8
A Dozen Tricks with Multitask Learning*

Rich Caruana

Just Research and Carnegie Mellon University,
4616 Henry Street, Pittsburgh, PA 15213
caruana@cs.cmu.edu
http://www.cs.cmu.edu/~caruana/

Abstract. Multitask Learning is an inductive transfer method that improves generalization accuracy on a main task by using the information contained in the training signals of other *related* tasks. It does this by learning the extra tasks in parallel with the main task while using a shared representation; what is learned for each task can help other tasks be learned better. This chapter describes a dozen opportunities for applying multitask learning in real problems. At the end of the chapter we also make several suggestions for how to get the most our of multitask learning on real-world problems.

When tackling real problems, one often encounters valuable information that is not easily incorporated in the learning process. This chapter shows a dozen ways to benefit from the information that often gets ignored. The basic trick is to create extra tasks that get trained on the same net with the main task. This *Multitask Learning* is a form of inductive transfer[1] that improves performance on the main task by using the information contained in the training signals of other *related* tasks. It does this by learning the main task in parallel with the extra tasks while using a shared representation; what is learned for each task can help other tasks be learned better.

We use the term "task" to refer to a function that will be learned from a training set. We call the important task that we wish to learn better the **main task**. Other tasks whose training signals will be used by multitask learning to learn the main task better are the **extra tasks**. Often, 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 the extra tasks a **domain**. Here we restrict ourselves to domains where the tasks are defined on a common set of input features, though some of the extra tasks may be functions of only a subset of these input features.

This chapter shows that most real-world domains present a number of opportunities for multitask learning (MTL). Because these opportunities are not

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

[1] Inductive transfer is the process of transfering anything learned for one problem to help learning of other related problems.

always obvious, most of the chapter is dedicated to showing different ways useful extra tasks arise in real problems. We demonstrate several of these opportunities using real data. The chapter ends with a few suggestions that help you get the most out of multitask learning. Some of these suggestions are so important that if you don't follow them, MTL can easily hurt performance on the main task instead of helping it.

8.1 Introduction to Multitask Learning in Backprop Nets

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

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

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 four 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 inputs, bits B_7 and B_8;
– each uses a common computed subfeature, $Parity(B_2 \cdots B_6)$;
– when $B_1 = 0$, Task 1 needs $Parity(B_2 \cdots B_6)$, but Task 2 does not, and vice versa;
– as with Tasks 1 and 2, when Task 3 needs $Parity(B_2 \cdots B_6)$, Task 4 does not need it, and vice versa.

We can train artificial neural nets on these tasks with backprop. Bits $B_1 \cdots B_8$ are the inputs to the net. The task values computed by the four functions are the target outputs. We create a data set 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.

8.1.1 Single and Multitask Learning of Task 1

Consider Task 1 the main task. Tasks 2, 3, and 4 are the extra tasks. That is, we are interested only in improving the accuracy of models trained for Task 1. We've done an experiment where we train Task 1 on the three nets shown in Figure 8.1. All the nets are fully connected feed-forward nets with 8 inputs, 100 hidden units, and 1–4 outputs. Where there are multiple outputs, each output is fully connected to the hidden units. Nets were trained in batch mode using backprop with MITRE's Aspirin/MIGRAINES 6.0 with learning rate $= 0.1$ and momentum $= 0.9$.

Task 1 is trained alone on the net on the left of Figure 8.1. This is a backprop net trained on a single task. We refer to this as single task learning (STL) or single task backprop (STL-backprop). The net in the center of Figure 8.1 trains Task 1 on a net that is also trained on Task 2. The hidden layer of this net is shared by Tasks 1 and 2. This is multitask backprop (MTL-backprop) with two tasks. The net on the right side of Figure 8.1 trains Task 1 with Tasks 2, 3, and 4. The hidden layer of this net is shared by all four tasks. This is MTL-backprop with four tasks. How well will Task 1 be learned by the different nets?

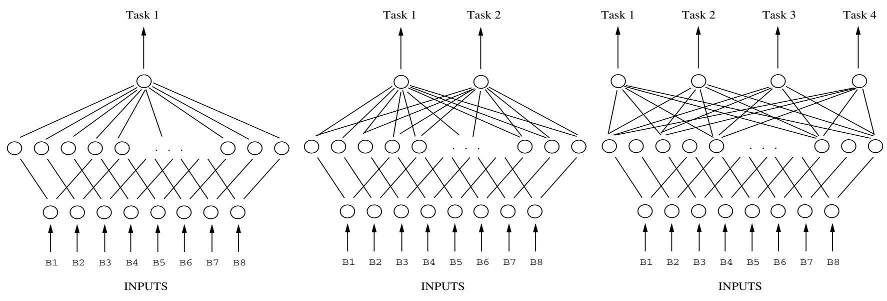

Fig. 8.1. Three Neural Net Architectures for Learning Task 1

We performed 25 independent trials by resampling training and test sets from the 256 cases. From the 256 cases, we randomly sample 128 cases for the training set, and use the remaining 128 cases as a test set. (For now we ignore the complexity of early stopping, which can be tricky with MTL nets. See section 8.3.2 for a thorough discussion of early stopping in MTL nets.)

For each trial, we trained three nets: an STL net for Task 1, an MTL net for Tasks 1 and 2, and an MTL net for Tasks 1–4. 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 backprop, 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.

8.1.2 Results

Every 5000 epochs we evaluated the performance of the nets on the test set. We measured the RMS error of the output with respect to the target values, the criterion being optimized by backprop. We also measured the accuracy of the output in predicting the boolean function values. If the net output is less than 0.5, it was treated as a prediction of 0, otherwise it was treated as a prediction of 1.

Figure 8.2 shows the RMSE for Task 1 on the test set during training. The three curves in the graph are each the average of 25 trials.[2] RMSE on the main task, 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 extra Task 2, and is further reduced when extra Tasks 3 and 4 are added. *Training multiple tasks on one net does not increase the number of training patterns seen by the net. Each net sees exactly the same training cases. The MTL nets do not see more training cases; they receive more training signals with each case.*

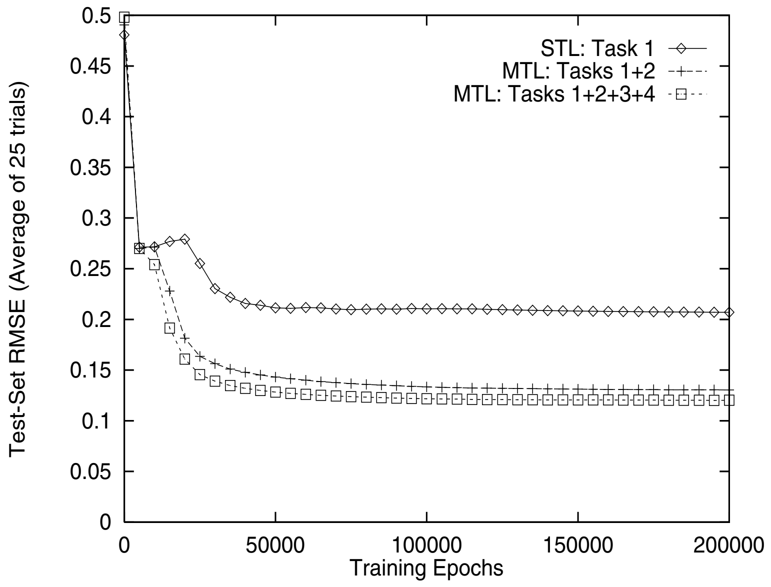

Fig. 8.2. RMSE Test-set Performance of Three Different Nets on Task 1

Figure 8.3 shows the average test-set accuracy on Task 1 for 25 trials with the three different nets. Task 1 has boolean value 1 about 75% of the time. A simple learner that learned to predict 1 all the time should achieve about 75% accuracy on Task 1. 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%. Table 8.1 summarizes the results of examining the training curve from each trial.

