Applying conditional computation methods by learning neuron partitions

Vincent Antaki, Emmanuel Bengio
School of Probably Science, McGill

Abstract

Conditional computation methods enable lazy evaluation of neural networks by teaching input-dependent gaters to partially evaluate these networks so as to minimize computation time but preserve accuracy. We propose to train such gaters on already-trained neural networks, and show that it is possible to recover a good accuracy while doing less computations. We also show that pretraining neural networks with the intent of applying conditional computation methods to them increases the accuracy of the resulting lazy model.

1 Introduction

Conditional computation brings the idea of having large powerful neural network models that are only partially evaluated depending on the input, thus saving many precious processor cycles (Bengio *et al.*, 2013; Denoyer and Gallinari, 2014; Bengio *et al.*, 2015; Almahairi *et al.*, 2016; Graves, 2016; Shazeer *et al.*, 2017; Odena *et al.*, 2017). Many challenges arise from such an approach; how should one partition a neural network? How and when should one train the policy that routes computation? How should such a large architecture be trained?

Since there are many moving parts in this paper, we will try to stick to the following terminology. The main network that does some task, e.g. classification, will be referred to as the **target network**. The algorithm or method that creates a partitioning of the computation will be referred to as the **partition model**. The algorithm, method or stochastic policy that predicts which partitions are activated given an input will be referred to as the **computation policy**. The union of these parts into a single entity that performs a task using lazy evaluation of parts of itself will be referred to as the **lazy model**. The *act* of lazily evaluating a network given an input is also referred to as *conditional computation*.

So far in prior work, the computation structure (the partition model) is fixed by hand by the experimenter, either to be a tree-like structure (Denoyer and Gallinari, 2014) or a masking method (Bengio *et al.*, 2015; Shazeer *et al.*, 2017; Odena *et al.*, 2017) over predefined blocks, blocks of nodes or layers in a neural net. In these scenarios both the computation policy and the target network are initialized and trained synchronously.

Learning such lazy models efficiently and correctly poses a major challenge due to the interaction of the computation policy and the target model. This problem may be similar to the interaction of two learning agents encoutered in Actor-Critic (Sutton *et al.*, 1999) and Generative Adversarial Network (Goodfellow *et al.*, 2014) methods. In these methods it is known that such interaction sometimes leads to bad solutions, where both parts coadapt in a local minima. We propose an alternative approach without such interaction where the target network is instead trained to convergence in a greedy setting where the entire network is evaluated, and then subjected to a computation policy that is learned via RL methods.

This creates a new problem. Since the target network is not pretrained for specific partitions (it is fully evaluated), then randomly cutting it into e.g. blocks of neurons doesn't have a semantic value that could be exploited by a computation policy. Instead, neurons that need to be activated together are "scattered" throughout the network, and we now have to group them and discover these interactions.

We propose to train learnable partition models in order to partition nodes in a neural network and minimize loss. Such models iteratively adapt partitioning via sampling. In other words, a partitioning of the nodes is set, matched to a computation policy, and its value determined by the validation loss of the resulting lazy model.

We show results for several such partition models, as well as for some policy gradient methods applied to computation policies.

2 Partitioning the network

In order to perform lazy evalutation of a neural network, we assign each neuron $h \in \mathcal{H}$ to a **partition** $\mathcal{H}_i$. This creates a partitioning $\mathcal{P} = \{\mathcal{H}_1, ..., \mathcal{H}_k\}$ with k partitions. Lazy evaluation of a network consists in selecting a subset of the partitions $M \subseteq \mathcal{P}$ that is to be evaluated, the rest being considered zero. This is achieved by an input-dependent policy $\pi_\omega(M|X)$ (see Section 3).

We decide to use learning methods to learn the optimal partitioning of a given target network T_θ.

