Algorithms and Applications for Multitask Learning

Rich Caruana
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
caruana@cs.cmu.edu

Abstract

Multitask Learning is an inductive transfer method that improves generalization by using domain information implicit in the training signals of *related* tasks as an inductive bias. It does this by learning multiple tasks in parallel using a shared representation. Multitask transfer in connectionist nets has already been proven. But questions remain about how often training data for useful extra tasks will be available, and if multitask transfer will work in other learning methods. This paper argues that many real world problems present opportunities for multitask learning if they are not first overly sanitized. We present eight prototypical applications of multitask transfer where the training signals for related tasks are available and can be leveraged. We also outline algorithms for multitask transfer in decision trees and k-nearest neighbor. We conclude that multitask transfer has broad utility.

1 Introduction

The goal of inductive transfer is to leverage additional sources of information to improve the performance of learning on the current task. The extra sources of information can take many forms, including:

- background domain knowledge
- models for the same task learned with other learning methods
- models for the same task learned from different distributions
- training signals for or learned models of related tasks from the same domain

Inductive transfer can be used to improve generalization accuracy, the speed of learning, and the intelligibility of learned models. Here we focus on improving accuracy. One way transfer improves generalization is by providing a stronger inductive bias [11] than would be available without the extra knowledge.

Multitask Learning (MTL) uses the training signals for related tasks as an inductive bias to improve generalization. This bias is domain-specific because it derives from knowledge sources specific to the domain. MTL works by training tasks in parallel using a shared representation; what is learned for each task can benefit others. The simplest way to do this in backprop nets is to add extra tasks (extra outputs) to a net using one large hidden layer for all tasks, as shown in Figure 1. The basic idea is that extra tasks' error gradients move the shared hidden layer towards representations that better reflect regularities of the domain as a whole, and these better support learning the main task. In Figure 1, Task 1 is affected by Tasks 2–4 only by their effect on the shared hidden layer. The outputs for Tasks 2–4 are ignored when the net is used to make predictions for Task 1. (More complex architectures and algorithms than backprop on a fully connected hidden layer sometimes work better.)

The benefit of MTL in connectionist nets has been established [4][22][3], and some of the underlying mechanisms elucidated [2]. But how useful is multitask transfer? How often will training data be available for extra tasks that are usefully related to the main task? Will multitask transfer work with learning methods other than neural nets? This paper addresses these questions by presenting eight prototypical applications where extra tasks will be available, and by presenting algorithms for multitask transfer in decisions trees and k-nearest neighbor. Most real world learning problems fit at least one of these prototypes and can use one of these learning methods.

Section 2 presents prototypical applications for MTL, some of which may be surprising. Empirical results

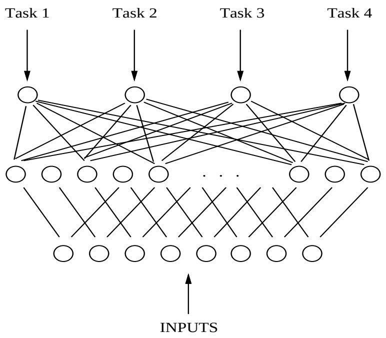

Figure 1: Multitask Learning With Backpropagation of 4 Related Tasks Defined on the Same Inputs

are presented for most of the applications. Section 3 outlines algorithms for multitask transfer in decision trees and k-nearest neighbor. An interesting feature of these algorithms is that they learn to explicitly weight the contributions of extra tasks. Section 4 discusses issues that arise when applying MTL to real problems.

1.1 Terminology

We use these terms: Single task learning (STL) refers to the common approach in machine learning of training tasks one at a time. MTL refers to training extra tasks at the same time the main task is learned using a shared representation. ANN refers to artificial neural nets trained with backpropagation [18]. Standard techniques such as early stopping and finding good net sizes are used in all our experiments. TDIDT is top-down induction of decision trees using ID3/C4.5-like algorithms [14][15]. KNN refers to k-nearest neighbor and kernel regression. SSE is sum-of-squares errors.

