Reinforcement learning of conditional computation policies for neural networks

Emmanuel Bengio
Pierre-Luc Bacon
Ryan Lowe
Joelle Pineau
Doina Precup
School of Computer Science, McGill University, Montreal, Canada

Abstract
Deep neural networks have achieved state-of-the-art performance in many areas, in large part due to the learning of hierarchical feature representations; however, evaluating these models can be unfeasible on resource-constrained devices. We present conditional computation methods for increasing the computation speed in these networks, by learning policies that actively select which parts of a network to activate at a given time. We present results from Bengio et al. (2015) showing how this can be implemented in a fully-connected neural network by dropping out blocks of units at a time. We then propose novel policy parameterizations for convolutional neural networks, which learn to drop out entire layers at a time. Although these policies are trained using the straightforward REINFORCE algorithm, our results indicate encouraging speedups, and we believe further investigation in this area could lead to drastic improvements.

1. Introduction
Deep learning architectures have achieved unprecedented empirical results in areas such as image and speech recognition (He et al., 2015), machine translation (Bahdanau et al., 2014), and playing Atari games with reinforcement learning (Mnih et al., 2015). Deep neural networks learn hierarchical, non-linear data abstractions, which are used to solve a particular task. One of the reasons for the state-of-the-art results using neural networks is that abstractions in these networks contain a certain structure that is well-suited to real-world problems; in particular, they assume abstractions are distributed (meaning that one can learn some components of abstractions independently from others), and that abstractions are deep (meaning that simpler features can be composed to make more complex features) (Bengio et al., 2007). This corresponds intuitively to the compositional structure inherent in real-world data, such as natural images, where one can for example learn abstractions independently for different objects in a scene, and can compose simple features such as edges and curves into more complex abstractions.

Unfortunately, the task of training and evaluating neural networks remains challenging: they can take a long time to train, can easily overfit to the training data, and are difficult to deploy on resource-constrained devices. To solve issues related to computation time, recent approaches have proposed conditional computation (Bengio et al., 2013; Davis & Arel, 2013). Conditional computation refers to activating only some of the units in a network, in an input-dependent fashion. As an intuitive example, if we think we’re looking at a car, we only need to compute the activations of the vehicle detecting units, not of all features that a network could possible compute. The immediate effect of activating fewer units is that propagating information through the network can be faster, both at training time and at test time. However, one needs to be able to decide intelligently which units to turn on and off from the input data. This is typically achieved with some form of gating structure, learned in parallel with the original network.

Conditional computation can naturally be viewed as sequential decision making problem. At each layer, given information on the current state of the computation (what has been computed so far), one has to decide which units to activate. There is also a natural, delayed reward measure, given by the error at the output of the network; one can further impose rewards that encourage sparsity in the computation, or a fast computation time. This view makes this problem a natural fit for reinforcement learning algorithms.

We propose to learn input-dependent activation probabilities for subsets of neural network parameters. In the case

Preliminary work. Under review by the ICML Workshop on Abstraction in Reinforcement Learning. Do not distribute.
of a fully-connected network we learn to activate contiguous blocks of nodes, while for our new architecture with convolutional networks we learn to activate or skip entire layers. We attempt to jointly minimize the prediction errors and the number of participating parameters, thus reducing the computational load. Our method can be thought of as using reinforcement learning to aid the learning of data abstractions. Further, since the dropout policies effectively reason about subsets of parameters and the features they calculate, we can think of these policies as learning abstractions of the abstraction learning procedure itself in deep neural networks.

2. Problem formulation

Our goal is to learn input-dependent activation probabilities for various parts of a neural network, which we formulate as learning policies. We define a discrete time, continuous state and discrete action MDP \((S, \mathcal{U}, P(\cdot | s, u), C)\). An action \(u \in \{0, 1\}^k\) in this model consists of the application of a binary mask over blocks or layers. The cost \(C\) is the loss of the neural network architecture (in our case the negative log-likelihood). As a first step, we do not consider the full sequential decision problem, and assume that after the action is taken, the trial ends and the reward is observed. A fully sequential (in terms of one layer being activated after another) version is conceptually straightforward, but we have not experimented with it yet.

