ROUTING NETWORKS: ADAPTIVE SELECTION OF NON-LINEAR FUNCTIONS FOR MULTI-TASK LEARNING

Anonymous authors
Paper under double-blind review

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 a 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 both single task and 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.

The architecture is very general allowing many possible Router implementations. For example, the router can condition its decision on both the input instance and a task label or just one or the other. It can also condition on the depth (number of router invocations), filtering the function module choices to allow layering. In addition, it can condition its decision for one instance on what was historically decided for other instances, to encourage collaboration. The function blocks may be simple fully-connected neural network layers or whole networks as long as the dimensionality of each function block allows composition with the previous function block choice. They needn't even be the same type of unit. Any neural network or part of a network can be "routed" by adding its layers to the set of function blocks, making the architecture applicable to a wide range of problems. Because the 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 3.1, but one way we have modeled this as an RL problem is to create 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.[1] 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 Related Work

Work on multi-task deep learning (Caruana, 1997) traditionally includes significant hand design of neural network architectures, attempting to find the right static mix of task-specific and shared parameters. For example, many architectures share low-level features like those learned in shallow layers of deep convolutional networks or word embeddings across tasks and add task specific architectures in later layers. By contrast, in the Routing Networks paradigm, we learn a fully dynamic, compositional model which can adjust its structure differently for each task.

Routing Networks share a common goal with techniques for automated selective transfer learning using attention (Rajendran et al., 2017) and learning gating mechanisms between representations (Stollenga et al., 2014), (Misra et al., 2016), (Ruder et al., 2017). However, these techniques have not been shown to scale to large numbers of routing decisions and tasks.

Our work is also related to mixtures of experts architectures (Jacobs et al., 1991), (Jordan & Jacobs, 1994) as well as their modern attention based (Riemer et al., 2016) and sparse (Shazeer et al., 2017) variants. The gating network in a typical mixtures of experts model takes in the input and chooses an appropriate weighting for the output of each expert network. This is generally implemented as a soft mixture decision as opposed to a hard routing decision, allowing the choice to be differentiable. Although the sparse and layer-wise variant presented in (Shazeer et al., 2017) does save some computational burden, the proposed end-to-end differentiable model is only an approximation and doesn't model important effects such as exploration vs. exploitation tradeoffs, despite their impact on the system. Mixtures of experts have recently been considered in the transfer learning setting (Aljundi et al., 2016), however, the decision process is modelled by an autoencoder reconstruction error based heuristic and is not scaled to a large number of tasks.

In the use of dynamic representations, our work is also related to single task and multi-task models that learn to generate weights for an optimal neural network (Ha et al., 2016), (Ravi & Larochelle, 2017), (Munkhdalai & Yu, 2017). While these models are very powerful, they have trouble scaling to deep models with a large number of parameters (Wichrowska et al., 2017) without tricks to simplify the formulation. In contrast, we demonstrate that Routing Networks can be applied to create dynamic network architectures for architectures like convnets by routing some of their layers.

Our work extends an emerging line of recent research focused on automated architecture search. In this work, the goal is to reduce the burden on the practitioner by automatically learning black box algorithms that search for optimal architectures and hyperparameters. These include techniques based on reinforcement learning (Zoph & Le, 2017), (Baker et al., 2017), evolutionary algorithms (Miikkulainen et al., 2017), approximate random simulations (Brock et al., 2017), and adaptive growth (Cortes et al., 2016). To the best of our knowledge we are the first to apply this idea to multi-task learning. Our technique can learn to construct a very general class of architectures without the need for human intervention to manually choose which parameters will be shared and which will be kept task-specific.

Perhaps most related to our work is the literature on minimizing computation cost for single-task problems by conditional routing. These include decisions trained with REINFORCE (Denoyer & Gallinari, 2014), (Bengio et al., 2015), (Hamrick et al., 2017), Q Learning (Liu & Deng, 2017), and actor-critic methods (McGill & Perona, 2017). Our approach differs however in the introduction of several novel elements. Specifically, our work explores the multi-task learning setting, it uses a

[1] All dataset splits and the code will be published with the publication of this paper.

multi-agent reinforcement learning training algorithm, and it is structured as a recursive decision process.

