GUMBEL-MATRIX ROUTING FOR FLEXIBLE MULTI-TASK LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

This paper proposes a novel per-task routing method for multi-task applications. Multi-task neural networks can learn to transfer knowledge across different tasks by using parameter sharing. However, sharing parameters between unrelated tasks can hurt performance. To address this issue, routing networks can be applied to learn to share each group of parameters with a different subset of tasks to better leverage tasks relatedness. However, this use of routing methods requires to address the challenge of learning the routing jointly with the parameters of a modular multi-task neural network. We propose the Gumbel-Matrix routing, a novel multi-task routing method based on the Gumbel-Softmax, that is designed to learn fine-grained parameter sharing. When applied to the Omniglot benchmark, the proposed method improves the state-of-the-art error rate by 17%.

1 INTRODUCTION

Multi-task learning (Caruana, 1998; 1993) based on neural networks has attracted lots of research interest in the past years and has been successfully applied to several application domains, such as recommender systems (Bansal et al., 2016) and real-time object detection (Girshick, 2015). For instance, a movie recommendation system may optimize not only the likelihood of the user clicking on a suggested movie, but also the likelihood that the user is going to watch it.

The most common architecture used in practice for multi-task learning is the so-called *shared bottom*, where the tasks share parameters in the early layers of the model, which are followed by task-specific heads. However, as our experiments on synthetic data show, when the tasks are unrelated, parameter sharing may actually hurt individual tasks performance. Therefore, resorting to flexible parameter sharing becomes very important. This can be achieved by manually trying several different static sharing patterns. However, this option is not scalable, since it requires significant effort. Instead, an approach where the sharing pattern is learned and adapted to the task relatedness, is preferable.

At the same time, routing networks (Rosenbaum et al., 2018) have been introduced as powerful models, which route each input sample through its own path, selectively activating only parts of the network. They have shown strong performance in various settings thanks to their high flexibility. Routing networks lend themselves as a natural choice for learning sharing patterns in multi-task modeling. However, they are typically rather hard to train in practice (see e.g., (Rosenbaum et al., 2019) for the challenges of routing networks and references there in).

In this work, we introduce a novel method, that learns the sharing pattern jointly with the model parameters using standard back-propagation. Assuming that the network consists of several layers, where each layer consists of several components, the core idea of the proposed method is to learn, for each component, a set of binary allocation variables indicating which tasks use this component. We rely on the Gumbel-softmax reparameterization method (Jang et al., 2017) in order to train these binary variables jointly with the parameters of the components.

We provide experiments on synthetic data as well as real-world data showing that the proposed method can adapt the sharing pattern to the task relatedness, outperforming strong baselines and previous state-of-the-art methods. In summary, the paper contributions are the following:

- We analyze positive and negative transfer effects on synthetic scenarios. This analysis motivates the need for task dependent parameter sharing in multi-task networks.
- We propose a novel routing method based on Gumbel binary variables, that allows for learning flexible parameter sharing which adapts to the task relatedness, and can be optimized with standard back-propagation.

The source code implementing the proposed method and the benchmarks described in this paper is publicly available at `http://github.com/anonymous`.

2 POSITIVE AND NEGATIVE TRANSFER BETWEEN TASKS

We start with a practical example showing that besides positive transfer, negative transfer may as well occur in practice. That is, when the tasks are unrelated, allowing them to interact in a bigger model instead of training them separately harms the model performance. To show that both positive and negative transfer occurs, we generate two synthetic tasks, where the task relatedness ρ can be explicitly controlled. Our synthetic data generation process is based on that of (Ma et al., 2018), and we describe it in detail in Appendix A.1. We consider two edge cases: two unrelated tasks ($\rho = 0$), and two tasks that are the same up to noise ($\rho = 1$).

We create a simple multi-task network. This network architecture consists of 4 parallel components, and each component contains a stack of fully connected layers. Each input example can be provided as input to any subset of the 4 parallel components; the outputs of the components are averaged before being passed to a task-specific linear head. We chose this network to have low enough capacity, so that there is visible competition between tasks. For more information about the architecture, refer to Appendix A.2.

For this experiment, we use two hard-coded sharing patterns. The *'full sharing'* pattern means that both tasks use all components, while *'no sharing'* means that tasks use disjoint halves of all components. Therefore, in the 'no sharing' pattern, the tasks are completely independent. Note that regardless of the pattern, the total amount of parameters in the model remains the same; the only difference is in which parameters get used by which tasks. In other words, 'no sharing' corresponds to imposing a constraint that the network should be evenly divided between the tasks, while 'full sharing' leaves that to the optimization algorithm to decide.

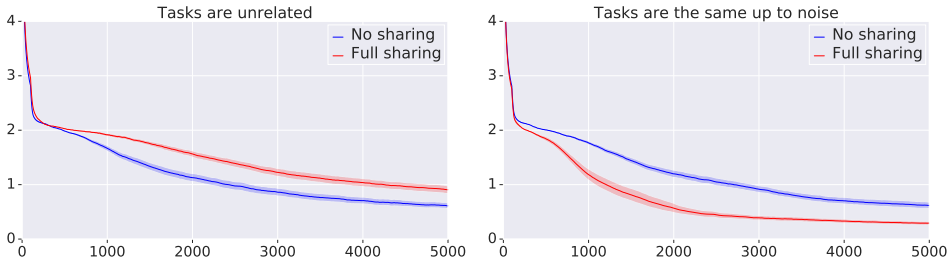

Figure 1: Comparison of the 'full sharing' and 'no sharing' patterns for the case of unrelated tasks (left) and almost equal tasks (right). The plots show loss over time (averaged over the two tasks and smoothed over a window of 100 steps). We ran each experiment 30 times, and the shaded area corresponds to the 90% confidence interval.

We run four experiments: one for every combination of sharing pattern ('full sharing' and 'no sharing'), and task relatedness ($\rho \in \{0, 1\}$). For each experiment, we report the L2 loss over time, averaged over the two tasks. The results are shown in Figure 1. Since for 'no sharing' there is no interaction between the tasks, the average loss behaves in the same way irrespective of the task relatedness. For 'full sharing', both tasks are allowed to update all parameters, and we see that while it improves performance if the tasks are related, it actually hurts for two completely unrelated tasks.