2.1 Using bandits

We first consider learning 1 bandit per neuron, creating a partitioner P_μ with parameters μ. Each bandit has k actions, the number of partitions. To create a partition $\mathcal{P}$ all bandits are queried. The reward that each bandit gets is the validation accuracy of a lazy network trained to convergence with partition $\mathcal{P}$ and policy π_ω (see Algorithm 1).

Bandits are trained until the resulting lazy network matches the accuracy of the target network, or no improvement is shown after several iterations. We train bandits with the greedy bandit method as well as the Upper-Confidence-Bound algorithm (Auer *et al.*, 2002).

Algorithm 1: Training the partition model based on validation feedback

Train target network T_θ, then freeze θ
while Z *does not match* T_θ **do**
  Sample partition $\mathcal{P} \sim P_\mu$
  Initialize random policy π_ω
  Train lazy network $Z = \langle T_\theta, \pi_\omega, \mathcal{P} \rangle$ by learning ω
  Update partitioner P_μ based on validation accuracy of Z
end

While the validity of using such an approach can be questioned, it is a reasonable baseline in the sense that it is the most trivial learned –not handcrafted– partition method. However, because each bandit is considered independently, there is no way for them to coordinate and have two similar neurons thrown into the same partition. Seeing them scale to many neurons and many partitions would be surprising.

2.2 Using contextual bandits

To take advantage of the fact that similar neurons should be in similar partitions, we also train a partitioner using contextual bandits. A single contextual bandit is applied to each neuron, the action is again choosing one of k partitions, and the context is the incoming weight column $\mathbf{w}_i = W_{:,i}$. Note that since the space of hidden activations changes at each layer, we train one contextual bandit per layer.

We use a Linear Response Bandit algorithm similar to the one described in Tewari and Murpy (2017) with the intent to iteratively generate and improve partitions of the network. Let the function $f(\mathbf{w}) = \hat{\beta}_a^T \mathbf{w}$ be the estimate of the Q-value for associating the partition a to a neuron with context $\mathbf{w}$. Using the set of previous examples where we've associated the neuron with weights $\mathbf{w}$ with the partition a, we compute $\hat{\beta}_a$ using Least Squares. We make no change from non-contextual bandit approach with regards to the reward signal.

To generate the best partition from our current estimate, we take $\arg\max_a \hat{\beta}_a \mathbf{w}$ over partitions $a \in [k]$. To deal with the explore and exploit dilemma, we make our policy epsilon-greedy; we will start with a high epsilon and gradually decrease it.

2.3 Gumbel-Softmax

One very large downside of bandit approaches (and approaches using Algorithm 1 in general), is that one has to reset π after each action of the partitioner, and wait for π and the resulting lazy network to converge before taking a new action. While such an approach eliminates the interaction of two learning agents, it is also extremely costly.

Another possibility is to also learn the partitioning *while* learning π. This creates a new two-agent interaction, but is possibly much faster. Another upside of such a two-agent interaction is that there is an obvious fallback solution for π, i.e. to activate every partition.

We make use of the Gumbel-Softmax (Jang *et al.*, 2016) reparameterization trick to achieve this. By assigning a vector of partition preference μ_i to each neuron, and by sampling the partition id of each neuron with the following Gumbel-Softmax, we can obtain $\nabla_\mu \mathcal{L}$, and thus optimize μ and ω simultaneously.

$$u_i \sim U[0,1] \qquad g = -\log(-\log(u_i + \epsilon) + \epsilon) \qquad y_i = \text{softmax}\left(\frac{\mu_i + g}{\tau}\right)$$

There are two ways to make y_i into a hard decision; set τ to be very small, or use $\hat{y}_i$:

$$\hat{y}_i = |\mathbb{1}_{y_i} - y_i|_{cst} + y_i$$

where $\mathbb{1}_x$ is a one-hot vector at index $\text{argmax}_j x_j$, and $|x|_{cst}$ signifies that x is considered constant when computing gradients. We use the second method of using $\hat{y}_i$, where the argmax of $\hat{y}_i$ expresses the partition of neuron h_i.