2 Prototypical Uses of MTL

This section presents eight domain types where training signals for extra tasks used by MTL are often available. We believe most real-world problems fall into one or more of these classes. This claim might sound surprising given that few of the test problems traditionally used in machine learning are multitask problems. Later, we suggest that most of the problems traditionally used in ML have been so preprocessed to fit STL that the opportunities for MTL were eliminated before learning was attempted.

2.1 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. 3,000,000 people are diagnosed with pneumonia each year in the U.S., 900,000 of whom are admitted to the hospital. A primary goal in medical decision making is to accurately, swiftly, and economically identify high risk patients so they may be hospitalized to receive aggressive testing and treatment; patients at low risk may more comfortably, safely, and economically be treated at home. A decision aid to help assess patient risk *prior to hospitalization* would be of great value.

The Medis Pneumonia Database [10] contains 15,000 pneumonia cases. 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. The diagnosis of pneumonia has already been made; the goal is to predict how much risk pneumonia poses to the patient. Some of the most useful tests for assessing risk become available only after the patient is hospitalized.

Performance on this problem is measured by how accurately one can select a fraction of the patients (FOP) that do not die. For example, we might wish to use learning to find 20% FOP at least risk of dying from pneumonia. If 10 of every 1000 patients predicted to be in this FOP die, we say the error rate at FOP = 0.20 is 0.01. The goal is to minimize the error rates.

The standard ANN approach uses the 30 measurements available prior to hospitalization as inputs to a net learning to predict risk, and ignores the 35 hospital lab results because they will not be available when the decision to admit must be made. MTL, however, benefits from the hospital lab tests in the database by using them as extra output tasks, as shown in Figure 2. The net learns to predict risk, *and the results of the hospital lab tests,* from the 30 measurements available prior to admission.

Table 1 shows the mean performance of ten runs of STL and MTL, and the percent improvement of MTL over STL. MTL lowers error at each FOP. This improvement is not easy to achieve—experiments with other learning methods such as Bayes Nets, Hierarchical Mixtures of Experts, and K-Nearest Neighbor (run not by us, but by experts in their use) indicate STL is an excellent performer in this domain [8]. Feature nets [9], an approach that trains nets to predict the future labs and then uses the predictions as extra *inputs* does

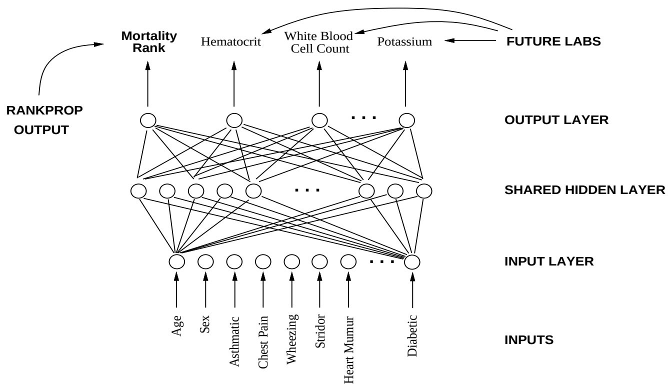

Figure 2: Using Future Lab Results as Extra Outputs To Bias Learning

not yield comparable benefits to MTL here.

Table 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—road stripes can be detected 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 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 when driving. As MTL outputs, though, they provide extra information that helps learning without requiring they be available at run time.

2.2 Multiple Representations and 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 2.2. There we used an error metric called *rankprop* [3] 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 can have 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. 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?

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 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.

Table 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%

Sometimes it is not apparent which output encoding will work best. Alternate codings of the main task can be used as extra outputs the same way alternate error metrics were used above. For 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 be correct, a non-distributed representation can be more accurate. MTL is one way to merge these conflicting requirements in one net.

2.3 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.

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. Figure 1 showed an MTL net with four outputs. If output k referred to the prediction for the time series task at time T_k, this net makes predictions for the same task at four different times. Often, the output used for prediction would be the middle one (temporally) so that there are tasks earlier and later than it trained on the net. Or, as input features temporally "slide" across the inputs, one can collect the outputs from a sequence of predictions and combine them.