While our the Routing Networks paradigm is a novel artificial neural network formulation, the high level idea of task specific "routing" as a cognitive function is well founded in biological studies and theories of the human brain (Gurney et al., 2001), (Buschman & Miller, 2010), (Stocco et al., 2010).

3 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.

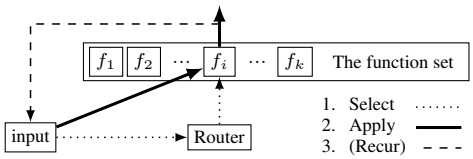

Figure 1: Routing

Algorithm 1: Routing Algorithm

Data: input x; d the recurrence depth
output $= x$
for i *in 1..d* **do**
  f = router(output)
  output = f(output)
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 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. Algorithm 2 summarizes the procedure. Many reinforcement learning

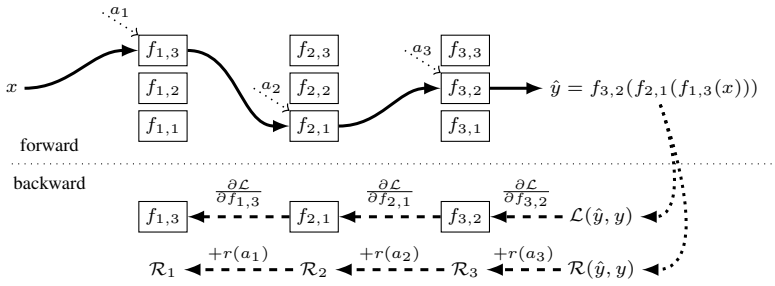

Figure 2: Forward and Backward Passes of Training. x is the input; a_i is the action representing decision i; $f_{i,j}$ is a function block; $r(a_i)$ is the per-action reward for taking action a_i; $\mathcal{R}_i$ is the cumulative reward for action a_i

Algorithm 2: Router-Trainer: Training of a Routing Network.

1 **for** *Each sample* x **do**
2 Do a forward pass through the network, applying Algorithm 1 to sample x, resulting in the network's prediction $\hat{y}$.
3 Compute the loss $\mathcal{L}(\hat{y}, y)$ between prediction $\hat{y}$ and ground truth y.
4 Do regular back propagation of $\mathcal{L}(\hat{y}, y)$ along the selected blocks.
5 Compute some prediction reward $\mathcal{R}(\hat{y}, y)$.
6 Compute the (discounted) cumulative reward at each decision point k:
$\mathcal{R}_k = \sum_{i=k}^{n} r_k + \gamma * \mathcal{R}(\hat{y}, y)$.
(r_k is the per-action reward; $\gamma \in [0, 1]$ is the discount factor hyper-parameter)
7 Train the router's decisions using the actions $\mathbf{a} = (a_1, a_2, \ldots, a_n)$, and their returns $\mathcal{R} = (\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_n)$.

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.

As we have described it, the training of the router and function blocks is performed independently after computing the loss. We have also experimented with adding the gradients from the router choices ($\Delta\mathcal{R}$) to those for the function blocks which produce their input. We found no advantage but leave a more thorough investigation for future work.

3.1 Router Training using RL

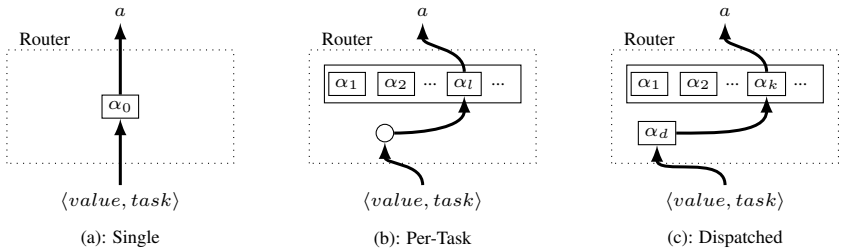

Figure 3: Task-based routing. $\langle value, task \rangle$ is the input consisting of $value$, the partial evaluation of the previous function block (or input x) and the task label $task$. α_i is a routing agent; α_d is a dispatching agent.

