;
- they all ignore the same bits in the inputs, B_7 and B_8;
- each is defined using a common computed subfeature, $Parity(B_2 \cdots B_6)$;
- on those inputs where Task 1 must compute $Parity(B_2 \cdots B_6)$, Task 2 does not need to compute parity, and vice versa (when $B_1 = 1$, Task 1 does not need to compute $Parity(B_2 \cdots B_6)$, but Task 2 does, and vice versa);

- on those inputs where Task 3 must compute $Parity(B_2 \cdots B_6)$, Task 4 does not need to compute parity, and vice versa.

We can train artificial neural nets on these tasks with backpropagation. Bits $B_1 \cdots B_8$ are the inputs to the net. The task values computed for the functions are the target outputs. We create a data set for these tasks by enumerating all 256 combinations of the eight input bits, and computing for each setting of the bits the task signals for Tasks 1, 2, 3, and 4 using the definitions above. This yields 256 different cases, with four different training signals for each case.

From the 256 synthesized cases, we randomly sample 128 cases and place them in a training set. We use the remaining 128 cases as a test set. The test set is not used for training, but is used to evaluate the trained nets on instances on which they were not trained. For simplicity, we ignore the complexity of early stopping and the need to set aside halt sets to determine when to stop training. The tasks have been carefully devised so overfitting is not too significant.

We've done an experiment where we train Task 1 on the three nets shown in Figure 1.3. In Figure 1.3,left Task 1 is trained alone. This is STL-backprop of Task 1. In Figure 1.3,center Task 1 is trained on a net with Task 2. This is MTL-backprop with two tasks. In Figure 1.3,right Task 1 is trained with Tasks 2, 3, and 4. This is MTL-backprop with four tasks. How well will Task 1 be learned by the different nets?

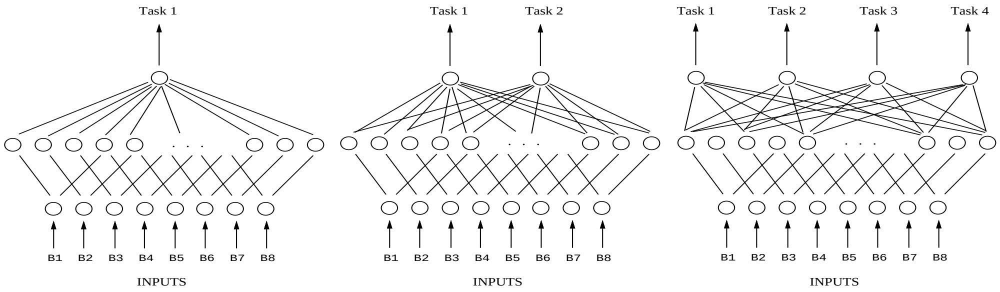


Figure 1.3: Three Neural Net Architectures for Learning Task 1

All nets are fully connected feed-forward nets with 8 input units, 100 hidden units, and 1–4 outputs. Where there are multiple outputs, each output is fully connected to the 100 hidden units. Nets are trained using backpropagation with MITRE's Aspirin/MIGRAINES

6.0 with learning rate = 0.1 and momentum = 0.9. Weights in the nets are updated each epoch, i.e., after each full pass through the training set. Every 5000 epochs we evaluate the performance of the nets on the held out test set. We measure performance two ways. First, we measure the root-mean-squared error (RMSE) of the output with respect to the target values. This is the error criterion being optimized by backpropagation. We also measure the percent accuracy of the output in predicting the boolean values of the function. If the net output is less than 0.5, it is treated as a prediction of 0, otherwise it is treated as a prediction of 1.

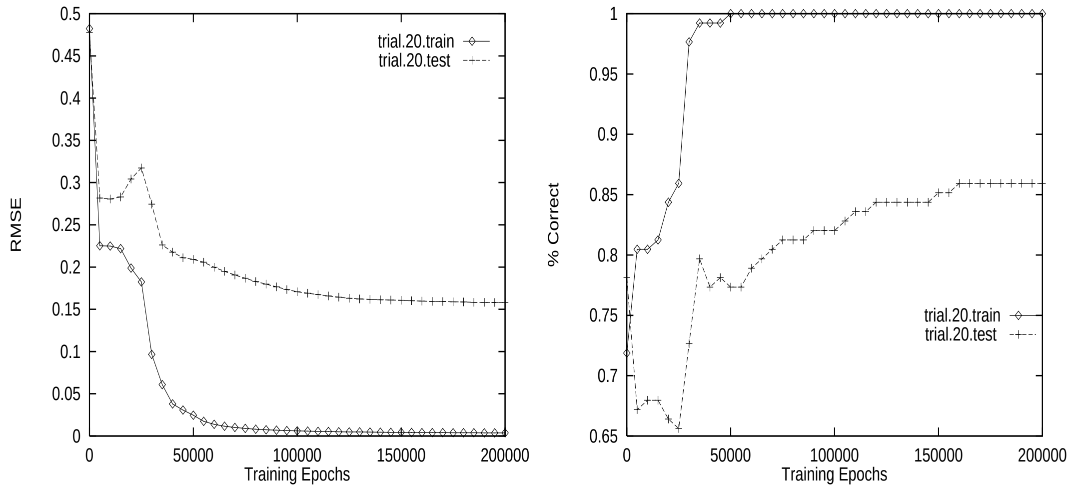


Figure 1.4: Training curves for one run of backprop STL on Task 1

Figure 1.4 shows the training curves for a single run of backpropagation on the STL net for Task 1. The graph on the left shows the RMSE error as a function of training epochs. Lower RMS error is better. The graph on the right shows the percent accuracy of the model. Both graphs show performance on both the training set (the data actually being used by backpropagation to train the net) and the test set. As often happens, the nets learn the training set so well that RMS error on the training set approaches 0.0 and accuracy approaches 100%. Performance on the independent test set, however, is not as good. The net has overfit to the training set.

We perform 25 independent trials of sampling training and test sets from the 256 cases, and training and evaluating nets on these sets. Different random seeds are used to generate the training and test sets and to initialize the nets in each trial. For each trial, we train three nets: an STL net for Task 1, an MTL net for Tasks 1 and 2, and an MTL net for Tasks 1–4. For this experiment we measure performance only on the output for Task 1. When there are extra outputs for Task 2 or Tasks 2–4, these are trained with backpropagation, but ignored when the net is evaluated. The sole purpose of the extra outputs is to affect what is learned in the hidden layer these outputs share with Task 1.

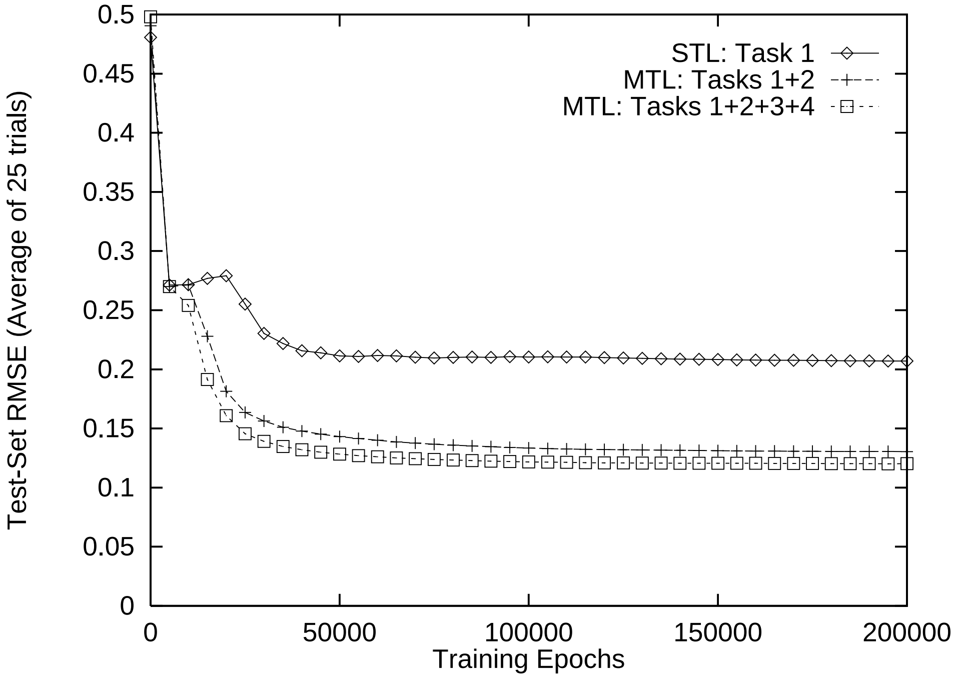


Figure 1.5: RMSE Test-set Performance of Three Different Nets on Task 1.

Figure 1.5 shows the test-set RMSE training curves for the three nets on Task 1. The three curves in the graph are each the average of 25 trials.[3] RMSE on Task 1 is reduced when Task 1 is trained on a net simultaneously trained on other related tasks. RMSE is reduced when Task 1 is trained with Task 2, and is further reduced when Tasks 3 and 4 are added. Training multiple tasks on one net does not increase the number of training patterns seen by that net. Each net sees exactly the same training cases. *The MTL nets*

[3]Average training curves can be misleading, particularly if training curves are not monotonic. For example, it is possible for method A to always achieve better error than method B, but for the average of method A to be everywhere worse than the average of method B because the regions where performance on method A is best do not align, but do align for method B. Before presenting average training curves, we always examine the individual curves to make sure the average curve is not misleading.

do not see more training cases; they receive more training signals with each case.

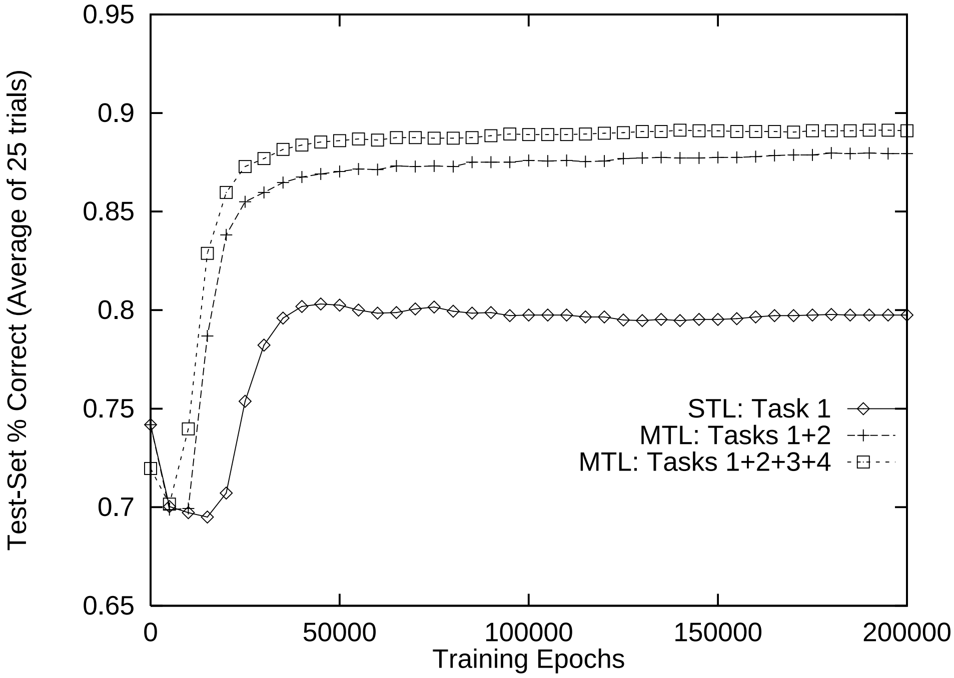


Figure 1.6: Test-set Percent Correct of Three Different Nets on Task 1.

Figure 1.6 shows the test-set percent correct on Task 1 for the three different nets. Again, each curve is the average of 25 trials. Table 1.1 summarizes the results of examining the training curve from each trial.

Table 1.1: Test-set performance on Task 1 of STL of Task 1, MTL of Tasks 1 and 2, and MTL of Tasks 1, 2, 3, and 4. *, **, *** indicate performance is statistically better than STL at .05, .01, .001, or better, respectively.

NET	STL: 1	MTL: 1+2	MTL: 1+2+3+4
Root-Mean-Squared-Error	0.211	0.134 ***	0.122 ***
Percent Correct	79.7%	87.5% ***	88.9% ***

On average, Task 1 has boolean value 1 75% of the time. A simple learner that learned to predict the output value 1 all the time should achieve about 75% accuracy. When trained alone (STL), performance on Task 1 is about 80%. When Task 1 is trained with Task 2, performance increases to about 88%. When Task1 is trained with Tasks 2, 3, and 4, performance increases further to about 90%.

Qualitatively similar graphs and measurements result if we examine the performance of any one of the four tasks when trained alone or in combination with the other tasks. There is nothing unique about Task 1; all the tasks help, and are helped by, each other. Performance is poorest when one task is trained in isolation, i.e., by STL. Better generalization is achieved when a single net is trained on several of the tasks at the same time. The more tasks trained on the MTL net, the better the performance.

1.4 A Brief Analysis

Why is each task learned better if trained on a net learning other related tasks at the same time? Is it because the tasks are related and what is learned for the tasks helps the other tasks, or is it because backpropagation just works better with nets that have more outputs, even when the outputs are not related? We ran a number of experiments to verify that the performance increase with MTL is due to the fact that the tasks are related, and not just a side effect of training multiple outputs on one net.

Adding noise to neural nets sometimes improves their generalization performance [Holden 1992]. To the extent that MTL tasks are *uncorrelated*, their contribution to the aggregate gradient may appear as noise to other tasks and this might improve generalization. To see if this effect explains the benefits we see from MTL, in the first experiment we train Task 1 on a net with three *random* tasks. This lets us test whether it is the *lack* of relationship between Tasks 1–4 that improves performance.

A second effect to be concerned about is that adding tasks might change backprop's weight update dynamics to somehow favor nets with more tasks. For example, having more outputs tends to increase the effective learning rate on the input-to-hidden layer weights because the gradients from the multiple outputs add at the hidden layer. Does performance improve on MTL using Tasks 1–4 just because extra outputs help backprop train multilayer nets? To test for this, we train an MTL net with four copies of Task 1. Each of the four outputs receives exactly the same training signals. This is a degenerate form of MTL where no extra information is given to the net by the extra task training signals.

A third effect that needs to be ruled out is net capacity. 100 hidden units is a lot for these tasks. Does the MTL net, which has to share the 100 hidden units among four tasks, generalize better just because each task has fewer hidden units? To test for this, we train Task 1 on STL nets with 200 hidden units and with 25 hidden units. This will tell us if generalization would be better with more or less capacity.

Finally, we run a fourth experiment based on the heuristic used in [Valdes-Perez & Simon 1994] to discover complex patterns in data. In this experiment we shuffle the training signals (the target output values) for Tasks 2, 3, and 4 in the data set before training an MTL net on the four outputs. We shuffle the training signals for Tasks 2, 3, and 4 independently, i.e., we do not mix the training signals between the tasks. Shuffling randomly reassigns the target values y_n to the input vectors $\mathbf{X}_n$) in the training set for each task. Task 1 is not affected by this shuffling and is still the same function of the inputs. The training signals for outputs 2–4 have the same distributions as those for Tasks 2–4, but they are no longer related to Task 1 because the functional relationship between the inputs and those task signals has been randomized. This is a powerful test that has the potential to rule-out most mechanisms that do not depend on there being relationships between the tasks.

We ran each experiment 25 times using exactly the same data sets used in the previous section. Figure 1.7 shows the generalization performance on Task 1 in the four experiments. For comparison, the performance of of STL, MTL with Tasks 1 and 2, and MTL with Tasks 1–4 from the previous section are also shown in the figure.

When Task 1 is trained with random extra tasks, performance on Task 1 drops below the performance on Task 1 when it is trained alone on an STL net. We conclude MTL of Tasks 1–4 probably does not learn Task 1 better because the *differences* between the tasks promotes generalization by adding noise to the learning process.

When Task 1 is trained with three additional copies of Task 1, the performance is comparable to that when Task 1 is trained alone with STL.[4] We conclude that MTL does not learn Task 1 better just because backprop works better when there are multiple outputs.

[4]We sometimes observe that training multiple copies of a task on one net does improve performance on that task. When we have observed this, the size of the benefit is not large enough to explain away the benefits observed with MTL. But it is an interesting and surprising effect, as the improvement is gained without any additional information being given to the net. The most likely explanation is that the multiple connections to the hidden layer allow different hidden layer predictions to be averaged and thus act as a boosting mechanism [Perrone 1994, ...].

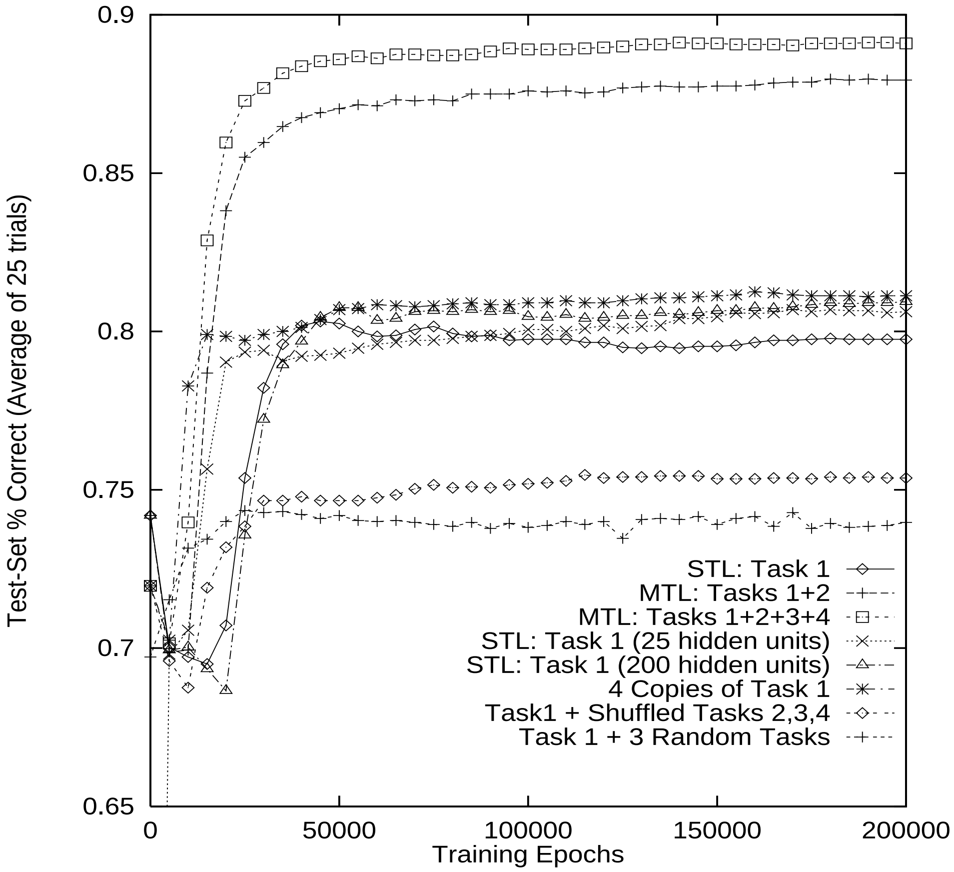


Figure 1.7: RMSE test-set performance of Task 1 when trained with: MTL with three random tasks; MTL with three more copies of Task 1; MTL with shuffled training signals for Tasks 2–4; STL on nets with 25 or 200 hidden units.

When Task 1 is trained on an STL net with 25 hidden units, performance is comparable to the performance with 100 hidden units. Moreover, when Task 1 is trained on an STL net with 200 hidden units, it is slightly better. (The differences between STL with 25, 100, and 200 hidden units are not statistically significant.) We conclude that performance on Task 1 is relatively insensitive to net size for nets between 25 and 200 hidden units, and, if anything, Task 1 would benefit from a net with more capacity, not one with less capacity. Thus it is unlikely that MTL on Tasks 1–4 performs better on Task 1 because Tasks 2–4 are using up extra capacity that is hurting Task 1. See Appendix 1 for a discussion of the effect of excess capacity on generalization in nets trained with backpropagation.

When Task 1 is trained with training signals for Tasks 2–4 that have been shuffled, the performance of MTL drops below the performance of Task 1 trained alone on an STL net.

Clearly the benefit we see with MTL on these problems is not due to some accident caused by the extra outputs. The extra outputs must be *related* to the main task to help it.

These experiments rule out most—if not all—explanations for why MTL outperforms STL on Task 1 that do not require Tasks 2–4 be related to Task 1. So why is Task 1 learned better when trained in parallel with Tasks 2–4?

One reason is that Task 1 needs to learn to compute a subfeature, $Parity(B_2 \cdots B_6)$, that it shares with Tasks 2–4. Tasks 2–4 give the net information about this subfeature that it would not get from Task 1 alone. For example, when $B_1 = 1$, the training signal for Task 1 contains no information about $Parity(B_2 \cdots B_6)$ because the disjunction operator that combines B1 with the Parity subfeature is insensitive to the value of the Parity subfeature when $B = 1$. We say B_1 *blocks* $Parity(B_2 \cdots B_6)$ when $B_1 = 1$. But the training signals for Task 2 provide information about the Parity subfeature in exactly those cases where Task 1 is blocked. Thus the hidden layer in a net trained on both Tasks 1 and 2 gets twice as much information about the Parity subfeature as a net trained on one of these tasks, despite the fact that they see exactly the same training cases. The MTL net is getting more information with each training case.

Another reason why MTL helps Task 1 is that all the tasks are functions of the same inputs, bits $B_1 \cdots B_6$, and ignore the same inputs, B_7 and B_8. Feature selection can be difficult when learning a complex function from a finite sample. Because the tasks overlap on the features they use and don't use, the MTL is better able select which input features to use. (As we discuss later in Section 3.3.3, this feature selection effect goes away if the different tasks are completely different functions of the same features.)

A third reason why MTL helps Task 1 is that there are relationships between the way the different tasks use the inputs that promote learning good internal representations. For example, all the tasks logically combine input B_1 with a function of inputs $B_2 \cdots B_6$. This similarity tends to prevent the net from learning internal representations that, for example, directly combine bits B_1 and B_2. A net trained on all the tasks together is biased to learn more modular, more correct internal representations that support the multiple tasks. We conjecture that this bias towards modular internal representations helps reduce the net's tendency to learn spurious correlations that occur in any finite training sample: there may be a random correlation between bit B_3 and the output for Task 1 that looks fairly strong in this one training set, but if that spurious correlation does not also help other Tasks, it is less likely to be learned. By biasing the net to learn hidden representations that support multiple tasks, overfitting to spurious details of the training set is reduced.

Before continuing, note that when Task 1 is trained with extra tasks that are random functions, or extra related tasks whose task signals have been shuffled, performance dropped below the performance of training the Task alone on an STL net. This is the first example of MTL hurting performance because the extra tasks are *not* related to the main task. This issue comes up again later in Section 4.3, Section 4.5, and in Sections 9.2.4–9.2.6.

1.5 A Second Example

One might be tempted to conclude from the previous example that MTL will only help on hard problems, such as functions that internally use hard-to-learn subfeatures like 5-bit parity. Is MTL worthwhile with simpler problems? One might also be tempted to conclude that unused inputs are important to success with MTL. Does MTL help when there is no feature selection problem? Finally, one might be tempted to conclude that most of the benefit from MTL comes from adding the first related task, particularly if that task "completes" the dataset on an important subfeature as Task 2 did for 5-bit Parity in the last example. Can MTL benefit from *many* tasks, particularly when many of those tasks don't have that special relationship?

Consider these boolean functions:

$$\begin{aligned}
TaskA &= \quad B1 \lor (2 * NO_BITS(B2 \cdots B4) < NO_BITS(B5 \cdots B8)) \\
TaskB &= \neg B1 \lor (2 * NO_BITS(B2 \cdots B4) < NO_BITS(B5 \cdots B8)) \\
TaskC &= \quad B1 \land (2 * NO_BITS(B2 \cdots B4) < NO_BITS(B5 \cdots B8)) \\
TaskD &= \neg B1 \land (2 * NO_BITS(B2 \cdots B4) < NO_BITS(B5 \cdots B8))
\end{aligned}$$

where $NO\_BITS()$ is a simple procedure that counts the number of bits set to 1 in its argument (e.g., $NO\_BITS(0101) = 2$), and $(? <?)$ is the standard less-than boolean conditional test. We multiply $NO\_BITS(B_2 \cdots B_4)$ (which is a function of three bits) by 2

before comparing it to $NO\_BITS(B_5 \cdots B_8)$ (which is a function of four bits) to balance the outcome of the conditional test so that it is true 50% of the time. (This balancing cannot be achieved if the number of bits in the two sides are the same. This balancing is not essential, but it simplifies the functions pedagogically.) Note that bits B_7 and B_8 are now used by the functions; there are no don't care bits.

Tasks A–D are much easier to learn than Tasks 1–4 because the principle subfeature they compute, $2 * NO\_BITS(B_2 \cdots B_4) < NO\_BITS(B_5 \cdots B_8)$, is much easier to learn than Parity. To prevent all nets trained on these functions from achieving nearly 100% generalization accuracy, we reduce the training set size from the 128 cases used with Tasks 1–4 to only 32 cases. The remaining 224 cases from each trial are used as a test sets.

Figure 1.8 is the test-set RMSE of Task A trained on three nets like those in Figure 1.3. The first net is STL of Task A. The second net is Task A trained with Task B. The third net is Task A trained with Tasks B–D. Figure 1.9 is the test-set percent correct of Task A trained on the three nets.

As before, MTL outperforms STL. It is better to train Task A on a net with other related tasks than to train it alone. Unlike before, the benefit of adding Task B to the Task A net is small. This is in spite of the fact that Task A and Task B share the same blocking relationship of the common subfeature, $2 * NO\_BITS(B_2 \cdots B_4) < NO\_BITS(B_5 \cdots B_8)$, that Tasks 1 and 2 shared in the previous problems. And unlike before, the largest increase in benefit to Task A occurs when Tasks C and D are added to the net.

Table 1.2 summarizes the performance of the three nets. The accuracy on Task A trained alone is about 90%. When Task 2 is added to the net, accuracy increases by a half percent or less. When Task A is trained with Tasks B–D, accuracy increases to about 92%. The improvement due to MTL with Tasks A–D is less than that observed for Tasks 1–4. Part of the reason why the improvement is less is because Tasks A–D are so much easier to learn. Even with only 32 cases in the training set, accuracy is higher on STL TASK A than on MTL with Tasks 1–4.

Because Tasks A–D do not have a feature selection problem, achieving good performance with MTL does not depend on the tasks having a feature selection problem. Also, most of the MTL benefit observed for Tasks A–D did not come when Task A's use of the blocked

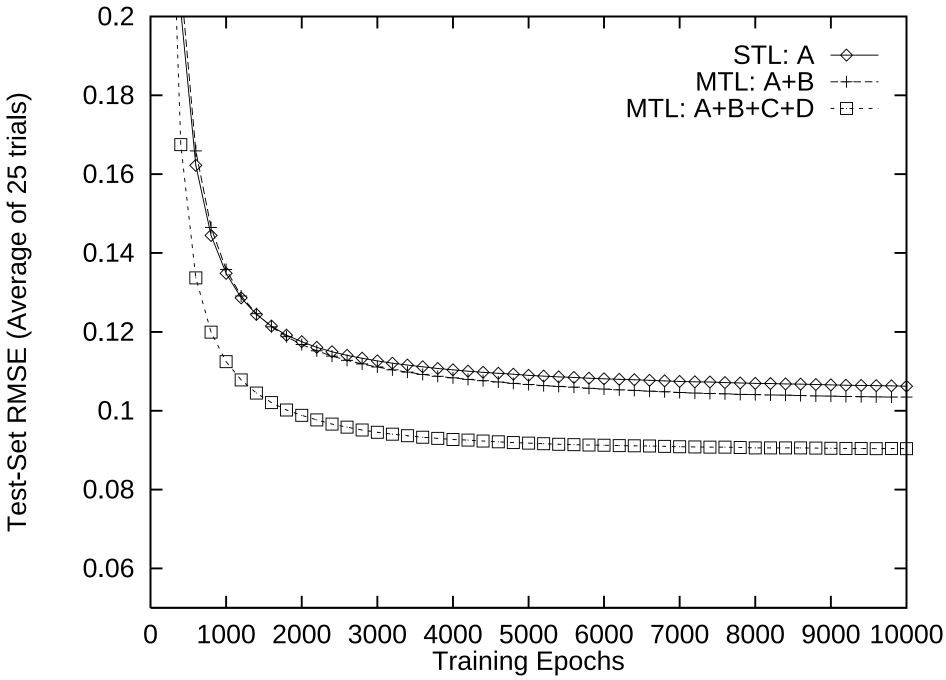


Figure 1.8: RMSE Test-set Performance of Three Different Nets on Task A.

Table 1.2: Test-Set Performance on Task A of STL of Task A, MTL of Tasks A and B, and MTL of Tasks A, B, C, and D. * indicates performance is statistically better than STL at 0.05 or better.

NET	STL: A	MTL: A+B	MTL: A+B+C+D
Root-Mean-Squared-Error	0.106	0.103	0.090 *
Percent Correct	90.2%	90.6%	92.2% *

subfeature was completed by Task B. MTL can work even when there are not unusual relationships between tasks. Perhaps the most interesting result learned from Tasks A–D, however, is that MTL can help even simple problems be learned better when the data set is small. Simple problems become hard when there is little training data.

1.6 Training Signals as an Inductive Bias

Inductive bias is anything that causes an inductive learner to prefer some hypotheses over other hypotheses. Bias-free learning is impossible; in fact, much of the power of an inductive learner follows directly from the power of its inductive bias [Mitchell 1980].

MTL is based on the notion that tasks can serve as mutual sources of inductive bias.

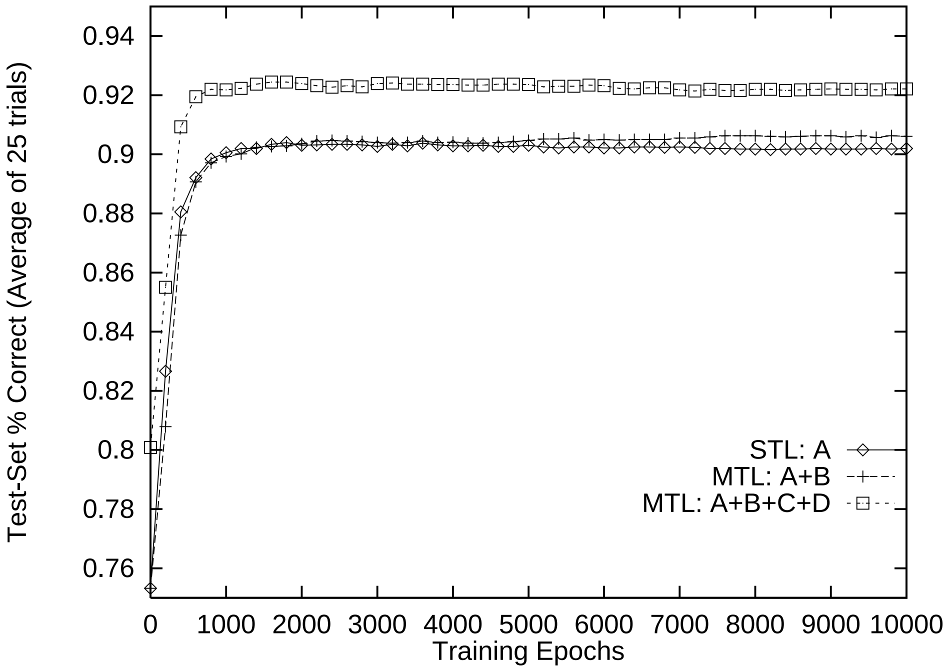


Figure 1.9: Test-set Percent Correct of Three Different Nets on Task A.

MTL is one particular kind of inductive bias. It uses the information contained in the training signal of related tasks to bias the learner towards hypotheses that benefit multiple tasks. One does not usually think of training data as a bias, but when the training data contains the teaching signal for more than one task, it is easy to see that, from the point of view of any one task, the other tasks' training signals may serve as bias. For this multitask bias to exist, the inductive learner must be biased to prefer hypotheses that have utility across multiple tasks.

MTL is one way to achieve inductive transfer between tasks. The goal of inductive transfer is to leverage additional sources of information to improve the performance of learning on the current task. Inductive transfer can be used to improve generalization accuracy, the speed of learning, and the intelligibility of learned models. In this thesis we focus solely on improving accuracy. We are not concerned about the computational cost of learning nor the intelligibility of what is learned. One way transfer improves generalization is by providing a stronger inductive bias than would be available without the extra knowledge. This can yield better generalization with a fixed training set, or it can reduce the number of training patterns needed to achieve some fixed level of performance.

1.7 Thesis Roadmap

This chapter introduced multitask learning in backprop nets and showed it can work on synthetic problems. Chapter 2 demonstrates that MTL works on real problems. We compare the performance of single task learning and multitask learning in backprop nets on three problems. One of these problems is a real-world problem created by researchers other than the author who did not consider using MTL when they collected the data.

Chapter 3 discusses what *related* tasks are and explains *how* MTL works in backprop nets. Section 3.1 reviews the evidence showing tasks must be related for MTL to improve learning. Section 3.2 discusses how tasks should and should not be related for backprop MTL to benefit from them. Section 3.3 presents seven specific kinds of relationships between tasks where MTL-backprop leverages information in the extra training signals to improve generalization. Section 3.4 introduces the Peaks Functions, a set of related tasks designed specifically for research in parallel transfer. Section 3.5 uses the Peaks Functions to show that backprop MTL nets *discover* how tasks are related without being given explicit training signals about task relatedness. Finally, Section 3.6 revisits the notion of relatedness and proposes a definition for it.

Chapter 4 is an important part of this thesis. It shows that there are many opportunities for MTL (and for inductive transfer in general) in real-world problems. This might seem surprising—at first glance most of the problems one sees in machine learning today do not look like multitask problems. We believe most current problems in machine learning appear to be single task because of how we have been trained to do machine learning. Many—in fact, we believe most—real-world problems are multitask problems and performance is being sacrificed when we treat them as single task problems.

Chapter 5 shows that extra tasks can be so useful that sometimes it is better to use an input feature as an extra output task instead. This is surprising, when a feature is used as an output instead of as an input, that feature is ignored when using the learned model for prediction. In this chapter we also show that some features are useful both as inputs and as extra outputs, though the benefits from these two uses are different. We present an approach to multitask learning that allows some features to be used as both inputs and extra outputs at the same time, and thus gain both benefits.

Chapter 6 discusses how to get the best performance from MTL in backprop nets. This thesis is the first to study thoroughly what happens when multiple outputs are trained on a backprop net. We have discovered several important heuristics that help MTL in backprop nets work better. Some of these heuristics are so important that without them MTL nets can perform worse than STL nets instead of better than them.

In Chapter 7 we present an MTL algorithm for k-nearest neighbor and kernel regression. In Chapter 9.2.12 we sketch an algorithm for MTL in decision trees. While these algorithms look rather different from MTL in backprop nets, there is strong overlap of mechanisms and issues; all MTL algorithms must address essentially the same set of problems, even if the specific mechanism in each algorithm is different. By showing how MTL can be used to leverage the same source of extra knowledge in backprop nets, k-nearest neighbor, and decision trees (three very different learning methods that are among the most successful machine learning methods to date) we are able to demonstrate the generality and utility of the MTL approach.

Related work is presented in Chapter 8. Chapter 9 summarizes the contributions of this thesis and discusses directions for future research. Appendix A discusses the effects of excess capacity on generalization in artificial neural nets trained with backprop. Appendix B discusses error metrics based on ranking the training data instead of directly learning a target function for the data.

1.8 Chapter Summary

The standard methodology in machine learning is to learn one thing at a time. Large problems are broken into small, reasonably independent subproblems that are learned separately and then recombined (see, for example, Waibel's work on connectionist glue [Waibel 1989]). This thesis argues that this modularity can be counterproductive because it ignores a potentially rich source of information available in many real-world problems: the information contained in the training signals of other related tasks.

Chapter 2

Does It Work?

Chapter 1 demonstrated multitask learning in backprop nets on two sets of problems carefully devised to introduce the reader to MTL. Before jumping into *how* multitask learning works, what related tasks are, and *when* to use MTL (the subjects of Chapters 3 and 4), we first demonstrate in this chapter that it works on real problems. We do this not only to convince the reader that MTL is useful on real problems, but because the examples will help the reader develop intuitions about how MTL works and where it is applicable.

This chapter presents four applications of MTL in backprop nets. The first uses simulated data for an ALVINN-like road-following domain. The second uses real data collected with a robot-mounted camera. This data was collected specifically to demonstrate MTL. The third and fourth apply MTL to medical decision-making domains. The data in these domains were collected by other researchers who did not consider using MTL when collecting the data. Most of this chapter is spent working with the two medical domains.

2.1 1D-ALVINN

2.1.1 The Problem

1D-ALVINN uses a road image simulator developed by Pomerleau to permit rapid testing of learning methods for road-following domains [Pomerleau 1992]. The original simulator generates synthetic road images based on a number of user defined parameters such as road width, number of lanes, angle and field of view of the camera. We modified the simulator to generate 1-D road images comprised of a single 32-pixel horizontal scan line instead of the original 2-D 30x32-pixel image. We did this to speed learning so more thorough experimentation could be done using the computers available to us in 1992 and 1993—training nets with the full 2-D retina was computationally too expensive with the size nets necessary for good MTL performance to allow many replications and rapid testing of ideas. Figure 2.1 shows several 2-D road images.

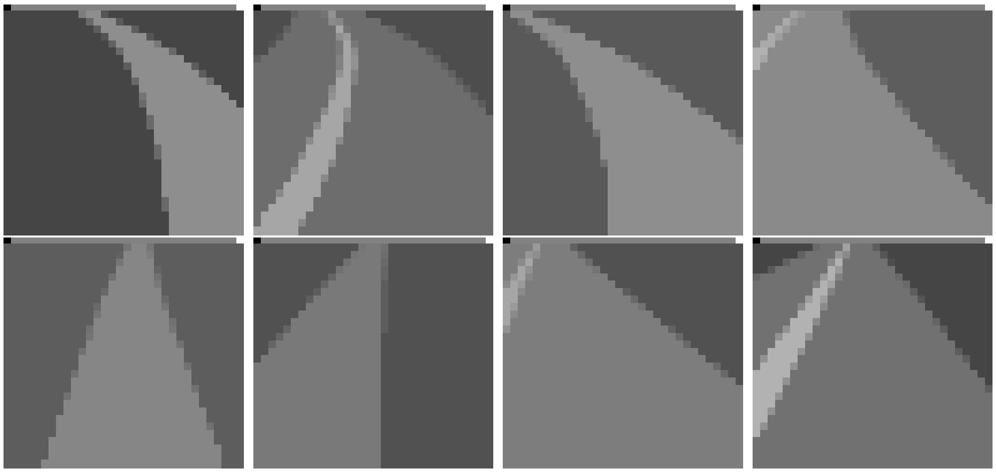


Figure 2.1: Sample single and two lane roads generated with Pomerleau's road simulator. The 1D images are horizontal stripes taken 1/3 of the way up the images from the bottom.

The smaller input size of the 1-D retinas (960 pixels vs. 32 pixels) makes learning easier so smaller training sets can be used. (Our training sets contain 250 images.) Nevertheless, 1D-ALVINN retains much of the complexity of the original 2-D domain. The main complexity lost is that road curvature is no longer visible. The simulated roads still have curvature in the simulator, and this affects the desired steering direction. The loss of curvature information thus limits the accuracy achievable by even perfect learning in the 1D-ALVINN domain. Contrary to what you might expect, road curvature is not the most important part of learning to steer in the ALVINN domain. The backprop nets do not know before training how input pixels are spatially related, so they must learn how the input pixels are arranged. This is difficult. Also, as is evident in Figure 2.1, the generated road images allow the vehicle to be positioned anywhere on the road, far to the left or right of where a good driver would keep a vehicle. This is done to promote robustness by insuring the net learns how to recover from a broad range of situations. Finally, we train on images containing both one and two-lane roads. This greatly increases the variety of the input images. The correct steering direction is very dependent on whether one is driving on a single lane road, or on a two-lane road where the vehicle should be centered in the right lane instead of the road center.

The principal task in both 1D-ALVINN and 2D-ALVINN is to predict steering direction. For the MTL experiments, eight additional tasks were used:

- whether the road is one or two lanes
- location of centerline (2-lane roads only)
- location of left edge of road
- location of right edge of road
- location of road center
- intensity of road surface
- intensity of region bordering road
- intensity of centerline (2-lane roads only)

These additional tasks are all computable from the internal variables in the simulator. We modified the simulator so that the training signals for these extra tasks were added to the synthetic data along with the training signal for the main steering task. (If we were learning from 2D-retinas, we would also use road curvature and its first derivative, both internal parameters in the generator, as additional extra tasks.)

2.1.2 Results

Table 2.1 shows the performance of ten runs of single and multitask learning on 1D-ALVINN using nets with one hidden layer. The MTL net has 32 inputs, 16 hidden units, and 9 outputs. The 36 STL nets have 32 inputs, 2, 4, 8 or 16 hidden units, and 1 output each.[1] Note that the size of the MTL nets was not optimized.

The entries under the STL and MTL headings are the generalization error for nets of the specified size when early stopping is used to halt training. The bold STL entries are the STL runs that yielded best performance. (Most of the differences between STL runs on different size nets are not statistically significant.) The last two columns compare STL and MTL. The first column is the percent reduction in error of MTL over the best STL run. Negative percentages indicate MTL performs better. This test is biased in favor of STL because it compares runs of MTL on an unoptimized net size with several independent

[1] A similar experiment using nets with 2 hidden layers containing 2, 4, 8, 16, or 32 hidden units per layer for STL and 32 hidden units per layer for MTL yielded similar results.

Table 2.1: Performance of STL and MTL with one hidden layer on tasks in the 1D-ALVINN domain. The underlined entries in the STL columns are the STL runs that performed best. Differences statistically significant at .05 or better are marked with an *.

	ROOT-MEAN SQUARED ERROR ON TEST SET						
TASK	**Single Task Backprop (STL)**				**MTL**	Change MTL	Change MTL
	2HU	4HU	8HU	16HU	16HU	to Best STL	to Mean STL
1 or 2 Lanes	.201	.209	.207	<u>.178</u>	<u>.156</u>	-12.4% *	-21.5% *
Left Edge	<u>.069</u>	.071	.073	.073	<u>.062</u>	-10.1% *	-13.3% *
Right Edge	.076	.062	.058	<u>.056</u>	<u>.051</u>	-8.9% *	-19.0% *
Line Center	.153	<u>.152</u>	.152	.152	<u>.151</u>	-0.7%	-0.8%
Road Center	.038	<u>.037</u>	.039	.042	<u>.034</u>	-8.1% *	-12.8% *
Road Greylevel	<u>.054</u>	.055	.055	.054	<u>.038</u>	-29.6% *	-30.3% *
Edge Greylevel	<u>.037</u>	.038	.039	.038	<u>.038</u>	2.7%	0.0%
Line Greylevel	.054	.054	<u>.054</u>	.054	<u>.054</u>	0.0%	0.0%
Steering	.093	<u>.069</u>	.087	.072	<u>.058</u>	-15.9% *	-27.7% *

runs of STL that use different random seeds and are able to find near-optimal net size. The last column is the percent improvement of MTL over the average STL performance. Differences marked with an "*" are statistically significant at 0.05 or better. Note that on the important steering task, MTL outperforms STL 15–30%. It does this without having access to any extra training patterns: exactly the same training patterns are used for both STL and MTL. The only difference is that the MTL training patterns have the training signals for all nine tasks, whereas the STL training patterns have training signals for only one task at a time.

2.2 1D-DOORS

2.2.1 The Problem

1D-ALVINN is not a real domain; the data is generated with a simulator. To test MTL on a more realistic problem, we created an object recognition domain similar in some respects to 1D-ALVINN. In 1D-DOORS, the main tasks are to locate doorknobs and to recognize door types (single or double) in images of doors collected with a robot-mounted color camera. We collected a thousand images as a robot wandered somewhat randomly around the 5th floor of Wean Hall at CMU. From these 1000 images, the 402 images where a doorknob was visible in the image were selected. There are two or more pictures of most doorways. The images were grouped according to the doorway they represented. Two thirds of the groups where used for training, the other 1/3 being used for testing. (We sample from doorways instead of images of doorways because we want test sets to contain doorways the nets are not trained on.) This sampling process yielded training sets containing about 270 images. Figure 2.2 shows several door images from the database.

As with 1D-ALVINN, the problem was simplified by using horizontal stripes from the images, one for the green channel and one for the blue channel. Each stripe is 30 pixels wide (accomplished by applying Gaussian smoothing to the original 150 pixel-wide image) and occurs at the vertical height in the image where the doorknob is located. Ten tasks were used. These are:

- horizontal location of doorknob
- horizontal location of doorway center
- horizontal location of left door jamb
- width of left door jamb
- horizontal location of left edge of door
- single or double door
- width of doorway
- horizontal location of right door jamb
- width of right door jamb
- horizontal location of right edge of door

Figure 2.2: Sample single and double doors from the 1D-DOORS domain.

As this is a real domain, training signals for these tasks had to be acquired manually. We used a mouse to click on the appropriate features in each image in the training and test sets. Since it was necessary to process each image manually to acquire the training signals for the two main tasks, it was not that difficult to acquire training signals for the extra tasks.

2.2.2 Results

The difficulty of 1D-DOORS precluded running as exhaustive a set of experiments as with 1D-ALVINN; comparison could be done only for the two tasks we considered most important: doorknob location and door type. STL was tested on nets using 6, 24, and 96 hidden units. MTL was tested on nets with 120 hidden units. The results of ten trials with STL and MTL are in Table 2.2.

MTL generalizes 20–30% better than STL on these tasks, even when compared to the best of three different runs of STL. Once again, note that the training patterns used for STL and MTL are identical except that the MTL training patterns contain additional training signals. It is the information contained in these extra training signals that helps the hidden layer learn a better internal representation for recognizing door types and the location of doorknobs.

Table 2.2: Performance of STL and MTL on the two main tasks in 1D-DOORS. The underlined entries in the STL columns are the STL runs that performed best. Differences statistically significant at .05 or better are marked with an *.

	ROOT-MEAN SQUARED ERROR ON TEST SET				
TASK	**Single Task Backprop (STL)**			**MTL**	Change MTL
	6HU	24HU	96HU	120HU	to Best STL
Doorknob Loc	.085	.082	<u>.081</u>	<u>.062</u>	-23.5% *
Door Type	.129	<u>.086</u>	.096	<u>.059</u>	-31.4% *

The 1D-ALVINN domain used simulated data. Although the simulator was not built with MTL in mind, it was modified to make extra task signals available in the training data. The 1D-DOORS domain used real data collected from a real camera on a real robot wandering around a real hallway. Although every attempt was made to keep this domain challenging (e.g., the robot was not kept parallel to the hallway and the distance to the doors and illumination was allowed to vary, and some of the training signals were collected using a trackball on a laptop computer while riding a public bus), it is still a domain contrived specifically to demonstrate MTL. How well will MTL work on a real domain which was not customized for it?

2.3 Pneumonia Prediction: Medis

The pneumonia risk prediction problem we now examine is an excellent test bed for MTL research. It is a complex problem for which an unusually large and complete data set is available. This makes it easier to do thorough experiments. Moreover, it is a real domain. The data was collected by researchers who did not know what learning methods might be applied to it. And it turns out that there are many opportunities to apply MTL to this domain and others like it.

2.3.1 The Medis Problem

In this problem the diagnosis of pneumonia has already been made. The goal is not to diagnose if the patient has pneumonia, but to determine how much risk the illness poses to the patient. Of the 3,000,000 cases of pneumonia each year in the U.S., 900,000 are admitted to the hospital. Most pneumonia patients recover given appropriate treatment, and many can be treated effectively without hospitalization. Nonetheless, pneumonia is serious: 100,000 of those hospitalized for pneumonia die from it, and many more are at elevated risk if not hospitalized.

A primary goal in medical decision making is to accurately, swiftly, and economically identify patients at high risk from diseases like pneumonia so they may be hospitalized to receive aggressive testing and treatment; patients at low risk may be more comfortably, safely, and economically treated at home. The goal in this problem is to use information available for patients with pneumonia before they are admitted to the hospital (e.g., patient history and the results of simple tests like blood pressure) to predict each patient's risk of dying from pneumonia. Low-risk patients can be considered for outpatient care. Note that the diagnosis of pneumonia has already been made. The goal is to assess how much risk the pneumonia represents.

Because some of the most useful tests for predicting pneumonia risk are usually measured after one is hospitalized, they will be available only if preliminary assessment indicates hospitalization and further testing is warranted. But low risk patients can often be identified using measurements made prior to admission to the hospital. We have a database in which all patients were hospitalized. It is the *extra* lab tests made after these patients are admitted to the hospital that will be used as extra tasks for MTL; they cannot be used as inputs because they will not be available for most future patients when the decision to hospitalize must be made.

2.3.2 The Medis Dataset

The Medis Pneumonia Database [Fine et al. 1995] contains 14,199 pneumonia cases collected from 78 hospitals in 1989. Each patient in the database was diagnosed with pneumonia and hospitalized. 65 measurements are available for most patients. These include 30 basic measurements acquired prior to hospitalization, such as age, sex, and pulse, and 35 lab results, such as blood counts or blood gases, usually not available until after hospitalization. The database indicates how long each patient was hospitalized and whether the patient lived or died. 1,542 (10.9%) of the patients died. The most useful decision aid for this problem would predict which patients will live or die. But this is too difficult. In practice, the best that can be achieved is to estimate a probability of death (POD) from the observed symptoms. In fact, it is sufficient to learn to *rank* patients by their POD so lower-risk patients can be discriminated from higher risk patients; patients at least risk may then be considered for outpatient care.

2.3.3 The Performance Criterion

The performance criteria used by others working with the Medis database [Cooper et al. 1995] is the accuracy with which one can select prespecified fractions of the patient population who will live. For example, given a population of 10,000 patients, find the 20% of this population at *least* risk. To do this we learn a risk model and a threshold for this model that allows 20% of the population (2000 patients) to fall below it. If 30 of the 2000 patients below this threshold die, the error rate is $30/2000 = 0.015$. We say that the error rate for FOP 0.20 is 0.015 (FOP stands for "fraction of population"). In this paper we consider FOPs 0.1, 0.2, 0.3, 0.4, and 0.5. Our goal is to learn models and model thresholds, such that the error rate at each FOP is minimized.

2.3.4 Using the Future to Predict the Present

The Medis database contains results from 35 lab tests that usually will be available only after patients are hospitalized. These results typically will not be available when the model is used because the patients will not yet have been admitted. We use MTL to benefit from these future lab results. The extra lab values are used as extra backprop *outputs,* as shown in Figure 2.3. The expectation is that the extra outputs will bias the shared hidden layer toward representations that better capture important features of each patient's condition.[2]

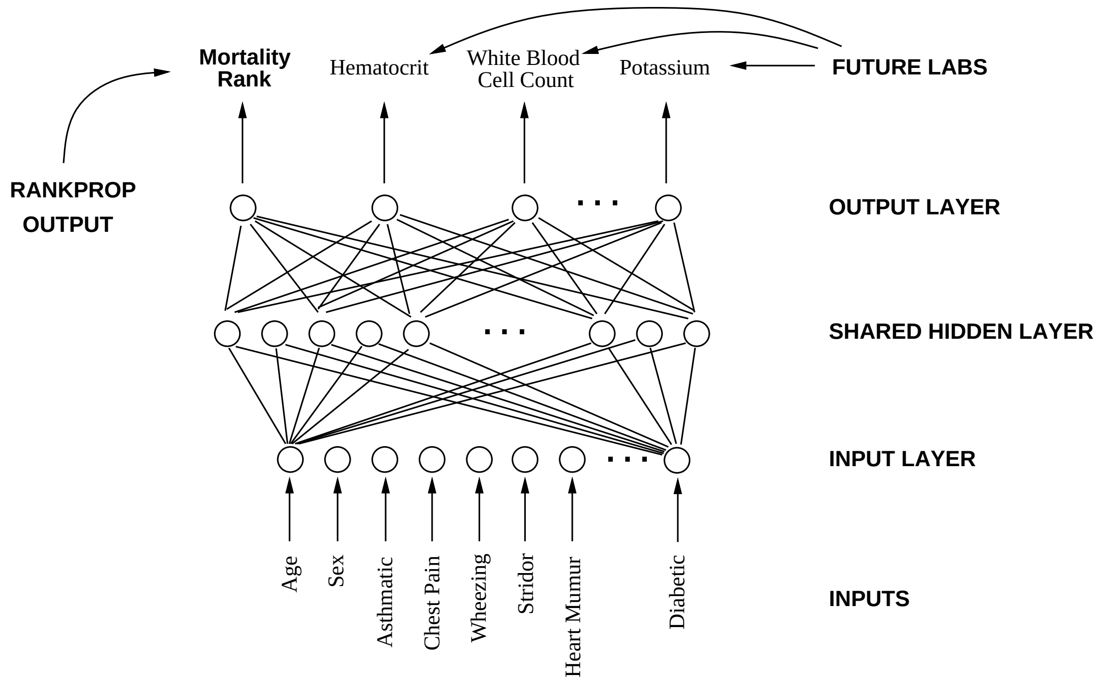


Figure 2.3: Using future lab results as extra outputs to bias learning for the main risk prediction task. (Rankprop is described in Section 2.3.5 and in Appendix B.) The lab tests would help most if they could be used as inputs, but will not yet have been measured when risk must be predicted, so we use them as extra MTL outputs instead.

[2]It is interesting to note that other researchers who tackled this problem using this database ignored the the extra lab tests because they knew the lab tests would not be available at run time and did not see ways to use them other than as inputs.

2.3.5 Methodology

The straightforward approach to this problem is to use backprop to train an STL net to learn to predict which patients live or die, and then use the real-valued predictions of this net to sort patients by risk. This STL net has 30 inputs for the basic measurements, a single hidden layer, and a single output trained with targets 0=lived, 1=died.[3] Given an infinite training set, a net trained this way should learn to predict the probability of death for each patient, not which patients live or die. In the real world, however, we rarely have an infinite number of training cases. If the training sample is small, the net will overfit and begin to learn a very nonlinear function that outputs values near 0/1 for cases in the training set, but which does not generalize well. It is critical to use early stopping to halt training before this happens.

We developed a method called *Rankprop* specifically for this domain that learns to rank patients without learning to predict mortality. "Rankprop" is short for "backpropagation using sum-of-squares errors (SSE) on repeatedly re-estimated ranks". Figure 2.4 compares the performance of SSE on 0/1 targets with rankprop on this problem. Rankprop outperforms traditional backprop using sum-of-squares errors on targets 0=lived,1=died by 10%-40% on this domain, depending on which FOP is used for comparison. See Appendix B for details about rankprop and a comparison of the performance of rankprop and traditional backprop on this domain.[4]

The STL net has 8 hidden units and one output for the rankprop risk prediction. The MTL net has 64 hidden units. (Preliminary experiments suggested 8–32 hidden units was optimal for STL, and that MTL would perform somewhat better with nets as large as 512 hidden units. We use 8 hidden units with STL and 64 hidden units with MTL so that we can afford to run many experiments.) The MTL net is shown in Figure 2.3. It has the same inputs as the STL net, and also has the same rankprop output that learns to order patients

[3] We tried both squared error and cross entropy with these outputs (0=lived,1=died). The differences between the two approaches was small, with squared error performing slightly better. The results we report in this thesis are for squared error.

[4] We use rankprop for our experiments with MTL because it is the best performer we know of on this problem. (It outperforms SSE for both STL and MTL.) We are not interested in developing methods that improve inferior algorithms. We want MTL to make the best algorithms better.

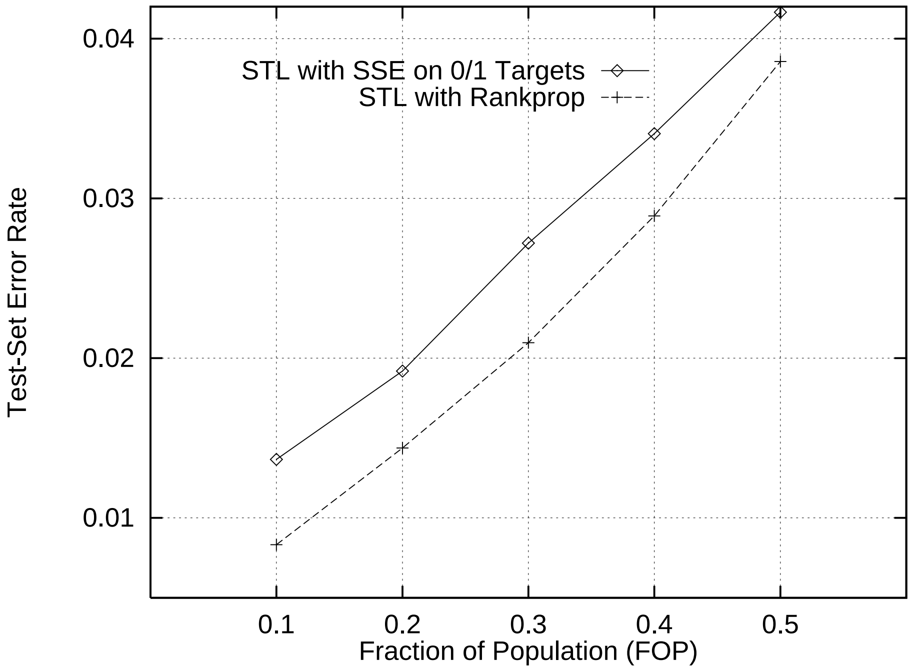


Figure 2.4: The performance of SSE with 0/1 targets and rankprop on the 5 FOPs in the pneumonia risk prediction domain. Lower error indicates better performance.

by risk. This is the main task. In addition to the main task, the MTL net also has 35 extra outputs. These are the extra tasks the net will learn while also learning to predict risk. In this domain the extra tasks are to predict the results of lab tests that usually will not be ordered unless a preliminary risk assessment suggests the patient should be hospitalized.

We train the net using training sets containing 1000 patients randomly drawn from the database. Training is halted using a halt set containing another 1000 patients drawn from the database. Training is halted on both the STL and MTL nets when overfitting is observed on the main rankprop risk task. Overfitting is detected by observing the performance of the backprop nets during training on an independent test set (often called the halt set) not used for backpropagation. When performance on the halt set stops improving or begins getting worse, training is stopped and the model weights are frozen. On the MTL net, the performance of the extra tasks is not taken into account for early stopping. Only the performance of output(s) for the main task are considered when deciding where to halt training. (See Section 6.1 for more discussion of early stopping with MTL nets. Figure 6.1 in that section shows an interesting assortment of halt-set curves used for early stopping.)

Once training is halted, the net is tested on the remaining unused patients in the database. This process of randomly sampling training and halt sets and testing on the remaining cases is repeated 10 times.

The Medis database contains 14,199 patients. We use training sets containing only 1000 cases for several reasons. The main reason is that at the time we began working with this data we were preparing to work with a different, more complex database that contains only about 2,400 cases. We viewed the Medis database as a warm-up exercise for this more interesting database. Unfortunately, access to the other database was delayed two years by complications in coding and verifying the data. That database is so complex that even routine manipulations can require days of manual labor if done correctly. We gained access to a subset of this other database only recently. The results of our first experiments with this other database are presented in Section 2.4 at the end of this chapter.

