Modular Networks: Learning to Decompose Neural Computation

Anonymous Author(s)
Affiliation
Address
email

Abstract

Scaling model capacity has been vital in the success of deep learning. For a typical network, necessary compute resources and training time grow dramatically with model size. Conditional computation is a promising way to increase the number of parameters with a relatively small increase in resources. We propose a training algorithm that flexibly chooses neural modules based on the data to be processed. Both the decomposition and modules are learned end-to-end. In contrast to existing approaches, training does not rely on regularization to enforce diversity in module use. We apply modular networks both to image recognition and language modeling tasks, where we achieve superior performance compared to several baselines. Introspection reveals that modules specialize to interpretable contexts.

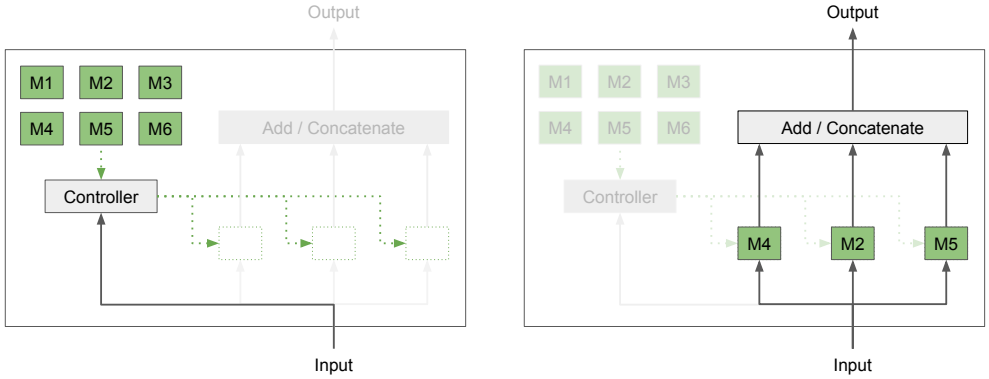
1 Introduction

When enough data and training time is available, increasing the number of network parameters typically improves prediction accuracy [14, 13, 1]. While the largest artificial neural networks currently only have a few billion parameters [8], the usefulness of much larger scales is suggested by the fact that human brain has evolved to have an estimated 150 trillion synapses [16] under tight energy constraints. In deep learning, typically all parts of a network need to be executed for every data input. Unfortunately, scaling such architectures results in a roughly quadratic explosion in training time as both more iterations are needed and the cost per sample grows. In contrast, usually only few regions of the brain are highly active simultaneously [17]. Furthermore, the modular structure of biological neural connections [22] is hypothesized to optimize energy cost [7], improve adaption to changing environments and mitigate catastrophic forgetting [21].

Inspired by these observations, we propose a novel way of training neural networks by automatically decomposing the functionality needed for solving a given task (or set of tasks) into reusable modules. We treat the choice of module as a latent variable in a probabilistic model and learn both the decomposition and module parameters end-to-end by maximizing a variational lower bound of the likelihood. Existing approaches for conditional computation [20, 2, 18] rely on regularization to avoid a module collapse (the network only uses a few modules repeatedly) that would result in poor performance. In contrast, our algorithm learns to use a variety of modules without such a modification and we show that training is less noisy compared to baselines.

A small fixed number out of a larger set of modules is selected to process a given input, and only the gradients for these modules need to be calculated during backpropagation. Different from approaches based on mixture-of-experts, our method results in fully deterministic module choices enabling low computational costs. Because the pool of available modules is shared between processing stages (or time steps), modules can be used at multiple locations in the network. Therefore, the algorithm can learn to share parameters dynamically depending on the context.

Submitted to 32nd Conference on Neural Information Processing Systems (NIPS 2018). Do not distribute.



(a) Based on the input, the controller selects K modules from a set of M available modules. In this example, $K = 3$ and $M = 6$.

(b) The selected modules then each process the input, with the results being summed up or concatenated to form the final output of the modular layer.

Figure 1: Architecture of the modular layer. Continuous arrows represent data flow, while dotted arrows represent flow of modules.

2 Modular Networks

The network is composed of functions (modules) that can be combined to solve a given task. Each module f_i for $i \in \{1, \ldots, M\}$ is a function $f_i(x; \theta_i)$ that takes a vector input x and returns a vector output, where θ_i denotes the parameters of module i. A modular layer, as illustrated in Figure 1, determines which modules (based on the input to the layer) are executed. The output of the layer concatenates (or sums) the values of the selected modules. The output of this layer can then be fed into a subsequent modular layer. The layer can be placed anywhere in a neural network.