[2] 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.

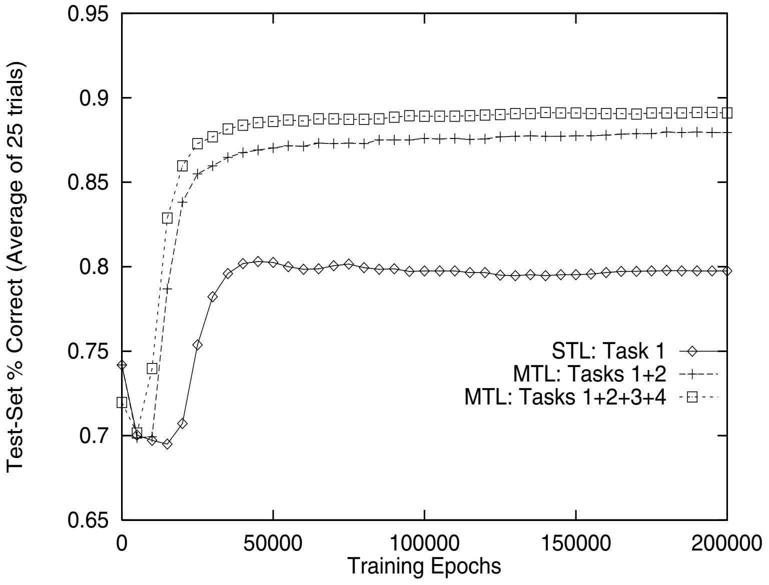

Fig. 8.3. Test-set Percent Correct of Three Different Nets on Task 1

Table 8.1. 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. *** indicates performance is statistically better than STL at .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% ***

8.1.3 Discussion

Why is the main task learned better if it is trained on a net learning other related tasks at the same time? 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 [22]. 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 explains the benefits we see from MTL, in one experiment we train Task 1 on a net with three *random* tasks.

A second effect to be concerned about is that adding tasks 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, and this might favor nets with multiple outputs. To test this, we train an MTL net with four copies of Task 1. Each of the four outputs receives exactly the same training signal. This is a degenerate form of MTL where no extra information is given to the net by the extra tasks.

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 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 ran a fourth experiment based on the heuristic used in [37]. We shuffle the training signals (the target output values) for Tasks 2, 3, and 4 before training an MTL net on the four tasks. Shuffling reassigns the target values to the input vectors in the training set for Tasks 2, 3, and 4. The main task, Task 1, is not affected. The distributions of the training signals for outputs 2–4 have not changed, but the training signals are no longer related to Task 1. This is a powerful test that has the potential to rule-out many mechanisms that do not depend on relationships between the tasks.

We ran each experiment 25 times using exactly the same data sets used in the previous section. Figure 8.4 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.

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 by adding noise to the learning process through the extra outputs.

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.[3] We conclude that MTL does not learn Task 1 better just because backprop works better with multiple outputs.

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.

[3] We sometimes observe that training multiple copies of a task on one net does improve performance. When we have observed this, the benefit is never large enough to explain away the benefits observed with MTL. But it is interesting and surprising, 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 weak boosting mechanism.

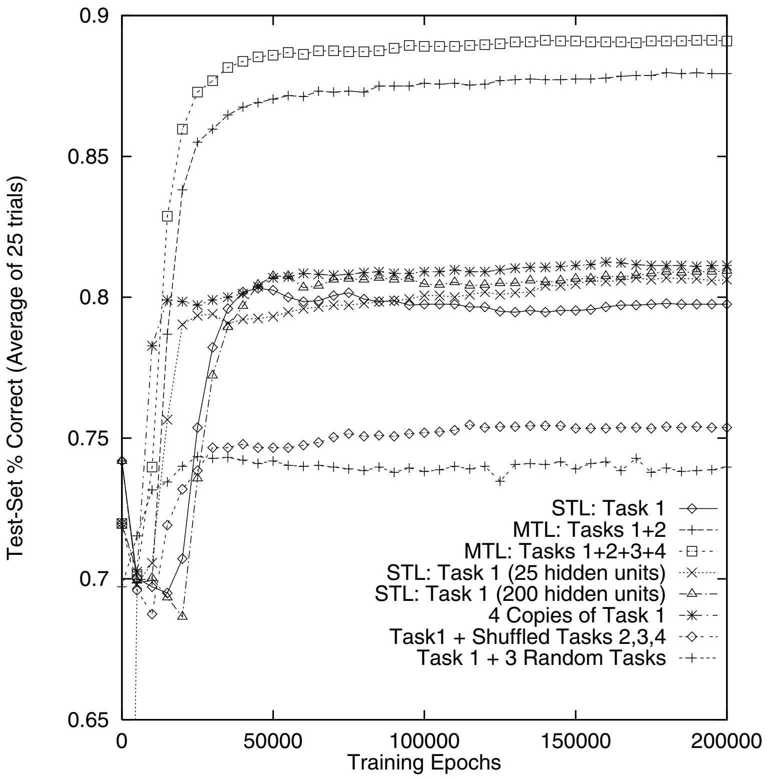

Fig. 8.4. RMSE test-set performance of Task 1 when trained with: MTL with three random tasks; MTL with three more copies of Task 1; MTL with shuffled training signals for Tasks 2–4; STL on nets with 25 or 200 hidden units.

When Task 1 is trained 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 distribution of the extra outputs. The extra outputs must be *related* to the main task to help it.

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

One reason is that Task 1 needs to learn the 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 $Parity(B_2 \cdots B_6)$. We say B_1 *masks* $Parity(B_2 \cdots B_6)$ 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 masked. 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. Because the tasks overlap on the features they use and don't use, the MTL net is better able select which input features to use.

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, in this case more correct, internal representations that support the multiple tasks. This bias towards modular internal representations reduces 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 the other tasks, it is less likely to be learned.

8.2 Tricks for Using Multitask Learning in the Real World

The previous section introduced multitask learning (MTL) in backprop nets using four tasks carefully designed to have relationships that make learning them in parallel work better than learning them in isolation. How often will real problems present extra tasks that allow multitask learning to improve performance on the main task?

This section shows that many real world problems yield opportunities for multitask learning. We present a dozen prototypical real-world applications where the training signals for related extra tasks are available and can be leveraged. We believe most real-world problems fall into one or more of these prototypical classes. 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 used in machine learning so far have been heavily preprocessed to fit the single task learning mold. Most of the opportunities for MTL in these problems were eliminated as the problems were defined.