Another reason we use small training sets with the Medis database is illustrated in Figure 2.5. This figure shows the performance of k-nearest neighbor on the Medis pneumonia problem as a function of the number of training cases. Performance asymptotes as the number of training cases increases above 5,000 cases. Preliminary experiments suggest backprop nets also benefit little from training sets containing more than about 5,000 cases. By definition, *consistent* learning procedures converge to the true function given enough data. Interesting differences between consistent procedures such as KNN and backprop (given large enough nets) show up at small-to-moderate sample sizes. Many of MTL's benefits derive from mechanisms where the extra tasks compensate for some difficulties of training with a limited sample (see Section 3.3). We expect little benefit from MTL on this problem with training sets containing more than about 5,000 cases. This was borne out by preliminary experiments which suggested MTL helped performance most with training sets containing 250–2,500 training cases. MTL seems to improve performance with training sets as large as 7,000 cases (the largest we tested), but the improvement is small and it is difficult to achieve statistical significance when comparing the methods. (We believe it is not worthwhile to try to show statistical significance where differences are too small to be interesting.)

A third reason we use small training sets is that the experiments run much faster.

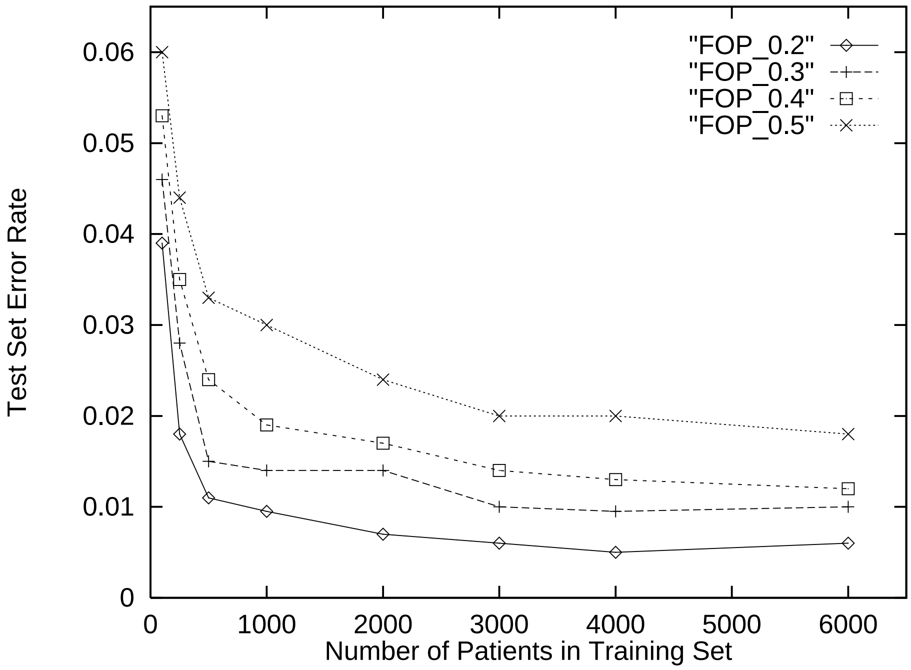


Figure 2.5: Generalization performance of k-nearest neighbor as a function of training set size for pneumonia FOP's 0.2, 0.3, 0.4 and 0.5.

The cost of each epoch increases linearly (or worse if one considers cache performance) with the number of training cases, and the number of epochs required for overfitting to begin also increases with the number of training patterns. To make things worse, rankprop trains slower than backprop with 0=lives/1=dies targets. But Rankprop is the best STL performer on this problems, so we want to use it. All these factors greatly favored using smaller training sets.

In the end, we settled on train and halt sets containing 1000 cases each because this number was not only practical, but also seemed to be the largest sizes that would be usable with the other pneumonia database we hoped to use. It is important to realize that it is possible to make more accurate predictions than we report here for the Medis pneumonia problem by training on more of the available data. We do not feel this diminishes the importance of the results we report here because few medical databases contain as many cases as the Medis database.

2.3.6 Results

Table 2.3 shows the mean performance of ten runs of rankprop using STL and MTL. The bottom row shows the percent improvement over STL. Negative percentages indicate MTL reduces error. Although MTL lowers the error at each FOP compared with STL, only the differences at FOP 0.3, 0.4, and 0.5 are statistically significant with ten trials using a standard t-test.

Table 2.3: Error Rates (fraction deaths) for STL with Rankprop and MTL with Rankprop on Fractions of the Population predicted to be at low risk (FOP) between 0.0 and 0.5. MTL makes 5–10% fewer errors than STL.

FOP	0.1	0.2	0.3	0.4	0.5
STL Rankprop	.0083	.0144	.0210	.0289	.0386
MTL Rankprop	.0074	.0127	.0197	.0269	.0364
% Change	-10.8%	-11.8%	-6.2% *	-6.9% *	-5.7% *

The improvement due to MTL is 5–10%. This improvement can be of considerable consequence in a medical domain. To verify that the benefits seen here are due to relationships between what is learned for the future labs and the main task, we ran the shuffle test (see Section 1.4) on the pneumonia problem. We shuffled the training signals for the extra tasks in the training sets before training the nets with MTL.

Figure 2.6 shows the results of MTL with shuffled training signals for the extra tasks. For comparison, the results of STL, and of MTL with unshuffled extra tasks, are also shown. Shuffling the training signals for the extra tasks reduces the performance of MTL below that of STL. We conclude that it probably is the relationship between the main task and the extra tasks that lets MTL perform better on the main task; the benefit disappears when these relationships are broken by shuffling the extra task signals.

2.3.7 How Well Does MTL Perform on the Extra Tasks?

One might be interested in how well the MTL net performed on the extra tasks compared with STL nets learning those same tasks. Because most extra tasks are not well learned given these inputs, we don't expect to see large differences in performance.

We trained STL nets on each extra task. The inputs to these nets are the same inputs

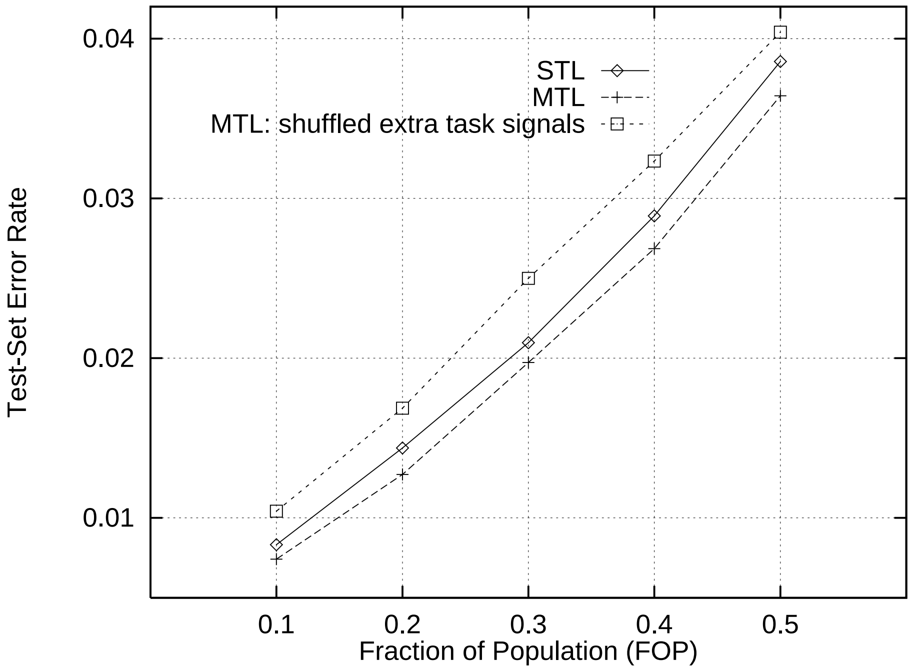


Figure 2.6: Performance of STL, MTL, and MTL with shuffled extra task signals on pneumonia risk prediction at the five FOPs.

used above, the regular pre-admission patient measurements. We repeated this 10 times, using the same samples used for the 10 trials above. Thus we trained $36 \times 10 = 360$ individual nets. Early stopping was used to stop training on each net individually. Computationally, training this many STL nets is far more expensive than training the 10 MTL nets the STL nets will be compared against.[5]

We compared the performance of the individual STL nets on the 35 tasks to the performance of the MTL net trained on the main task with the 36 extra tasks. Figure 2.7

[5]We use a trick when training STL nets like this that makes running the experiment simpler and that saves some computation. Instead of actually training a separate net for each output, we train one net with all 36 outputs. This hidden layer of this net, however, is broken into smaller pieces that connect only to one output. Thus the first 8 hidden units connect to output 1, the next 8 to output 2, etc. The hidden layers are fully connected to the inputs. This is equivalent to training the outputs separately, as long as one does early stopping on the outputs individually. The advantage of this procedure is that whatever machinery has been developed to work on the MTL net will also work on the STL net because the nets have the same inputs and outputs. The only difference is that the hidden layers are not allowed to share what is learned by the different tasks.

shows the percent change in RMS error of going from STL to MTL for the 36 extra tasks. MTL yields lower error than STL for 28 of the 36 outputs. A sign test shows that this is significant at the .001 level. Moreover, in the few cases where STL performs better than MTL, the differences are all small. MTL, however, sometimes outperforms STL by several percent. (The average improvement due to STL is 0.22%; the average improvement due to MTL is 0.69%.)

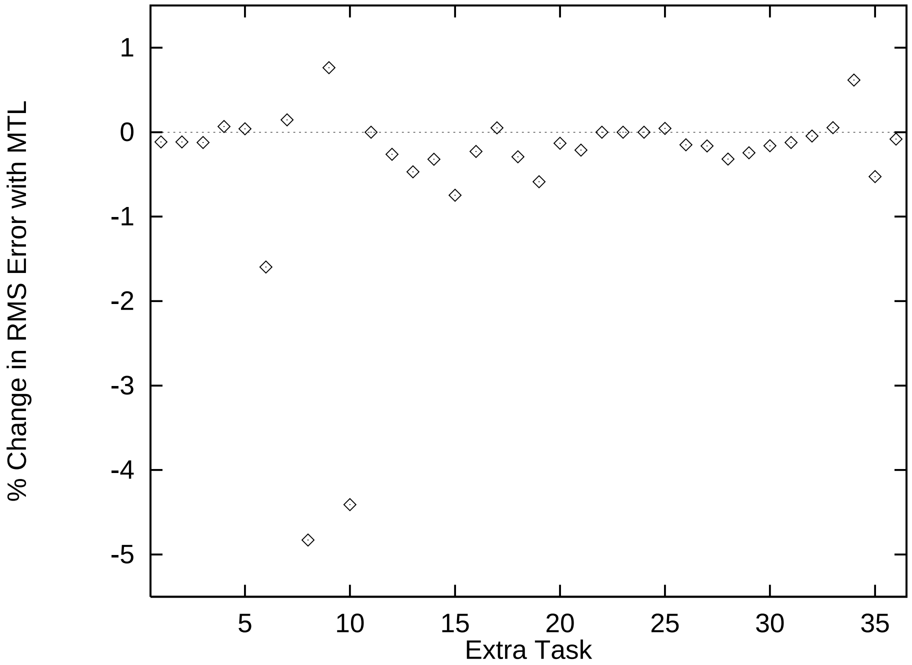


Figure 2.7: Percent improvement in RMS Error on the extra tasks when MTL is compared with STL. Negative improvements indicate MTL performs better.

Caution: one might be tempted to view the results on the extra tasks as 35 different experiments comparing STL and MTL. For example, one might interpret the results as suggesting that MTL will outperform STL roughly $28/36 = 78\%$ of the time. Because these tasks are drawn from the same domain, the tests are not independent, so this is a risky inference.

2.3.8 What Extra Tasks Help the Main Task?

A natural question to ask when using MTL is what extra tasks *affected* the main task most. A related, though different, question is what extra tasks *helped* the main task most.

Unfortunately, answering these questions can be expensive. The most thorough approach to addressing these questions is to train all possible combinations of extra tasks with the main task and see which sets of extra tasks change or improve performance on the main task most. This combinatorial approach is, of course, usually impractical. A more practical approach is to ask which extra tasks were themselves learnable. Presumably an extra task that is not learnable from the inputs can contribute little to the main task because nothing is learned for it. It is possible, however, that a task learned just a little bit better than random might still have substantial effect on what is learned in the hidden layer. Poor performance on a task does not necessarily imply that nothing of value is learned for that task. The task may be intrinsically difficult to predict even given the optimal learned representation. Conversely, a task that is easily learned to high accuracy may not lead to the development of interesting internal representations that are useful to other tasks. As an extreme example of this, consider an output that duplicates one of the inputs (i.e., the training signal for the output has the same values as one of the input features). Such an output can be learned almost perfectly by a backprop net because the net need only learn to feed the value of the input directly through the hidden layer to the output it duplicates. In doing this, nothing new is learned in the hidden layer that was not already available as an input.

We examined the training curves for all the outputs of the MTL net. Many of the extra outputs are not learnable from the inputs. This should not come as a surprise. First, we assume that a lab test is worth measuring only if the outcome of the test could not be well predicted before doing the test. Second, most of the lab tests reflect more specific, more accurate measurements of the patient than the measurements available before that patient is admitted to the hospital. It is going to be difficult, if not impossible, to predict most detailed internal measurements such as blood chemistry from simpler, more general measurements such as age, sex, and blood pressure.

Figure 2.8 shows the training curves for the eight extra outputs that were *most* learnable. We judged how learnable an output was by examining all the test-set learning curves. If the learning curve showed significant or interesting improvements in accuracy during training, we assumed this was because something interesting was learnable for that output. For comparison, the SSE output for VITSAT, the main task, is also included.

The rapid drop in RMSE that occurs in the first few passes of backprop is not that meaningful. The net is just learning the mean of the distribution for that output. It is the changes in RMSE that occur after this drop that are consequential. Most changes are small. None of these tasks are well learned. This does not, however, mean that what is learned is not meaningful or useful. For example, the output for VITSAT, the non-rankprop output for the main task, shows a similarly small drop in error. Yet the model that is learned to create this small drop in error allows patients to be ranked by risk with good accuracy. Small changes in RMSE can reflect significant changes in what the model has learned.

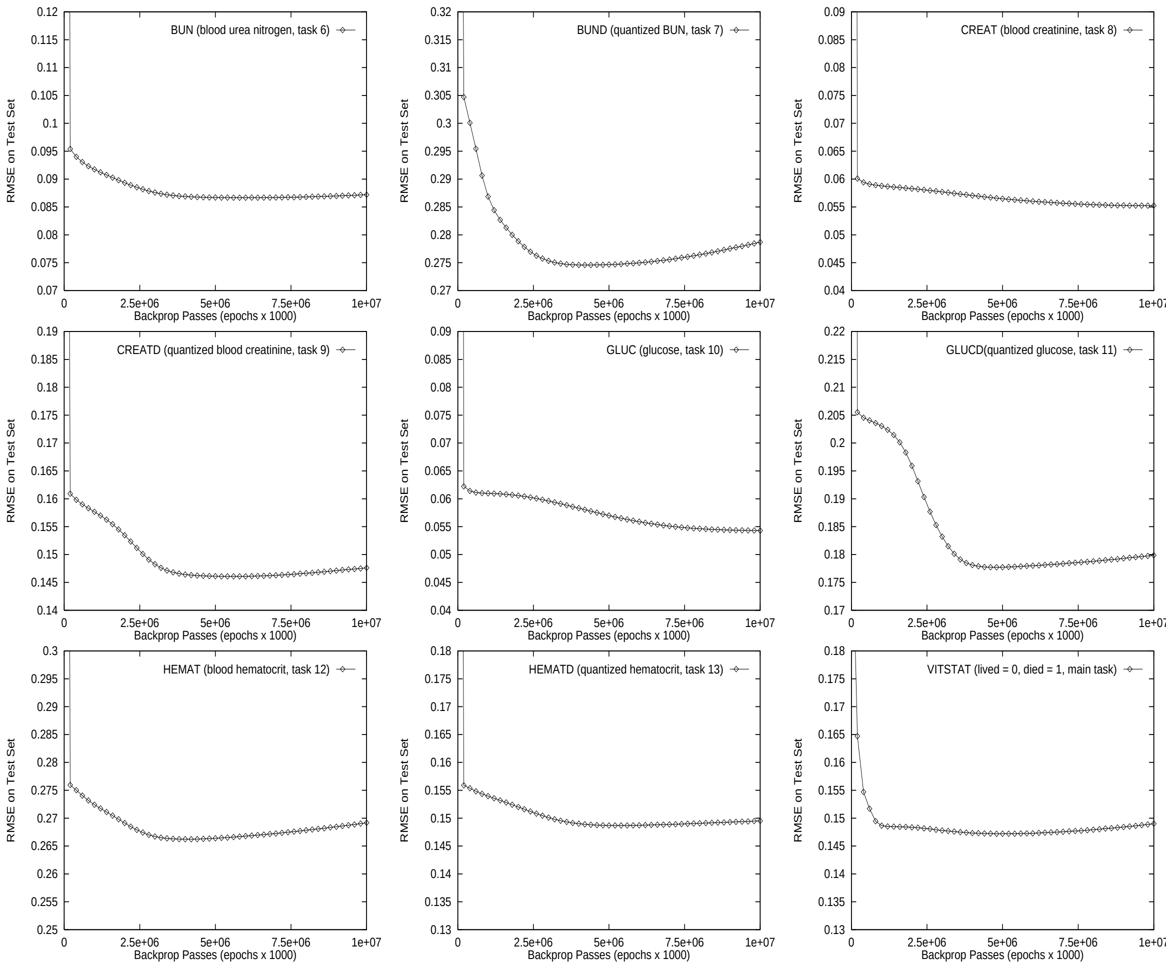


Figure 2.8: Learning curves for the eight most learnable extra tasks. The learning curve for the SSE 0/1 output for the main task (VITSTAT) is shown for comparison.

Interestingly, the tasks that we judged to be most learnable are tasks which also show some of the largest differences in performance between STL and MTL in Figure 2.7. (Learnable tasks were selected before we had seen the results in Figure 2.7, so we were not biased by those results.)

2.3.9 Comparison with Feature Nets

The future lab tests used as extra tasks by MTL are *the* lab tests doctors order to help assess pneumonia risk and decide on patient treatment. They are excellent features for pneumonia risk prediction. Figure 2.9 shows the performance that could be obtained if we were able to use the future lab tests as extra inputs.

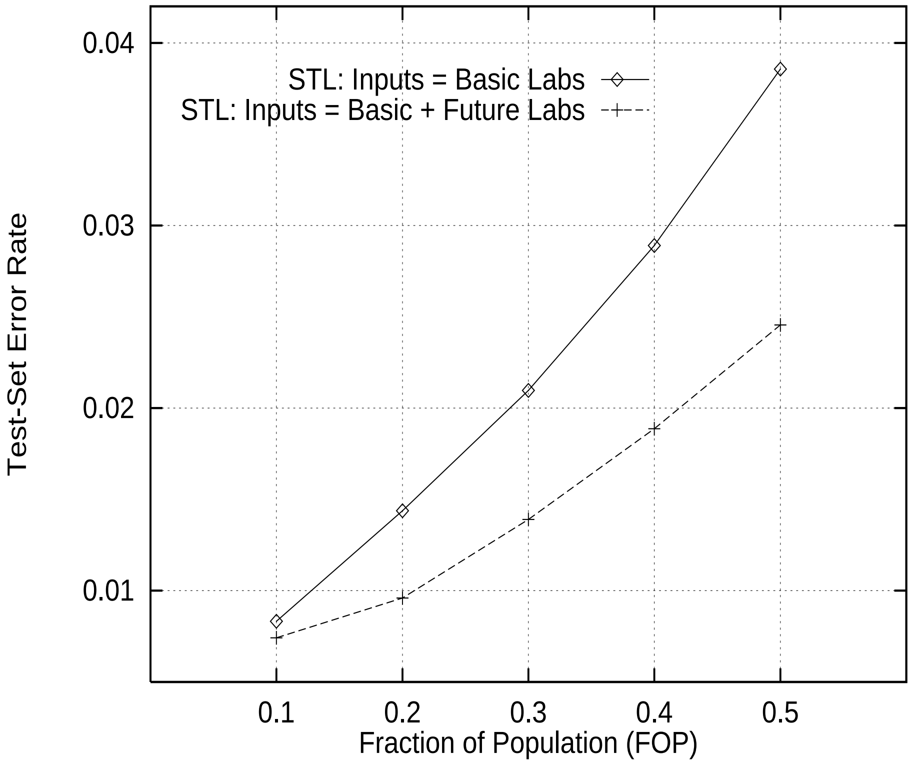


Figure 2.9: The Basic Labs that are available for use as inputs include some items from the patient histories and simple measurements such as weight and blood pressure that can conveniently be measured prior to hospitalization. Using the future lab tests as extra *inputs* would improve performance considerably.

Unfortunately, we cannot use them as regular inputs to a net learning to predict risk because they usually will not be available when the model would be used. They will be missing. But they are available in the training set. An alternate approach to using the training signals for the future labs as extra outputs is to learn models to predict the lab tests that will be missing, and provide these predictions as extra inputs to a net learning the main risk prediction task. Feature nets [Davis & Stentz 1995] is a competing approach to MTL that does this. It trains nets to predict the missing future measurements and uses the predictions, or the hidden layers learned for these predictions, as extra *inputs*. See Figure 2.10. Using predictions imputed for missing values as inputs when learning to predict another is a standard technique in statistics. Using the hidden layers that are trained to imput these values as extra inputs, however, is a new and interesting approach that often outperforms simple value imputation.

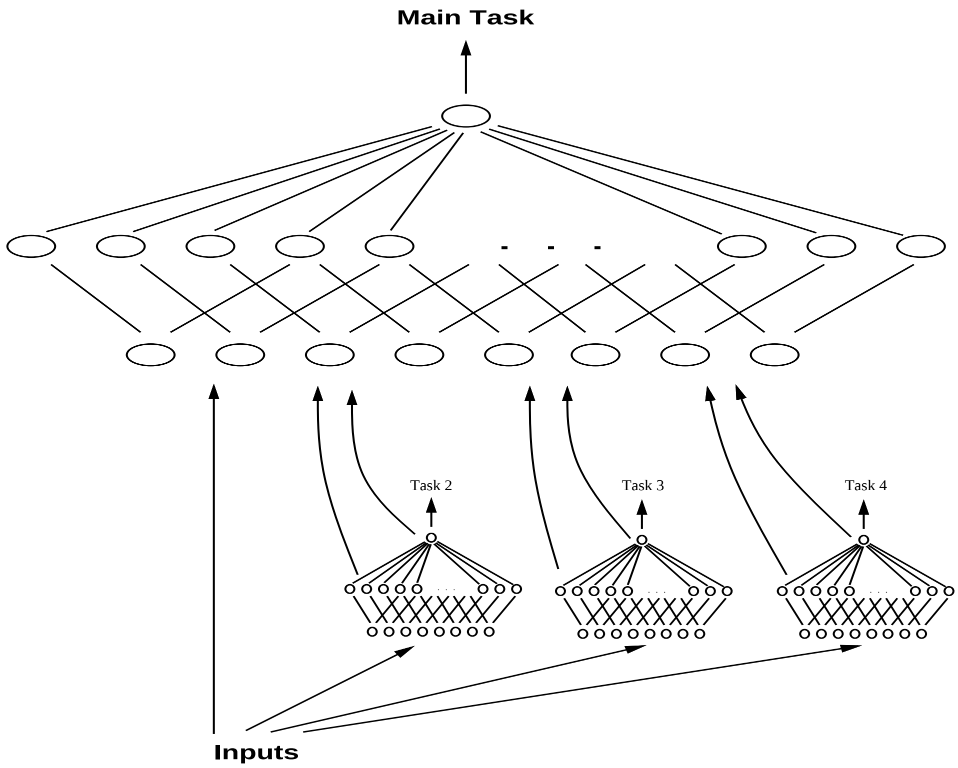


Figure 2.10: Feature Nets allow the main task to be trained with STL but still benefit from what can be learned for auxiliary tasks.

We tried feature nets on the pneumonia problem. We trained each of the extra tasks on STL nets with 8 hidden units. There are 35 extra tasks, so we trained 35 STL nets for each of the ten trials. Early stopping is used to halt training on these nets when performance on the tasks is maximized as measured with a halt set. We used the same training and halt sets to train all the nets. These are the same samples of data that were used to train the

MTL net.

After training the 35 individual STL nets for each trial, we did two experiments. In one experiment, the predictions made by the nets for the 35 future labs were provided as extra input features for an STL net learning the main risk prediction task. This net has 30 regular inputs, plus 35 additional inputs for the predictions made by the STL feature nets. In the second experiment we did not use the predictions of the STL nets, but used the hidden layer activations of those nets as additional inputs to a net learning the main task. This net has 30 regular inputs, and 8*35=280 extra inputs for the hidden layers of the 35 STL nets.

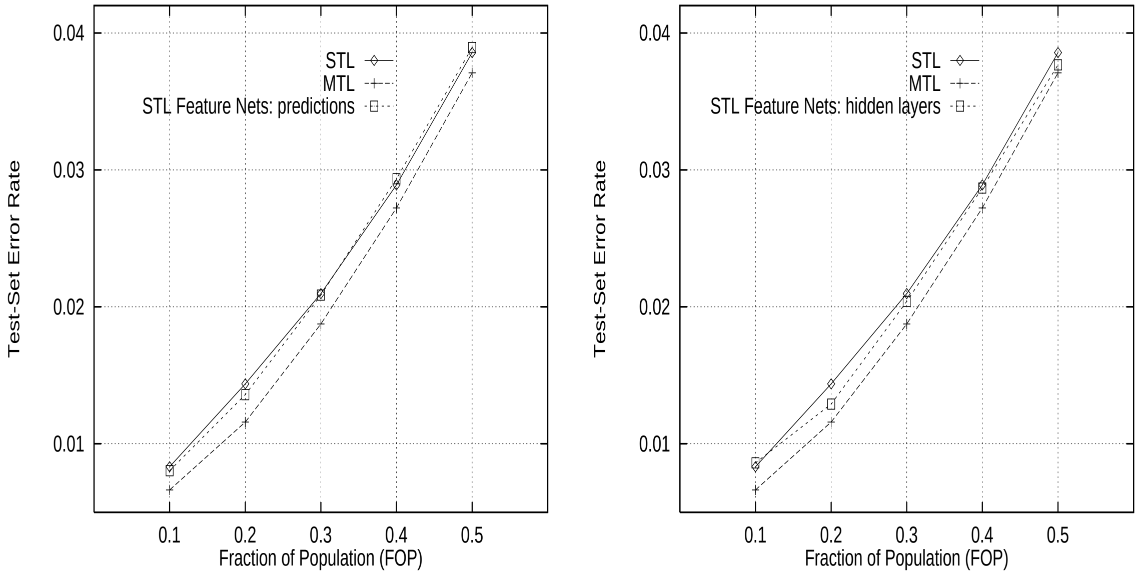


Figure 2.11: Performance of both approaches to feature nets compared with STL and MTL.

Figure 2.11 compares the performance of using feature nets for future lab tests with that of STL and MTL. The left graph is the performance when the predictions from the feature nets are used as extra inputs to the net learning the main risk prediction task. The graph on the right is the performance when the hidden layers learned in the 35 feature nets are used as extra inputs to the net learning the main task.

Neither approach to feature nets yields improvements comparable to MTL on this domain. This does not mean MTL will always outperform feature nets. If the extra tasks are highly predictable, using predictions for them as extra inputs should yield performance comparable to what could be achieved if the real values for these inputs could be measured and used as regular inputs. However, in this domain, the future lab tests cannot be predicted accurately, so the models learned for them provide little useful extra information when used as inputs. Yet as extra *outputs*, they do help. This might seem surprising. A more complete analysis of why this can happen will be made in Chapter 5. To summarize, when *predictions* for tasks are used as extra *inputs*, any noise in those predictions is injected into the input of the network. This not only makes learning more difficult, but also makes the learned model sensitive to the noise in those input predictions later when the net is tested. If the output of a net is a function of noisy inputs, the output must be noisy, too. Using noisy predictions as extra inputs may reduce bias, but is also likely to increase variance. If variance increases more than bias is reduced, prediction accuracy is hurt.

When used as extra outputs, however, this noise is not a problem. The training signals the net sees when given extra tasks as outputs are not noisy because they are the real training signals collected for those training cases. The training signals have not been corrupted by a marginal learning process.

2.4 Pneumonia Prediction: PORT

In Section 2.3 we used the Medis pneumonia risk problem. In this section we apply MTL to another pneumonia risk problem. The pneumonia problem we now tackle uses a database called the PORT Database. We'll refer to this problem as the "PORT" problem to avoid confusion with the Medis pneumonia problem in the previous section.

2.4.1 The PORT Problem

Like the Medis pneumonia problem used in Section 2.3, the PORT problem is a real domain for which data was collected by researchers who did not know about MTL methods. As before, the diagnosis of pneumonia has already been made. The goal is not to diagnose if the patient has pneumonia, but to determine how much risk the the pneumonia poses to the patient.

Unlike the previous pneumonia domain, not all patients in the Port domain have been admitted to the hospital. Roughly 1/3 of the patients have been treated as outpatients. The principal task in the PORT domain is not to predict which patients are at low enough risk to be considered for outpatient treatment as in the previous section, but to predict which patients are at high risk from their pneumonia. In the PORT domain, patients at high risk are defined as those that develop one (or more) of the following dire outcomes:

- will need treatment in a critical care unit (ICU)
- develop any of five different severe complications
- die

The goal in this domain is to accurately predict those patients at highest risk of developing one or more of these dire outcomes.

2.4.2 The PORT Dataset

Unlike the Medis Database used in Section 2.3 that contained 14,199 patients, the PORT Pneumonia Database contains only 2,287 pneumonia cases. The total number of cases available in the PORT database is similar to the number of cases we used for training with the Medis pneumonia problem. (This is not an accident. We knew about the PORT database was when we began working with the larger Medis database, and viewed the work with the Medis database as a warm-up exercise for working with the richer, more complex, and smaller PORT database.)

Each patient in the PORT database was diagnosed with pneumonia. More than a thousand variables are available in the database for most patients. The effort required to preprocess all the variables, however, is daunting, so only the 203 measurements in the core database have been coded thus far and are available for our use. These 203 measurements are available for most patients. These include all basic measurements acquired prior to hospitalization (e.g., age, sex, and pulse) and all lab results (e.g., blood counts and blood gases) available after hospitalization. Because some patients were not hospitalized, some variables have missing values. We used a k-nearest neighbor method we devised specifically for this dataset to impute (fill-in) missing values.[6]

[6]We do not know how important imputing missing values is to our results on this problem because we

The PORT database also indicates how long each patient was hospitalized, the total cost of hospitalization, whether the patient was admitted to intensive care, whether the patient developed any of a number of complications, and whether the patient was still alive 30, 60, and 90 days after they were admitted.

2.4.3 The Main Task

A useful decision aid for PORT is to predict which patients are at high risk. In this problem high risk patients are defined as:

$$DireOutcome = ICU \vee Comp_1 \vee Comp_2 \vee \cdots \vee Comp_5 \vee Death \tag{2.1}$$

where ICU is a boolean indicating whether or not the patient entered the intensive care unit, $Comp_1 \cdots Comp_5$ are booleans for five different severe complications, and $Death$ is a boolean indicating whether or not the patient survived at least 90 days.

The main task in the PORT domain is to predict *DireOutcome*. As with the Medis pneumonia problem, predicting which patients will experience a dire outcome is too difficult. In practice, the best that can be achieved is to estimate a probability of a dire outcome from the observed variables so that patients with higher probability of *DireOutcome* can be distinguished from those with lower probability.

The performance criteria used by others working with the PORT database to predict *DireOutcome* are the accuracy, positive predictive value, negative predictive value, sensitivity, specificity, and ROC curve area. These are defined as follows:

```
                         MODEL PREDICTION

                    |     1       0       |
              - - - + - - - - - - - - - - + - - - - -
   TRUE         1   |     A       B       |    A+B
                    |                     |
   OUTCOME      0   |     C       D       |    C+D
              - - - + - - - - - - - - - - + - - - - -
                    |    A+C     B+D      |  A+B+C+D
```

have not tried running experiments without filled-in missing values. We do not discuss the method for filling in missing values because we are patenting it.

```
Accuracy    = (A+D) / (A+B+C+D)
Positive PV = A / (A+C)
Negative PV = D / (B+D)
Sensitivity = A / (A+B)
Specificity = D / (C+D)
```

Accuracy is the usual measure of the fraction of cases properly predicted. In this domain we do not expect to achieve accuracies significantly better than the default accuracy achieved by predicting all patients to be in the most frequent class, $DireOutcome = 0$. Positive Predictive Value is the accuracy with which cases with true outcome 1 are predicted. It is a measure of how often the model misses cases that have true outcome 1. Negative Predictive Value is the accuracy with which cases with true outcome 0 are predicted. It is a measure of how often the model misses cases that have true outcome 0. Sensitivity is the fraction of cases predicted to be outcome 1 that actually are outcome 1. It is a measure of how reliable the prediction is when it predicts an outcome of 1. Specificity is the fraction of cases predicted to be outcome 0 that actually are outcome 0. It is a measure of how reliable the prediction is when it predicts an outcome of 0.

An ROC curve is a graph of Sensitivity vs. 1-Specificity as the threshold for the boundary between high and low risk is sweeped from one limit to the other. (See Figure 2.12 for examples of ROC curves.) If there is no relationship between the model prediction and the true outcome, the ROC curve is a (possibly noisy) diagonal line and the area under the curve is about 0.5. If the model's predictions strongly predict the true outcome, the ROC curve rises quickly and has area near 1.0. If the model prediction strongly predicts anti-truth, the ROC area is less than 0.5. In summary, good models have ROC areas greater than 0.5, with better models having ROC areas closer to 1.0.

2.4.4 Extra Tasks In Port

The PORT database indicates how long each patient was hospitalized, the total cost of the hospitalization, whether the patient was in intensive care, whether the patient developed any severe complications, the predictions of a logistic regression model developed by medical experts to predict pneumonia risk using this database, and whether the patient was still alive 30, 60, and 90 days after they were admitted. We use these variables as extra outputs for MTL while training the main task output to predict *DireOutcome*. As before, the expectation is that the extra outputs will bias the shared hidden layer toward representations that better capture important features of each patient's condition, and this will improve performance on the main *DireOutcome* prediction task. Some of the extra tasks in PORT are disjunctive subcomponents of the main task (see Equation 2.1), which was not the case when we applied MTL to the Medis problem in Section 2.3.

2.4.5 Methodology

The straightforward approach to this problem is to use backprop to train an STL net to learn to predict the boolean *DireOutcome*, and then compute the error measures such as the ROC area using the real-valued predictions of the trained net. This STL net has 203 inputs, a single hidden layer, and a single output trained with boolean 0/1 targets. Because our training sets are small, the net will eventually overfit and begin to learn a very nonlinear function that outputs values near 0/1 for cases in the training set, but which does not generalize well. It is critical to use early stopping to halt training before this happens.[7]

The STL net has 64 hidden units and one output for *DireOutcome* prediction. The MTL net also has 64 hidden units. The MTL net has the same 203 inputs as the STL net, and also has the same *DireOutcome* output. This is the main task. In addition to the main task, the MTL net also has 34 extra outputs.

We split the 2,287 cases in the PORT database into a final test set containing 686 patients, and a training set containing the remaining 1601 cases. This was done once: there is only one final test set and training is never done using any of the cases in this final test set. We randomly split the 1601 training cases into training sets containing 1200 cases, and early stopping test-sets containing 401 cases. We did this ten different times so we could run ten trials. For each trial, STL and MTL nets are trained with backpropagation on

[7]Because the PORT database became available only recently, we have not yet applied Rankprop to it. The experiments we report here train the nets using standard squared error on 0/1 targets. We do not know if Rankprop would yield better performance than SSE on 0/1 targets, though it probably would. This difference makes the results we present here more independent from the results presented in Section 2.3.

the training sets. Training is halted when performance on the halt-set is maximized. We use ROC area as the performance criterion for early stopping, not SSE error. As always, performance is only measured using the main task output; the extra tasks on the MTL nets are ignored when deciding where to halt training.

2.4.6 Results

Table 2.4 shows the average performance of the 10 STL and MTL nets on the held-out test set. MTL outperforms STL on every measure.[8] The average improvement of MTL over STL is 8.75%. On the important ROC area measure, MTL improves the ROC area 10.5%. Note that we do not know what the best ROC area that could be achieved in this domain is. Because the domain is stochastic, however, it is unlikely that ROC areas near 1.0 are achievable. Thus MTL probably improves the ROC area more than 10.5%.

Table 2.4: Performance of STL and MTL on the PORT *DireOutcome* prediction problem using a number of different performance measures. The Difference column is the percent reduction in error of MTL over STL.

METRIC	STL	MTL	Difference
Accuracy	0.8717	0.8819	-7.95 %
Positive PV	0.4348	0.4861	-9.08 %
Negative PV	0.9206	0.9283	-9.70 %
Sensitivity	0.3797	0.4430	-10.20 %
Specificity	0.9357	0.9390	-5.13 %
ROC Area	0.8617	0.8748	-10.46 %
Average			-8.75 %

Figure 2.12 shows the average ROC curves for ten trials of STL and MTL. The ROC curve for MTL dominates the ROC curve for STL over most of the graph. This difference is also reflected in the ROC area for STL and MTL in Table 2.4; the ROC area for MTL is 10.5% closer to 1.0 than the STL ROC area.

We performed our experiments on the PORT Database as members of a larger project studying this problem. Members of this project applied other learning methods to the PORT problem using the same test and training sets we used for STL and MTL. These

[8] Because the same held-out test set is used in each of the ten trials, it is inappropriate to use t-tests to estimate the significance of the improvements of MTL over STL as the test sets are not independent.

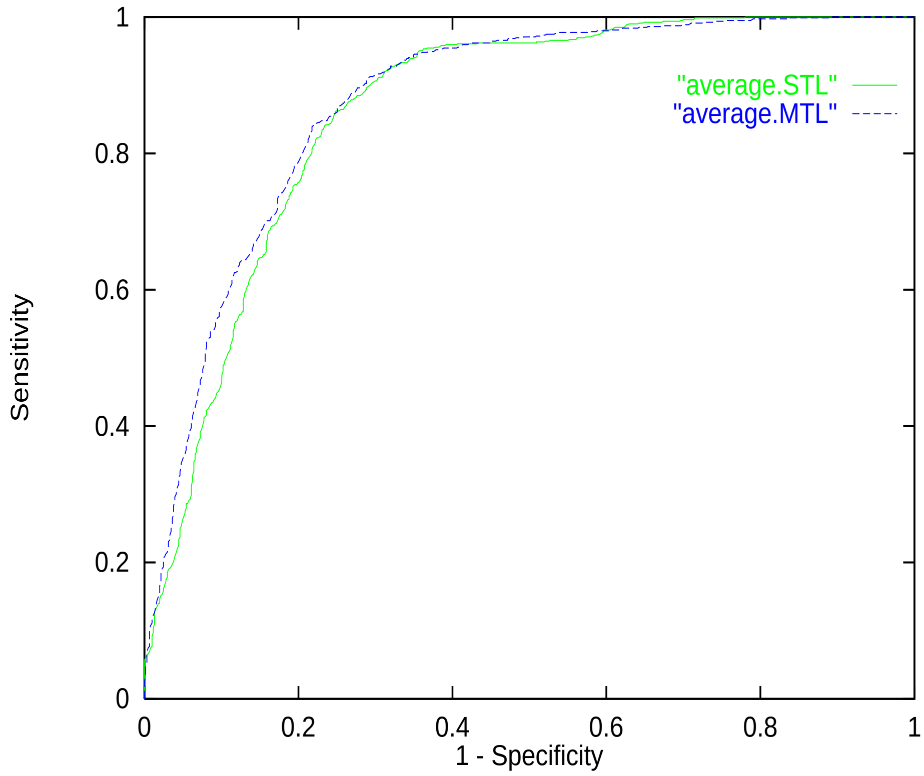


Figure 2.12: The ROC curve for MTL dominates the ROC curve for STL in most regions of the graph.

researchers applied rule-learning methods, Bayesian methods, and logistic regression to the PORT problem. These researchers are experts in the methods they tried, and ran their experiments concurrently with our STL and MTL experiments; we did not know how well other methods would perform at the time we ran our experiments . Backprop nets trained with STL and MTL yielded better performance than all other methods tried on this problem. MTL is currently the best performing learning method we know of on this problem.

2.4.7 Combining Multiple Models

The results reported in Table 2.4 are the average performance of models trained on 1200 cases and early stopped on 401 cases. Because of the need to do early stopping, none of the models was ever trained on the entire 1601 cases available for training. Because the same test set is used for all ten trials, we can combine the predictions from each of the models for the ten trials to make one prediction for each case in the test set. Because the 1200 cases used for training are randomly re-sampled from the 1601 cases for each trial, this combined prediction better utilizes the total 1601 cases available for training.

To combine the ten models we averaged their predictions for each case in the test set. Table 2.5 shows the ROC areas for STL and MTL before and after combining model predictions this way. As expected, combining model predictions does improve the performance of both STL and MTL. The effect, however, is not large. Moreover, the benefit of using MTL is almost as large after model predictions are combined as it is before they are combined. This demonstrates that in this domain the benefits of MTL are not the same benefits one achieves by combining multiple experts, and that some of the benefits achieved by combining multiple experts are additive with the benefits of MTL.

Table 2.5: ROC Area of STL and MTL on PORT DIREOUT prediction before and after the predictions of the ten different models (from the ten trials) are combined. The entry in the bottom right cell is the difference between STL without model combining and MTL with model combining.

	STL	**MTL**	**Difference**
Before Combining	0.8617	0.8748	-10.46 %
After Combining	0.8649	0.8769	-8.88 %
Difference	-2.31 %	-1.68 %	-11.00 %

2.5 Chapter Summary

In this chapter we demonstrated multitask learning on four realistic domains. The first domain, 1D-ALVINN, used data generated using a simulator written by Pomerleau. The only modifications made to this simulator were to make some internal variables available as extra task signals, and to simplify the problem by learning from a single horizontal stripe in the image. MTL reduced error on the important steering task in this domain by about 20%.

The second domain, 1D-DOORS, was an attempt to create a domain similar to 1D-ALVINN that used real data instead of data generated with a simulator. In this domain, a robot wandered down the hallway in Wean Hall and collected images of doorways. The main tasks in this domain were to find the horizontal location of doorknobs in the images and to estimate the width of the doorways. Although the domain was designed and the data were collected specifically for experiments with MTL, every attempt was made to make this domain realistic. The robot imaged doorways from different distances, different angles, and under different illuminations, and the types of doorways included in the data set were as varied as the 5th floor of Wean Hall permits. In 1D-DOORS, MTL improved accuracy on the main tasks by about 25%.

The third domain, Medis pneumonia risk prediction, is a real medical decision making problem. The data were collected by researchers other than the author. We got involved in this project not because it looked promising for MTL, but because we were asked to apply backprop to the domain for comparison with other learning methods. Moreover, the rankprop method developed by the author (with Shumeet Baluja and Tom Mitchell) for *single task learning* in this domain is the strongest single task competitor we know of. (When we first saw how much Rankprop improved the performance of STL on this domain we didn't think there was room left for improvement with MTL.) MTL reduces error 5–10% when trained on samples containing 1000 training cases. Additional experiments demonstrated that this benefit could not be achieved with feature nets, an alternate method for using the extra task signals. (In Section 6.3.3 we show how to combine MTL with feature nets to get even better performance.)

The fourth domain is the PORT pneumonia risk prediction problem. The data used in the PORT problem is very different from the data used in the Medis problem. There is an order of magnitude less data, an order of magnitude more features per case, the features used as extra outputs in Medis are used as inputs in PORT, and most of the extra tasks used for MTL in PORT are measurements not available in the Medis dataset. The prediction problems are also different. In Medis the goal was to predict which patients are at least risk from pneumonia. In PORT the goal is to predict patients that will experience a dire outcome (i.e., those patients at high risk) from their pneumonia. Rankprop, which was used for the Medis problem, was not used for the PORT problem. With PORT we used traditional SSE on 0/1 targets. The error metrics used for PORT and Medis also differ. In PORT we measure the overall accuracy of the prediction, it's positive and negative predictive values, it's sensitivity and specificity, and the ROC curve area. On the PORT *DireOoutcome* problem, MTL reduces error on all measures, yielding an average reduction in error of 8.75%, including an 10.5% reduction in error on the important ROC area measure.

From the results reported in this chapter, we conclude MTL works on real problems and yields large enough improvements to be worthwhile.

Chapter 3

How Does It Work?

Chapters 1 and 2 showed that MTL works in backprop nets. How do MTL backprop nets benefit from the information in the training signals of the other tasks? What relationships between tasks enable MTL backprop to work? How do backprop nets trained with MTL know *how* tasks are related?

This chapter is an attempt to define what "related" means. Before discussing what related tasks are, Section 3.1 briefly reviews the experiments in Section 1.4 that show MTL only works if tasks are related. Section 3.2 is an informal discussion of how tasks must be related for backprop MTL to work. Section 3.3 presents seven detailed task relationships that allow backprop MTL nets to learn better internal representations for related tasks. The mechanism that enables MTL-backprop to benefit from these relationships is the constructive and destructive interference of backprop error gradients when they are summed in the hidden layer shared by the tasks. Section 3.4 introduces the Peaks Functions, a set of synthetic problems specifically designed to elucidate how MTL in backprop works. After demonstrating that MTL works on the peaks functions, we use the peaks functions to demonstrate some of the task relationships described in Section 3.3. Section 3.5 employs the peaks functions to show that backprop MTL discovers how tasks are related without being given explicit training signals about task relatedness. In this section we introduce a measure of task sharing that examines how different tasks overlap in their use of the hidden layer. Finally, Section 3.6 returns to the issue of related tasks. In this section we propose a definition of "related" that, while not perfect, is (hopefully) a step in the right direction.

3.1 MTL Requires Related Tasks

There are many potential reasons why adding extra outputs to a backprop net might improve generalization performance. In Section 1.4 we mentioned several of these:

- Adding noise to backpropagation sometimes improves generalization [Holmstrom & Koistinen 1992]. To the extent that tasks are *uncorrelated*, their contribution to the aggregate gradient (the gradient that sums the error fed back from each layer's outputs) can appear as noise to other tasks. Thus uncorrelated tasks might improve generalization by acting as a source of noise.
- Adding tasks might change weight updating dynamics to somehow favor nets with more tasks. For example, adding extra tasks increases the effective learning rate on the input-to-hidden layer weights relative to the hidden layer-to-output weights. Maybe larger learning rates on the first layer improves learning.
- Reduced net capacity might improve generalization. MTL nets share the hidden layer between all tasks. Perhaps the reduced capacity improves generalization.

There are many others. For example, backprop might be prone to getting stuck in local minima. The extra tasks might help push the hidden layer out of inferior local minima because the local minima for different tasks might be in different places. As another example, MTL might introduce a competitive effect in the hidden layer that acts as a regularizer. When only one task is trained on a net, any hidden unit that correlates with errors on the output will be tuned for that task. But with many tasks trained on the same net, there is a competition in the hidden layer for hidden units (assuming tasks do not all require the identical hidden layer representation). This competition will cause hidden units to be trained for a task only if the force exerted by that task on that hidden unit is stronger than the forces exerted on the hidden unit by other tasks. This means only those hidden units that are most relevant to each task are trained by that task. As a final example, neural nets trained on single task suffer from a problem called the "herd" effect [Fahlman 1989] that causes all hidden units to try to fit the largest source of error early in training. Then, after some hidden units fit that error, the remaining hidden units "herd" again to try to fit the largest residual error, etc. In other words, learning in backprop nets is more serial than parallel if one looks at the sequence of concepts being learned [Weigend 1993]. But training multiple tasks on one net might reduce the herd effect by causing early differentiation of hidden units because there are many different error signals to try to fit at the same time.

There are many more potential explanations of why training multiple tasks in parallel on one backprop net might improve performance. **In MTL we are mainly interested in improvements that are due to tasks being related.** This is not because other effects might not also be worthwhile or interesting, but because MTL is a method designed for inductive transfer, and inductive transfer between unrelated tasks does not seem sensible.

Because we are only interested in MTL if it works because tasks are related, in Section 1.4 we ran experiments to try to disprove alternate mechanisms. In one experiment we trained the main task on an MTL net with extra random functions. In another experiment we trained multiple copies of the main task on an MTL net. In a third experiment we varied the number of hidden units to see if restricting the capacity of the net might improve generalization.

Each of these experiments attempts to disprove a single alternate explanation. There are many possible alternate explanation. It is impractical to devise tests to disprove each one individually. In Section 1.4 we ran an experiment that was not aimed at disproving any specific alternate mechanism. In this experiment, we shuffled the training signals for the extra tasks to disrupt the relationships between the main task and the extra tasks, while preserving the distributions of the extra tasks. The shuffle test directly tests the assumption that MTL works because tasks are related. If the benefit disappears when the relationship between tasks is broken by shuffling, this is evidence that the relationships are what was important.

In the experiments, the performance benefits from MTL disappear when the relationship between the main task and extra tasks is disrupted. Similar findings resulted from performing the shuffle test on the pneumonia risk prediction problem in Section 2.3. We conclude that the improvement from MTL is due to mechanism(s) that depend on the tasks being *related*. MTL leverages the information contained in the training signals for related tasks. If tasks are related, MTL can learn them better. If tasks are not related, MTL may learn worse than STL.

3.2 What are *Related* Tasks?

What do we mean by related? This is a difficult question. Ideally, we would like to know what extra tasks would improve performance on the main task:

$$\begin{gathered} \mathit{Related}(MainTask, ExtraTask) = 1 \\ \equiv \\ \mathit{Learning}(MainTask \| ExtraTask) > \mathit{Learning}(MainTask) \end{gathered}$$

This says we would like to define a relation *Related* which, given the main task and an extra task, is true iff learning generalizes better on that main task when it learns the extra task in parallel with the main task.

One potential problem with this definition is that it does not specify the learning procedure. Clearly, not all learning methods are equivalent. Some learning methods may be better at multitask learning than others, and some methods may be able to exploit relationships between tasks that other methods cannot. Given this, a more useful definition of *Related* might be the following:

$$\begin{gathered} \mathit{Related}(MainTask, ExtraTask, LearningAlg) = 1 \\ \equiv \\ LearningAlg(MainTask \| ExtraTask) > LearningAlg(MainTask) \end{gathered}$$

This definition acknowledges that the benefit may depend on the learning algorithm used for multitask learning.[1] This definition of *Related* is appealing, but raises several important issues. The first of these issues was mentioned in the previous section. Suppose

$$\mathit{Related}(MainTask_1, ExtraTask_1, LearningAlg_1) = 1$$

This says $MainTask_1$ is learned better when trained with $ExtraTask_1$. Suppose, however, that $ExtraTask_1$ is a random function of the inputs. $MainTask_1$ benefits from being trained with $ExtraTask_1$ because of some effect $ExtraTask_1$ has on the learning procedure $LearningAlg_1$, not because what is learned for the extra task is useful for the main task. Wouldn't it be better to modify the learning method so that it benefits from this effect

[1] The benefit also may depend on other factors such as the number of training patterns and the parameters used to control learning such as learning rates. We assume all such information is encapsulated in the descriptions of the main task, the extra tasks, and the learning algorithm.

without needing the training signals for $ExtraTask_1$? If $ExtraTask_1$ regularizes the backprop net by acting as a source of random signals, wouldn't it be better to add randomness to backpropagation by a method that does not need extra training signals? If $ExtraTask_1$ helps because it alters learning rates, wouldn't it be better to find a way to directly control learning rates? Should *Related* be defined so that extra tasks are considered *related* that are beneficial solely because they perturb learning in ways that could be achieved by modifying the learning algorithm and throwing away the extra task training signals? The extra task would become unrelated once we improved the algorithm. Should we have called the extra task *Related* in the first place?

The second issue is exemplified by the following:

$$\begin{gathered} Related(MainTask_1, ExtraTask_1, LearningAlg_1) \\ \neq \\ Related(MainTask_1, ExtraTask_1, LearningAlg_2) \end{gathered}$$

$ExtraTask_1$ benefits $MainTask_1$ with one of two different learning algorithms. Assume $LearningAlg_1$ benefits and $LearningAlg_2$ does not. Further assume that this is not a case where the extra task helps by some effect that could be achieved without the extra task training signals by modifying the algorithm. $LearningAlg_1$ is able to exploit relationships between tasks that $LearningAlg_2$ cannot exploit. As an example, suppose we have two tasks that are identical except that their training signals have been corrupted by independent noise processes.

$$Task_1 = Task + noise_1$$
$$Task_2 = Task + noise_2$$

We have more information about the task to be learned given both task signals. $Task_1$ and $Task_2$ are clearly related. An inductive transfer method that does not recognize nor benefit from this relationship is clearly imperfect. Is it reasonable to say that $Task_1$ and $Task_2$ are unrelated just because $LearningAlg_2$ is not able to benefit from the relationship? Wouldn't it be better to recognize the relationship between the tasks, and then devise algorithms that could benefit from this type of relationship?

The third issue is probably the most important. Suppose we are given two tasks. Can we determine whether the *Related* relationship is true or false for these tasks *before* applying learning? Can relatedness be judged by mechanisms different (and hopefully easier and/or more reliable) than those used for multitask learning?

We do not know of reliable ways to judge task relatedness using the information typically present when tackling real-world machine learning problems. Does this mean we must resort to trying all possible extra tasks to see which help? No. We believe heuristics can be developed to make judging task relatedness reliable enough for most practical applications of multitask learning in the real world.

Do such heuristics exist? Yes. In Chapter 2 we did not use hospital tests from pneumonia patients as extra tasks for 1D-ALVINN. It is possible that the training signals from a patient's medical history might help us learn to drive their car better, but it is unlikely. Largely unconsciously, we dismissed a nearly infinite number of potential training signals because our models of driving did not suggest relationships between those extra tasks and steering. While it is almost certainly true that we failed to recognize some extra tasks that would have benefited the steering main task in 1D-ALVINN, it is also certainly true that we successfully ignored many tasks that would not have helped steering. We did this without formal models of backpropagation, multitask learning, and autonomous navigation. We used heuristics.

Where possible, heuristics should be made precise. In this thesis we are interested in two kinds of heuristics:

1. heuristics that define what relationships between tasks can be exploited by some particular learning algorithm

2. heuristics that define relationships between tasks that should be exploitable by any good multitask learning algorithm

The remainder of this chapter is devoted to heuristics of the first type. (The next chapter is devoted to heuristics of the second type.) In Section 3.3 we will present seven specific relationships between tasks that allow backprop to benefit when learning those tasks parallel. Those relationships are the most precise definition we have for what "related" means in backprop MTL. But those relationships are somewhat abstract. In the remainder of this section we address the issue of task relatedness at a more informal level before jumping to that level of detail.

3.2.1 Related Tasks are not Correlated Tasks

Correlation is one of the simplest ways of measuring a relationship between two variables. Correlation measures the joint variation of two variables. Unlike regression, correlation does not assume one of the variables is dependent on the other. Instead, correlation assumes both variables are measured by being drawn together from a population of instances. In many ways, though, correlation is similar to regression. In this section we use linear correlation. We are not interested in models that are necessarily linear. However, we have found that many real-world relationships that are more complex than linear still have reasonably strong linear components. Thus linear correlation serves as a useful, easily computable proxy for more precise relationships between variables such as mutual information.

One definition of the correlation coefficient, ρ, for a sample of points X and Y is:

$$\rho = \frac{\sum_{i=1}^{n}(X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \overline{X})^2(Y_i - \overline{Y})^2}} \tag{3.1}$$

This formula makes it clear that correlation is a measure of the degree to which the variation of X above and below its mean co-occurs with variation of Y above and below (or below and above for negative correlations).

One might assume that for two tasks to be related, they would have to be correlated. This is not true. What counts for MTL is not that the task signals themselves be correlated, but that the internal representations that could be used for the different tasks be correlated. Related tasks are correlated, but at the level of representation, not necessarily at the output level.

To see this, consider the following synthetic tasks:

$F1(A, B) = SIGMOID(A + B)$

$F2(A, B) = SIGMOID(A - B)$

where $SIGMOID(x) = 1/(1 + e^{(-x)})$

F1(A,B) and F2(A,B) do not correlate because $A + B$ and $A - B$ do not correlate. The correlation coefficients for the training sets we create for this problem are typically less than ±0.01. Figure 3.1 shows a scatter plot of F1(A,B) vs. F2(A,B) for A and B uniformly sampled from the interval [-5,+5].

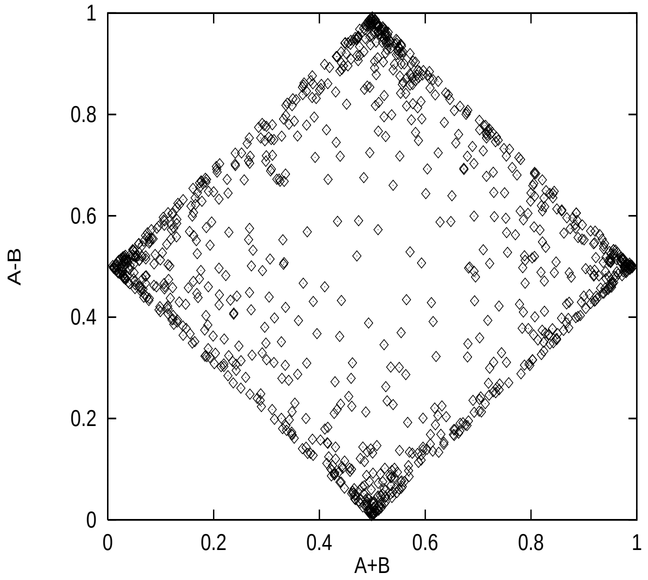


Figure 3.1: Scatter plot of F1(A,B) vs. F2(A,B) shows there is no correlation between them.

Suppose A and B are presented as inputs to a backprop net by coding them first using the standard powers-of-2 binary code. If we use 10 bits for A and 10 bits for B, there are 10 inputs to the net coding for A, and 10 inputs to the net coding for B. A net trying to learn either F1(A,B) or F2(A,B) must learn to decode the inputs. Although F1(A,B) and F2(A,B) do not correlate, the internal representations each might learn in a hidden layer to decode the inputs strongly correlate.

We trained an STL net on just F1(A,B), and an MTL net on both F1(A,B) and F2(A,B). The STL net has 20 inputs, 16 hidden units, and one output. The MTL net has the same architecture, but two outputs, one for F1(A,B) and one for F2(A,B). We generate training sets containing 50 patterns. This is enough data to get good performance with STL on F1(A,B) or F2(A,B), but not so much that there is not room for improvement. Each trial uses new random training, halt, and test sets. We use large halt and test sets—1000 cases each—to minimize the effect of sampling error in the measured performances. The target outputs are the unary real (unencoded) values for F1(A,B) and F2(A,B).

Table 3.1 shows the mean performance of 50 trials of STL and MTL. The MTL net generalizes a little better. F1(A,B) and F2(A,B) are not correlated, but they are related because both benefit from decoding the binary input encodings for A and B. That is, F1(A,B) and F2(A,B) have correlated hidden layer representations. All related tasks are not correlated (but all correlated tasks are related).

Table 3.1: Mean Test Set Root-Mean-Squared-Error on F1

Network	Trials	Mean RMSE	Significance
STL	50	0.0648	-
MTL	50	0.0631	0.013*

The improvement of MTL over STL in the previous experiment is small (though statistically significant). Many of our experiments with synthetic functions, both in this chapter and throughout the rest of this thesis, show modest improvements. This is because the benefit from any one task is usually small, particularly if that task has been designed to take advantage of only one or two of the task relationships to be presented in Section 3.3. Larger effects can be obtained by adding more extra tasks to these synthetic domains. We use as few extra tasks as possible with synthetic problems to keep them simple, and count on careful experiments with many trials and large test sets to show the effect. Using only *one* extra task is a kind of worst case for MTL.