More fully, each modular layer $l \in \{1, \ldots, L\}$ is defined by the set of M available modules and a controller which determines which K from the M modules will be used. The random variable $a^{(l)}$ denotes the chosen module indices $a^{(l)} \in \{1, \ldots, M\}^K$. The controller distribution of layer l, $p(a^{(l)}|x^{(l)}, \phi^{(l)})$ is parameterized by $\phi^{(l)}$ and depends on the input to the layer $x^{(l)}$ (which might be the output of a preceding layer).

While a variety of approaches could be used to calculate the output $y^{(l)}$, we used concatenation and summation in our experiments. In the latter case, we obtain

$$y^{(l)} = \sum_{i=1}^{K} f_{a_i^{(l)}}\left(x^{(l)}; \theta_{a_i^{(l)}}\right) \tag{1}$$

Depending on the architecture, this can then form the input to a subsequent modular layer $l + 1$. The module selections at all layers can be combined to a single joint distribution given by

$$p(a|x, \phi) = \prod_{l=1}^{L} p(a^{(l)}|x^{(l)}, \phi^{(l)}) \tag{2}$$

The entire neural network, conditioned on the composition of modules a, can be used for the parameterization of a distribution over the final network output $y \sim p(y|x, a, \theta)$. For example, the final module might define a Gaussian distribution $\mathcal{N}\left(y|\mu, \sigma^2\right)$ as the output of the network whose mean and variance are determined by the final layer module. This defines a joint distribution over output y and module selection a

$$p(y, a|x, \theta, \phi) = p(y|x, a, \theta)p(a|x, \phi) \tag{3}$$

Since the selection of modules is stochastic we treat a as a latent variable, giving the marginal output

$$p(y|x, \theta, \phi) = \sum_{a} p(y|x, a, \theta)p(a|x, \phi) \tag{4}$$

Selecting K modules at each of the L layers means that the number of states of a is M^{KL}. For all but a small number of modules and layers, this summation is intractable and approximations are required.

Algorithm 1 Training modular networks with generalized EM

Given dataset $D = \{(x_n, y_n) \,|n = 1, \ldots, N\}$
Initialize a_n^* for all $n = 1, \ldots, N$ by sampling uniformly from all possible module compositions
repeat
  Sample mini-batch of datapoint indices $I \subseteq \{1, \ldots, N\}$ ▷ Partial E-step
  for each $n \in I$ **do**
    Sample module compositions $\tilde{A} = \{\tilde{a}_s \sim p(a_n|x_n, \phi)|s = 1, \ldots, S\}$
    Update a_n^* to best value out of $\tilde{A} \cup \{a_n^*\}$ according to Equation 11
  end for
  repeat k times ▷ Partial M-step
    Sample mini-batch from dataset $B \subseteq D$
    Update θ and ϕ with gradient step according to Equation 8 on mini-batch B
until convergence

2.1 Learning Modular Networks

From a probabilistic modeling perspective the natural training objective is maximum likelihood. Given a collection of input-output training data $(x_n, y_n), n = 1, \ldots, N$, we seek to adjust the module parameters θ and controller parameters ϕ to maximize the log likelihood:

$$\mathcal{L}(\theta, \phi) = \sum_{n=1}^{N} \log p(y_n|x_n, \theta, \phi) \tag{5}$$

To address the difficulties in forming the exact summation over the states of a we use generalized Expectation-Maximisation (EM) [15], here written for a single datapoint

$$\log p(y|x, \theta, \phi) \geq -\sum_a q(a) \log q(a) + \sum_a q(a) \log \left(p(y|x, a, \theta)p(a|x, \phi)\right) \equiv L(q, \theta, \phi) \tag{6}$$

where $q(a)$ is a variational distribution used to tighten the lower bound L on the likelihood. We can more compactly write

$$L(q, \theta, \phi) = \mathbb{E}_{q(a)}[\log p(y, a|x, \theta, \phi)] + \mathbb{H}[q] \tag{7}$$

where $\mathbb{H}[q]$ is the entropy of the distribution q. We then seek to adjust q, θ, ϕ to maximize L. The partial M-step on (θ, ϕ) is defined by taking multiple gradient ascent steps, where the gradient is

$$\nabla_{\theta,\phi} L(q, \theta, \phi) = \nabla_{\theta,\phi} \, \mathbb{E}_{q(a)}[\log p(y, a|x, \theta, \phi)] \tag{8}$$

