Routing Networks: Adaptive Selection of Non-linear Functions for Multi-Task Learning

Clemens Rosenbaum
College of Information and Computer Sciences
University of Massachusetts Amherst
140 Governors Dr., Amherst, MA 01003
`cgbr@cs.umass.edu`

Tim Klinger & Matthew Riemer
IBM Thomas J. Watson Research Center
1101 Kitchawan Rd, Yorktown Heights, NY 10598
`{tklinger,mdriemer}@us.ibm.com`

Abstract

Multi-task learning (MTL) with neural networks leverages commonalities in tasks to improve performance, but often suffers from task interference which reduces transfer. To address this issue we introduce the routing network paradigm, a novel neural network unit and training algorithm. A routing network is a kind of self-organizing neural network consisting of two components: a *router* and a set of one or more *function blocks*. A function block may be any neural network – for example a fully-connected or convolutional layer. Given an input the router makes a routing decision, choosing a function block to apply and passing the output back to the router recursively, terminating when the router decides to stop or a fixed recursion depth is reached. In this way the routing network dynamically composes different function blocks for each input. We employ a collaborative multi-agent reinforcement learning (MARL) approach to jointly train the router and function blocks. We evaluate our model on multi-task settings of the MNIST, mini-imagenet, and CIFAR-100 datasets. Our experiments demonstrate significant improvement in accuracy with sharper convergence over challenging joint training baselines for these tasks.

1 Introduction

Multi-task learning (MTL) is a paradigm in which multiple tasks must be learned simultaneously. Tasks are typically separate prediction problems, each with their own data distribution. In an early formulation of the problem, (Caruana, 1997) describes the goal of MTL as improving generalization performance by "leveraging the domain-specific information contained in the training signals of related tasks." This means a model must leverage commonalities in the tasks (positive transfer) while minimizing interference (negative transfer). In this paper we propose a new architecture for MTL problems called a *routing network*, which consists of two trainable components: a *router* and a set of *function block*s. Given an input, the router selects a function block from the set, applies it to the input, and passes the result back to the Router, recursively. The router can decide at any point to stop routing and is in practice limited to a fixed recursion limit. Intuitively, the architecture allows the network to dynamically self-organize in response to the input, sharing function blocks for different tasks when positive transfer is possible, and using separate blocks to prevent negative transfer.

Because routers make a sequence of hard decisions, which are not differentiable, we use reinforcement learning (RL) to train them. We discuss the training algorithm in Section 2, but we model this as an RL problem by creating a separate RL agent for each task (assuming task labels are available in the dataset). Each such task agent learns its own policy for routing instances of that task to one of the function blocks.

To evaluate this architecture we have created a "routed" version of the convnet architecture used in (Ravi & Larochelle, 2017) and use three image classification datasets adapted for MTL learning: a multi-task MNIST dataset that we created, a Mini-imagenet data split as introduced in (Vinyals et al., 2016), and the 20 superclasses of CIFAR-100 (Krizhevsky, 2009) each treated as different tasks. We conduct extensive experiments comparing against both single task and the popular strategy of joint training with layer sharing as described in Caruana (1997). Our results indicate a significant improvement in accuracy over these strong baselines with a speedup in convergence.

2 Routing Networks

A routing network consists of two components: a *router* and a set of *function blocks*, which can be any neural network. At each iteration the router processes its input to select a block, applies the block to the input, and recursively processes the result. This process is illustrated in Figure 1. The dashed line shows the router accepting the input and selecting a function f_i. The solid line shows the application of f_i to the input and the wide dashed line indicates the recurrent loop. If the function blocks are of different dimensions then the router is constrained to select dimensionally conformant blocks to apply. Algorithm 1 gives the algorithm for evaluating a routing network on an input. If the routing network is run for d invocations then we say it has *depth* d. For N function blocks a routing network run to a depth d can select from N^d distinct trainable functions.

Any neural network can be represented as a routing network by adding copies of its layers as routing network function blocks. We can group the function blocks for each network layer and constrain the router to pick from layer 0 function blocks at depth 0, layer 1 blocks at depth 1, and so on. If the number of function blocks differs from layer to layer in the original network, then the router may accommodate this by, for example, maintaining a separate decision function for each depth.