3.2.2 Related Tasks Must Share Input Features

Suppose we have two tasks, T1 and T2, that are functions of *subsets* of the inputs to the backprop net, I_{T1} and I_{T2}. If $I_{T1} \cap I_{T2} = \emptyset$, the tasks are not related in a way that backprop MTL can benefit from. If T1 uses only I_{T1}, and T2 uses only I_{T2}, then the functions they compute from these disjoint sets of input features are also disjoint, and no sharing at the hidden layer can occur. Simple MTL nets share only the weights in the input-to-hidden layer (i.e., they share the hidden layer). They do not share the hidden-to-output layer weights. (See [Ghosn & Bengio 1996] for experiments with MTL architectures that do share the output weights.)

This is a limitation of the backprop MTL method discussed in this thesis. Other approaches to inductive transfer might not have this limitation. The basic problem is that backprop nets are propositional. They cannot learn first-order theories. This restricts MTL to benefiting from tasks that are related by propositional functions of the inputs.

One way to measure task relatedness is to measure the amount of overlap in the input features they share. Tasks that share many input features are likely to be more related because they are more likely to share hidden units computed on those input features. We use this measure of task relatedness later in Sections 3.4 and 3.5.

3.2.3 Related Tasks Must Share Hidden Units to Benefit Each Other when Trained with MTL-Backprop

It is not sufficient that tasks overlap in the input features they use. Two tasks that compute completely different functions (and subfunctions) of the same input features probably will not benefit each other when trained together. For MTL-backprop to benefit from related tasks, the tasks must share some of the representation learned at the hidden layer. Given that there are usually many internal representations that could be learned for a task, it is difficult to make this notion of task relatedness operational. Backprop nets often learn internal representations surprisingly different from the representations we expect (or want) them to learn. Tasks that look completely unrelated when we write down the mathematical description for them often turn out to be related in the hidden layer representations actually learned by the nets.

3.2.4 Related Tasks Won't Always Help Each Other

Just because tasks are related does not mean they will necessarily help each other. The effect of learning related tasks together depends on the algorithm. Better algorithms will benefit more from different task relationships. To see this, consider the simple case of a modified backprop algorithm that trained multiple outputs by first training the net to completion on output 1, then trained the net to completion on output 2, but using only those hidden units not already used by output 1. This is a legitimate algorithm for training nets with multiple outputs, but it specifically precludes MTL. It would be wrong to decide that tasks are unrelated just because this algorithm did not benefit from the relationships. The tasks are related, but the algorithm failed to take advantage of it. The goal for a theory of task relatedness in MTL, and inductive transfer in general, is to find those task relationships that different algorithms can take advantage of. If we find task relationships that an algorithm cannot benefit from (or is hurt by), then the mission is to improve the algorithm.

3.2.5 Summary

Tasks can be related in ways that backprop MTL cannot benefit from. For backprop MTL to work, tasks must be functions of some of the same inputs, and some of the subfunctions computed from those inputs must be correlated with each other. In the next section we present seven specific relationships between tasks that MTL-backprop is able to use to develop a better hidden layer representation.

3.3 Task Relationships that MTL-Backprop Can Exploit

This section presents seven specific task relationships that can improve generalization in backprop nets trained simultaneously on tasks with these relationships. The mechanism that allows backpropagation to benefit from these relationships is the summing of error gradient terms at the hidden layer from different task outputs. Each of the seven relationships benefits from this error gradient summing in a somewhat different way. For any of these task relationships to benefit learning, backprop must perform an *unsupervised* clustering of what is learned at the hidden layer for different tasks because it is not given explicit training signals about how tasks are related. This critical unsupervised learning component of MTL is examined in Section 3.5.

3.3.1 Data Amplification

Data amplification is an *effective* increase in sample size due to extra information in the training signal of related tasks. There are three types of data amplification.

Statistical Data Amplification

Consider two tasks, T and T'. T and T' are two target functions to be learned from finite training sets. For simplicity, we assume that there are the same number of training patterns for T and T', that T and T' are both to be learned from the same input features, I, and that we have the training signals for both T and T' for the same input vectors. In other words, we have one training set and for each pattern in this training set we have the training signal for both T and T'.

Statistical amplification, occurs when there is noise in the training signals. Suppose T and T' have independent noise added to their training signals. Further suppose that both T and T' benefit from computing a hidden layer feature F of their common inputs.

$T(I) = F(I) + G(I) + epsilon1$

$T'(I) = F(I) + H(I) + epsilon2$

where $epsilon1$ and $epsilon2$ are independent noise sources. A net learning both T and T' can, if it recognizes that the two tasks share F, use the two training signals to learn F better by averaging F through the different noise processes. The simplest case is when $G(I) = H(I)$, so that $T = T'$. In this case the two outputs are independently corrupted versions of the same signal. If this is known apriori, we can train a net on one task whose training signals are the average of the training signals for T and T'. But this situation rarely occurs in practice, and training on the average training signal is incorrect otherwise. Training one MTL net with separate outputs for T and T' is more widely applicable.

Sampling Data Amplification

Sampling amplification is similar to statistical amplification, but occurs when there is no noise in the training signals. Consider two tasks, T and T', with no noise added to their training signals, that both benefit from computing a hidden layer feature F of the inputs.

$T(I) = F(I) + G(I)$

$T'(I) = F(I) + H(I)$

Learning T or T' well from a small training sample may be difficult because nonlinear regions in the functions T or T' may not be sampled adequately by the small training sample. The number of data points necessary to adequately sample a nonlinear function of

I input dimensions grows exponentially with the number of input dimensions in the worst case. We rarely have enough training data in high dimensional spaces to fully characterize complex functions. Because of this, there may be many different models in the class of functions being learned that have similar error on this particular finite training set. In other words, we have more free parameters in our model class than we can set reliably with the small training sample.

By supposition, both T and T' benefit from computing the same hidden layer feature F from the inputs. A net learning both T and T' can, if it recognizes that the two tasks share F, use the two training signals to learn F better because the two different uses of F made by T and T' can yield different samples of F. If T and T' are such simple functions of F that nets training on T or T' would both see high-fidelity error signals at the hidden layer for F, then the net may not benefit from training T and T' if the training samples for T and T' are identical. This is because F is a function of the inputs I, and there is little benefit to internally computing the same error signals for F on the same sets of inputs. If, however, the training patterns for T and T' are different samples from the input space, then the error signals computed internally for F are not redundant. In this case, training both T and T' together on one net doubles the effective sample size for F. This can improve the learning of F, which may then help T and T' be learned better. The simplest case of this is when $T = T'$, i.e., when the two outputs are the same function and we have different training samples for T and T'. This is a rather trivial case, however, and if we knew T and T' had this relationship we would do better by pooling the independent samples.

A more interesting, and more realistic, case of sampling amplification occurs when T and T' both benefit from F as before, but T and T' are different enough, and complex enough, functions of F that small training sets do not provide reliable error signals for F in nets learning T or T'. In these cases, the error signals for F provided by T and T' are not redundant, even if T and T' are sampled by the same training set. In this case, a net learning both T and T' in parallel can, if it recognizes that T and T' share F, learn a better internal model of F by combining the error signals for F backpropagated by T and T'. One of the simplest cases of this is:

$T(I) = c_1 * F(I)$

$T'(I) = c_2 * F(I)$

Here T and T' are both simple linear functions of a common F, but the slopes c_1 and c_2 in the linear relationships must be estimated from the data. Given a small training sample (relative to the complexity of F), the estimates for c_1 and c_2 will be imperfect and thus will yield different error signals for F, even at the same points in the input space. A net learning both T and T' in parallel can combine these estimates to yield an improved internal model of F, which in turn will yield improved estimates of c_1 and c_2 and, therefore, learned models for T and T'.

Blocking Data Amplification

The third form of data amplification is really just an extreme form of sample amplification. Consider two tasks, T and T', that use a common feature F computable from the inputs, but each uses F for different training patterns. A simple example is:

$T = A \bigvee F$

$T' = NOT(A) \bigvee F$

(The parity functions from Section 1.3 are instances of this class.) T uses F when $A = 0$ and provides no information about F when $A = 1$. Conversely, T' provides information about F only when $A = 1$. A net learning just T gets information about F only on training patterns for which $A = 0$, but is *blocked* when $A = 1$. But a net learning both T and T' at the same time gets information about F on every training pattern; it is never blocked. *It does not see more training patterns, it gets more information for each pattern.* If the net learning both tasks recognizes the tasks share F, it will see a larger sample of F. Experiments with blocked functions like T and T' (where F is a hard but learnable function of the inputs such as parity) indicate backprop does learn common subfeatures better due to the larger effective sample size.

3.3.2 Eavesdropping

Consider a feature F, useful to tasks, T and T', that is easy to learn when learning T, but difficult to learn when learning T' because T' uses F in a more complex way. A net learning T will learn F, but a net learning just T' may not. If the net learning T' also learns T, T'

can *eavesdrop* on the hidden layer learned for T (e.g., F) and thus learn better. Moreover, once the connection is made between T' and the evolving representation for F, the extra information from T' about F will help the net learn F better from the other relationships. The simplest case of eavesdropping is when $T = F$. Abu-Mostafa calls these *catalytic hints* [Abu-Mostafa 1990]. In this case the net is being told explicitly to learn a feature F that is useful to the main task. Eavesdropping sometimes causes non-monotonic generalization curves for the tasks that eavesdrop on other tasks. This happens when the eavesdropper begins to overfit, but then finds something useful learned by another task, and begins to perform better as it starts using this new information.

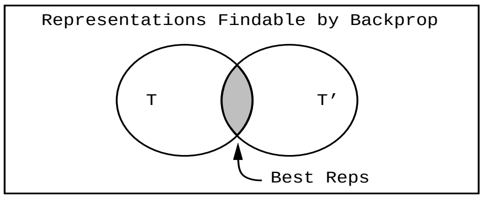
3.3.3 Attribute Selection

Consider two tasks, T and T', that use a common subfeature F. Suppose there are many inputs to the net, but F is a function of only a few of the inputs. A net learning T will, if there is limited training data and/or significant noise, have difficulty distinguishing inputs relevant to F from those irrelevant to it. A net learning both T and T', however, will better select the attributes relevant to F because data amplification provides better training signals for F and that allows it to better determine which inputs to use to compute F. (Attribute selection depends on data amplification. Data amplification, however, does not depend on there being an attribute selection problem.)

We've run experiments with synthetic problems where we artificially create an attribute selection problem. MTL's attribute selection mechanism is observed by comparing the magnitude of the weights in the input-to-hidden layer for the relevant input features with those for the irrelevant features. The weights to the relevant features grow more quickly when the net is trained with MTL. See Section 3.4.5.

3.3.4 Representation Bias

Because nets are initialized with random weights, backprop is a stochastic search procedure; multiple runs rarely yield identical nets. Consider the set of all nets (for fixed architecture) learnable by backprop for task T. Some of these generalize better than others because they better "represent" the domain's regularities. Consider one such regularity, F, learned differently by the different nets. Now consider the set of all nets learnable by backprop for another task T' that also learns regularity F. If T and T' are both trained on one net and the net recognizes the tasks share F, search will be biased towards representations of F near the intersection of what would be learned for T or T' alone. We conjecture that representations of F near this intersection often better capture the true regularity of F because they satisfy more than one task from the domain.



A form of representation bias that is easier to experiment with occurs when the representations for F sampled by the two tasks represent different local minima in representation space. Suppose there are two minima, A and B, a net can find for task T. This means that a net learning T can find two distinct hidden layer representations for T, and that all paths in weight space joining these two distinct representations yield higher training-set error than either of the local minima. The local minima located at A and B are surrounded by basins of attraction where the error gradient leads to only one of the minima. For simplicity, assume that A and B are the only two minima for task T, and that the basins of attraction for A and B are similar in size.

Suppose a net learning task T' also has only two local minima, A and C, both of which also have attractor basins of similar size. Note that both T and T' share the local minimum at A, but B and C are dissimilar, i.e., are located in different regions of weight space. Suppose A, B, and C are roughly equidistant from each other and from the origin. (This is likely in a high dimensional weight space.) Backprop nets are initialized near the origin in weight space. The backpropagated error signals for a net trained on both T and T' in parallel tend to constructively interfere in the direction towards A, but not in directions towards B or C. In fact, because of our assumption that B and C are equidistant from the origin and each other, gradients towards B partially cancel gradients towards C on average, and vice-versa. (This destructive interference is only partial.) Because of this, the combined gradient for the hidden layer towards A will likely be stronger than the combined gradient for B or C, so the net is more likely to fall into the representation for A.

Because A is a local minimum for both T and T', there is little pressure for either the output for T or T' to pull away from the A representation once one of the two tasks falls into A's attractor basin. If T moves towards B, however, this does not reduce error on T'. (By supposition T' has local minima only at A and C, and B is not near A, and thus not in the direction of A.) Task T' must either learn zero-valued weights to the representation forming in the hidden layer for A, or counter the "tide" towards B. If T is already too deep in the attractor basin for B to be significantly affected by T', the hidden layer representations for F learned by T and T' will become disjoint. But this situation is unlikely to arise given the bias early in search towards A, the representation they share.

We ran two experiments to test this. In the first, we selected the minima so that nets trained on T alone are equally likely to find A or B, and nets trained on T' alone are equally likely to find A or C. Nets trained on both T and T' usually fall into A for both tasks.[2] *Backprop nets with two or more outputs tend to use hidden layer representations that can be used by two or more tasks.*

In the second experiment we selected the minima so that T has a strong preference for B over A: a net trained on T always falls into B. T', however, still has no preference between A or C. When both T and T' are trained on one net, T falls into B as expected: the bias from T' is unable to pull it to A. Surprisingly, T' usually falls into C, the minimum it does not share with T! T creates a "tide" in the hidden layer representation towards B that flows away from A. T' has no preference for A or C, but is subject to the tide created by T. Thus T' usually falls into C; it would have to fight the tide from T to fall into A. *Backprop nets with two or more outputs tend to not use hidden layer representations for any one output that other outputs tend to avoid.*

[2]In these experiments the nets have sufficient capacity to find independent minima for the tasks. They are not forced to share the hidden layer representations. But because the initial weights are random, they do initially share the hidden layer and will separate the tasks (i.e., use independent chunks of the hidden layer for each task) only if learning causes them to.

3.3.5 Overfitting Prevention

Suppose tasks T and T' both use feature F. Suppose T has trained to the point where it would begin to overfit F if T were trained in isolation. Two situations can help prevent T from overfit F, and this in turn will tend to prevent T from overfitting.

Suppose T' has not yet reached the point where it will overfit F. Then T' provides a gradient that continues to drive F towards better models instead of towards overfitted models. If the net recognizes that both T and T' overlap on F, and this leads to sharing the hidden layer representation for F, T' will provide a pressure that tends to keep F from overfitting.

The second situation is similar. Suppose T and T' both depend on F in different ways. Perhaps both T and T' are ready to begin overfitting F. However, because they use F differently, changes in F will affect T and T' in different ways. Since they share F, any direction that reduces error on T by changing F, but which raises error on T' as a result of that change, will be disfavored. The only changes in F that will be allowed are those that lower error on both T and T'. We conjecture that there are fewer changes available that lower error on both T and T' and which are overfitting F, so overfitting of F should be less likely. Tasks should be less likely to overfit if they share more features with other tasks.

3.3.6 How Backprop Benefits from these Relationships

The "tide" mentioned while discussing representation bias results from the aggregation of error gradients from multiple tasks at the hidden layer. Because nets are randomly initialized, most movement in the hidden layer caused by one output is "felt" by all tasks. This random initialization is critical to MTL-backprop. It allows error gradients to constructively and destructively interfere in the shared hidden layer, and this biases the search trajectory towards better performing regions of weight space. Because the different relationships between tasks are all exploited by this same mechanism, it easy for the relationships to act in concert. Their combined effect can be substantial.

Each relationship between tasks has different effects on what is learned. Changes in architecture, representation, and the learning procedure affect may alter the way backprop benefits from tasks with different relationships in different ways. One particularly noteworthy difference between the relationships is that if there are local minima, representation bias affects learning even with infinite sample size. The other relationships are effective only with finite sample size: data amplification (and thus attribute selection), eavesdropping, and overfitting prevention are beneficial only when the sample size is too small for the training signal for one task to provide enough information to the net for it to learn good models.

3.4 The Peaks Functions

Section 3.3 presented seven task relationships that MTL-backprop can benefit from. It would help if we had a set of tasks where there are many different relationships between the tasks and where the relationships between the tasks is known apriori. We created the Peaks Functions to serve this purpose.

3.4.1 The Peaks Functions

We devised a set of test problems called the Peaks Functions. Each peak function is of the form:

```
IF (?1 > 1/2), THEN ?2, ELSE ?3
```

where ?1, ?2, and ?3 are instantiated from the alphabet {A,B,C,D,E,F} without duplication. There are 120 such functions:

```
P001 = IF (A > 1/2) THEN B, ELSE C
P002 = IF (A > 1/2) THEN B, ELSE D
...
P014 = IF (A > 1/2) THEN E, ELSE C
...
P024 = IF (B > 1/2) THEN A, ELSE F
...
P120 = IF (F > 1/2) THEN E, ELSE D
```

The variables A–F are defined on the real interval [0,1]. A–F are provided as inputs to a backprop net learning peaks functions. The values for A–F are given to the net via an encoding, rather than as simple continuous inputs. A net learning peaks functions must not only learn the functions, but must learn to properly decode the input encodings. The encoding we used has ten inputs for each of the six inputs A–F, so there are 60 inputs altogether. Figure 3.2 shows the input representation used for peaks functions. Each variable has 10 inputs. We use a Gaussian peak with standard deviation 0.1 and height 0.5 centered at the real value for that variable to code for the value.

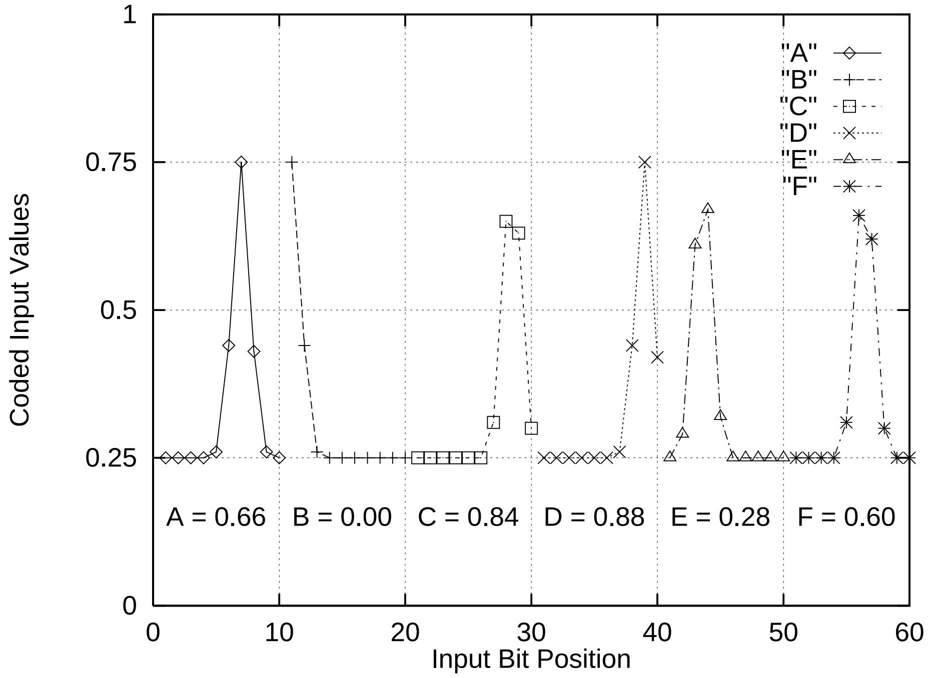


Figure 3.2: Input Representation Used for the Peaks Functions

We generate synthetic data for the Peaks Problems by uniformly sampling values for the variables A–F from the interval [0,1]. The numbers are then coded using the input encoding in Figure 3.2. This gives us the 60 real-valued inputs for the net that code for the 6 input variables. For each set of values, we compute all 120 function values for the functions P001–P120. Since this is a synthetic domain defined on reals, we can generate as much data as we want. We use relatively small training sets to keep learning interesting; these problems can be learned perfectly by backprop given large enough training samples.

The relatedness of two peaks functions depends on how many variables they have in common, and whether they use those variables in the same way. For example, P001 does not share any variables with P120, so it is not related to P120. P001 shares two variables with P024, though neither of these is used in the same way; P001 is moderately related to P024. P001 also shares two variables with P014, and both variables are used the same way. Thus P001 is more related to P014 than to P024.

We have run enough experiments on the peaks functions to fill a book. Out of consideration for the reader, we present here only the results of some of the more interesting experiments.

3.4.2 Experiment 1

The first experiment demonstrates that MTL is effective with the peaks functions. We trained six strongly related peaks functions on an MTL net, and compared the performance with an STL net trained on one of these at a time. We used the six peaks functions that are all defined on the variables A, B, and C. These are:

```
P001 = IF (A > 1/2) THEN B, ELSE C
P005 = IF (A > 1/2) THEN C, ELSE B
P021 = IF (B > 1/2) THEN A, ELSE C
P025 = IF (B > 1/2) THEN C, ELSE A
P041 = IF (C > 1/2) THEN A, ELSE B
P045 = IF (C > 1/2) THEN B, ELSE C
```

These are strongly related functions as they are all defined on the same subfeatures. The nets have 60 inputs, 10 inputs for the codings for each of the six variables A–F.

Figure 3.3 shows two graphs. The left graph is the test-set training curves for Task P001. The right graph is for Task P005. Each graph shows the performance of STL on the task, MTL with 5 additional copies of the same task (i.e., the six outputs of the net all receive the same training signals), and MTL when that task is trained with the other 5 strongly related peaks tasks listed above. The training sets contain 25 training patterns (the peaks functions have been carefully tweaked by adjusting the input representations and

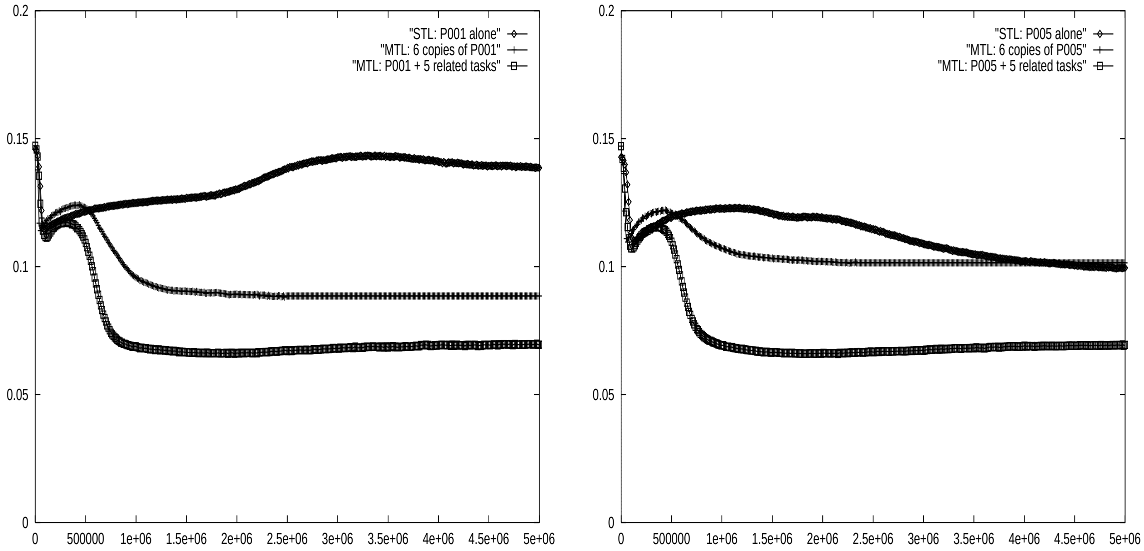


Figure 3.3: Generalization performance of STL, MTL with six copies of the same task, and MTL with six strongly related tasks on P001 (left) and P005 (right). The vertical axis is the RMS error on the test set. Lower error indicates better performance. The horizontal axis is the number of backprop passes. In each graph the best performing curve is for MTL with six strongly related tasks. In the left graph MTL with six copies of P001 performs better than STL of P001, but in the right graph STL of P005 performs slightly better than six copies of P005.

the complexity of the boolean function that combines the input terms so that reasonable performance can be obtained with training sets as small as 25–100 cases, making it practical to run many experiments.) The test sets contain 350 cases. Note, these graphs are not the average of multiple runs. These are the results from single trials. We've examined the training curves for additional runs to insure that this behavior is typical.

The vertical axis is root-mean-squared error on the test set. Zero error would be perfect generalization. It is clear from the graphs that STL performs poorest. Training a net with six copies of the task improves performance. But training an MTL net on six strongly related peaks tasks yields the best performance. MTL works on the peaks tasks when the tasks are related.

3.4.3 Experiment 2

In Section 3.4.2 we trained six strongly related peaks functions on an MTL net and observed better generalization. What happens if we train five strongly related peaks functions, and one completely unrelated peak function, on an MTL net with six outputs? The five related tasks should benefit each other, but the unrelated sixth task should see no benefit from the other five tasks.

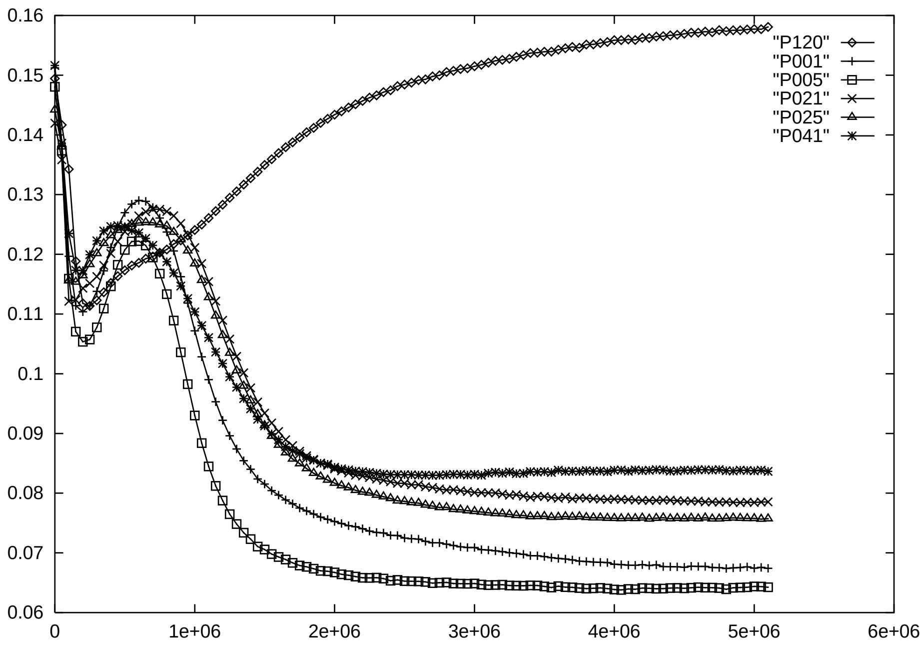


Figure 3.4: Generalization performance of five strongly related functions compared with the performance of one completely unrelated function when trained on one MTL net.

Figure 3.4 shows the test-set training curves for six tasks trained on an MTL net. Five of the tasks are P001, P005, P021, P025, and P041 from above. They are strongly related. The sixth task is P120. It does not share any features with the other five. Once again, lower error indicates better generalization. It is clear from the graph that the five related tasks benefit from each other. But the task that has no related tasks does not benefit. In fact, performance on task P120 is somewhat worse than it would be if it were trained alone on an STL net. Training it with the other tasks hurts it.

The results from this experiment provide the strongest evidence we have that some of the benefits of MTL depend on tasks being related. What makes this evidence so strong is the fact that the output distributions for all peaks functions are identical. The training signals for each peak function is the value of one of the variables A–F. Variables A–F are all drawn uniformly from the interval [0,1]. They all have the same distribution. Thus changing which peaks functions are trained on a net has no effect on the distribution of the outputs. Moreover, unlike with the shuffle test, any peaks function is as learnable from the inputs as any other peaks function. Shuffling maintains the output distributions, but makes the shuffled functions harder to learn because it also destroys the relationship between the inputs and the task signals for the shuffled function.

This experiment convincingly demonstrates that some of the benefit we see with MTL is due to relationships between the tasks trained on the MTL net.

3.4.4 Experiment 3

This experiment compares the generalization performance of STL and MTL on the six strongly related peaks functions as a function of the number of training patterns in the training set. This experiment was run by Joseph O'Sullivan. He is using the peaks functions to explore combining MTL with Explanation-Based Neural Nets (EBNN), a serial inductive transfer method developed by Mitchell and Thrun. (See Section 8.3 for more information about EBNN and the results of an experiment run by O'Sullivan to compare MTL and EBNN on a robotics task.)

Figure 3.5 shows the test-set performance of STL and MTL on Task P001. The measure used is the average percent accuracy of the prediction for the task. The error bars are 95% confidence intervals. When the number of training patterns is low, both methods perform comparably. As the number of training patterns increases, MTL begins to outperform STL. In the region between 50 and 100 training patterns, MTL performs as well as STL with 30–100% more data. STL performs so well with 120 or more cases in the training set that there is less room for MTL to do better.

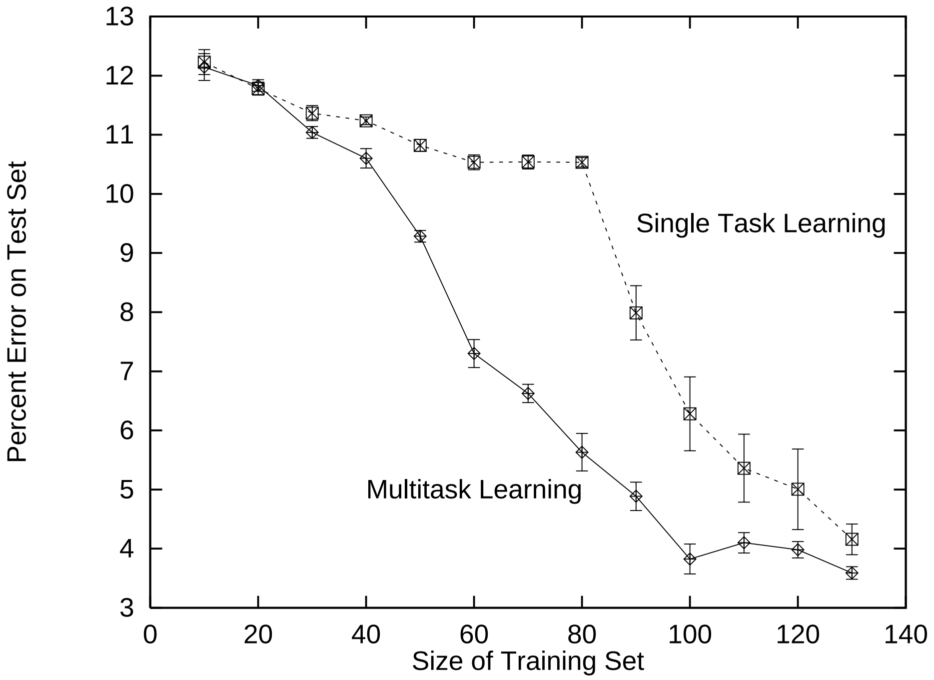


Figure 3.5: Generalization performance of STL and MTL on P001 as a function of the number of training patterns.

3.4.5 Feature Selection in Peaks Functions

One use of the peaks functions is to demonstrate that MTL-backprop can benefit from the relationships between tasks described earlier in Section 3.3. In this section we demonstrate the feature selection mechanism described in Section 3.3.3. Feature selection allows backprop to better differentiate between relevant and irrelevant inputs for the tasks being learned. We trained STL and MTL nets on the strongly related tasks used in Experiment 1. The STL net is only trained on one task at a time. All of these tasks are functions of only the inputs that code for A, B, and C. The inputs coding for D, E, and F are still given to the nets, but they are irrelevant for these tasks.

During training, we measured the sensitivity of what was learned to the different sets of inputs. We did this by computing the average derivative of the output for P001 to each input on the input vectors in a large test set. There are 60 inputs, so we computed sixty derivatives for each point in the test set. The average of each derivative tells us how sensitive the output is to changes in that input. We then averaged the absolute value of the sensitivities for inputs A–C, and for inputs D–F. This gives us the average sensitivity of what is learned to the relevant (A–C) and irrelevant (D–F) inputs.

Figure 3.6 plots the average sensitivities to the two different groups of inputs for STL (left) and MTL (right). Sensitivity to all inputs increases with the number of backprop passes. This is expected. Backprop does a poor job of keeping weights low for irrelevant inputs. (This is one of the reasons why overfitting is such a problem with backprop.) In both graphs, sensitivity to the relevant inputs (those for A,B,C) increases faster than sensitivity to the irrelevant inputs (those for D,E,F). But the relative sensitivity to the relevant vs. irrelevant inputs grows faster in the MTL net than in the STL net. The MTL net does a better job of determining what inputs are relevant for P001.

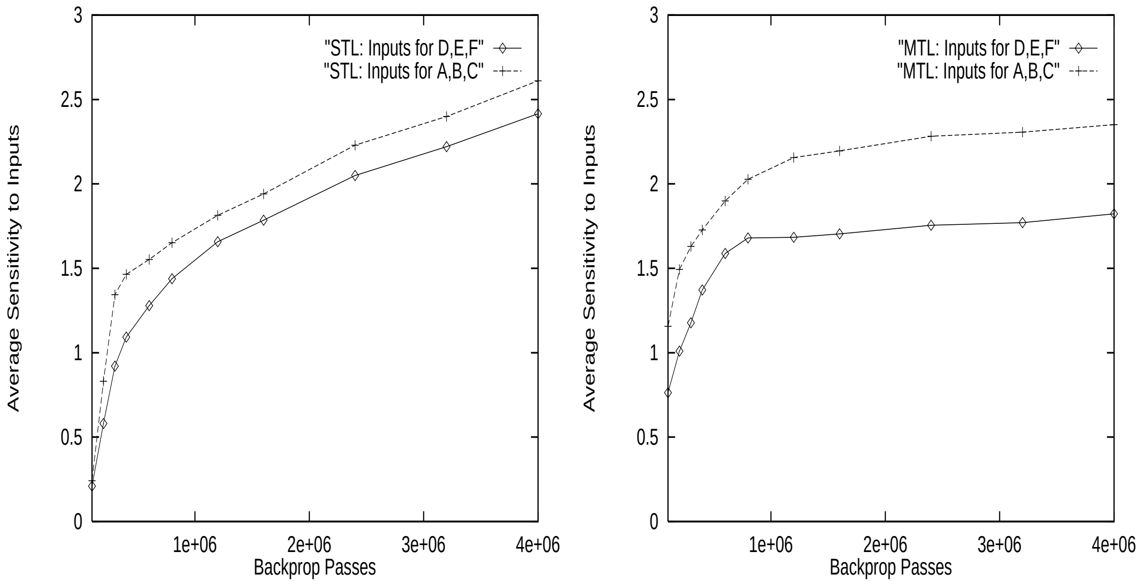


Figure 3.6: Sensitivity of STL (left) and MTL (right) to the inputs coding for A, B, and C, and the inputs coding for D, E, and F. Only A, B, and C are relevant inputs for the functions being trained.

The graphs also demonstrate that MTL helps prevent overfitting. The sensitivity to the inputs levels off and is nearly flat beyond about 1,000,000 passes for MTL. The STL net, however, which is poorer at distinguishing the two sets of inputs, is still rapidly increasing its sensitivity to the inputs after 3,000,000 passes. The STL net is overfitting more than the MTL net.

3.5 Backprop MTL Discovers How Tasks Are Related

Section 3.3 presented seven relationships between tasks that MTL backprop nets can exploit to learn those tasks better. In that section we frequently used the phrase "a net learning both T and T' can, if it recognizes that the two tasks share F, use the two training signals to learn F better by...." MTL nets are not told how tasks are related. They are not told which tasks share F, or even what the many F for the different tasks are. Do MTL backprop nets discover how tasks are related? Clearly they must. We have shown on synthetic problems that the benefit of MTL-backprop depends on the tasks being related. If tasks are not related, or if the relationship between tasks is broken by shuffling training signals, the benefit of MTL-backprop disappears. If the benefit of MTL-backprop depends on exploiting task relationships, MTL backprop nets must discover some of the relationships between tasks themselves. They are not told how tasks are related.

Backprop nets, though primarily used for supervised learning, perform a limited kind of unsupervised learning on the hidden layer features learned for different tasks (different outputs). The details of how this unsupervised learning occurs and how well it works are not yet fully understood. It is worthwhile, however, to demonstrate here that backprop does discover task relatedness.

We trained one MTL net on all 120 peaks functions. This net has 60 inputs and 120 outputs, one for each of the 120 peaks functions. We examined the weights to see how much different outputs shared the hidden layer. We did a sensitivity analysis for each output with each hidden unit at the input points contained in a representative test set. There are 120 outputs and 64 hidden units, so we did 15,360 sensitivity analyses. By comparing the sensitivity of output P001 to each hidden unit with that of output P002 to each hidden unit, we are able to measure how much outputs P001 and P002 share the hidden units in the hidden layer.

We compare the sensitivity of different outputs to the 64 hidden units by first ranking the hidden units for each output sensitivity. Hidden units that are more important to the output are ranked higher. We compare how much different outputs share the hidden layer by computing the rank correlation on the rankings obtained for different hidden units. Rank-correlation is a nonparametric measure similar to correlation. One way to compute the rank correlation is to compute the ranks for the measurements X and Y independently and then compute the usual continuous correlation coefficient of these ranks. A simpler method is to use the formula:

$$RankCorrelation = 1 - \frac{6(\sum_{i=1}^{N}(R(X_i) - R(Y_i)))}{N(N^2 - 1)} \tag{3.2}$$

where $R(X_i)$ is the rank assigned to the X value and $R(Y_i)$ is the rank assigned to the Y value. We used the non-parametric rank correlation because we were uncertain of the distributions of sensitivities. Rank correlations behave similarly to regular correlations. A value of 1 indicates the two rankings agree perfectly, a value of -1 indicates the two rankings disagree perfectly, and a value of 0 indicates the two rankings are not related.

There are 120 output tasks on the net. For each pair of tasks we compute the degree of sharing using the rank correlation procedure described above. If the rank correlation is 0.0, then the two tasks do not agree (or disagree) about which hidden units are important and not important. The expected value of the rank correlation is 0.0 for random tasks if the hidden layer contains two or more hidden units. If the rank correlation is significantly greater than 0.0, this indicates the two tasks are sensitive to the same hidden units. The hidden units important to one task are important to the other. In other words, the two tasks share parts of the hidden layer. If the rank correlation is below 0.0, this indicates the hidden units that are important to one task are not important to the other, and vice-versa. This anti-correlation means the tasks use different parts of the hidden layer. There is less overlap than would be expected by chance.

There are 120 tasks, so there are 7140 pairs of tasks. We computed the degree of sharing for all 7140 pairs of tasks. Rather than try to show the raw correlations between pairs of tasks, we summarize the results by computing the average rank correlation for tasks that are related in different ways. For example, there are 360 pairs of tasks that do not have any features in common. One such pair is:

```
P001 = IF (A > 1/2) THEN B, ELSE C
P120 = IF (F > 1/2) THEN E, ELSE D
```

There are 3240 pairs of tasks that have exactly one feature in common. One such pair is:

```
P001 = IF (A > 1/2) THEN B, ELSE C
P063 = IF (D > 1/2) THEN A, ELSE E
```

Similarly, there are 3240 pairs of tasks that have two features in common. One such pair is:

```
P001 = IF (A > 1/2) THEN B, ELSE C
P002 = IF (A > 1/2) THEN B, ELSE D
```

Finally, there are 300 pairs of tasks that have all three features in common. One such pair is:

```
P001 = IF (A > 1/2) THEN B, ELSE C
P021 = IF (B > 1/2) THEN A, ELSE C
```

We compute the average degree of sharing for groups of tasks such as these. The expectation is that groups that contain more related tasks will have higher average degree of sharing than groups containing less related tasks.

Figure 3.7 shows the average degree of sharing between tasks as a function of how related they are using the groups defined above. In this graph, the data point at "0 features in common" compares how much tasks having no features in common share the hidden layer. The data points at "3 features in common" show the degree of hidden unit sharing between tasks that use exactly the same three features (though these features are not necessarily in the same places in the tasks). The line labelled "any_feature" disregards the position of the features in the tasks. Tasks that have one feature in common might or might not use that common feature the same way. The line labelled "test_must_match", however, requires that the feature in the conditional test be the same. Thus if two tasks have one feature in common, this feature must be the feature used in the conditional.

The general trend of both lines is that tasks share hidden units more if they are more related. The small negative correlation for tasks that do not share any variables suggests that a complete lack of relatedness between functions leads to anti-correlated sharing, i.e., outputs for unrelated functions tend to use different hidden units. Because tasks are most sensitive to only a few hidden units, and tend to be somewhat randomly sensitive to the

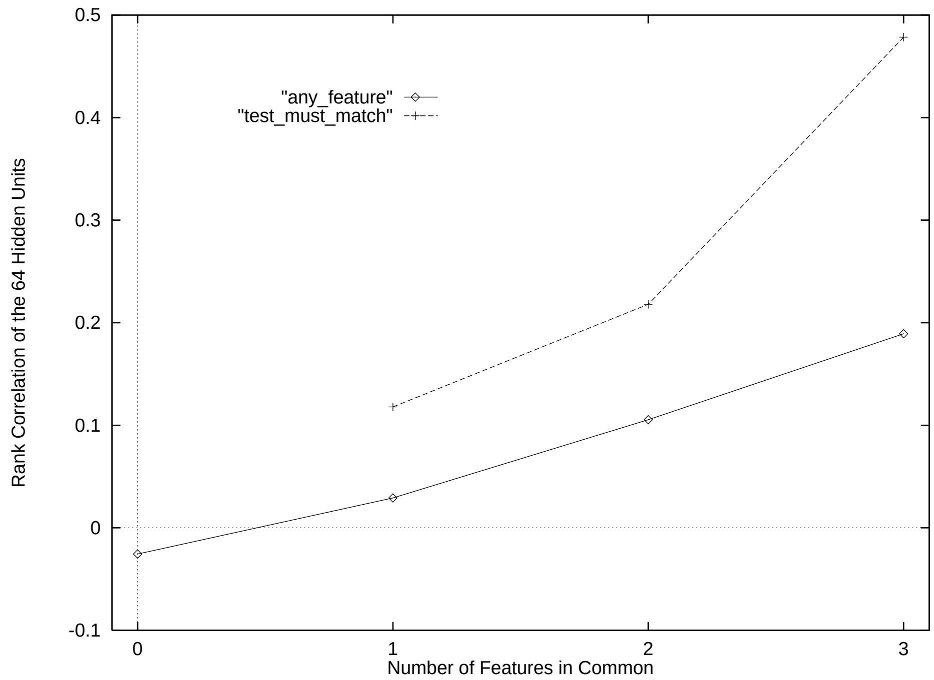


Figure 3.7: Sharing in the hidden layer as a function of the similarity between tasks. Tasks that are more related share more hidden units.

remaining hidden units, we do not expect to see strong negative rank correlations for unrelated tasks if the hidden layer is large. The negative correlation observed at 0, though small, is statistically significantly different from 0.0. (We don't show confidence intervals on the graph because most are too small to see.) Unless the net capacity is tightly constrained (something which usually hurts performance on all tasks), unrelated tasks tend to act more like randomly related tasks (correlations near zero) than anti-correlated tasks (negative correlations).

The correlations for the "test_must_match" line is higher than the correlations for the "any_feature" line. The only thing that makes these two lines different is that the "test_must_match" line contains only pairs of tasks that share the feature used in the conditional. This suggests that overlap in the conditional IF test is more important for hidden layer sharing than overlap in the THEN or ELSE part of the tasks.

The degree of sharing for the "test_must_match" line when the task relatedness score equals one tells us something interesting. In Section 3.2.1 we used two synthetic test functions to show that tasks that are uncorrelated can still be related in ways that backprop

MTL can benefit from. If only one feature in the pair of functions matches, and the test must match, then the only feature in common is the test. This data point represents pairs of functions like:

```
P001 = IF (A > 1/2) THEN B, ELSE C
P011 = IF (A > 1/2) THEN D, ELSE E
```

As was mentioned in Section 3.4.4, the distributions for all variables are the same. That means the distributions for "THEN B ELSE C" and "THEN D ELSE E" are the same. But B and C, and D and E are sampled randomly, so there is no correlation between them, and thus there is no correlation between "THEN B ELSE C" and "THEN D ELSE E". This means there is no correlation between task P001 and P011, nor between any other pair of tasks in this group. Yet, the tasks in this group are related because they all share the variable used in the conditional test. Not only will training these kinds of tasks together improve their performance, but Figure 3.7 shows that the degree of sharing between the tasks is substantial. Section 3.2.1 showed task correlation is not necessary for MTL benefit. Figure 3.7 shows that the degree of sharing between two tasks can be high despite them being uncorrelated.

Figure 3.8 shows the average degree of sharing for different groupings of the tasks than in Figure 3.7. There are two differences in this figure. First, there are seven groups in Figure 3.8 instead of the four groups in Figure 3.7. Tasks are placed in the first six groups, groups 0–5, by the following procedure: Initialize the group similarity score to 0. Add 1 for each feature common to both tasks, but in different places. Add 2 for each feature in the THEN or ELSE clause of the two tasks that is the same feature in the same place. Add 3 if the feature in the conditional IF part of the task is the same (because Figure 3.7 suggests sharing the conditional feature is more important than sharing either the THEN or ELSE features.). The maximum score a pair can achieve is 5. This requires the feature in the conditional IF test match, and one of the other features be the same feature in the same place.

If two tasks were identical, the maximum score they could achieve would be 7 (3 points for overlapping on the conditional, and 2 points each for overlapping on the THEN and

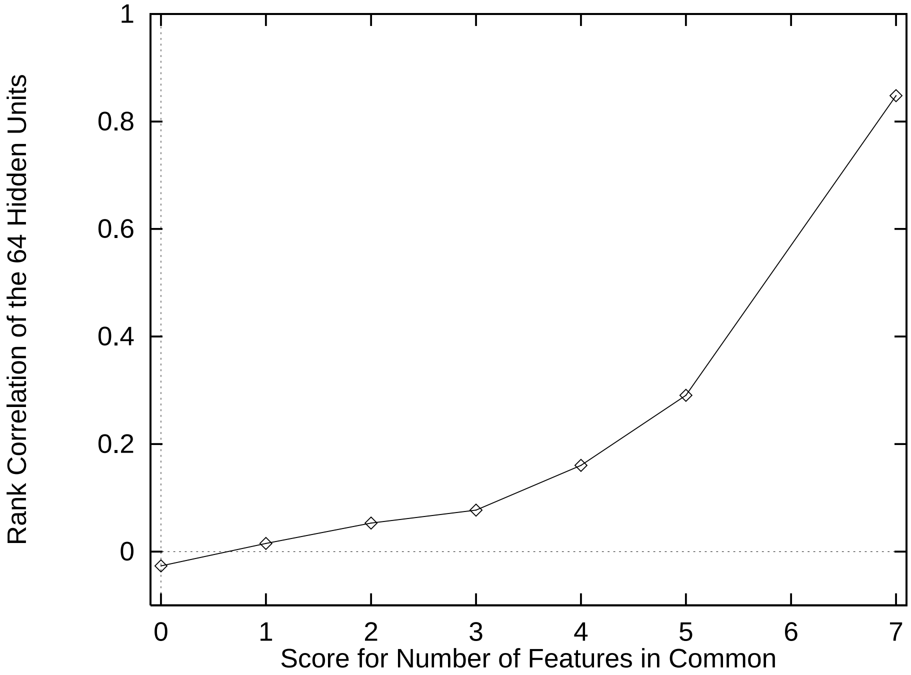


Figure 3.8: Sharing in the hidden layer as a function of the similarity score between tasks. See the text for how the score is computed. A score of 7 is for pairs of identical tasks trained on the same net.

ELSE features). Group 7 is the other difference between Figures 3.7 and 3.8. We trained an MTL net on 240 peaks tasks instead of 120 tasks as before by training two copies of each task on the net. Thus there are two outputs trained on task P001, two for P002, etc. The MTL net does not know which output tasks are copies of each other. In Figure 3.8, groups that have a score of 7 are the 120 pairs of identical tasks trained on different outputs.

There are other relationships between peaks functions we could examine. For example, we have looked at the degree of sharing for tasks that do not share the conditional IF feature, for tasks that have both the THEN and ELSE features in the same place, ... *For every relationship between peaks functions we examined, degree of sharing was positively correlated with hidden unit sharing.* This suggests that, for the peaks functions at least, backpropagation using a shared hidden layer is able to discover how tasks are related on hidden layer features without being given explicit training signals about task relatedness. Backprop MTL discovers the relationships between tasks via the constructive and destructive interference of the error gradients summed at the shared hidden layer.

One final note before leaving the peaks functions: In these experiments we restricted peaks functions to functions using each feature only once. We did not allow functions like:

```
P121 = IF (A > 1/2) THEN A, ELSE A
```

It would be interesting to run experiments that use all 216 possible peaks functions. Part of what makes this interesting is that functions like P121 above directly map coded inputs to their unencoded output values. This kind of function could serve as a strong *hint* to the net because it would help the net learn to decode the inputs without the complexity of having to learn a boolean function that combines three variables at the same time.[3]

3.6 Related Revisited

This thesis does not purport to propose a general theory or general mechanisms for all inductive transfer. The scope of this thesis is the inductive transfer performed by training related tasks in parallel while using a shared representation. In backprop nets, this shared representation is a hidden layer shared by the outputs. With other learning procedures, something else will be shared. With k-nearest neighbor, the distance function will be shared. With decision trees, the splits will be shared. For each of these MTL algorithms, we can devise a definition for relatedness. For multitask learning in backprop nets such a definition is:

Two tasks are related for backprop MTL if there is correlation (positive or negative) between the training signal for one task (or more correctly the backpropagated errors for that task's training signals) and what is learned in the hidden layer for the other task when they are trained together.

It is important to note that the fact that there is this correlation during learning does not necessarily mean MTL will benefit from it. If the algorithm is not good enough, some kinds of correlation may hurt performance instead of helping it. In theory, one would always expect such a correlation between a task and hidden layer representation to improve

[3]Note that the net would not necessarily learn a complete representation for tasks like P121 coding for an input at the hidden layer because the output for P121 also can use the weights from the hidden layer to the output (which are not shared with other tasks) to learn a model for P121.

performance; correlation suggests the extra signal provides additional information. But theory and practice do not always agree. Because we deal with imperfect learning procedures, there are going to be correlations between what is learned that, in theory, should help learning, but which in practice will hurt learning. Part of the problem is that we often deal with small data sets, and we have little or no apriori information about the nature of the relationships between tasks. This means the correlations must be discovered from the small data set, a process fraught with uncertainty and error.

3.7 Chapter Summary

Section 3.1 reviewed work showing that MTL helps learning because tasks are related. Section 3.2 discussed some basic properties of related tasks. For backprop MTL to benefit from related tasks, the tasks must be related by having overlapping sets of relevant input features and must be able to share hidden units. Tasks which are functions of different inputs cannot help each other via backprop MTL. Surprisingly, we were able to show that related tasks need not be correlated. Section 3.3 presented seven relationships between tasks that a backprop net trained on multiple tasks can exploit to learn the tasks better. These relationships are: statistical data amplification, sampling data amplification, blocking data amplification, attribute selection, eavesdropping, representation bias, and overfitting prevention. All the relationships can improve learning because of the constructive and destructive interference of the error gradients summed at the shared hidden layer. Because of this, it is easy for the mechanisms to act in concert.

Section 3.4 introduced the Peaks Functions, a set of 120 problems designed to serve as a test-bed for inductive transfer algorithms. One of the experiments with the peaks functions provides very solid evidence that some of the benefit of backprop MTL is due to the relationships between tasks. In Section 3.5 we "opened up" an MTL net trained on 120 peaks functions and showed that tasks that are more related share more in the hidden layer. Part of what makes this result so interesting is that backprop MTL is not given explicit training signals about how tasks are related. It discovers task relationships itself using a form of unsupervised learning that clusters tasks by the similarity of the hidden layer representation learned for them. This is a heretofore unrecognized and unstudied capability of backpropagation (that we plan to investigate further). Finally, Section 3.6 proposed a heuristic definition of relatedness. Two tasks are related (for inductive transfer) if there is correlation between what is learned for one task and the loss function applied to the training signals of the other task. The next chapter presents more than a dozen cases where real-world tasks are likely to satisfy this heuristic.

"Half the trick of finding clues is knowing that they're there."

– *Sherlock Holmes*

Chapter 4

When To Use It

Chapter 3 presented a number of mechanisms that allow backprop to benefit from different kinds of relationships between tasks and attempted to define relatedness. In practice, however, it is more important to be able to find useful related tasks than it is to be able to define precisely what a related task is. This chapter is designed to help us recognize related tasks in real-world problems.

4.1 Introduction

How often will training data for useful extra tasks be available? This chapter shows that many real world problems present opportunities for multitask learning. This is important—it doesn't matter how well multitask learning works if it won't be applicable to many problems in the real world. This chapter presents more than a dozen prototypical applications of multitask transfer where the training signals for related tasks are often available and can be leveraged. Each prototypical application is described and one or two concrete examples are described. In a few cases, empirical data for sample problems is also presented.

We believe most real-world problems fall into one or more of these domain types. This claim might sound surprising given that few of the test problems traditionally used in machine learning are multitask problems. We believe most of the problems traditionally used in machine learning have been so heavily preprocessed to fit STL that most opportunities for MTL were eliminated before learning was attempted.

This is one of the simplest chapters in this thesis. It is also one of the most important. This chapter shows that there are many opportunities for using MTL on real-world problems because potentially useful extra tasks are available in many different kinds of domains.

4.2 Using the Future to Predict the Present

Often valuable features become available after the predictions must be made. These features cannot be used as inputs because they will not be available at run time. If learning is done offline, however, they can be collected for the training set and used as extra MTL tasks. The predictions the learner makes for these extra tasks are ignored when the system is used. Their sole function is to provide extra information to the learner during training.

One application of learning from the future is medical risk evaluation. Consider the pneumonia risk problem used in Section 2.3. Each patient has been diagnosed with pneumonia and hospitalized. 65 measurements are available for most patients. These include 30 basic measurements acquired prior to hospitalization such as age, sex, and pulse, and 35 lab tests, such as blood counts and blood gases, made in the hospital. Some of the most useful tests for assessing risk are these lab tests that become available only after the patient is hospitalized.

Table 4.1 shows the improvement in performance that is obtained on this domain by using the future lab measurements as extra outputs as shown in Figure 4.1. (This is the same as Table 2.3 and the same figure as Figure 2.3.)

Table 4.1: STL and MTL Errors on Pneumonia Risk

FOP	0.1	0.2	0.3	0.4	0.5
STL	.0083	.0144	.0210	.0289	.0386
MTL	.0074	.0127	.0197	.0269	.0364
% Change	-10.8%	-11.8%	-6.2%	-6.9%	-5.7%

Future measurements are available in many *offline* learning problems because they can be added to the training set after the fact. As a very different example, a robot or autonomous vehicle can more accurately measure the size, location, and identity of objects later as it passes near them. For example, road stripes or the edge of the road can be de-

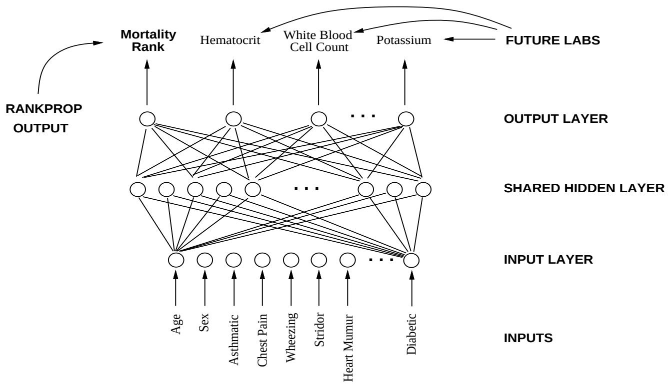


Figure 4.1: Using Future Lab Results as Extra Outputs To Bias Learning

tected reliably as a vehicle passes alongside them, but detecting them far ahead of a vehicle is hard. Since driving brings future road closer to the car, stripes and road borders can be measured accurately when passed and added to the training set. They can't be used as *inputs* because they will not be available in time while driving. As we have already seen with 1D-ALVINN, extra tasks of this kind can be used as extra MTL outputs to provide extra information to help learning without requiring they be available at run time. Using future measurements as extra output tasks will probably be one of the most frequent sources of extra tasks in real problems.

4.3 Multiple Metrics

Sometimes it is hard to capture everything that is important in one error metric or one output representation. When alternate metrics or representations capture different, but useful, aspects of a problem, MTL can be used to benefit from them.

An example of using MTL with different metrics is the pneumonia domain from Section 2.3. There we used an error metric called *rankprop* (see Appendix B) designed specifically for tasks where it is important to learn to *order* instances correctly. Rankprop outperforms backprop using traditional SSE by 20-40% on this problem. Rankprop, however, can have trouble learning to rank cases at such low risk that virtually all patients survive. Rankprop outperforms SSE on these low risk patients, but this is where it has the most difficulty learning a stable rank.

Interestingly, SSE is at its best when cases have high purity, as in regions of feature space where most cases have low risk (e.g., FOPs 0.1 or 0.2). SSE has the most difficulty in regions where similar cases have *different* outcomes. *SSE is at its best where rankprop is weakest.* Suppose we add an additional SSE output to a network learning to predict risk using rankprop?

Table 4.2: Adding an Extra SSE Task to Rankprop

FOP	0.1	0.2	0.3	0.4	0.5
w/o SSE	.0074	.0127	.0197	.0269	.0364
with SSE	.0066	.0116	.0188	.0272	.0371
% Change	-10.8%	-8.7%	-4.6%	+1.1%	+1.9%

Adding an extra SSE output has the expected effect. It lowers error at the rankprop output for the low risk FOPs, while slightly increasing error at the high risk FOPs. Table 4.2 shows the results with rankprop before and after adding the extra SSE output. Note that the extra output is completely ignored when predicting patient risk. It has been added solely because it provides a useful bias to the net during training. We have not examined if combining the extra output with the predictions of the rankprop output might yield further improvements. The earliest example of using multiple output representations we know of is [Weigend 1991] which uses both SSE and cross-entropy outputs for the same task.

4.4 Multiple Output Representations

Sometimes it is not apparent what output encoding to use for a problem. Alternate codings of the main task can be used as extra outputs the same way alternate error metrics were used above. For example, when using sigmoid output units with outputs on the interval [0,1], sometimes it is not clear if boolean task values should be represented as 0.0 and 1.0, or some other values such as 0.15, 0.85, or 0.25, 0.75, that do not force the sigmoid output units to their extreme values. If there is reason to believe that different sets of output values might yield different benefits, MTL can be used to achieve both benefits by using both output representations at the same time. We can train multiple outputs on the MTL net and code the boolean as 0.0/1.0 on one output, 0.15/0.85 on another, and 0.25/0.75 on a third. We can combine the multiple outputs by addition or voting, or can select whichever output appears to perform best. The reason why this can improve performance is that the output trained on the 0.0/1.0 representation is the one that learns to separate the classes most, but is also the output most driven to learn a nonlinear mapping. The 0.25/0.75 output does not learn to separate the classes as much, but may be less prone to learning unnecessarily nonlinear mappings. Learning models that strongly distinguish different classes is usually good. So is learning less nonlinear functions. MTL provides one way to bias a backprop net to try to accomplish both objectives. Similar reasoning can be applied to other cases such as the use of polar and cartesian output representations for problems involving spatial recognition. The radius or angle may more easily represent some regularities in the problem, but the x,y,z coordinates may more easily represent other regularities. The two coordinate systems are redundant. Either output representation should be sufficient. But learning might be improved by using both representations on one MTL net.