In practice we randomly select a mini-batch of datapoints at each iteration. Evaluating this gradient exactly requires a summation over all possible values of a. We experimented with different strategies to avoid this and found that the Viterbi EM [15] approach is particularly effective in which $q(a)$ is constrained to the form

$$q(a) = \delta(a, a^*) \tag{9}$$

where $\delta(x, y)$ is the Kronecker delta function which is 1 if $x = y$ and 0 otherwise. A full E-step would now update a^* to

$$a^*_{\text{new}} = \operatorname*{argmax}_a p(y|x, a, \theta)p(a|x, \phi) \tag{10}$$

for all datapoints. For tractability we instead make the E-step partial in two ways: Firstly, we choose the best from only S samples $\tilde{a}_s \sim p(a|x, \phi)$ for $s \in \{1, ..., S\}$ or keep the previous a^* if none of these are better (thereby making sure that L does not decrease):

$$a^*_{\text{new}} = \operatorname*{argmax}_{a \in \{\tilde{a}_s | s \in \{1,\ldots,S\}\} \cup \{a^*\}} p(y|x, a, \theta)p(a|x, \phi) \tag{11}$$

Secondly, we apply this update only for a mini-batch, while keeping the a^* associated with all other datapoints constant.

The overall stochastic generalized EM approach is summarized in Algorithm 1. Intuitively, the algorithm clusters similar inputs, assigning them to the same module. We begin with an arbitrary

assignment of modules to each datapoint. In each partial E-step we use the controller $p(a|x, \phi)$ as a guide to reassign modules to each datapoint. Because this controller is a smooth function approximator, similar inputs are assigned to similar modules. In each partial M-step the module parameters θ are adjusted to learn the functionality required by the respective datapoints assigned to them. Furthermore, by optimizing the parameters ϕ we train the controller to predict the current optimal module selection a_n^* for each datapoint.

Figure 2 visualizes the above clustering process for a simple feed-forward neural network composed of 6 modular layers with $K = 1$ modules being selected at each layer out of a possible $M = 3$ modules. The task is image classification, see Section 3.3. Each node in the graph represents a module and each datapoint uses a path of modules starting from layer 1 and ending in layer 6. The width of the edge between two nodes n_1 and n_2 represents the number of datapoints that use first module n_1 followed by n_2; the size of a node represents how many times that module was used. Figure 2 shows how a subset of datapoints starting with a fairly uniform distribution over all paths ends up being clustered to a single common path. The upper and lower graphs correspond to two different subsets of the datapoints. We visualized only two clusters, but in general many such clusters (paths) form, each for a different subset of datapoints.

2.2 Alternative Training

Related work [20, 3, 18] uses two different training approaches that can also be applied to our modular architecture. REINFORCE [23] maximizes the lower bound

$$B(\theta, \phi) \equiv \sum_a p(a|x, \phi) \log p(y|x, a, \theta) \leq \mathcal{L}(\theta, \phi) \tag{12}$$

on the log likelihood $\mathcal{L}$. Using the log-trick we obtain the gradients

$$\nabla_\phi B(\theta, \phi) = \mathbb{E}_{p(a|x,\phi)}[\log p(y|x, a, \theta) \nabla_\phi \log p(a|x, \phi)] \tag{13}$$

$$\nabla_\theta B(\theta, \phi) = \mathbb{E}_{p(a|x,\phi)}[\nabla_\theta \log p(y|x, a, \theta)] \tag{14}$$

These expectations are then approximated by sampling from $p(a|x, \phi)$. An alternative training algorithm is the noisy top-k mixture of experts [20]. A mixture of experts is the weighted sum of several parameterized functions and therefore also separates functionality into multiple components. A gating network is used to predict the weight for each expert. Noise is added to the output of this gating network before setting all but the maximum k units to $-\infty$, effectively disabling these experts. Only these k modules are then evaluated and gradients backpropagated. We discuss issues with these training techniques in the next section.

2.3 Avoiding Module Collapse

Related work [20, 3, 18] suffered from the problem of missing module diversity ("module collapse"), with only a few modules actually realized. This premature selection of only a few modules has often been attributed to a self-reinforcing effect where favored modules are trained more rapidly, further increasing the gap [20]. To counter this effect, previous studies introduced regularizers to encourage different modules being used for different datapoints within a mini-batch. In contrast to these approaches, no regularization is needed in our method. However, to avoid module collapse, we must take sufficient gradient steps within the partial M-step to optimize both the module parameters θ, as well as the controller parameters ϕ. That is, between each E-step, there are many gradient updates for both θ and ϕ. Note that this form of training is critical, not just to prevent module collapse, but to obtain a high likelihood. When module collapse occurs, the resulting log-likelihood is *lower* than the log-likelihood of the non-module-collapsed trained model. In other words, our approach is not a regularizer that simply biases the model towards a desired form of sub-optimal minimum – it is a critical component of the algorithm to ensure finding a high-valued optimum.