2.1. Fully-connected models

Similarly to the way dropout is described (Hinton et al., 2012), each node or block in a given layer has an associated Bernoulli distribution which determines its probability of being activated. We train a different policy for each layer \(l\), and parameterize it (separately of the neural network) such that it is input-dependent. For every layer \(l\) of \(k\) units (or blocks of units), we define a policy as a \(k\)-dimensional Bernoulli distribution:

\[
\pi^{(l)}(u | s) = \prod_{i=1}^{k} \sigma_i^{u_i} (1 - \sigma_i)^{(1-u_i)}
\]

where \(\sigma_i = [\text{sigm}(Z^{(l)}s + d^{(l)})]_i\) denotes the participation probability, to be computed from the activations \(s\) of the layer below and the policy parameters \(\theta_l = \{Z^{(l)}, d^{(l)}\}\). We denote the sigmoid function by sigm, the weight matrix by \(Z\), and the bias vector by \(d\). This mask \(u\), sampled from the probabilities \(\sigma\), is multiplied element-wise with the output of the same hidden layer, \(s\), to become \((s \otimes u)\). For clarity we did not superscript \(u\), \(s\) and \(\sigma\) with \(l\), but this superscript should be considered implicit.

2.2. Convolutional models

The natural extension of the fully-connected parameterization to the convolutional network case is to output a dropout probability for each channel, or block of channels. However, this could be computationally expensive in the backwards pass, and might not result in significant performance gains. Instead, we propose a simpler model that requires fewer predictions, where the policy learns to drop out layers of the convolutional network. Although there are several ways to achieve this, we explore methods where the policy is parameterized by a second, smaller convolutional network, called the policy network. The output of this convolutional network is a vector representing the \(n_L\)-dimensional Bernoulli distribution for layer dropout in the primary recognition network, where \(n_L\) is the number of layers.

More specifically, rather than setting \(\sigma_i = [\text{sigm}(Z^{(l)}s + d^{(l)})]_i\), we have \(\sigma_i = [g(x)]_i\), where \(g(\cdot)\) denotes the application of a convolutional neural network, \(x\) denotes the input (such as an image), and the subscript \(l\) denotes the activation probability for layer \(l\). In order to allow for the skipping of layers in the network, we impose the additional constraint on \(g(\cdot)\) that most of the convolutional layers have the same input and output shape dimensions. In this way, it is possible to skip layers by simply feeding the input of the previous layer directly into the next layer. Note that we also explored an alternative architecture that is closer to the fully-connected case, where a single-variable Bernoulli for dropout at the next layer is computed based on the recognition network activations at the previous layer. However, the speedups were not as significant compared to having a separate policy network.

The idea of dropping out layers in a convolutional network is inspired by (Huang et al., 2016), who train deep networks with over 1000 layers using layer-wise dropout. However, similar to regular dropout, this is a form of ‘unconditional computation’, and is thus unlikely to learn specialized data-dependent pathways for improving efficiency at test time.

3. Learning dropout policies

We use REINFORCE (Williams (1992), also see the appendix of Bengio et al. (2015)) to learn the parameters of our policies. REINFORCE, also known as the likelihood-ratio method, is a policy search algorithm which uses gradients to improve a given parametrized policy.

The REINFORCE estimator of the gradient with respect to some minibatch loss \(\mathcal{L}\) can be computed as:

\[
\nabla_\theta \mathcal{L} \approx \nabla_\theta \sum_{i=1}^{m_b} (c_i - b) \log \pi_\theta(u_i | x)
\]

Where \(m_b\) is the minibatch size, \(c_i\) is the loss incurred by
the $i$th example, $u_i$ is the sampled action for the $i$th example, $x$ the input, and $\pi_0(u_i|x)$ the probability of the sampled action given the current policy. $b$ is a scalar baseline, and serves to reduce variance in the estimator without incurring bias. We simply use an exponential moving average and serves to reduce variance in the estimator without incurring bias. We simply use an exponential moving average with decay 0.99.

3.1. Sparsity regularization

In order to favour activation policies with sparse actions, we add two penalty terms that depend on a desired target sparsity rate $\tau$. The first term pushes the policy distribution $\pi$ to activate parameters with probability $\tau$ in expectation over the data. The second term pushes the policy distribution to have the desired sparsity of activations for each example.

$$L_a = \sum_{j} \left\{ \mathbb{E}\{\sigma_j\} - \tau \right\}_2 + \mathbb{E}\left\{ \left\| \frac{1}{k} \sum_{j} \sigma_j \right\|_2 \right\}$$

For a low $\tau$, a valid configuration would be to learn a few high probability activations for some part of the data and low probability activations for the rest of the data, resulting in activation probability $\tau$ in expectation.