Motivated by this example, we argue that general multi-task models should be able to learn flexible sharing patterns that are able to adapt to the task relatedness. In Section 4 we introduce a framework which can flexibly learn parameter sharing patterns, including but not limited to 'no sharing' and

'full sharing'. Since our method is based on routing networks, we provide some background about them first.

3 ROUTING NETWORKS

Standard neural networks process every input example in the same way. Routing networks provide a more flexible alternative, where every input is passed through a subgraph of the entire model (also referred to as the *supernetwork*). Typically, the model is divided into layers, and some layers contain multiple parallel components (also called *modules* or *experts*). Inputs to the large network are *routed through* some sets of components in subsequent layers. This general framework was used by multiple recent works (Fernando et al., 2017; Rosenbaum et al., 2018; Ramachandran & Le, 2019), although with some small differences or additional constraints depending on the routing method.

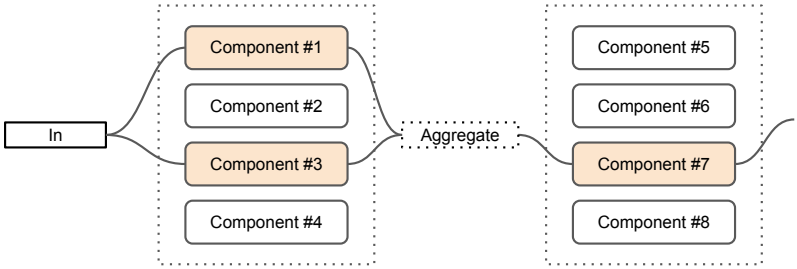

Figure 2: A general example of a routing model. The input is passed through two components in the first layer, and one component in the second layer.

In this paper, we use a relatively general view depicted in Figure 2, which is relevant to both single-task and multi-task setups. Note that we do not impose any constraints on the set of components used for a specific input; in particular, the number of components used can vary between layers and between individual inputs. After passing a sample through a single routed layer, the outputs of all activated components are aggregated to form an input to the next layer. In all experiments in this paper we simply use the average as the aggregation.

4 GUMBEL-MATRIX ROUTING FRAMEWORK

We design our framework so that it is able to learn to select task-conditioned subgraphs of a larger computational graph. In Figure 3, we show an example of a routing model, where the routing is conditioned on the task only. In that example, there are two tasks, and two routed layers. Each task selects some subset of components within every layer. For every routed layer, we encode the component assignment as a binary matrix. Concretely, if there are T tasks, and C components in a given layer, the matrix has shape T x C, where the entry in the i-th row and j-th column is 1 if the i-th task uses the j-th component. We show these matrices at the bottom of Figure 3. Our goal is to learn the routing matrices in the way that maximizes the average per-task performance of the model. In order to learn the routing matrices we condition on the task id, which implies that all samples from the same task will go through the same path in the network. We refer to our framework as the *Gumbel-matrix routing* framework. While at training time our method samples many different routing matrices, at the end a single matrix is selected per layer. At inference time, the matrices are fixed, and thus our routing does not add any overhead over the underlying non-routed model.

4.1 TRAINING

For each layer, we want to maintain a probability distribution over all possible binary routing matrices. To that end, we assume this distribution to be factorized, and we explicitly maintain a matrix of per-connection probabilities. Each probability is represented as a pair of two complementary logits.

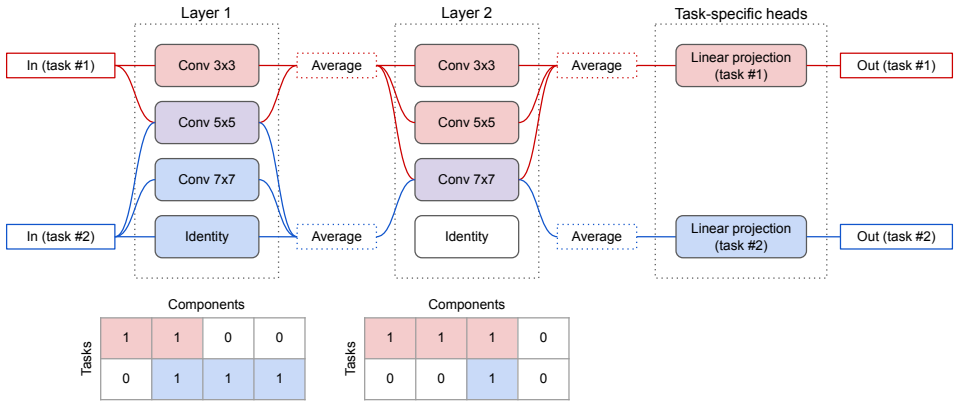

Figure 3: An example routing network with two tasks. Some components are used by both tasks (purple), some by only one of the tasks (red or blue, respectively), and one identity component is completely unused (white). Below each layer we show the corresponding routing matrix.

To perform a forward pass, we sample all binary connections independently according to the probability matrix. In principle, it is possible that for a given task, all connections are sampled to 0. In that case, the output of the routed layer would be a zero vector, independently of the input. In practice, we found that this happens very rarely, and mostly at the beginning of training, since usually one of the connection probabilities quickly becomes close to 1. Therefore, we did not try to devise a method which would artificially prevent an all-zeros connection pattern from being sampled.

To initialize our method, we have to set the connection probabilities to some initial values. In principle, it is possible to introduce prior knowledge, and set these probabilities in a way that encourages or discourages certain patterns. However, here we consider the most general approach, where all connection probabitlies are initialized to the same constant value p_{init}. Setting $p_{\text{init}} = 0.5$ gives the highest routing entropy, and corresponds to the weakest prior, therefore we considered it as a default choice. However, in our experiments, we also found that for routing in large and deep networks, it is beneficial to set p_{init} closer to 1, in order to enhance the trainability of the components and to stabilize the initial learning phases.