3 Learning computation policies

There are many possible ways to train input-dependent computation policies. Generally in prior work, a score is assigned to each possible partition. This score can be arbitrary, used to perform top-k selection of the partitions (Odena *et al.*, 2017; Shazeer *et al.*, 2017), can be independent probabilities used to perform n-Bernoulli sampling (Bengio *et al.*, 2015), or can be categorical probabilities used to either deterministically pick top-1 or perform softmax sampling over partitions (Denoyer and Gallinari, 2014).

In this work we use the n-Bernoulli scenario, where for some partition set $\mathcal{P}$ a computation policy π_ω will map $X \to M = [0,1]^{|\mathcal{P}|}$, i.e. the probability of a mask M of $|\mathcal{P}|$ Bernoullis bits, one for each partition. At inference, each of these bits is sampled, a 1 indicating that the associated neurons in $\mathcal{H}_i$ are to be computed, 0 that the neurons are set a (virtual) 0 value.

3.1 REINFORCE

The first easy choice for optimizing a stochastic parameterized policy π_ω is to use the REINFORCE gradient estimator (Williams, 1992):

$$\nabla_\omega \mathcal{L}(X) = (\mathcal{L} - b)\nabla_\omega \log \pi_\omega(M|X)$$

We use an average loss baseline b which is the mean loss of a minibatch.

3.2 Deterministic Policy Gradient

Deterministic Policy Gradient (DPG, Silver *et al.* (2014)), and its Deep equivalent DDPG (Lillicrap *et al.*, 2015), is typically used to provided a biased gradient estimate of a deterministic policy $a = \pi_\omega(s)$ through the estimate of $Q_\theta(s,a)$ and the gradient obtained from the chain rule $\delta\omega = \nabla_a Q(s,a)\nabla_\omega a$.

DDPG can also be used to learn a stochastic policy, by considering that the deterministic action is in fact the probability distribution $\pi_\omega(a|s)$, and that the environment takes care of sampling it (in possibly unknown ways but presumably without any bias).

We use such a scenario to optimize ω, and train a deep network $Q_{\omega'}(X, M)$ to estimate $\mathcal{L}(X)$ of the lazy network, which allows us to get gradient estimates for ω and minimize the loss.

4 Pretraining Networks with Conditional Computation in mind

What if it were possible to push the units of a target network to organize themselves in such a way that applying conditional computation methods to the trained target network was easy? We investigate this here.

4.1 Pretraining with random dropout rates

During the partitioning and learning of the computation policy, two things happen because of the "random exploration" phase of these two steps. First, units that usually fire together aren't, and second, much less units fire in total that normally fire in the forward pass.

We suggest that it is possible to render the target network robust to such problems by applying dropout (Srivastava *et al.*, 2014) during the training of the target network. Furthermore, when the computation policy explores, it will activate randomly and with a varying amount of sparsity; sometimes it will activate 80% of units, sometimes 20%. As such we also randomly vary the dropout probability uniformly during training sampled from a $U[\epsilon, 1]$ distribution.

Varying dropout rates during training has revealed in our limited experiments to allow lazy networks to be much more robust to having a bad policy early on in training. Additionally, we observed both an increased validation accuracy of the fully-evaluated target network, and an increased validation accuracy of lazy models.

4.2 Pretraining with information flow prior

We define information flow through the connection between neuron i and j as the product between its weight w_{ij} and the post-activation value of neuron i, h_i. Given the prior that different classes of examples should have different flow through the network, we investigate applying a penalty that depends on the similarity in the average magnitude of the signal for each connection for each class. An interesting property induced by using ReLUs it that the information flow for a given connection always has the same sign (or is null).