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. For example, 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. 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 travelled, so our data might be described as a spatial series.)

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 on each prediction distance is compared in Table 3 with separate 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 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. When we recognize this we use STL where it is best and MTL everywhere else. Note that the simpler STL tasks must still be used on the MTL net to help the harder tasks.

2.4 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.

Figure 3: Sample Doors from the Doors Domain

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. As an example, we used a mouse to define the following features from images of doorways collected from a robot-mounted camera like those in Figure 3. The first two, the horizontal location of the doorknob and doorway center, are the main tasks. A human had to process each image to define the doorknob location and doorway center, so it was easy to collect the additional features at the same time.

- horizontal location of doorknob

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

Table 4 shows the SSE performance of STL and MTL nets learning to predict doorknob location and doorway center. The STL nets each have one output: doorknob location or doorway center. The MTL net has 8 outputs, one for each task. Note that doorknob location is used as an extra task when learning doorway center, and vice versa. The MTL net achieves 20-30% lower error when trained on the same data set. *The MTL data set does not contain more training patterns than the STL data set, it just contains extra training signal for each training pattern.*

Table 4: STL and MTL on Doorway Domain

TASK	STL	MTL	% Change
Doorknob Loc	.081	.062	-23.5 %
Door Type	.086	.059	-31.4 %

2.5 Using Extra Tasks to Focus Attention

Learners often learn to use large, ubiquitous patterns in the inputs, while ignoring small or less common inputs that are 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 input patterns it might ignore. 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 to 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.

We used a road image simulator developed by Pomerleau to generate synthetic road images. The main tasks are 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:

- is there a centerline in image?
- does the road have 1 or 2 lanes?
- x-location of centerline (2 lane roads only)

Table 5 shows the performance of STL and MTL on the two main tasks. Error is reduced 10-25% by requiring the MTL net to also learn the centerline tasks. Once again, note that the training samples used for STL and MTL are the same; the sample sizes have not changed. All that has changed is that more training signal is available in the MTL training patterns. This extra training signal forces the MTL net to learn to attend to regions in the input image that support the centerline tasks, and what is learned for the centerline tasks is beneficial to the main tasks. We call this *eavesdropping* [2].

Table 5: STL and MTL on Autonomous Navigation

TASK	STL	MTL	% Change
Road Center	.039	.034	-12.8 %
Steering	.080	.058	-27.5 %

2.6 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 understanding experts, but good at learning from examples. MTL is a way to collect domain-specific inductive bias from an expert and give it to a learner using the strengths of each. Having domain experts define "helper" tasks is a convenient way to use human expertise to bias learning. Using extra outputs to hint ANNs is well documented in [1], so we do not discuss it here.

2.7 Sequential Transfer

Sometimes we already have a domain theory for similar tasks from prior learning. The data used to train these models, however, may no longer be available. Can MTL benefit from the models without the training data? Yes. One way is to generate synthetic data from the models and use it as extra MTL tasks. This approach to sequential transfer elegantly sidesteps catastrophic interference (forgetting old tasks while learning new ones), and is applicable even where analytical methods of evaluating domain theories used by other serial transfer methods [17][21] are not available. For example, the domain theory need not be differentiable.

We've only tested this on synthetic problems. It works well if the prior models were accurate. An 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 those models make as extra MTL outputs when learning the new main task. This sampling may not always be satisfactory. If the 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 [5] for a thorough discussion of synthetic sampling.

2.8 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 [19]. 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 [7].

A recent example of multiple tasks arising naturally is Mitchell's Calendar Apprentice System (CAP) [6][12]. 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 by as much as 10%.

2.9 Outputs Can Beat Inputs

Sections 2.1–2.8 present domains where it is impractical to use some features as *inputs*. In these, MTL provides a way of benefiting from these features (instead of ignoring them) by using them as extra tasks. It is worth noting, however, that features in some domains can be more useful when used as outputs than as inputs. In fact, it is possible to construct problems with features that hurt performance when used as inputs but help performance when used as outputs. We are currently working to precisely characterize when features are better used as inputs or as outputs.