In the backwards pass, only the components that are activated will get gradients, as the inactive components do not contribute to the final output of the network. However, in order to get a gradient for the connection probabilities, we would have to backpropagate through sampling. In the next section, we describe a method which we use to accomplish that.

4.2 Gumbel-Softmax trick

In order to get gradients to the connections probabilites, we follow (Jang et al., 2017), and reparameterize sampling from a Bernoulli distribution by using the Gumbel-Softmax trick. The Gumbel distribution can be defined by the following forward sampling procedure:

$$u \sim \text{Uniform}(0, 1) \Rightarrow g = -\log(-\log(u)) \sim \text{Gumbel}.$$

Instead of using the logits to directly sample a binary value, we add independent noise from the Gumbel distribution to each of the logits, and then select the binary value with the highest logit (i.e. argmax) as the sample z. Formally, to sample from Bernoulli(p), we use the following procedure. Let $\pi = [p, 1 - p]$; we draw g_0 and g_1 from the Gumbel distribution, and produce the sample z as

$$z = \underset{i \in \{0,1\}}{\arg\max}\, v_i, \text{where } v := \log(\pi) + [g_0, g_1].$$

The argmax operation is not differentiable, but it can be approximated by a softmax with annealing temperature. Therefore, on the forward pass, we use the argmax to obtain a binary connection value, while on the backwards pass, we approximate it with softmax, similarly to (Guo et al., 2018). This approach is known as the *Straight-Through Gumbel-Softmax estimator* (Jang et al., 2017). Note that

the backwards pass actually requires all components to be evaluated, irrespective of whether they are used in the forward pass or not. Therefore, if a connection is sampled to be inactive, then the corresponding component will not get gradients, but its output will be used to compute the gradient for the connection probability.

4.3 INFERENCE

At inference time, it is possible to follow the same procedure as at training time, i.e. sample the connection pattern for every test batch. In our experiments we found that this works well, and does not introduce a large amount of noise in the evaluation result, since the connection probabilities naturally tend to converge to either 0 or 1 during training. An alternative approach is to fix the connections to their maximum likelihood variants, and use that pattern for every forward pass. We do that in the evaluation phase of all our experiments, since we believe this is closer to how the Gumbel-Matrix framework should be used in practice. Note that for the maximum likelihood approach, we can discard all connection probabilities after the training has completed. The probabilities are used only to describe how to select a subgraph of the network for each task.

4.4 BUDGET PENALTY

We have found that Gumbel-Matrix routing generally trains well in its vanilla form. However, we note some ways in which it is possible to alter its default behavior. For example, one might want to learn a routing pattern with a certain degree of sparsity. To that end, we introduce the *budget penalty*, which penalizes the model from exceeding a given computational budget. Let us assume we define the budget as a maximum percentage of active connections. Since we explicitly know all connection probabilities, by summing them up over all layers we obtain the *expected* number of connections e_c for a forward pass at a given point in time. Therefore, we can set a budget $b \in (0, 1\rangle$, corresponding to the maximum allowed fraction of active connections, and define the *budget auxiliary loss* as: $\lambda \max(0, e_c - b)$, where λ is a constant that controls the strength of the penalty. For a sufficiently large λ, this penalty can be viewed as a hard constraint in practice.

5 RELATED WORK

Traditionally, work on multi-task learning includes hand-designing the sharing pattern, in order to strike a good balance between shared and task-specific parameters. Our method is more related to works in the intersection of routing models and multi-task learning, where the sharing pattern is learned jointly with the model parameters. These works can be mainly divided based on the algorithm that is used to learn the routing.

Some methods cast learning the routing as a Reinforcement Learning problem. The authors in (Rosenbaum et al., 2018) propose a framework based on multi-agent Reinforcement Learning, where the positive-negative transfer problem is taken care of by finding a Nash equilibrium. In contrast, our work does not rely on Reinforcement Learning, and can be trained with standard back-propagation. A notion of task-specific paths is also present in (Fernando et al., 2017); these paths are learned using evolutionary algorithms.

Many other works use the Sparsely-Gated Mixture-of-Experts (Shazeer et al., 2017), initially developed for a single-task model. This idea is extended in (Ma et al., 2018) by introducing a separate gating function per task, and in (Ramachandran & Le, 2019) by using architecturally diverse experts and increasing the routing depth. However, (Ramachandran & Le, 2019) report that their models are often hard to train. In contrast, we show that our method can improve accuracy even when routing in large and deep neural networks.

Sub-Network Routing (Ma et al., 2019) proposes a routing method for multitask networks. This approach is not task-dependent, as it learns a routing that is applied to all the tasks. It is based on learned binary variables, that control the connections between modules, rather than controlling the activation of a module itself. Another related approach relies on cross-stitch models (Misra et al., 2016). It uses single-task models that are stitched together with cross-stitch units, which learn shared representations by linearly combining intermediate representations from the single task models.

Additionally, some works use concepts similar to the ones we use, but not for multi-task learning. In particular, Guo et al. (2018) also rely on the Gumbel Softmax trick to learn binary variables, with a focus on fine-tuning for transfer learning applications. The binary variables are used to decide which layers of a pre-trained model should be fine-tuned on the target task. Another related method (Bengio et al., 2016) learns binary variables that mask the outputs of each layer, conditioning on the activations of the previous layers. In order to learn these binary variables, the REINFORCE algorithm (Williams, 1992) is used. A similar notion of adaptive inference graphs has been proposed in (Veit & Belongie, 2019) that uses convolutional neural nets for image classification based on a ResNet-type of architecture, where some layers are skipped using learned gating functions. However, note that these methods have not been designed for multi-task learning.

Finally, our approach is related to methods for Neural Architecture Search (NAS) (Zoph & Le, 2017; Real et al., 2017), which automatically design neural network architectures for a given task. Our method is of similar spirit as the efficient NAS methods (Pham et al., 2018; Liu et al., 2019) in the sense that the architecture parameters are jointly optimized with the model parameters. Our method searches for a multi-task architecture that learns a flexible parameter sharing pattern according to the task relatedness, and uses a simpler architecture encoding based on binary variables.