8.2.1 Using the Future to Predict the Present

Often valuable features become available *after* predictions must be made. If learning is done offline, these features can be collected for the training set and used for learning. These features can't be used as inputs, because they will not be available when making predictions for future test cases. They can, however, be used as extra outputs for multitask learning. The predictions the learner makes for these extra tasks will be ignored when the system is used to make predictions for the main task. Their sole function is to provide extra information to the learner during training so that it can learn the main task better.

One source of applications of learning from the future is sequential decision making in medicine. Given the initial symptoms, decisions are made about what tests to make and what treatment to begin. New information becomes available when the tests are completed and as the patient responds (or fails to respond) to the treatment. From this new information, new decisions are made. Should more tests be made? Should the treatment be changed? Has the patient's condition changed? Is this patient now high risk, or low risk? Does the patient need to be hospitalized? Etc.

When machine learning is applied to early stages in the decision making process, only those input features that typically would be available for patients at this stage of the process are usually used. This is unfortunate. In an historical database, all of the patients may have run the full course of medical testing and treatment and their final outcome may be known. Must we ignore the results of lab tests and other valuable features in the database just because these will not be available for patients at the stage of medical decision making for which we wish to learn a model?

The Pneumonia Risk Prediction Problem. Consider pneumonia. There are 3,000,000 cases of pneumonia each year in the U.S., 900,000 of which get hospitalized. 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.

Consider the problem of predicting a patient's risk from pneumonia before they are hospitalized. (The problem is not to diagnose if the patient has pneumonia, but to determine how much risk the pneumonia poses to the patient.) A primary goal in medical decision making is to accurately, swiftly, and economically identify patients at high risk from diseases like pneumonia so that they may be hospitalized to receive aggressive testing and treatment; patients at low risk may be more comfortably, safely, and economically treated at home.

Some of the most useful features for assessing risk are the lab tests that become available only after a patient is hospitalized. It is the *extra* lab tests made after patients are admitted to the hospital that we use as extra tasks for MTL; they cannot be used as inputs because they will not be available for most future patients when making the decision to hospitalize.[4]

The most useful decision aid for this problem would be to predict which patients will live or die. 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 order 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.

[4] Other researchers who tackled this problem ignored the the lab tests because they knew they would not be available at run time and did not see ways to use them other than as inputs.

The performance criteria used by others working with this database [15] is the accuracy with which one can select prespecified fractions of a 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 risk 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"). Here we consider FOPs 0.1, 0.2, 0.3, 0.4, and 0.5. Our goal is to learn models and thresholds such that the error rate at each FOP is minimized.

Multitask Learning and Pneumonia Risk Prediction. 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 30 basic pre-hospitalization measurements, a single hidden layer, and a single output trained with targets 0=lived, 1=died.[5] Given a large training set, a net trained this way should learn to predict the probability of death for each patient, not which patients live or die. If the training sample is small, the net will overfit and 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. Rankprop learns to rank patients without learning to predict mortality (0=lived,1=died). Figure 8.5 compares the performance of squared error on 0/1 targets with rankprop on this problem. Rankprop outperforms traditional backprop using squared error on targets 0=lived,1=died by 10%-40% on this domain, depending on which FOP is used for comparison. See [9] for details about rankprop.[6]

There are 35 future lab values that we use as extra backprop *outputs,* as shown in Figure 8.6. The expectation is that these extra outputs will bias the shared hidden layer toward representations that better capture important features of each patient's condition, and that this will lead to more accurate predictions of patient risk at the main task output.

The STL net has 30 inputs, 8 hidden units, and one output trained to predict risk with rankprop. The MTL net has the same 30 inputs, 64 hidden units, one output for rankprop, and 35 extra outputs trained with squared error. (Preliminary experiments suggested 8–32 hidden units was optimal for STL, and that MTL performs best with nets as large as 512 hidden units. We used 8 and 64 hidden units so that we could run many experiments.) The 35 extra outputs on

[5] We tried both squared error and cross entropy. The difference between the two was small. Squared error performed slightly better.

[6] We use rankprop for the rest of our experiments on this domain because it is the best performer we know of on this problem. We want to see if MTL can make the best method better.

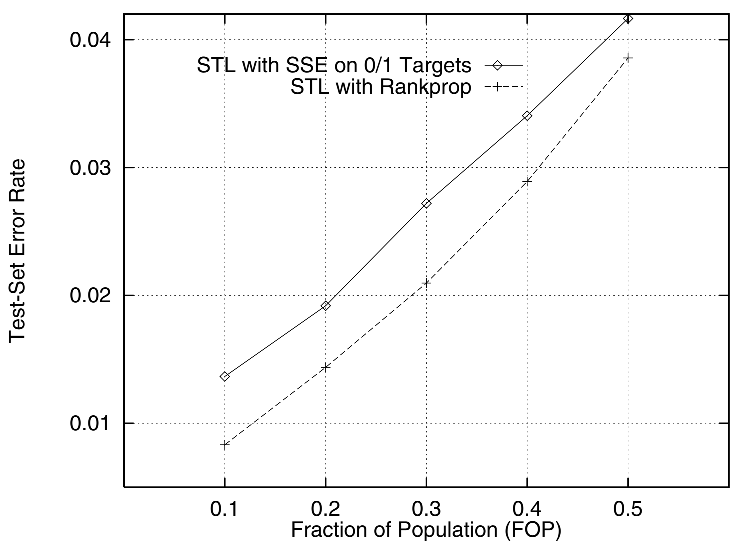

Fig. 8.5. The performance of SSE (0/1 targets) and rankprop on the 5 FOPs in the pneumonia domain. Lower error indicates better performance.

the MTL net (see Figure 8.6) are trained at the same time the net is trained to predict risk.

We train the net using training and validation sets containing 1000 patients randomly drawn from the database. Training is halted on both the STL and MTL nets when overfitting is observed on the main rankprop risk task. On the MTL net, the performance of the extra tasks is not taken into account for early stopping. Only the performance of the output for the main task is considered when deciding when to stop training. (See section 8.3.2 for more discussion of early stopping with MTL nets.) Once training is halted, the net is tested on the remaining unused patients in the database.

Results. Table 8.2 shows the mean performance of ten runs of rankprop using STL and MTL. The bottom row shows the percent improvement in performance obtained on this problem by using the future lab measurements as extra MTL outputs. 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.

The improvement from MTL is 5–10%. This improvement can be of considerable consequence in medical domains. To verify that the benefit from MTL is due to relationships between what is learned for the future labs and the main task, we ran the shuffle test (see section 8.1.3). We shuffled the training signals for the extra tasks in the training sets before training the nets with MTL.