As another example, distributed output representations often help *parts* of a problem be learned because the parts have separate error gradients. But if prediction requires *all* outputs in the distributed representation to be correct, a non-distributed representation can be more accurate. MTL is one way to merge these conflicting requirements in one net. For example, consider the problem of learning to classify a face as one of twenty faces. One output representation for this problem is to have one output for each of the twenty persons the net is supposed to recognize. Another output representation is to train the net on a set of face features that are sufficient to classify the twenty faces. These might be features like beard/no_beard, mustache/no_mustache, glasses/no_glasses, long_hair/short_hair/bald, hair_color(blonde, red, white, brown, black), eye_color(blue, brown), male/female, etc. It is easy to imagine that a set of these features might be sufficient to correctly classify the twenty individuals. Correct classification, however, might require each feature be correctly predicted. Yet some of these features may be difficult to predict with high accuracy. The non-distributed output representation that uses one output for each individual may be more reliable on average. But training the recognition net to recognize specific traits should help training, too. MTL allows us to use both output representations for the problem, even if only one representation will be used for prediction.

When using redundant output representations, an interesting issue is what outputs to use to make the final prediction. Sometimes better accuracy might be achieved by combining the predictions from the redundant representations rather than choosing one of the representations. This issue is orthogonal to MTL whose principle goal is to provide a mechanism for improving the accuracy of one of the representations. Where similar accuracy is achieved with several redundant output representations, we suspect accuracy often would improve by combining their predictions. If it is possible, however, to optimize the performance of any one of these output representations at the expense of the other representations, better performance often will be achieved by using the better performing representation. If combining predictions is still is desirable, multiple independent MTL nets can be trained and their predictions combined, each net possibly being optimized to perform well on one representation at a time. (See Chapter **??** for discussion of how to optimize MTL nets for one task at a time.)

An interesting, related approach to using multiple alternate output encodings for the same problem is error correcting codes [Dietterich & Bakiri 1995]. In this approach, the multiple encodings for the outputs are designed so that predictions from the multiple outputs can be combined such that the combined prediction is less sensitive to occasional errors in some of the outputs. It is not clear at this time how much error correcting codes benefit from MTL-like mechanisms. In fact, ECOC methods may benefit more from being trained on STL nets (instead of MTL nets) so that different outputs do not share the same hidden layer and thus make less correlated predictions.

4.5 Time Series Prediction

Applications of this type are a subclass of using the future to predict the present, where future tasks are identical to the current task except that they occur at a later time. This is a large enough subclass to deserve special attention. Also, the additional knowledge that the future tasks are identical to the current task sometimes allows additional structure to be brought to bear on MTL approaches to these problems.

The simplest way to use MTL for time series prediction is to use a single net with multiple outputs, each output corresponding to the same task at a different time. If output k refers to the prediction for the time series task at time T_k, this net makes predictions for the same task at K different times. Often, good performance is obtained if the output used for prediction is the middle output (temporally) so that there are tasks earlier and later than it trained on the net.

We tested MTL on time sequence data in a robot domain where the goal is to predict future sensory states from the current sensed state and the planned action. We were interested in predicting the sonar readings and camera image that would be sensed N meters in the future given the current sonar and camera readings, for N between 1 and 8 meters. Figure 4.2 shows the MTL architecture for this problem. As the robot moves, it collects a stream of sense data. (Strictly speaking, this sense data is a time series only if the robot moves at constant speed. We used dead reckoning to determine the distance the robot traveled, so our data might be described as a spatial series.)

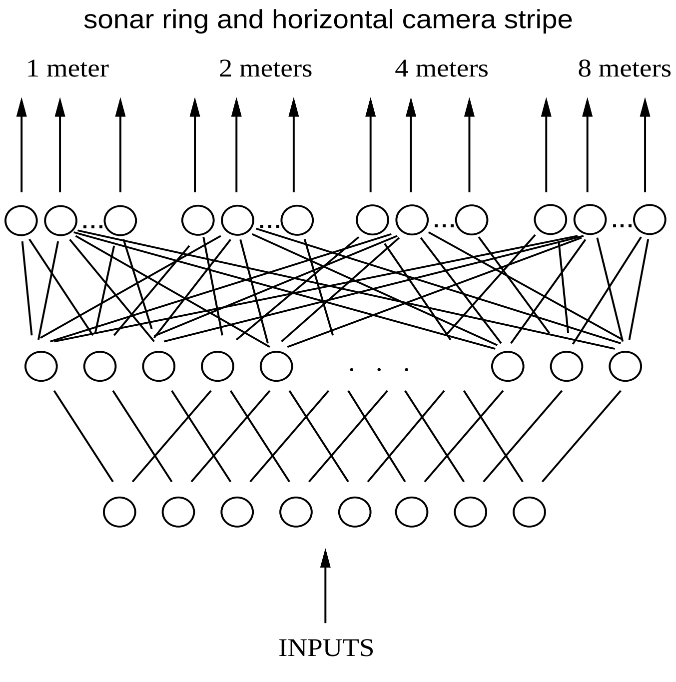


Figure 4.2: Predicting a temporal sequence of sense measurements with MTL

We used a backprop net with four *sets* of outputs. Each set predicts the sonar and camera image that will be sensed at a future distance. Output set 1 is the prediction for 1 meter, set 2 is for 2 meters, set 3 is for 4 meters, and set 4 for 8 meters. The performance of this net at each prediction distance is compared in Table 4.3 with STL nets learning to predict each distance separately. Each entry is the SSE averaged over all sense predictions. Error increases with distance, and MTL outperforms STL at all distances except 1 meter.

Table 4.3: STL and MTL SSE on Sensory Prediction

METERS	1	2	4	8
STL	.074	.098	.145	.183
MTL	.076	.094	.131	.165
% Change	+2.7%	-4.1%	-9.7% *	-10.9% *

The loss of accuracy at 1 meter is not statistically significant. We conjecture, however, that a pattern in this data may be common. That is, MTL may often help harder predictions most, possibly at the expense of easier predictions. This is not an insurmountable problem. Where this is known to be true, one can use STL for shorter term predictions, but use MTL for the harder longer term predictions. Note that one must still use the shorter term predictions in the MTL net that will be used for the longer term predictions, even though they may not be used, because they provide benefit for the longer term predictions. The goal in MTL is to use whatever extra tasks are available that might help the main task, even if performance on those extra tasks is not important, or is made worse, by MTL.

4.6 Using Non-Operational Features

Some features are impractical to use at run time. Either they are too expensive to compute, or they need human expertise that won't be around or would be too slow. Training sets, however, are often small, and we usually have the luxury to spend more time preparing them. Where it is practical to compute non-operational feature values for the training set, these may be used as extra MTL outputs.

A good example of this is in scene analysis where human expertise is often required to label important features. Usually the human will not be in the loop when the learned system is to be used. Does this mean that features labelled by humans cannot be used for learning? No. If the labels can be acquired for the training set, they can be used as extra tasks for the learner; features used as extra tasks will not be required later when the system is used.

An example is the 1D-DOORS domain from Section 2.2. There we used a mouse to define several features in images of doorways collected from a robot-mounted camera. The main tasks were the horizontal location of the doorknob and the doorway center. The extra features were created just to give the backprop net more information. But because a human had to manually define these features, they cannot be used as inputs. The human will not be there to define these features when the robot is operating autonomously. Because a human had to process each image to define the training signals for the doorknob location and doorway center main tasks, it was easy to collect the additional features at the same time for the training set. The following extra tasks were employed:

- horizontal location of doorknob
- horizontal location of doorway center
- horizontal location of left door jamb
- width of left door jamb
- horizontal location of left edge of door
- single or double door
- width of doorway
- horizontal location of right door jamb
- width of right door jamb
- horizontal location of right edge of door

These extra tasks helped MTL learn the main tasks 20–30%. See Section 2.2 for more information about this domain.

There are many domains where human-labelled data is available for the training sets, but will not be available when the trained system is used. Examples include hand-labelled images, hand-labelled text data, hand-labelled medical data, hand-labelled acoustic or speech data, etc. Whenever there are hand-labelled features that are not themselves the main focus of learning, these may be used as extra output tasks that potentially will benefit the tasks that are the focus of learning.

4.7 Using Extra Tasks to Focus Attention

Learning often learns to use large, ubiquitous patterns in the inputs, while ignoring small or less common inputs that might also be useful. MTL can be used to coerce the learner to attend to patterns in the input it would otherwise ignore. This is done by forcing it to learn internal representations to support related tasks that depend on such patterns.

A good example of this is road following. Here, STL nets often ignore lane markings when learning to steer because lane markings are usually a small part of the image, are constantly changing, and are often difficult to see (even for humans) because of poor lighting and wear. If a net learning to steer is also required to learn to recognize road stripes, the net will learn to attend to those parts of the image where stripes occur. To the extent that the stripe tasks are learnable, the net will develop internal representations to support them. Since the net is also learning to steer using the same hidden layer, the steering task can use the parts of the stripe hidden representation that are useful for steering.

In Section 2.1 we used a road image simulator developed by Pomerleau to generate synthetic road images. The main tasks were to predict steering direction and the location of the center of the road. The following extra tasks related to the centerline were used for MTL:

- does the road have 1 or 2 lanes?
- horizontal-location of centerline 10 meters ahead (if present)
- intensity of the centerline

The performance benefit of MTL using these extra tasks is shown in Section 2.1. It is not easy to analyze a net trained on 1D-ALVINN to see what it has learned about centerlines. Instead, we ran an experiment using the STL and MTL nets trained for steering on 1D-ALVINN to test how important centerstripes in the images are to the STL and MTL nets. Because the data is generated with a simulator, we were able to eliminate the stripes from the generated road images in the test set (the training data still contains centerstripes). The centerstripes are replaced in the image with the same grey levels as the surrounding road. If MTL learned more about centerstripes than STL, and uses what it learned about centerstripes for the main steering task, we expect to see steering performance degrade more for MTL than for STL when we remove the centerstripes from the images.

On images with removed centerstripes, error increased by a factor of 2.0 for the STL nets, whereas error increased by a factor of 3.1 for the MTL nets. MTL's performance on the main steering task is more sensitive to the presence of centerstripes in the image, presumably because many of the extra tasks trained on the MTL net coerce the net to learn about centerstripes.

4.8 Tasks Hand-Crafted by a Domain Expert

Experts excel at *applying* their expertise, but are poor at codifying it. Most learning algorithms are poor at incorporating unstructured advice from experts, but are good at learning from examples. MTL is one way to collect domain-specific inductive bias from an expert and give it to the learning procedure by capitalizing on the strengths of each. Having domain experts define "helper" tasks is a convenient way to use human expertise to bias learning. Using extra outputs or extra error terms applied to the existing outputs to provide hints to ANNs is well documented in [Abu-Mostafa 1989]. One example of hints is to use extra tasks to help a net learn desired properties such as monotonicity or symmetry. For example, one might want the relationship between household income and the maximum size loan that can safely be awarded to that household to rise monotonically with income, all other things being equal. Hints provide one way of biasing the net to learn models satisfying this condition. Although this can be done by constructing carefully designed extra tasks, it is most easily accomplished by applying additional error terms that penalize nonmonotonicity on the output for the main task.

4.9 Handling *Other* Categories in Classification

Consider the problem of digit recognition. This is a classification problem where the goal is to label input images with the classes 0–9. In real-world applications of digit recognition, it is common that some of the images that will be given to the classifier will not be digits. For example, sometimes images containing alphabetic characters or punctuation marks will be given to the classifier. We do not want the classifier to accidentally classify a "t" as the digit one or seven. One common way to help prevent this is to create an additional category called "other" which is the correct classification for all images that do not contain digits.

MTL provides a better way to do this. Consider the large variety of characters that are to be mapped to this one "other" class. Because of this diversity, and the need of the classifier to not allow images of any of the legal digits 0–9 to be confused with this class, learning this class will be very difficult. By throwing so many different images together into one class, we have made learning that class potentially very difficult. A better approach is to split the "other" class into separate classes for the individual characters that are trained in parallel with the main digit tasks. By breaking-out the separate other tasks into separate MTL tasks, the net has a better chance of learning to discriminate digits from non-digits. [LeCun, 1997 (private communication)]

4.10 Sequential Transfer

MTL is parallel transfer. It might seem that sequential transfer [Pratt & Mostow 1991; Pratt 1992; Sharkey & Sharkey 1992; Thrun & Mitchell 1994; Thrun 1995] should be easier. This may not be the case. Some of the advantages of parallel transfer are:

- The full detail of what is being learned for all tasks is available to all tasks because all tasks are learned at the same time.

- In many applications extra tasks are available when the main task is to be learned. Parallel transfer does not require one to choose a training sequence—the order in which tasks are trained usually has significant impact in serial transfer.

- Tasks often benefit each other mutually, something a linear sequence cannot capture. If task 1 is learned before task 2, task 2 can't help task 1. This not only reduces performance on task 1, but it can also reduce task 1's ability to help task 2.

- Sequential transfer does not benefit prior learned tasks unless they are re-trained, and this is probably best done through a parallel method.

- In Chapter 6 we will see that it can be important to optimize MTL technique to favor performance on the main task at the expense of worse performance on the extra tasks. It is difficult to perform this optimization if the extra tasks have already been learned and are now fixed.

Often, however, tasks do arise serially and it is not prudent to wait for all the tasks to begin learning. In these cases it is straightforward (though not necessarily computationally efficient) to use parallel transfer to do sequential transfer. If the training data can be stored, perform MTL using whatever tasks have become available, re-learning as new tasks and/or new data arise. If training data cannot be stored, or if we already have models learned from previous data that is no longer available, synthetic data can be generated from models that have already been learned and used as extra training signals. This approach to sequential transfer avoids the serious problem of catastrophic interference (forgetting old tasks while learning new ones). Moreover, it is applicable even where the analytical methods of evaluating domain theories used by other serial transfer methods [Pratt 1992; Thrun & Mitchell 1994] are not available. For example, the domain theory need not be differentiable or inspectable. It merely needs to be able to do prediction.

We've only tested this synthetic data approach on synthetic problems. Its performance is indistinguishable from having the original training data if the prior models were learned accurately. Interestingly, the performance often does not degrade that rapidly as the prior learned models become less accurate because MTL nets are less affected by noise in extra outputs than an STL net would be to that same amount of noise in its inputs. (See Section 5.1.2 for a more thorough discussion of this difference.)

One issue that arises when synthesizing data from prior models is what distribution to sample from. We used the distribution of the training patterns for the *current* task. We pass the input features for current training patterns through the prior learned models and use the predictions of those models as extra MTL outputs when learning the new main task. This sampling may not always be satisfactory. If the learned models are complex (suggesting a large sample would be needed to represent them with high fidelity), but the new sample of training data is small, it is beneficial to sample the prior model at more points than the current sample. See [Craven & Shavlik 1994] for a thorough discussion of synthetic sampling.

It is interesting to note that it is relatively straightforward to use parallel transfer to do serial transfer, but it does not seem to be easy to use serial transfer to do parallel transfer. It is also important to note that it is possible to combine serial and parallel transfer to get some of the benefits of each. (O'Sullivan and Mitchell are currently doing research on methods that combine MTL and EBNN for life-long learning in robots.)

4.11 Multiple Tasks Arise Naturally

Often the world gives us *sets* of related tasks to learn. The traditional approach is to separate these into independent problems trained in isolation. This is counterproductive—related tasks can benefit each other if trained together. An early, almost accidental, use of multitask transfer in ANNs is NETtalk [Sejnowski & Rosenberg 1986]. NETtalk learns the phonemes and stresses to give a speech synthesizer to pronounce the words given it as inputs. NETtalk used one net with many outputs, partly because the goal was to control a synthesizer that needed both phonemes and stresses at the same time. Although they never analyzed the contribution of multitask transfer to NETtalk, there is evidence that NETtalk is harder to learn using separate nets [Dietterich, Hild & Bakiri 1990; 1995].

A recent example of multiple tasks arising naturally is Mitchell's Calendar Apprentice System (CAP) [Dent 1992; Mitchell et al. 1994]. In CAP, the goal is to learn to predict the *Location*, *Time_Of_Day*, *Day_Of_Week*, and *Duration* of the meetings it schedules. These tasks are functions of the same data and share many common features. Early results using MTL decision trees on this domain suggest that MTL outperforms STL 2%–10%.

4.12 Similar Tasks With Different Data Distributions

Often there are many instances of virtually the same problem, but the distribution of instances from which the data are sampled differ for each instantiation. For example, most hospitals diagnose and treat pneumonia patients, but the demographics of the patients each hospital serves may be different. Hospitals in Florida may see older patient populations, urban hospitals may see poorer patient populations that have had less access to health care, hospitals in San Francisco may see more AIDS patients, rural hospitals may see fewer AIDS patients, etc.

Predictive models learned for one hospital will not be as accurate for another hospital as models learned for that other hospital. In medicine, however, often there are few cases of some ailments at many hospitals. It may not be possible to collect a large enough training sample for each hospital. A rural hospital might see fewer than 100 cases of pneumonia each year, and in any one year none of these cases may be AIDS-related. Yet next year that hospital might see three AIDS-related cases. Clearly pneumonia prediction in a hospital in California is a strongly related problem to pneumonia prediction in a hospital in Wyoming. But it is probably suboptimal to pool data from all hospitals and learn one model to make predictions for all hospitals. MTL provides one solution to this problem.

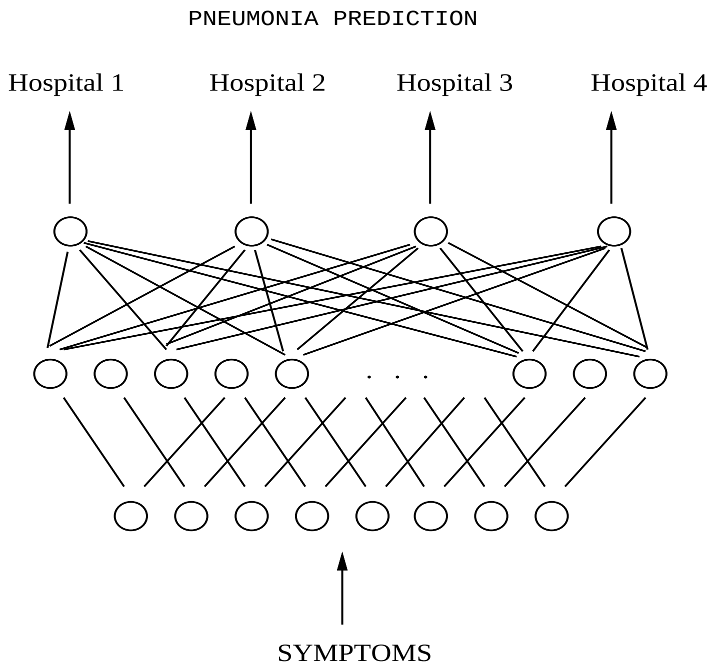


Figure 4.3: Predicting the same problem for different hospitals with MTL

Consider the MTL-backprop net shown in Figure 4.3. The net takes patient histories, symptoms, and lab tests as inputs. It has four outputs. Each output predicts the same medical condition (e.g., pneumonia risk), but for a different hospital. There may be 10 training cases for Hospital 1, 100 cases for Hospital 2, 1000 cases for Hospital 3, and 2,000 cases for Hospital 4. Note that each patient is a training case for only one hospital. That means we only have a target value for one output for each input vector. When backpropagation is done on this MTL net, errors are backpropagated only through the output that has a target value for that input vector.

Because the outputs share a hidden layer, however, the representation learned for each hospital's prediction model is available to be used by other hospital models. Thus Hospitals 1 and 2 can benefit from the larger pneumonia populations seen by Hospitals 3 and 4.

Nevertheless, the model learned for Hospital 1 will not (necessarily) be the same as the models learned for Hospitals 2, 3, or 4. Hospital 1 can benefit from what was learned for the other hospital models, but it is not constrained to be the same model. Each output (i.e., each hospital model) has different hidden-to-output weights trained only on the patient population at that hospital. Although these tasks are, presumably, very strongly related, some features may develop in the hidden layer that are useful to only some of the hospitals.

There are many domains where there are multiple instances of strongly related problems, but where each instance of the problem is different enough to make pooling the data inappropriate. Yet collecting sufficient data to learn models for each instance of the problem may be impractical. MTL provides one approach to sharing the data collected for these problems without committing to the very strong sharing that results from pooling the data. Other domains of this type include:

- learning to steer different types of cars, or cars with different types of tires, or cars driven on different types of roads or in different countries.
- learning to control multiple manufacturing and process control lines in manufacturing, where each line is composed of similar equipment and processes similar jobs.

4.13 Learning from Quantized or Noisy Data

Suppose the main task we wish to predict is a variable that has been heavily quantized and/or polluted with noise. Quantization is any process that takes a variable with N distinct values and re-represents it with $M < N$ values. For example, temperature is a continuous variable, but some process might quantize this continuous variable to a few discrete values such as cold, warm, and hot. Quantization of this type is common when human judgment is part of the measurement process.

Quantization can make it more difficult to train a model to predict the quantized variable from other measurements of the system. This is because quantization represents some instances that are similar as different, and classifies other instances that are different as similar. For example, temperatures of 100 degrees and 200 degrees might both be classified as warm, but 201 degrees might be classified as hot. This abrupt change in the quantized function makes learning hard by requiring the model to be relatively flat from 100 to 200 degrees, but also to make a sharp transition between 200 and 201 degrees. Quantization does not map similarity in input space to similarity in output space. If there is another less quantized task that correlates with the unquantized variable, training it as an extra task on the MTL net can help the main quantized task.

As a different example, consider a stochastic process that converts a probability to an outcome. In medicine, for example, we rarely know the probability of an adverse outcome for patients. We can, however, collect a training set consisting of patients for which we know if the adverse outcome happened or not. We still do not know the original probability of the outcome for the patients, only the outcome. A system learning to predict outcomes for new patients can have great difficulty if the probability of a positive outcome is not much higher for the high risk patients than it is for the low risk patients. For example, suppose high risk patients have a positive outcome probability of 0.10 and low risk patients have an outcome probability of 0.05. High risk patients have twice the probability of a positive outcome as the low risk patients. In a large training set, however, 1/3 of the patients with positive outcomes will actually be low risk cases, and the vast majority of cases will have negative outcome. Learning to distinguish high risk from low risk cases with 33.3% class noise will be difficult for much the same reason that quantization makes learning difficult. Patients with virtually identical symptoms may have different outcomes, and patients with very dissimilar symptoms may have the same outcomes. Similarity in input space is not well mapped to similarity in output space by the random process. As with quantization, if there are other tasks less disrupted by random sampling, using these as extra outputs on the MTL net can aid learning of the main task. We observe this in the pneumonia domain. There, a patient's probability of death (i.e., risk) is unknown to us. All we know is if the patient lived or died. But other tasks, such as hospital duration, admission to the ICU, and doctor's assessment of patient risk along a three point scale (low, medium, high) can be used to help guide an MTL net to assess risk better. Hospital stay is a particularly interesting extra task because long stays are indicators of higher risk, but short stays may be due to very low risk causing the patient to be discharged or very high risk causing the patient to die. The relationship between tasks that help each other may be complex.

4.14 Learning With Hierarchical Data

Many classification domains admit a structuring of the classes into a semantic hierarchy. For example, the classification of living things to species is usually treated as a hierarchical classification problem that descends a hierarchy of classes and subclasses. But there are other examples. Motorized vehicles can be classified via a hierarchy. Documents in a library are usually classified hierarchically. And recently considerable manual effort has gone into hierarchically classifying documents on the web into the Yahoo! hierarchy.

Surprisingly, most applications of machine learning to data that can be hierarchically classified make little use of the hierarchical information. MTL provides one way of exploiting hierarchical information. When training a classifier to make class distinctions at one particular point in the hierarchy, include as extra tasks all classification tasks that arise for ancestors and descendants of the current classification task. For example, when training a backprop net to distinguish between student and faculty web pages in a department, use extra tasks on the MTL net that require the net to distinguish between associate professor, full professor, research faculty (descendant tasks), extra tasks that require the net to distinguish undergraduate from graduate students (also descendant tasks), and extra tasks that require the net to distinguish between different departments in the university or between academic and other institutions (both ancestor tasks). One way to accomplish this is to train one MTL net to predict all class distinctions in the total hierarchy.

4.15 Outputs Can Beat Inputs

Most of the sections in this chapter present domains where it is impractical to use some features as *inputs*, because they will not be available in time, are too expensive to compute, or require human expertise. In these cases, MTL provides a way of benefiting from these features (instead of ignoring them) by using them as extra tasks. Some domains contain features that can be used as inputs, but which are more useful when used as extra outputs instead. In some cases this is because the features are actually harmful when used as inputs, but helpful when used as outputs. In other cases this is because the feature is useful as an input, but *more* useful as an extra output. This application of MTL is so surprising, and potentially so ubiquitous, that we dedicate the next chapter to it.

4.16 Chapter Summary

This chapter shows that there are many domains where potentially useful extra tasks will be available. One of the contributions of this thesis is to show that there are many different kinds of extra tasks to be used with MTL. The list of prototypical domains provided in this chapter is not complete. We are confident more types of extra tasks will be identified in the future.

Chapter 5

Some Inputs Work Better as Extra Outputs

The previous chapter presented a number of different domain types where training signals for extra tasks are available. This chapter presents a source of extra tasks that may be more ubiquitous. In this chapter we show that sometimes it is beneficial to move features that would normally be used as inputs from the input side of a backprop net to the output side of the net and use them instead as extra tasks for MTL.[1]

In supervised learning there is usually a clear distinction between inputs and outputs—inputs are what you will measure, outputs are what you will predict from those measurements. MTL blurs this distinction; some features are more useful as extra *outputs* than as *inputs*. By using a feature as an output we get more than just the case values, but can learn a mapping from the other inputs to that feature. Although we no longer have access to the case values when doing prediction, we do have access to the mapping learned by the MTL net from the case values in the training set. For some features, this mapping may be more useful than the feature value itself.

In Section 5.1 we present two regression problems and one classification problem where performance improves if features that could have been used as inputs are used as extra outputs instead. This section uses synthetic problems carefully constructed to show this

[1]The material presented in this chapter is joint work with Virginia de Sa.

effect and make it clear why the features work better as extra outputs than as inputs.

The problems in Section 5.1 are synthetic and were carefully designed to demonstrate that some features can be more useful as extra outputs than as input. Do real-world problems also have features that would be more useful as extra outputs? If they do, how would we determine which features are more useful as extra outputs? In Section 5.2 we demonstrate that there are features in real-world problems that are better used as extra output tasks. In this section we use feature selection to determine which features should be used as inputs, and treat the remaining features not used as inputs as candidates for use as extra output tasks.

The results in Sections 5.1 and 5.2 make it clear that some features help learning when used as input features, and also help learning if used as extra outputs, instead. Is it possible to use some features as both inputs and as extra outputs at the same time and accrue both benefits? In Section 5.3 we present an MTL architecture that is able to achieve some of the benefits of features used as inputs and used as extra outputs on one net. We demonstrate this approach using two of the synthetic problems from Section 5.1.

5.1 Promoting Poor Features to Supervisors

The goal in supervised learning is to learn functions that map inputs to outputs with high predictive accuracy. The standard practice in neural nets is to use all features that will be available for the test cases as inputs, and use as outputs only the features to be predicted.

MTL shows that using instance attributes as outputs can be very useful. Since any input could be used as an output, would some inputs be more useful as outputs? Surprisingly, yes. Sometimes it is *more* effective to give information to a backprop net through an *output* than through an input. Some features are less useful (or even harmful) when used as inputs than when used as extra outputs. This demonstrates that the benefits of outputs are different from the benefits of inputs.

This section presents three synthetic problems where it is better to use some features as extra outputs than as inputs. The basic approach in our synthetic problems is to move features to the output side of an MTL net that are either too noisy, or too poorly correlated with the main task, to be helpful when used as inputs. As outputs, they bias the learning of the input-to-hidden layer weights. This leads the shared hidden layer to develop more useful features, thus improving performance on the main task. All the problems are simple functions of 1 or 2 input variables. These input variables are encoded, however, to make the task more challenging. The extra features are chosen so that if the inputs were properly decoded there would be little or no benefit from the extra features as inputs. The information in the extra features is redundant with that in the regular inputs. The extra features, however, are not coded like the regular input variables. This makes it potentially easier for learning to use the extra features than to use the encoded regular inputs. If the extra features are noisy, however, there is a tradeoff between ease of learning from the uncoded extra features and the problems created by the noise in those features.

Section 5.1.1 uses the regression problem from Section 3.2.1 where, because a feature has no correlation with the main task, it is not useful as an input. But, because the underlying subfeatures it depends on are the same as the main task, it is useful if used as an extra output. Section 5.1.2 presents a similar regression problem where there are features useful as inputs if their noise is low, but which become harmful as inputs if their noise increases. However, because noise has different effects on inputs and outputs, these features remain useful as extra outputs even when the noise makes them harmful as inputs. Section 5.1.3 presents a binary classification problem where the information in an extra input feature is unnecessary because the optimal class boundary is provably independent of this feature. Because the extra feature correlates with the other input features the class boundary is defined on, however, it helps classification. By adjusting the problem in a well defined way, we are able to create variations where the benefit of the extra feature is greatest when used as an extra output instead of as an extra input. Section 5.1.4 discusses issues common to all three problems.

In this chapter we use the following terms: The Main Task is the output to be learned. The goal is to improve performance on the Main Task. Regular Inputs are the features provided as inputs in all experiments. The Regular inputs are always used as inputs, never as outputs. Extra Inputs are the extra features when they are used as inputs. Extra Outputs are the same extra features, but when used as extra outputs.

5.1.1 Poorly Correlated Features

This section presents a simple synthetic problem where it is easy to see why using a feature as an extra output is better than using that same feature as an extra input. This is the same problem used in Section 3.2.1 to show that tasks do not need to be correlated to be useful for MTL.

Consider the following function:

$F1(A, B) = SIGMOID(A + B)$

where $SIGMOID(x) = 1/(1 + e^{(-x)})$

The STL net in Figure 5.1 has 20 inputs, 16 hidden units, and one output. We use backpropagation on this net to learn $F1(A, B)$. Data is generated by uniform random sampling of A and B from the interval $[-5, 5]$. The inputs to the network are a binary coding for A and B. The range $[-5, 5]$ is discretized into 2^{10} bins and the binary code of the resulting bin number is used as the input coding. The first 10 input units receive the code for A and the second 10 receive the code for B. The target output is the unary real (unencoded) value $F1(A, B)$.

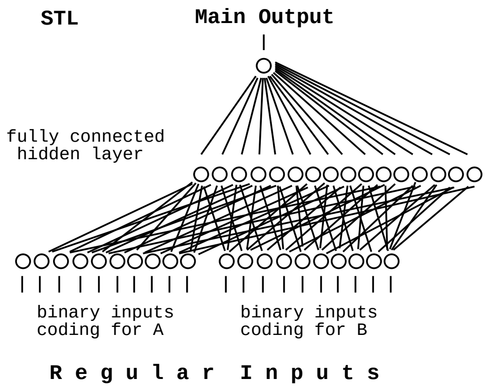


Figure 5.1: STL architecture for learning F1 from the regular inputs.

Backpropagation is done with per-epoch updating and early stopping. Each trial uses new random training, halt, and test sets. Training sets contain 50 patterns. This is enough data to get good performance, but not so much that there is not room for improvement. We use large halt and test sets—1000 cases each—to minimize the effect of sampling error in the measured performances. Halt and test sets containing 5000 cases each yield similar results.

Table 5.1 shows the mean performance of 50 trials of STL with the regular inputs when trained with backpropagation and early stopping.

Table 5.1: Mean Test Set Root-Mean-Squared-Error on F1

Network	Trials	Mean RMSE	Significance
STL	50	0.0648	-
STL+IN	50	0.0647	ns
MTL+OUT	50	0.0631	0.013*

Now consider a similar function:

$F2(A,B) = SIGMOID(A - B)$

Suppose, in addition to the 10-bit codings for A and B, you are given the unencoded unary value $F2(A,B)$ as an extra input feature. Will this extra input help you learn $F1(A,B)$ better? Probably not. $A + B$ and $A - B$ do not correlate for random A and B. The correlation coefficient for our training sets is typically less than ± 0.01. Because of this, knowing the value of $F2(A,B)$ does not tell you much about the target value $F1(A,B)$ (and vice-versa).[2] Figure 5.2 shows a scatter plot of $F1(A,B)$ vs. $F2(A,B)$ for a large sample of randomly generated values for A and B.

$F1(A,B)$'s poor correlation with $F2(A,B)$ hurts backprop's ability to learn to use $F2(A,B)$ to predict $F1(A,B)$. The STL net shown in Figure 5.3 has 21 inputs—20 for the binary codes for A and B, and an extra input for $F2(A,B)$. The second line in Table 5.1 shows the performance of STL+IN for the same training, halting, and test sets used by STL; the only difference is that there is an extra input feature in the data sets for STL+IN. Note that the performance of STL+IN is not significantly different from that of STL—the extra information contained in the feature $F2(A,B)$ does not help backpropagation learn $F1(A,B)$ *when used as an extra input.*

If $F2(A,B)$ does not help backpropagation learn $F1(A,B)$ when used as an input, should we ignore it altogether? No. $F1(A,B)$ and $F2(A,B)$ are strongly related. They

[2] Note that $A+B$ does correlate with $|A-B|$, just not with $A-B$ itself. Knowing $F2(A,B)$ does provide *information* about $F1(A,B)$. Unfortunately, backprop usually needs correlation between inputs and the current output errors to benefit from the inputs.

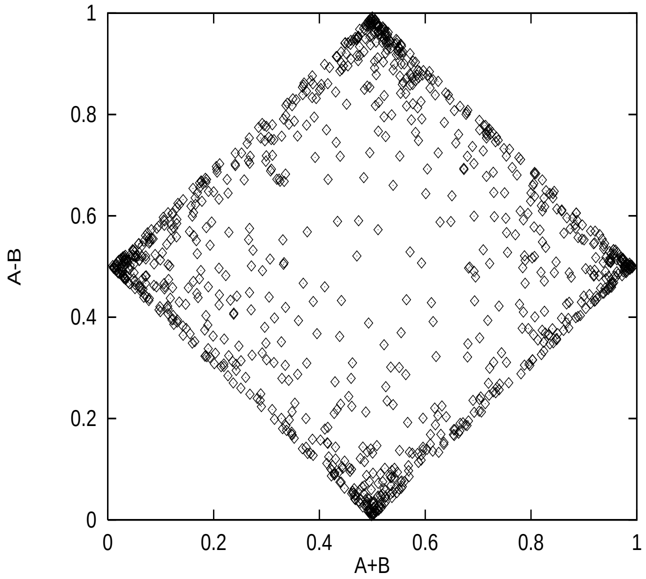


Figure 5.2: Scatter plot of F1(A,B) vs. F2(A,B) shows no correlation between them.

both benefit from decoding the binary input encoding to compute the subfeatures A and B. If, instead of using $F2(A, B)$ as an extra input, it is used as an extra output trained with backpropagation, it will bias the shared hidden layer to learn A and B better, and this will help the net learn to predict $F1(A, B)$ better.

Figure 5.4 shows a net with 20 inputs for A and B, and 2 outputs, one for $F1(A, B)$ and one for $F2(A, B)$. Error is back-propagated from both outputs, but the performance of this net is evaluated only on the output $F1(A, B)$ and early stopping is done using only the performance of this output. The third line in Table 5.1 shows the mean performance

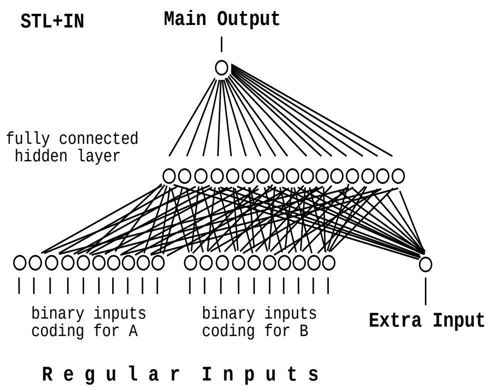


Figure 5.3: STL architecture for learning F1 from the regular inputs plus the extra input.

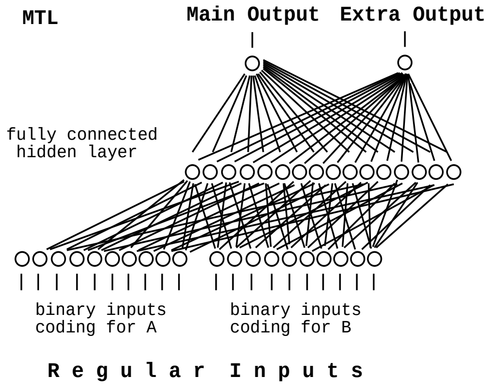


Figure 5.4: MTL architecture for learning F1 and the extra task from the regular inputs.

of 50 trials of this MTL net on $F1(A,B)$. Using $F2(A,B)$ as an extra output improves performance on $F1(A,B)$. Using the extra feature as an extra output is better than using it as an extra input. *By using $F2(A,B)$ as an output we make use of more than just the individual output values $F2(A,B)$, we learn to extract information about the function mapping the inputs to $F2(A,B)$. This is a key difference between using features as inputs and outputs.*

Why does $F2$ help $F1$ when used as an extra output? Because $F1$ and $F2$ both would benefit from computing the same features of the inputs at the hidden layer, the gradients backpropagated from the two outputs constructively reinforce in directions in the weight space that lead to the shared features. The extra information contained in the training signal for $F2$ biases the shared network representation towards features useful to $F1$.

5.1.2 Noisy Features

This section presents two problems where extra features are more useful as inputs if they have low noise, but which become more useful as outputs as their noise increases. Because the extra features are ideal features for these problems, this demonstrates that what we observed in the previous section does not depend on the extra features being contrived so that their correlation with the main task is low—features with high correlation to the main task training signals can still be more useful as outputs.

Problem 1

Once again, consider the main task from the previous section:

$F1(A, B) = SIGMOID(A + B)$

Now consider these extra features:

$EF(A) = A + NOISE\_SCALE * Noise1$

$EF(B) = B + NOISE\_SCALE * Noise2$

where the features $EF(A)$ and $EF(B)$ are not encoded and where $Noise1$ and $Noise2$ are uniformly sampled on $[-1, 1]$. If $NOISE\_SCALE$ is not too large, $EF(A)$ and $EF(B)$ are excellent input features for learning $F1(A, B)$ because the net can avoid learning to decode the binary input representations coding for A and B, and instead needs only to learn to add the new inputs $EF(A)$ and $EF(B)$. However, as $NOISE\_SCALE$ increases, $EF(A)$ and $EF(B)$ become less useful, and it is better for the net to learn $F1(A, B)$ from the binary inputs for A and B.

As before, we try using the extra features as either extra inputs or as extra outputs. Again, the training sets have 50 patterns, and the halt and test sets have 1000 patterns. We ran preliminary tests to find the best net size. The results showed 256 hidden units to be about optimal for the STL nets with early stopping on this problem.

Figure 5.5 plots the average performance of 50 trials of STL with the extra inputs and MTL with the same features used as extra outputs as $NOISE\_SCALE$ varies from 0.0 to 10.0. The performance of STL with the regular inputs, which does not use $EF(A)$ and $EF(B)$, is shown as a horizontal line; it is independent of $NOISE\_SCALE$. Let's first examine the results of STL using $EF(A)$ and $EF(B)$ as extra inputs. As expected, when the noise is small, using $EF(A)$ and $EF(B)$ as extra inputs improves performance considerably. As the noise increases, however, this improvement decreases. Eventually there is so much noise in $EF(A)$ and $EF(B)$ that they no longer help the net if used as inputs. And, if the noise increases further, using $EF(A)$ and $EF(B)$ as extra inputs actually hurts performance. Finally, as the noise gets very large, performance asymptotes back towards the performance obtained without the extra features.

Using $EF(A)$ and $EF(B)$ as extra outputs yields quite different results. When the noise is low, they do not help as much as they did as extra inputs. As the noise increases,

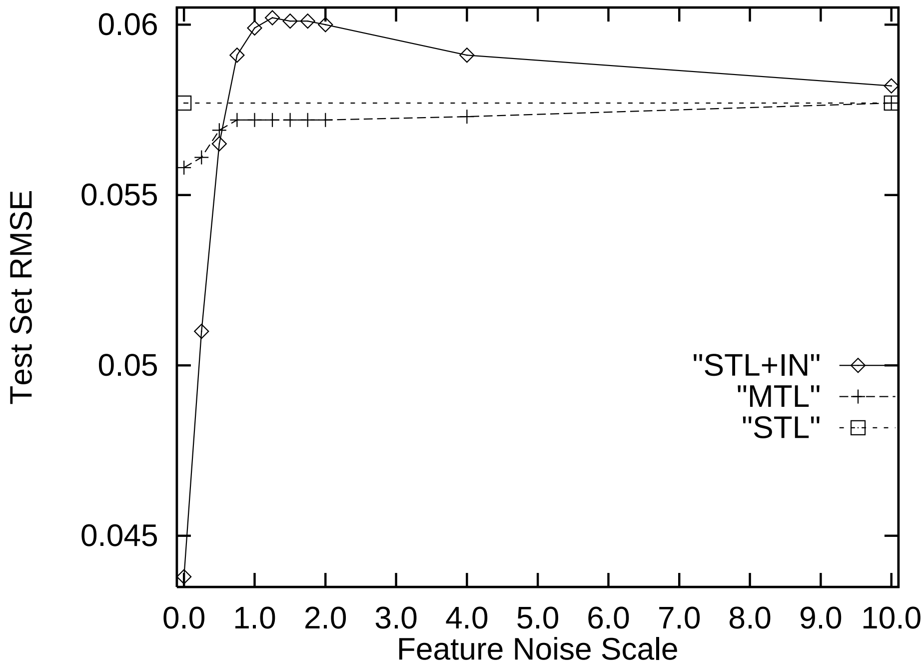


Figure 5.5: Performance on F1(A,B) of STL with the regular inputs, STL with the extra input for F2(A,B), and MTL with the extra output task for F2(A,B).

however, at some point they help more as extra outputs than as extra inputs, and never hurt performance the way the noisy extra inputs did. For $NOISE\_SCALE$ greater than about 0.5, it is better to use the extra feature as an extra output than as an input.

Why does noise cause STL with the extra inputs to perform worse than STL without those inputs? With a finite training sample, correlations in the sample between noisy inputs and the main task cause the network to use the noisy inputs. To the extent that the main task is a function of the noisy inputs, it must pass the noise to the output, causing the output to be noisy. Also, as the net comes to depend on the noisy inputs, it depends less on the noise-free binary inputs. The noisy inputs *explain away* some of the training signal, so less is available to encourage learning to decode the binary inputs.

Why does noise in the extra outputs not hurt MTL as much as noise in those extra inputs hurts STL? As outputs, the net is learning the mapping from the regular inputs to $EF(A)$ and $EF(B)$. Early in training, the net learns to interpolate through the noise and thus learns smooth functions for $EF(A)$ and $EF(B)$ that have reasonable fidelity to the true mapping. This makes learning less sensitive to the noise added to these features.[3]

Problem 2

$F1(A, B)$ is only mildly nonlinear because A and B do not go far into the tails of the $SIGMOID$. Do the results depend on this smoothness? To check, we modified $F1(A, B)$ to make it more nonlinear. Consider this function:

$F3(A, B) = SIGMOID(EXPAND(SIGMOID(A) - -SIGMOID(B)))$

where $EXPAND$ scales the inputs from $(SIGMOID(A) - -SIGMOID(B))$ to the range $[-12.5, 12.5]$, and A and B are drawn from $[-12.5, 12.5]$. $F3(A, B)$ is significantly more nonlinear than $F1(A, B)$ because the expanded scales of A and B, and expanding the difference to $[-12.5, 12.5]$ before passing it through another sigmoid, cause much of the data to fall in the tails of either the inner or outer sigmoids.

Consider these extra features:

$EF(A) = SIGMOID(A) + NOISE\_SCALE * Noise1$

$EF(B) = SIGMOID(B) + NOISE\_SCALE * Noise2$

where the Noises are sampled as before. Figure 5.6 shows the results of using extra features $EF(A)$ and $EF(B)$ as extra inputs or as extra outputs. The trend is similar to that in Figure 5.5, but the benefit of MTL with the extra outputs is even larger at low noise. A blow-up of the region probably of most interest in real problems is shown in the left graph in Figure 5.11.

The similarity between the graphs in Figure 5.5 and Figure 5.6 might raise concern that the behavior we are seeing is an artifact of these problems or some aspect of how they are trained. The similarity between the two graphs is due to the ubiquity of the phenomena. The data for the two sets of experiments were generated using different seeds. The first experiment was run using steepest descent and Mitre's Aspirin simulator. The second experiment used conjugate gradient and Toronto's Xerion simulator. And the two functions are not as similar as their definitions might suggest: expanding the range of the

[3] It is easy to shift the graphs of STL with extra inputs and MTL relative to each other by changing the functions or the sampling distributions. For example, sampling A and B so that their values are closer makes $SIGMOID(A) - SIGMOID(B)$ smaller and thus more sensitive to input noise. This shifts the tradeoff point between STL with extra outputs and MTL towards lower $NOISE\_SCALE$.

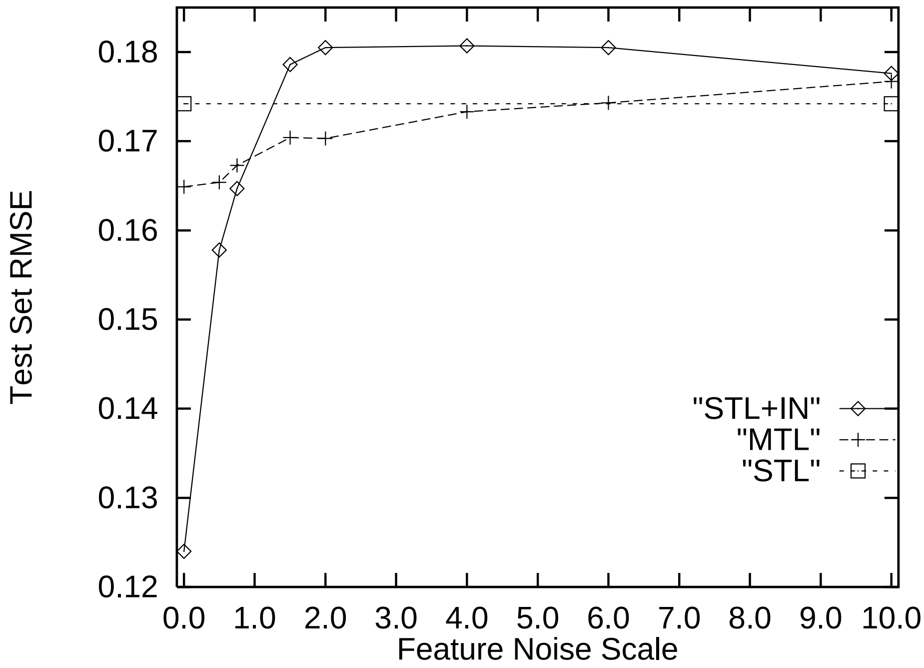


Figure 5.6: Performance on F3(A,B) of STL with the regular inputs, STL with the extra input for EF(A) and EF(B), and MTL with extra output tasks for EF(A) and EF(B).

data before passing it through a sigmoid, and then passing the differences through another sigmoid creates a much more nonlinear problem. Moreover, we used very large halt and test sets, and we were able to run 50 trials of each method at each noise level. The results are reliable.

5.1.3 A Classification Problem

This section presents a problem that combines feature correlation (Section 5.1.1) and feature noise (Section 5.1.2) into one problem.

Consider the 1-D classification problem, shown in Figure 5.7, of separating two Gaussian distributions with means 0 and 1, and standard deviations of 1. This problem is simple to learn if the 1-D input is coded as a single, continuous input. It can be made harder by embedding it non-linearly in a higher dimensional space. Consider encoding input values defined on [0.0,15.0] using an *interpolated* 4-D Gray code($\overline{GC}$). In an interpolated 4-D Gray code integer values are mapped to a 4-D binary Gray code in the usual way. The intervening non-integers are mapped linearly to intervening 4-D vectors between the binary Gray codes for the bounding integers. Because the Gray code flips only one bit between neighboring integers, this mapping involves simply interpolating along the one dimension in the 4-D unit cube that changes. For example, the value 3.4 is encoded as $.4(\overline{GC}(4) - \overline{GC}(3)) + \overline{GC}(3)$. $\overline{GC}(3) = 0010$. $\overline{GC}(4) = 0110$. Only the 2nd bit (from the left) changes going from 3 to 4. To represent the decimal part of the value 3.4, we scale this 2nd bit by 0.4, yielding 0,.4,1,0. Interpolated Gray codes are useful because they have the property of regular Gray codes that only one bit changes for transitions between neighboring values, but the representation is still continuous. Classification now requires decoding the Gray code input representation and determining the classification threshold.

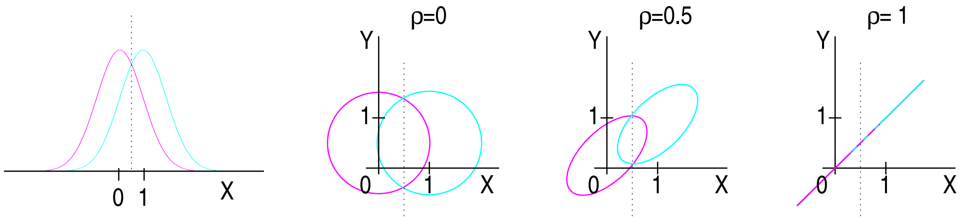


Figure 5.7: Two overlapped Gaussian classes (left), and an extra feature (y-axis) correlated different amounts ($\rho = 0$: no correlation, $\rho = 1$: perfect correlation) with the unencoded version of the regular input (x-axis)

The extra feature is a 1-D value correlated (with correlation ρ) with the original unencoded regular input, X. The extra feature is drawn from a Gaussian distribution with mean $\rho \times (X - .5) + .5$ and standard deviation $\sqrt{(1 - \rho^2)}$. Examples of the distributions of the unencoded original dimension and the extra feature for various correlations are shown in Figure 5.7. This problem has been carefully constructed so that the optimal classification boundary does not change as ρ varies.

Consider the extreme cases. At $\rho = 1$, the extra feature is exactly an unencoded version of the regular input. An STL net using this feature as an extra input could ignore the encoded inputs and solve the problem using this feature alone. An MTL net using this extra feature as an extra output would have its hidden layer biased towards representations that decode the Gray code, which is useful to the main classification task. At the other extreme ($\rho = 0$), we expect nets using the extra feature to learn no better than one using just the regular inputs because there is no useful information provided by the uncorrelated extra feature. The interesting case is between the two extremes. We can imagine a situation where as an output, the extra feature is still able to help MTL by guiding it to decode the Gray code, but as an input does not help STL because of the high level of noise.

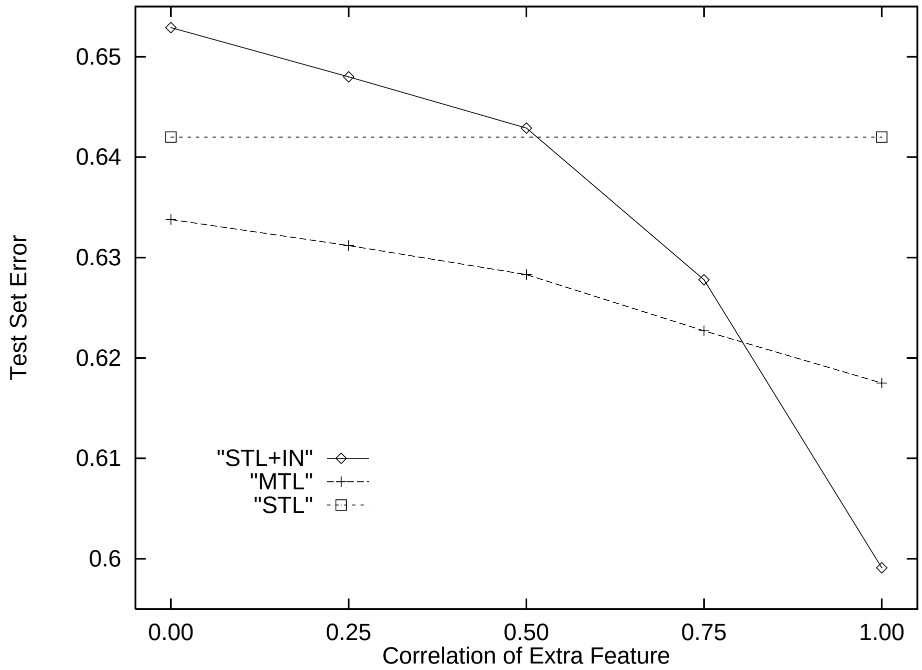


Figure 5.8: STL, STL with the feature used as an extra input, and MTL where the feature used as an extra output vs. ρ on the Classification Problem. The improvement of MTL over STL at $\rho = 0$ is not statistically significant, but does appear to fit the MTL trend well. We suspect this improvement may be due to noise injected by this uncorrelated output into the hidden layer acting as a regularizer. See Section 1.4 for more discussion of this effect.

The class output unit uses a sigmoid transfer function and cross-entropy error measure. The output unit for the correlated extra feature uses a linear transfer function and squared error measure. Figure 5.8 shows the average performance of 50 trials of STL, STL using the extra feature as an extra input, and MTL which uses the extra feature as an extra output as a function of ρ using networks with 20 hidden units, 70 training patterns, and halt and test sets of 1000 patterns each. As in the previous section, STL is much more sensitive to changes in the extra feature than MTL, so that by $\rho = 0.75$ the curves cross and for ρ less than 0.75, the dimension is actually more useful as an output dimension than as an extra input. The graph on the right side of Figure 5.11 shows a blow-up of the more interesting region of Figure 5.8.

5.1.4 Discussion

The problems used in this section are contrived. They are the simplest problems we could devise that exhibit the phenomenon. But we may have observed evidence for this sensitivity to noise in inputs on the pneumonia problem. Recall that when we tried feature nets on the pneumonia problem (Section 2.3.9), we did not observe improvements in performance comparable to MTL. The MTL net is learning an internal representation for the extra tasks. Feature nets provide a learned internal representation for the same extra tasks as extra inputs to a net. Are these two approaches that different? One difference is that the models learned for the extra tasks in the pneumonia domain are noisy; the extra tasks are not learned well. Because the models are poor, they act as noisy inputs. As we see above, noise in inputs can hurt learning even if those inputs contain otherwise useful information. As extra MTL outputs, however, there is no noise in the task signals because we can use the signals in the database, not predictions for them. If one compares the zero-noise points for MTL with the moderate-noise points for STL with extra inputs in Figures 5.5, 5.6, and 5.8, it is easy to see why MTL could outperform feature nets in domains like this where the extra tasks are not learned well. When the choice is between using extra tasks as extra MTL outputs, or using poor predictions of the extra tasks as extra inputs, using them as extra MTL outputs will probably work better.

5.2 Selecting Inputs and Extra Outputs

In the previous section we used synthetic problems to demonstrate that some input features are more useful when used as extra *outputs* than when used as inputs. In this section we use feature selection on a real problem, DNA splice-junction, to find features that are more useful when used as outputs than as inputs. We use the feature selection method devised by Koller and Sahami [Koller & Sahami 1996] to select which features to use as inputs. Instead of ignoring the features eliminated by feature selection, we will use them as extra outputs for MTL. On the DNA splice-junction problem, using some features as inputs and other features as extra outputs yields better performance than using all features as inputs or using just selected features as inputs. Since many real-world problems have excess features, this process makes MTL applicable to real-world domains where other sources of extra tasks may not exist.

5.2.1 Feature Selection

Feature selection is the process of selecting a subset of the available features to use as inputs for learning. When there are few features and all of these are relevant for learning, feature selection is not important. In real-world problems, however, there is often an excess of features. Many of the features may be irrelevant to the learning task at hand. And many features may be redundant with information contained in other features. Because learning algorithms usually have difficulty coping with large numbers of redundant and irrelevant features, performance often improves considerably when the learning method is given only the subset of features most useful for the learning task.

Here we use the feature selection method developed by Koller and Sahami. This method is a learning-algorithm independent feature selector that uses information theoretic measures of feature importance to select the features most likely to be useful as inputs. Currently, the Koller-Sahami algorithm is applicable only to classification problems defined on boolean features.

The theoretical motivation behind the Koller-Sahami algorithm is to remove attributes which have markov blankets in the remaining attributes. The markov blanket of an attribute is the minimal set of other attributes such that all other attributes not in the blanket are conditionally independent of the attribute when conditioned on the markov blanket attributes. A markov blanket isolates an attribute from all other attributes. If an attribute has a markov blanket in the other attributes, then this attribute provides no additional information, and the class decision is independent of the value of this attribute conditioned on the values of the other attributes in the blanket. In practice, finding markov blankets is not practical given a large number of attributes. The Koller-Sahami algorithm makes several simplifying approximations that allow the degree to which an attribute is blanketed by other attributes to be estimated in reasonable time. These approximations yield an algorithm capable of doing feature selection on problems containing hundreds to several-thousands of features in a few hours on a workstation.

The Koller-Sahami feature selector is a greedy feature selector. The preferred way of using the algorithm is backward-elimination. Start with all features in the set, and remove attributes one-at-a-time, at each step removing the attribute most covered by the other attributes remaining in the set. That is, at each step the algorithm removes the attribute that appears to provide the least additional information for the class given the other remaining attributes. This greedy approach, while suboptimal in many domains, is effective and works well on many domains, including the DNA splice-junction domain.

5.2.2 The DNA SPLICE-JUNCTION Problem

DNA contains coded information that is used by cells to construct proteins. In the process of building a messenger RNA (mRNA) molecule that will be used as a template from which to build a protein, large sections of the original DNA coding are ignored. The coded sequences that are used are known as "exons", while the ignored sequences are known as "introns". The nature of the boundaries between exons and introns, known as "splice junctions", is a subject of active research.

The DNA splice-junction Problem we use is available in the UCI machine learning repository [Noordewier, Towell & Shavlik 1991]. For each case in the database we are given a sequence of 60 nucleotides. The goal is to predict if the center of the nucleotide sequence codes for an exon-to-intron boundary, an intron-to-exon boundary, or neither. 25% of the cases are EI boundaries, 25% are IE boundaries, and 50% are neither EI or IE boundaries. For compatibility, we use the nucleotide coding scheme used by Koller and Sahami. This scheme codes each of the 60 nucleotides in the DNA sequence using 3 bits. This yields a total of 180 boolean attributes that might be used as inputs. Typical performance on this problem is 92-94% accuracy when trained on training sets containing 1000-2000 cases.

5.2.3 Experiments

We've run two experiments with the DNA problem. In the first experiment, we determine how many hidden units to use in the backprop nets. This experiment also shows the benefit of using feature selection to limit the number of features used as inputs to the nets. In the second experiment we determine if using some features as extra outputs improves performance on splice-junction recognition.

In all our experiments we use backprop nets composed of sigmoid units. We train the nets using conjugate gradient in the Xerion simulator from the University of Toronto. We use early stopping on an independent halt set to determine when to stop training. The performance of the net is then measured on an independent test set not used for backpropagation or early stopping. The dataset we are using contains 2000 cases. We randomly split this set into train, halt, and test sets containing 667, 666, and 667 cases, respectively. We repeatedly sample the dataset this way to generate multiple trials.