6 EXPERIMENTS

In all our experiments, we impose an additional constraint that each input batch contains samples for only one task. Since the routing is conditioned on the task only, this allows us to sample the connection pattern once per forward pass. To train a network in a multi-task setting, we draw one batch of input samples per task, pass them through the network in random order, and repeat that process for a predefined number of steps. To start, we have tested our method on the same synthetic data as in Section 2. Due to lack of space, we provide the experimental results in Appendix A.3.

6.1 MNIST

Experimental setup To test our routing method in a controlled environment where we know which pairs of tasks are more related, we create the following *4-MNISTs setup* based on the MNIST dataset.

We first define the *MNIST-rot* task, by taking the input-output pairs of MNIST, and rotating all input images clockwise by 90 degrees. We run experiments on 4 tasks, where the first two tasks are copies of MNIST, and the next two are copies of MNIST-rot. Note that two copies of the same task have the same training and test datasets, but the order of batches is different. In order to make the setup difficult, we use a relatively small routed network. It consists of three routed layers, containing four components each. The components in the first routed layer are 5x5 convolutions, while in the second and third layers are 3x3 convolutions. After the last routed layer, the output feature map is flattened, and passed through a task-specific linear head. For more details, see Appendix B.

Results We first run two baselines, corresponding to the 'no sharing' and 'full sharing' patterns introduced in Section 2. In this case, 'no sharing' corresponds to the i-th of the four tasks using only the i-th component in every layer - again, this means that there is no interaction between tasks. 'Full sharing' means that all tasks use all components. We see that 'full sharing' strongly outperforms 'no sharing', which shows that this routed network is small even for MNIST, and using one component per layer is not enough to reliably learn the task.

Next, we train two variants of Gumbel-Matrix routing: one without any auxiliary penalties, and one with the budget constraint set to 0.75. We report all of these results in Table 1. We found that the two copies of MNIST end up using the same routing patterns, as well as the two are copies of MNIST-rot. However, patterns used by the copies of MNIST are different from the ones used by MNIST-rot. As seen in the results, this gives better performance, since the processing is task-dependent. Furthermore, we see that by using the budget penalty, we can reduce the number of active connection without sacrificing the test accuracy.

Table 1: Results on the 4-MNISTs multitask setup. Each experiment was run 30 times, we report mean and standard deviation of the error.

Method	Test accuracy (%)	Active connections (%)
No sharing	93.5 ± 5.1	25
Full sharing	95.7 ± 0.5	100
Gumbel-Matrix	$\mathbf{96.8 \pm 0.2}$	96
Gumbel-Matrix (budget = 0.75)	$\mathbf{96.7 \pm 0.3}$	75

6.2 OMNIGLOT

Next, we test our method on the Omniglot multi-task setup (Lake et al., 2015). The Omniglot dataset consists of 50 different alphabets, each containing some number of characters. Input samples for each of the characters are handwritten grayscale images of size 105×105.

Experimental setup We create our setup following Meyerson & Miikkulainen (2018), where each alphabet is treated as a separate task of predicting the character class. We follow previous works (Liang et al., 2018; Ramachandran & Le, 2019) and use a fixed random subset of 20 alphabets, splitting every alphabet into training/validation/test sets with proportions $50\%/20\%/30\%$. For completeness, we list these 20 alphabets in Appendix C.1.

In order to have a direct comparison with the state-of-the-art of Ramachandran & Le (2019), we use the same underlying network, optimizer, and regularization techniques, and only change the routing algorithm. We have reached out to the authors of (Ramachandran & Le, 2019) to obtain various details and make sure the setups match. We report the architecture here for completeness.

The network consists of: one shared 1x1 convolution, then 8 routed layers, and finally linear task specific heads. Each routed layer contains 7 different components: conv 3x3 $\rightarrow$ conv 3x3, conv 5x5 $\rightarrow$ conv 5x5, conv 7x7 $\rightarrow$ conv 7x7, conv 1x7 $\rightarrow$ conv 7x1, 3x3 max pooling, 3x3 average pooling, identity. The number of channels is 48 throughout the network. All components use padding to make sure the output shape is the same as the input shape; the spatial dimensions are reduced by adding a stride of 2 to 5 of the routed layers. We use GroupNorm (Wu & He, 2018) and ReLU after each convolution and after each routed layer.

We regularize the model with Dropout and L2-regularization. For training, we use the Adam optimizer. Since the routing logits are updated only once every T steps (where T is the number of tasks), we have found that for $T = 20$ it is beneficial to use a larger learning rate for the routing logits than for the components. Therefore, we set the learning rate for the routing logits to be T times larger than the one for the other weights, and found this rule of thumb to work well in practice. We set the training length to be larger than needed for the methods to attain their peak performance, select the best checkpoint for each method based on validation accuracy, and evaluate that single checkpoint on the test set. For hyperparameter values, as well as more details about the network, see Appendix C.2.

Results Before training a model based on Gumbel-Matrix routing, we train a 'full sharing' variant, where all tasks use all components. We do not evaluate a 'no sharing' variant, since the number of tasks T is larger than the number of components per layer. Then, we train a routing model, where we use Gumbel-Matrix routing to model the connections in each routed layer. We show the results of the experiments in Table 2.

We see that the underlying non-routed model actually outperforms the Mixture-of-Experts routing of Ramachandran & Le (2019), which we found to be very surprising. We conjecture that even though the Mixture-of-Experts variant is more powerful, in the case of multi-task learning on Omniglot optimization difficulties outweigh that benefit. In contrast to our method, the Mixture-of-Experts framework hard-codes the required sparsity for each layer. This can bring immense computational savings (Shazeer et al., 2017), but may also sacrifice accuracy. In some cases, like the one of Shazeer et al. (2017), the 'full sharing' variant would be prohibitively expensive to run, making the comparison infeasible. However, we encourage routing models researchers to compare their routed networks with their 'full sharing' counterparts whenever possible.

Table 2: Results on multitask Omniglot setup. Each experiment was run 10 times, we report mean and standard deviation of the error.