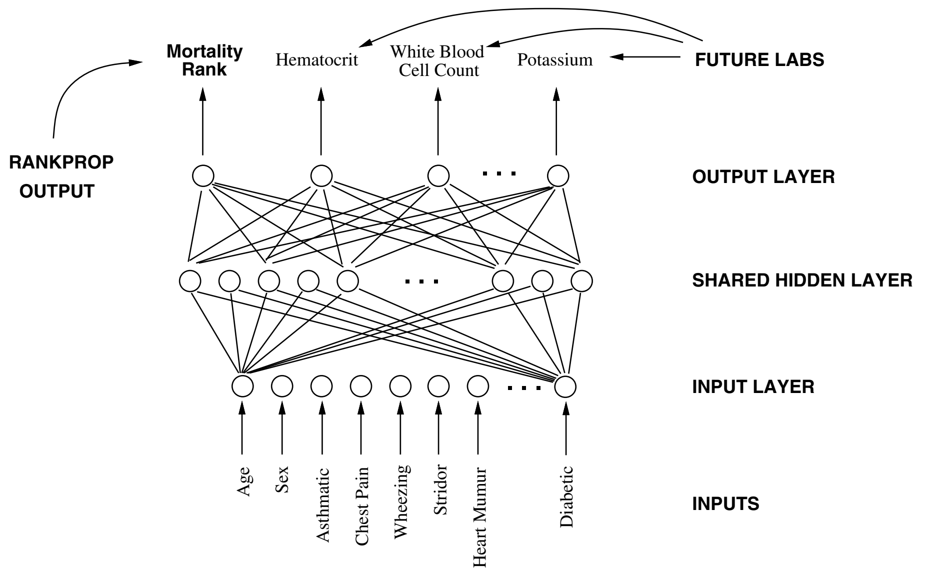

Fig. 8.6. Using future lab results as extra outputs to bias learning for the main risk prediction task. The lab tests would help most if they could be used as inputs, but will not yet have been measured when risk must be predicted, so we use them as extra outputs for MTL instead.

Table 8.2. 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% *

Figure 8.7 shows the results of MTL with shuffled training signals for the extra tasks. The results of STL and 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 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.

We have also run experiments where we use the future lab tests as inputs to a net trained to predict risk, and impute the values for the lab tests when they are missing on future test cases. Imputing missing values for the lab tests did not yield performance comparable to MTL on this problem. Similar experiments with feature nets [17] also failed to yield improvements comparable to MTL.

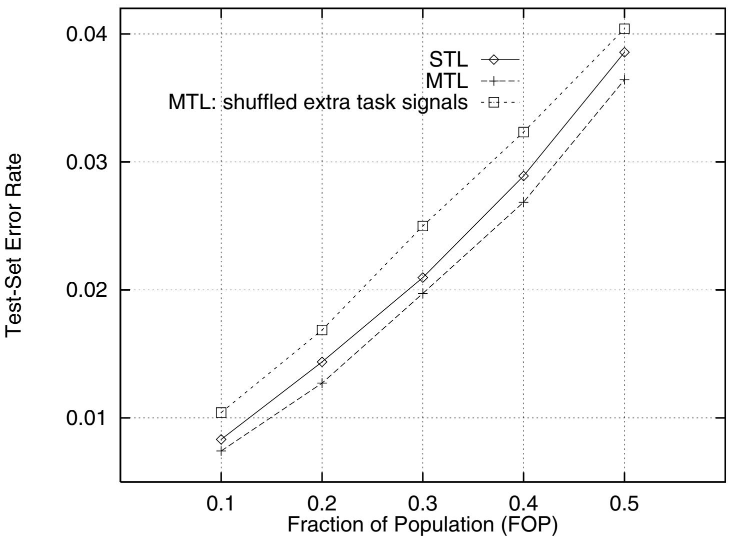

Fig. 8.7. Performance of STL, MTL, and MTL with shuffled extra task signals on pneumonia risk prediction at the five FOPs

Future measurements are available in many *offline* learning problems. As just one very different example, a robot or autonomous vehicle can more accurately measure the size, location, and identity of objects if it passes nearer them in the future. For example, road stripes and the edge of the road can be detected reliably as a vehicle passes alongside them, but detecting them far ahead of the vehicle is hard. Since driving brings future road closer to the car, stripes and road borders can be measured accurately as the car passes them. Dead reckoning allows these future measurements to be added to the training set. They can't be used as *inputs*; They won't be available in time while driving. But they can be used to augment a training set. We suspect that using future measurements as extra outputs will be a frequent source of extra tasks in real problems.

8.2.2 Multiple Metrics

Sometimes it is hard to capture everything that is important in one error metric. When alternate metrics capture different, but useful, aspects of a problem, MTL can be used to benefit from the multiple metrics. One example of this is the pneumonia problem in the previous section. Rankprop outperforms backprop using traditional squared error on this problem, but has trouble learning to rank cases at such low risk that virtually all patients survive because these cases provide little ordering information. Interestingly, squared error performs best when cases have high purity, such as in regions of feature space where most cases have low risk. *Squared error is at its best where rankprop is weakest.* Adding an extra output trained with squared error to a net learning to predict pneumonia risk with rankprop improves the accuracy of the rankprop output an additional 5-10% for the least-risk cases. The earliest example of using multiple output representations we know of is [38] which uses both SSE and cross-entropy outputs for the same task.

8.2.3 Multiple Output Representations

Sometimes it is not apparent what output encoding is best for a problem. Distributed output representations often help *parts* of a problem be learned well because the parts have separate error gradients. But non-distributed output representations are sometimes more accurate. Consider the problem of learning to classify a face as one of twenty faces. One output representation is to have one output code for each face. Another representation is to have outputs code for features such as beard/no_beard, glasses/no_glasses, long_hair/short, eye_color(blue, brown), male/female, that are sufficient to distinguish the faces. Correct classification, however, may require that each feature be correctly predicted. The non-distributed output coding that uses one output for each individual may be more reliable. But training the net to recognize specific traits should help, too. MTL is one way to merge these conflicting requirements in one net by using both output representations, even if only one representation will be used for prediction.

A related approach to multiple output encodings is error correcting codes [18]. Here, multiple encodings for the outputs are designed so that the combined prediction is less sensitive to occasional errors in some of the outputs. It is not clear how much ECOC benefits from MTL-like mechanisms. In fact, ECOC may benefit from being trained on STL nets (instead of MTL nets) so that different outputs do not share the same hidden layer and thus are less correlated. But see [27] for ways of using MTL to *decorrelate* errors in multiple outputs to boost committee machine performance.

8.2.4 Time Series Prediction

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. This net makes predictions for the same task at different times. We tested this on a robot domain where the goal is to predict what the robot will sense 1, 2, 4, and 8 meters in the future as it moves forward. Training all four of these distances on one MTL net improved the accuracy of the long range predictions about 10% (see chapter 17 where MTL is used in a time series application).

8.2.5 Using Non-operational Features

Some features are impractical to use at run time, either because they are too expensive to compute, or because they need human expertise that won't be available. We usually have more time, however, to prepare our training sets.

When it is impractical to compute some features on the fly at run time, but practical to compute them for the training set, these features can be used as extra outputs to help learning. Pattern recognition provides a good example of this. We tested MTL on a door recognition problem where the goal is to recognize doorways and doorknobs. The extra tasks were features such as the location of door edges and doorway centers that required laborious hand labelling that would not be applied to the test set. The MTL nets that were trained to predict these additional hand-labelled features were 25% more accurate at locating doors and doorknobs. Other domains where hand-labelling can be used to augment training sets this way include text domains, medical domains, acoustic domains, and speech domains.

8.2.6 Using Extra Tasks to Focus Attention

Learning often uses 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 these patterns.

A good example 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). 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.

We tested this idea using a road image simulator developed by Pomerleau to permit rapid testing of learning methods for road-following domains [28]. Figure 8.8 shows several 2-D road images.