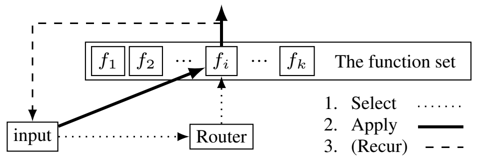
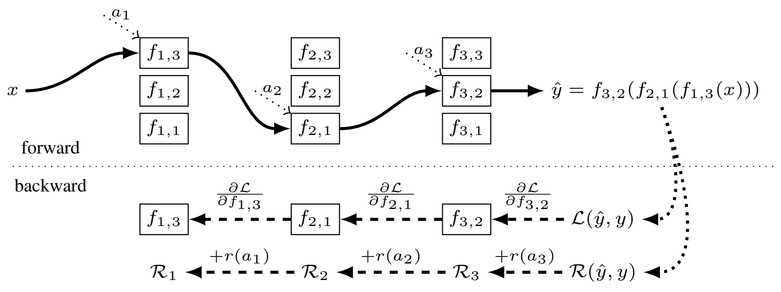


Figure 1: Routing

Algorithm 1: Routing Algorithm

```
Data: input $x$; $d$ the recurrence depth
1 output = $x$
2 for $i$ in $1..d$ do
3     $f$ = router(output)
4     output = $f$(output)
5 return output
```

Both the router and function blocks need training. A high-level view of the training procedure is shown in Figure 2. First, given an input x (which may include a task label), we compute a route and output value $\hat{y}$ using the procedure given in Algorithm 1. We then apply a loss function $\mathcal{L}$ and



a_1
a_2
a_3
x
$f_{1,3}$
$f_{1,2}$
$f_{1,1}$
$f_{2,3}$
$f_{2,2}$
$f_{2,1}$
$f_{3,3}$
$f_{3,2}$
$f_{3,1}$
$\hat{y} = f_{3,2}(f_{2,1}(f_{1,3}(x)))$
forward
backward
$\frac{\partial \mathcal{L}}{\partial f_{1,3}}$
$\frac{\partial \mathcal{L}}{\partial f_{2,1}}$
$\frac{\partial \mathcal{L}}{\partial f_{3,2}}$
$\mathcal{L}(\hat{y}, y)$
$+r(a_1)$
$+r(a_2)$
$+r(a_3)$
$\mathcal{R}_1$
$\mathcal{R}_2$
$\mathcal{R}_3$
$\mathcal{R}(\hat{y}, y)$

Figure 2: Forward and Backward Passes of Training. x is the input; a_i is the action representing decision i; $f_{i,j}$ is a function block; $r(a_i)$ is the per-action reward for taking action a_i; $\mathcal{R}_i$ is the cumulative reward for action a_i

compute gradient updates using SGD/backprop. To train the router we employ a reinforcement learning approach. Specifically, we use the output $\hat{y}$ to compute a reward value of +1 if the prediction is correct and -1 if it is incorrect. We then accumulate this reward back through each of the routing decisions (along with any per-action rewards) to compute the cumulative discounted reward $\mathcal{R}_i$ at each decision point i. Many reinforcement learning algorithms are applicable in this setting. We experimented with Policy Gradient (PG), Q-Learning, and multi-agent (MARL) algorithms which we describe in more detail below.

One important aspect of the router training is the design of a reward structure which incentivizes good function block choices. We consider two rewards: a final reward, and a per-action reward that we shape to motivate collaboration. The final reward is $+1$ if the instance is correctly classified and -1 if not. We also only consider per-action rewards that are constrained by per-action reward $\leq$ final reward $\cdot \rho$ for $\rho \leq 10$. To incentivize the agents to use as few function blocks as possible we use a collaboration reward which rewards each action an amount proportional to the average agreement among all agents who took that action in the past. We use a collaboration reward proportional to the average over the probabilities (or Q-Values) assigned to that action by any agent that took it in the past.