Given a batch of M examples, let f_{ij}^c be the average flow through w_{ij} for the examples of class c and $\mathcal{L}_{flow}$ be the loss associated with the flow of all classes through.

$$f_{ij}^c = \frac{1}{n_c} \sum_{m=1}^{M} |h_{mi} w_{ij}| \mathbb{1}_{c_m=c}$$

$$\mathcal{L}_{flow} = \sum_{ij} \prod_{c} f_{ij}^c$$

We hope to push the network to separate information as early as it can layer-wise, even if it means using more neurons on the latter layers. (Add details?****)

There are a few things to notice about the current formulation of this regularization. For one, we need at least one example of every class in every batch, else we have $\frac{1}{n_c} = \infty$ multiplied by $\sum_{m=1}^{M} |h_{mi} w_{ij}| \mathbb{1}_{c_m=c} = 0$. Furthermore, the regularization varies with respect to our minibatch class distribution. It is also observed that with bigger batches, less variance is experienced. Finally, if all examples from a given class in our minibatch have no flow through a certain connection, then the penalty is 0 for that connection. This could be avoided by defining e.g. $\mathcal{L}_{flow} = \sum_{ij} \sum_{c_1,c_2} f_{ij}^{c_1} f_{ij}^{c_2}$. However, such definition would imply much more computation, and our current formulation is already very computationally expensive.

5 Results

We pretrain our target networks with various regularizations. We report validation accuracy of the resulting lazily evaluated networks, as well as the original accuracy of the target network. We also report the average ratio of these accuracies for each method, which represents how much accuracy each method has recovered from the original target network.

For all our experiments, the target sparsity is set to 25%, but in practice policies achieve sparsity of about $35 \pm 5\%$. Unfortunately, since sometimes partitioners create unbalanced partitions, the sparsity is sometimes lost to a single partition containing most units that is often activated. This is especially true when using contextual bandits, as Gumbel-Softmax are more inherently random they tend not to put all units in the same partition.

We train ReLU networks of 2 to 4 layers of 200, 400, or 800 units each. We train them on the Street View House Numbers dataset (Netzer *et al.*, 2011), with learning rates of 0.05, 0.005 or 0.001. All implementations are done using Theano (Theano Development Team, 2016).

5.1 Bandit partitioning

Despite our best efforts, regular bandits did not perform much better than a random partition of the nodes. From our limited experiments both optimistic greedy bandits and UCB-bandits reached at most an accuracy ratio of .69, while a random partition has an accuracy ratio of around .66.

Due to the setup, we may not perform as many trials for these bandits as is typically used in bandits litterature, as each trial involves learning a computation policy. For small networks, this means tens of minutes to hours to trial. As such, it isn't possible for bandits to discover meaningful actions and to coordinate through repetition.

5.2 Contextual Bandit partitioning

Due to the extensive computation cost of our approach, we perform our experiences on a 3-layer, 800 hidden-units-per-layer trained with sampled dropout rate. Having been trained with varying dropout, this network is very robust to random desactivation of neurons; this helps drastically with potential variance in the partition policy. The target model's validation accuracy is 92.75 %.

We use an initial epsilon of 0.8 and gradually decrease it to 0.1 on a span of 10 lazy net epochs; we let computations run for another 10 epochs afterward. The partition is updated after each time the computation policy has converged.

Using a random partition ($\sim$150 neuron by partitions, standard deviation of $\sim$10), we achieve 79.62% validation accuracy and the sparsity activation on the validation set is 35%. After the first epoch, we have a partition which is 80% random and our validation accuracy reaches 87.5%. However, the average sparsity of activation reaches 54.3%. We also observe that standard deviation of the average activation by partition to be of 12.06% instead of 2.17% for the random partition. Subsequent partitions and activation policy neither overperform or underperform significantly the second one ($\pm 1.5\%$). After 15 lazy net epoch, our average sparsity of activation is 46.94% and the std of average partition activation is 15.98%

Interesting phenomena can be observed as epsilon decreases throughout our model training; the average size of partition stays very close to 150 but the variance in the size of partitions increases as our partition policy becomes more greedy. At lazy net epoch 15, the standard deviation in the size of partitions is 14.55. Furthermore, the size of partition seems to converges again due to our policy generation becoming greedier.