3 Other MTL Algorithms

The goal of this paper is to show that MTL is broadly applicable. The previous section showed that there are opportunities for MTL in many domains. This section shows that MTL can be used with learning methods other than backprop nets by presenting MTL algorithms for decision tree induction and k-nearest neighbor. Details of the algorithms, and empirical demonstration, are the subject of future papers.

3.1 MTL TDIDT

The basic recursive step in TDIDT is to determine which split 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.

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...

Why do this? Much effort is spent in TDIDT to find *good* splits. Usually, the only information available is how well the splits separate classes on a single task. In a multitask decision tree one can evaluate splits by how well they separate classes from multiple tasks. If the tasks are related by being drawn from a common domain, preferring splits that have utility to multiple tasks drawn from that domain should improve the quality of the selected splits.

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 TDIDT algorithm presented in [4] combines task gains by averaging them; the selected splits are the ones whose average utility across all tasks is highest.

There is a problem 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.[1] As the number of tasks grows large, fewer splits in the tree will be optimal for any one task, and 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

[1] 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.

MTL decision tree for it. This gives us freedom to use the splits prefered by other tasks only if they help the main task. How do we do this?

Averaging the split 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.

How are task weights assigned? We hillclimb on a hold-out test set 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 tree to keep track of the smallest weight changes that will alter the learned tree. This allows us to use steepest descent hillclimbing.

This approach to learning task weights requires TDIDT 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.

3.2 MTL KNN/Kernel Methods

K-nearest neighbor methods use a distance metric defined on attributes to locate training cases close to the probe case. The predicted class for the probe is the majority class of the k closest cases. Kernel methods are similar to k-nearest neighbor in that they use a distance metric to determine how similar the probe case is to each case in the training set. This distance is then used to weight the contribution of each case to the prediction for the probe. We use KNN to refer to both these methods because they are so similar from the MTL point-of-view.

The performance of KNN depends on the quality of the distance metric used to define similarity. Sometimes, simple Euclidean distance is adequate. Usually, better performance is obtained by using a weighted Euclidean distance where the weights determine how important each attribute is when determining similarity. The attribute weights must be learned from the training data, usually by some form of gradient descent or line search.

Finding good attribute weights is essential to good performance with KNN. MTL can be used to find better weights. The basic approach is to find attribute weights that yield good KNN performance on a set of related tasks drawn from some domain. Each case in the training set has labels for multiple tasks, and the attribute weights are optimized to yield good performance across these tasks.

As in MTL TDIDT, it is possible for many extra tasks to swamp the main task if the main task is too different. To prevent this, we use a task weighting scheme similar to that used in MTL TDIDT. *Do not confuse the attribute weights learned for the distance function with the task weights that determine how sensitive the multitask procedure is to each task.*

One attractive feature of MTL kernel methods is that their performance is continuous with respect to both the attribute weights and the MTL task weights. This makes gradient descent feasible for both searches. Optimizing the task weights is, however, still an expensive process because we have nested optimization. Partial compensation for this comes from the efficiency of leave-one-out cross-validation with KNN. Also, it is not necessary to wait until the inner attribute weight search converges before making a new step in the outer task weight search; in practice it usually suffices to use a slower learning rate for the outer MTL task weight loop than for the inner weighted distance loop.

4 Discussion

4.1 Predictions for Multiple Tasks

MTL trains many tasks on one learner. This does not mean one learned model must be used to *predict* those tasks. The motivation for training multiple tasks on one learner is to benefit the main task by the information contained in the training signals for other tasks, *not to reduce the number of models!* Often, a tradeoff is made between mediocre performance on all tasks and optimal performance on any one. The task weighting mechanism in the MTL TDIDT and KNN algorithms presented in Section 2 makes this explicit; the learner can ignore some tasks to achieve better performance on the main task.