Vanilla policy gradient methods are less well adapted to the changing environment and changes in other agent's behavior, which may degrade their performance in this setting. One MARL algorithm specifically designed to address this problem, and which has also been shown to converge in non-stationary environments, is the weighted policy learner (WPL) algorithm (Abdallah & Lesser, 2006). WPL is a PG algorithm designed to dampen oscillation and push the agents to converge more quickly. This is done by scaling the gradient of the expected return for an action a according the probability of taking that action $\pi(a)$ (if the gradient is positive) or $1 - \pi(a)$ (if the gradient is negative). Intuitively, this has the effect of slowing down the learning rate when the policy is moving away from a Nash equilibrium strategy and increasing it when it approaches one.

To apply WPL in the routing network context, we use the WPL-Trainer shown in Algorithm 2, that computes a running average of the returns for each action which is then passed to the WPL-Update algorithm. WPL-Update uses these estimates of the expected return for each action to determine the gradient and update the policy. The function simplex-projection projects the updated policy values to make it a valid probability distribution. The projection is defined as: $clip(\pi)/\sum(clip(\pi))$, where $clip(x) = \max(0, min(1, x))$.

Algorithm 2: WPL-Trainer: Routing Network Training Algorithm

```
Params: decision actions $\mathbf{a}$;
decision returns $\mathcal{R}$;
expected decision return estimate learning rate $\lambda_r$;
policy learning rate $\lambda_\pi$
1 $\hat{\mathcal{R}}_k \leftarrow (1 - \lambda_r) * \hat{\mathcal{R}}_k + \lambda_r * \mathcal{R}_k$
2 for each decision $a_k$ do
3     $\Delta(a_k) \leftarrow \mathcal{R}(a_k) - \mathcal{R}_k$
4     if $\Delta(a_k) < 0$ then
5         $\Delta(a_k) \leftarrow \Delta(a_k)(1 - \pi(a_k))$
6     else
7         $\Delta(a_k) \leftarrow \Delta(a_k)(\pi(a_k))$
8     $\pi \leftarrow$ simplex-projection$(\pi + \lambda_\pi \Delta)$
```

3 Results

We experiment with three datasets: multi-task versions of MNIST (MNIST-MTL) (Lecun et al., 1998), Mini-Imagenet (MIN-MTL) (Vinyals et al., 2016) as introduced by (Ravi & Larochelle, 2017), and CIFAR-100 (CIFAR-MTL) (Krizhevsky, 2009) where we treat the 20 superclasses as tasks. In the binary MNIST-MTL dataset, the task is to differentiate instances of a given class c from non-instances. We create 10 tasks and for each we use 1000 instances of the positive class c and 1000 each of the remaining 9 negative classes for a total of 10,000 instances per task during training, which we then test on 200 samples per task. MIN-MTL is a smaller version of ImageNet (Deng et al., 2009) from which we randomly choose 50 labels. We create tasks from 10 random subsets of 5 labels each chosen from these. Each label has 800 training instances and 50 testing instances – so 4000 training and 250 testing instances per task. Finally, CIFAR-100 has coarse and fine labels for its instances. We follow existing work (Krizhevsky, 2009) creating one task for each of the 20 coarse labels and include 500 instances for each of the corresponding fine labels. All results are reported on the test set and are averaged over 3 runs.

Our experiments are conducted on a convnet architecture (SimpleConvNet) which appeared recently in (Ravi & Larochelle, 2017). This model has 4 convolutional layers, each consisting of a 3x3 convolution and 32 filters, followed by batch normalization and a ReLU. The convolutional layers are followed by 3 fully connected layers, with 128 hidden units each. Our routed version of the network routes the 3 fully connected layers and for each routed layer we supply one randomly initialized function block per task in the dataset.

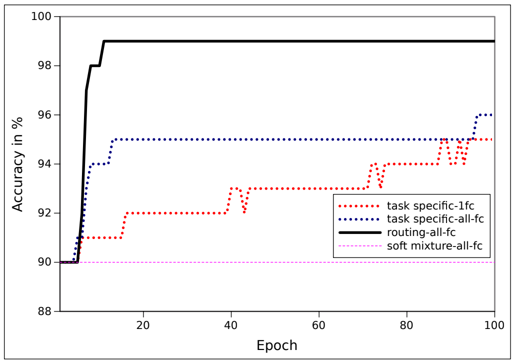


Figure 3: Results on domain MNIST-MTL

We dedicate a special action to allow the agents to skip layers during training which leaves the current state unchanged.