Method	Valid. error (%)	Test error (%)
Single Task (Meyerson & Miikkulainen, 2018)	36.41 ± 0.53	39.19 ± 0.50
Soft Ordering (Meyerson & Miikkulainen, 2018)	32.33 ± 0.74	33.41 ± 0.71
CMTR (Liang et al., 2018)	11.80 ± 1.02	12.81 ± 1.02
MoE (Ramachandran & Le, 2019)	7.95 ± 0.37	7.81 ± 0.54
Full sharing	6.16 ± 0.50	6.75 ± 0.33
Gumbel-Matrix	$\mathbf{5.69 \pm 0.22}$	$\mathbf{6.48 \pm 0.28}$

Next, we train a routed model based on Gumbel-Matrix routing. We do not use any auxiliary losses, and find that the model naturally removes some of the connections to allow for task-specific processing. Even though the network is not explicitly penalized for high routing entropy, connection probabilities still converge to be either close to 0 or close to 1. We report the resulting accuracy in Table 2. We see that the Gumbel-Matrix routing improves the accuracy over a very strong 'full sharing' baseline.

Analysis In order to analyze the routing patterns learned by our network, we compute several statistics. First, for each task, we concatenate the routing matrices from all routed layers to form a larger matrix, which we call a *global pattern*. If two tasks have exactly the same global pattern, the network will process them in the same way (up to the task-specific linear heads). We find that, on average, the model converges to having 10.1 different global patterns. That means that tasks are clustered into groups that are processed in the same way, and the average size of such group is approximately 2.

Next, we analyze what kinds of components are used at every layer in the network. Recall that each layer in the model consists of 7 components, out of which 4 are different kinds of convolutions, 2 are pooling (max and average), and the last one is identity, which can be though of as a skip connection. We found that the usage trends are consistent across experimental runs, and mostly depend on the layer in question. The trends are the following (please see Appendix C.3 for relevant illustrations):

- In layers $1 - 4$, we found that the connections to all components are being dropped at a similar rate; most of the diversity in processing of different tasks concentrates in these layers.
- In layers $5 - 7$, almost all components are always used, and the only source of differences between tasks is that some of them use the average pooling component, while others don't.
- In layer 8, the average pooling component is never used, and the other components are always used.

Our interpretation of this result is that the Gumbel-Matrix routing shows a mixture of two behaviors. First, the connections can be dropped in order to obtain task-specific processing; this trend is visible in the first layers of the model. Second, all connections to specific components can be removed, which corresponds to a form of network pruning. We speculate that the task-specific processing in the initial layers 'aligns' the feature spaces of samples coming from different tasks. In the final layers, the tasks are processed in a uniform way, but some components are pruned to improve the model quality.

7 Conclusions and future work

We introduced a new method for multi-task learning that learns the pattern of parameter sharing together with the model parameters using standard back-propagation. We provided experimental results showing that our method can learn flexible sharing patterns, and adapt to the task relatedness, which results in significantly improved performances over the state-of-the-art. Right now, the routing decision in our method is conditioned only on the task id. Conditioning on richer information such as a task embedding will be the focus of future work.

REFERENCES

T. Bansal, D. Belanger, and A. McCallum. Ask the GRU: Multi-task learning for deep text recommendations. In *10th ACM Conference on Recommender Systems*, pp. 107–114. ACM, 2016.

E. Bengio, P.-L. Bacon, Joelle Pineau, and D. Precup. Conditional computation in neural networks for faster models. *arXiv preprint arXiv:1511.06297*, 2016.

R. Caruana. Multitask learning: A knowledge-based source of inductive bias. *In Machine Learning: Proceedings of the Tenth International Conference*, pp. 41–48, 1993.

R. Caruana. Multitask learning. *Learning to learn*, 1998.

C. Fernando, D. Banarse, C. Blundell, Y. Zwols, D. Ha, A. A. Rusu, A. Pritzel, and D. Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017.

R. Girshick. Fast R-CNN. In *International Conference on Computer Vision (ICCV)*, pp. 1440–1448. IEEE, 2015.

Y. Guo, H. Shi, A. Kumar, K. Grauman, T. Rosing, and R. Feris. SpotTune: Transfer Learning through Adaptive Fine-tuning. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. URL https://arxiv.org/abs/1811.08737.

E. Jang, S. Gu, and B. Poole. Categorical reparametrization with Gumbel-softmax. *International Conference on Learning Representations (ICLR)*, 2017.

M. B. Lake, R. Salakhutdinov, and J. B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350:13321338, 2015.

J. Liang, E. Meyerson, and R. Miikkulainen. Evolutionary architecture search for deep multitask networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 466–473. ACM, 2018.

H. Liu, K. Simonyan, and Y. Yang. DARTS: Differentiable architecture search. In *International Conference on Learning Representations (ICLR)*, 2019.

J. Ma, Z. Zhao, Yi X., J. Chen, L. Hong, and E. Chi. Modeling Task Relationships in Multi-task Learning with Multi-gate Mixture-of-Experts. *KDD*, 2018.

J. Ma, Z. Zhao, J. Chen, A. Li, L. Hong, and E. Chi. SNR: Sub-Network Routing for Flexible Parameter Sharing in Multi-task Learning. *AAAI Conference on Artificial Intelligence*, 2019. URL http://www.jiaqima.com/papers/SNR.pdf.

E. Meyerson and Risto Miikkulainen. Beyond shared hierarchies: Deep multitask learning through soft layer ordering. *International Conference on Learning Representations (ICLR)*, 2018.

I. Misra, A. Shrivastava, A. Gupta, , and M. Hebert. Cross-stitch networks for multi-task learning. In *Conference on Computer Vision and Pattern Recognition*, pp. 3994–4003. IEEE, 2016.

H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. Efficient neural architecture search via parameters sharing. *International Conference on Machine Learning (ICML)*, 2018.

P. Ramachandran and Q. V. Le. Diversity and Depth in Per-Example Routing Models. *International Conference on Learning Representations (ICLR)*, 2019. URL https://openreview.net/pdf?id=BkxWJnC9tX.

E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin. Large-Scale Evolution of Image Classifiers. *International Conference on Machine Learning (ICML)*, 2017.