The simplest implementation of a router in our approach is as a single RL agent whose policy predicts the next function block given the output value of the previous layer (and possibly a task label). This configuration is shown in Figure 3(a). In practice we found it difficult to achieve good performance in this setting (for reasons we discuss in Section 4) and our best results divide the decision making across multiple separate agents, one per task, as shown in Figure 3(b). Here the assignment of tasks to agents is fixed, not learned during training. When a new instance is presented to the network, its task label is used to identify the corresponding agent which then makes the routing decision. It is also possible to learn to assign the tasks to agents as shown in Figure 3(c). The routing prediction is now made hierarchically with one agent, the *dispatcher* agent, denoted α_d in the figure, choosing from among the task-specific agents based on the task label and input value. We have experimented with all of these architectures and discuss the results in the Section 4.

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 experimented with two versions: a *policy-based* reward, where the reward is proportional to the average over the probabilities (or Q-Values) assigned to that action by any agent that took it in the past; and a *count-based* reward,

where the reward is proportional to the average frequency with which that action was taken in the past. There was no significant difference between them and we use the policy-based reward for all of our experiments.

In the next sections we describe each RL setting used in the experiments in detail. We consider variations in both the policy representation (tabular or approximation function) and learning strategies (PG, Q, and WPL/MARL).

3.1.1 RL ALGORITHMS

We evaluate three reinforcement learning strategies: Policy Gradient (PG), Q-Learning, and the Weighted Policy Learner (WPL), a MARL strategy. Both the Q-Learning and Policy Gradient algorithms are applicable with tabular and approximation function policy representations. We use vanilla PG with the tabular representation and REINFORCE (Williams, 1992) for approximation function representations. The tabular representation has the invocation depth as its row dimension and the function block as its column dimension with the entries containing the probability of choosing a given function block at a given depth. For Q-Learning we use a similar table but store the expected return in the entries. We use regular Q-Learning for the tabular representation and train the approximators to minimize the ℓ_2 norm of the temporal difference error. Specifically, we define a state s to be the current output value of the last function block chosen up to that point (including a task label if one is provided with the input) and the current depth. The depth is necessary to include when routing a layered network which requires filtering function block choices based on depth. If $V(s)$ is the estimated return from state s and the agent has taken action a to reach state s', then the loss function tries to minimize: $((r + \gamma V(s')) - V(s))^2$, where α is the learning rate; r is the reward for taking action a; and γ is the discount factor (set to 1 for all experiments). We have also experimented with a "combinatorial" version of tabular Q learning which maintains a separate row for each combination of function block choices. Since this approach suffers from a combinatorial state explosion it is not scalable to high depths and we include it only for comparison.

In the multi-agent reinforcement learning (MARL) setting multiple agents interact in the environment and the expected return for any given agent may change without any action on that agent's part. Worse, for routing networks, the environment is non-stationary since the function blocks are being trained as well as the router policy. This makes the training considerably more difficult than in the single-agent (MDP) setting. But the MARL approach offers a helpful interpretation of the problem as a stochastic game in which each agent competes and collaborates in the use and training of function blocks. In this view, incompatible agents need to compete for blocks to train, since negative transfer will make collaboration unattractive, while compatible agents can gain in transfer by training the same function block. The agent's (locally) optimal policies will correspond to the game's Nash equilibrium [2].