Our coding for the main splice-junction task uses three outputs, one for IE, one for EI, and one for neither. We use a normalized cross-entropy loss function for the outputs for the main task. Normalized cross entropy is a standard way of preserving probability semantics when multiple outputs code for mutually exclusive classes. The output activations are normalized by dividing each output's activation by the sum of the activations of the three outputs coding for IE,EI,Neither. This normalization is done prior to computing and backpropagating the error at each output. The classification of the net prediction is done in the usual way by finding which of the three outputs has the highest activation. When some of the boolean attributes are used as extra outputs, we use a non-normalized cross-entropy loss function to train them.

Experiment 1: Performance vs. Net Size

The purpose of this experiment is to determine what net size yields the best performance on the DNA domain. We tried nets containing 5, 20, 80, 320, and 1280 hidden units. The nets have 3 outputs that code for the main task.

Figure 5.9 shows the test set cross-entropy error for the nets of the different sizes when trained with all 180 input features, and when trained with the 30 input features selected by the Koller-Sahami feature selector. (We select 30 features from the 180 because this is the number of features Koller and Sahami selected for their experiments with this domain.) Each data point is the average of 12 trials; the vertical bars are 95% confidence intervals for the estimates. From the graph it is clear that better performance is achieved with nets with 80, 320, or 1280 hidden units. (It is common for early stopping to favor large nets.)

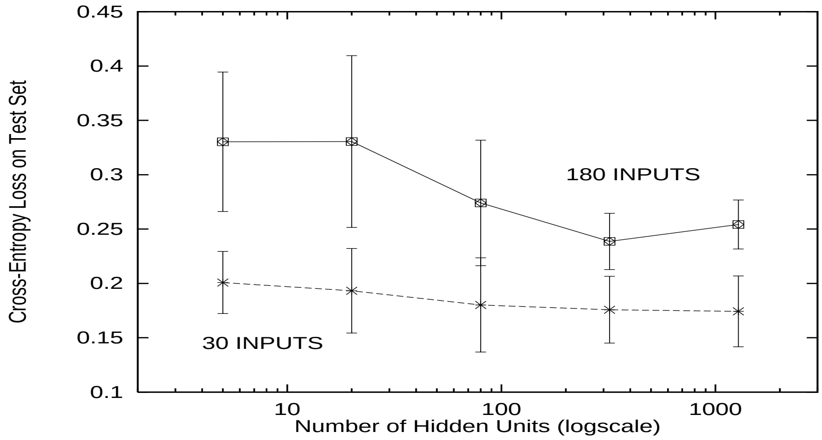


Figure 5.9: Cross-Entropy performance of different size nets with all 180 inputs and 30 selected inputs

From the graph it is also clear that cross-entropy performance is significantly better for nets that use only 30 selected features as inputs as compared to nets that use all 180 features as inputs.

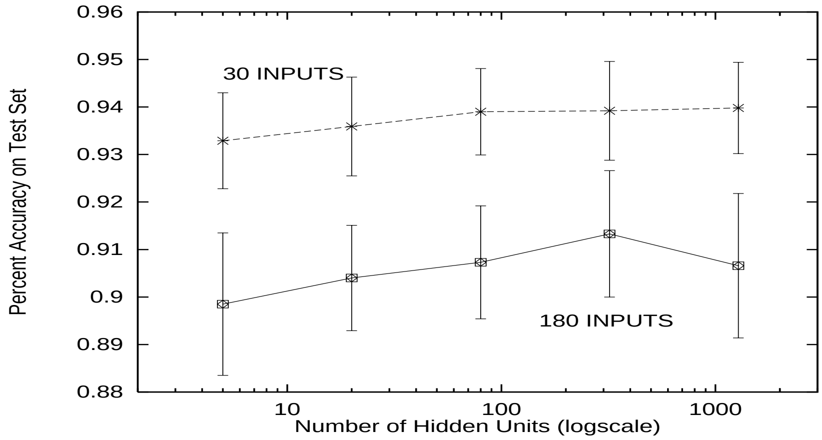


Figure 5.10: Prediction accuracy of different size nets with all 180 inputs and 30 selected inputs

Figure 5.10 shows the splice-junction prediction accuracy of the 180 and 30 input nets as a function of net size. Once again, performance is best with large nets, although this graph suggests that accuracy is best with nets nearer to 320 hidden units than to 1280 hidden units. This graph also shows that using the reduced set of input features improves accuracy on this domain.

We conclude from the experiments in this section that the optimal net size is approximately 320 hidden units, and that training nets with the 30 features selected with the Koller-Sahami feature selector yields better performance than using all 180 inputs.

Experiment 2: Using Unused Features as Extra Outputs

In the previous experiment, the 150 features *not* selected for use as inputs by feature selection were thrown away. They were not used as inputs or as outputs. In this section we use some of the features not used as inputs as extra outputs for multitask learning.

The Koller-Sahami algorithm is told how many attributes to remove. It does not automatically determine when to stop removing attributes. We ran the Koller-Sahami feature selector and had it remove all 180 attributes in the DNA problem. Because the feature selector is a greedy algorithm that removes attributes one-at-a-time, this creates an ordering on the attributes. As above, we use the last 30 attributes removed by the algorithm as inputs.

Rather than ignore the remaining 150 attributes, we use the *next* 30 attributes as extra outputs for multitask learning. These are the attributes of the remaining 150 attributes that the feature selector considers most useful. (We use 30 of the remaining attributes instead of all 150 as extra outputs mainly for computational efficiency. When the number of extra outputs on a net is large, the number of hidden units must be increased almost proportionately so that each output is guaranteed a certain minimum number of hidden units. Using all 150 unused inputs as extra outputs yields nets so large that we could not afford to run many trials.)

The multitask learning net has 30 inputs, 3 outputs for the main task, and an additional 30 outputs for the 30 attributes that were not selected to be used as inputs. The multitask learning net has 1280 hidden units (instead of 320) because it is learning many more tasks. We have not attempted to find the optimal number of hidden units for the multitask learning net, and we suspect that the multitask learning net would perform better with more than 1280 hidden units. This biases our experiments in favor of the nets that do not use extra outputs because we are using a nearly optimal number of hidden units on those nets (and that near-optimal net size was determined using the same dataset).

Table 5.2: Cross-Entropy Performance of Different Combinations of Inputs and Outputs. All differences except the difference between Net2 and Net4 are statistically significant at .05 or better.

Net	**HiddenUnits**	**Inputs**	**ExtraOutputs**	**CrossEntropy**	**StdErr**
Net1	320	180	0	0.257	0.006
Net2	320	30	0	0.180	0.009
Net3	1280	30	30	0.167	0.006
Net4	320	60	0	0.187	0.006

Table 5.2 shows the cross-entropy error for four different nets. Table 5.3 shows the prediction accuracy for the same four nets. Net1 has 320 hidden units and uses all 180 inputs. This is the traditional way of training on this task without any feature selection. Net2 has the same number of hidden units, but uses only the 30 features selected by feature selection as inputs. Net3 is the multitask learning net. It uses the same 30 attributes as inputs as Net2. Net3, however, also uses the next best 30 attributes as extra outputs. These are attributes that are ignored by Net2. Finally, Net4 uses as inputs both the 30 attributes used by Net2 as inputs, and the 30 attributes used by Net3 as extra outputs. Net4 thus uses all the attributes Net3 uses, but Net4 uses all of them as inputs. It has no extra outputs.

Table 5.3: Predictive Accuracy of Different Combinations of Inputs and Outputs. The difference between Net2 and Net3 just misses being significant at .05 with 10 trials.

Net	**HiddenUnits**	**Inputs**	**ExtraOutputs**	**% Accuracy**	**StdErr**
Net1	320	180	0	90.98%	0.35
Net2	320	30	0	94.16%	0.19
Net3	1280	30	30	94.32%	0.18
Net4	320	60	0	93.66%	0.21

Net1 is clearly the worst performer on this problem. Using all 180 attributes as inputs is not the best thing to do. Using only the 30 features selected with the Koller-Sahami algorithm as inputs (Net2) yields significantly better performance. Net3, however, performs even better. It is best to use some of the features not used as inputs as extra outputs instead of ignoring them. Net4, which uses all features used by Net3 (but as inputs), does not perform as well as Net3 (nor even as well as Net2).

Net3 reduces the cross entropy 7.2% compared with Net2, and 10.7% compared with Net4, which uses the same features. In predictive accuracy, Net3 reduces the error 2.7%

compared to Net2, and 10.4% compared with Net4. *In the DNA splice-junction domain it is better to use some of the features as extra outputs than as inputs. Moreover, in this domain at least, the Koller-Sahami feature selection algorithm is an effective way of selecting some input features as candidates for use as extra outputs.* We do not, however, know if the Koller-Sahami algorithm is an effective way of selecting which of the available outputs should be used for MTL. Although we used the next 30 features not used as inputs as the extra MTL outputs, we do not know if this yields the best 30 outputs for MTL.

The results with splice-junction show that the benefit of using a feature as an extra output is different from the benefit of using that feature as an input. As an input, the net has access to the feature's values on the training and test cases for prediction. As an output, however, the net is instead biased to learn a mapping from the other input features to that output. The splice-junction results demonstrate that what is learned for this mapping is sometimes more useful than the feature value itself, particularly if the value of the feature as an *additional* input is marginal or even harmful.

This section showed that the DNA splice-junction domain contains features that improve recognition accuracy more if used as extra outputs than if used as inputs. This result confirms our expectation that there are real-world problems where some features could be better used as extra outputs than as inputs. Using the features selected with the Koller-Sahami feature selector as inputs yields better accuracy than using all the features as inputs. Even better accuracy can be obtained by using some of the features not used as inputs as extra outputs instead of ignoring them (or using them as additional inputs). Many real-world problems might benefit from a similar combination of feature selection and multitask learning.

5.3 Using Features as Both Inputs and MTL Outputs

We have extra tasks that can be used as inputs, but using them as inputs sometimes hurts performance (e.g., if they are too noisy). The benefit of using the extra MTL outputs, however, is sometimes not as large as the benefit of using them as inputs (e.g., when they are not too noisy). Furthermore, there is an interesting regime where they help learning when used as an output, or as an input. Figure 5.11 shows a blow-up of these regions from two of the problems presented in Section 5.1.

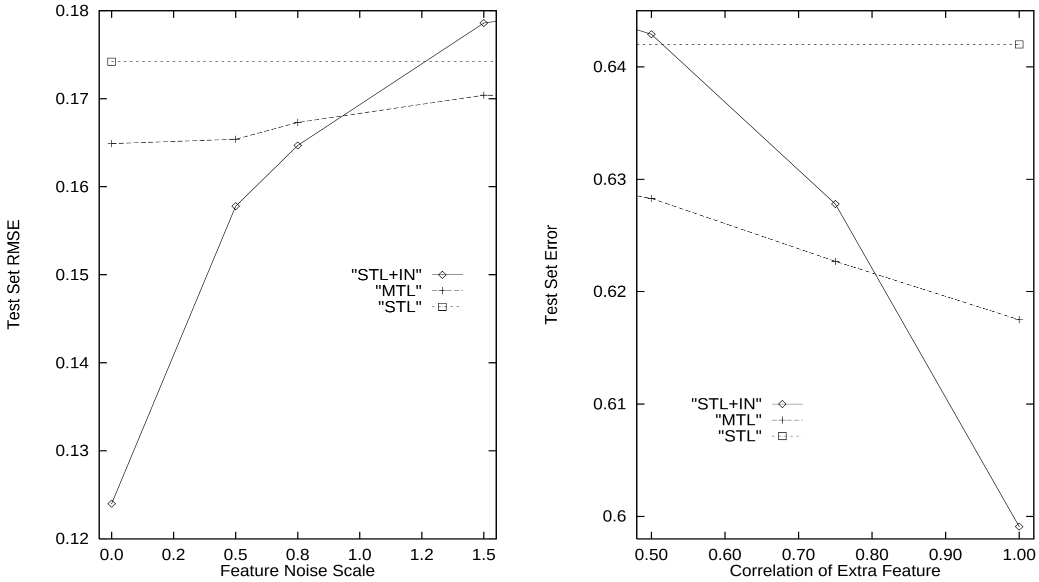


Figure 5.11: Blow-ups of the regions of most practical interest for the problem in Section 5.1.2 and the classification problem in Section 5.1.3. STL is STL using the regular inputs. STL+IN is STL using the extra feature(s) as extra inputs. MTL uses the extra feature as an extra output.

In this region there are benefits from using the extra features both as extra inputs and as extra outputs.

Wouldn't it be nice if we could use these extra tasks both as extra input features and as extra output tasks? The difficulty of using the same task values as inputs and outputs is that backprop would almost certainly learn connections that would directly map the input to its corresponding output. Since a direct connection like this would present zero error to the output, nothing of interest would be learned for the extra outputs in the hidden layer. *Putting the same task values on the inputs and outputs of a simple, fully-connected feedforward MTL net effectively turns MTL off.*

5.3.1 Using Network Architecture to Isolate Outputs from Inputs

Figure 5.12 shows an architecture that combines the architectures in Figures 5.3 and 5.4. It uses two disjoint hidden layers to prevent output tasks from "seeing" the same task value that is used as an input. The architecture allows some of the benefits of MTL. That is, it learns models internally for the extra tasks (even though they are used as inputs) and makes what is learned by those models available to the main task.

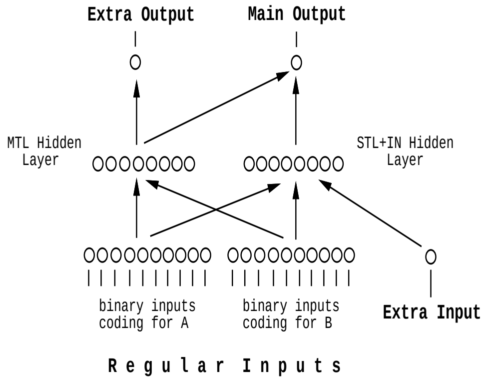


Figure 5.12: An MTL architecture that combines learning the main task using an extra input feature, and learning an extra task that is this input feature. Each arrow represents full feed-forward connections between the two layers.

The two hidden layers in Figure 5.12 are not at different depths, but connect to different inputs and outputs. The hidden layer on the right "sees" all inputs, and connects on the output side to the main task (and to any extra tasks that would not be used as extra inputs). This is similar to the hidden layer in the STL net that has extra inputs (Figure 5.3). The hidden layer on the left, however, does not see the extra inputs; it sees only the regular inputs. On the output side, however, it connects to not only the main task, but also to the extra tasks that are also being used as extra inputs. This hidden layer is similar to the hidden layer in the MTL net in Figure 5.4. It learns the extra tasks, but does not see those tasks as inputs. Because this hidden layer does not see the extra tasks as inputs, it cannot learn direct connections to map tasks used as inputs to those same tasks used as outputs. It must learn a model for the extra outputs, even though a different part of the same net is using those same extra tasks as extra inputs.

5.3.2 Results

Figure 5.13 shows the results of using the architecture in Figure 5.12 on problem F1(A,B) using EF(A) and EF(B) both as extra input features and as extra output tasks. MTL using the extra features as both inputs and outputs performs as well as MTL with just the extra outputs in the high noise regions of the graph, and better in the low noise regions of the graph. The performance of STL using the extra features as extra inputs is still better when the extra features have very low noise. But the tradeoff point where it becomes better to use the extra features as extra outputs shifts towards lower noise when compared with Figure 5.5. Although MTL+IN appears to perform better than MTL for high noise regions of the graph, these differences are not significant. All methods are not statistically distinguishable from each other for noise scales 10.0 or above.

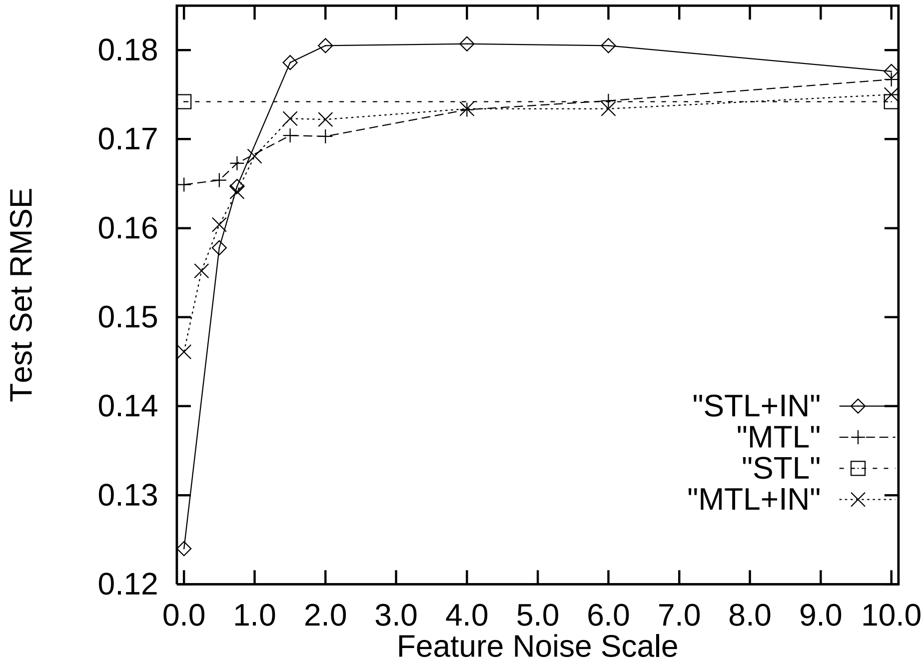


Figure 5.13: Performance on F3(A,B) of STL with the regular inputs, STL with the extra input for EF(A) and EF(B), and MTL with extra output tasks for EF(A) and EF(B).

Figure 5.14 shows the results of using the same architecture on the classification problem from Section 5.1.3. Again, using the extra feature as both inputs and outputs improves the performance of MTL in the region where ρ is closer to one (i.e., where the features have less noise and are more useful as inputs) and shifts the point where the tradeoff between using the extra feature only as an extra input, and using it as both extra inputs and outputs, happens. Although performance appears to be slightly worse at the low ρ part of the graph, the differences there are not statistically significant with 50 trials.

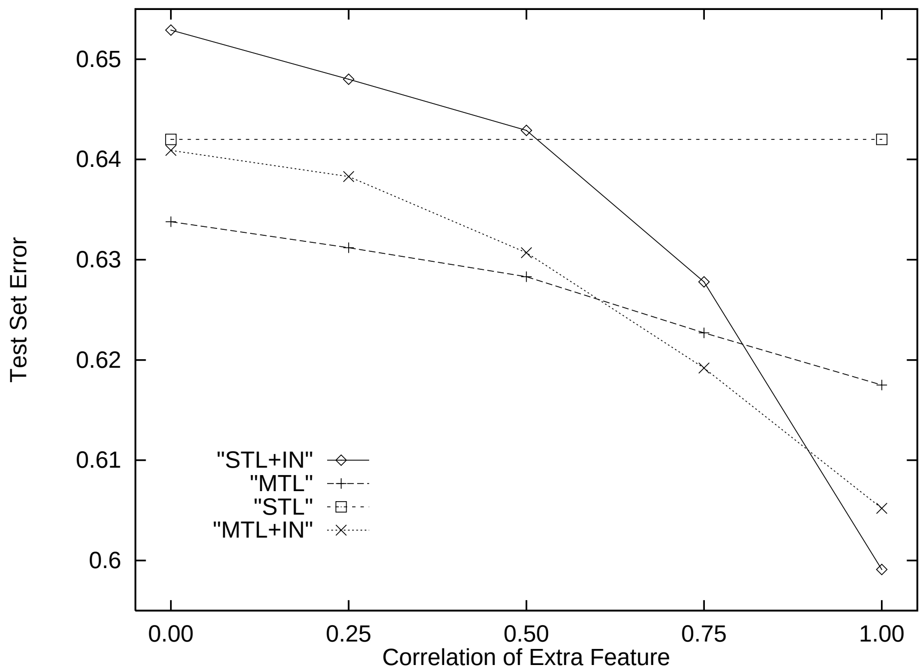


Figure 5.14: STL, STL with the feature used as an extra input, and MTL where the feature used as an extra output vs. ρ on the Classification Problem

Figure 5.15 shows blow-ups of Figures 5.13 and 5.14 similar to those shown in Figure 5.11. In both graphs, the curve for MTL+IN yields performance between STL+IN and MTL where the extra feature is most useful (low noise or high correlation). This shifts the tradeoff points where it is better to just use the extra feature as an input for STL. This suggests MTL+IN will outperform STL+IN on real problems. Moreover, it demonstrates the existence of regions (noise scales 0.75–1.5 and ρ 0.6–0.8) where MTL+IN is the best performer. If you want best performance on these problems in these regions you must use MTL+IN.

5.3.3 Discussion

We conclude that this approach to using some features as both inputs and as extra outputs has promise. If we can devise better architectures that allow MTL+IN to always perform as well as, or better than, the best of STL and MTL, there will never be a reason to use STL or STL+IN.

Not all features can be used as extra outputs using the architecture in Figure 5.12. In

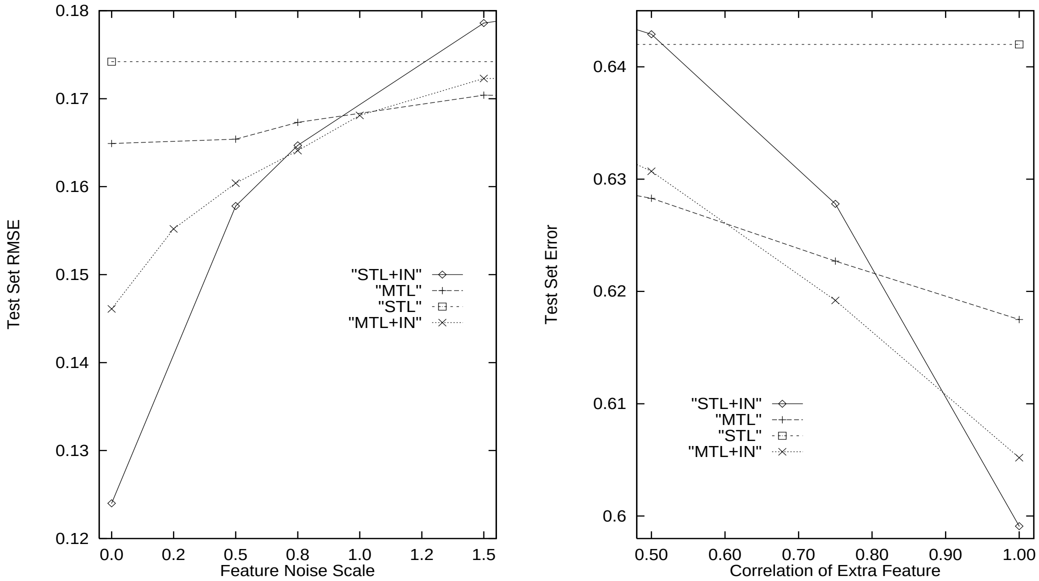


Figure 5.15: Blow-ups of the same regions shown in 5.11. STL is STL using the regular inputs. STL+IN is STL using the extra feature(s) as extra inputs. MTL uses the extra feature as an extra output. MTL+IN is the addition to the graphs. It uses the extra feature(s) as both inputs and extra MTL outputs.

order for extra outputs to be more useful than the extra inputs, there must be enough information in the other inputs to learn the problem. Fortunately, in many real world domains there is considerable redundancy in the input features. We are convinced many real problems would benefit from using some of the inputs as outputs instead. We are currently exploring alternate architectures that may improve performance further, and that will allow *all* features to be used as both inputs and outputs.

5.4 Chapter Summary

This chapter shows that the benefit of using a feature as an extra output is different from the benefit of using that feature as an input. As an input, the net has access to the value of the feature on both the training and test cases. As an output, the net only has access to the value of the feature during training, and it is biased to learn a mapping from the other inputs in the training set to that output. It is this mapping that is then available to the net during testing instead of the value of the feature itself. Because of this difference, sometimes it is better to use some features as outputs rather than as inputs.

In Section 5.1 we demonstrated that some features are more useful as extra outputs. In this section we used synthetic problems where it is easy to understand which features would more useful as outputs than as inputs and why. In Section 5.2 we showed a real-world problem, DNA SPLICE-JUNCTION, where some features were more useful as outputs than as inputs. We also showed that feature selection could be employed to find inputs that might be more useful as extra outputs. Feature selection finds a good subset of the features to use as inputs. Features not used as inputs are then candidates for use as extra outputs. This approach avoids the combinatorial explosion that would occur if we tried to evaluate all possible combinations of features used as inputs or as extra outputs.

The graphs in Section 5.1 make it clear that some features help when used either as an input, or as an output. Since the benefit of using a feature as an extra output is different from that of using it as an input, can we get both benefits? Yes. The approach we followed is to break the single fully connected hidden layer used in previous MTL nets into two disjoint hidden layers, one of which is not able to see all the inputs. This architecture reaps both benefits by allowing some features to be used simultaneously as both inputs and outputs while preventing learning direct feedthrough identity mappings. This method is still being refined, but the early results are promising. We were able to achieve performance on two synthetic problems from Section 5.1 using some features as both inputs and extra outputs that we could not achieve by using those features as just inputs or just extra outputs. Further development and testing on real problems is necessary before we can judge how useful it will be in practice to use features as both inputs and extra MTL outputs.

Most of the performance differences we observe in this chapter are small. The benefit of using some features that could be used as inputs as extra outputs instead is often not that large. Using inputs as extra MTL outputs probably will not be the most important application of MTL to real problems. This chapter does, however, show that MTL can potentially be applied to any real world problem because most real problems have a feature selection problem and this creates the possibility that some of those features not selected for use as inputs might best be used as outputs instead of discarded. Moreover, if we can improve the technology that allows features to be used as both inputs and extra MTL outputs on one net, MTL potentially would be applicable to all problems in machine learning.

Chapter 6

Beyond Basics

The basic machinery for doing multitask learning in neural nets is already present in back-propagation. Backprop, however, was not designed to do MTL well. This chapter is presents a few techniques that make multitask learning in backprop nets work better. Some of the techniques may be counterintuitive. Some are so important that if they are not followed, multitask learning can hurt generalization performance instead of helping it.

MTL trains multiple tasks in parallel not because this is a more efficient way to learn many tasks, but because the information in the training signals for other tasks can help one task be learned better. Sometimes what is optimal for one task is not optimal for all tasks. When this is the case, it is important to optimize the technique so that performance on the *main* task is best, even if this hurts performance on the extra tasks. If several or all of the tasks are important, it may be best to rerun learning for each important task, with the technique optimized for each important task one at a time. This point-of-view is important. It allows us to develop asymmetric methods that favor performance on one task at the expense of poorer performance on other tasks. Unlike previous chapters, all the methods in this chapter improve performance on the main task(s) by treating the main task(s) differently than the extra tasks. The only method described in this chapter used on problems earlier in this thesis is early stopping on tasks individually. The other methods presented in this chapter could be applied to most problems examined in this thesis. We expect that in many cases this would improve the performance of MTL.

6.1 Early Stopping

Most of the neural net experiments reported in this thesis use early stopping. Early stopping is a way of preventing overfitting by halting the training of error-driven procedures like backprop before they achieve minimum error on the training set. Early stopping is usually done using an independent test set (the "halt set") that is not used for backpropagating errors. Performance on the halt set is monitored as backpropagation is done on the training set, and training is halted when performance on the halt set stops improving or starts getting worse. Early stopping is important in most applications of backprop nets if one is to achieve best performance.

Recall the 1D-ALVINN domain used in Section 2.1. In this problem the main task is to predict the steering direction for an autonomous vehicle given images of the road in front of the vehicle. We applied MTL to 1D-ALVINN by training a neural net on eight extra tasks while it trained on the main steering task:

- whether the road is one or two lanes
- location of left edge of road
- location of road center
- intensity of region bordering road
- location of centerline (2-lane roads only)
- location of right edge of road
- intensity of road surface
- intensity of centerline (2-lane roads only)

The MTL net for 1D-ALVINN has nine outputs, one for the main steering task and one for each of the eight extra tasks. Figure 6.1 shows nine graphs, one for each of the nine tasks trained on the MTL net. Each graph is the root-mean-squared-error of one of the outputs on a halt set as the MTL net trains. Usually, training causes the error on the halt set to fall, then level off, then begin to rise again; performance on the training set (not shown because it quickly falls off the bottom of the scale) continues to fall throughout training. Early stopping halts training at the epoch where performance on the halt set is best.

By examining the graphs, it is clear that the best place to halt training differs for each task. The road_center task reaches peak performance after 200,000 backprop passes, but the main steering task would perform best if halted at 125,000 passes. (Table 6.1 shows the best place to halt each task.) There is no one epoch where training can be stopped so as to achieve maximum performance on all tasks. If all the tasks are important, and one net

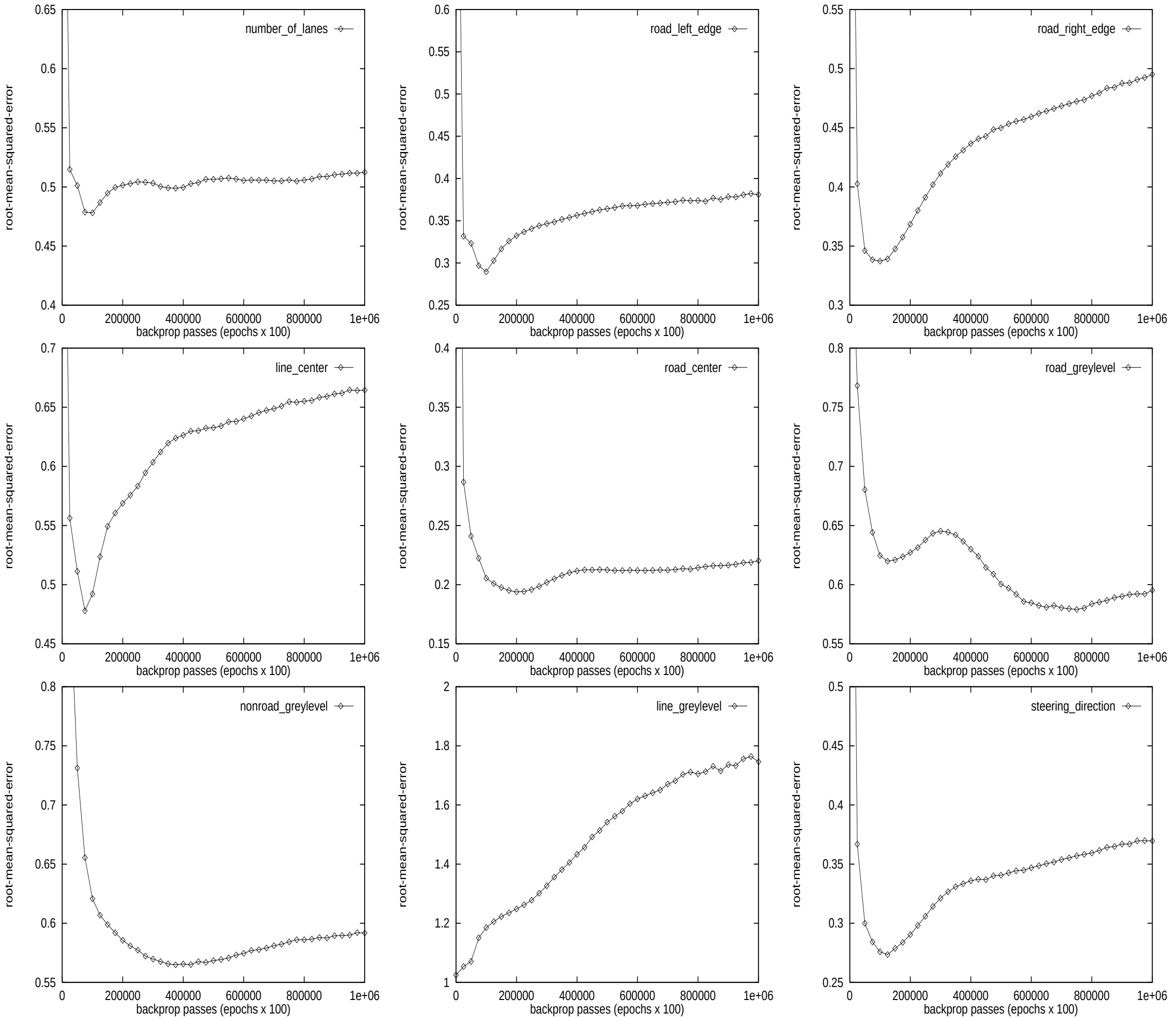


Figure 6.1: Test-Set Performance of MTL Net Trained on Nine 1D-ALVINN Tasks.

is to be used to predict all the tasks, one place to halt training is where the error on all the outputs combined is minimized. Figure 6.2 shows the mean RMS error of the nine tasks combined as the MTL net is trained. The best average RMSE occurs at 75,000 backprop passes.

But using one net to make predictions for all the tasks is suboptimal. Better performance can be achieved by halting training on each output individually and using the snapshot of the net taken at that epoch to make predictions for that task. We refer to the net halted when performance on particular task is best as a "snapshot" because the MTL net is still trained on all tasks until a snapshot has been made for all tasks of interest. In 1D-ALVINN,

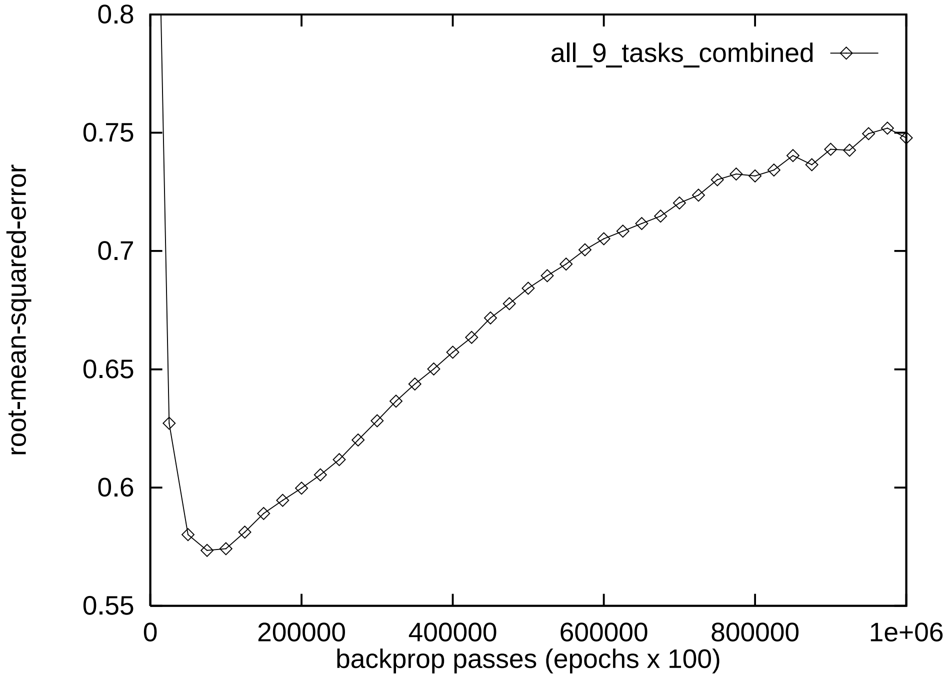


Figure 6.2: Combined Test-Set Performance on all 1D-ALVINN Tasks.

if all tasks were of interest (unlikely since there probably is little use for predictions of the road intensity), there would be nine snapshots made of the MTL net, one for each of the nine tasks.

Table 6.1 compares the performance on 1D-ALVINN if early stopping is done per task with the performance that would be obtained by halting training for the entire MTL net at one place using the combined RMSE. On average, halting tasks individually reduces error 9.0%. This is a large difference. For some tasks, the performance of the MTL net is worse than the performance of STL on this task if the MTL net is not halted on that task individually but is halted at the point where the combined error is lowest. We conclude that if tasks do not all reach peak performance at the same time (the usual case), it is important to do early stopping individually on each task of interest. Early stopping on tasks individually is very important. We used it for all experiments reported in this thesis. (Section 8.1 shows how performance on NETtalk could be improved by stopping on tasks individually.

Before leaving this topic, it is important to recognize that the training curves for the

Table 6.1: Performance of MTL on each 1D-ALVINN task when training is halted on each task individually compared with the performance on the tasks when halting using the combined RMSE across all tasks. Halting on each task individually reduces error about 9% on average.

TASK	**Halted Individually**		**Halted Combined**		**Difference**
	BP Pass	Performance	BP Pass	Performance	
1: 1 or 2 Lanes	100000	0.444	75000	0.456	2.7%
2: Left Edge	100000	0.309	75000	0.321	3.9%
3: Right Edge	100000	0.376	75000	0.381	1.3%
4: Line Center	75000	0.486	75000	0.486	0.0%
5: Road Center	200000	0.208	75000	0.239	14.9%
6: Road Greylevel	750000	0.552	75000	0.680	23.2%
7: Edge Greylevel	375000	0.518	75000	0.597	15.3%
8: Line Greylevel	1	1.010	75000	1.158	14.7%
9: Steering	125000	0.276	75000	0.292	5.8%

individual outputs on a net with multiple outputs like an MTL net are not necessarily monotonic. While it is not unheard of for the test-set error of a single-output net to be non-monotonic, the training-set error for a single-output net should descend monotonically or become flat. If batch gradient descent is done properly, the training-set error should never increase. This does not hold for errors measured for individual outputs on a multiple-output net. The aggregate training-set error summed across all outputs should never increase, and the aggregate test-set error will usually have a single minimum, but any one output may exhibit more complex behavior. The graph for road_greylevel (graph number 6) in Figure 6.1 shows a strong multimodal test-set curve. The corresponding training set curve for this output is similar in shape. The extra complexity of MTL training curves sometimes makes judging when to halt training more difficult.

Because the generalization curves for individual tasks often are multimodal, we do early stopping by training MTL nets until the halt-set performance of *all* tasks appears to have levelled off or begun overfitting. Often this is well beyond the point where the main task stopped improving. We then examine the entire generalization curve for the main task to find the backprop pass where performance was maximized. We use the network saved at that time for future predictions (or repeat training but stop at that backprop pass).

6.2 Learning Rates

The graphs showing the learning curves for the nine 1D-ALVINN tasks in the previous section raise some interesting questions. Is it possible to control the rates at which different tasks train so they each reach their best halt-set performance at the same time? Might controlling the rates at which different tasks train yield better MTL performance for the main task? Would best performance on each task be achieved if each task reached peak performance at the same time? If not, is it better for extra tasks to learn slower or faster than the main task? Can we make a general statement about how fast extra tasks should learn relative to the main task, or will this be different for each extra task?

The rate at which different tasks learn using vanilla backpropagation is almost certainly not optimal for MTL. Consider a task that trains many times slower than the main task. Most of what is learned for the slow task is learned after the snapshot for the main task is taken. Thus most of what is learned for the slow task cannot benefit the main task.

Surprisingly, a task that is learned many times faster than the main task can also hurt the performance of the main task. If the fast task is well into overfitting by the time the main task is learned, the representation in the hidden layer for the fast task probably is not as useful to the main task as it would have been before it began to overfit. Moreover, if the main task shares considerably with this faster task, the faster task may pull the main task into premature overfitting.

6.2.1 Learning Rate Optimization

The easiest and most direct method of controlling the rate at which different tasks learn is to use different learning rate on each task, i.e., on each output. Consider again the 1D-ALVINN problem. The MTL net for 1D-ALVINN has nine outputs, one for the main task and one for each of the eight extra tasks. In the previous experiments we used the same learning rate for each output. To roughly balance the importance of each task to training, we preprocessed the data sets by rescaling the outputs for each task so that they have the same variance. Even with this variance preprocessing, however, we observed that tasks train at different rates as shown in Figure 6.1.

Rather than use the same learning rate for each task, can we adjust learning rates so that performance on the main task is optimized? For problems like 1D-ALVINN that have a small number of extra tasks, we can use gradient descent to optimize the learning rates of the extra tasks.

We use perturbation to estimate the gradient of the generalization performance of the main task with respect to the learning rates of the extra tasks. First, we estimate the generalization performance of the initial learning rates (usually we initialize the learning rates of all tasks to 1.0) by training an MTL net until early stopping on the main task halts training. The performance on the halt set is an estimate of the performance given those learning rates. Then we perturb the learning rate for one task by increasing it. Again we train a net (from the same initial weights as before, and using the same train and halt sets), halt training on the main task using the halt set, and measure the performance of the net on the halt set. We do this for each extra task. If there are eight extra tasks, we must train nine neural nets to estimate the gradient once.[1]

6.2.2 Effect on the Main Task

We applied gradient descent on the learning rates for the eight extra tasks in 1D-ALVINN using the estimated gradients computed as described above. To minimize the number of times the gradient must be computed, we do line searches so that each gradient is fully exploited before a new one must be computed. Once the line search converges, we recompute the gradient at the new point. We repeat the process until the early-stopping performance on the halt set stops improving.

Because this procedure repeatedly evaluates performance on the same halt set, overfitting to that halt set is more likely than it is when using the halt set once for early stopping.

[1] It is possible to analytically compute the gradient of the learning rate for each output that will reduce error on the main task output fastest on each backprop pass. This has the disadvantage, however, of favoring learning rates that optimize performance on the training set, not generalization performance. Often what works best on the training set does not generalize well. The main goal of MTL is to improve generalization accuracy. Because of this, we use the performance on the test set as the criterion we wish to optimize. Unfortunately, we know of no general way to analytically compute or estimate the effect of different learning rates on test set performance, though there may heuristic methods that work well in practice..

We have not found this to be a problem so far. As usual, we report results from a second independent test set to prevent this from yielding optimistically good estimated generalization performance. (Where overfitting to the halt set becomes a problem, one solution is to split the halt set into two halt sets, one which is used to do early stopping when each net is trained, and the other used to halt the outer level learning rate optimization process before it overfits to the first halt set.)

Table 6.2 shows the performance on the main task before and after optimizing the learning rates of the eight extra tasks. The first five rows in the table are the results from five different runs using different data sets and initial network weights. The bottom row in the table is the average of the five trials. Optimizing the learning rates for the extra MTL tasks improved the performance on the main task an additional 11.5%. This improvement is over and above the original improvement of 15%–25% for MTL over STL.

Table 6.2: Performance of MTL on the main Steering Direction task before and after optimizing the learning rates of the eight extra task. The performance is measured using an an independent test set not used for backpropagation or training learning rates.

TRIAL	**Before Optimization**	**After Optimization**	**Difference**
Trial 1	0.227	0.213	-6.2%
Trial 2	0.276	0.241	-12.7%
Trial 3	0.249	0.236	-5.2%
Trial 4	0.276	0.231	-16.3%
Trial 5	0.276	0.234	-15.2%
Average	0.261	0.231	-11.5% *

6.2.3 Learning Rates and How Fast the Tasks Train

Table 6.3 shows the final optimized learning rates for each of the eight extra MTL tasks in each of the five trials. Each of the learning rates was initialized to 1.0 before gradient descent.

From Table 6.2 we know that optimizing the learning rates for the extra tasks improves performance considerably on the main task. Yet no strong pattern emerges from the learning rates learned for these extra tasks in Table 6.3. (Because we are using small training sets here, considerable variation between runs is expected.) One clear pattern is that the average learning rate learned for the extra tasks is somewhat smaller than that for the main task.

Table 6.3: Learning rates learned for the eight extra tasks in 1D-ALVINN. All learning rates were initialized to 1 before training.

TASK	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
1: 1 or 2 Lanes	1.22	0.83	0.92	0.99	0.94	0.98
2: Left Edge	0.56	1.24	1.05	0.99	1.05	0.98
3: Right Edge	0.74	0.62	1.01	0.96	0.87	0.88
4: Line Center	1.43	0.58	0.91	0.51	0.98	0.88
5: Road Center	0.68	0.59	1.04	0.75	1.11	0.83
6: Road Greylevel	1.19	1.16	0.91	1.42	1.02	1.14
7: Edge Greylevel	1.02	0.89	0.83	0.64	0.90	0.86
8: Line Greylevel	0.95	0.79	0.92	1.02	0.98	0.93
Average	0.97	0.84	0.95	0.91	0.98	0.93

It also looks like the Road_Greylevel task consistently receives large learning rates, and the Road_Center task often receives low learning rates.

Examining the training curves for all the tasks as the learning rates are optimized shows that the changes in the learning rates of the extra tasks has a significant effect on the rate at which the extra tasks are learned. And, perhaps more interestingly, it also has a significant effect on the rate at which the main task is learned.

It is difficult to present learning curves for all the tasks during gradient descent on learning rates in a concise and intelligible fashion without some form of animation. Figure 6.7 (at the end of this chapter) is an attempt to summarize the effect of learning rate optimization during one of the shorter optimization runs. The horizontal axis is the number of epochs required for peak generalization performance on each task to be achieved. This is a measure of how fast tasks are training. Points towards the left side of the graph indicate tasks that trained quickly (in few epochs); points toward the right side of the graph indicate tasks that trained more slowly (required more epochs). The figure shows five snapshots during learning. The first snapshot is before optimization has begun when the learning rates are all equal to 1. The second snapshot is after 2 steps of gradient descent, the third after 3 steps of descent, etc. Each snapshot shows the learning speed of all nine tasks.

Several things are evident from looking at graphs like those in Figure 6.7. Task 8 (predicting the greylevel of the centerline) is always predicted best before the MTL net has been trained with backpropagation. This is consistent with the learning curve for this task shown in Figure 6.1 which shows that it begins overfitting immediately. This is not too surprising: the centerline is an extremely small feature in the images, so learning it is probably too difficult given the relatively small training sets used in these experiments.

The speed at which Task 9, the main steering task, reaches optimum performance changes considerably during learning rate optimization. This is surprising because the learning rate of the main task is not changed. The only way learning rate optimization can affect the main task is by affecting the hidden layer representation developing for the extra tasks, which the main task can share.

By the end of learning rate optimization, all tasks except Task 6 reach their peak performance before the main task, Task 9, even though they did not all train faster than the main task before the learning rates were optimized. We see this in most trials. In some trials, the speed of the main task (Task 9) is slowed so that the other tasks come before it, as in this trial. In other trials where the speed of the main task is not slowed, the other tasks are speeded up so that they fall before or nearer the main task. In general, it seems that learning rate optimization drives the extra tasks so that they learn roughly 1 to 2 times faster than the main task. This suggests that tasks beneficial to the main task help it most if they are learned somewhat faster than the main task, but not too much faster.

So what about Task 6, predicting the road greylevel? In all trials this task ended up being learned last. Surprisingly, if one looks at the learning rates for this task in Table 6.3, it is the one task that consistently receives a learning rate greater than 1. In the absence of other effects, one expects increasing the learning rate to make an output train faster.[2] We can only conclude that just as the learning speed of the main task can be slowed by reducing the learning rates on other tasks it shares with, Task 6 is also slowed by the reduced learning rates on the other tasks (and is, perhaps, even more sensitive to this). The increase in the learning rate for Task 6 may be an attempt by optimization to partially overcome this slowing – Task 6 would be learned even later if its learning rate were not increased.

[2]Pushing a task to train faster does not necessarily mean it will reach its peak performance faster, but we observe that it usually does in practice.

6.2.4 The Performance of the Extra Tasks

What is the performance on the eight extra tasks after the learning rates for these tasks have been optimized to maximize the performance of the main task? Is performance on the extra tasks sacrificed to obtain better performance on the main task?

Table 6.4 shows the RMS Error for the eight extra outputs before and after learning rate optimization. Performance on 7 of the 8 extra tasks also improves. This is surprising because the criterion being optimized is the performance on the main task, not the performance on any of these extra tasks.[3] What is good for the goose appears to be good for the gander. The main task is learned best if performance on the extra tasks is also improved. In this domain, learning rate optimization improves performance on the main task by adjusting the learning rates of the extra tasks so that a hidden layer representation is learned that has broad utility to most tasks in the domain. It would be interesting to see if this optimization increases sharing in the hidden layer.

Table 6.4: Performance of the MTL nets on the extra tasks before and after learning rates have been optimized to improve performance on the main task.

TASK	**Before Optimization**	**After Optimization**	**% Difference**
1: 1 or 2 Lanes	0.398	0.387	-2.8%
2: Left Edge	0.282	0.272	-3.5%
3: Right Edge	0.356	0.287	-19.4%
4: Line Center	0.459	0.381	-17.0%
5: Road Center	0.232	0.176	-24.1%
6: Road Greylevel	0.531	0.586	+10.4%
7: Edge Greylevel	0.632	0.537	-15.0%
8: Line Greylevel	1.007	0.920	-8.6%
Average	0.487	0.443	-9.0%

6.2.5 Learning Rates for Harmful Tasks

Interestingly, the only task whose performance is hurt by learning rate optimization is Task 6, the task that learns slowest after learning rate optimization, and the task that consistently gets the largest learning rates from the optimization. This raises an interesting question: is

[3] For this experiment we do early stopping on the final training run for each task individually so that we see how well each extra task is learned by the learning rates that were optimized for the main task.

Task 6 beneficial to the main task or not? Does learning rate optimization slow the rate at which Task 6 learns to mitigate conflict between Task 6 and the main task? Or, does the larger learning rate Task 6 receives suggest Task 6 is beneficial to the main task, but it is unnecessary to learn Task 6 faster than the main task for the main task to get this benefit? What does learning rate optimization do to the learning rate for extra tasks not related to the main task?

To try to answer this, we added an additional task to the 1D-ALVINN problem that consists of a noisy training signal. The training targets are random numbers uniformly distributed on an interval similar to the other training signals. Because the training signals for this extra task are random, they cannot be related to the main steering task. This is not to say that adding an extra noisy output to a backprop net might not improve performance on the main task. But in general, we do not expect extra tasks such as these to be very useful.

Table 6.5: Performance on the main Steering Direction Task when an unrelated, noisy extra task is added to the MTL net along with the other eight extra tasks, before and after learning rate optimization. The learning rates learned for the noisy extra task are shown in the third column.

TRIAL	Before Optimization	After Optimization	Noise Learning Rate
Trial 1	0.240	0.216	0.74
Trial 2	0.265	0.228	0.77
Trial 3	0.290	0.252	0.89
Trial 4	0.283	0.220	0.24
Trial 5	0.330	0.253	1.13
Average	0.282	0.234	0.76

Table 6.5 shows the performance on the main steering task before and after learning rate optimization. As expected, the average performance before optimization is worse with an extra noise task on the MTL net than without it (compare the before optimization column in Tables 6.5 and 6.2). Comparing the final performance after learning rate optimization with and without the extra noise task, however, shows that learning rate optimization is able to achieve performance with the extra noise task (Table 6.2) comparable to the performance of learning rate optimization without the extra noise task (Table 6.1). Learning rate optimization is able to mitigate much of the impact of having extra output tasks that

are harmful to the main task.

Table 6.5 also shows the learning rate learned for the noisy extra output for the same five trials used before. The average learning rate learned for the extra noise task is 0.76. This is smaller than the learning rates learned for any of the other eight tasks. This confirms our suspicion that a noisy extra task should not be as useful as a real task drawn from the domain. It also demonstrates that learning rate optimization can perform a limited kind of output task selection. We say limited because it did not drive the learning rate of the irrelevant extra task to zero. This is because backprop nets are fairly good at learning disjoint hidden layer representations for outputs that have little overlap if there is sufficient capacity in the hidden layer to do so.

6.2.6 Computational Cost

Using gradient descent to optimize learning rates can be expensive. On average, each trial computed the gradient 10 times, and performed an average of 7 line search steps with each gradient. Thus each trial required training about 150 MTL nets. For important applications (e.g., medicine, autonomous vehicle navigation) this cost is not so prohibitive as to be impractical. Other currently popular procedures for improving generalization performance such as boosting and bagging also require that many models be trained. It may be possible to amortize the cost of training the learning rates of the extra tasks by combining a form of boosting with learning rate gradient descent.

6.2.7 Learning Rate Optimization For Other Tasks

Optimizing the learning rates for different outputs is applicable to problems that may not be thought of as multitask problems. For example, classification problems with more than two classes are often trained on one net using multiple outputs, one for each class. We suspect that learning rate optimization would yield improved accuracy in these kinds of problems as well. One potentially valuable use of this approach may be when different output classes have very different frequencies. Increasing the learning rate on low frequency classes may improve the accuracy of the learned models on those classes.

6.3 Beyond Fully Connected Hidden Layers

6.3.1 Net Capacity

One might think that it would be important to keep MTL hidden layers small to promote sharing. Usually this is not correct. In our experience, limited capacity hurts MTL more than it hurts STL. It is better not to think of MTL as a way of providing additional *constraint* on what is learned, but to think of MTL as providing an *opportunity* for tasks to share what they learn. This might seem like a minor difference in point-of-view. In practice, it makes a significant difference. If one adopts the "constraint" point-of-view, one begins to apply methods that falsely assume there is a compact representation that will fully support all tasks with high accuracy. In real-world problems, the multiple tasks are often more different from each other than they are the same. Because of this, much of what is learned for each task is not useful to other tasks. When possible, it is better to allow sufficient capacity for tasks to be learned independently. In this way, sharing will only happen when there is sufficiently strong statistical correlation to cause it to happen.

6.3.2 Private Hidden Layers

Providing sufficient capacity for the main task and all extra tasks to beneficially coexist can be difficult when there are many tasks. As is discussed in Appendix 1, it is surprising how many hidden units are needed for optimal performance on even one task. We often find the optimal number of hidden units is more than 100 hidden units per output. If there are a hundred extra tasks this translates into many thousands of hidden units. This not only creates computational difficulties, but eventually can degrade performance on the main task. The reason for this is simple: if there are 10,000 hidden units, most of which have developed representations useful mainly to other tasks, the output unit for the main task has a massive feature selection problem. It must find the relatively small number of hidden units that are useful to it.

It might seem that this would limit the usefulness of MTL to situations where there are not too many extra tasks. This is not the case. Instead of using a single, very large, fully connected hidden layer shared equally by all tasks, we can develop an asymmetric architecture that is optimal for the main tasks but not for the extra tasks. This allows us to use an arbitrarily large number of extra tasks without risking swamping the main task in a sea of largely irrelevant hidden units.

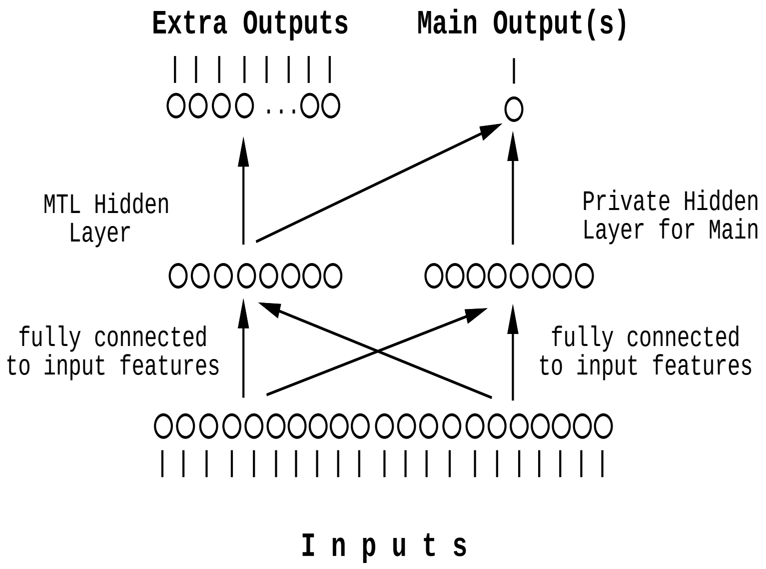


Figure 6.3: MTL Architecture With a Private Hidden Layer Used Just by the Main Task(s), and a Shared Hidden Layer Used by the Main Task(s) and the Extra Tasks. The Shared Hidden Layer can be Much Smaller than Would be Necessary for Good Performance on the Extra Tasks.

Figure 6.3 shows the simplest asymmetric net architecture that accomplishes this. Instead of one hidden layer shared equally by all tasks, there are now two disjoint hidden layers. Hidden layer 1 is a private hidden layer used only by the main task(s). Hidden layer 2 is the hidden layer shared by the main task and the extra tasks. This is the hidden layer that supports MTL transfer. This net architecture is asymmetric because the main task can see and affect the hidden layer used by the extra tasks, but the extra tasks can not see or affect the hidden layer reserved for the main tasks(s).

The size of the private hidden layer can be optimized in the usual way with STL on the main task(s). This hidden layer will probably contain hundreds of hidden units. The MTL hidden layer shared by the main and extra tasks does not need to be large enough to support optimal performance on the extra tasks. The size of this hidden layer should be optimized to provide maximum performance on the main task(s).

6.3.3 Combining MTL with Feature Nets

Feature nets is an approach to using extra tasks by learning models for them that then are used to provide extra inputs for a net learning the main task. This is a good idea. We compared feature nets with MTL on the pneumonia risk prediction domain in Section 2.3.9. Although feature nets did not work as well as MTL on this domain, we are confident that in other domains feature nets will work well, and will sometimes outperform MTL. It would be nice to have a way to combine feature nets with MTL.

Figure 6.4 shows the feature net architecture. (This figure is a copy of Figure 2.10.) It is possible to combine MTL and feature nets at two different levels. At the first level, note that in Figure 6.4 separate nets were used to learn the models for the extra tasks before the main net was trained on the main task. In other words, the models for the extra tasks were learned with STL. Instead, we can use MTL to learn the models for the extra tasks. MTL should yield better predictions on average for the extra task signals and should yield a more useful hidden layer in the MTL net learned for the extra tasks. This level is a straightforward application of MTL to feature nets.

The second level is a little more interesting. Suppose, as in Figure 6.4, there are inputs given to the main net (which is learning the main task) coming from the feature net models learned for the extra tasks. Can the main net also be an MTL net learning extra tasks as well? Yes.

Figure 6.5 shows a net that uses MTL at both levels in feature nets. Each STL net in the traditional feature net architecture (Figure 6.4) has been replaced by an MTL net. MTL feature nets uses an MTL net to learn the models for the extra tasks that are to be used as extra inputs. MTL feature nets also uses an MTL net to learn the main task; the extra outputs on this net are the same extra tasks for which predictions are being provided as extra inputs from the previous net.

One might be concerned that providing *inputs* to an MTL net that are predictions for some of the tasks that will be used as extra outputs might prevent the MTL net from learning anything interesting for those extra tasks. This is because backprop could learn direct connections between the predictions for a task provided as an input and that task's

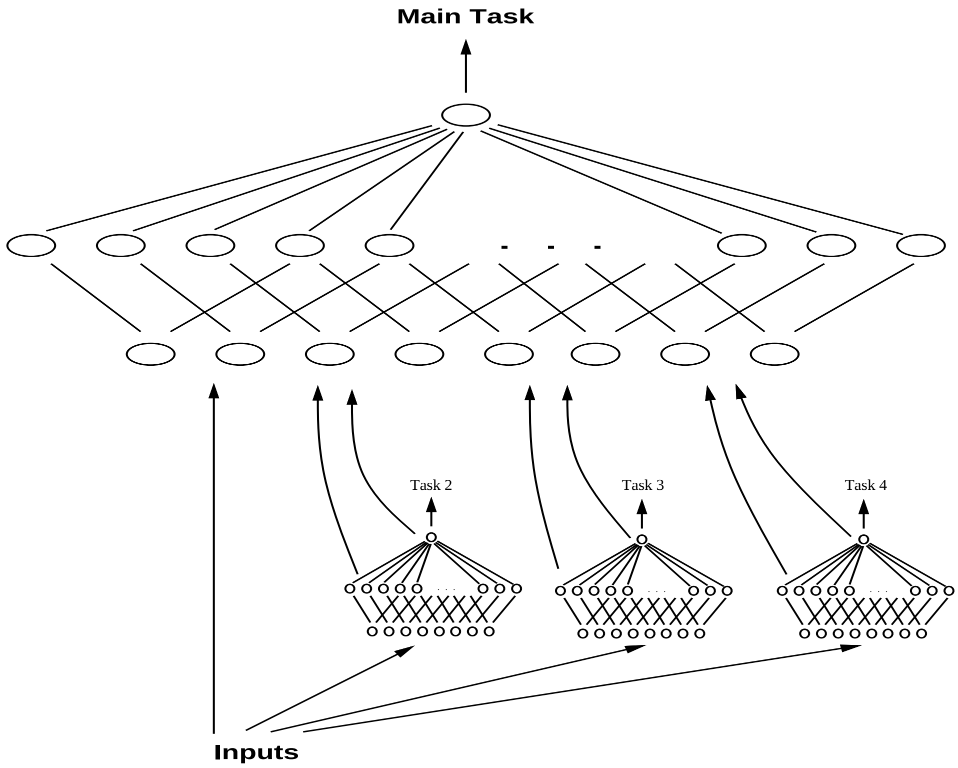


Figure 6.4: Feature Nets allows on to train a main task with STL but still benefit from what can be learned for auxiliary tasks.

output.[4] This is not as large a problem as it might seem. Consider two cases. In the first case, the predictions for the extra tasks are poor and there is room for improvement. Because the MTL net's extra outputs are training signals more accurate than the input predictions, backprop will attempt to learn models in the MTL more accurate than the predictions provided as inputs. The input predictions will probably be a strong component of those models, but the MTL net will learn more if it is possible given the information available. Moreover, the MTL net now has the possibility of using the predictions for the other extra inputs as inputs.