We also performed this experience on a 2-layer, 200 hidden-units-per-layer MLP, trained without dropout. The target network's validation accuracy is 93.32%. The initial accuracy with a random partion is 25.45 % and 27.96 % the first epoch. After 10 lazy net epochs, we see a validation accuracy of 48.78 %. Computations are still running as we submit this work.

5.3 Gumbel-Softmax partitioning

We test using Gumbel-Softmax as a partition method on all aforementioned configurations. We pretrain networks without dropout, with regular $p = 0.5$ dropout, as well as $p \sim U[\epsilon, 1]$ dropout. From Tables 1 and 2, we find that pretraining with random dropout and using DPG works best, although by a small margin. Unfortunately our results for the No Dropout case are incomplete, but from the logs of our experiments, we suspect that the fact that these networks are not robust to bad partitions and policies makes it hard for them to reach acceptable accuracies.

One possible problem with this method is that the Gumbel-Softmax remains very stochastic even after much training because temperature is not properly scheduled. Making the partition more deterministic towards end of training might help the policy adjust and be confident after much training. This may explain why this method performs slightly worse than contextual bandits, who choose a fixed partition throughout training of a lazy model.

accuracy ratio	REINFORCE	DPG
No Dropout [1]	0.484 ± 0.110	-
0.5 Dropout	0.882 ± 0.006	0.886 ± 0.013
Random Dropout	0.888 ± 0.008	0.908 ± 0.012

Table 1: Top-5 Accuracy ratios for Gumbel-Softmax partioners using either regular 0.5 dropout or random dropout, and computation policies trained with either REINFORCE or DDPG.

validation accuracy (%)	REINFORCE	DPG	Original
No Dropout[1]	45.0 ± 10.8	-	92.8 ± 1.4
0.5 Dropout	80.5 ± 0.8	81.0 ± 0.5	92.5 ± 0.3
Random Dropout	82.6 ± 1.0	83.6 ± 1.2	93.7 ± 0.0

Table 2: Top-5 validation accuracies for Gumbel-Softmax partioners using either no dropout, regular 0.5 dropout or random dropout, and computation policies trained with either REINFORCE or DDPG. Original accuracies (of the fully evaluated target network) are also reported.

5.4 How adapted are neural networks to conditional computation?

Given how hard the task we are attacking may be, we try to understand whether it makes sense at all to try to perform conditional computation on neural networks. One experiment we run compares coactivation of networks trained with and without dropout (Figure 1). We measure coactivation between h_i and h_j (regardless of layer) as $c_{ij} = \mathbb{E}[\mathbb{I}_{h_i>0}, \mathbb{I}_{h_j>0}]$, in other words we measure how much h_i and h_j tend to activate at the same time.

What we observe is that while dropout does reduce the amount of coactivation inside a neural network, there still remains groups of strongly codependent neurons. Additionally, neurons tend to coactivate with most other neurons instead of just a subset of neurons, which is what would be ideal for conditional computation (because then it would be easy to bundle these coactivating neurons together!) As such, while they seem to be helping, our current methods of pretraining are arguably still not suited to create conditional computation-compatible neural networks.

5.5 On flow regularization

A major drawback of this regularization is the major cost it has in term of memory usage. To give an intuition on this cost, consider that the vectorized implementation of flow regularization makes usage of 4 dimension tensors for every layer (number of class $\times$ minibatch size $\times$ number of nodes on previous layer $\times$ number of node on this layer).

[1] Due to a bug the "No Dropout" results are almost all wrong. We include here the average of a few experiments that survived the bug. They are representative of the early stages of learning, but clearly full convergence results would be better; although we speculate from looking at experiment logs that they would still be much worse than when using Dropout.