Algorithm 3: WPL-Update: Policy update using the WPL algorithm

Params: decision actions $\mathbf{a}$;
decision returns $\mathcal{R}$;
expected decision return estimates $\hat{\mathcal{R}}$;
policy learning rate λ_π

for *each decision* a_k **do**
  $\Delta(a_k) \leftarrow \mathcal{R}(a_k) - \mathcal{R}_k$
  if $\Delta(a_k) < 0$ **then**
    $\Delta(a_k) \leftarrow \Delta(a_k)(1 - \pi(a_k))$
  else
    $\Delta(a_k) \leftarrow \Delta(a_k)(\pi(a_k))$
  $\pi \leftarrow$ simplex-projection$(\pi + \lambda_\pi \Delta)$

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), shown in Algorithm 3. 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. We The full WPL-Update algorithm is shown in Algorithm 3. Details, including convergence proofs

[2] A Nash equilibrium is a set of policies for each agent where each agent's expected return will be lower if that agent unilaterally changes its policy

and more examples giving the intuition behind the algorithm, can be found in (Abdallah & Lesser, 2006). The WPL-Update algorithm is defined only for the tabular setting. It is future work to adapt it to work with function approximators.

To apply Algorithm 3 in the routing network context, we use the WPL-Trainer shown in Algorithm 4, that computes a running average of the returns for each action which is then passed to the WPL-Update algorithm. The complete router training algorithm using WPL is then as follows. Replace line 7 of Algorithm 2 with WPL-Trainer($\mathbf{a}$, $\mathcal{R}$), then initialize $\hat{\mathcal{R}}_k \leftarrow 0$ for all actions a_k and run Algorithm 2. After the actions $\mathbf{a}$ and returns $\mathcal{R}$ are computed on line 6, WPL-Trainer will compute an update to the running return averages for each decision a_k then compute the WPL policy update using WPL-Update. 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 4: WPL-Trainer: Routing Network Training Algorithm

Params: decision actions $\mathbf{a}$;
decision returns $\mathcal{R}$;
expected decision return estimate
learning rate λ_r;
policy learning rate λ_π

1 $\hat{\mathcal{R}}_k \leftarrow (1 - \lambda_r) * \hat{\mathcal{R}}_k + \lambda_r * \mathcal{R}_k$

2 WPL-Update($\mathbf{a}$, $\mathcal{R}$, $\hat{\mathcal{R}}$, λ_π)

4 Quantitative 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) which is easier to train in reasonable time periods. For mini-ImageNet we randomly choose 50 labels and 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. For all 10 tasks we have a total of 40000 training instances. 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. There are 20 tasks with a total of 2500 instances per task; 2500 for training and 500 for testing. All results are reported on the test set and are averaged over 3 runs. The data are summarized in Table 1.

Each of these datasets has interesting characteristics which challenge the learning in different ways. CIFAR-MTL is a "natural" dataset whose tasks correspond to human categories. MIN-MTL is randomly generated so will have less task coherence. And MNIST-MTL, while simple, has the difficult property that the same instance can appear with different labels in different tasks, causing interference. For example, in the "0 vs other digits" task, "0" appears with a positive label but in the "1 vs other digits" task it appears with a negative label.

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.

Dataset	**# Training**	**# Testing**
CIFAR-MTL	50k	10k
MIN-MTL	40k	2.5k
MNIST-MTL	100k	2k

Table 1: Dataset training and testing splits

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

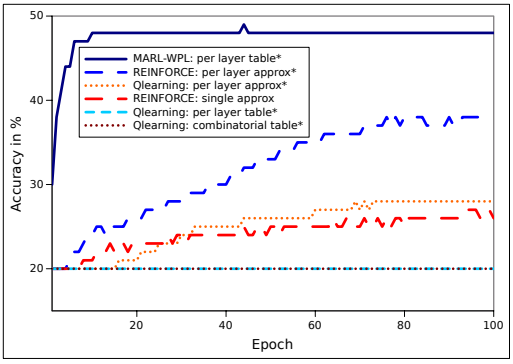

Figure 4: Influence of the RL algorithm on CIFAR-MTL. * means that the approach relies on meta information, i.e. the task label.

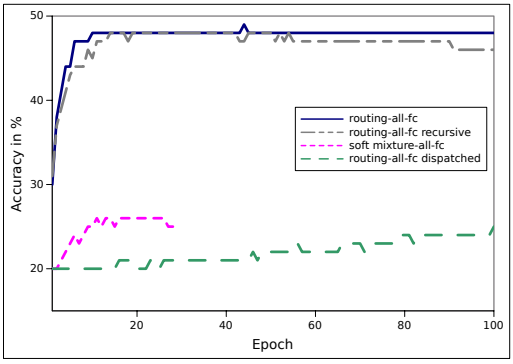

Figure 5: Comparison of Routing Architectures on CIFAR-MTL

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.