In the second case, the predictions for the extra task are good. Little improvement can be made on them given the data at hand. In this case the MTL net will learn models that are mainly jump connections feeding the predicted inputs through to the corresponding

[4]In Chapter 5 we presented an MTL architecture that is able to use the same task signals as both inputs and extra outputs.

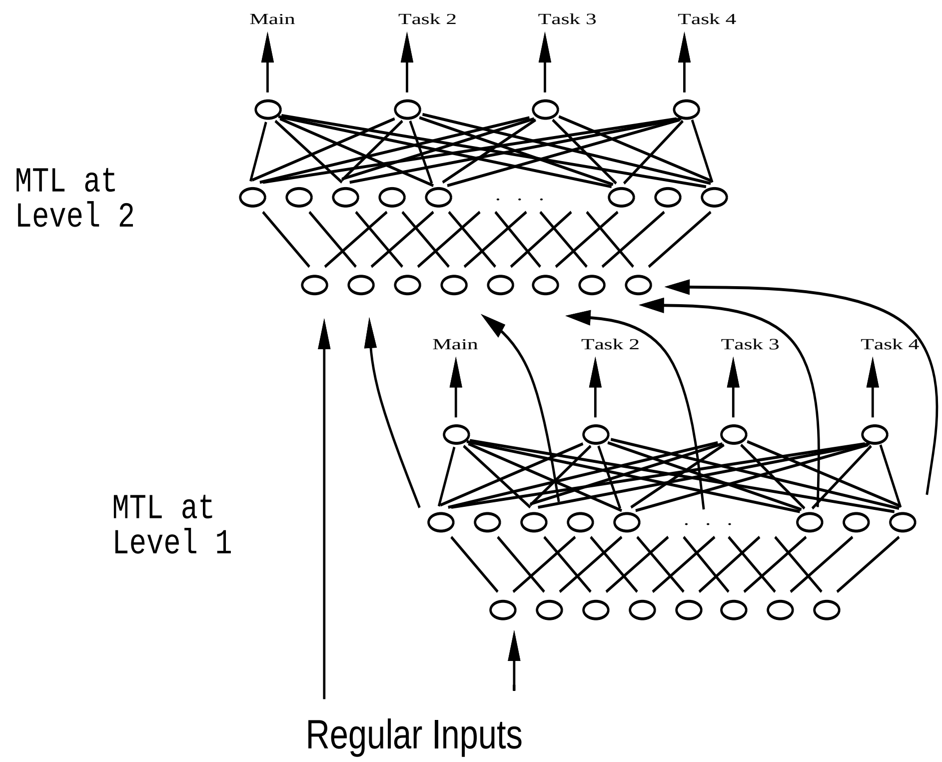


Figure 6.5: Feature Nets can be used with MTL at two different levels. Level 1: the net used to learn models for the extra tasks can be an MTL net. Level 2: the main net used to learn the main task using the regular inputs and extra task models as extra inputs can also be an MTL net. This diagram shows MTL Feature Nets where the hidden layer representation learned by the MTL net at Level 1 is passed as inputs to the MTL net at Level 2. It is also possible to pass the output predictions of the MTL net at Level 1 to the net at Level 2. Usually, passing the hidden layer activations works better than passing the output predictions. (We have not tried passing both.)

outputs. This is good, however. If we assume the main task could be learned better if the extra tasks could be provided as extra inputs, the MTL net learning the main task now has the opportunity to use high-fidelity predictions for the extra tasks as inputs, and thus will perform well.

In summary, MTL feature nets can provide the advantages of both MTL and feature nets. MTL helps better models be learned for the extra tasks. It also provides the MTL bias for the main task when it is being learned. And it allows the main task to benefit from using high-fidelity predictions for the extra tasks as inputs when these are possible.

Figure 6.6 shows the performance on the pneumonia problem using MTL nets for the

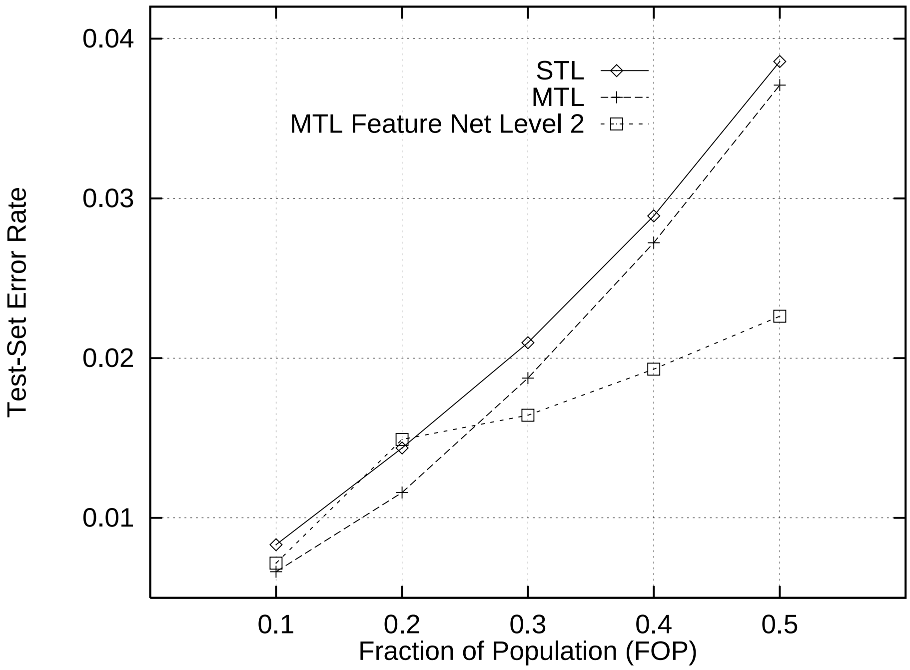


Figure 6.6: Performance of Feature Nets with MTL applied at both levels.

nets at both levels in feature nets. Performance improves considerably at the larger FOPs, but may be worse than MTL alone at the lowest FOPS. We do not know if the data point for FOP 0.2 is representative or not.

6.4 Chapter Summary

The goal in MTL is to learn the main task better by taking advantage of the information contained in the training signals of the extra tasks. Sometimes the main task and extra tasks can be treated equally. Often, however, better performance on the main task can be achieved by employing methods that favor performance on the main task possibly at the expense of worse performance on the extra tasks.

In Section 6.1 we saw that even if tasks are trained as equal outputs on an MTL net, it is important to do early stopping for the main task without considering the error on other outputs. Note that this is not the usual way of doing early stopping in the machine learning community. Typically, net training is halted when the total error measured across all outputs reaches a minimum on the test-set. This is almost certainly an inferior way to train any net that has multiple outputs. Better performance can usually be achieved on each output by halting training on each output individually, and using the different snapshots acquired this way to make future predictions for those different outputs. This is true even if one does not think of the learning problem as a multitask problem. If there are multiple outputs, it *is* a multitask problem.

In Section 6.2 we saw that training all output tasks on an MTL net the same way probably does not yield optimal performance on the main task. We used optimization to find the learning rates for the extra tasks that yielded the best performance on the main task. Although the gradient descent procedure we used is expensive, this approach should be practical for most problems having 25 or fewer tasks. For these problems the benefit can be substantial. Tuning the learning rates yielded a 10% further reduction in error for MTL in the 1D-ALVINN domain. In domains where there are many extra tasks, applying gradient descent to the learning rate of each extra task probably will not be practical. In these domains, one approach is to group extra tasks into clusters that all receive the same learning rate, thereby reducing the number of learning weights that need to be optimized. The simplest way to do this is to use one learning rate for all extra tasks and to optimize this single parameter so as to maximize performance on the main task. One conclusion we drew from the experiments with learning rate optimization is that extra tasks seem to benefit the main task most if they are learned just before the main task. This suggests another simple approach to tuning learning rates: increase the learning rates on all tasks that are learned after the main task until they are learned at a rate similar to the main task. This is a fairly simple optimization problem that is much more tractable when there are many extra tasks.

In Chapter 5 we used feature selection to find features that might be used as outputs because they are not useful as inputs. In Section 5.3 we showed that some features are so useful as extra outputs that it is worth using an architecture that allows these features to be used as both inputs and as outputs. Both of these approaches beg the question: how can we tell what extra tasks are useful?

Even if we use some task selection method to find useful extra outputs, there may still be hundreds or thousands of extra outputs to train. This creates two problems. First, training a hidden layer large enough to support both the main tasks and extra tasks may be prohibitively costly. Second, even if one could afford to train this large hidden layer, the number of hidden units dedicated to the extra tasks will eventually grow so large that the main task output will suffer from an internal "representation" selection problem that will hurt its performance. To avoid these problems, in Section 6.3.2 we present an architecture that uses a private hidden layer for the main task and a public hidden layer shared by the main tasks and the extra tasks. The public hidden layer is not large enough to support optimal performance on the extra tasks. Keeping the public hidden layer small promotes generalization in what is learned for the extra tasks, insures that there will not be so many hidden units dedicated to extra tasks that the main task is unable to do selection at the hidden layer, and makes the total number of weights that must be trained much smaller.

In Section 6.3.3 we examined another architecture that expanded the usefulness of MTL. Here we combined feature nets with multitask learning. This architecture is our final example of treating the main task differently than the extra tasks. One reason why we present this architecture last is because it demonstrates an important point about MTL nets: an MTL net can be employed *anywhere* an STL net can be used. The extra tasks on the MTL net are there only to improve performance on the main task. Because the MTL net has the same inputs as the STL net, and because the extra outputs on the MTL net can be ignored after the net is trained, MTL nets are functionally equivalent to STL nets in every way, except that they often generalize better.

```
.........................................................................step.0
1 or 2 lanes    |     1                                                       |
Left Edge       |     2                                                       |
Right Edge      |     3                                                       |
Line Center     |    4                                                        |
Road Center     |           5                                                 |
Road Greylevel  |                                            6                |
Edge Greylevel  |                      7                                      |
Line Greylevel  |8                                                            |
Steering (main) |       9                                                     |
.........................................................................step.2
1 or 2 lanes    |                1                                            |
Left Edge       |       2                                                     |
Right Edge      |       3                                                     |
Line Center     |     4                                                       |
Road Center     |       5                                                     |
Road Greylevel  |                                                            6|
Edge Greylevel  |                             7                               |
Line Greylevel  |8                                                            |
Steering (main) |       9                                                     |
.........................................................................step.3
1 or 2 lanes    |                1                                            |
Left Edge       |              2                                              |
Right Edge      |       3                                                     |
Line Center     |       4                                                     |
Road Center     |       5                                                     |
Road Greylevel  |                                            6                |
Edge Greylevel  |                                  7                          |
Line Greylevel  |8                                                            |
Steering (main) |                                   9                         |
.........................................................................step.4
1 or 2 lanes    |                1                                            |
Left Edge       |              2                                              |
Right Edge      |        3                                                    |
Line Center     |       4                                                     |
Road Center     |              5                                              |
Road Greylevel  |                                        6                    |
Edge Greylevel  |                             7                               |
Line Greylevel  |8                                                            |
Steering (main) |                                     9                       |
.........................................................................step.5
1 or 2 lanes    |                                1                            |
Left Edge       |              2                                              |
Right Edge      |        3                                                    |
Line Center     |       4                                                     |
Road Center     |              5                                              |
Road Greylevel  |                                                        6    |
Edge Greylevel  |                                7                            |
Line Greylevel  |8                                                            |
Steering (main) |                                   9                         |
...............................................................................
                0      200000      400000      600000      800000      1000000
                   Pattern Presentations (epochs x 100) To Peak Performance
```

Figure 6.7: Task Learning Speeds During Learning Rate Optimization

Chapter 7

MTL in K-Nearest Neighbor

7.1 Introduction

Multitask Learning is an inductive transfer method that improves generalization by learning extra tasks in parallel with the main task while using a shared representation; what is learned for the extra tasks can help the main task be learned better. Previously we demonstrated multitask learning in backprop nets. In backprop MTL, the representation used for multitask transfer is a hidden layer shared by all tasks. Is multitask learning useful only in artificial neural nets?

Many learning methods do not have a representation naturally shared between tasks. Can MTL be used with methods like these? Yes. This chapter shows that multitask learning can be used with case-based methods such as k-nearest neighbor and kernel regression that do not have a built-in means of sharing a representation between multiple tasks. The approach we follow is to use an error metric that combines performance on the main task with a weighted contribution of the performance on the extra tasks. This causes models to be learned that perform well on both the main task and the extra tasks.

We demonstrate this approach to multitask transfer on the same pneumonia risk prediction domain used with MTL in backprop nets. As we will see, MTL reduces the error of kernel regression 5–10% on this problem. We also introduce *soft ranks*, a ranking procedure that makes rank-based error metrics differentiable and thus more amenable to gradient search.

7.2 Background

The basic assumption underlying most machine learning methods is that similarity in feature space correlates with similarity in prediction space. K-nearest neighbor and kernel regression explicitly use this heuristic for prediction. They search the training set for cases most similar to the new case, and return as a prediction for the new case the class (or value) of those most similar cases.

Similarity is usually defined by a distance metric computed over case features. The distance metric most commonly used is weighted Euclidean distance:

$$DISTANCE(c1, c2) = \sqrt{\sum_{i=1}^{NO\_FEATS} WEIGHT_i * (FEAT_{i,c1} - FEAT_{i,c2})^2}$$

where $c1$ and $c2$ are two cases, NO_FEATS is the dimensionality of feature space, $FEAT_{i,c}$ is feature i for case c, and $WEIGHT_i$ is a weight for each feature dimension that controls how important that dimension is to the distance calculation. In simple *unweighted* Euclidean distance, $\forall i$, $WEIGHT_i = 1.0$.

7.2.1 K-Nearest Neighbor

K-nearest neighbor (KNN) searches the training set for the K cases closest to the new case. In classification problems, KNN returns as its prediction for the new case either the predominant class of the K nearest neighbors, or a probability for each class estimated from the number of nearest neighbors in each class:

$$PROBABILITY(C) = \frac{1}{K} \sum_{i=1}^{K} CLASS(i = C)$$

where K is the number of neighbors, $PROBABILITY(C)$ is the predicted probability that the new case belongs in class C, and $CLASS(i = C)$ is an indicator equal to 1 if case i is a member of class C, and 0 otherwise. The predominant class is the class with the highest predicted probability.

In regression, KNN returns the average of the K nearest neighbors:

$$PREDICTED\_VALUE = \frac{1}{K} \sum_{i=1}^{K} VALUE_i$$

where $PREDICTED\_VALUE$ is the regression value KNN will predict for the new case, and $VALUE_i$ is the value of training case i.

The best value of K to use depends both on the structure of the problem space and the density of samples. If K is too large, KNN becomes insensitive to the fine structure of the problem because the neighborhoods are too large. If K is too small, predictions become noisy as they are based on smaller samples than necessary to capture the problem's structure. Good values for K are usually found via cross validation.

7.2.2 Locally Weighted Averaging

Locally weighted averaging (LCWA, also known as kernel regression) is similar to KNN in that it uses a distance metric to determine how similar the new case is to each case in the training set. This distance is used to weight the contribution of each training case to the prediction for the new case, training cases closest to the new case having the largest effect. In classification problems, the weight is used to accumulate the probability that the new case belongs to each class:

$$PROBABILITY(C) = \frac{\sum_{i=1}^{NO\_TRAIN} CLASS(i = C) * e^{-\frac{DISTANCE(c_i, c_{new})}{KERNEL\_WIDTH}}}{\sum_{i=1}^{NO\_TRAIN} e^{-\frac{DISTANCE(c_i, c_{new})}{KERNEL\_WIDTH}}}$$

where NO_TRAIN is the number of cases in the training set, $DISTANCE(c_i, c_{new})$ is the distance between case c_i in the training set and the new case measured using some distance metric, $CLASS(i = C)$ is an indicator as before, the numerator accumulates the predictions of the training cases, weighted inversely by an exponentially decreasing function of the distance relative to the $KERNEL\_WIDTH$, and the denominator is a normalization factor.

In regression problems, distance is used to weight how much the value of each training case adds to the predicted value:

$$PREDICTED\_VALUE = \frac{\sum_{i=1}^{NO\_TRAIN} VALUE_i * e^{-\frac{DISTANCE(c_i, c_{new})}{KERNEL\_WIDTH}}}{\sum_{i=1} NO\_TRAIN e^{-\frac{DISTANCE(c_i, c_{new})}{KERNEL\_WIDTH}}}$$

Predictions made with LCWA are affected by all cases in the training set, though cases far away have little effect. Where K controls the size of the neighborhood used for prediction in KNN, the $KERNEL\_WIDTH$ controls the scale of the neighborhood that has most effect in LCWA. Cases closer to the new case than $KERNEL\_WIDTH$ have significant impact on the prediction for that case, cases further away than $KERNEL\_WIDTH$ have impact that falls off exponentially with distance.

7.2.3 Feature Weights and the Distance Metric

The performance of KNN and LCWA depends on the quality of the distance metric. Simple Euclidean distance (all feature weights equal to 1.0) often works reasonably well, but is usually suboptimal. Finding good feature weights is essential to optimal performance. Search for good feature weights can be cast as an optimization problem using cross validation. Leave-one-out cross validation (LOOCV) is particularly efficient and easy to implement in KNN and LCWA: remove each case from the training set one at a time, and use the remaining $N-1$ cases as the pool from which to find neighbors for prediction.

Gradient descent and LOOCV can be used to search for good feature weights. Feature weights are initialized to some starting value. The LOOCV performance of these initial weights are calculated, yielding an estimate of their generalization performance. The gradient of the LOOCV performance with respect to the feature weights is calculated, either analytically or by numerical approximation.[1] A step is taken along the negative gradient. This step changes the feature weights, usually by a small amount. If the updated feature weights yield improved performance as measured by a new LOOCV on the training set, the step is accepted. If not, the step is rejected and the step size is reduced. When a step is accepted, the gradient at the new point is computed and the process repeats. Search terminates when a local minimum in LOOCV performance is found. Local minima are detected when the step size needed to improve LOOCV performance becomes very small.

Because this search repeatedly uses the same training set for optimization, overfitting to the training set is likely, particularly if the training set is small. To protect against this, a separate test set is used to determine when to halt training. This *halt* set is not used to calculate gradients, but performance on the halt set (using the training set as the case database) is watched during gradient search to determine when overfitting occurs. Gradient search is terminated when performance on the halt set appears to be getting worse.

[1] One advantage of LCWA over KNN is that because LCWA predictions use all cases, LCWA performance is differentiable with respect to the feature weights. KNN is usually discontinuous when cases move in and out of the K nearest neighborhoods. There are modifications to KNN that make it differentiable, or search can be done using an optimization method that does not require differentiability.

7.3 Multitask Learning in KNN and LCWA

Finding good feature weights is essential to optimal performance with KNN and LCWA. MTL can be used to find better weights. In KNN and LCWA, the distance metric (i.e., the feature weights) is what we will share between tasks.[2] The assumption is that feature weights appropriate to one task drawn from the domain will tend to be appropriate to other tasks in the domain, so finding feature weights that perform well on all tasks should, on average, improve performance on each task.[3] The basic approach is to find feature weights that yield good performance on both the main task and a set of related tasks drawn from the same domain. The approach we follow is to use an error metric that combines performance on the main task with a weighted contribution of the performance on the extra tasks. This causes models to be learned that perform well on both the main task and the extra tasks.

It is possible for many extra tasks to swamp the error signal of the main task if the extra tasks are too dissimilar. To prevent this, we weight the contribution of the extra tasks in a combined error metric. The degree to which each extra task t affects the evaluation function is controlled by weights λ_t:

$$Eval\_Metric = Perf\_Main\_Task + \sum_{t=1}^{NO\_TASKS} \lambda_t * Perf\_Task_t$$

where $Eval\_Metric$ is the criterion being minimized via gradient descent on the feature weights, $Perf\_Main\_Task$ is the performance on the main task, $Perf\_Task_t$ is the performance on extra task t, and λ_t is a nonnegative weight. $\lambda_t = 0$ causes learning to ignore the extra task t, $\lambda_t \approx 1$ causes learning to give as much weight to performance on the extra task t as to the main task, and $\lambda_t \gg 1$ causes learning to pay more attention to performance on the extra tasks than to the main task.

In this chapter we consider only the case where all λ_t take on the same value, λ. We do this to reduce the computational cost of the experiments we run and to simplify the presen-

[2]Other things such as K and the $KERNEL\_WIDTH$ can also be shared for MTL.

[3]This is the same assumption made in [O'Sullivan & Thrun 1996]. There, feature weights learned for a previous task are used for a new task when the number of samples for the new task is too small to support learning. This approach differs from MTL where the goal is to learn better feature weights by learning all available extra tasks in parallel. Learning one set of feature weights for multiple tasks and using feature weights learned separately for multiple independent tasks are not the same thing.

tation of results. We have run experiments where each λ_t is permitted to vary individually, using gradient descent on the halt set to determine the optimal task weights. Although this sometimes outperforms using a single λ for all extra tasks, in the experiments we have run so far it does not appear to perform so much better that it is worth the extra complexity it would add to the discussion here. Also, overfitting appears to be more of a problem when separate λ_t values are used for each extra task.

To further simplify the presentation, we've scaled *lambda* to the interval [0,1]. Now,

$$Eval\_Metric = (1 - \lambda) * Perf\_Main\_Task + \sum_{t=1}^{NO\_TASKS} \lambda * Perf\_Task_t$$

When $\lambda = 0$, all weight is given to the main task and the extra tasks are completely ignored. This is traditional single task learning (STL). When $\lambda = 1/2$, equal weight is given to the main task and to *each* extra task. This is multitask learning where all tasks have comparable weight. When $\lambda = 1$, all weight is given to the extra tasks and the main task is ignored.

7.4 Pneumonia Risk Prediction (review)

We demonstrate MTL in KNN and LCWA using the Medis pneumonia risk prediction task used for MTL backprop in Chapter 2. In this problem, the primary goal is to identify patients at high risk from pneumonia so they may be hospitalized to receive aggressive testing and treatment. Some of the most useful tests for predicting pneumonia risk usually require hospitalization and will be available only if preliminary assessment indicates further testing and hospitalization is warranted. But low-risk patients can often be identified using measurements made prior to hospitalization.

The Medis Pneumonia Database [Fine et al. 1993] indicates whether each patient lived or died. 1,542 (10.9%) of the patients died. As before, the most useful decision aid for this problem would predict which patients will live or die. But this is too difficult. In practice, the best that can be achieved is to estimate a probability of death (POD) from the observed symptoms. In fact, it is sufficient to learn to *rank* patients by POD so lower risk patients can be discriminated from higher risk patients. Patients at least risk may then be considered for outpatient care.

As before, the criterion used to evaluate learning is the accuracy with which one can select a fraction of the patients that do not die. For example, given a population of 10,000 patients, find the 20% of this population at *least* risk. To do this we learn a risk model and a model threshold that allows 20% of the population (2000 patients) to fall below it. If 30 of the 2000 patients below this threshold die, the error rate is $30/2000 = 1.5\%$. We say that the error rate for FOP 0.20 is 1.5% ("FOP" stands for fraction of population). We consider FOPs 0.1, 0.2, 0.3, 0.4, 0.5, and 0.6. The goal is to learn models that minimize the error rate at each FOP.

The Medis database contains results of 33 lab tests that will be available only after patients are hospitalized. These results will not be available when the model is used because the tests will not yet have been ordered. Previously, we used MTL in backpropagation to benefit from these future lab results. The extra lab values were used as extra tasks (extra outputs) on a backprop net learning the main risk prediction task. The extra tasks biased the shared hidden layer to learn representations that yielded better performance on the main risk prediction task. Here we use the same extra tasks to demonstrate MTL in LCWA.

7.5 Soft Ranks

It is difficult to directly learn models that minimize error rates at FOPs between 0.1 and 0.6. Not only are there six different criteria, but error rates measured on FOPs are discrete because lives/dies is boolean. Discrete metrics cannot be used with gradient descent.

In the neural net solution to the pneumonia problem we devised an error metric and training procedure called *rankprop* that learned to predict ranks of the data. Rankprop can be adapted to KNN and LCWA. Rankprop, however, requires rank models be learned gradually with many epochs of learning interleaved with the repeated re-rankings performed internal to the Rankprop algorithm. (See Appendix B for the Rankprop algorithm.) This is not an issue with backpropagation which usually trains *very slowly*. But in our experience, feature weights in KNN and LCWA can be trained much faster. It would be unfortunate to slow KNN and LCWA learning by using a very small gradient step size just so that rankprop is given the opportunity to frequently re-rank the training data. This led us to search for another way to learn ranks with KNN and LCWA. The result is the *soft rank sum*, which is based on a generalization of standard discrete ranks that we call *soft ranks.* This modification gives ranks a continuous flavor, making it easy to create differentiable error metrics based on the ranks.

Qualitatively, soft ranks behave like traditional ranks, but have the nice additional property that they are continuous: small changes to item values yield small changes in the soft ranks. Moreover, if small changes in the values cause items to swap positions with neighboring items, the soft ranks reflect this in a smooth way. See Appendix 1 for more detail about soft ranks.

7.6 The Error Metrics

The main prediction task is mortality risk. KNN and LCWA are used to predict the risk of each new case by examining whether its neighbors in the training set lived or died. Predicted cases are then sorted by this predicted risk. The optimization error metric we use here for this task is the sum of the soft ranks for all patients in the sample *who live.* The goal is to order patients by risk, least risk first. Successfully ordering all patients that live before all patients that die minimizes this sum. We scale the sum of soft ranks so that 0 indicates all patients who live have been ranked with lower risk than all patients who die. This is ideal performance. The scaling is done so that a soft rank sum of 1 indicates that all patients who die have been ranked with lower risk than all patients who live. This is maximally poor performance. Random ordering of the patients yields soft rank sums around 0.5. Good performance on this domain requires soft rank sums less than 0.05.

The extra tasks include tasks such as predicting the white blood cell count and the partial pressure of oxygen in the blood. The error metric for the extra tasks is the standard sum-of-squares error (SSE). Note that extra tasks are not used to predict the risk sort order of patients—we do not know what values (or combinations of values) of the extra tasks raise or lower risk. The soft rank sum is computed only for the main mortality prediction task using only the KNN and LCWA predictions for mortality. Learning to predict the SSE of the extra tasks is useful only if it helps learn feature weights that improve performance on the main risk task.

7.7 Empirical Results

We've run experiments using KNN and LCWA on pneumonia. With small to medium size training sets, LCWA consistently outperforms KNN. The reason for this is simple. Roughly 90% of the patients survive, and even patients at high risk have a relatively low probability of death (POD). If K is small, most KNN neighborhoods will include few (if any) deaths. This means the predicted POD for most cases will be $0/K$, with a few PODs of $1/K$, fewer PODs of $2/K$, etc. A small number of discrete PODs does not support useful ranking of patients. Using large K reduces this problem, but sensitivity to fine structure in the problem is then lost. Because LCWA accumulates contributions from *all* cases in the training set, it can make small distinctions between cases without being forced to gloss over fine structure, and thus provides a better basis for predicting relative risk. Because of this we report only the LCWA results. *The results of using MTL with KNN are qualitatively similar. MTL is as applicable to KNN as to LCWA.*

7.7.1 Methodology

In this chapter we report the results of three large experiments using MTL in LCWA with soft ranks. In these experiments, the goal is to find feature weights that yield good performance on the main task: predicting relative mortality risk from pneumonia. The three experiments use similar methodology. The original dataset of 14,199 cases is randomly subsampled to create learning and test sets. The learning set is then split into equally sized training and halt sets. The training set is used for gradient descent on the feature weights, and the halt set is used to halt gradient descent on the feature weights when overfitting begins. The remaining cases in the test set are used as an independent test set to evaluate the performance of the learned feature weights.

Caution: be careful not to confuse feature weights with task weights. Feature weights are what will be trained with gradient descent. They determine the distance metric used to find neighbors. Task weights control how much each task contributes to the error criterion optimized by gradient descent. In this chapter, all task weights are controlled by a single parameter λ set via cross-validation.

The feature weights for the 30 input features are initialized to 1.0. Gradient descent with line searches is done using LOOCV on the training set. We use $KERNEL\_WIDTH = 1.0$, a value preliminary experiments indicated performed well on this problem. The $KERNEL\_WIDTH$ is not trained via gradient descent because training the feature weights allows the distance metric to adjust the effective $KERNEL\_WIDTH$. During learning, performance at each gradient step is evaluated on the halt set. Because halt set performance is not always monotonic, premature stopping is a potential problem. To prevent this, gradient descent is run for a large, fixed number of steps, and the feature weights yielding best performance on the halt set are found by examining the entire training curve. The learned weights are then evaluated on the independent test set.

7.7.2 Experiment 1: Learning Task Weights with MTL

The first experiment examines how much attention learning should pay to the extra tasks. Is it better to ignore the extra tasks and optimize performance only on the main task, or is better performance on the main task achieved by optimizing performance on all the tasks?

500 Training Cases

Figure 7.1 shows the mean rank sum error on the independent test sets for 50 trials of learning as a function of λ for training and halt sets containing 500 patterns each. Each data point has error bars indicating the standard error of the mean. In the graphs and tables, the performance shown is for the main mortality risk prediction task.

The horizontal line at the top of the graph is the performance of LCWA when all feature weights are 1.0, before any training has been done. This is LCWA using simple unweighted Euclidean distance. Because there are no weights to train (i.e., no weight learning), this performance is independent of λ; no multitask learning has yet taken place.

The lower curve in Figure 7.1 shows how varying λ affects MTL performance. All points on the curve have better performance than the the horizontal line representing untrained weights. Unweighted Euclidean distance (feature weights all equal to 1.0) performs worse

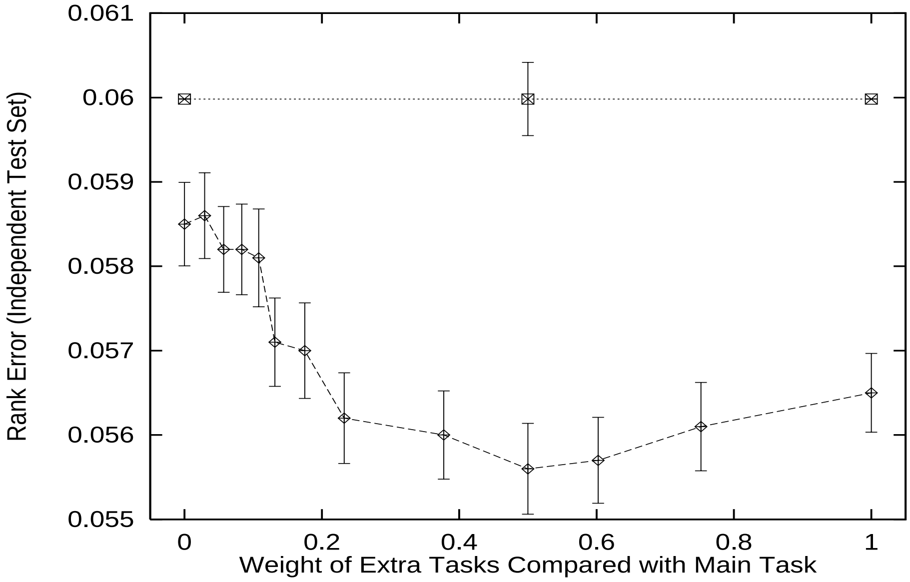


Figure 7.1: Rank Sum Error as a function of MTL's λ (Train = 500 cases; Halt = 500 cases). $\lambda = 0$ is STL; the extra tasks are ignored. The horizontal line near the top of the graph is the performance before feature weights are trained. This is the performance of simple, unweighted Euclidean distance.

than feature weights trained with gradient descent.

The point at $\lambda = 0$ is the performance of LCWA trained with traditional single task learning (STL); the extra tasks are completely ignored by gradient descent when $\lambda = 0$. STL training of the feature weights reduces rank sum error about 5% compared with weights equal to 1.0.

The points for $\lambda > 0$ correspond to multitask learning. Larger values of λ give more weight to the extra tasks.[4] From the graph it is clear that learning yields better performance on the main task if it searches for feature weights that are good for both the main task and for the extra tasks. MTL does not use any extra training patterns, it uses extra training signals in each pattern. The optimal value of λ is between 0.2–0.8. Interestingly, these λs give similar weight to the main task and each extra task. At $\lambda = 0.5$ all tasks—*including the main task*—are given equal weight. At the optimal λ, MTL reduces error another 5–10%. The improvement of MTL over STL is larger than the improvement of STL with trained

[4]The vertical axis shows performance on only the main risk prediction task. The contribution of the extra tasks to the error metric that gradient descent is optimizing is not shown—though its effect on what is learned should be apparent.

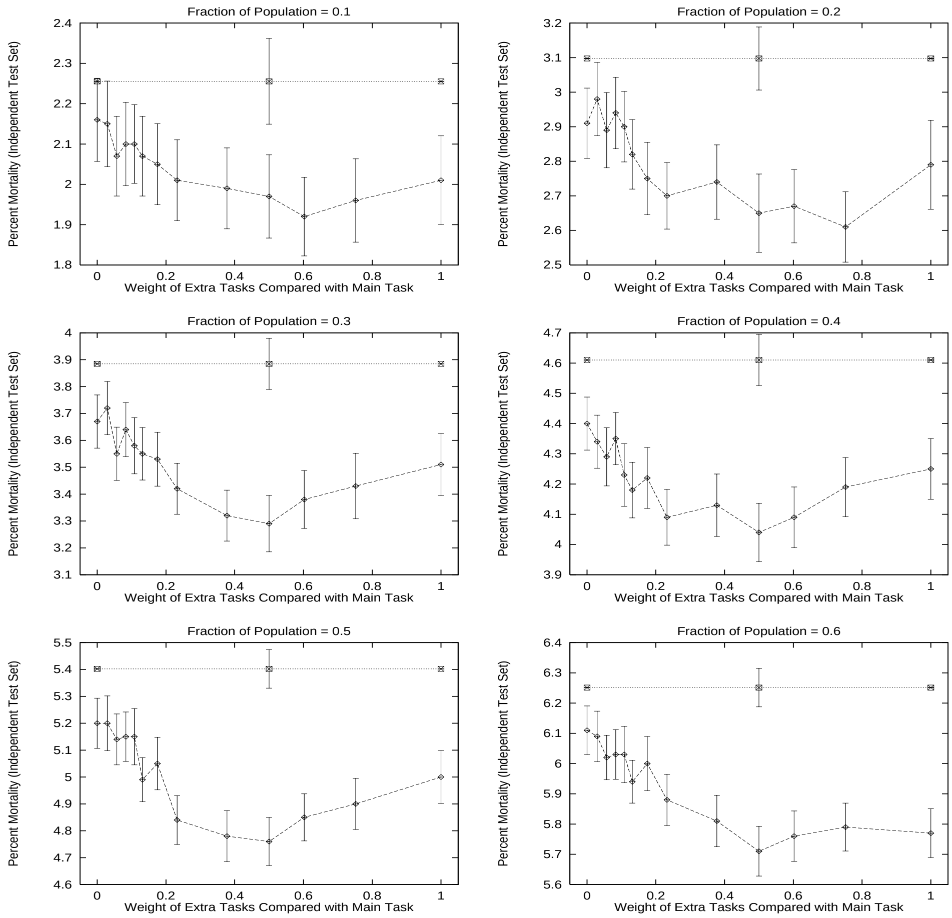


Figure 7.2: Performance on Different Fractions of the Population as λ Varies (Train = 500; Halt = 500)

weights over STL with untrained weights.

Figure 7.2 shows the performance of MTL as a function of λ for the six different FOP metrics.[5] The shape of the FOP curves is qualitatively similar to the Soft Rank Sum curve in Figure 7.1.

Table 7.1 summarizes the performance of STL and MTL LCWA at $\lambda = 0.5$. $\lambda = 0.5$ is not necessarily the optimal value. We choose $\lambda = 0.5$ because performance is good at this value, because performance appears to be flat for λ on both sides of this value, and because

[5]Because the 20% FOP contains the cases in the 10% FOP, the graphs are not statistically independent. However, because more errors (deaths) occur later in an FOP (when learning is working well), the graphs are more independent than the population fractions suggest. ROC curves would probably be more appropriate. We use FOP metrics for compatibility with previous studies using this database.

it represents the point where all tasks are given equal weight. Differences marked with "*", "**" and "***" are statistically significant at 0.05, 0.01, and 0.001 or better, respectively. MTL with $\lambda \approx 0.5$ reduces FOP errors 5–10% compared with STL.

Table 7.1: Error rates of STL LCWA and MTL LCWA ($\lambda = 0.5$) on the pneumonia problem using train and test sets with 500 cases each.

FOP	0.1	0.2	0.3	0.4	0.5	0.6	SoftRankSum
STL LCWA	.0216	.0291	.0367	.0440	.0520	.0611	.0585
MTL LCWA	.0197	.0265	.0329	.0404	.0476	.0571	.0556
% Change	-8.8%	-8.9%	-10.4% **	-8.2% **	-8.5% ***	-6.5% ***	-5.0% ***

1000 Training Cases

Figures 7.1 and 7.2 show the performance with training and halt sets sets containing 500 patterns each. In the backprop MTL experiments with this domain in Chapter 2 we used train and halt sets containing 1000 patterns each. Figures 7.3 and 7.4 and Table 7.2 show the performance of STL and MTL LCWA with 1000 cases in the train and halt sets. The results of MTL LCWA with training and halt sets of size 1000 are qualitatively similar to those of size 500; the main difference is that performance improves considerably with the larger training sets. The improvement due to doubling the size of the training sets is larger than the improvement due to using MTL instead of STL with the smaller training sets. (We return to this issue in Section 7.7.4.)

Table 7.2: Error rates of STL LCWA and MTL LCWA ($\lambda = 0.5$) on the pneumonia problem using train and test sets with 1000 cases each.

FOP	0.1	0.2	0.3	0.4	0.5	0.6	SoftRankSum
STL LCWA	.0147	.0216	.0285	.0364	.0447	.0540	.0530
MTL LCWA	.0141	.0196	.0263	.0343	.0425	.0522	.0516
% Change	-4.1%	-9.3%	-7.7% *	-5.8% *	-4.9% *	-4.0%	-2.6% **

7.7.3 Taking Full Advantage of LCWA

Figures 7.3 and 7.4 showed the performance of LCWA using 1000 patterns in the training and halt sets. Because KNN and LCWA use the training set itself to make predictions, they do not need to be "retrained" if new training data becomes available. We can take

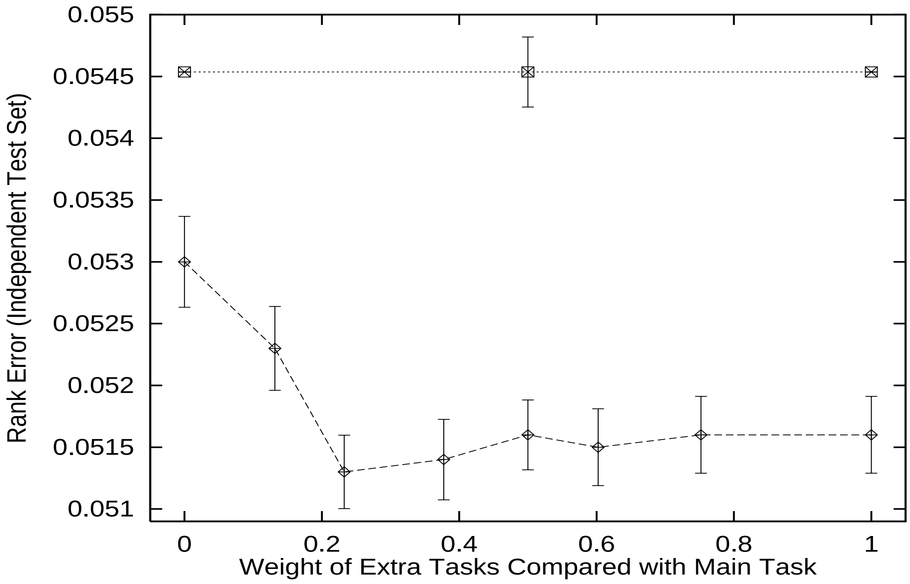


Figure 7.3: Rank Sum Error as a Function of λ During MTL (Train = 1000; Halt = 1000)

advantage of this to make better use of the data in the halt sets. As before, train the feature weights using gradient descent on the training set, and stop training when performance on the halt set starts to get worse. Then, instead of discarding the halt set, add the cases in the halt set to the training set before making predictions on unseen instances. In our experiments, this doubles the number of cases used to make predictions. Although the feature weights were trained with gradient descent on only half the data, KNN and LCWA will almost certainly perform better when the learned feature weights are used with more data.[6]

Figures 7.5 and 7.6 and Table 7.3 show the performance of LCWA using 1000 patterns in the train and halt set, but when the halt set is added to the case base before making predictions on the test set. The feature weights are trained using 1000 cases, and training is halted using another 1000 cases, but final predictions are made using 2000 cases. Using the halt set this way improves performance considerably. MTL still improves performance,

[6]It would be difficult to use a backprop halt set this same way. With backprop, one would need to retrain the neural net model from scratch using both the training and halt set patterns. Not only is this expensive, but, because learning with the extra patterns might follow a different trajectory, it is not clear that halting the net trained on both sets of data at the same number of epochs used to halt the net trained on just the training set is a reliable procedure.

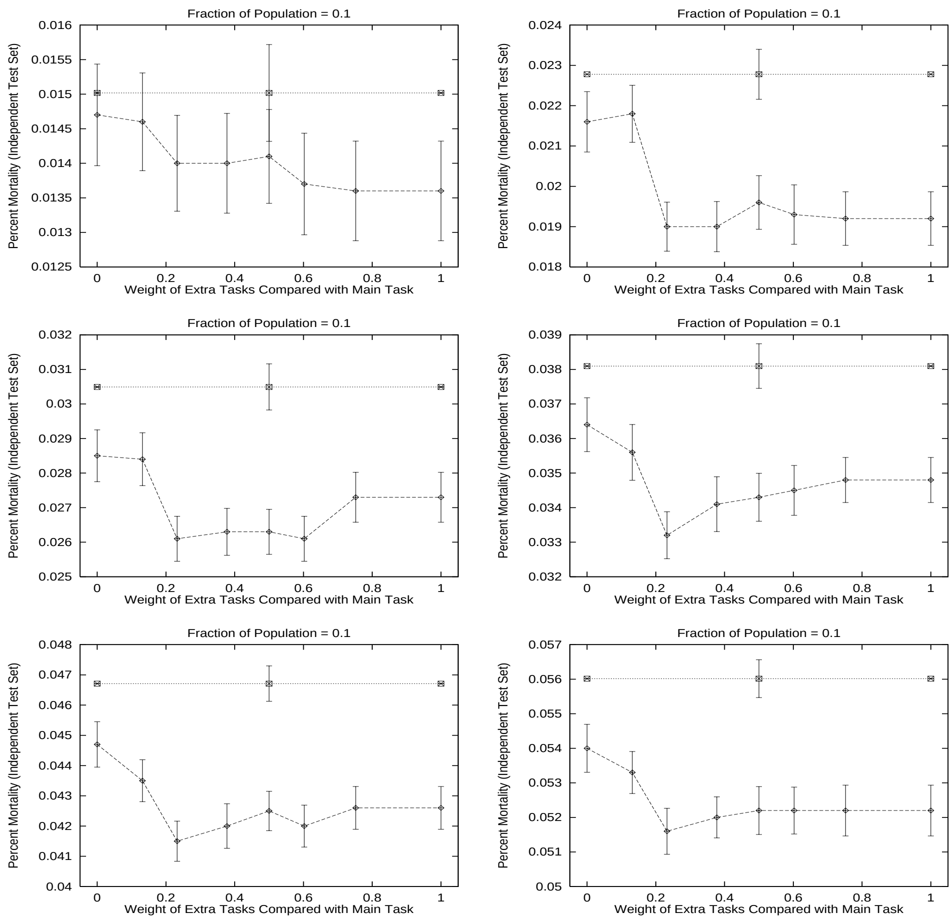


Figure 7.4: Performance on Different Fractions of the Population as λ Varies (Train = 1000; Halt = 1000)

though with this many training patterns there is less room for improvement than there was with fewer patterns.

Summary of Experiment 1

Experiment 1 examined the benefit of MTL as a function of λ. For values of $\lambda \approx 0.5$, MTL outperforms STL 2–15%, depending on the criterion and sample size. MTL does this without having access to any additional cases. It merely has more information available in each case. MTL uses the extra information in the cases in a way that does not require that information to be available for future test cases. It uses the extra feature values as extra tasks, not as extra inputs. As extra tasks, the extra information is used solely to

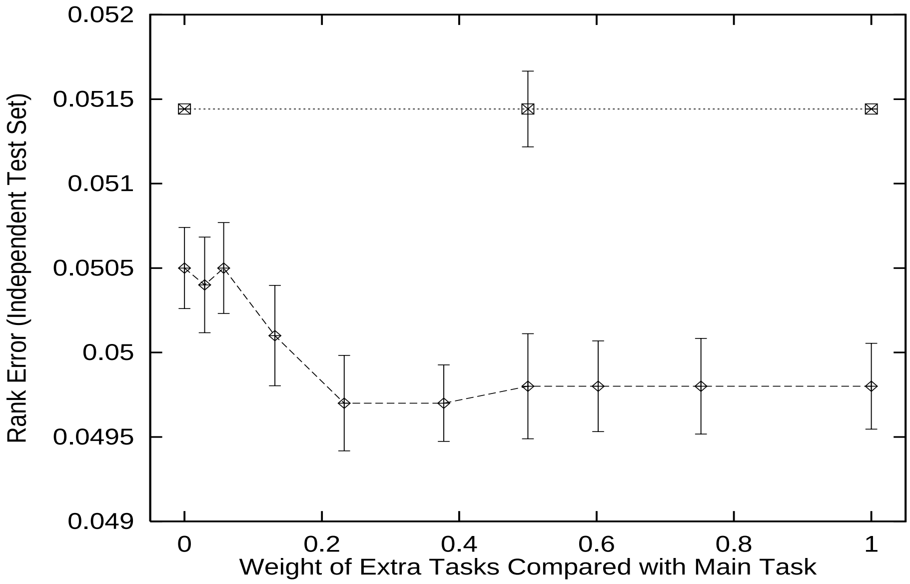


Figure 7.5: Rank Sum Error as a Function of λ During MTL (Train = 1000; Halt = 1000; RunTime = 1000+1000)

Table 7.3: Error rates of STL LCWA and MTL LCWA ($\lambda = 0.5$) on the pneumonia problem using train and test sets with 1000 cases each, and then combining the train and halt sets to form a runtime set with 2000 cases.

FOP	0.1	0.2	0.3	0.4	0.5	0.6	SoftRankSum
STL LCWA	.0091	.0146	.0201	.0282	.0368	.0483	.0505
MTL LCWA	.0088	.0124	.0184	.0259	.0360	.0461	.0498
% Change	-3.3%	-15.1% **	-8.5%	-8.2%	-2.2%	-4.6%	-1.4%

bias what is learned to capture better regularities in the domain. Before proceeding to the next experiment, it is interesting to note that an experiment that considered many possible different weightings for the extra tasks (including giving them no weight at all) determined that, for this problem at least, best performance resulted from giving the same weight to each extra task as to the main task. We do not believe this will be true in general. We suspect that in many domains it will be better to give more weight to the main task than to some or all of the extra tasks.

7.7.4 Experiment 2: How Large Is the MTL Benefit?

Multitask LCWA outperforms traditional single task LCWA about 5–10% on this domain when the training is done with 500 or 1000 cases. But the improvement in STL due to

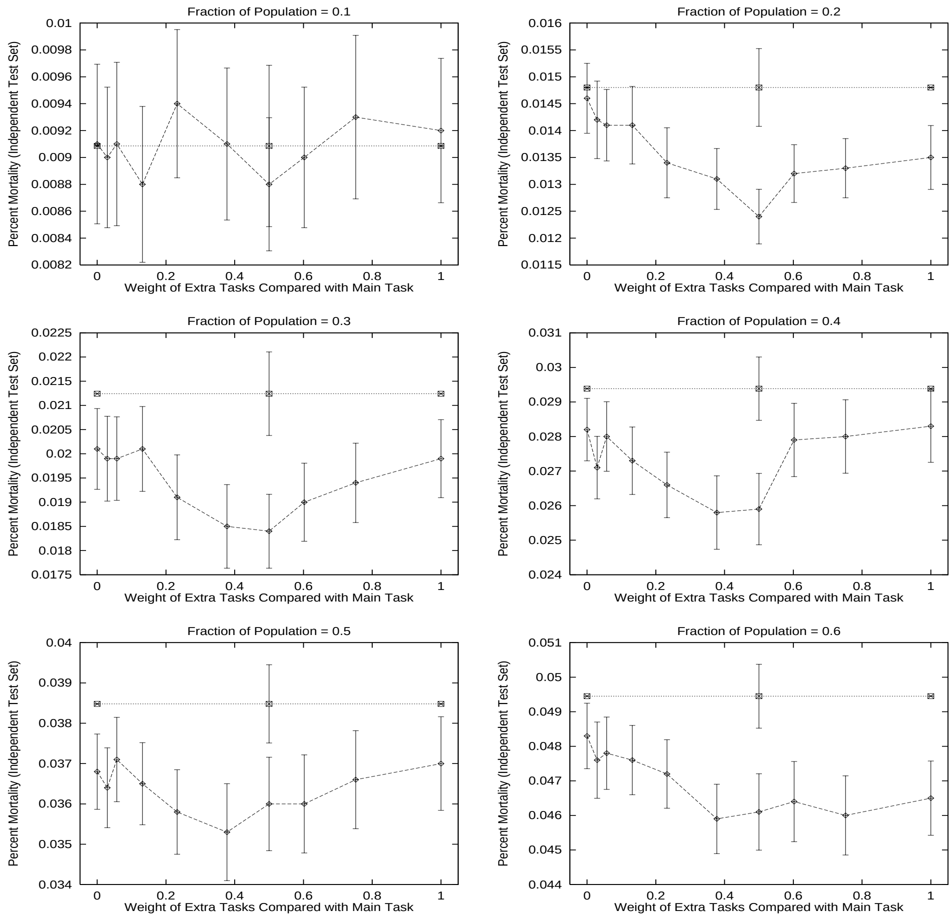


Figure 7.6: Performance on Different Fractions of the Population as λ Varies (Train = 1000; Halt = 1000; RunTime = 1000+1000)

doubling the size of the training set was larger than the improvement of MTL over STL with the same size training set. How many more training examples are needed with STL to yield improvement comparable to MTL? We ran experiments using training sets containing 200, 400, 800, 1600, and 3200 training patterns, using STL and MTL with $\lambda = 0.5$.

Figure 7.7 shows the performance of 25 trials of traditional STL LCWA ($\lambda = 0$) and MTL LCWA with $\lambda = 0.5$ as a function of training set size. Performance improves quickly with increasing sample size when the sample size is small. In the region between 200 and 1000 cases, the improvement due to MTL is equivalent to a 50–100% increase in the number of training patterns, but decreases when the sample size gets large. With training sets larger than 3200 cases each, the performance with $\lambda = 0.5$ may be worse than with $\lambda = 0$ because

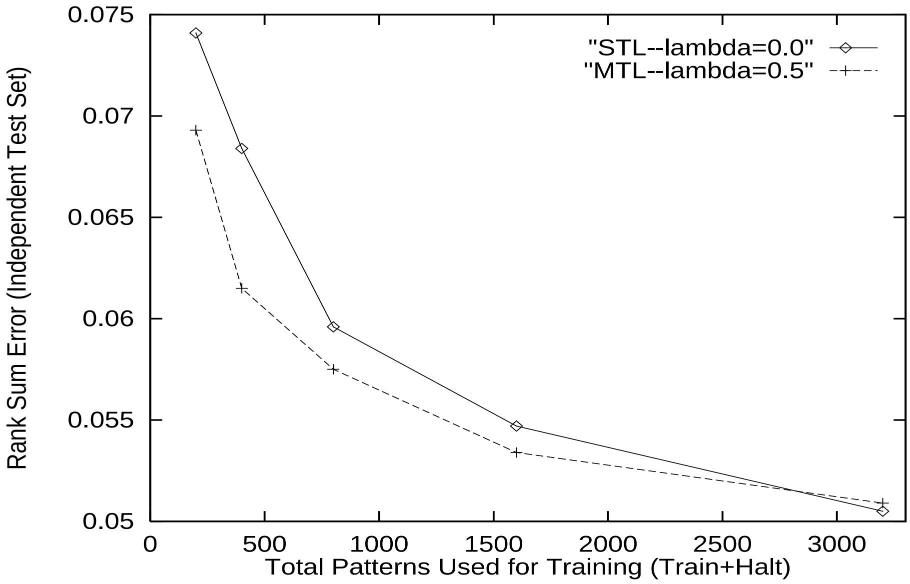


Figure 7.7: Rank Sum Error of STL ($\lambda = 0$) and MTL ($\lambda = 0.5$) as a Function of the Size of the Training+Halt Sets

one does not need the inductive bias from the extra tasks if the sample size is large enough to insure excellent performance from optimizing the main task alone. Values of λ in the range $0 < \lambda < 0.5$ however, may yield better performance than STL ($\lambda = 0$) even with large training sets. (It would be interesting to see the optimal value of λ as a function of the amount of training data for this problem. We haven't done this experiment, but reducing the strength of the MTL bias as more data is available should improve performance. In the limit with infinite data, $\lambda = 0$ should be best.)

7.7.5 Experiment 3: MTL without Extra Tasks

Experiments 1 and 2 used lab tests that were available for the training data that will not usually be available for future patients as extra tasks to help learn good feature weights for the main task. Suppose there are no extra tasks. Can MTL still be used? Yes, sort of. The elegance of KNN and LCWA allows us to treat each *input* as an extra task that is used to help learn a better distance metric for the main task.

Suppose there is one main task to learn given N attributes. When predicting the main task, we apply KNN or LCWA with a distance metric defined on all N attributes. But we treat each attribute as an extra task, predicted from all the other attributes, using a distance metric defined on N-1 attributes.[7] The feature weights used in this distance metric are the same feature weights used to predict the main task, except that the feature weight for each attribute is ignored when predictions for that attribute are made. Thus we have N+1 tasks in total.

Figures 7.8 and 7.9 show the performance of STL LCWA and MTL LCWA on pneumonia risk prediction using train and test sets containing 500 cases each when the pre-hospitalization basic measurements are used as extra tasks predicted from the remaining attributes. In this experiment the future lab tests are not used at all; we are doing MTL solely using the input features as extra tasks.

Using the attributes themselves as extra tasks does not improve performance on the main risk prediction task nearly as much as using the future lab tests as extra features. Performance on the rank error metric is possibly worse using MTL with the attributes as extra tasks. MTL Performance for FOP 0.1 and 0.2 has probably improved compared with STL, performance for FOP 0.3, 0.4, and 0.5 is comparable to STL, and performance for FOP 0.6 is probably worse than STL. This is an interesting result. When $\lambda = 1.0$, KNN MTL using features as the extra tasks is equivalent to performing unsupervised learning on the problem and then seeing how well the resulting clusters predict the main risk task. The fact that we do not see improvements in performance when doing MTL this way strongly suggests that unsupervised learning would not be effective on this domain if the goal is to predict pneumonia risk using the unsupervised model.

[7]One might imagine using the main task as an attribute, but we are not interested in learning models that are functions of the main task signal. Treating the main task as an attribute and giving equal weight to all tasks would effectively turn MTL KNN and LCWA into unsupervised learning algorithms. In a few domains this might be the best thing to do. But supervised learning typically outperforms unsupervised learning when the main prediction task is known. MTL is an intermediate point between supervised and unsupervised learning. Usually MTL performs best if it takes advantage of the fact that the goal is to maximize performance on one task at a time.

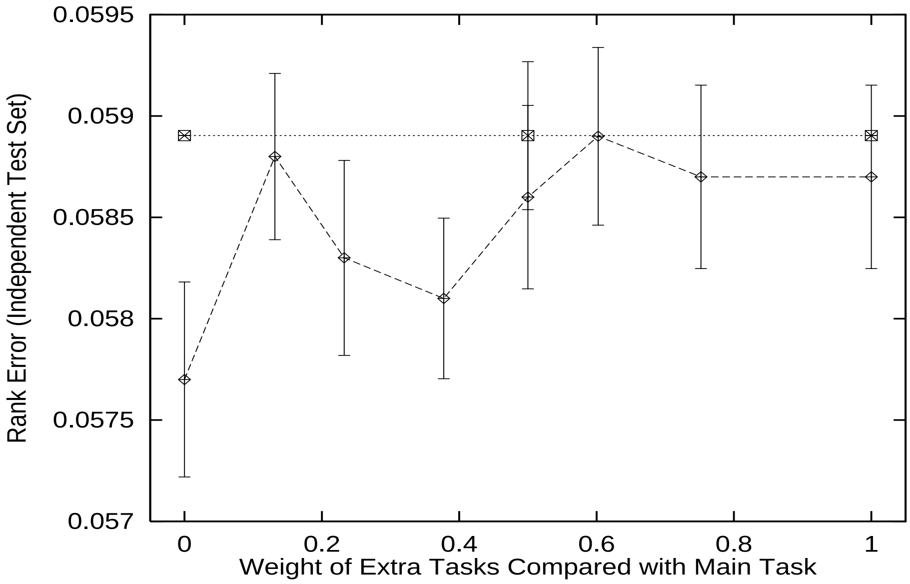


Figure 7.8: Rank Sum Error as a Function of λ During MTL using the Attributes as the Extra Tasks (Train = 500; Halt = 500) The horizontal line is the baseline performance with all weights initialized to 1. The lower curve is the performance of the weights trained with MTL as a function of λ. As before, $\lambda = 0$ ignores the extra tasks completely and is thus equivalent to STL.