The principal task is to predict steering direction. For the MTL experiments, we used eight additional tasks:

- 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 (if any)
- location of right edge of road
- intensity of road surface
- intensity of centerline (if any)

These additional tasks are all computable from the internal variables in the simulator. Table 8.3 shows the average performance of ten runs of single and multitask learning on each of these tasks. 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.

The last two columns compare STL and MTL. The first is the percent reduction in error of MTL over the best STL run. Negative percentages indicate MTL

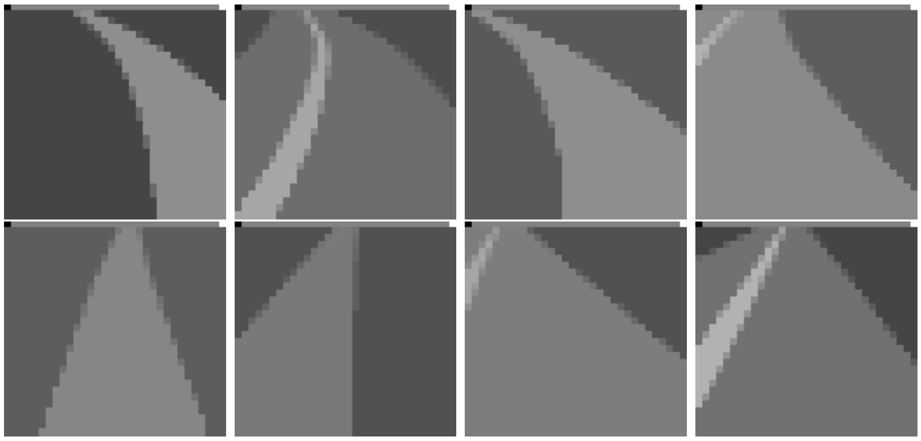

Fig. 8.8. Sample single and two lane roads generated with Pomerleau's road simulator

performs better. The last column is the percent improvement of MTL over the average STL performance. On the important steering task, MTL outperforms STL 15–30%.

We ran a follow-up experiment to test how important centerstripes are to the STL and MTL nets. We eliminated the stripes from the images in a test set. 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. Error increased more for the MTL nets than for the STL nets, suggesting the MTL nets are making more use of the stripes in the images.

Table 8.3. Performance of STL and MTL on the road following 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% *

8.2.7 *Hints:* Tasks Hand-Crafted by a Domain Expert

Extra outputs can be used to *inject rule hints* into nets about what they should learn [32, 33]. This is MTL where the extra tasks are carefully engineered to coerce the net to learn specific internal representations. Hints can also be provided to backprop nets via extra terms in the error signal backpropagated for the main task output [1, 2]. The extra error terms constrain what is learned to satisfy desired properties of main task such as monotonicity [31], symmetry, or transitivity with respect to certain sets of inputs. MTL, which does not use extra error terms on task outputs, could be used in concert with these techniques.

8.2.8 Handling *other* Categories in Classification

In real-world applications of digit recognition, some of the images given to the classifier may be alphabetic characters or punctuation marks instead of digits. One way to prevent accidentally classifying a "t" as a one or seven is to create an "other" category that is the correct classification for non-digit images. The large variety of characters mapped to this "other" class makes learning this class potentially very difficult. MTL suggests an alternate way to do this. Split the "other" class into separate classes for the individual characters that are trained in parallel with the main digit tasks. A single output coding for the "other" class can be used, as well. Breaking the "other" category into multiple tasks gives the net more learnable error signal for these cases [26].

8.2.9 Sequential Transfer

MTL is parallel transfer. Often tasks arise serially and we can't wait for all of them to begin learning. In these cases we can use parallel transfer to perform sequential transfer. If the training data can be stored, do MTL using whatever tasks are available when it is time to start learning, and re-train as new tasks or new data arise. If training data cannot be stored, or if we already have models for which data is not available, we can still use MTL. Use the models to generate synthetic data that is then used as extra training signals. This approach to sequential transfer avoids catastrophic interference (forgetting old tasks while learning new ones). Moreover, it is applicable where the analytical methods of evaluating domain theories required by some serial transfer methods [29, 34] are not available. For example, the domain theory need not be differentiable, it only needs to make predictions. One issue that arises when synthesizing data from prior models is what distribution to sample from. See [16] for a discussion of synthetic sampling.

8.2.10 Similar Tasks With Different Data Distributions

Sometimes there are multiple instances of the same problem, but the distribution of samples differs for each instantiation. For example, most hospitals diagnose and treat the same diseases, but the demographics of the patients each hospital serves is different. Hospitals in Florida see older patients, urban hospitals see poorer patients, etc. Models trained separately for each hospital would perform best, but often there is insufficient data to train a separate model for each hospital. Pooling the data, however, may not lead to models that are accurate for each hospital. MTL provides one solution to this problem. Use one net to make predictions for each hospital, using a different output on the net for each hospital. Because each patient is a training case for only one hospital, error can be backpropagated only through the one output that has a target value for each input vector.

8.2.11 Learning with Hierarchical Data

In many domains, the data falls in a hierarchy of classes. Most applications of machine learning to hierarchical data make little use of the hierarchy. MTL provides one way of exploiting hierarchical information. When training a model to classify data at one level in the hierarchy, include as extra tasks the classification tasks that arise for ancestors, descendants, and siblings of the current classification task. The easiest way to to accomplish this is to train one MTL net to predict all class distinctions in the hierarchy at the same time.

8.2.12 Some Inputs Work Better as Outputs

The common practice in backprop nets is to use all features that will be available for test cases as inputs, and have outputs only for tasks that need to be predicted. On real problems, however, learning often works better given a carefully selected subset of the features to use inputs[7, 23, 24]. One way to benefit from features not used as inputs is to use them as extra outputs for MTL. We've done experiments with both synthetic and real problems where moving some features from the input side of the net to the output side of the net improves performance on the main task. We use feature selection to select those features that should be used as inputs, and then treat some of the remaining features as extra tasks.

Figure 8.9 shows the ROC Area on a pneumonia problem as the number of input features on the backprop net varies.[7] ROC Areas closer to 1 indicate better performance. There are 192 features available for most patients. Using all 192 features as inputs (Net1) is suboptimal. Better performance is obtained by using the first 50 features selected with feature selection (Net2). The horizontal line at the top of the graph (Net3) shows the ROC Area obtained by using the first 50 features as inputs, and the *next* 100 features as extra *outputs*. Using these same 150 features all as inputs (Net4) yields worse performance.[8]

[7] This is not the same pneumonia problem used in section 8.2.1.

[8] Although the 95% confidence intervals for Net2 and Net3 overlap with ten trials, a paired t-test shows the results are significant at .01.