3 Experiments

To investigate how modules specialize during training, we first consider a simple toy regression problem. We then apply our modular networks to language modeling and image classification. Alternative training methods for our modular networks are noisy top-k gating [20], as well as

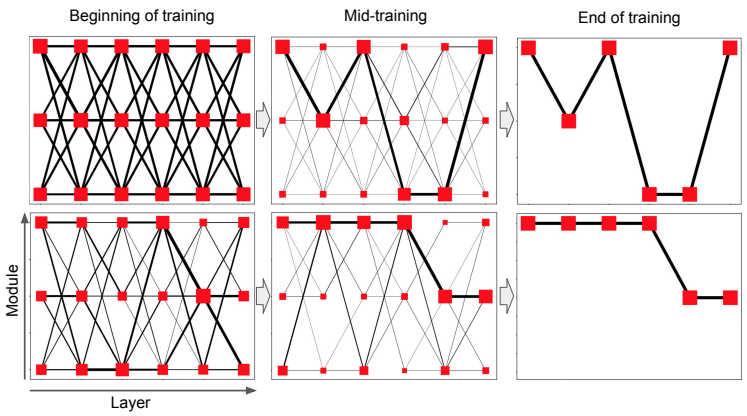


Figure 2: Two different subsets of datapoints (top and bottom) that use the same modules at the end of training (right) start with entirely different modules (left) and slowly cluster together over the course of training (left to right). Nodes in the graph represent modules with their size proportional to the number of datapoints that use this module. Edges between nodes n_1 and n_2 and their stroke width represent how many datapoints first use module n_1 followed by n_2.

REINFORCE [3, 18] to which we will compare our approach. Except if noted otherwise, we use a controller consisting of a linear transformation followed by a softmax function for each of the K modules to select. Our modules are either linear transformations or convolutions, followed by a ReLU activation. Additional experimental details are given in the supplementary material.

In order to analyze what kind of modules are being used we define two entropy measures. The module selection entropy is defined as

$$H_a = \frac{1}{BL} \sum_{l=1}^{L} \sum_{n=1}^{B} \mathbb{H} \left[p(a_n^{(l)} | x_n, \phi) \right] \tag{15}$$

where B is the size of the batch. H_a has larger values for more uncertainty for each sample n. We would like to minimize H_a (so we have high certainty in the module being selected for a datapoint x_n). Secondly, we define the entropy over the entire batch

$$H_b = \frac{1}{L} \sum_{l=1}^{L} \mathbb{H} \left[\frac{1}{B} \sum_{n=1}^{B} p(a_n^{(l)} | x_n, \phi) \right] \tag{16}$$

Module collapse would correspond to a low H_b. Ideally, we would like to have large H_b so that different modules will be used, depending on the input x_n.

3.1 Toy Regression Problem

To demonstrate the ability of our algorithm to learn conditional execution, we constructed a simple toy regression problem with artificially generated data. Our modular networks successfully decompose the problem into modules yielding perfect training and generalization performance. See supplement.

3.2 Language Modeling

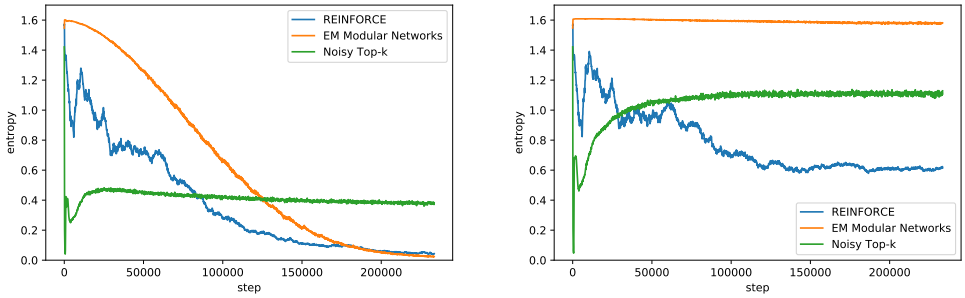
Modular layers can readily be used to update the state within an RNN. This allows us to model sequence-to-sequence tasks with a single RNN which learns to select modules based on the context. For our experiments, we use a modified Gated Recurrent Unit [5] in which the state update operation is a modular layer. Therefore, K modules are selected and applied at each time step. Full details can be found in the supplement.

We use the Penn Treebank[1] dataset, consisting of 0.9 million words with a vocabulary size of 10,000. The input of the recurrent network for each timestep is a jointly-trained embedding vector of size 32 that is associated with each word.