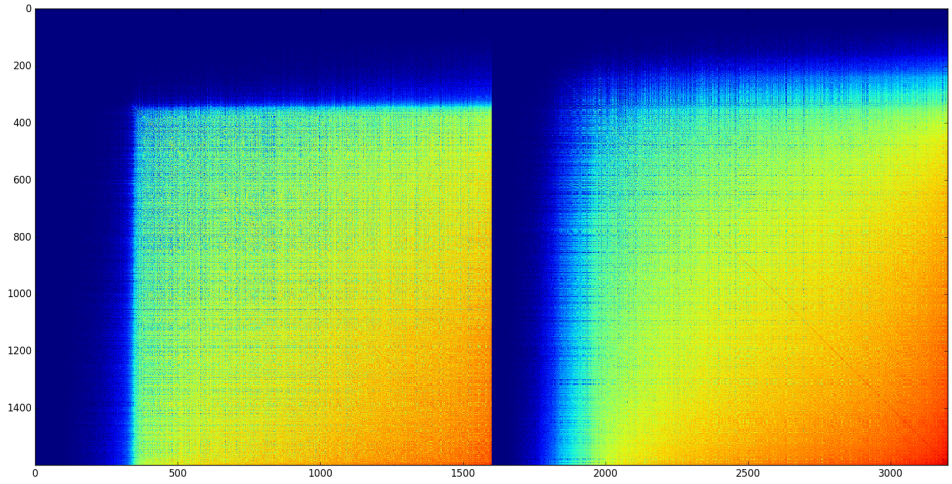

Figure 1: Coactivation of (sorted) neurons. The left part represents coactivation of a network trained with dropout; the right part is for a network trained without dropout.

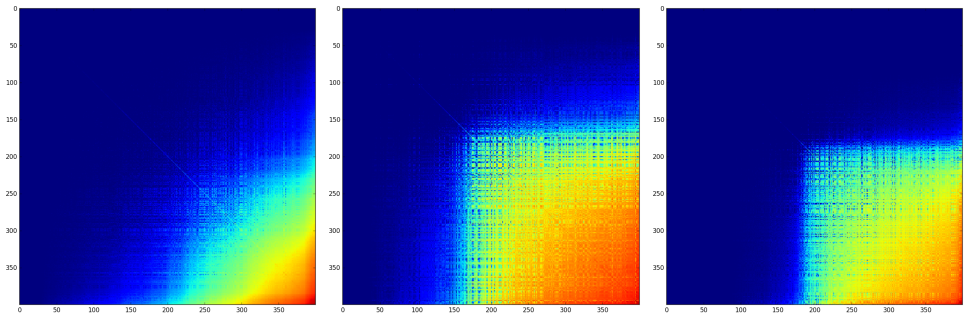

Figure 2: Coactivation of (sorted) neurons, using class-flow regularization. From left to right is the same plot as training progresses. We speculate that this regularization creates groups of coactivating units rather than spreading coactivation as dropout does.

For our computation, we initially attempted to train a 2-layer, 250 neurons by hidden layer with a batch size of 32. This caused memory overflow on our 4GB of gpu-RAM and our 8GB cpu-RAM. By reducing the minibatch size to 20, the model trained at an approximative speed of 835 minibatches by hour on cpu (3 % of the training set). Not wanting to wait month to obtain results, we aborted this operation.

To be able to acquire results on our regularization, we tranfered our computing on a NVIDIA Titan X with 12 Gb of VRAM. We also limited the maximum number of minibatch by epochs to 200. We trained a 2-layer network with 200 neurons by hidden layer and monitor the co-activation of neurons throughout at different time during our training.

On figure 2, we can see some interesting evolution in the coactivation of neurons. However, the validation accuracy of our model peaks at 46.95% (correspond rightmost image of fig 2). It is reasonnable to speculate that, similarly as dropout, applying that regularization increases our need in term of minimal model complexity required for decent performance.

With this in mind and our GPU with a lot of RAM, we train a 2-layer 500 hidden units per layer network. That network reached a top validation accuracy of 74.37% after 16 lazy net epoch. Figure 3 shows evolution of the co-activation of that net.