All data are presented in Table 2 in the Appendix.

In the first experiment, shown in Figure 4, we compare different RL training algorithms on CIFAR-MTL against the MARL:WPL algorithm which keeps a separate table for each depth to accommodate layering. We compare against an agent-per-task REINFORCE learner which maintains a separate approximation function (a two-layer MLP) for each layer; an agent-per-task Q learner with a separate approximation function (a two-layer MLP) per layer; a single-agent REINFORCE with a separate approximation function (a two-layer MLP) per layer; an agent-per-task Q learner with a separate table for each layer; and an agent-per-task Q learner with a "combinatorial" table which holds the decision history. The best performer is the WPL algorithm which outperforms the nearest competitor (agent-per-task REINFORCE with a separate per-layer decision function) by 10%. We can see that (1) the WPL algorithm works better than a similar vanilla PG; (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.

The next experiment compares the best performing algorithm MARL:WPL against other routing approaches. The MARL:WPL algorithm has the full-connected layers of the SimpleConvNet routed using the layering approach we discussed earlier. We now keep the WPL algorithm but experiment with variations on this setup. To make the next comparison clear we rename this algorithm "routing-all-fc" in Figure 5 to reflect the fact that it routes all the fully connected layers of the SimpleConvNet.

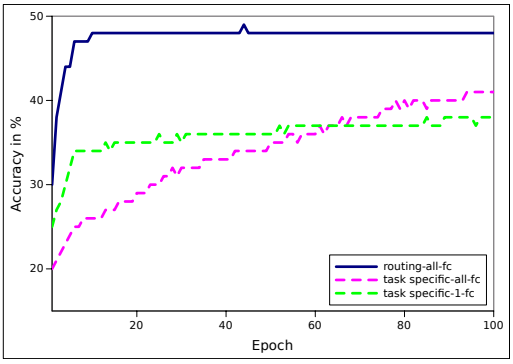

Figure 6: Results on domain CIFAR-MTL

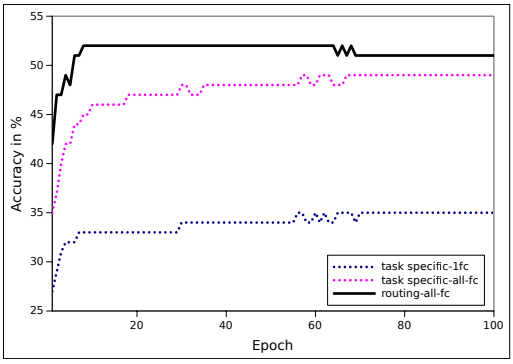

Figure 7: Results on domain MiniImageNet MTL

We compare against several other approaches. One approach, routing-fc-no-layering, has the same setup as routing-all-fc, but does not constrain the router to pick only from layer 0 function blocks at depth 0, etc. It is allowed to choose any function block from two of the layers (since two of the layers have identical input and output dimensions). Another approach, soft-mixture-fc, is a soft version of the router architecture. This soft version utilizes the same function blocks as the routed version, but replaces the hard selection with a trained softmax average over the blocks. We also compare against the dispatched architecture shown in Figure 3(c). Here there is not much difference between the routing-all-fc architecture and the non-layered version. But both architectures are considerably better than the soft and dispatched versions. The soft version has data only to 30 epochs because it is very time-consuming to train. We use the routing-all-fc layer in the remainder of our experiments.

Next we compare the best algorithm 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 6, 7, and 8. 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 the soft version here since it was easy to train 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.

To explore why the multi-agent approach seems to do better than the single-agent, we compared their policy dynamics. In the cases we examined we found that the single agent often chose just 1 or 2 function blocks at each depth, and then routed all tasks to those. We suspect that there is simply too little signal available to the agent in the early, random stages, and once a bias is established its decisions suffer from a lack of diversity.