[1] `http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz`

Table 1: Test perplexity after 50,000 steps on Penn Treebank

Type	#modules (M)	#parallel modules (K)	test perplexity
EM Modular Networks	15	1	**229.651**
EM Modular Networks	5	1	236.809
EM Modular Networks	15	3	246.493
EM Modular Networks	5	3	236.314
REINFORCE	15	1	240.760
REINFORCE	5	1	240.450
REINFORCE	15	3	274.060
REINFORCE	5	3	267.585
Noisy Top-k ($k = 4$)	15	1	422.636
Noisy Top-k ($k = 4$)	5	1	338.275
Baseline	1	1	247.408
Baseline	3	3	241.294



(a) Module selection entropy H_a

(b) Batch module selection entropy H_b

Figure 3: Modular networks are less noisy during optimization compared to REINFORCE and more deterministic than noisy top-k. Our method uses all modules at the end of training, shown by a large batch module selection entropy. The task is language modeling on the Penn Treebank dataset.

We compare the EM-based modular networks approach to unregularized REINFORCE and noisy top-k, as well as a baseline without modularity, that uses the same K modules for all datapoints. This baseline uses the same number of module parameters per datapoint as the modular version. For this experiment, we test four configurations of the network being able to choose K out of M modules at each timestep: 1 out of 5 modules, 3 out of 5, 1 out of 15, and 3 out of 15. We report the test perplexity after 50,000 iterations for the Penn Treebank dataset in Table 1.

When only selecting a single module out of 5 or 15, our modular networks outperform both baselines with 1 or 3 fixed modules. Selecting 3 out of 5 or 15 seems to be harder to learn, currently not outperforming a single chosen module ($K = 1$). Remarkably, apart from the controller network, the baseline with three static modules performs three times the computation and achieves worse test perplexity compared to a single intelligently selected module using our method. Compared to the REINFORCE and noisy-top-k training methods, our approach has lower test perplexities for each module configuration.

We further inspect training behavior in Figure 3. Using our method, all modules are effectively being used at the end of training, as shown by a large batch module selection entropy in Figure 3b. Additionally, the optimization is generally less noisy compared to the alternative approaches and the method quickly reaches a deterministic module selection. Figure 4 shows how the module selection changes over the course of training for a single batch. At the beginning of training, the controller essentially has no preference over modules for any instance in the batch. Later in training, the selection is deterministic for some datapoints and finally becomes fully deterministic.

For language modeling tasks, modules specialize on certain grammatical and semantic contexts. This is illustrated in Table 2, where we observe specialization on numerals, the beginning of a new sentence and the occurrence of the definite article *the*, indicating that the word to be predicted is a noun or adjective.

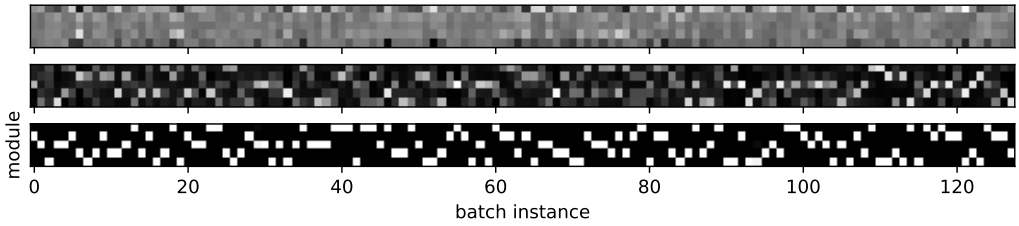


Figure 4: A visualization of the controller distribution for a particular mini-batch, choosing $K = 1$ out of $M = 5$ modules. Training progresses from the top image to the bottom image. A black pixel represents zero probability and a white pixel represents probability 1.

Table 2: For a few out of $M = 15$ modules (with $K = 1$), we show examples of the corresponding input word which they are invoked on (highlighted) together with surrounding words in the sentence.

Module 1	Module 3	Module 14
... than **<number>** <number> ...	... Australia **<new sentence>** A ...	... said **the** acquired ...
... be **substantially** less ...	... opposition **<new sentence>** I ...	... on **the** first ...
... up **<number>** <number> ...	... said **<new sentence>** But ...	... that **the** carrier ...
... <number> **million** was ...	... teachers **for** the ...	... to **the** recent ...
... $ **<number>** billion ...	... result **<new sentence>** That ...	... and **the** sheets ...
... <number> **million** of ...	... <new sentence> **but** the ...	... and **the** naczelnik ...
... $ **<number>** billion ...	... based **on** the ...	... if **the** actual ...
... by **<number>** to ...	... business **<new sentence>** He ...	... say **the** earnings ...
... yield **<number>** <number> ...	... rates **<new sentence>** This ...	... in **the** third ...
... debt **from** the ...	... offer **<new sentence>** Federal ...	... brain **the** skin ...
...	...	...