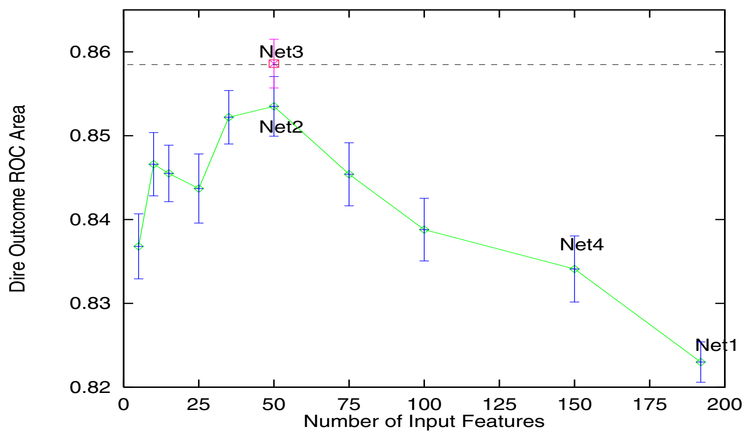

Fig. 8.9. ROC Area on the Pneumonia Risk Prediction Task vs. the number of input features used by the backprop net

8.3 Getting the Most Out of MTL

The basic machinery for doing multitask learning in neural nets is present in backprop. Backprop, however, was not designed to do MTL well. This chapter presents suggestions for how to make MTL in backprop nets work better. Some of the suggestions are counterintuitive, but if not used, can cause MTL to hurt generalization on the main task instead of helping it.

8.3.1 Use Large Hidden Layers

The basic idea behind MTL in backprop nets is that what is learned in the hidden layer for one task can be useful to other tasks. MTL works when tasks share hidden units. One might think that small hidden layers would help MTL by promoting sharing between tasks.

For the kinds of problems we've examined here, this usually does not work. Usually, tasks are different enough that much of what each task needs to learn does not transfer to many (or any) other tasks. Using a large hidden layer insures that there are enough hidden units for tasks to learn independent hidden layer representations when they need to. Sharing can still occur, but only when the overlap between the hidden layer representations for different tasks is strong. In many real-world problems, the loss in accuracy that results from forcing tasks to share by keeping the hidden layer small is larger than the benefit that arises from the sharing. Usually it is important to use large hidden layers with MTL.

8.3.2 Do Early Stopping for Each Task Separately

The classic NETtalk application [30] used one trained both phonemes and stresses on one backprop net. NETtalk is an early example of MTL. But the builders

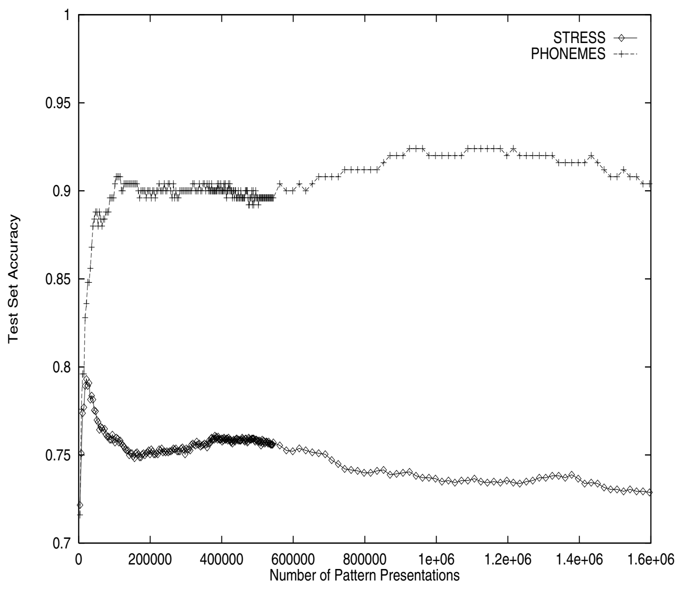

Fig. 8.10. On NETtalk, the Stress task trains very quickly and overfits long before the Phoneme task reaches peak performance

of NETtalk viewed the multiple outputs as codings for a single problem, not as independent tasks that benefited each other by being trained together.

Figure 8.10 shows the learning curves for the phoneme and stress subtasks separately. It is clear that the stress tasks begin to overfit before the phoneme tasks reach peak performance. Better performance could be obtained on NETtalk by doing early stopping on the stress and phoneme tasks individually, or by balancing their learning rates so they reach peak performance at the same time.

Early stopping prevents overfitting by halting the training of error-driven procedures like backprop before they achieve minimum error on the training set (see chapter 2). Recall the steering prediction problem from section 8.2.6. We applied MTL to this problem by training a net on eight extra tasks in addition to the main steering task. Figure 8.11 shows nine learning curves, one for each of the tasks on this MTL net. Each graph is the validation set error during training.

Table 8.4 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 tasks are important, and one net is used to predict all the tasks, halting training where the error summed across all outputs is minimized is the best you can do. Figure 8.12 shows the combined RMS error of the nine tasks. The best average RMSE occurs at 75,000 backprop passes.

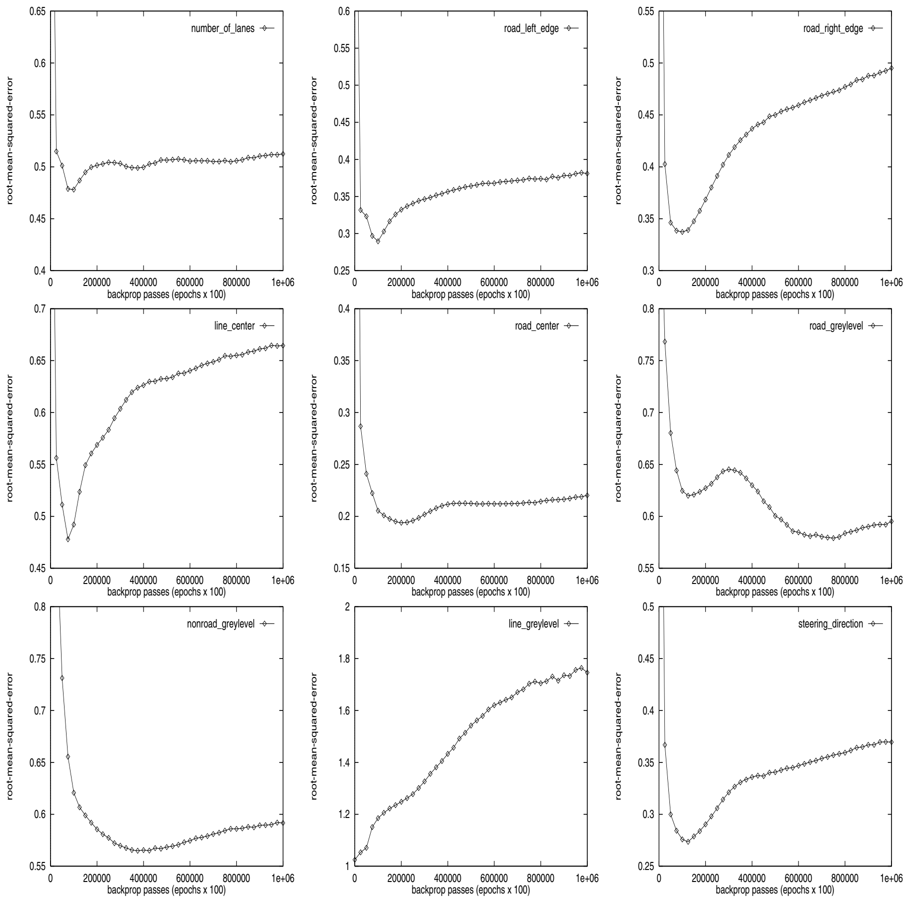

Fig. 8.11. Test-Set Performance of MTL Net Trained on Nine Tasks

But using one net to make predictions for all the tasks is suboptimal. Better performance is achieved by using the validation set to do early stopping on each output individually. The trick is to make a copy of the net at the epoch where performance on each task is best, and use this copy to make predictions for that task. After making each copy, continue training the net until the other tasks reach peak performance. Sometimes, it is best to continue training all outputs on the net, including those that have begun to overfit. Sometimes, it is better to stop training (or use a lower learning rate) for outputs that have begun to overfit. Keep in mind that once an output has begun to overfit, we no longer care how well the net performs on that task because we have a copy of the net from an earlier epoch when performance on that task was best. The only reason to continue training the task is because it may benefit other tasks that have not reached peak performance yet.