7.7.6 Feature Weights Learned with STL and MTL

The goal of using MTL with LCWA is to learn feature weights that yield a distance function that makes the predictions returned by LCWA more accurate. The experiments in the previous sections clearly demonstrate that MTL using the future labs as extra MTL tasks significantly improves performance on pneumonia risk prediction. But how do the weights learned by STL and MTL differ? Table 7.4 lists the 30 feature weights learned by STL and MTL ($\lambda = 0.5$) with train and halt sets containing 500 cases each at the point where training was stopped, i.e., where performance on the halt set was best. The first column gives the feature number. The second column gives the feature weights learned by STL for trial 1. The third column gives the feature weights learned for the same trial with MTL. The fourth column gives the average of the feature weights learned with STL over the 50 trials. The fifth column is this same average for MTL. We present both the results of a single trial, and the average weights, because the averages tend to obliterate some of the detail of what is learned in the individual trials. The sixth column is the significance of the

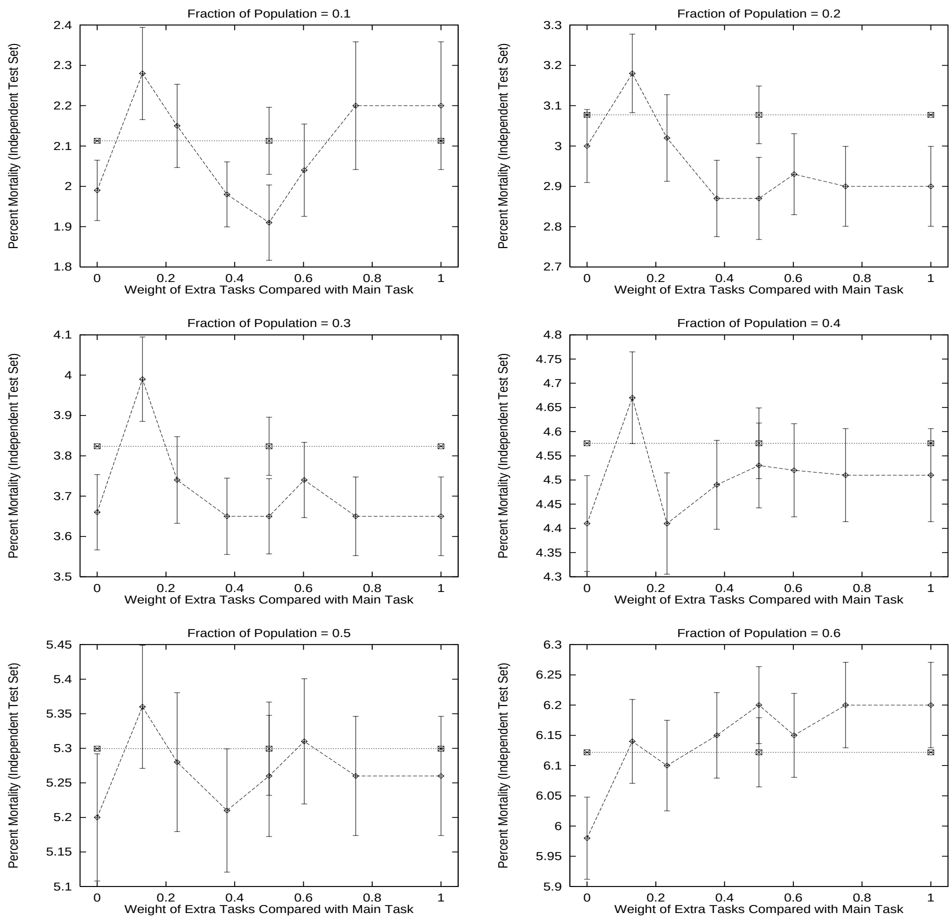


Figure 7.9: Performance on Different Fractions of the Population as λ Varies using the Attributes as the Extra MTL Tasks (Train = 500; Halt = 500)

differences between the weights learned by STL and MTL. "*", "**", and "***" indicate that the feature weight learned by 50 trials of STL is significantly different from the weight learned by MTL at the 0.05, 0.01, and 0.001 confidence level, respectively. Examining weights given to individual dimensions in a high dimension space this way is questionable. We do it not because the tests have any strong meaning, but because they are valuable in focusing one's attention on interesting differences in the learned feature weights.

Feature weights are initialized to 1.0 before learning. STL typically does not learn feature weights that differ as much from 1.0 as those learned by MTL. Examining the feature weights explored by STL during gradient descent, one observes that STL does explore settings of the feature weights that are far from 1.0, but these typically do not yield good performance on the halt set. In other words, STL appears to overfit significantly to the training set. MTL learns feature weights further from 1.0 that perform well on the halt sets. *MTL appears to overfit less then STL.* In some cases the difference between the feature weights learned by STL and MTL are considerable. See, for example, features 2 and 30.

One final note. Most learned feature weights are not that far from their initial value 1.0. The pneumonia problem is a real problem. It is hard to believe all features are useful—let alone roughly similarly useful—for predicting risk. Why do none of the features have values near 0.0 or larger than 2.0? We suspect that the search space has many local minima, and this prevents search from successfully exploring regions far from the initial weights. It would be interesting to run these experiments many times starting from different initial conditions, or to use a search procedure more immune to local minima such as a genetic algorithm. This might lead to further improvements in performance, or might lead to increased overfitting to the training sets. It might also be interesting to to run these experiments with weight decay regularization to keep feature weights near zero except where larger weights benefited learning significantly.

7.8 Summary and Discussion

Soft ranks and the soft rank sum should be useful in many domains, in part because ranks of functions are often easier to learn than the functions themselves. Soft ranks should broaden the utility of rank-based error metrics by making it easier to use gradient-based learning methods with them.

Currently there is no theory to predict which extra tasks are helpful. There is also no theory to predict the best λ. We were surprised to find $\lambda \approx 0.5$ was best on this problem, and we do not expect $\lambda \approx 0.5$ will be optimal for all sample sizes (or for all domains). KNN and LCWA perform well in the large sample limit. The extra information from the training signals for the extra tasks is unnecessary, and possibly misleading, near the asymptotic limit. Given large training sets, $\lambda \approx 0$ would probably be best. (We are running an experiment to test this.) Currently, the only way to choose λ is by cross validation. We have devised an algorithm that uses LOOCV with the *halt set* so that λ search does not require additional training examples. This algorithm is efficient enough to be able to learn a separate λ_t for each extra task. This is useful because it explicitly learns how useful each extra task is to the main task, and can learn to ignore harmful extra tasks by setting $\lambda_t \approx 0$ for those tasks.

KNN and LCWA often do not perform well in feature spaces with many dimensions. This is because points tend to become equidistant as dimension increases. KNN and LCWA depend on the differences in the distances between cases to make estimates. Another way to view this is that all samples are sparse in a high dimensional space, and sparse samples are rarely much closer to some points than they are to all the other points. Neighborhoods become rural in high dimensional spaces. Because neighborhoods are less dense in high dimensional spaces, finding good feature weights is more important, and more difficult, in high dimensional spaces. We conjecture that MTL is most useful in high dimensional feature spaces. In the pneumonia domain, MTL improved the performance of LCWA 5-10% when the future lab tests were used as extra tasks. Cross-validation on λ, the parameter that determines how much emphasis is given to the extra tasks, indicates that for this domain optimal performance results from giving near equal weight to the main task and the extra tasks. Experiments with MTL using the input features as extra tasks indicate that unsupervised learning is not able to yield benefits comparable to MTL with extra tasks in this domain.

Table 7.4: Feature weights learned by STL and MTL. Columns 2 and 3 are the weights learned by the first trial. Columns 4 and 5 are the average feature weights learned from the 50 trials.

Feature	STL (trial 1)	MTL (trial 1)	STL (avg. 50 trials)	MTL (avg. 50 trials)	Significance
1	1.650	1.700	1.04912	1.42282	***
2	1.141	0.525	0.89608	0.63568	***
3	0.491	1.222	0.93732	0.96712	
4	0.672	0.659	0.89578	0.62448	***
5	1.478	0.739	1.01854	0.97756	
6	0.249	0.645	0.96282	0.9648	
7	0.677	0.874	1.02142	0.9441	*
8	0.974	1.314	0.96666	1.1382	***
9	1.152	0.580	1.03448	0.88668	***
10	0.234	0.660	0.91214	0.65284	***
11	0.722	1.042	0.97104	0.80258	***
12	1.123	0.859	1.00142	0.94856	**
13	1.906	1.571	1.0289	1.39302	***
14	1.101	1.148	0.992	0.93146	*
15	1.643	1.264	1.07256	1.08256	
16	0.996	0.773	1.00506	0.99514	
17	0.976	1.057	0.9911	0.91536	**
18	0.451	0.715	0.93096	0.79924	***
19	1.000	1.000	1.00048	0.99994	
20	0.672	0.659	0.9007	0.62628	***
21	0.770	0.408	0.99382	0.8478	**
22	1.009	0.994	1.00294	0.99636	
23	0.987	0.991	0.99396	0.98934	
24	1.000	1.000	0.99608	0.99234	
25	0.895	0.949	0.99508	0.92924	*
26	0.993	0.966	0.99812	1.01138	
27	1.063	0.606	1.01532	0.92734	*
28	0.639	0.851	0.99914	1.03478	
29	1.270	1.704	1.08908	1.16508	
30	0.900	1.868	0.99522	1.58114	***

Chapter 8

Related Work

8.1 Backprop Nets With Multiple Outputs

It is common to train neural nets with multiple outputs. When this is done, usually the multiple outputs encode what is effectively a single task. For example, in classification tasks it is common to use one output to code for each class (see, for example, [Le Cun et al. 1989]). Although it is usually beneficial to train these multiple outputs on one net instead of on separate nets for the reasons described in this thesis, outputs coding for the different classes of one task are not usually thought of as different tasks. This is unfortunate. Treating the multiple outputs coding for different classes as different tasks allows one to optimize performance for each class individually by stopping early on the individual outputs, by adjusting the relative learning rates of the outputs, etc. Although this thesis emphasizes MTL with tasks more heterogeneous than the outputs coding for a multi-class problem, most of what has been learned can be used to improve learning in multi-class problems. The benefits of MTL in multi-class problems will probably be smaller, however, than we sometimes see with more heterogeneous sets of tasks.

Using one backprop net to train a few strongly related tasks at one time is also not new. The classic NETtalk [Sejnowski & Rosenberg 1986] application uses one backprop net to learn both phonemes and the stresses to be applied to those phonemes. Using one backprop net seems natural for NETtalk because the goal is to learn to control a synthesizer that needs both phoneme and stress commands at the same time. NETtalk is an early example

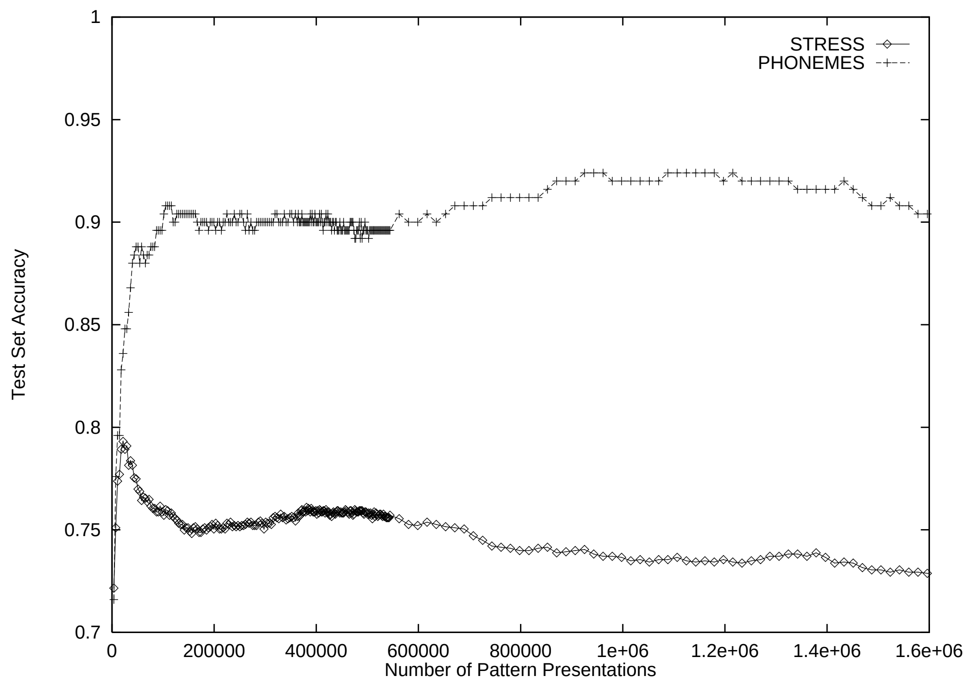


Figure 8.1: On NETtalk, the Stress task trains very quickly and then overfits long before the Phoneme task reaches peak performance.

of MTL. But the builders of NETtalk viewed the multiple outputs as codings for a single problem, not as independent tasks that benefited each other by being trained together. For example, Figure 8.1 graphs the NETtalk learning curves for the phoneme and stress tasks separately. From the figure it is clear that the stress tasks begin to overfit long before the phoneme tasks reach peak performance. The stress tasks reach peak performance by 5,000 backprop passes, whereas the phoneme tasks don't reach peak performance until about 1,100,100 backprop passes, by which time the stress task has significantly overfitted. Better performance could easily have been obtained in NETtalk by doing early stopping on the stress and phoneme tasks individually, or by balancing the learning rates of the different outputs so they reach peak performance at roughly the same time.

[Dietterich, Hild & Bakiri 1990, 1995] performed a thorough comparison of NETtalk and ID3 on the NETtalk text-to-speech domain. One explanation they considered as to why backpropagation outperformed ID3 on this problem is that backpropagation benefits from sharing hidden units between different outputs, something ID3 does not do. They conclude that although hidden unit sharing (i.e., MTL) does help, it is not the largest difference between the two learning methods, and suggest that adding sharing to ID3 probably would not be worthwhile.

8.2 Constructive Induction

Constructive Induction is one of the branches of machine learning most interested in learning representations. Most work in this field, however, attempts to assess representation quality for a single task. The best summary of the current state of the field and its relation to MTL is the following comments made by Sutton in 1994 at a constructive induction workshop:

> "Everyone knows that good representations are key to 99% of good learning performance. Why then has constructive induction, the science of finding good representations, been able to make only incremental improvements in performance of machine learning systems?
>
> People can learn amazingly fast because they bring good representations to the problem, representations they learned on previous problems. For people, then, constructive induction does make a large difference in performance. The difference, I argue is not the difference between people and machines, but in the way we are assessing performance.
>
> The standard machine learning methodology is to consider a single concept to be learned. That itself is the crux of the problem. Within this paradigm, constructive induction is doomed to appear a small, incremental, second-order effect. Within a single problem, constructive induction can use the first half of the training set to learn a better representation for the second half, and thus potentially improve performance during the second half. But by then most of the learning is already over. Most learning occurs very early in training. It may be possible to detect improvements due to constructive induction in this paradigm, but they will always be second order. They will always be swamped by first-order effects such as the quality of the base learning system, or, most importantly, by the quality of the original representation.

> This is not the way to study constructive induction! We need a methodology, a way of testing our methods, which will emphasize, not minimize, the effect of constructive induction. The standard one-concept learning task will never do this for us and must be abandoned. Instead we should look to natural learning systems, such as people, to get a better sense of the real task facing them. When we do this, I think we find the key difference that, for all practical purposes, people face not one task, but a series of tasks. The different tasks have different solutions, but they often share the same useful representations.
>
> This completely breaks the dilemma facing facing constructive induction, which now becomes a first order effect. If you can come to the nth task with an excellent representation learned from the preceding n-1 tasks, then you can learn dramatically faster than a system that does not use constructive induction. A system without constructive induction will learn no faster on the nth task than on the 1st. Constructive induction becomes a major effect, a 99% effect rather than a 1% effect. Most importantly, we now have a sensitive measure of the quality of our constructive induction methods, a measure unpolluted by tricky issues such as the original learner or the original representation. All those things are factored out. For the first time we will see pure effects due to changes in representation. This, I hope, would enable us to evaluate our methods better and lead to faster progress in the field."

Some of Sutton's early work in reinforcement learning also recognizes the importance of learning from multiple examples. Although the methods he develops do not significantly address MTL, it is clear some of his original motivation is identical to MTL:

> "Finally, a broad conclusion I make from this work has to do with the importance of looking at a series of related tasks, such as here in a non-stationary tracking task, as opposed to conventional single learning tasks. Single learning tasks have certainly proved extremely useful, but they are also limited as ways of exploring important issues such as representation change and identification of relevant and irrelevant features. Such meta-learning issues may have only a

small, second-order effect in a single learning task, but a very large effect in a continuing sequence of related learning tasks. Such cross-task learning may well be the key to powerful human-level learning abilities." [Sutton 1992]

8.3 Serial Transfer

Transferring learned structure between related tasks is not new. The early work on sequential transfer of learned structure between neural nets [Pratt et al. 1991; Pratt 1992; Sharkey & Sharkey 1992] clearly demonstrates that what is learned for one task can be used as a bias for other tasks. Unfortunately, this work failed to find improvements in generalization performance; the main benefit was speeding up learning.

More recently, Mitchell and Thrun devised a serial transfer method called Explanation-Based Neural Nets (EBNN) [Thrun & Mitchell 1994; Thrun 1995, 1996] based on tangent prop [Simard et al. 1992] that yields improved generalization when trained on a sequence of learned tasks. EBNN trains the partial derivative of outputs with respect to the inputs at the same time the outputs are being trained with the target values. The partial derivative information is computed from previously learned related models. For example, if the partial derivative of the output of a previously learned model is strong and positive with respect to some input, EBNN biases the new model that is being trained to also have a strong, positive derivative with respect to this input. One noteworthy component of EBNN is the heuristic it uses to moderate the strength of this bias. If the current instance is not well predicted by the previous model, EBNN assumes that the partial derivative information from this previous model is less likely to be relevant to the new model and gives less weight to the tangent prop error term. This is important because applying a constraint directly to the output being used for a new task is a much stronger bias than applying it through an additional output that shares a large hidden layer with the new task. This ability to moderate the strength of the EBNN bias allows EBNN to degrade gracefully to the performance of STL in some cases where the previous tasks are not well related to the new task.

One disadvantage of EBNN compared with MTL-backprop is that EBNN does not have access to the internal representations learned by backprop for previously learned tasks.

Because of this, one might expect EBNN to have feature selection capabilities comparable to or better than MTL-backprop (because this information is well captured by input/output relations like partial derivatives), but to be poorer at eavesdropping on features developed by other tasks. Also, because derivatives are sensitive to noise and other sources of nonlinearity, MTL-backprop is likely to benefit more than EBNN from noisy related tasks, or related tasks that are highly nonlinear in ways that do not match nearly perfectly between tasks.

Because EBNN and MTL-backprop use different mechanisms and may benefit from different kinds of task relationships, combining them may be beneficial. O'Sullivan is exploring a thesis that does this in an attempt to build life-long learning robots. As part of this work, he performed a comparison of EBNN and MTL-backprop on a robot perception task. Figure 8.2 shows the percent improvement due to EBNN and MTL-backprop over traditional STL-backprop in this robot domain as a function of the number of training examples.

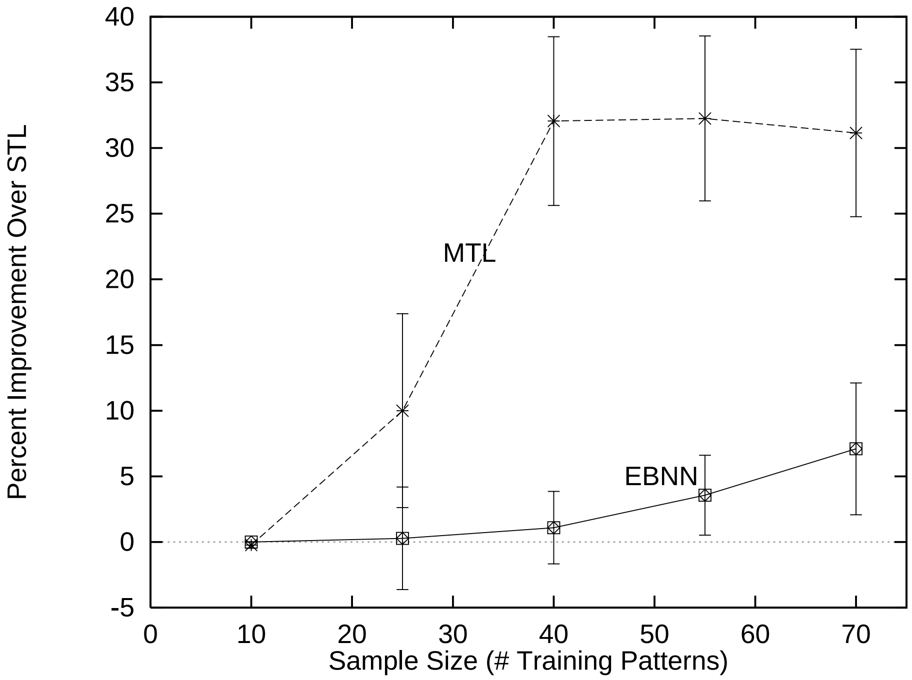


Figure 8.2: Improvement of EBNN and MTL-backprop over STL-backprop on a Robot Life-Long Learning Task (courtesy Joseph O'Sullivan).

Although in this domain MTL-backprop yields significantly more benefit than EBNN, O'Sullivan reports that combining the two methods on this problem performs better than MTL-backprop or EBNN alone, suggesting that some of the benefits of EBNN and MTL-backprop are different.

[O'Sullivan & Thrun 1996] devised a serial transfer mechanism called TC for KNN that clusters previously learned tasks into sets of related tasks. In TC, the KNN attribute weights learned for previous tasks in the cluster most similar to the new task are used for the new task when the number of training patterns for the new task are too small to support accurate learning. TC is similar in some ways to the MTL-KNN method presented in Chapter 7. The main difference is that MTL-KNN attempts to learn the set of attribute weights that yields optimal performance on both the main task and the extra tasks. The TC method does not learn a set of attribute weights optimal for a group of tasks. Instead, it uses the weights optimized for some previous task for the new task until enough training patterns for the new task are available to make optimizing attribute weights for the new task alone feasible. Combining MTL-KNN with TC should yield a method with the benefits of each.

[Breiman & Friedman 1995] present a method called Curds & Whey that also combines sequential and parallel learning. Curds & Whey takes advantage of correlations between different prediction tasks. Models for different tasks are trained separately (i.e., via STL), but predictions from the separately learned models are combined before making the final predictions. This sharing of the *predictions* of the models instead of the *internal* structure learned by the models is quite different from MTL; combining the two methods is straightforward and might be advantageous in some domains.

8.4 Hints

[Hinton 1986] suggested that generalization in artificial neural nets would improve if nets learned to represent underlying regularities of the domain better. Suddarth and Abu-Mostafa were among the first to recognize that this might be accomplished by providing extra information at the *outputs* of a net. [Suddarth & Kergosien 1990; Suddarth & Holden 1991] used extra outputs to *inject rule hints* into networks about what they should learn. This is MTL where the extra tasks are carefully engineered to coerce the net to learn specific internal representations. The centerline extra tasks in the 1D-ALVINN domain in

Section 2.1 are good examples of rule-injection hints. Suddarth and Holden were not very successful, however, in making their approach work on real problems.

[Abu-Mostafa 1990, 1993, 1996] provides hints to backprop nets via extra terms in the error signal backpropagated for the main task output. The extra error terms constrain what is learned to satisfy desired properties of main task such as monotonicity [Sill & Abu-Mostafa 1997], symmetry, or transitivity with respect to certain sets of inputs. MTL, which currently does not use extra error terms on the task outputs, could easily be used in concert with these techniques. EBNN, which also provides extra information through an output by adding an extra error term to that output (the error term for the partial derivative) is a form of hint where the information for the extra error term comes not from expertise about the domain but from previously learned related tasks.

8.5 Unsupervised Learning

We showed in Section 3.5 that MTL-backprop depends on a heretofore unrecognized ability of backprop to discover relationships between tasks without being given explicit training information about task relationships. In effect, MTL-backprop does unsupervised clustering of outputs based on their hidden layer representations. It should come as no surprise, then, that MTL is similar in some ways to other methods used for clustering and unsupervised learning. For example, small changes to the indices in COBWEB's [Fisher 1987] probabilistic information metric yields a metric suitable for judging splits in multitask decision trees. Whereas COBWEB considers *all* features as tasks to predict, MTL decision trees allow the user to specify which signals are inputs and which are training signals. This makes it easier to create additional tasks without committing to extra training information being available at run time, and also makes learning simpler in domains where some input features cannot reasonably be predicted. [Martin 1994, Martin & Billman 1994] explore how concept formation systems such as COBWEB can be extended to acquire overlapping concept descriptions. Their OLOC system is an incremental concept learner that learns overlapping probabilistic descriptions that improve predictive accuracy.

de Sa's Minimizing Disagreement Algorithm (MDA) [de Sa 1994] is an unsupervised learning method similar in spirit to MTL-backprop. In MDA, multiple unsupervised learning tasks are trained in parallel and bias each other via supervisory signals from the other unsupervised tasks. One point-of-view that helps unify these two approaches is as follows: MTL-backprop is an attempt to bring an unsupervised learning component to backpropagation (a supervised learning method) by giving extra information to the MTL net through the extra training signals applied to extra outputs on the net. MDA is an attempt to bring a supervised learning component to vector quantization (an unsupervised learning method) by giving auxiliary signals to each vector quantization process through the extra training signals derived from vector quantization processes learning related unsupervised problems. The goal in both MTL-backprop and MDA is to improve the generalization capability of what is learned. (The work in Chapter 5 is joint work with de Sa. Because both MTL-backprop and MDA have supervised and unsupervised learning components, de Sa and I are both interested in understanding what information is best used for supervised and unsupervised learning.)

8.6 Theories of Parallel Transfer

Attempts have been made to develop theories of parallel transfer [Abu-Mostafa 1993; Baxter 1994, 1995, 1996]. The most advanced thus far is Baxter's theory for representation learning in neural nets. It shows that the number of training patterns required to learn N *strongly related* tasks on one net grows as

$$O(a + \frac{b}{N})$$

where $O(a)$ is a bound on the minimum number of training examples required to learn a single task, and $O(a+b)$ is a bound on the number of examples required to learn all the tasks independently. What this bound says is that for a sufficiently large number of tasks, N, the number of training examples needed to learn the Nth task gets smaller if the previous $N-1$ tasks already have been learned by the same net. For this to be true, one must assume that all tasks can be learned accurately from a common, compact representation, that the training patterns for each task are independently sampled, and that the learning procedure is an empirical risk minimizer that is able to benefit from the training signals of the multiple tasks.

Unfortunately, the theory developed so far has little relation to real-world uses of MTL. Limitations of the current theory are the following:

- Because the theory assumes the training patterns for each task are independent, it does not apply to most uses of MTL where the same training patterns are used for all (or most) tasks.

- The theory lacks a well defined notion of task relatedness. In lieu of this, the current theory makes assumptions about how much tasks overlap. For example, Baxter's representation learning theory assumes that all tasks can be learned from one shared hidden layer that is small relative to the number of tasks to be trained on it. In other words, the theory assumes a bottleneck hidden layer is the appropriate architecture for the tasks. This bottleneck assumption is rarely satisfied in practice unless the tasks are *very strongly related*. We usually find optimal performance requires increasing the number of hidden units in the shared hidden layer as the number of tasks increases. This empirical finding conflicts with assumption made by the theory that the hidden layer is constant size for any number of tasks. MTL is often applied to a number of tasks that is small relative to the number of hidden units in the hidden layer because MTL is often applied to tasks that are not so strongly related.

- The theory yields loose worst-case bounds. It is possible to create synthetic problems that satisfy the assumptions of the theory, but where increasing the number of tasks hurts performance instead of helping as the theory predicts. The results are consistent with the theory, but only because the bounds are loose enough to allow it.

- The theory makes assumptions about the search procedure that are not easily justified and is unable to account for behaviors of the search procedure that, in practice, are critical. As just one example, if early stopping is not done correctly, MTL-backprop often hurts performance instead of helping it. The theory is unable to account for critical phenomena such as early stopping because it does not model how nets are trained. This failure to model the search procedure is more serious than it might seem. Consider an artificial data set where training signals for multiple tasks are generated from a common pre-defined hidden layer. MTL-backprop nets trained on these tasks often are unable to learn the tasks well when trained with the same number of hidden units as the generator net even though the theory predicts that the net with the same number of hidden units as the generator net should perform best. In practice, often it is necessary to use more hidden units in the net being trained than in the generator net. This is because backpropagation is a greedy search procedure that does not perform well in tightly constrained search spaces. Nets trained with backpropagation often need extra hidden units to facilitate search, even though a smaller number of hidden units is sufficient to represent the model to be learned. Because the theory does not model the search procedures used to train nets in practice, it makes different predictions from what often is observed in practice, and is thus difficult to use prescriptively.

The fundamental problem with the theories of inductive transfer developed so far is that they are all based on restricted capacity arguments: if one fixes the size of the model class (e.g., by fixing the size of the hidden layer), but increases the complexity of what is to be learned (e.g., by making it predict more outputs), then the odds of finding a model in the fixed-model space that falsely *appears* to have high accuracy is reduced. This is equivalent to saying that if more task training signals can be accounted for using a fixed number of bits, the probability that the model captured by those bits will reliably predict future patterns is greater. But capacity in artificial neural nets trained with backprop is poorly understood. The number of hidden units is a poor measure of net capacity. Training the weights in a net changes its effective capacity. Since we will train all nets, both STL and MTL, until we achieve maximum accuracy on the training and halt sets, we cannot claim that one net has more or less capacity once it is trained. Any net that is large enough can be trained to zero error on the training set. Thus all large enough nets can be trained to the same final effective capacity. Our empirical results suggest best performance usually requires more hidden units as the number of tasks increases. Part of the reason for this is that if small nets are to fit a nonlinear training set they must use larger weights. This makes them more nonlinear. Larger nets can fit the same nonlinear training set using smaller weights because more weights are available. This makes them less nonlinear. Thus larger nets often learn smoother functions for the same training set. But the theory does not yet account for phenomena like this because it ignores the training procedure (e.g., backprop) and only applies to nets with restricted capacity.

8.7 Methods for Handling Missing Data

One application of MTL is to take features that will be missing at run time, but that are available for the training set, and use them as outputs instead of inputs. There are other ways to handle missing values. One approach is to treat each missing feature as a separate learning problem, and use predictions for missing values as inputs. (We tried this on the pneumonia problem and did not achieve performance comparable to MTL, but in some domains this works well.) Other approaches to missing data include marginalizing over the missing values in learned probabilistic models [Little & Rubin 1987; Tresp, Ahmad & Neuneier 1994], and using methods like EM to iteratively reestimate missing values from current estimates of the data density [Ghahramani & Jordan 1994, 1997].

8.8 Bayesian Graphical Models

Models trained with supervised learning usually model only the likelihood of the output given the inputs, i.e., they usually learn only the conditional probability of the output given the inputs. MTL makes better predictions for the outputs given the same inputs because it learns more complete models; training one model on multiple tasks drawn from the same domain helps the model better learn the regularities of that domain, and this helps prediction. Some Bayesian methods (frequently called graphical models) also learn domain models more complete than those usually learned with supervised learning. They learn the full joint probability distribution function that relates inputs and outputs. The comprehensiveness of these models allows them to handle missing values. Graphical models and MTL are attempting to do similar things: both try to make better predictions by learning more complete models. The largest differences between graphical models and MTL are the following:

1. Graphical models have a strong probability semantics with normative (or normative approximating) procedures for doing learning and making predictions from what has been learned.

2. Graphical models often attempt to model the entire joint probability density for all features and outputs, and in doing so treat inputs and outputs alike.

3. Graphical models sometimes attempt to learn models with causal semantics.

Difference 1 is a point in favor of graphical models; where practical, strong probability semantics is desirable. Difference 2, however, may not be an advantage. The main strength of supervised learning over unsupervised learning is that supervised learning knows what the outputs and inputs are, and it is free to do anything that yields better prediction on the outputs given the inputs. Supervised learning does not tradeoff reduced accuracy on the outputs in order to achieve increased fidelity of the models for some of the inputs. Because graphical models often attempt to learn models that capture the causal structure of the domain, they are more constrained than traditional unsupervised learning techniques, and this may minimize some of the difficulties associated with trying to learn models of all features. Nevertheless, the models learned by Bayesian graphical model methods may still learn more structure about a domain than is needed for optimal prediction on a prespecified set of outputs. It is not clear just now whether graphical models represent a better approach to accomplishing what MTL accomplishes, or whether the extra complexity of their models will make them inferior to traditional supervised learning and MTL.

8.9 Other Uses of MTL

8.9.1 Committee Machines

[Munro & Parmanto 1997] use extra tasks to improve the generalization performance of a committee machine that combines the predictions of multiple learned experts. Because committee machines work better if the errors made by different committee members are decorrelated, they use a different extra task for each committee member to bias *how* it learns the main task. Each committee member learns the main task in a slightly different way, and the performance of the committee as a whole improves. Committee machines trained with extra tasks can be viewed as MTL with architectures more complex than the simple, fully connected MTL architectures presented here. One interesting feature of committee MTL architectures is that multiple copies of the main task are used, and this improves performance on the main task. Sometimes this same effect is observed with simpler, fully connected MTL nets, too [Caruana 1993]. [Dietterich & Bakiri 1995] examine a much more sophisticated approach to benefitting from multiple copies of the main task by using multi-bit error-correcting codes as the output representation.

8.9.2 Input Reconstruction (IRE)

Pomerleau's ALVINN system used artificial neural nets to learn to steer an autonomous vehicle. One of the important issues that arose in the ALVINN research was how to assess confidences in the steering predictions made by the net. Pomerleau developed a method called IRE (Input Reconstruction) to help assess confidences in the predictions. Figure 8.3 shows an IRE net.

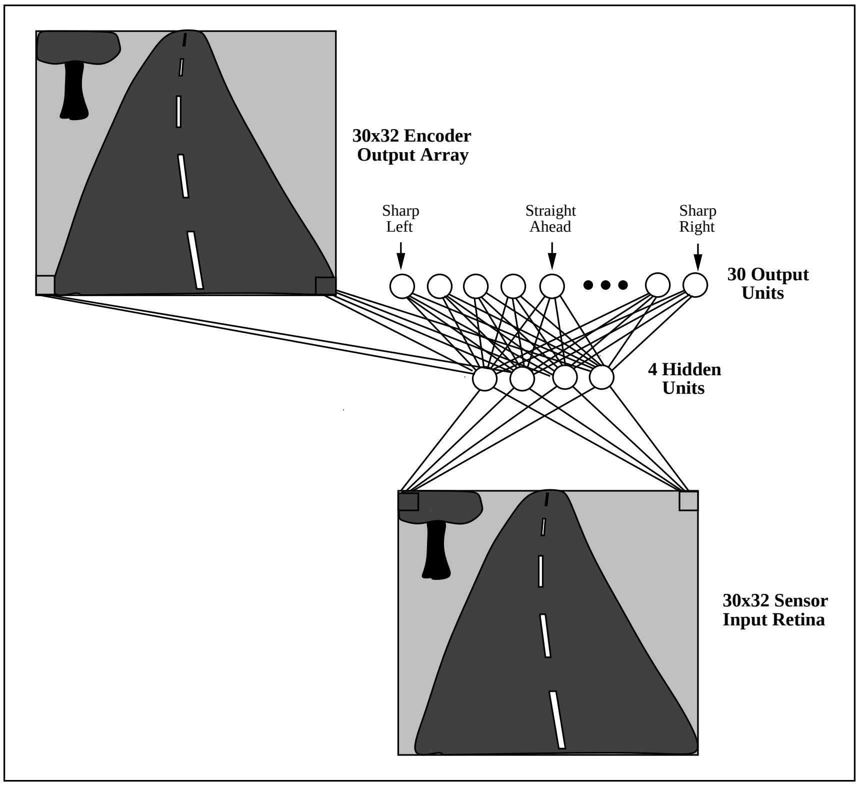


Figure 8.3: IRE Net for Assessing Prediction Confidences in ALVINN (courtesy Dean Pomerleau).

The input to the net is a retina image of the road in front of the vehicle. There are two sets of outputs. The first set of 30 outputs is a distributed output encoding used to represent steering direction. We would call this the main task. The second set of outputs is 30X32 rectangular image of outputs trained to reconstruct the input retina image. This reconstruction is trained in parallel with the main steering task, and it shares the hidden layer with the steering task. The hidden layer is small to prevent the net from learning direct connections between the retina inputs and the reconstructed output, and to insure that the hidden layer representation used for retina reconstruction is also used for the main steering task.

By assessing how closely the reconstructed image matches the input images, IRE is able to estimate how well the internal representations capture the structure in this particular road image. If the image is well reconstructed, presumably the hidden layer representation does a good job "understanding" this road image and thus one expects the steering direction predicted by the net should also be reliable. If, however, the image is poorly reconstructed, then the hidden layer presumably does not "recognize" this image and thus one might expect the steering prediction to be poorer.

The goal of IRE is to provide a mechanism for assessing confidences in predictions made by an ALVINN net, not to use image reconstruction as an extra task to improve performance on the main steering task. Pomerleau points out that IRE has the potential to hurt performance on the main steering task by training one hidden layer to learn steering and whatever features are necessary to reconstruct the image, even if they are unrelated to steering. Pomerleau notes, however, that in practice IRE did not hurt performance on the main steering task. Pomerleau also goes on to suggest a more complex IRE architecture that separates the hidden layer used for reconstruction from the hidden layer used for the main task. This architecture is virtually identical to the MTL-backprop architecture with separate private and public hidden layers described in Section 6.3.2.

As shown in Figure 8.3, Pomerleau's ALVINN nets use a distributed output coding for the main steering task. Instead of using one continuous output unit to code for the entire steering range, he uses 30 outputs trained to reproduce a Gaussian bump centered at the correct steering angle. This distributed output representation improves the accuracy and reduces the variance of the steering predictions. It is not clear what relation this multiple output representation has to MTL-backprop. It may be more similar to error-correcting output codes [Dietterich & Bakiri 1995] than to MTL. For example, steering accuracy might even *improve* if the 30 distributed outputs where trained on separate nets if this would de-correlate their errors enough to make the fitted Gaussian bump more accurate. When trained on a single ALVINN net as Pomerleau does, steering performance probably could be improved by doing early stopping for each of the 30 outputs individually. Better performance might also be achieved by training 30 MTL nets on the outputs where each net has 30 outputs, but each net is optimized to perform best on one output at a time.

8.9.3 Task-Specific Selective Attention

Baluja's thesis [Baluja 1996] considers the problem of training backprop nets on problems where knowing where to look improves recognition accuracy. For example, he trains backprop nets to locate and recognize +'s and x's in a sequence of images where the +'s and x's move around in the input retina in a predictable (i.e., learnable) fashion. His main result shows that if there is noise in the input images (in the form of spurious +'s or x's that show up at random locations), the net learns better to locate and identify the non-spurious +'s and x's if the net is also used to predict where the non-spurious symbol will appear in the next image and this prediction is used to focus the attention of the net when it "sees" the next image. This focus of attention mechanism does not train the hidden layer jointly for the prediction and recognition tasks as MTL would. It does, however, use the prediction task to modify the inputs the net "sees" on the next image, and this affects what the net learns as it is trained with backpropagation. This is a different approach to using related extra tasks (in this case location prediction) to improve the performance on the main tasks (in this case symbol localization and recognition). Because the main task has two components, location and recognition, Baluja performs experiments comparing separate nets trained on the two tasks with a single net trained on both tasks. The results of these experiments that when the noise is high, there is a benefit to training the two tasks together on one backprop net.

Chapter 9

Contributions, Discussion, and Future Work

9.1 Contributions

This thesis provided a clear demonstration that generalization performance can be improved by learning sets of related tasks together instead of learning them one at a time (Chapter 2). Although much of the work in this thesis was done using backpropagation in artificial neural nets (Chapters 1–6), we also presented an algorithm for multitask learning in k-nearest neighbor/kernel methods (Chapter 7), and sketch an algorithm for multitask learning in decision trees (Section 9.2.12). These are three of the more mature, and more successful, learning methods to date. The fact that multitask learning can be applied to each of these machine learning methods demonstrates the generality and potential impact of research in multitask transfer.

We applied multitask learning in artificial neural nets and in k-nearest neighbor to a number of problems, both real and synthetic. We found that multitask learning almost always helps generalization performance, and rarely hurts generalization performance. Our experience on these problems yielded a number of prescriptions for how to get the best results, some of which are counter intuitive, yet important to success (Chapter 6). We developed a method for automatically balancing the relative importance of extra tasks in k-nearest neighbor (Chapter 7). For artificial neural nets, we presented a method of automatically adjusting the rate at which different tasks learn that maximizes the opportunity for beneficial cross-task transfer (Section 6.2). This method not only makes multitask learning work better and easier to use, but may subsume some of the class frequency balancing techniques that have been applied in many real world problems.

We showed that most learning problems have opportunities for multitask learning if the problems are not first overly sanitized by those following the standard practice of the single task learning paradigm that currently dominates machine learning (e.g., Sections 2.3–2.4 and Chapter 4). With STL, we have been throwing away valuable information, often without realizing it, merely because we did not know how to use it. Multitask learning is one way of making use of information that does not easily fit into the traditional single task mold.

This thesis demonstrated that the benefit of using extra tasks can be substantial. Although the benefit from any one task is usually small, the effect from multiple extra tasks can be additive, so the benefit from a large number of extra tasks (10 or more) can be large. Through careful experiments, we were able to show that the benefits of multitask learning are due to the extra information contained in the training signals for the extra tasks, not due to some other property of backpropagation nets that might be achieved in another way (Sections 1.4, 2.3.6, and 3.4.3). We were able to elucidate seven different kinds of relationships between tasks that enable them to benefit from each other when learned in parallel (Chapter 3). These relationships serve as heuristics to help us identify when backpropagation can benefit from the training signals of extra tasks. Although we may never have an adequate theory to predict what extra tasks are and are not helpful, we believe a similar set of heuristics can be developed for any multitask learning procedure.

To examine what is happening inside artificial neural nets learning related tasks in parallel, we developed a method of measuring how much different tasks share hidden layer representations (Section 3.5). When applied to tasks where we know beforehand how related the different tasks are, we found that the more the tasks are related, the more the tasks share features developed at the hidden layer (Section 3.4). Interestingly, for this to happen, backpropagation must be doing a form of unsupervised clustering of the tasks based on their hidden layer representations. That is, MTL-backprop is clustering tasks in function space, not in feature space. We suspect that this heretofore unrecognized and largely unexploited capability of backpropagation to perform unsupervised clustering may be useful not only for multitask learning, but also for general clustering as well. This is particularly interesting because it helps forge a link between supervised and unsupervised learning.

An empirical analysis of the benefit of using certain features as extra inputs or as extra outputs clearly demonstrates that the benefit of a feature when used as an input can be quite different from the benefit of that same feature when used as an output. We were able to demonstrate that in some domains some features that could be used as inputs were more useful when used as outputs (Chapter 5). Because some features are beneficial when used as inputs, and when used as outputs, we devised and demonstrated a multitask learning method that allows a backprop net to use the same features as both inputs and as outputs at the same time (Section 5.3). This method yielded better performance than using these features just as extra input features or just as extra multitask outputs.

Acquiring domain-specific inductive bias is subject to the usual knowledge acquisition bottleneck. Multitask learning allows inductive bias to be acquired via the training signals of related additional tasks drawn from the same domain. Inductive learning probably isn't going to keep getting better if background knowledge isn't brought to bear on the learning problem. Multitask learning is one way of bringing domain specific background knowledge to bear by exploiting the one thing the current learning algorithms are best at—tabula rasa learning from examples.

Other contributions of the thesis that are not directly connected to multitask learning are an empirical study of generalization performance vs. capacity for backprop nets trained with early stopping (Appendix 1), and the development of two rank-based error metrics, Rankprop (Section 2.3.5 and Appendix 2) and Soft Ranks (Section 7.5 and Appendix 2). We believe rank-based metrics will be useful for a number of problems currently of interest in machine learning (e.g., information retrieval).

Perhaps the most important contribution of this thesis is that we have identified a number of situations that commonly arise in real-world domains where multitask learning should be applicable (Chapter 4). This is surprising—few of the standard test problems used in machine learning today are multitask problems. We conjecture that as machine learning is applied to unsanitized, real-world problems, the opportunities for multitask learning will increase.

9.2 Discussion and Future Work

9.2.1 Predictions for Multiple Tasks

MTL trains multiple tasks in parallel on one learner. This does not mean one learned model should be used to make predictions for those multiple tasks. The reason for training multiple tasks on one learner is so tasks can benefit from the information contained in the training signals of other tasks, *not to reduce the number of models that must be learned.* Tradeoffs often can be made between mediocre performance on all tasks and optimal performance on one task. Where this is the case, it is better to optimize performance on each important task one at a time and allow performance on the extra tasks to degrade. The adaptive learning rates used in MTL-backprop (see Section 6.2) and the λ weight(s) used for extra tasks in MTL-KNN/LCWA (see Section 7.3) make this tradeoff explicit; learning can even ignore some of the extra tasks to achieve better performance on the main task. The MTL-backprop architecture that reserves a private hidden layer for the main task, while training extra tasks using a more resource-limited public hidden layer (see Section 6.3.2), is also an example of optimizing training to favor the main task.

The current standard approach to early stopping in backprop nets is to stop training when some aggregate test-set measure like the sum squared error across all outputs begins to get worse. When early stopping is used in MTL, it is important to apply it to each task individually because not all tasks train—or overfit—at the same rate. If other regularization procedures such as weight decay are used instead of early stopping, it is important to adjust the weight decay parameters for each task individually. This is one reason to use early stopping instead of weight decay with MTL. With early stopping, we can sometimes train one MTL net for all the tasks, take snapshots of the net when performance on each task is best, and continue training the MTL net until peak performance on the last important task is reached. This is more efficient than training nets for each main tasks tasks with different weight decay parameters optimized for each net's main task. Another advantage of early stopping over weight decay for MTL is that weight decay restricts net capacity which forces sharing. As we discuss in the next section, too much sharing can be bad. Early stopping allows more independent control of regularization to prevent overfitting and the pressure for sharing. Sharing pressure can be controlled by the architecture and the size of the hidden layer(s), whereas regularization can be controlled by early stopping. This extra flexibility gives more control and better performance with MTL.

9.2.2 Sharing, Architecture, and Capacity

One important lesson we learned is that it is important not to provide too strong a bias for sharing, as this usually hurts performance. If tasks are more different than they are alike (which is often the case), it is important to allow tasks to learn reasonably independent models and overlap only where there is common structure in the learned models. For example, we observe that MTL-backprop performance often drops if the size of the shared hidden layer is much smaller than the sum of the sizes of the STL hidden layers that would provide good performance on the tasks when trained separately. Making the hidden layer tight to promote sharing usually hurts performance. (The only time we observe that a tight net helps MTL performance is with synthetic tasks that were all generated by one generator net that has a small hidden layer. Training tasks like these on a net of just the right size (or often a little larger) usually does improve performance. But real tasks are rarely this strongly related.)

Many applications of MTL-backprop work well with a single fully connected hidden layer shared equally by all tasks. Sometimes, however, more complex net architectures work better. For example, sometimes it is beneficial to have a private hidden layer for the main task and a separate public hidden layer shared by both the main task and extra tasks (see Section 6.3.2). Similarly, if some features are to be used both as inputs and as extra output tasks, architectures with disjoint hidden layers must be used to prevent those outputs from "seeing" the corresponding input signals (see 5.3). Too many private hidden layers (e.g., a private hidden layer for each task), however, can reduce the opportunities for sharing and thus reduce the benefits of MTL. We do not have principled ways to determine what architecture is best for each problem. Fortunately, simple architectures often work well, if not optimally. [Ghosn & Bengio 1997] experiment with several different architectures for MTL in backprop nets.

It might seem that deeper net architectures would work better with MTL-backprop; an extra hidden layer would make it possible for one task to form non-linear functions of the hidden-layer representation developed for other tasks. We have run a few MTL experiments with nets deeper than one hidden layer. In those experiments we did not observe benefits from using deeper nets. Moreover, the nets took longer to train. We suspect that one of the reasons we did not observe a benefit from the deeper nets is that current backprop algorithms are not very good at training deep nets. The following modification to the way deep nets are trained might prove beneficial for MTL: train a standard MTL-net with one hidden layer, and stop training when either the first task begins to overfit, the main task begins to overfit, or when the aggregate error across all tasks begins to overfit. Then insert a new (randomly initialized) hidden layer between this hidden layer and the outputs, and continue training this new deeper net. Early in training, the inputs to the new hidden layer in this deeper net are the hidden layer representation learned by the shallower MTL net. As training progresses, both the first hidden layer and the new second hidden layer are updated by backprop. The potential advantage of this approach is that the second hidden layer has the opportunity to nonlinearly combine the representations developed for different tasks in the first hidden layer. (This approach is similar to the architecture we used to combine MTL with feature nets, except that there we did not allow backpropagation to change the representation learned on the first hidden layer. See Section 6.3.3.) This process of inserting new hidden layers can be repeated until additional hidden layers begin to injure performance. We have not tried this.

9.2.3 Computational Cost

The main goal of MTL is to improve generalization, not reduce computational cost. What effect does MTL have on learning speed? In backprop nets, the MTL net is usually larger than the STL net and thus requires more computation per backprop pass. However, if all tasks need to be learned, training one MTL net usually requires less total computation than training individual STL nets for all tasks. This advantage disappears, however, if the architecture or learning rates will be optimized for each main task because training the multiple MTL nets will probably be more costly than training the multiple STL nets.

We sometimes find that tasks trained with MTL need fewer epochs than the same tasks trained alone. This partially compensates for the extra computational cost of each MTL epoch. But MTL nets often have more complex training curves—both the training and test set error curves can be multimodal for any one output—so early stopping can be more difficult. Because of this, we must sometimes train MTL nets well past the apparent early stopping point to make sure performance on the main task does not start getting better again later.

Table 9.1 shows the time required to train STL and MTL nets for four of the problems used in this thesis. These are timing results for the actual experiments. They are a realistic measure of the relative cost of STL and MTL for these problems, not the results of a theoretical analysis of complexity. If the larger MTL nets do not fit the machine's cache size as well as the smaller STL nets, the penalty for cache misses is included in the measured cost. (Theoretical analyses of algorithm complexity often ignore important issues such as this.) The results in the table are from experiments run on different workstations (Sun Sparcs, DEC Alphas, and Intel PPros) that have different memory size, memory architectures, and CPU speeds. For each problem, however, STL and MTL were run on the same architecture.

The main extra cost when training MTL nets is that MTL nets often require larger hidden layers. The number of weights in a backprop net grows linearly with the number of hidden units if the hidden layer is fully connected to the inputs and outputs. The next largest cost when training MTL nets is that sometimes MTL nets must be trained longer (more epochs) than STL nets trained on the same tasks. This is not because MTL nets train slower, but because the multimodal behavior of some MTL training curves makes early stopping more difficult; one trains longer to insure one has not stopped prematurely. The third largest cost when training MTL nets is the cost of updating the weights that connect the hidden layer to the extra outputs. Unless there are very many extra tasks, this cost is usually small. Finally, because there are extra training signals in the MTL training and test sets, these are larger than the STL training and test sets, and may not fit in computer memory as easily. The results in Table 9.1 combine all these effects. For the four tasks in

Table 9.1: Relative Speed of STL-Backprop and MTL-Backprop on several problems used in this thesis. The "Time Per Trial" columns are the clock time required to run a single trial of that problem. Because problems were run on different machines, the speed of different problems should not be compared. For each problem, however, STL and MTL were run on the same architectures and are comparable. Each problem and method (STL or MTL) were run using the number of hidden units that preliminary experiments suggested worked well. (In some cases MTL was run with fewer hidden units than was optimal because larger nets were too expensive to train. This means optimal results with MTL would take longer in some cases than those reported here.) The final column is a ratio telling how much longer it took to train MTL than STL for that problem.

Problem	Input Features	Extra Outputs	# of Hidden Units STL	# of Hidden Units MTL	Time Per Trial STL	Time Per Trial MTL	Time Ratio
Parity	8	3	100hu	100hu	2.41 hr	2.42 hr	1.00
1D-ALVINN	32	8	8hu	32hu	0.56 hr	2.57 hr	4.59
Medis	30	35	8hu	64hu	0.57 hr	5.95 hr	10.44
Port	204	33	64hu	64hu	1.50 hr	1.72 hr	1.15
Average							4.30

the table, MTL nets take on average about 4 times more computation to train than STL nets. As can be seen from the entries in the table for each problem, the difference between STL and MTL is very problem dependent.

Once an MTL net is trained, there is no extra cost using the MTL net to make predictions compared with a similar sized STL net because the extra weights from the hidden layer to the extra tasks are not needed once training is completed. Because predictions for the extra tasks are usually ignored, the weights and nodes associated with the extra tasks can be removed if the net used for prediction must be kept as small and fast as possible. Often, however, MTL nets must be larger than the STL nets for the same main task. Thus MTL nets will often be larger and slower than STL nets even if the weights and outputs for the extra tasks are removed.

In k-nearest neighbor, kernel regression, and decision trees, MTL adds little to the cost of training the MTL model. The only extra cost is the computation needed to evaluate performance on the multiple tasks instead of the one main task. This is a small constant factor that is easily dominated by other more expensive steps in these methods, such as computing distances between cases, finding nearest neighbors, finding the best threshold for splits of continuous attributes in decision trees, etc. The most significant additional cost of using MTL with these algorithms is cross-validating the λ parameters that control the relative weight of the main and extra tasks.

9.2.4 Task Selection

In many real-world domains there are so many features that feature selection is a serious problem. MTL provides machinery that allows us to use some features as inputs, some features as extra outputs, and some features as both inputs and extra outputs. Multitask learning, by giving us more options, makes the feature selection problem worse. We need to develop methods that will efficiently determine which of the available features should be used as inputs, outputs, or inputs and outputs.

Even where input features will not be used as extra outputs, we may still have a large number of extra tasks that could be used for MTL. Some of these extra tasks may be unrelated to the main task. Others might be related, but harmful, given the MTL algorithm we are using. We need task selection procedures, analogous to the feature selection procedures currently used in machine learning, that will select from a large set of potential extra tasks those most related to or most helpful to the main task. As we discuss in the next two sections, finding tasks that are related to the main task, and finding tasks that are helpful to the main task, are not necessarily the same thing.

9.2.5 Inductive Transfer Can Hurt

MTL does not always improve performance. In the Medis pneumonia domain, prediction performance dropped for high-risk patients when an extra SSE output was added to the MTL rankprop net predicting risk (see Section 4.3), even though performance on the low-risk patients improved. This result was consistent with our model of the relative strengths and weaknesses of the main and extra task on this problem. *MTL is a source of inductive bias. Some inductive biases help. Some inductive biases hurt. It depends on the problem.*

Because we are *learning* tasks, we don't *know* their internal structure in advance. Since internal structural sharing is critical to the success of MTL, there is no way we can predict in advance whether one task will benefit from MTL with another task. There are measurements that can be made (e.g., mutual information between tasks, mutual information between tasks and the input features, pairwise correlations between tasks) that might suggest that there is an opportunity for structural sharing, but these are only heuristics. Furthermore, even if we could reliably predict structural overlap between tasks, we don't know whether some particular algorithm such as backprop would be able to discover and exploit it.

We may never have an operational theory of what tasks will help or hurt for MTL. For now, the safest approach is to treat MTL as a tool that must be tested on each problem. Fortunately, on most problems where we have tried it, MTL helps. There are several reasons for this: First, the intuition about what are useful extra tasks is often correct. (Chapter 4 is our attempt to collect these heuristics in one place.) Second, most tasks that do not help the main task do not appear to hurt the main task either. We can easily create harmful extra tasks for synthetic problems, but tasks like these don't seem to arise often in practice. Finally, the benefit from the helpful extra tasks seems to be larger than the loss due to harmful or not helpful extra tasks.

9.2.6 What are *RELATED* Tasks?

One of the most important open problems in inductive transfer is to characterize, either formally or heuristically, how tasks need to be *related* for MTL to improve generalization accuracy. The lack of an adequate operational definition of task relatedness is one of the obstacles standing in the way of the development of more useful theories of inductive transfer. Some characteristics of task relatedness are already clear. If two different tasks are each the same function, but have independent noise added to their task signals, clearly the two tasks are related. If two tasks are to predict different aspects of the health of the *same* individual, these tasks are more related than two tasks to predict different aspects of the health of *different* individuals. Two tasks that are *negatively* correlated or have *negative* mutual information are still related. Tasks which have no correlation or mutual information can still be related (see Section 3.2.1) if there is correlation between the representations learn for each one by some learning algorithm.

We must be careful. There is a difference between tasks being related to each other and tasks being helpful to each other. Some related tasks may be harmful when trained with some MTL algorithms. This does not mean the tasks are not related, just that that

MTL algorithm was unable to benefit from the relationship. The algorithm needs to be improved. On the other hand, some unrelated tasks may be helpful. Just because two tasks help each other when trained together does not necessarily mean they are related. For example, sometimes injecting noise through an extra output on a backprop net improves generalization on other outputs by regularizing the hidden layer, but this does not mean the noise task is related to the other tasks it helps. And tasks that are most strongly related are not necessarily the ones that are most helpful to each other. For example, two identical tasks are maximally related, yet neither provides additional information for the other. Finally, while relatedness is symmetric, benefit may not be. For some MTL algorithm, Task A might help Task B, but Task B might hurt Task A (see Section 4.5). Although Task A and B are related to each other, one benefits the other, but not vice-versa.

The most precise definition for relatedness we have been able to devise so far is the following: *Tasks A and B are related if there exists an algorithm M such that M learns better when given training data for B as well, and if there is no modification to M that allows it to learn A this well when not given the training data for B.* While precise, this definition is not very operational. It may be easy to demonstrate that the training signals for B improve learning on A with any particular algorithm, but it is difficult to show that there do not exist other algorithms (modifications to M) that might perform as well or better given just the training signals for A.

It might seem that this notion of task relatedness misses the mark. That we don't care what tasks are related. We just care what tasks help each other. This is shortsighted. What we really want are MTL algorithms that benefit whenever tasks are related. To attempt to achieve this, we need a notion of task relatedness that goes beyond statements like "Task A improves the performance of Task B when trained with Method M." If Task A does not improve the performance of Task B when trained with Method M-, we want to know why M benefits and M- does not. And we may want to improve M- so it benefits, too. Furthermore, if Task A helps Task B on Method M just because Task A injects noise into Method M, or just because Task A reduces the capacity Method M has left for Task B, or just because Task A increases the effective learning rate of Method M, or for any similar reason, *but not because Task A and Task B are related,* we want to know this. There probably are better, more controllable ways of introducing this effect into Method M, and once we have that control we won't need the training signals for Task A to train Task B.

We may never have a theory of relatedness that allows us to reliably predict which tasks will help or hurt each other when used for inductive transfer. Because of this, we are focusing part of our effort on ways to efficiently determine which tasks are beneficially related to each other (task selection) and on developing methods robust to interference from unrelated extra tasks. Algorithms that automatically adjust the MTL bias using heuristics or cross-validation are important steps for making MTL more useful in practice. It is important, however, to continue developing heuristics that better enable us to characterize and recognize related tasks in real problems.

9.2.7 Is MTL Psychologically Plausible?

MTL is not intended as a cognitive model. To use a well worn analogy, MTL is intended to help planes fly better, not to explain how birds fly. Nevertheless, it is interesting to ask if there is evidence for MTL-like mechanisms in natural learning, or if MTL might suggest directions for cognitive or neurophysiological research.

That inductive transfer occurs in human learning is probably not subject to debate. The common saying is you can't learn something unless you almost already know it. Since you need to have learned something to almost already know it, this suggests you can't learn something unless you have already learned something very much like it. When humans tackle new problems, they bring to bear what they have learned before for related problems.

It is interesting to speculate about how natural intelligences define *related* problems. There may be some evidence in psychology as to how natural intelligences cluster tasks. Natural intelligences usually use context information as cues for learning, recall, and future performance. As a contrived example, if you learn to do all your math while scuba diving, you will learn to do proofs better underwater than on dry land. The context of being underwater while learning an activity is somehow linked to future performance of that activity. Contextual clues like these might be essential elements of a system in natural intelligences that attempts to predict what tasks are related to each other. This heuristic may help prevent natural intelligences from trying to share representations between tasks that are not strongly related. Learning to cluster tasks using auxiliary information such as the context in which the data is collected, or the kind of task the data is for (e.g., recognition tasks vs. speaking tasks vs. positioning tasks), may provide valuable heuristics for finding related tasks for MTL. As one example example, if all tasks in machine learning had associated with them a short text description, methods like those used in information retrieval might be useful for clustering the tasks for MTL.