3.3 Image Classification

We applied our method to image classification on CIFAR10 [12] by using a modularized feed-forward network. Compared to [18], we not only modularized the two fully connected layers, but also the remaining three convolutional layers. Details can be found in the supplement.

Figure 5 shows how using modules achieves higher training accuracy compared to the non-modularized baseline. However, in comparison to the language modeling tasks, this does not lead to improved generalization. We found that the controller overfits to specific features. In Figure 5 we therefore compared to a more constrained convolutional controller that reduces overfitting considerably. Shazeer et al. [20] make a similar claim in their study and therefore only train on very large language modeling datasets. More investigation is needed to understand how to take advantage of modularization in tasks with limited data.

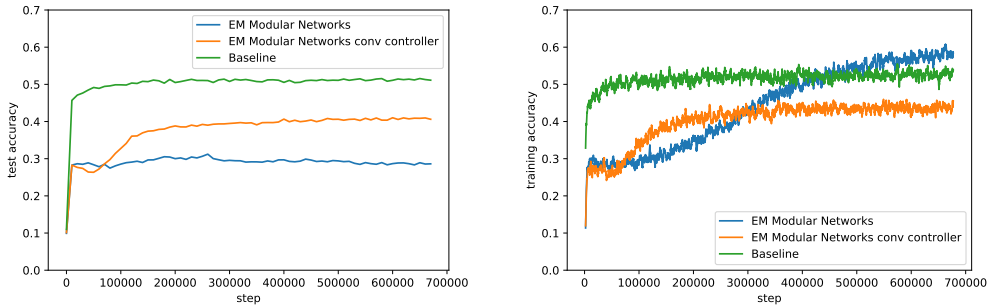


Figure 5: Modular networks test (left) and training accuracy (right) for a linear controller and a convolutional controller compared to the non-modularized baseline.

4 Related Work

Learning modules and their composition is closely related to mixtures of experts, dating back to [10, 11]. A mixture of experts is the weighted sum of several parameterized functions and therefore also separates functionality into multiple components. Our work is different in two major aspects. Firstly, our training algorithm is designed such that the selection of modules becomes fully deterministic instead of a mixture. This enables efficient prediction such that only the single most likely module has to be evaluated for each of the K distributions. Secondly, instead of having a single selection of modules, we compose modules hierarchically in an arbitrary manner, both sequentially and in parallel. The latter idea has been, in part, pursued by [9], relying on stacked mixtures of experts instead of a single selection mechanism. Due to their training by backpropagation of entire mixtures, summing over all paths, no clear computational benefits have yet been achieved through such a form of modularization.

Different approaches for limiting the number of evaluations of experts are stochastic estimation of gradients through REINFORCE [23] or noisy top-k gating [4]. Nevertheless, both the mixture of experts in [3] based on REINFORCE as well as the approach by [20] based on noisy top-k gating require regularization to ensure diversity of experts for different inputs. If regularization is not applied, only very few experts are actually used. In contrast, our modular networks use a different training algorithm, generalized Viterbi EM, enabling the training of modules without any artificial regularization. This has the advantage of not forcing the optimization to reach a potentially sub-optimal log-likelihood based on regularizing the training objective.

Our architecture differs from [20] in that we don't assign a probability to every of the M modules and pick the K most likely, but instead we assign a probability to each composition of modules. In terms of recurrent networks, in [20] a mixture-of-experts layer is sandwiched between multiple recurrent neural networks. However, to the best of our knowledge, we are the first to introduce a method where each modular layer is updating the state itself.

The concept of learning modules has been further extended to multi-task learning with the introduction of routing networks [18]. Multiple tasks are learned jointly by conditioning the module selection on the current task and/or datapoint. While conditioning on the task through the use of the multi-agent Weighted Policy Learner shows promising results, they reported that a single agent conditioned on the task and the datapoint fails to use more than one or two modules. This is consistent with previous observations [3, 20] that a RL-based training without regularization tends to use only few modules. We built on this work by introducing a training method that no longer requires this regularization. As future work we will apply our approach in the context of multi-task learning.

There is also a wide range of literature in robotics that uses modularity to learn robot skills more efficiently by reusing functionality shared between tasks [19, 6]. However, the decomposition into modules and their reuse has to be specified manually, whereas our approach offers the ability to learn both the decomposition and modules automatically. In future work we intend to apply our approach to parameterizing policies in terms of the composition of simpler policy modules.