C. Rosenbaum, T. Klinger, and M. Riemer. Routing Networks: Adaptive Selection of Non-Linear Functions for Multi-Task Learning. *International Conference on Learning Representations (ICLR)*, 2018. URL https://openreview.net/pdf?id=ry8dvM-R-.

C. Rosenbaum, I. Cases, M. Riemer, and T. Klinger. Routing networks and the challenges of modular and compositional computation. *arXiv preprint arXiv:1904.12774*, 2019.

N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

A. Veit and S. Belongie. Convolutional Networks with Adaptive Inference Graphs. In *International Journal of Computer Vision (IJCV)*, pp. 1–12, 2019.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Machine Learning*, volume 8(3–4), pp. 229–256, 1992.

Y. Wu and K. He. Group normalization. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 3–19, 2018.

B. Zoph and Q. V. Le. Neural Architecture Search with Reinforcement Learning. *International Conference on Learning Representations (ICLR)*, 2017.

A SYNTHETIC DATA

A.1 GENERATION

We follow a process for synthetic data generation based on (Ma et al., 2018), which allows for explicit control of the relatedness between tasks. In particular, we generate the data as follows:

1. We generate two orthogonal unit vectors $u_1, u_2 \in R^d$. That is, u_1 and u_2 satisfy $u_1^\top u_2 = 0$ and $\|u_1\| = \|u_2\| = 1$.
2. Given a desired score of task relatedness $0 \leq \rho \leq 1$, we generate two weight vectors w_1, w_2 such that
$$w_1 = cu_1, w_2 = c(\rho u_1 + \sqrt{1-\rho^2}u_2) \tag{1}$$
3. We randomly sample a data point $x \in R^d$ where each dimension is sampled from $\mathcal{N}(0, 1)$.
4. Generate two labels y_1, y_2 for the two tasks as follows:
$$y_1 = w_1^\top x + \sum_{i=1}^{m} \sin(\alpha_i w_1^\top x + \beta_i) + \epsilon_1 \tag{2}$$
$$y_2 = w_1^\top x + \sum_{i=1}^{m} \sin(\alpha_i w_2^\top x + \beta_i) + \epsilon_2 \tag{3}$$
where $\epsilon_1, \epsilon_2 \sim \mathcal{N}(0, 0.01)$.

In the procedure above, the values of d, c, m, and the sequences α_i, β_i are hyperparameters. For all of our experiments with synthetic data, we set $d = 128$, $c = 1$, $m = 6$, $\alpha_i = i$, $\beta_i = (i-1)^2$. We found m to be the most important of these hyperparameters, as it controls the non-linearity (and hence the difficulty) of the tasks. In particular, for $m = 0$ the input-output relation for the produced task will be linear (up to noise).

A.2 ARCHITECTURE FOR THE 'POSITIVE AND NEGATIVE TRANSFER' EXPERIMENT

Here we describe in detail the network used for the experiment in Section 2.

The network starts with a layer containing 4 parallel fully connected components. Two of these components components consist of 1 fully connected layer, and the other two consist of 2 layers with 4 hidden units. All components have 4 output units, and use ReLU activations on the hidden and output units. Note that a specific input can be passed through multiple components; the outputs of the components are averaged before being passed to a task-specific linear head. We present this architecture in Figure 4.

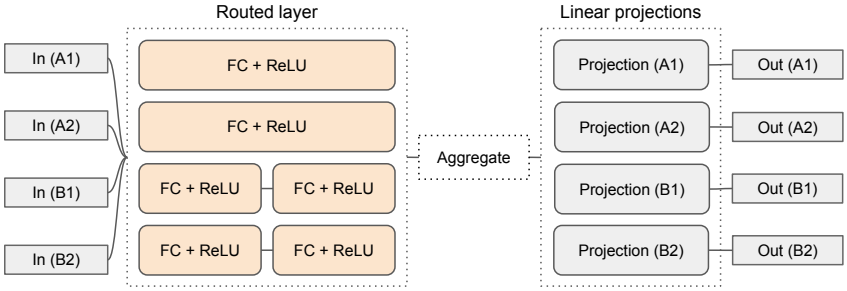

Figure 4: Multi-task model architecture for the experiment on the synthetic data. We denote fully connected layers by FC, and name the two pairs of tasks A1, A2 and B1, B2, respectively.

A.3 GUMBEL-MATRIX EXPERIMENT

We additionally test the Gumbel-Matrix routing on synthetic data using a setup that emphasizes the routing pattern. To that end, we create a routed network that does not contain task specific heads. Instead, it consists of a routed layer with 4 components, each with 1 output unit. The components are relatively large for this kind of simple data: each is 3 layers deep, with 16 hidden units in every hidden layer, ReLU activations on the hidden units, and no activation on the output units.

We use batch size of 64, learning rate 0.01, and we clip the gradient norm to 1.0.

We generate two pairs of tasks according to A.1, so that each pair contains tasks that are the same up to noise, but tasks in different pairs are unrelated. Since we use a network with no task-specific heads, if two tasks get routed through the same set of components, the network will implement the same function for these tasks. Therefore, if the routing is incorrect (i.e. routes two tasks from different pairs through the same subset of components), the rest of the network cannot make up for it, making it a good sanity check for our method.

Note that in this network, each component is capable of learning the synthetic task on its own, and using a 'no sharing' pattern is a solid baseline. However, routing should still be able to improve performance: if it discovers the related pairs of tasks, they can be routed through a shared set of components, and parameters in these components would get twice as much training data compared to the 'no sharing' case.

We train for 3000 batches per task, and track the average loss over tasks. The result is shown in Figure 5. We found that the related tasks converge to using the same subset of components, but the subsets used by the unrelated tasks are distinct. Because of that, the routed model outperforms the 'no sharing' variant. In this setup, the 'full sharing' setup works very poorly, since it is unable to learn both pairs of tasks at the same time. Hence, we omit the results for the 'full sharing' baseline.