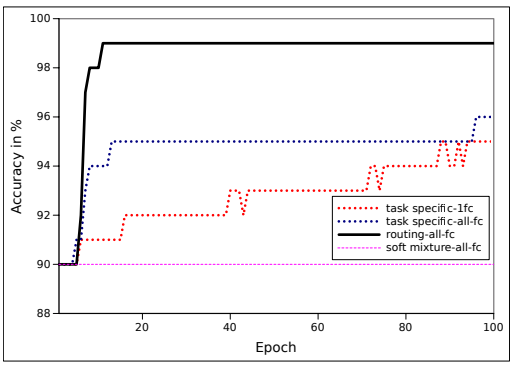

Figure 8: Results on domain MNIST-MTL

The routing network on the other hand learns a policy which, unlike the baseline static models, partitions the network quite differently for each task, and also achieves considerable diversity in its choices as can be seen in Figure 11. This figure shows the routing decisions made over the whole MNIST MTL dataset. Each task is labeled at the top and the decisions for each of the three routed layers are shown below. We believe that because the routing network has separate policies for each task, it is less sensitive to a bias for one or two function blocks and each agent learns more independently what works for its assigned task.

5 Qualitative Results

To better understand the agent interaction we have created several views of the policy dynamics. First, in Figure 9, we chart the policy over time for the first decision. Each rectangle labeled T_i on the left represents the evolution of the agent's policy for that task. For each task, the horizontal axis is number of samples per task and the vertical axis is actions (decisions). Each vertical slice shows the probability distribution over actions after having seen that many samples of its task, with darker shades indicating higher probability. From this picture we can see that, in the beginning, all task agents have high entropy. As more samples are processed each agent develops several candidate

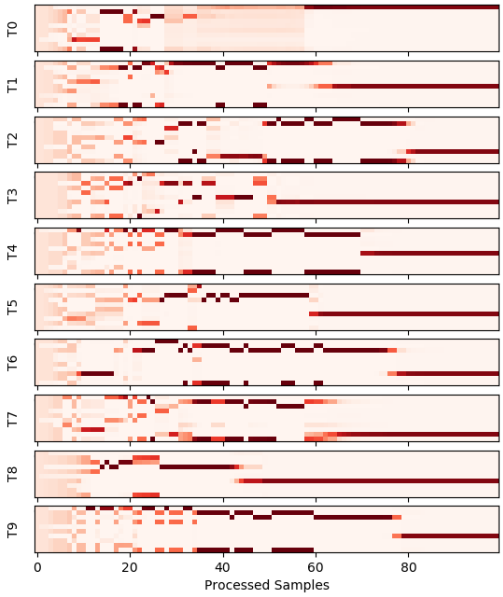

Figure 9: The Policies of all Agents for the first function block layer for the first 100 samples of each task of MNIST-MTL

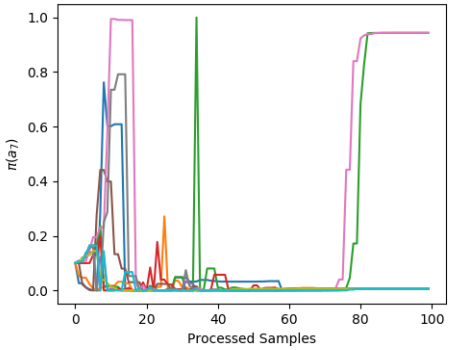

Figure 10: The Probabilities of all Agents of taking Block 7 for the first 100 samples of each task (totalling 1000 samples) of MNIST-MTL

function blocks to use for its task but eventually all agents converge to close to 100% probability for one particular block. In the language of games, the agents find a pure strategy for routing.