5 Conclusion

We introduced a novel method to decompose neural computation into modules, learning both the decomposition as well as the modules jointly. Compared to previous work, our method produces fully deterministic module choices instead of mixtures, does not require any regularization to make use of all modules, and results in less noisy training. Modular layers can be readily incorporated into any neural architecture. We introduced the modular gated recurrent unit, a modified GRU that enables minimalistic sequence-to-sequence models based on modular computation. We applied our method in language modeling and image classification, showing how to learn modules for these different tasks.

Training modular networks has long been a sought-after goal in neural computation, since this opens up the possibility to significantly increase the power of neural networks without an increase in parameter explosion. We have introduced a simple and effective way to learn such networks, opening up a range of possibilities for their future application in areas such as transfer learning, reinforcement learning and lifelong learning.

References

[1] D. Amodei et al. “Deep Speech 2: End-to-End Speech Recognition in English and Mandarin”. In: *ICML* (2015).

[2] E. Bengio. “On Reinforcement Learning for Deep Neural Architectures: Conditional Computation with Stochastic Computation Policies”. PhD thesis. McGill University Libraries, 2017.

[3] E. Bengio et al. “Conditional Computation in Neural Networks for Faster Models”. In: *ICLR Workshop* (2016).

[4] Y. Bengio, N. Léonard, and A. Courville. “Estimating or propagating gradients through stochastic neurons for conditional computation”. In: *arXiv preprint arXiv:1308.3432* (2013).

[5] K. Cho et al. “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: *Conference on Empirical Methods in Natural Language Processing* (2014).

[6] I. Clavera, D. Held, and P. Abbeel. “Policy transfer via modularity and reward guiding”. *IEEE International Conference on Intelligent Robots and Systems*. Vol. 2017-Septe. 2017, pp. 1537–1544.

[7] J. Clune, Mouret J-B., and H. Lipson. “The evolutionary origins of modularity”. In: *Proceedings of the Royal Society of London B: Biological Sciences* 280.1755 (2013).

[8] A. Coates et al. “Deep learning with COTS HPC systems”. In: *ICML* (2013), pp. 1337–1345.

[9] D. Eigen, M. Ranzato, and I. Sutskever. “Learning Factored Representations in a Deep Mixture of Experts”. In: *ICLR Workshop* (2013).

[10] R. A. Jacobs et al. “Adaptive Mixtures of Local Experts”. In: *Neural Computation* 3.1 (1991), pp. 79–87.

[11] M. I. Jordan and R. A. Jacobs. “Hierarchical Mixtures of Experts and the EM Algorithm”. In: *Neural Computation* 6.2 (1994), pp. 181–214.

[12] A. Krizhevsky and G. Hinton. “Learning Multiple Layers of Features from Tiny Images”. MSc thesis. University of Toronto, 2009.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “ImageNet classification with deep convolutional neural networks”. In: *NIPS* (2012), pp. 1097–1105.

[14] L. Li, Z. Ding, and D. Huang. “Recognizing location names from Chinese texts based on Max-Margin Markov network”. In: *International Conference on Natural Language Processing and Knowledge Engineering* (2008).

[15] R. M. Neal and G. E. Hinton. “Learning in Graphical Models”. In: ed. by M. I. Jordan. Cambridge, MA, USA: MIT Press, 1999. Chap. A View of, pp. 355–368.

[16] B. Pakkenberg et al. “Aging and the human neocortex”. In: *Experimental gerontology* 38.1-2 (2003), pp. 95–99.

[17] M. Ramezani et al. “Joint sparse representation of brain activity patterns in multi-task fMRI data”. In: *IEEE Transactions on Medical Imaging* 34.1 (2015), pp. 2–12.

[18] C. Rosenbaum, T. Klinger, and M. Riemer. “Routing Networks: Adaptive Selection of Non-linear Functions for Multi-Task Learning”. *ICLR*. 2018.

[19] H. Sahni et al. “Learning to Compose Skills”. In: *NIPS Workshop* (2017).

[20] N. Shazeer et al. “Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer”. In: *ICLR* (2017).

[21] O. Sporns and R. F. Betzel. “Modular Brain Networks”. In: *Annual Review of Psychology* 67.1 (2016), pp. 613–640.

[22] P. Sternberg. “Modular processes in mind and brain”. In: *Cognitive Neurophysiologogy* 28.4 & 4 (2011), pp. 156–208.

[23] R. J. Williams. “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning”. In: *Machine Learning* 8 (1992), pp. 229–256.