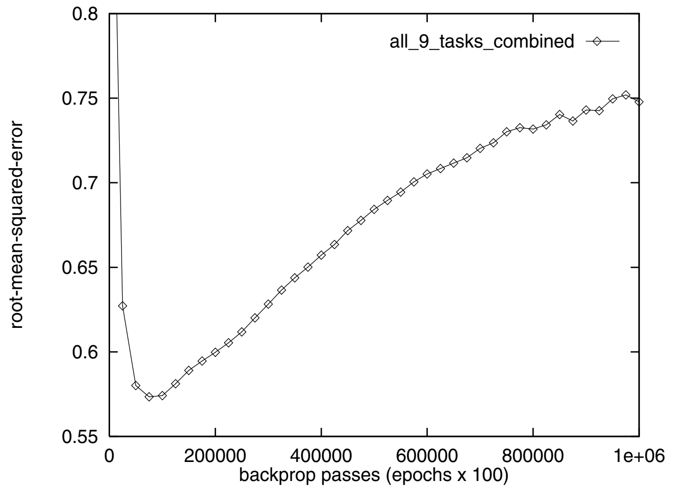

Fig. 8.12. Combined Test-Set Performance on all Nine Tasks

Table 8.4. Performance of MTL on the nine tasks in the steering domain when training is halted on each task individually compared with halting using the combined error across all tasks

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%

Table 8.4 compares the performance of early stopping done per task with the performance one obtains by halting training for the entire MTL net at one place using the combined error. On average, early stopping for 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.

Before leaving this topic, it should be mentioned that the training curves for the individual outputs on an MTL net are not necessarily monotonic. While it is not unheard of for the test-set error of an STL net to be multimodal, the training-set error for an STL net should descend monotonically or become flat. This is not true for the errors of individual outputs on an MTL net. The training-set error summed across all outputs should never increase, but any one output may exhibit more complex behavior. The graph for road_greylevel (graph number 6) in Figure 8.11 shows a multimodal test-set curve. The training set curve for this output is similar. This makes judging when to halt training more difficult with MTL nets. Because of this, we always do early stopping on MTL nets by training past the epoch where performance on each task appears to be best, and either retrain the net a second time (with the same random seed) to get the copies, or are careful to keep enough copies during the first run that we have whatever copies we will need.

8.3.3 Use Different Learning Rates for Different Tasks

Is it possible to control the rates at which different tasks train so they each reach their best performance at the same time? 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?

The rate at which different tasks learn using vanilla backprop is rarely optimal for MTL. Task that train slower than the main task will not have learned enough to help the main task when training on the main task is stopped. Tasks that train faster than the main task may overfit so much before the main task is learned well that either what they have learned is no longer useful to the main task, or they may drive the main task into premature overfitting.

The most direct method of controlling the rate at which different tasks learn is to use a different learning rate for each task, i.e., for each output. We have experimented with using gradient descent to find learning rates for each extra output to maximize the generalization performance on the main task. Table 8.5 shows the performance on the main steering task before and after optimizing the learning rates of the other eight extra tasks. Optimizing the learning rates for the extra MTL tasks improved 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.

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. Interestingly, it also has a significant effect on the rate at which the main task is learned.

Table 8.5. Performance of MTL on the main Steering Direction task before and after optimizing the learning rates of the other eight extra tasks

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% *

This is surprising because we keep the learning rate for the main task fixed during optimization. Perhaps even more interesting is the fact that optimizing the learning rate to maximize the generalization accuracy of the main task also improved generalization on the extra tasks nearly as much as it helped the main task. What is good for the goose appears to be good for the gander.

8.3.4 Use a Private Hidden Layer for the Main Task

Sometimes the optimal number of hidden units is 100 hidden units or more per output. If there are hundreds of extra tasks this translates to thousands of hidden units. This not only creates computational difficulties, but degrades performance on the main task because most of the hidden layer repersentation is constructed for other tasks. The main task output unit has a massive hidden unit selection problem as it tries to use only those few hidden units that are useful to it.

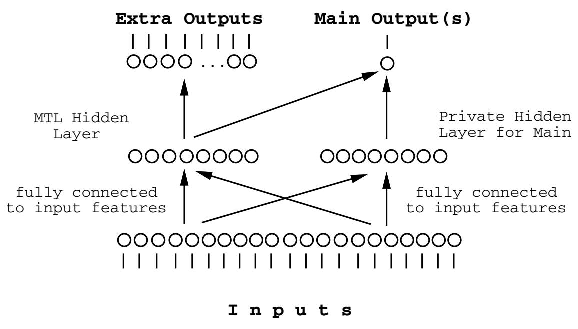
Extra Outputs
Main Output(s)
MTL Hidden Layer
Private Hidden Layer for Main
fully connected to input features
fully connected to input features
I n p u t s

Fig. 8.13. MTL Architecture With a Private Hidden Layer for the Main Task(s), and a Shared Hidden Layer Used by the Main Task(s) and the Extra Tasks

Figure 8.13 shows a net architecture that solves this problem. Instead of one hidden layer shared equally by all tasks, there are two disjoint hidden layers. Hidden layer 1 is a private hidden layer used only by the main task(s). Hidden layer 2 is shared by the main task(s) and the extra tasks. This is the hidden layer that supports MTL transfer. Because the main task sees and affects the shared hidden layer, but the extra tasks do not affect the hidden layer reserved for the main tasks(s), hidden layer 2 can be kept small without hurting the main task.

8.4 Chapter Summary

We usually think of the inputs to a backprop net as the place where information is given to the net, and the outputs as the place where the net outputs predictions. Backprop, however, pushes information into the net through the *outputs* during training. The information fed into a net through its outputs is as important as the information fed into it through its inputs.

Multitask Learning is a way of using the outputs of a backprop net to push additional information into the net during training. If the net architecture allows sharing of what is learned for different outputs, this additional information can help the main task be learned better. (See [5, 6, 4, 8, 3, 9, 11, 10, 20, 35, 36, 12, 21, 13, 14] for additional discussion about multitask learning.)

MTL trains multiple tasks in parallel not because this is a more efficient way to learn multiple tasks, but because the information in the training signals for the extra tasks can help the main task be learned better. Sometimes what is optimal for the main task is not optimal for the extra tasks. 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 the extra tasks are important too, it may be best to rerun learning for each important extra task, with the technique optimized for each task one at a time.

This chapter presented a number of opportunities for using extra outputs to leverage information that is available in real domains. The trick in most of these applications is to view the outputs of the net as inputs that are used only during learning. Any information that is available when the net is trained, but which would not be available later when the net is used for prediction, can potentially be used as extra outputs. There are many domains where useful extra tasks will be available. The list of prototypical domains provided in this chapter is not complete. More kinds of extra tasks will be identified in the future.