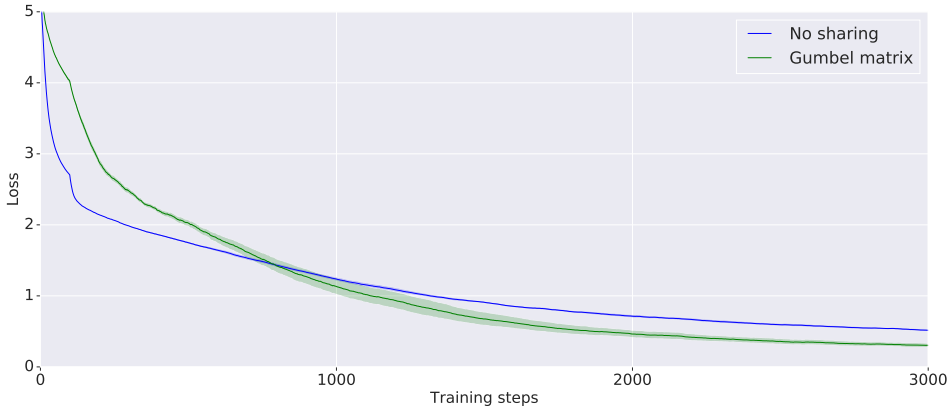

Figure 5: Comparison of the 'no sharing' pattern with the learned Gumbel-Matrix routing on the synthetic data experiment. The plot shows loss over time (averaged over the four tasks and smoothed over a window of 100 steps). We ran each experiment 20 times, and the shaded area corresponds to the 90% confidence interval.

B ARCHITECTURE FOR THE MULTI-TASK MNIST EXPERIMENT

Here we report additional details on the network used for the MNIST experiments.

As stated before, the network consists of three routed layers, containing convolutions with kernels 5x5, 3x3 and 3x3, respectively. We do not use padding in any of the convolutions, which means spatial dimensions are slightly reduced in each routed layer. After the first and second routed layer, we further reduce the spatial dimensions by applying 2x2 average pooling. All convolutions throughout the network have 4 filters.

During training, we regularize the model with Dropout (dropout probability 0.5, applied right before the tasks-specific heads). We set the training length to 10 epochs per task, and use a batch size of 16. We present this network in Figure 6.

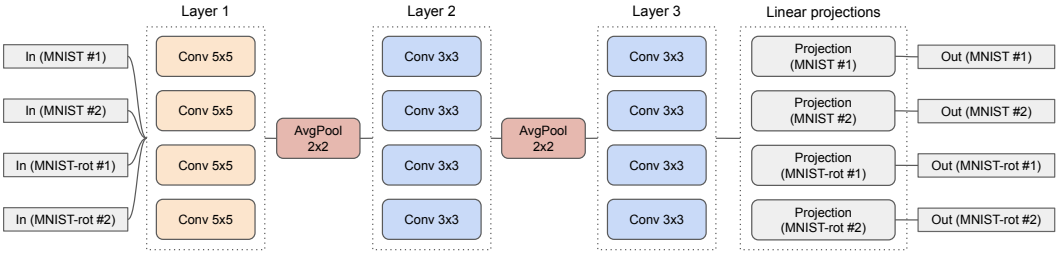

Figure 6: MNIST multi-task network.

C OMNIGLOT

C.1 ALPHABETS

In our Omniglot experiments, we followed prior works (Liang et al., 2018; Ramachandran & Le, 2019) and used a fixed set of 20 (out of 50) Omniglot alphabets. These alphabets are the first 20 in the order originally reported by (Meyerson & Miikkulainen, 2018). The names of these alphabets are the following: Gujarati, Sylheti, Arcadian, Tibetan, Old Church Slavonic (Cyrillic), Angelic, Malay (Jawi-Arabic), Sanskrit, Cyrillic, Anglo-Saxon Futhorc, Syriac (Estrangelo), Ge'ez, Japanese (katakana), Keble, Manipuri, Alphabet of the Magi, Gurmukhi, Korean, Early Aramaic, Atemayar Qelisayer.

C.2 ARCHITECTURE AND HYPERPARAMETERS

For the Omniglot experiments, we use a large convolutional network, developed by the authors of Diversity and Depth in Per-Example Routing Models (Ramachandran & Le, 2019).

One detail that we copy from Ramachandran & Le (2019) is that the identity component, in the case when it belongs to a routed layer with a stride larger than 1, is actually implemented as a strided 1x1 convolution. While it makes the name 'identity component' slightly inappropriate, we follow this to be directly comparable.

We tuned the network hyperparameters using the validation set, and used the following values for all of our Omniglot experiments. Dropout probability is 0.5, L2-regularization strength 0.0003, and the learning rate is set to 0.0001. We train with a batch size of 16.

For the Gumbel-Matrix routing, we set $p_{init} = 0.97$. We found that increasing p_{init} does slightly increase the average number of active connections at convergence, however, even with initialization as high as 0.97, the model still confidently removes many connections. On the other hand, we found $p_{init} = 0.97$ trains much better than $p_{init} = 0.5$. We believe that in a deep routed network with many components in every layer setting p_{init} to 0.5 introduces too much noise during training.

We present the overview of the architecture in Figure 7. Additionally, in Figure 8, we show the components within a single routed layer.

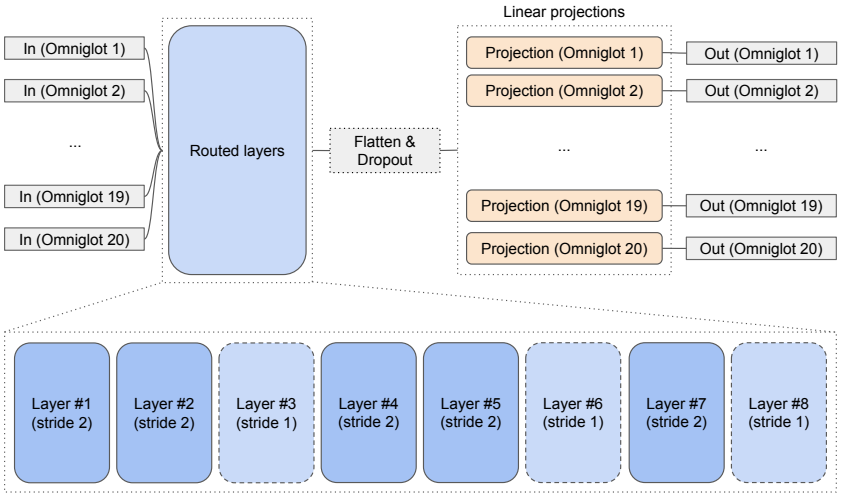

Figure 7: Omniglot multi-task network.