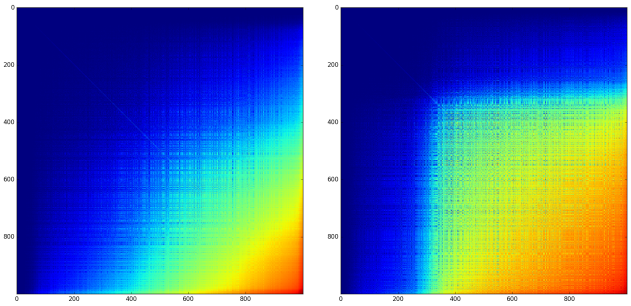

Figure 3: Coactivation of (sorted) neurons, using class-flow regularization on a 2-layers 500 neurons-by-hidden-layer network. From the left to right, the image are at epoch 4 (48.98% valid accuracy) and epoch 8 (68.42%).

6 Discussion

We have proposed a novel approach to conditional computation, which involves splitting up pretrained neural networks to somewhat avoid the problems linked to previous approaches.

To create neuron partitions, it is possible to use Bandit and Contextual Bandit approaches. These methods repeatedly create partitions which are used to train lazy networks, and get feedback from validation accuracies of the resulting lazy networks. It is also possible to reparameterize a per-neuron partition preference using a differentiable Gumbel-Softmax, which allows to stochastically create partitions during training of a lazy network and update said partition preferences.

Such methods allow to train lazy networks that perform conditional computation but retain most of the original target network's accuracy.

We have successfully trained a MLP up to 74 % validation accuracy on the SVHN dataset with our newly defined class-flow regularization. Observing the co-activation of the neurons in that layer, we noticed what seems to be sparser co-activation vector.

In addition, it is possible to pretrain target networks with conditional computation in mind. Through application of dropout, it is possible to make target networks very robust to being converted to lazy networks, and achieve higher validation accuracy. We suggest that this is also possible with the class-flow regularization, although we have not tried it yet.

Also, it is noteworthy to mention that we empirically verified our intuition of the high variance of the class-flow regularization throughout the minibatches. A main factor that induce that behavior comes from our definition of co-flow as a product of all class-flow; for a given weight w_ij, the product between all classes flow is 0 if at least one class never activate neuron i. In future experiment, we may consider applying a layer-wise normalization of the flow to reduce variance in our regularization.

Empirical time evaluation While we measure sparsity rates in our experiments, we did not adapt previous sparse computation code to our current setup. An obvious selling point of our method is that the number of computation is reduced, and time is gained. It would thus be interesting to have specialized sparse code to measure true speedup.

Contextual Bandit : extended context In our current implementation the context of a neuron is its incoming weights. It could be advantageous to also consider outgoing weights of a neuron as to help the contextual bandit group units together.

Statement of contributions

Emmanuel worked on Bandit and Gumbel-Softmax partitioners and dropout pretraining. Vincent worked on flow regularization for pretraining of models as part of COMP652 class and contextual bandits as part of the COMP767 class. Implementations and report are joint work.

References

Amjad Almahairi, Nicolas Ballas, Tim Cooijmans, Yin Zheng, Hugo Larochelle, and Aaron Courville. Dynamic capacity networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning-Volume 48*, pages 2091–2100. JMLR. org, 2016.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. Conditional computation in neural networks for faster models. *arXiv preprint arXiv:1511.06297*, 2015.

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

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, volume 2011, page 5, 2011.

Augustus Odena, Dieterich Lawson, and Christopher Olah. Changing model behavior at test-time using reinforcement learning. *arXiv preprint arXiv:1702.07780*, 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. *arXiv preprint arXiv:1701.06538*, 2017.

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 387–395, Bejing, China, 22–24 Jun 2014. PMLR.

Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99, pages 1057–1063, 1999.

Ambuj Tewari and Susan A. Murpy. From ads to interventions: Contextual bandits in mobile health. *Mobile Health: Sensors, Analytic Methods, and Applications*, 2017.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

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