One seemingly large difference between MTL and natural intelligence is that animals don't save up all their experiences and then learn all the things they need to learn in parallel at one point in time. MTL emphasizes parallel learning and transfer, not the serial process that seems evident in natural learning. Does this make MTL psychologically implausible? Not necessarily. In natural intelligences, experience for different tasks is usually interleaved. We do not learn everything about one task before moving on to the next task. (Though there is a general trend to learning simpler tasks before learning more complex tasks, a phenomenon MTL addresses only obliquely.) The learning algorithms used for MTL in this thesis are memoryless batch algorithms. They do not themselves store experience, and they do all learning during a single explicit training phase. The algorithms depend on the system they are embedded in to collect training data and present it in a coherent fashion during training. But learning systems that use memory and interleave learning and experience for multiple tasks can perform parallel learning and transfer. An online (i.e., non-batch) model of MTL can easily perform interleaved learning of and transfer between related tasks. The key to parallel transfer is that learning never completely finishes for any one task, and all tasks have the potential to share what is learned with other related tasks.

This brief discussion is not an attempt to argue that MTL is plausible psychologically. Given how little is known about the mechanisms underlying natural intelligence, we prefer not to make any claims about the psychological plausibility of MTL. Nonetheless, given how easy it is to do MTL online, and the observation that experience in nature rarely comes in monolithic single-task chunks, we see little reason to believe that MTL is less psychologically plausible than other inductive transfer methods. Perhaps the only statement we are confident in making is that if natural intelligences do MTL, the mechanisms they use will be much more complex than the simple ones we explore in this thesis.

9.2.8 Why *PARALLEL* Transfer?

Inductive transfer is a good idea. But why try to do it by learning all the related tasks at one time? Wouldn't it be easier to learn tasks one-at-a-time, and save learning the main task until last?

Probably not. There are several reasons why parallel transfer can work better than serial transfer. Perhaps the most important is that if you are doing tabula rasa learning, almost everything you know about a task is contained in the training data for that task. If you train tasks independently, and then do transfer using only the models learned for each task (instead of the training signals for those tasks), you have probably lost information contained in the training signals but not captured by the models. The representations learned to achieve good performance on tasks trained individually may not be the representations that a learner learning a related task will find most useful. Since it is difficult to know what information in the training data is useful for another task, doing inductive transfer from models trained without considering the main task risks losing valuable information compared with training the tasks at the same time. This problem becomes more important if the technique is being optimized to maximize performance on the main task. How do you optimize the learning of extra tasks if the extra tasks are learned before training on the main task begins? What is the advantage of sequential transfer if the steps in the sequence need to be optimized to maximize the performance of the last step?

Another advantage to parallel learning is that each task has access to the representation for other tasks as they evolve during learning. MTL allows tasks to see the full trajectory of other tasks during learning, not just the final state after learning completes. It is the difference between dancing with a partner, and dancing alone after your partner has finished. The advantages of this are subtle, but potentially significant. For example, tasks might become entwined early in search in ways that lead to more complex representation shaping and sharing later in search. This can only happen if the tasks have access to each other's representation early in search.

A related difficulty is that sometimes there are several different representations that learning could use for a given problem. Where the learning system must make choices like these, it is better to select the representation that is most useful for inductive transfer (i.e., most useful to other tasks). It is not possible to do this if all learning for the previous task is completed before learning for the related task is attempted.

The key to MTL is that the two tasks must share substructure. To detect this shared substructure, it is best to search the space of hypotheses to find hypotheses that fit *both* tasks. Searching a space of hypothesis for one task before considering how these hypotheses perform on another task reduces transfer. If we choose one particular structural hypothesis for A and it is not one that shares substructure with task B, then we lose the transfer.

One significant advantage of parallel learning over serial learning is that tasks often benefit each other mutually. Task A learns better when trained with Task B, and Task B learns better when trained with Task A. Sequential transfer, by being forced to train these tasks in a sequence, cannot achieve both benefits. It might seem that this is a problem only if we are interested in both tasks. If Task A is there only to help the main task, Task B, then maybe this doesn't matter. Just train A before B. But this is not necessarily the case. We expect that Task A will help Task B more if Task A is learned better (otherwise we wouldn't worry about training Task A well, or at all). One way to learn Task A better is to train it in parallel with Task B. When task signals are available simultaneously, it is probably suboptimal to define a sequence on those tasks to train them serially.

Serial transfer, however, does seem to fit more naturally than parallel transfer in domains where tasks naturally arrive in sequences. This is the principle motivation behind life-long learning. Although we showed in Section 4.10 how parallel transfer can be used for serial transfer by using previously learned models to generate synthetic data to use for extra MTL outputs when learning the current task, it is unfortunate to incur the expense of training new models as each new task arrives. Can we reuse the old models and just add the new models to them?

O'Sullivan is exploring techniques that combine serial and parallel transfer in his thesis. One approach that may be worth trying is to use net growing algorithms such as cascade-correlation and C2 [Fahlmann 1992, 1997] which have the potential to not only be effective at growing MTL nets, but to be able to grow MTL nets dynamically as new tasks arise. The beauty of methods like C2 is that they freeze the hidden units learned previously, making them available to subsequent learning as new features. It may be that methods like C2 would have to be modified to work well with MTL. For example, adding one hidden unit at a time might promote sharing too aggressively. Perhaps hidden units need to be added N at a time, one new unit for each MTL task.

9.2.9 Intelligibility

The main goal in MTL is to improve generalization performance, not to learn more intelligible models. Breiman recently suggested the following "uncertainty" principle relating model intelligibility and accuracy:

$$Intelligibilty * Accuracy \geq Breiman'sConstant \tag{9.1}$$

This principle says you can't have it both ways. Models that are more accurate are also going to be less intelligible. Because MTL models are usually more accurate, this relation suggests they may often be less intelligible. The nets used for MTL-backprop usually have more hidden units and connection weights than those used for STL, and this probably makes understanding the net model more difficult. The main reason we have not attempted to "open up" more MTL nets trained in this thesis is because optimal MTL performance usually requires the nets to be large, and this makes analyzing the nets very hard. Even in domains where there are natural one or two dimensional representations for the input variables (e.g., image recognition domains where retinas are used for inputs), it is difficult to interpret the kind of hidden unit activation diagrams others have used when the nets are as large as most MTL nets. Even if a few of the hidden units in an MTL net appear to have sensible activation patterns, this does not necessarily mean we understand how the net as a whole makes predictions. When there are few hidden units (say 10 or less) this problem is less severe and we can attempt to understand the role of each hidden unit and how they are combined to form the final prediction.

Rather than trying to understand STL and MTL nets, we are currently trying to use methods like rule extraction to try to understand the difference between STL and MTL nets. That is, rather than learn rules that mimic the STL net and the MTL net and look for differences between those rules, we are trying to learn rules that directly represent the difference between the STL and MTL nets. If we are successful, this approach should give us insight into what the MTL net learns that makes it generalize better than the STL net.

9.2.10 MTL Thrives on Complexity

Perhaps the most important lesson we have learned from applying MTL to real problems is that the MTL practitioner must get involved *before* the problem and data have been sanitized. MTL benefits from the extra information that often would be engineered away because traditional STL techniques would not be able to use it. Few of the standard problems in collections such as the UCI Machine Learning Repository are suitable for MTL research; most of them have been carefully simplified to make them suitable for STL.

The opportunities for applying MTL often decrease as one becomes further removed from the raw data or the data collection process. MTL provides new ways of using information that are not obvious from the traditional STL point-of-view. Sometimes it is necessary to "nudge" (sometimes not so gently!) those in charge of data collection and data management to provide the extra information MTL can use. It is not uncommon to be told "You can't use that because..." or asked "What can you do with that?"

Even someone experienced in applying MTL can easily miss opportunities for using it. Our best suggestion for applying MTL to a new domain is to collect, or at least consider collecting, every bit of information that you can possibly imagine collecting, and then throw away (or don't collect) only those pieces which you are absolutely certain you can't use. Even then you'll probably be wrong half the time. Part of what makes MTL and inductive transfer so exciting is that it provides hooks for so many different kinds of information that traditionally have been difficult to exploit.

9.2.11 Combining MTL and Boosting

Bagging [Breiman 1994], Boosting [Schapire 1990][Freund 1995], Error-Correcting Codes [Dietterich & Bakiri 1995], and other voting schemes that combine multiple predictions for a task from different learned models are an exciting recent advance in machine learning. Hopefully the benefits provided by these mechanisms are partially orthogonal to the benefits of MTL. If this is the case, combining MTL with boosting methods should generalize better than either method alone. One interesting direction to explore along these lines is the discovery in this thesis that training multiple copies of the same task on an MTL-backprop net can improve the performance. We suspect that this may be due to a boosting-like mechanism that arises because the net is initialized with different random weights from the copies of the task on the outputs to the shared hidden units.

9.2.12 MTL With Other Learning Methods

This thesis discusses MTL mainly in the context of artificial neural nets trained with back-propagation and in methods like k-nearest neighbor and kernel regression. MTL is not any particular algorithm. It is an approach to learning that attempts to improve accuracy by leveraging the information contained in the training signals of related tasks. MTL can be applied to many different learning algorithms, and often there are many ways to do MTL with each algorithm.

Sketch of an Algorithm for MTL in Decision Trees

The basic recursive step in the top-down induction of decision trees (TDIDT) is to determine which of the available splits to add to the current node in a growing decision tree. Typically this is done using an information gain metric that measures how much class purity is improved by the available splits. Class purity is a measure of how much accuracy on the task is improved by adding the split.

Decision trees are usually single task: leaves assign cases to one class for one task. Multitask decision trees are possible if leaves assign cases to classes for many tasks. For example, a leaf might assign cases to class A for Task 1, class C for Task 2, etc... What is the advantage of multiclass decision trees? Because decision trees are induced top-down in greedy fashion, significant effort is spent during the greedy induction process to find *good* splits. Once a split is installed in the tree, all subsequent decisions are affected by it. Moreover, splits in a decision tree cause data rapidly to become sparse. We usually cannot afford to install useless or suboptimal splits if we do not have a large training set.

Usually, the only information available is how well the splits separate classes on a single task. In a multitask decision tree one evaluates splits by how well they separate classes from multiple tasks. If the multiple tasks are related, preferring splits that have utility to multiple tasks should improve the quality of the selected splits. The basic assumption is that the boundaries between classes for different but related tasks tend to lay in similar regions of the input space. Preferring splits that benefit multiple tasks should help class boundaries be found more reliably.

The benefit of preferring splits that have utility across multiple tasks should be most dramatic when inducing decision trees from small samples. When there is little training data it is important to find splits that properly discriminate the underlying structure of the problem. Failure to do this results in leaf classes with high purity on the training data that do not represent regions of high purity for the real problem distribution.

How do we select splits good for multiple tasks? The basic approach is straightforward: compute the information gain of each split *for each task individually*, combine the gains, and select the split with the best aggregate performance. The MTL decision tree algorithm presented in [Caruana 1993] combines task gains by averaging them; the selected splits are the ones whose average utility across all tasks is highest.

There is a problem, however, with simple averaging. Splits good for Task 1 are not necessarily good for Task 2. Because each split in a decision tree affects all nodes below it, it is difficult for a multitask decision tree to *isolate* tasks that differ. This is something MTL-backprop nets can do if there is enough capacity for hidden units to specialize to different tasks. MTL backprop often *worsens* performance if there is insufficient capacity for this specialization.

As the number of tasks grows large, fewer splits in the tree will be optimal for any one task. Recursive splitting dilutes the data before the structure needed for any one task is learned. In other words, tasks can *starve* in a multiclass decision tree. This is bad if the task that starves is the main task.

The goal of MTL is to improve performance on one task by leveraging information contained in the training signals of other tasks. We do not care if the MTL decision tree grown for Task 1 performs well on Task 2. If Task 2 is also important, we can grow a separate MTL decision tree for it. This gives us freedom to use the splits preferred by other tasks only if they help the main task.

Averaging the information gain across all tasks places all tasks on equal footing. Using a weighted average allows us to bias splits in favor of specific tasks. Assume the main task has weight 1. If the weight for some extra task is near 0, that task is ignored because it contributes little to the aggregate information gain. Conversely, if a task has weight $\gg 1$, the installed splits are very sensitive to how much they gain for this task.

Hillclimbing on a hold-out set can be used to learn task weights that yield good generalization on the main task. Unfortunately, the partial derivatives of performance with respect to task weights are discontinuous and thus not differentiable: small weight changes often have no affect on the learned tree, and large changes in performance sometimes occur when the test installed at some node suddenly changes. So we can't use gradient descent. Fortunately, there are simple accounting tricks that can be used at interior nodes in the decision tree to keep track of the smallest weight changes that will alter the learned tree. This allows us to use steepest descent hillclimbing to learn the task weights.

This approach to learning task weights requires decision tree induction be fast enough to run many times. It may not be practical for problems with large data sets and many attributes. There is also a danger of overfitting the test set if it is too small. An attractive feature of this scheme is that after the weights have been learned, they can be inspected to see which extra tasks are most beneficially related to the main task.

9.2.13 Combining MTL With Other Learning Methods

There are other learning methods that also might benefit from MTL. As one example, reinforcement learning is a challenging learning method where the supervisory signals received from the environment are infrequent rewards (success or failure, or a positive or negative reward on some scale). This makes learning very difficult because the learner may need to perform a complex series of actions before a reward is received. (By contrast, in traditional supervised learning each action has an associated training signal that guides the learner towards good performance.) Training a reinforcement learner on multiple related reinforcement learning problems that give rewards in different states and for different sequences of behavior might be one way to guide reinforcement learning to learn good behaviors. For example, a reinforcement learner that is learning to navigate from a start position to a goal position, might also be given rewards for related tasks such as crossing the room, following a wall, passing through a doorway, crossing its own trajectory a second time (a negative reward), trajectories that are smooth, trajectories that do not hit objects, trajectories that do not go too close to objects, etc. These extra tasks can be viewed as a way of giving reinforcement learning partial credit for successfully executing behaviors beneficial to success on the main task.

Applying MTL to other learning methods is an important direction of future research. The two learning areas we are most interested in applying MTL to are methods for scientific discovery and methods for unsupervised learning. We were led to first consider MTL by trying to answer the simple question "how could one use a backprop net to do scientific discovery?" A backprop net trained on a single set of data, say data points for a planet orbiting the sun, would likely learn a simple interpolating model for the data. It would not discover anything general such as the law of gravitation. But a backprop net trained on dozens of different problems, all of which depended on gravity in some way (e.g., planets orbiting the sun, stars orbiting the galactic center, moons orbiting planets, apples falling to earth) might, if biased to learn a single comprehensive model that captured all these discovery tasks, discover something like gravity. This is, of course, a fanciful example. But it was the original motivation for this thesis, and still poses a challenging and potentially rewarding direction for future research.

Forging a stronger connection between MTL and unsupervised learning is important because it is now clear that MTL depends implicitly on unsupervised learning. We may have supervisory training signals for each task individually, but we are not given training signals or supervision about how tasks are related and what should be shared between them. This must be discovered by the MTL algorithm via an unsupervised learning mechanism. Better understanding the role of this unsupervised learning in MTL, and learning how to make it work better, are important directions for improving MTL.

Chapter 10

Bibliography

Abu-Mostafa, Y. S., "Learning from Hints in Neural Networks," *Journal of Complexity,* 1990, 6(2), pp. 192–198.

Abu-Mostafa, Y. S., "Hints and the VC Dimension," *Neural Computation,* 1993, 5(2).

Abu-Mostafa, Y. S., "Hints," *Neural Computation,* 1995, **7,** pp. 639-671.

Baluja, S., "Expectation-Based Selective Attention," Doctoral Thesis, Carnegie Mellon University, *CMU-CS-96-182,* 1996.

Baluja, S. and Pomerleau, D. A., "Using the Representation in a Neural Network's Hidden Layer for Task-Specific Focus of Attention," *Proceedings of the International Joint Conference on Artificial Intelligence 1995,* IJCAI-95, Montreal, Canada, 1995, pp. 133-139.

Baum, E. B. and Haussler, D., "What Size net gives valid Generalization?," *Neural Computation,* 1989, **1**:1, pp. 151-160.

Baxter, J., "Learning Internal Representations," Ph.D. Thesis, The Flinders Univeristy of South Australia, Dec. 1994.

Baxter, J., "Learning Internal Representations," *Proceedings of the 8th ACM Conference on Computational Learning Theory,* (COLT-95), Santa Cruz, CA, 1995.

Baxter, J., "A Bayesian/Information Theoretic Model of Bias Learning,", *Proceedings of the 9th International Conference on Computational Learning Theory,* (COLT-96), Desenzano del Gardo, Italy, 1996.

Becker, S. and Hinton, G. E., "A Self-organizing Neural Network that Discovers Surfaces in Random-dot Stereograms," *Nature,* 1992, **355,** pp. 161-163.

Breiman, L., "Bagging Predictors," TR No. 421 (1994), Department of Statistics, University of California, Berkeley, 1994.

Breiman, L. and Friedman, J. H., "Predicting Multivariate Responses in Multiple Linear Regression," 1995, ftp://ftp.stat.berkeley.edu/pub/users/breiman/curds-whey-all.ps.Z.

Caruana, R., "Multitask Learning: A Knowledge-Based Source of Inductive Bias," *Proceedings of the 10th International Conference on Machine Learning,* ML-93, University of Massachusetts, Amherst, 1993, pp. 41-48.

Caruana, R., "Multitask Connectionist Learning," *Proceedings of the 1993 Connectionist Models Summer School,* 1994, pp. 372-379.

Caruana, R., "Learning Many Related Tasks at the Same Time with Backpropagation," *Advances in Neural Information Processing Systems 7,* (Proceedings of NIPS-94), 1995, pp. 656-664.

Caruana, R., Baluja, S., and Mitchell, T., "Using the Future to "Sort Out" the Present: Rankprop and Multitask Learning for Medical Risk Prediction," *Advances in Neural Information Processing Systems 8,* (Proceedings of NIPS-95), 1996, pp. 959-965.

Caruana, R. and de Sa, V. R., "Promoting Poor Features to Supervisors: Some Inputs Work Better As Outputs," to appear in *Advances in Neural Information Processing Systems 9,* (Proceedings of NIPS-96), 1997.

Cooper, G. F., Aliferis, C. F., Ambrosino, R., Aronis, J., Buchanan, B. G., Caruana, R., Fine, M. J., Glymour, C., Gordon, G., Hanusa, B. H., Janosky, J. E., Meek, C., Mitchell, T., Richardson, T., and Spirtes, P., "An Evaluation of Machine Learning Methods for Predicting Pneumonia Mortality," *Artificial Intelligence in Medicine 9,* 1997, pp. 107-138.

Craven, M. and Shavlik, J., "Using Sampling and Queries to Extract Rules from Trained Neural Networks," *Proceedings of the 11th International Conference on Machine Learning,* ML-94, Rutgers University, New Jersey, 1994, pp. 37-45.

Davis, I. and Stentz, A., "Sensor Fusion for Autonomous Outdoor Navigation Using Neural Networks," *Proceedings of IEEE's Intelligent Robots and Systems Conference,* 1995.

Dent, L., Boticario, J., McDermott, J., Mitchell, T., and Zabowski, D., "A Personal Learning Apprentice," *Proceedings of 1992 National Conference on Artificial Intelligence,* 1992.

de Sa, V. R., "Learning Classification with Unlabelled Data," *Advances in Neural Information Processing Systems 6,* (Proceedings of NIPS-93), 1994, pp. 112-119.

Dietterich, T. G., Hild, H., and Bakiri, G., "A Comparative Study of ID3 and Backpropagation for English Text-to-speech Mapping," *Proceedings of the Seventh International Conference on Artificial Intelligence,* 1990, pp. 24-31.

Dietterich, T. G., Hild, H., and Bakiri, G., "A Comparison of ID3 and Backpropagation for English Text-to-speech Mapping," *Machine Learning,* 18(1), 1995, pp. 51-80.

Dietterich, T. G. and Bakiri, G., "Solving Multiclass Learning Problems via Error-Correcting Output Codes," *Journal of Artificial Intelligence Research,* 1995, 2, pp. 263-286.

Fine, M. J., Singer, D., Hanusa, B. H., Lave, J., and Kapoor, W., "Validation of a Pneumonia Prognostic Index Using the MedisGroups Comparative Hospital Database," *American Journal of Medicine*, 1993.

Fisher, D. H., "Conceptual Clustering, Learning from Examples, and Inference," *Proceedings of the 4th International Workshop on Machine Learning*, 1987.

Freund, Y., "Boosting a Weak Learning Algorithm by Majority," *Information and Computation,* 121(2):256-285, 1995.

Ghahramani, Z. and Jordan, M. I., "Supervised Learning from Incomplete Data Using an EM Approach," Advances in Neural Information Processing Systems 6, (Proceedings of NIPS-93,) 1994, pp. 120-127.

Ghahramani, Z. and Jordan, M. I., "Mixture Models for Learning from Incomplete Data," *Computational Learning Theory and Natural Learning Systems, Vol. IV*, R. Greiner, T. Petsche and S.J. Hanson (eds.), Cambridge, MA, MIT Press, 1997, pp. 67-85.

Ghosn, J. and Bengio, Y., "Multi-Task Learning for Stock Selection," *Advances in Neural Information Processing Systems 9,* (Proceedings of NIPS-96), 1997.

Hinton, G. E., "Learning Distributed Representations of Concepts," *Proceedings of the 8th International Conference of the Cognitive Science Society,* 1986, pp. 1-12.

Holmstrom, L. and Koistinen, P., "Using Additive Noise in Back-propagation Training," *IEEE Transactions on Neural Networks,* 1992, 3(1), pp. 24-38.

Hsu, G. T. and Simmons, R., "Learning Footfall Evaluation for a Walking Robot," *Proceedings of the 8th International Conference on Machine Learning,* 1991, pp. 303-307.

Lang, K. "NewsWeeder: Learning to Filter News," *Proceedings of the 12th International Conference on Machine Learning,* 1995, pp. 331-339.

Le Cun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackal, L. D., "Backpropagation Applied to Handwritten Zip-Code Recognition," *Neural Computation,* 1989, 1, pp. 541-551.

Little, R. J. A. and Rubin, D. B., *Statistical Analysis with Missing Data*, 1987, Wiley, New York.

Liu, H. and Setiono, R., "A Probibilistic Approach to Feature Selection—A Filter Solution," *Proceedings of the 13th International Conference on Machine Learning,* ICML-96, Bari, Italy, 1996, pp. 319-327.

Martin, G. L. and Pittman, J. A., "Recognizing Hand-Printed Letters and Digits Using

Backpropagation Learning," *Neural Computation,* 1991, **3**, pp. 258-267.

Martin, J. D., "Goal-directed Clustering," *Proceedings of the 1994 AAAI Spring Symposium on Goal-directed Learning,* 1994.

Martin, J. D. and Billman, D. O., "Acquiring and Combining Overlapping Concepts," *Machine Learning*, 1994, 16, pp. 1-37.

Mitchell, T., "The Need for Biases in Learning Generalizations," Rutgers University: *CBM-TR-117,* 1980.

Mitchell, T., Caruana, R., Freitag, D., McDermott, J., and Zabowski, D., "Experience with a Learning Personal Assistant," *Communications of the ACM: Special Issue on Agents,* July 1994, 37(7), pp. 80-91.

Munro, P. W. and Parmanto, B., "Competition Among Networks Improves Committee Performance," to appear in *Advances in Neural Information Processing Systems 9,* (Proceedings of NIPS-96), 1997.

O'Sullivan, J. and Thrun, S., "Discovering Structure in Multiple Learning Tasks: The TC Algorithm," *Proceedings of the 13th International Conference on Machine Learning,* ICML-96, Bari, Italy, 1996, pp. 489-497.

Pomerleau, D. A., "Neural Network Perception for Mobile Robot Guidance," Doctoral Thesis, Carnegie Mellon University: *CMU-CS-92-115,* 1992.

Pratt, L. Y., Mostow, J., and Kamm, C. A., "Direct Transfer of Learned Information Among Neural Networks," *Proceedings of AAAI-91*, 1991.

Pratt, L. Y., "Non-literal Transfer Among Neural Network Learners," Colorado School of Mines: *MCS-92-04,* 1992.

Pratt, L. Y., Mostow, J., and Kamm, C. A., "Direct Transfer of Learned Information Among Neural Networks," *Proceedings of AAAI-91,* 1991.

Quinlan, J. R., "Induction of Decision Trees," *Machine Learning,* 1986, 1, pp. 81-106.

Quinlan, J. R., *C4.5: Programs for Machine Learning,* Morgan Kaufman Publishers, 1992.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J., "Learning Representations by Back-propagating Errors," *Nature,* 1986, 323, pp. 533-536.

Sejnowski, T. J. and Rosenberg, C. R., "NETtalk: A Parallel Network that Learns to Read Aloud," John Hopkins: *JHU/EECS-86/01,* 1986.

Schapire, R., "The Strength of Weak Learnability," *Machine Learning,* 5(20:197-227, 1990.

Sharkey, N. E. and Sharkey, A. J. C., "Adaptive Generalisation and the Transfer of Knowledge," University of Exeter: *R257,* 1992.

Sill, J. and Abu-Mostafa, Y., "Monotonicity Hints," to appear in *Neural Information Processing Systems 9,* (Proceedings of NIPS-96), 1997.

Simard, P., Victorri, B., LeCun, Y., and Denker, J., "Tangent Prop—A Formalism for Specifying Selected Invariances in an Adaptive Neural Network," *Advances in Neural Information Processing Systems 4,* (Proceedings of NIPS-91) 1992, pp. 895-903.

Suddarth, S. C. and Holden, A. D. C., "Symbolic-neural Systems and the Use of Hints for Developing Complex Systems," *International Journal of Man-Machine Studies,* 1991, 35(3), pp. 291-311.

Suddarth, S. C. and Kergosien, Y. L., "Rule-injection Hints as a Means of Improving Network Performance and Learning Time," *Proceedings of the 1990 EURASIP Workshop on Neural Networks,* 1990, pp. 120-129.

Sutton, R., "Adapting Bias by Gradient Descent: An Incremental Version of Delta-Bar-Delta,", Proceedings of the 1-th International Conference on Artificial Intelligence (AAAI-92), 1992, pp. 171-176.

Thrun, S. and Mitchell, T., "Learning One More Thing," Carnegie Mellon University: *CS-94-184,* 1994.

Thrun, S., "Lifelong Learning: A Case Study," Carnegie Mellon University: *CS-95-208,* 1995.

Thrun, S., "Is Learning the N-th Thing Any Easier Than Learning The First?," *Advances in Neural Information Processing Systems 8,* (Proceedings of NIPS-95), 1996, pp. 640-646.

Thrun, S., *Explanation-Based Neural Network Learning: A Lifelong Learning Approach,* 1996, Kluwer Academic Publisher.

Tresp, V., Ahmad, S., and Neuneier, R., "Training Neural Networks with Deficient Data," *Advances in Neural Information Processing Systems 6*, (Proceedings of NIPS-93), 1994, pp. 128-135.

Utgoff, P. and Saxena, S. "Learning a Preference Predicate," *Proceedings of the 4th International Conference on Machine Learning,* 1987, pp. 115-121.

Valdes-Perez, R., and Simon, H., "A Powerful Heuristic for the Discovery of Complex Patterned Behavior," *Proceedings of the 11th International Conference on Machine Learning,* ML-94, Rutgers University, New Jersey, 1994, pp. 326-334.

Waibel, A., Sawai, H., and Shikano, K., "Modularity and Scaling in Large Phonemic Neural Networks," *IEEE Transactions on Acoustics, Speech and Signal Processing,* 1989, 37(12), pp. 1888-1898.

Weigend, A., Rumelhart, D., and Huberman, B., "Generalization by Weight-Elimination with Application to Forecasting," *Advances in Neural Information Processing Systems 3*, (Proceedings of NIPS-90), 1991, pp. 875-882.

Appendix A

Net Size and Generalization in Backprop Nets

A.1 Introduction

This appendix summarizes the results from experiments we ran in 1992 and 1993.

The conventional wisdom is that artificial neural networks that are too big generalize poorly. In this appendix we present empirical results that suggest otherwise: if early stopping is used to prevent overfitting, excess capacity does not significantly reduce the generalization performance of fully connected feed-forward backprop nets. Moreover, too little capacity hurts generalization performance more than too much capacity.

Analysis suggests that all backprop nets, regardless of size, learn task subcomponents in similar sequence. Big nets pass through intermediate stages similar to those learned by small nets. Thus early stopping can stop big nets at a point that yields generalization performance comparable to smaller nets. If the big net is too large, the penalty is higher computational cost, not poorer generalization. But if a small net is too small, bigger nets will generalize better.

This work is important for this thesis because an MTL net being trained on multiple tasks needs more capacity than the STL net being trained on one of those tasks. If we are to perform a fair comparison between the results obtained with STL and MTL, we need to train STL and MTL using net sizes appropriate to each method. Unfortunately, because backprop nets are expensive to train, it is often infeasible to search for the optimal net size. Fortunately—and quite unexpectedly—our experiments suggest that generalization performance is remarkably insensitive to net size as long as the net is large enough. Excess capacity rarely hurts generalization if early stopping is used to prevent overfitting. This is great news. It means we can do a fair comparison between STL and MTL as long as we don't use nets that are too small with either method. It is more feasible to determine what net size is large enough than to determine what net size is optimal. In fact, if this work had suggested generalization performance was so sensitive to net size that it was necessary to find the optimal net size before each experiment, we would not have pursued this thesis using neural nets.

A.2 Why Nets that are "Too Big" Should Generalize Poorly

It is commonly believed that artificial neural networks that are too big will generalize poorly. The argument for this is that if the net has too much capacity, then it is more likely to learn to do *table lookup* on the training data than to generalize on it. The way to obtain good generalization is to restrict capacity so that the net is *forced* to generalize because it has insufficient capacity to memorize the training data.

This argument is consistent with a VC-dimension analysis of net capacity and generalization. The more free parameters in the net the larger the VC-dimension of the hypothesis space for the net (i.e., the larger the number of hypotheses the net could represent by changing weights). The higher the VC-dimension, the less likely the training sample is large enough to select a correct or nearly correct net hypothesis. [Baum & Haussler 1989]

A.3 An Empirical Study of Generalization vs. Net Capacity

A.3.1 Goals

Several groups using backprop have noted that performance on their task does not worsen (and sometimes continues to improve) as they try larger networks.

> We find only marginal and inconsistent indications that constraining net capacity improves generalization. [Martin & Pittman 1991]

Unfortunately, the anecdotal evidence suffers methodological flaws. It comes from experiments that use questionable criteria to halt training and that do not consider large enough networks. This is not a critique of the studies. The studies were motivated to achieve performance on their respective tasks, not to investigate capacity effects.

Our goal is to do a methodologically sound investigation of capacity effects in backprop nets. We want to empirically answer these questions:

1. How sensitive is generalization performance to network capacity? Is it critical to find just the right size, or is performance insensitive to small changes in capacity?

2. Does excess capacity reduce generalization performance as is widely believed? If so, how much?

3. How effective is early stopping at mitigating the effects of overfitting?

4. Do large networks learn qualitatively different internal representations than small networks?

A.3.2 Methodology

We selected seven test problems and trained nets of different sizes (2 to 800 hidden units) on them using backpropagation. We used hold-out sets to measure generalization performance during training. Where possible, we collected complete generalization curves, i.e., we trained until generalization performance began to fall or became so flat that we were reasonably confident it would not improve later. Where possible we ran multiple trials. We kept training sets small to make generalization challenging. The study required approximately one Sparc year of computation.

The problems we used are:

- NETtalk [Sejnowski 1986]
- parity (7 bit and 12 bit)
- an inverse kinematic model for a robot arm[1]
- two sensor modeling tasks using real sonar data collected from a real robot
- vision data used to steer an autonomous vehicle [Pomerleau 1992]

Some of these are boolean, others are continuous. Some have noise, others are noise-free. Some have large numbers of inputs or outputs, others have small numbers of inputs or outputs. Some are real, others are synthetic.

A.3.3 Results

Figure A.1 shows the generalization curves we obtained on four of the test problems. The results surprised us. On a few tasks, nets that were too large did have poorer peak generalization performance than smaller networks. But where there was a drop in generalization performance, it was always very small. Many replications were required before statistical tests could confirm that the

[1] Many thanks to Sebastian Thrun for letting us use his robot arm simulation code.

differences were statistically significant. Moreover, the data suggest that generalization performance is more likely to be hurt by using a network that is too small than by using one that is too large. It is better to err on the side of making the network too large than too small if generalization performance (and not training time) is the important criterion.

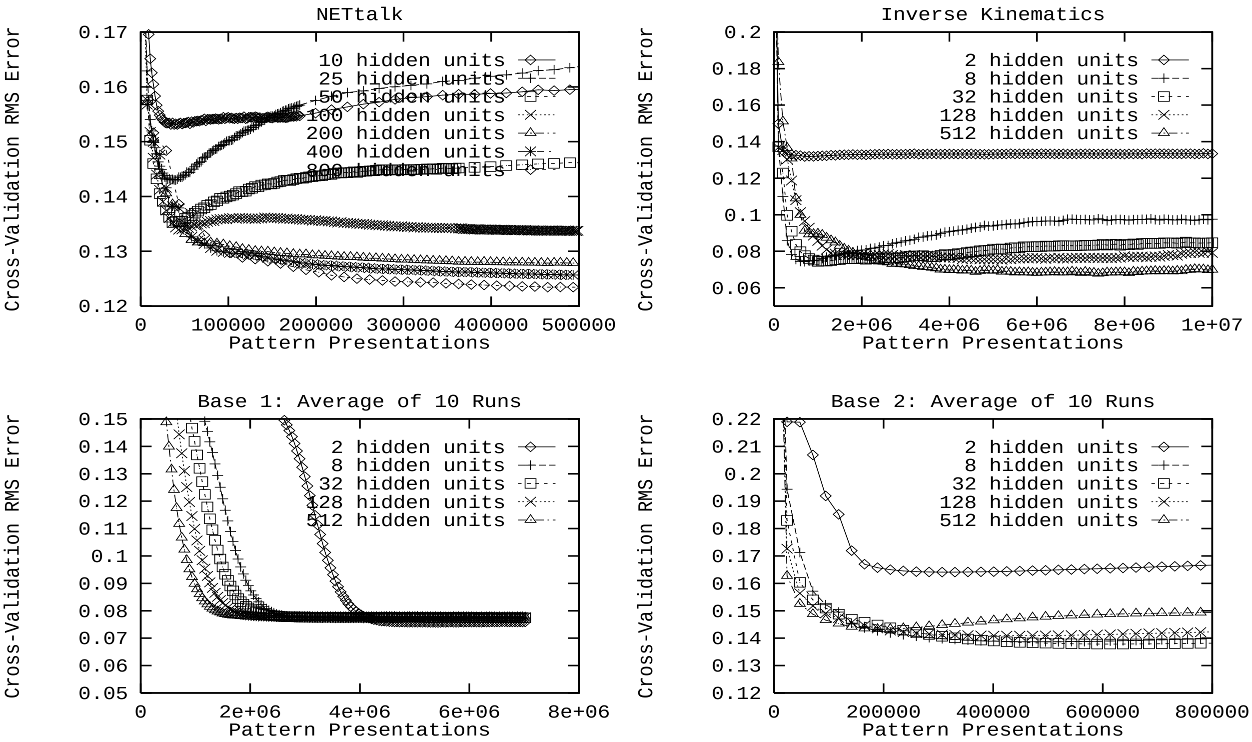


Figure A.1: Generalization Performance vs. Network Size for Four of the Test Problems

For most tasks and net sizes we trained well beyond the point where generalization performance peaked. Because we had such complete generalization curves, we noticed something we did not expect. On some tasks, small nets overfitted considerably. We conclude that early stopping is critical for nets of all sizes—not just ones that are *too big.* It is *not* safe to assume that because a net has restricted capacity that it is unlikely to overfit the data.

A.4 Why Excess Capacity Does Not Hurt Generalization

A proper theoretical analysis of net capacity and generalization must to take into account the search procedure used to find hypotheses consistent with the data. If backpropagation is used to train the net, hypotheses of different complexity are not given equal opportunity. Usually one initializes weights to small values. Weights become large only when the data require it and when the number of weight updates is large enough to allow it. Thus backprop is biased to consider net hypotheses with small weights before considering hypotheses with large weights. Net with larger weight ranges have greater representational power, so this is tantamount to searching simpler hypotheses before searching more complex hypotheses.

We analyzed *what* nets of different sizes learned *as* they train. We compared the input/output behavior of networks at different stages of learning on large samples of test patterns drawn from their domain. For example, we compared the input/output behavior of a large net as it trained with the behavior of smaller nets as they trained on the same problem.

We discovered that large nets go through intermediate stages during training similar (in I/O behavior) to smaller nets. That is, a large net first learns something most similar to what a very small net learns. Then it begins to learn what somewhat larger nets learn. Later, its behavior is most similar to intermediate sized nets. And so on. Thus nets with excess capacity first learn what small nets can learn. Then, as small nets run out of capacity, large nets begin to behave more like intermediate size nets...

If the large net is too big, early stopping will detect where its generalization performance begins to drop. At this point it is functionally similar to *some* smaller net trained on that task. The only penalty of using a net that is too big is the extra computation required to train it. Early stopping provides an apparently reliable means of stopping the training on this net at a stage functionally equivalent to what might have been obtained with a smaller net. If the net is not too big, then it will perform better than smaller net.

A.5 Conclusions

1. Early stopping is just as important with small nets as it is with large nets.
2. On some tasks, generalization performance does decrease with excess net capacity. But generalization performance is remarkably insensitive to excess net capacity. When excess capacity reduces generalization, it reduces it very little.
3. Using nets that are too small hurts generalization more than using nets that are too large.
4. Nets with excess capacity appear to go through a sequence of stages of learning that are functionally similar to what smaller nets learn.
5. When comparing methods that need different sized nets (e.g., STL and MTL), a fair comparison can be made if one is carefully not to use nets that are too small. One does not need to precisely tune net size to each task or set of tasks.

A.6 Final Note

Since completing this empirical study in 1992–1993, we have trained thousands of backprop nets on more than a dozen additional problems. For many of these problems we performed preliminary studies to quickly assess the effect of net size on generalization performance. Because computers have become significantly more powerful, in some of these studies we were able to consider nets with thousands of hidden units. Also, we ran some of later studies using conjugate gradient backprop instead of steepest descent. The results of all our experiments are consistent with the results we report here. We find little, if any, loss in generalization performance as nets become very large if early stopping is used to halt training. Where there is some small sensitivity to excess net capacity, the net size that is found to be optimal is usually much larger than one might expect. For example, it is not uncommon for the optimal net size on a problem with a few dozen inputs, a few outputs, and several hundred training patterns to be 1000 hidden units. Given that a net this size contains well over 10,000 free parameters, it is clear that the traditional rules that the number of free parameters should be less than the number of training cases does not apply to backprop nets trained with early stopping.

Appendix B

Rank-Based Error Metrics

The methods described in this appendix are applicable to domains where the goal is to learn to rank instances, usually according to some unknown function or probability distribution. Instead of learning the function or probability Often we according to some unknown function such a probability function. In addition to medical decision making, this class includes problems as diverse as investment analysis in financial markets.

B.1 Motivating Problem: Pneumonia Risk Prediction

The Medis Pneumonia Database [Fine et al. 1995] contains 14,199 patients diagnosed with pneumonia. The database indicates whether each patient lived or died. 1,542 (10.9%) of the patients died. The most useful decision aid for this problem would predict which patients will live or die. But this is too difficult. In practice, the best that can be achieved is to estimate a probability of death (POD) from the observed symptoms. In fact, it is sufficient to learn to *rank* patients by POD so lower risk patients can be discriminated from higher risk patients. The patients at least risk may then be considered for outpatient care.

The performance criterion used by others working with the Medis database [Cooper et al. 1996] is the accuracy with which one can select a prespecified fraction of the patient population that do not die. For example, given a population of 10,000 patients, find the 20% of this population at *least* risk. To do this we learn a risk model and a threshold for this model that allows 20% of the population (2000 patients) to fall below it. If 30 of the 2000 patients below this threshold died, the error rate is $30/2000 = 0.015$. We say that the error rate for FOP 0.20 is 0.015 for this model ("FOP" stands for fraction of population). In this paper we consider FOPs 0.1, 0.2, 0.3, 0.4, and 0.5. Our goal is

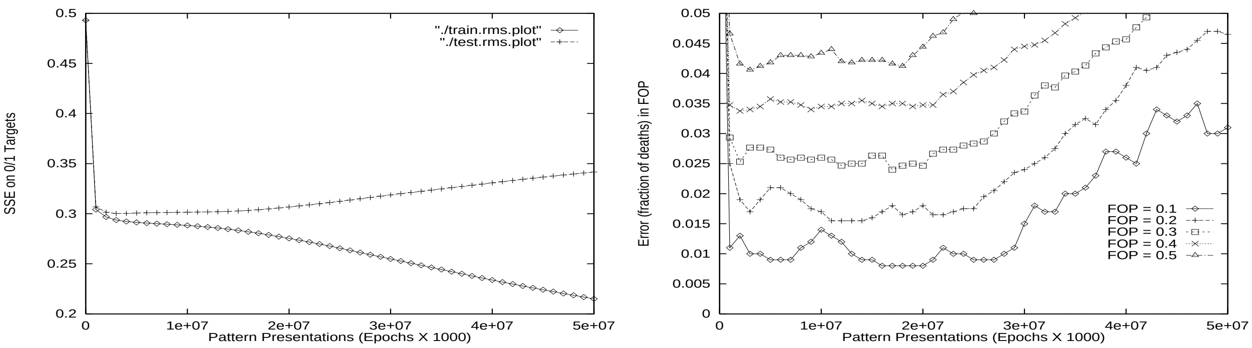


Figure B.1: Learning Curves (left graph) and Error Rates For Each FOP (right graph)

to learn models and model thresholds, such that the error rate at each FOP is minimized. Models with acceptably low error rates might then be employed to help determine which patients do not require hospitalization.

B.2 The Traditional Approach: SSE on 0/1 Targets

The straightforward approach to this problem is to use backprop to train a net to learn to predict which patients live or die, and then use the real-valued predictions of this net to sort patients by risk. This net has 30 inputs, 1 for each of the observed patient measurements, a hidden layer with 8 units[1], and a single output trained with 0=lived, 1=died.[2] Given an infinite training set, a net trained this way should learn to predict the probability of death for each patient, not which patients live or die. In the real world, however, where we rarely have an infinite number of training cases, a net will overfit and begin to learn a very nonlinear function that outputs values near 0/1 for cases in the training set, but which does not generalize well. In this domain it is critical to use early stopping to halt training before this happens.

Figure B.1 shows the learning curve for fully connected feedforward nets with 8 hidden units trained as described above with learning rate = 0.1 and momentum = 0.9.[3] It is clear from the plot that significant overfitting can occur. Figure 1 also shows the error rates for the different FOP values

[1]To make comparisons between methods fair, we first found hidden layer sizes and learning parameters that performed well for each method.

[2]Different representations such as 0.15/0.85 and different error metrics such as cross entropy did not perform better than SSE on 0/1 targets.

[3]To make the comparison between the methods fair, the size of the hidden layer and the learning parameters have been tuned for each method via preliminary experiments to find settings that yielded good performance with that method.

(measured on the 1K halting set, not on final 12K test set) as a function of training. Note that minimum FOP error does *not* occur at the same epoch where SSE is minimized. In fact, different FOPs reach minimum error at different epochs.

The fact that error rate performance at FOPs between 0.1 and 0.5 do not reach their best value when SSE on the test set is minimized suggests two things. First, better performance can be obtained by halting training based on FOP error rates instead of SSE. Second, if SSE on 0/1 targets does not correlate well with our performance criterion, perhaps it should not be used as the training criterion. The second issue is addressed by rankprop in the next section.

Given that our goal is to predict patient risk, this represents a serious lack of the appropriate training information that motivates our use of rank-based learning methods.

Table B.1 shows the error rates of nets trained with SSE on 0/1 targets for the five FOPs. Each entry is the mean of ten trials. The first entry in the table indicates that on average, in the 10% of the test population predicted by the nets to be at least risk, 1.4% died. We do not know the best achievable error rates for this data.

Table B.1: Error Rates of SSE on 0/1 Targets

FOP	0.1	0.2	0.3	0.4	0.5
Error Rate	.0140	.0190	.0252	.0340	.0421

B.3 Rankprop

Because the goal is to find the fraction of the population least likely to die, it is sufficient just to learn to rank patients by risk. Rankprop learns to rank patients without learning to predict mortality. "Rankprop" is short for "backpropagation using sum of squares errors on estimated ranks". The basic idea is to sort the training set using the target values, scale the ranks from this sort (we scale uniformly to [0.25,0.75] with sigmoid output units), and use the scaled ranks as target values for standard backprop with SSE instead of the 0/1 values in the database.

Ideally, we'd rank the training set by the true probabilities of death. Unfortunately, all we know is which patients lived or died. In the Medis database, 89% of the target values are 0's and 11% are 1's. There are many possible sorts consistent with these values. Which sort should backprop try to fit? It is the large number of possible sorts of the training set that makes backpropagating ranks challenging. Rankprop solves this problem by using the net model *as it is being learned* to order the training set *when target values are tied.* In this database, where there are many ties because there are only two target values, finding a proper ranking of the training set is a serious problem. Rankprop learns to adjust the target ranks *of* the training set at the same time it is learning to predict ranks *from* that training set.

How does rankprop do this? Rankprop alternates between rank passes and backprop passes. On the rank pass it records the output of the net for each training pattern. It then sorts the training patterns using the *target* values (0 or 1 in the Medis database), *but using the network's predictions for each pattern as a* **secondary** *sort key to break ties.*[4] The basic idea is to find the legal rank of the target values (0 or 1) maximally consistent with the ranks the current model predicts. This *closest match* ranking of the target values is then used to define the target ranks used on the next backprop pass through the training set. Rankprop's pseudo code is:

```
foreach epoch do {
   foreach pattern do {
      network_output[pattern] = forward_pass(pattern)}
   target_rank = sort_and_scale_patterns(target_value, network_output)
   foreach pattern do {
      backprop(target_rank[pattern] - network_output[pattern])}}
```

where "sort_and_scale_patterns" sorts and ranks the training patterns using the sort keys specified in its arguments, the second being used to break ties in the first.

One might worry that the net could learn to order the 0's (or the 1's) backwards, ranking lower risk 0's above the higher risk 0's. In theory this is possible. What prevents this in practice is that the net must learn to rank 0's below 1's because patterns with 0 targets are

[4] Actually, our implementation collects the net's outputs at the same time it is backpropagating errors computed using the sorted outputs collected during the previous epoch. This saves time because the collection forward pass is eliminated on all except the first pass, but means that the error signals fed back are always one epoch out of date.

always ranked less than patterns with 1 targets and this biases the net toward models that give lower rank to cases with less risk.

Table B.2 shows the mean rankprop performance using nets with 8 hidden units. The bottom row shows improvements over SSE on 0/1 targets. All differences are statistically significant at .05 or better.

Table B.2: Error Rates of Rankprop and Improvement Over Standard Backprop

FOP	0.1	0.2	0.3	0.4	0.5
Error Rate	.0083	.0144	.0210	.0289	.0386
% Change	-40.7% *	-24.2% *	-16.7% *	-15.0% *	-8.3% *

B.4 Soft Ranks

Here's how soft ranks work. Suppose we have five items, each with an associated real value that will be used to order the data. Sorting and ranking this data the usual way yields:

```
A: 0.25            E: 0.08  ->  1
B: 0.13            B: 0.13  ->  2
C: 0.54     =>     A: 0.25  ->  3
D: 0.27            D: 0.27  ->  4
E: 0.08            C: 0.54  ->  5
```

Because ranks are discrete, small changes to the predicted value of a case will not usually affect the rankings of the cases. For example, if the predicted value of B changes from 0.13 to 0.14, it still gets ranked second. It is difficult to apply gradient descent to error metrics defined on discrete ranks because of these plateaus in the error metric.

A small modification to ranks eliminates this problem while preserving the semantics. Order the data as usual and temporarily assign to each item the traditional rank. Then, post-process the traditional ranks as follows:

$$SoftRank(i) = TradRank(Prev(i)) + 0.5 + \frac{Val(i) - Val(Prev(i))}{Val(Post(i) - Val(Prev(i))}$$

/noindent where $TradRank(i)$ is the traditional rank of item i, $SoftRank(i)$ is the continuous rank of item i, $Val(i)$ is the value of item i, and $Prev(i)$ and $Post(i)$ denote the items that rank just before and just after item i, respectively.

The soft ranks computed this way for the five items above are:

```
E: 0.08  ->  1  ->  1
B: 0.13  ->  2  ->  2.088
A: 0.25  ->  3  ->  3.300
D: 0.27  ->  4  ->  3.569
C: 0.54  ->  5  ->  5
```

Consider item A. Its value, 0.25, is closer to 0.27 (the value of D, the item ranked just after it) than to 0.13 (the value of B, the item ranked just before it). The soft rank reflects this by assigning to item A a soft rank closer to the soft rank of D than to the soft rank of B.

Notice that the soft rank of the first and last items is the same as their traditional rank. For items in the interior, the SoftRank is always within ± 0.5 of its TradRank. Also, the TradRank and SoftRank have the same range.[5]

Qualitatively, soft ranks behave like traditional ranks, but have the nice additional property that they are continuous: small changes to item values yield small changes in the soft ranks. Moreover, if small changes in the values cause items to swap positions with neighboring items, the soft ranks reflect this in a smooth way. For example, if we increase the value of item A in the example above from 0.27 to 0.29, the new soft ranks are:

```
E: 0.08  ->  1  ->  1
B: 0.13  ->  2  ->  2.088
D: 0.27  ->  3  ->  3.433
A: 0.28  ->  4  ->  3.537
C: 0.54  ->  5  ->  5
```

[5] With a little more effort, it is possible to define a SoftRank similar to the above where the range of the TradRank and SoftRank is not the same, but their means are the same. This SoftRank is more useful in some circumstances, but the extra complexity is not necessary here so we use the simpler definition.

The traditional ranks of items A and D change abruptly as A passes D, but the soft ranks do not. It is possible to use soft ranks in most error metrics that use traditional ranks. Most error metrics defined on soft ranks will have behavior similar to their behavior when defined on traditional ranks, except that smoothness of soft ranks will not make the error metric discontinuous as the traditional ranks would. This means that we can apply gradient descent to error metrics based on soft ranks.

The main prediction task in the pneumonia domain is mortality risk. KNN and LCWA are used to predict the risk of each new case by examining whether its neighbors in the training set lived or died. Predicted cases are then sorted by this predicted risk. The optimization error metric we use pneumonia risk is the sum of the soft ranks for all patients in the sample *who live*. The goal is to order patients by risk, least risk first. Successfully ordering all patients that live before all patients that die minimizes this sum. We scale the sum of soft ranks so that 0 indicates all patients who live have been ranked with lower risk than all patients who die. This is ideal performance. The scaling is done so that a soft rank sum of 1 indicates that all patients who die have been ranked with lower risk than all patients who live. This is maximally poor performance. Random ordering of the patients yields soft rank sums around 0.5. Good performance on this domain requires soft rank sums less than 0.05.

B.5 Discussion

Why do sort-based methods like rankprop and soft rank sums work better than learning to rank patients by first learning to estimate patients' probabilities of death? Consider traditional SSE on 0/1 targets on Medis. It attempts to drive every person who lived to a value of 0, and every person who dies to a value of 1, regardless of their true, but unknown, probability of death. Now compare this with a rank-based metric on the same database. Let's assume 90and 10does not have to drive all patients that live to one fixed value. Instead, it has to find some ordering of the patients that live. This means it is possible that the patients who live and have low probability of death will be sorted to the left of patients who live but have high(er) probability of death. Ditto for patients who die. If such orderings can be found, the function that is to be learned should be less nonlinear than the function that would have to be learned by SSE on 0/1/ targets. The reasoning behind this is similar to that for quantized data: there exists a ranking of the database (which only contains 0's and 1's) such that the ranking is more similar to the original underlying probability function than the function SSE on 0/1 targets tries to learn.

Consider the function in Figure B.2a. Backprop should have no difficulty learning it—it's the sigmoid. Now imagine that this function has been quantized by some process to five discrete levels as in Figure B.2b. We no longer know the function in Figure B.2a, we only get training data from Figure B.2b. Given a large training set, backprop can learn this new function well. As the number of training patterns is reduced, however, it will begin to have difficulty. Suppose our true objective is to sort patterns according to the original underlying function in Figure B.2a, but we are only given a small number of samples from Figure B.2b. Must we learn the function $f(x)$ in Figure B.2b, or might it be beneficial to learn something else instead? All we really need to do is learn some function $g(x)$ such that $[g(x_1) \leq g(x_2)] \rightarrow [f(x_1) \leq f(x_2)]$. There can be many such functions $g(x)$ for a given $f(x)$, and some of these may be easier to learn well given a small training set.

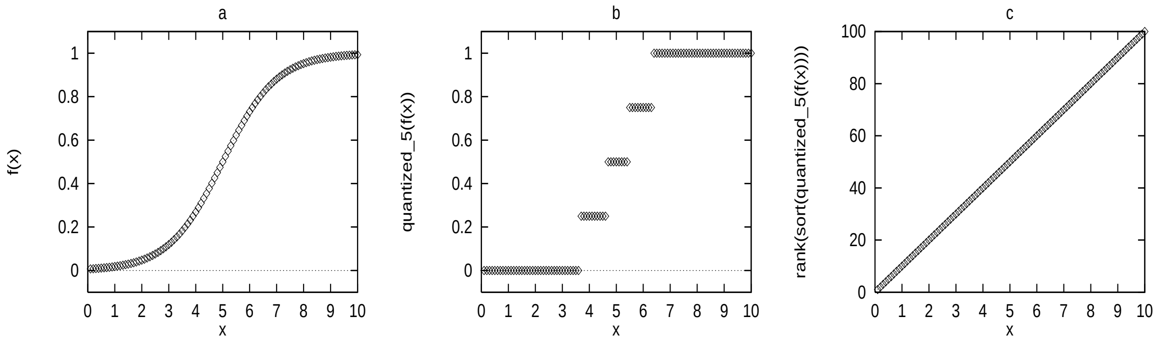


Figure B.2: Effects of Quantizing and then Ranking a Friendly Function

Figure B.2c shows one *possible* ranking of the data of Figure B.2b. Note that because ranks are evenly spaced, Figure B.2c is less nonlinear than Figure B.2b. If a net can learn a *ranking* similar to Figure B.2c for a *training sample* drawn from Figure B.2b, while at the same time learning a *ranking function* that predicts outputs for patterns corresponding to the rankings, it will sort future patterns well. On the other hand, if the net learns Figure B.2b directly from a small sample, it will have difficulty finding a function nonlinear enough to fit Figure B.2b, but not so nonlinear as to become flat or nonmonotonic, either of which would hurt sort performance. The more coarsely quantized the data, the larger the potential difference between rankprop and SSE on the quantized targets. The pneumonia database is a worst case scenario: the quantization is maximally coarse and has been corrupted by binomial noise. A net using SSE on *corrupted* 0/1 targets must not only learn the nonlinear quantization function, but also attempts to learn a function nonlinear enough to drive to 0 and 1 similar cases that by chance had different outcomes.

To make this more precise, we are given data from a target function $f(x)$. Suppose the goal is not to learn a model *of* $f(x)$, but to learn to sort patterns *by* $f(x)$. Must we learn a model of $f(x)$ and use its predictions for sorting? No. It suffices to learn a function $g(x)$ such that for all x_1, x_2, $[g(x_1) \leq g(x_2)] \rightarrow [f(x_1) \leq f(x_2)]$. There can be many such functions $g(x)$ for a given $f(x)$, and some of these may be easier to learn than $f(x)$.

Consider the probability function in Figure B.3a that assigns to each x the probability $p = f(x)$ that the outcome is 1; with probability $1 - p$ the outcome is 0. Figure B.3b shows a training set sampled from this distribution. Where the probability is low, there are many 0's. Where the probability is high, there are many 1's. Where the probability is near 0.5, there are 0's and 1's. *This region causes problems for backprop using SSE on 0/1 targets: similar inputs are mapped to dissimilar targets.*

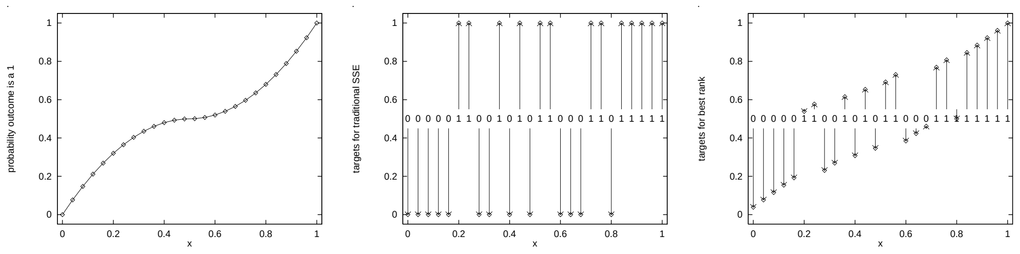


Figure B.3: SSE on 0/1 Targets and on Ranks for a Simple Probability Function

Learning sees a very nonlinear function when trained on Figure B.3b. This is unfortunate: Figure B.3a is smooth and maps similar inputs to similar outputs. If the goal is to learn to rank the data, we can learn a simpler, *less nonlinear* function instead. There exists a ranking of the training data such that if the ranks are used as target values, the resulting function is less nonlinear than the original target function. Figure B.3c shows these target rank values. Similar input patterns have more similar rank target values than the original target values.

Rank-based methods like rankprop try to learn *simple* functions that directly support ranking. One difficulty with this is that rankprop must learn a ranking of the training data while also training the model to predict ranks. We do not yet know under what conditions this parallel search will converge. We conjecture that when rank-based methods do converge, it will often be to simpler models than it would have learned from the original target values (0/1 in Medis), and that these simpler models will often generalize better.

Rankprop should work best when the following assumptions are satisfied: 1) the original underlying function, $f(x)$, is less nonlinear than the discretized function, $quantized(f(x))$, for which we have training data; 2) there exists a function $rank$ such that $rank(quantized(f(x)))$ is less nonlinear than $quantized(f(x))$; 3) rankprop can learn this $rank(quantized(f(x)))$; 4) given the ranks imposed on the training set, backprop with SSE can learn to predict these ranks from the inputs; and 5) backprop generalizes better from limited data when learning less nonlinear functions.

B.5.1 Other Applications of Rank-Based Methods

Rank-Based Methods are applicable wherever a relative assessment is more useful or more learnable than an absolute one. One application is domains where quantitative measurements are not available, but relative ones are [Hsu 1991]. For example, a game player might not be able to evaluate moves quantitatively , but might excel at relative move evaluation [Utgoff & Saxena 1987]. Another application is where the goal is to learn to order data drawn from a probability distribution, as in medical risk prediction. But it can also be applied wherever the goal is to order data. For example, in information filtering it is usually important to present more useful information to the user first, not to predict how important each is [Lang 1995].

B.6 Summary

This appendix presents two rank-based methods that can improve generalization on a broad class of problems. The first method, rankprop, tries to learn simple models that support ranking future cases while simultaneously learning to rank the training set. The second method, the soft rank sum, is an error criterion that results from generalizing ranks so that they are differentiable. Experiments using a database of pneumonia patients indicate that rankprop outperforms standard backpropagation by 10-40%.