Where predictions for several of the tasks are required (as in CAP, Section 2.8), it is important to retrain for each task. This is less important with MTL in backprop nets if one uses an architecture that treats all tasks equally, and if there is sufficient capacity in the hidden layer to allow parts of the hidden layer to become dedicated exclusively to single tasks. If early stopping is used, however, it is important to stop tasks individually; not all tasks train—or overtrain—at the same time.

4.2 Parallel vs. Sequential Transfer

MTL is parallel transfer. It might seem that sequential transfer [16][17][20][21][22] would be easier. This does not seem to be the case. Difficulties with serial transfer include:

- the sequence often must be defined manually
- it is difficult to scale sequential transfer to many tasks
- if Task 1 is learned before Task 2, Task 2 can't help Task 1, and this can reduce Task 1's ability to help Task 2
- in sequential transfer it is difficult to balance the need to be strongly biased by what has been learned before with the flexibility to learn new things (this problem is called catastrophic interference)
- if tasks are learned poorly in isolation, sequential transfer will perform poorly because it lacks the combined inductive bias at the critical early stages of learning

The advantage of parallel transfer over sequential transfer is that the full detail of what is being learned internally for all tasks is available during parallel transfer (because the learning for all tasks happens at the same time), and tasks often benefit each other mutually, something a linear sequence cannot capture. It is interesting to note that it is relatively straightforward to use parallel transfer to accomplish sequential transfer via the synthetic data method of Section 2.7, but it does not appear to be easy to use sequential transfer to gain the advantages of parallel transfer.

4.3 When Inductive Transfer Hurts

We've seen several cases where MTL reduces performance by a small amount. In pneumonia, performance dropped for high risk cases when an extra SSE output was added to the rankprop net (see Section 2.2). This 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 are good. Some inductive biases are bad. It depends on the problem.* Since MTL derives its inductive bias from the training signals of the related tasks, the appropriateness of the MTL bias to the task at hand depends on the relationship(s) between the main task and the extra tasks, and on the transfer mechanism(s) employed.

There may never be strong theory to predict when the bias acquired from related tasks will help or hurt. In the absence of such a theory, the safest approach is to treat MTL as a new tool that must be tested on each domain. Fortunately, on most problems we have tried, MTL seems to help. Algorithms that automatically adjust the MTL bias using cross-validation, such as those used for TDIDT and KNN, are important steps for making MTL useful in practice.

4.4 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 too sanitized. MTL thrives on extra information. The traditional practice in machine learning is to carefully engineer away complexity so that traditional learning techniques will work. We've discovered that opportunities for MTL usually decrease as one gets further removed from the raw data or the data collection process! For example, if on the pneumonia problem we had not also been involved in a different experiment where all features could be used as inputs, we would not have known about the availability of the extra lab tests because the database creators planned to eliminate them for the admission task. MTL provides new ways of using information that are not always immediately obvious from the traditional STL point-of-view.

5 Summary

Multitask transfer can be applied to backprop nets, decision trees, and k-nearest neighbor or kernel methods. It improves generalization performance by using domain-specific knowledge implicit in the training signals for related tasks as an inductive bias. Few traditional machine learning tasks have extra tasks associated with them. Nevertheless, there are a surprising number of ways in which the training signals for extra related tasks can become available on real-world problems. Eight of these are:

1. using the future to predict the present
2. multiple representations and error metrics
3. time series prediction
4. using non-operational features
5. using extra tasks to focus attention
6. hints (knowledge acquisition via extra tasks)
7. sequential transfer
8. multiple tasks arise naturally

We conjecture that as machine learning is applied to unsanitized, real-world problems, the opportunity for multitask transfer will increase dramatically. This, combined with the ability to incorporate multitask transfer in three of the most successful learning algorithms to date, suggests to us that multitask transfer may have considerable utility in machine learning.

Acknowledgements

We thank Greg Cooper, Michael Fine, and other members of the Pitt/CMU Cost-Effective Health Care group for help with the Medis Database; 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, David Zabowski, and other members of the Calendar Apprentice Project for help in collecting and using the CAP data. The work to characterize which features are more useful as inputs or as outputs is joint work with Virginia de Sa.