We use the Adam optimization algorithm Kingma & Ba (2014). The collaboration reward parameter $\rho = 0.3$; learning rate = 1e-3 annealed by dividing by 10 every 10 epochs; batch size = 10. The SimpleConvNet has batch normalization layers but we use no dropout and perform no hyper-parameter search or other tuning.

Although limited here by space we have compared the WPL algorithm to many Q Learning and Policy Gradient (PG) variants and have found: (1) the WPL algorithm works best; (2) having multiple agents works better than having a single agent; and (3) with the exception of WPL, using a function approximator works better than tabular versions. (We have not yet modified WPL to work with function approximators).

We experimented by comparing a routed version of the SimpleConvNet, which we denote "routing-all-fc" on different domains against two challenging baselines: task specific-1-fc and task specific-all-fc. task-specific-1-fc has a separate last fully connected layer for each task and shares the rest of the layers. task specific-all-fc has a separate set of all the fully connected layers for each task. These baseline architectures allow considerable sharing of parameters but also grant the network private parameters for each task to avoid interference. The decision about what layers to share is static and does not change on a per-task basis.

The results are shown in Figures 4, 5, and 3. In each case the routing net does better than the baselines, beating them by 7% on CIFAR-MTL, 2% on MIN-MTL. We surmise that the results are better on CIFAR because the task instances have more in common whereas the MIN-MTL tasks are randomly constructed, making sharing less profitable. On MNIST-MTL the random baseline is 90% and the routing net beats the nearest baseline by about 4%. We also ran a soft version that replaced the hard decision over block choices with a softmax but it didn't make progress.

In all cases routing makes a significant difference over the baselines and we conclude that a dynamic policy which learns the function blocks to compose on a per-task basis yields better accuracy and sharper convergence than simple static decisions or the soft attention approach.

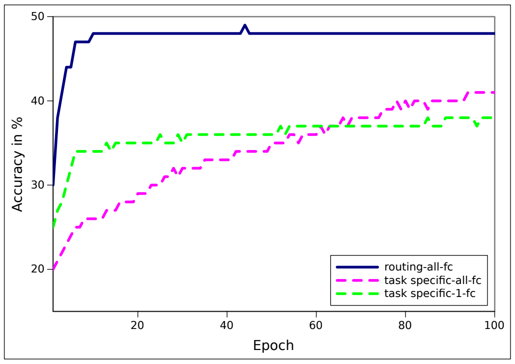


Figure 4: Results on domain CIFAR-MTL

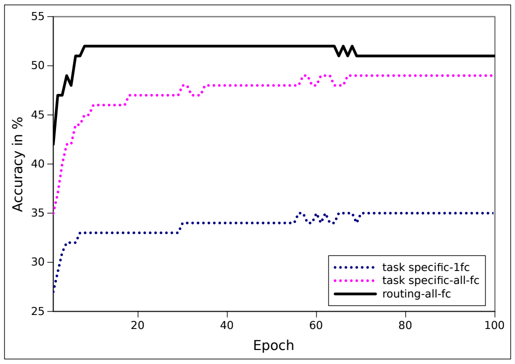


Figure 5: Results on domain MiniImageNet MTL

References

Sherief Abdallah and Victor Lesser. Learning the task allocation game. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pp. 850–857. ACM, 2006. URL http://dl.acm.org/citation.cfm?id=1160786.

Rich Caruana. Multitask learning. *Machine Learning*, 28(1):41–75, Jul 1997. ISSN 1573-0565. doi: 10.1023/A:1007379606734. URL https://doi.org/10.1023/A:1007379606734.

J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980.

Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pp. 2278–2324, 1998.

Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. *ICLR*, 2017.

Oriol Vinyals, Charles Blundell, Timothy P. Lillicrap, Koray Kavukcuoglu, and Daan Wierstra. Matching networks for one shot learning. *CoRR*, abs/1606.04080, 2016. URL http://arxiv.org/abs/1606.04080.