Acknowledgements. R. Caruana was supported by ARPA grant F33615-93-1-1330, NSF grant BES-9315428, and Agency for Health Care Policy grant HS06468. The work to find input features that worked better when used as extra outputs is joint work with Virginia de Sa, who was supported by postdoctoral fellowship from the Sloan Foundation. We thank the University of Toronto for the Xerion Simulator, and D. Koller and M. Sahami for the use of their feature selector.

References

[1] Abu-Mostafa, Y.S.: Learning from Hints in Neural Networks. Journal of Complexity 6(2), 192–198 (1990)
[2] Abu-Mostafa, Y.S.: Hints. Neural Computation 7, 639–671 (1995)
[3] Baxter, J.: Learning Internal Representations. In: COLT 1995, Santa Cruz, CA (1995)
[4] Baxter, J.: Learning Internal Representations. Ph.D. Thesis, The Flinders Univeristy of South Australia (December 1994)
[5] Caruana, R.: Multitask Learning: A Knowledge-Based Source of Inductive Bias. In: Proceedings of the 10th International Conference on Machine Learning, ML 1993, University of Massachusetts, Amherst, pp. 41–48 (1993)
[6] Caruana, R.: Multitask Connectionist Learning. In: Proceedings of the 1993 Connectionist Models Summer School, pp. 372–379 (1994)
[7] Caruana, R., Freitag, D.: Greedy Attribute Selection. In: ICML 1994, Rutgers, NJ, pp. 28–36 (1994)

[8] Caruana, R.: Learning Many Related Tasks at the Same Time with Backpropagation. In: NIPS 1994, pp. 656–664 (1995)
[9] Caruana, R., Baluja, S., Mitchell, T.: Using the Future to "Sort Out" the Present: Rankprop and Multitask Learning for Medical Risk Prediction. In: Proceedings of Advances in Neural Information Processing Systems, NIPS 1995, pp. 959–965 (1996)
[10] Caruana, R., de Sa, V.R.: Promoting Poor Features to Supervisors: Some Inputs Work Better As Outputs. In: NIPS 1996 (1997)
[11] Caruana, R.: Multitask Learning. Machine Learning 28, 41–75 (1997)
[12] Caruana, R.: Multitask Learning. Ph.D. thesis, Carnegie Mellon University, CMU-CS-97-203 (1997)
[13] Caruana, R., O'Sullivan, J.: Multitask Pattern Recognition for Autonomous Robots. In: The Proceedings of the IEEE Intelligent Robots and Systems Conference (IROS 1998), Victoria (1998) (to appear)
[14] Caruana, R., de Sa, V.R.: Using Feature Selection to Find Inputs that Work Better as Outputs. In: The Proceedings of the International Conference on Neural Nets (ICANN 1998), Sweden (1998) (to appear)
[15] 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., Spirtes, P.: An Evaluation of Machine Learning Methods for Predicting Pneumonia Mortality. Artificial Intelligence in Medicine 9, 107–138 (1997)
[16] Craven, M., Shavlik, J.: Using Sampling and Queries to Extract Rules from Trained Neural Networks. In: Proceedings of the 11th International Conference on Machine Learning, ML 1994, Rutgers University, New Jersey, pp. 37–45 (1994)
[17] Davis, I., Stentz, A.: Sensor Fusion for Autonomous Outdoor Navigation Using Neural Networks. In: Proceedings of IEEE's Intelligent Robots and Systems Conference (1995)
[18] Dietterich, T.G., Bakiri, G.: Solving Multiclass Learning Problems via Error-Correcting Output Codes. Journal of Artificial Intelligence Research 2, 263–286 (1995)
[19] Fine, M.J., Singer, D., Hanusa, B.H., Lave, J., Kapoor, W.: Validation of a Pneumonia Prognostic Index Using the MedisGroups Comparative Hospital Database. American Journal of Medicine (1993)
[20] Ghosn, J., Bengio, Y.: Multi-Task Learning for Stock Selection. In: NIPS 1996 (1997)
[21] Heskes, T.: Solving a Huge Number of Similar Tasks: A Combination of Multitask Learning and a Hierarchical Bayesian Approach. In: Proceedings of the 15th International Conference on Machine Learning, Madison, Wisconsin, pp. 233–241 (1998)
[22] Holmstrom, L., Koistinen, P.: Using Additive Noise in Back-propagation Training. IEEE Transactions on Neural Networks 3(1), 24–38 (1992)
[23] John, G., Kohavi, R., Pfleger, K.: Irrelevant Features and the Subset Selection Problem. In: ICML 1994, Rutgers, NJ, pp. 121–129 (1994)
[24] Koller, D., Sahami, M.: Towards Optimal Feature Selection. In: ICML 1996, Bari, Italy, pp. 284–292 (1996)
[25] Le Cun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackal, L.D.: Backpropagation Applied to Handwritten Zip-Code Recognition. Neural Computation 1, 541–551 (1989)
[26] Le Cun, Y.: Private communication (1997)

[27] Munro, P.W., Parmanto, B.: Competition Among Networks Improves Committee Performance. In: Proceedings of Advances in Neural Information Processing Systems, NIPS 1996, vol. 9 (1997) (to appear)
[28] Pomerleau, D.A.: Neural Network Perception for Mobile Robot Guidance. Doctoral Thesis, Carnegie Mellon University: CMU-CS-92-115 (1992)
[29] Pratt, L.Y., Mostow, J., Kamm, C.A.: Direct Transfer of Learned Information Among Neural Networks. In: Proceedings of AAAI 1991 (1991)
[30] Sejnowski, T.J., Rosenberg, C.R.: NETtalk: A Parallel Network that Learns to Read Aloud. John Hopkins: JHU/EECS-86/01 (1986)
[31] Sill, J., Abu-Mostafa, Y.: Monotonicity Hints. In: Proceedings of Neural Information Processing Systems, NIPS 1996, vol. 9 (1997) (to appear)
[32] Suddarth, S.C., Holden, A.D.C.: Symbolic-neural Systems and the Use of Hints for Developing Complex Systems. International Journal of Man-Machine Studies 35(3), 291–311 (1991)
[33] Suddarth, S.C., Kergosien, Y.L.: Rule-injection Hints as a Means of Improving Network Performance and Learning Time. In: Proceedings of EURASIP Workshop on Neural Nets, pp. 120–129 (1990)
[34] Thrun, S.: Explanation-Based Neural Network Learning: A Lifelong Learning Approach. Kluwer Academic Publisher (1996)
[35] Thrun, S., Pratt, L. (eds.): Machine Learning. Second Special Issue on Inductive Transfer (1997)
[36] Thrun, S., Pratt, L. (eds.): Learning to Learn. Kluwer (1997)
[37] Valdes-Perez, R., Simon, H.A.: A Powerful Heuristic for the Discovery of Complex Patterned Behavior. In: Proceedings of the 11th International Conference on Machine Learning, ML 1994, Rutgers University, New Jersey, pp. 326–334 (1994)
[38] Weigend, A., Rumelhart, D., Huberman, B.: Generalization by Weight-Elimination with Application to Forecasting. In: Proceedings of Advances in Neural Information Processing Systems, NIPS 1990, vol. 3, pp. 875–882 (1991)