In the next view of the dynamics, we pick one particular function block (block 7) and plot the probability, for each agent, of choosing that block over time. The horizontal axis is time (sample) and the vertical axis is the probability of choosing block 7. Each colored curve corresponds to a different task agent. Here we can see that there is considerable oscillation over time until two agents, pink and green, emerge as the "victors" for the use of block 7 and each assign close to 100% probability for choosing it in routing their respective tasks. It is interesting to see that the eventual winners, pink and green, emerge earlier as well as strongly interested in block 7. We have noticed this pattern in the analysis of other blocks and speculate that the agents who want to use the block are being pulled away from their early Nash equilibrium as other agents try to train the block away.

Finally, in Figure 11 we show a map of the routing for MNIST-MTL. Here tasks are at the top and each layer below represents one routing decision. Conventional wisdom has it that networks will benefit from sharing early, using the first layers for common representations, diverging later to accommodate differences in the tasks. This is the setup for our baselines. It is interesting to see that this is not what the network learns on its own. Here we see that the agents have converged on a strategy which first uses 7 function blocks, then compresses to just 4, then again expands to use 5. It is not clear if this is an optimal strategy but it does certainly give improvement over the static baselines.

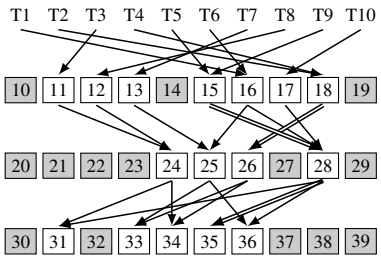

Figure 11: An actual routing map for MNIST-MTL.

6 Future Work

We have presented a general architecture for routing and multi-agent router training algorithm which performs significantly better than baselines and non-multi-agent approaches. The paradigm can eas-

ily be applied to a state-of-the-art network to allow it to learn to dynamically adjust its representations.

As described in the section on Routing Networks, the state space to be learned grows exponentially with the depth of the routing, making it challenging to scale the routing to deeper networks in their entirety. It would be interesting to try hierarchical RL techniques (Barto & Mahadevan (2003)) here.

Our most successful experiments have used the multi-agent architecture with one agent per task, trained with WPL-Trainer. Currently this approach is tabular but we are investigating ways to adapt it to use neural net approximators.

We have also tried routing networks in an online setting, training over a sequence of tasks for few shot learning. To handle the iterative addition of new tasks we add a new routing agent for each and overfit it on the few shot examples while training the function modules with a very slow learning rate. Our results so far have been mixed, but this is a very useful setting and we plan to return to this problem.

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.

Rahaf Aljundi, Punarjay Chakravarty, and Tinne Tuytelaars. Expert gate: Lifelong learning with a network of experts. *arXiv preprint arXiv:1611.06194*, 2016.

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *ICLR*, 2017.

Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379, 2003. URL http://link.springer.com/article/10.1023/A:1025696116075.

Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. Conditional computation in neural networks for faster models. *CoRR*, abs/1511.06297, 2015. URL http://arxiv.org/abs/1511.06297.

Andrew Brock, Theodore Lim, James M. Ritchie, and Nick Weston. SMASH: one-shot model architecture search through hypernetworks. *CoRR*, abs/1708.05344, 2017. URL http://arxiv.org/abs/1708.05344.

Timothy J Buschman and Earl K Miller. Shifting the spotlight of attention: evidence for discrete computations in cognition. *Frontiers in human neuroscience*, 4, 2010.

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.

Corinna Cortes, Xavi Gonzalvo, Vitaly Kuznetsov, Mehryar Mohri, and Scott Yang. Adanet: Adaptive structural learning of artificial neural networks. *arXiv preprint arXiv:1607.01097*, 2016.

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.

Ludovic Denoyer and Patrick Gallinari. Deep sequential neural network. *arXiv preprint arXiv:1410.0510*, 2014.

Kevin Gurney, Tony J Prescott, and Peter Redgrave. A computational model of action selection in the basal ganglia. i. a new functional anatomy. *Biological cybernetics*, 84(6):401–410, 2001.

David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016.

Jessica B Hamrick, Andrew J Ballard, Razvan Pascanu, Oriol Vinyals, Nicolas Heess, and Peter W Battaglia. Metacontrol for adaptive imagination-based optimization. *ICLR*, 2017.

Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

Michael I Jordan and Robert A Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural computation*, 6(2):181–214, 1994.

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

Lanlan Liu and Jia Deng. Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. *arXiv preprint arXiv:1701.00299*, 2017.

Mason McGill and Pietro Perona. Deciding how to decide: Dynamic routing in artificial neural networks. *International Conference on Machine Learning*, 2017.

Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving deep neural networks. *arXiv preprint arXiv:1703.00548*, 2017.

Ishan Misra, Abhinav Shrivastava, Abhinav Gupta, and Martial Hebert. Cross-stitch networks for multi-task learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3994–4003, 2016.

Tsendsuren Munkhdalai and Hong Yu. Meta networks. *International Conference on Machine Learning*, 2017.

Janarthanan Rajendran, P. Prasanna, Balaraman Ravindran, and Mitesh M. Khapra. ADAAPT: attend, adapt, and transfer: Attentative deep architecture for adaptive policy transfer from multiple sources in the same domain. *ICLR*, abs/1510.02879, 2017. URL http://arxiv.org/abs/1510.02879.

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

Matthew Riemer, Aditya Vempaty, Flavio Calmon, Fenno Heath, Richard Hull, and Elham Khabiri. Correcting forecasts with multifactor neural attention. In *International Conference on Machine Learning*, pp. 3010–3019, 2016.

Sebastian Ruder, Joachim Bingel, Isabelle Augenstein, and Anders Søgaard. Sluice networks: Learning what to share between loosely related tasks. *arXiv preprint arXiv:1705.08142*, 2017.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *ICLR*, 2017.

Andrea Stocco, Christian Lebiere, and John R Anderson. Conditional routing of information to the cortex: A model of the basal ganglias role in cognitive coordination. *Psychological review*, 117 (2):541, 2010.

Marijn F Stollenga, Jonathan Masci, Faustino Gomez, and Juergen Schmidhuber. Deep networks with internal selective attention through feedback connections. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (eds.), *Advances in Neural Information Processing Systems 27*, pp. 3545–3553. Curran Associates, Inc., 2014.

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.

Olga Wichrowska, Niru Maheswaranathan, Matthew W Hoffman, Sergio Gomez Colmenarejo, Misha Denil, Nando de Freitas, and Jascha Sohl-Dickstein. Learned optimizers that scale and generalize. *arXiv preprint arXiv:1703.04813*, 2017.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992. ISSN 0885-6125.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *ICLR*, 2017.

APPENDIX

TABLE OF RESULTS

	Epoch	1	5	10	20	50	100
RL (Figure 4)	REINFORCE: per layer approx	20	21	24	26	33	38
	REINFORCE: single approx	20	20	21	23	25	26
	Qlearning: per layer approx	20	20	20	21	26	28
	Qlearning: per layer table*	20	20	20	20	20	20
	Qlearning: combinatorial table*	20	20	20	20	20	20
	MARL-WPL: per layer table*	30	44	48	48	48	48
arch (Figure 5)	routing-all-fc	30	44	48	48	48	48
	routing-all-fc recursive	31	43	45	48	48	46
	routing-all-fc dispatched	20	20	20	20	22	25
	soft mixture-all-fc	20	23	25	26		
	routing-all-fc dispatched	20	20	20	20	22	25
	task specific-all-fc	20	24	26	29	35	41
CIFAR (Figure 6)	routing-all-fc	30	44	48	48	48	48
	task specific-all-fc	20	24	26	29	35	41
	task specific-1-fc	25	32	34	35	36	38
MIN (Figure 7)	routing-all-fc	42	48	52	52	52	51
	task specific-all-fc	35	42	46	47	48	49
	task specific-1fc	27	32	33	33	34	35
MNIST (Figure 8)	routing-all-fc	90	90	98	99	99	99
	task specific-all-fc	90	91	94	95	95	96
	task specific-1fc	90	90	91	92	93	95
	soft mixture-all-fc	90	90	90	90	90	90

Table 2: Numeric results (in % accuracy) for Figures 4 through 8