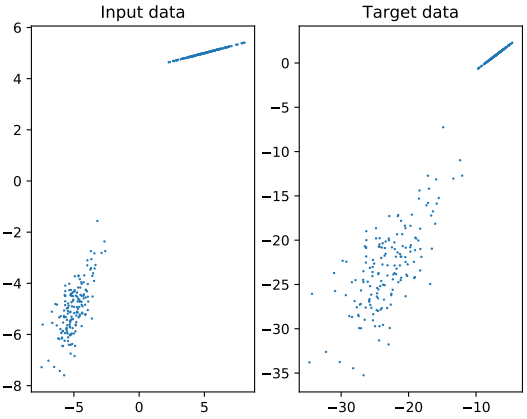


Figure 6: The toy regression problem. Input data from a mixture of Gaussians (left) is transformed by scaling data from one mixture component and rotating the other around the origin yielding the target data (right).

A Modular GRU

We propose the *modular GRU*, defined by

$$
\begin{aligned}
z_t &= \sigma(W_z \cdot [h_{t-1}, x_t]) \\
r_t &= \sigma(W_r \cdot [h_{t-1}, x_t]) \\
\tilde{h}_t &= \text{ReLu}(\sum_{i=0}^{K} f_{a_i^{(t)}}([r_t \odot h_{t-1}, x_t])) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t
\end{aligned}
$$

where h_t is the RNN's hidden state at time t, z_t and r_t are gate values and $\tilde{h}_t$ is the candidate state produced by the modular layer at time step t. Both the controller $p(a_t|h_{t-1}, x_t, \phi)$ and the modules $f_m([r_t * h_{t-1}, x_t]; \theta_m)$ depend on the current input x_t and previous state h_{t-1}. The parameters ϕ and θ_m for $m = 1, \ldots, M$ are shared across all timesteps.

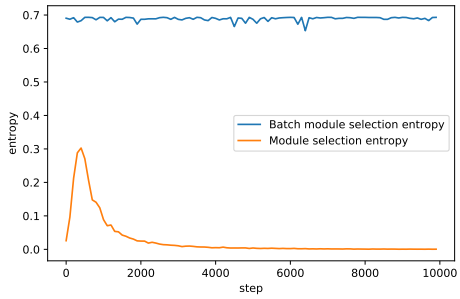
B Toy Regression Problem

To demonstrate the ability of our algorithm to learn conditional execution, we constructed the following regression problem: For each data point (x_n, y_n), the input vectors x_n are generated from a mixture of Gaussians with two components with uniform latent mixture probabilities $p(s_n = 1) = p(s_n = 2) = \frac{1}{2}$ according to $x_n|s_n \sim \mathcal{N}(x_n|\mu_{s_n}, \Sigma_{s_n})$. Depending on the component s_n, the target y_n is generated by linearly transforming the input x_n according to

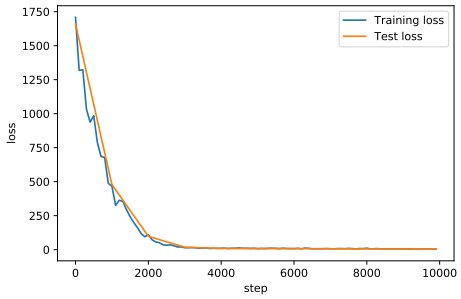
$$
y_n = \begin{cases} Rx_n & \text{if } s_n = 1 \\ Sx_n & \text{otherwise} \end{cases} \tag{17}
$$

where R is a randomly generated rotation matrix and S is a diagonal matrix with random scaling factors. Figure 6 shows one possible such dataset.

In the case of our toy example, we use a single modular layer, $L = 1$, with a pool of two modules, $M = 2$, where one module is selected per data point, $K = 1$. Loss and module selection entropy quickly converge to zero, while batch module selection entropy stabilizes near $\log 2$ as shown in Figure 7. This implies that the problem is perfectly solved by the architecture in the following way: Each of the two modules specializes on regression of data points from one particular component by learning the corresponding linear transformations R and S respectively and the controller learns to pick the corresponding module for each data point deterministically, thereby effectively predicting the identity of the corresponding generating component.



(a) Module composition learned on the toy dataset.



(b) Minimization of loss on the toy dataset.

Figure 7: Performance of one modular layer on toy regression.

C Language Modeling Experiments

We use a batch size of 128 elements and unroll the network for 35 steps, training with input sequences of that length. Each module has only 8 units. For the E-step we sample $S = 10$ paths and run 15 gradient steps in each partial M-step.

D Image Classification Experiments

We use three modular layers with 3x3 convolutional kernels, striding two, and a single output channel followed by two fully connected modular layers with 48 hidden units and 10 output units respectively. We execute 2 out of 10 modules in each layer and concatenate their outputs. The baseline is non-modular with the same number of parameters that 2 modules would have. In effect, this roughly matches the amount of computation that is required to run the modular and non-modular variant.