This work was supported by ARPA grant F33615-93-1-1330, NSF grant BES-9315428, and Agency for Health Care Policy and Research grant HS06468

References

[1] Y.S. Abu-Mostafa, "Learning From Hints in Neural Networks," *Journal of Complexity* **6**:2, pp. 192-198, 1989.

[2] R. Caruana, "Learning Many Related Tasks at the Same Time With Backpropagation," *Advances in Neural Information Processing Systems 7,* pp. 656-664, 1995.

[3] R. Caruana, "Using the Future to "Sort Out" the Present: Rankprop and Multitask Learning for Medical Risk Prediction," to appear in *Advances in Neural Information Processing Systems 8,* 1996.

[4] R. Caruana, "Multitask Learning: A Knowledge-Based Source of Inductive Bias," *Proceedings of the 10th International Conference on Machine Learning,* pp. 41-48, 1993.

[5] M. Craven and J. Shavlik, "Using Sampling and Queries to Extract Rules from Trained Neural Networks," *Proceedings of the 11th International Conference on Machine Learning,* pp. 37-45, 1994.

[6] L. Dent, J. Boticario, J. McDermott, T. Mitchell, and D. Zabowski, "A Personal Learning Apprentice," *Proceedings of 1992 National Conference on Artificial Intelligence*, 1992.

[7] T.G. Dietterich, H. Hild, and G. Bakiri, "A Comparative Study of ID3 and Backpropagation for English Text-to-speech Mapping," *Proceedings of the Seventh International Conference on Artificial Intelligence,* pp. 24–31, 1990.

[8] G. Cooper, et al., "An Evaluation of Machine Learning Methods for Predicting Pneumonia Mortality," submitted to *AI in Medicine,* 1995.

[9] I. Davis and A. Stentz, "Sensor Fusion For Autonomous Outdoor Navigation Using Neural Networks," *Proceedings of IEEE's Intelligent Robots and Systems Conference,* 1995.

[10] M. Fine, D. Singer, B. Hanusa, J. Lave, and W. Kapoor, "Validation of a Pneumonia Prognostic Index Using the MedisGroups Comparative Hospital Database," *American Journal of Medicine,* **94** 1993.

[11] T.M. Mitchell, "The Need for Biases in Learning Generalizations," Rutgers University: *CBM-TR-117,* 1980.

[12] T. Mitchell, R. Caruana, D. Freitag, J. McDermott, D. Zabowski, "Experience with a Learning Personal Assistant," *Communications of the ACM: Special Issue on Agents,* Vol. 37:7, July 1994.

[13] D.A. Pomerleau, "Neural Network Perception for Mobile Robot Guidance," Carnegie Mellon University: *CMU-CS-92-115,* 1992.

[14] J.R. Quinlan, "Induction of Decision Trees," *Machine Learning,* **1**:81-106, 1986.

[15] J.R. Quinlan, *C4.5: Programs fro Machine Learning,* Morgan Kaufman Publishers, 1992.

[16] L.Y. Pratt, J. Mostow, and C.A. Kamm, "Direct Transfer of Learned Information Among Neural Networks," *Proceedings of AAAI-91,* 1991.

[17] L.Y. Pratt, "Non-literal Transfer Among Neural Network Learners," Colorado School of Mines: *MCS-92-04,* 1992.

[18] D.E. Rumelhart, G.E. Hinton, and R.J. Williams, "Learning Representations by Back-propagating Errors," *Nature,* **323**:533-536, 1986.

[19] T.J. Sejnowski and C.R. Rosenberg, "NETtalk: A Parallel Network that Learns to Read Aloud," John Hopkins: *JHU/EECS-86/01,* 1986.

[20] N.E. Sharkey and A.J.C Sharkey, "Adaptive Generalisation and the Transfer of Knowledge," University of Exeter: *R257,* 1992.

[21] S. Thrun and T. Mitchell, "Learning One More Thing," CMU TR: CS-94-184, 1994.

[22] S. Thrun, "Lifelong Learning: A Case Study," CMU TR: CS-95-208, 1995.