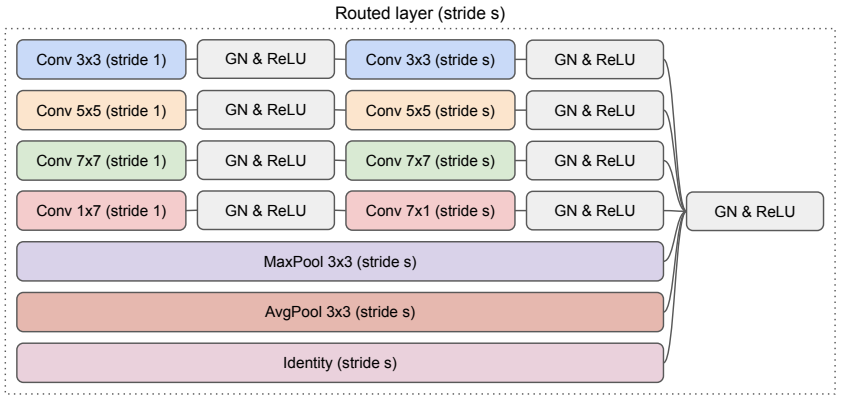

Figure 8: Components inside a routed layer in the Omniglot multi-task network. We denote GroupNorm by GN, and the layer stride as s. Note that for this specific architecture we have $s \in \{1, 2\}$.

C.3 ADDITIONAL ANALYSIS

As we have discussed in Subsection 6.2, the connection to the average pooling component is being dropped most often, while other connections are only dropped in the early layers.

Figure 9 illustrates these trends and shows how the connection probabilities are evolving over time for a certain task (the 'Syriac' alphabet). Note that for the experiments in Subsection 6.2 we ran 10 experiments for every method; here, for ease of presentation, we only show one of the runs. We see for example that for this specific task, the connection to the 3×3 convolutional component gets dropped in the first layer, while connections to average pooling are dropped in layers 3, 6 and 8. In the remaining three layers, the task uses all components.

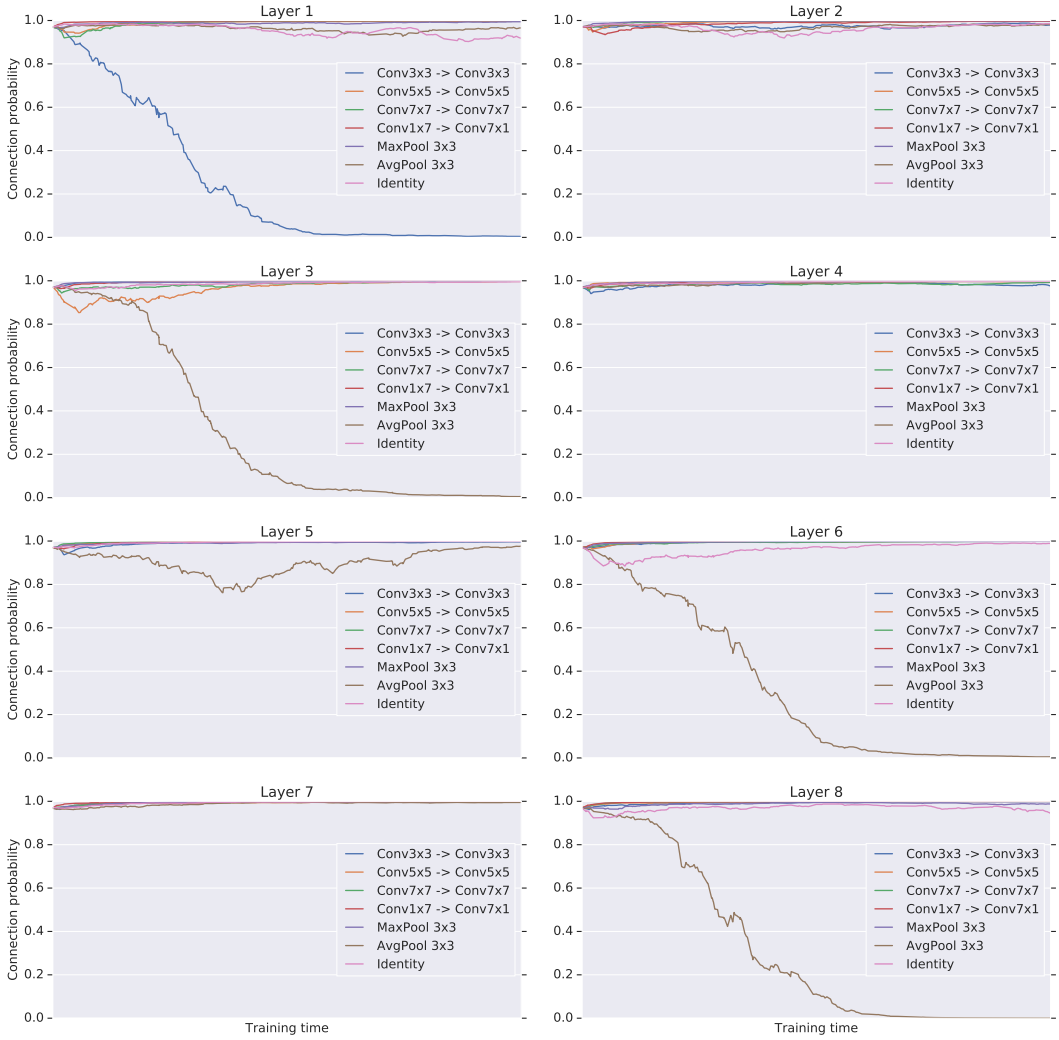

Figure 9: Connection probabilities over time (for one run, and one task). Each plot above shows the connection probabilities for all components of a certain routed layer in the network.