3.2. Gradient Descent

Learning of the policies and the main network is done simultaneously. Updates to the weights are computed using the gradient with respect to the weights of both the loss and the REINFORCE term (2):

$$\nabla_{\theta} \mathcal{L} = \nabla_{\theta} \left[ c(x) + \lambda_p (\hat{c}(x) - b) \log \pi_0(u|x) + \lambda_s L_s \right]$$

Where $c(x)$ is the loss of example $x$ (e.g. the cross entropy or MSE), $\hat{c}$ the same but treated as a constant term when differentiating, $\lambda_p$ the ‘relative learning rate’ of the policy and $\lambda_s$ the weight of sparsity regularizations.

3.3. Block activation policy

To achieve computational gain when using fully-connected networks, instead of activating single units in hidden layers, we activate contiguous (equally-sized) groups of units together (independently for each example in the minibatch), thus reducing the action space as well as the number of probabilities to compute and sample. In other words, we compute $(H \otimes M_H)W \otimes M_O$, where $M_H$ and $M_O$ are binary mask matrices. $M_O$ is obtained for each layer from the sampling of the policy as described in eq. 1: each sampled action (0 or 1) is repeated so as to span the corresponding block. $M_H$ is simply the mask of the previous layer. This allows us to quickly perform matrix multiplication by only considering the non-zero output elements as well as the non-zero elements in $H \otimes M_H$.

<table>
<thead>
<tr>
<th>layer</th>
<th># channels</th>
<th>image $w, h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>raw RGB input</td>
<td>3</td>
<td>32</td>
</tr>
<tr>
<td>$3 \times 3$ conv, str. 2</td>
<td>$n_L$</td>
<td>16</td>
</tr>
<tr>
<td>$2 \times 2$ max pool</td>
<td>$n_L$</td>
<td>8</td>
</tr>
<tr>
<td>$3 \times 3$ conv, str. 2</td>
<td>$n_L$</td>
<td>3</td>
</tr>
<tr>
<td>$3 \times 3$ conv</td>
<td>$n_L$</td>
<td>1</td>
</tr>
<tr>
<td>fully-connected sigmoid</td>
<td>$n_L$</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 1. Convolutional policy network architecture

<table>
<thead>
<tr>
<th>layer</th>
<th># channels</th>
<th>image $w, h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>raw RGB input</td>
<td>3</td>
<td>32</td>
</tr>
<tr>
<td>$3 \times 3$ conv, str. 2</td>
<td>96</td>
<td>16</td>
</tr>
<tr>
<td>$N$ droppable layers:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$3 \times 3$ conv</td>
<td>96</td>
<td>16</td>
</tr>
<tr>
<td>$2 \times 2$ max pool</td>
<td>96</td>
<td>8</td>
</tr>
<tr>
<td>dropout(0.5)</td>
<td>96</td>
<td>8</td>
</tr>
<tr>
<td>$3 \times 3$ conv, str. 2</td>
<td>192</td>
<td>8</td>
</tr>
<tr>
<td>$2N$ droppable layers:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$3 \times 3$ conv</td>
<td>192</td>
<td>8</td>
</tr>
<tr>
<td>dropout(0.5)</td>
<td>192</td>
<td>8</td>
</tr>
<tr>
<td>$3 \times 3$ conv</td>
<td>96</td>
<td>8</td>
</tr>
<tr>
<td>mean pooling</td>
<td>96</td>
<td>1</td>
</tr>
<tr>
<td>fully-connected softmax</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2. Convolutional recognition network architecture

4. Experiments

4.1. Model implementation

The proposed model was implemented within Theano (Bergstra et al., 2010), a standard library for deep learning and neural networks. In addition to using optimizations offered by Theano, we also implemented specialized matrix multiplication code for the operation exposed in section 3.3. A straightforward and fairly naive CPU implementation of this operation yielded speedups of up to 5-10x, while an equally naive GPU implementation yielded speedups of up to 2-4x, using sparsity rates of under 20% and acceptable matrix and block sizes. To ‘skip’ convolutional layers, we simply use the set_subtensor theano operation, only modifying the feature maps of examples when they activate a layer. The overhead of such an operation is minimal relative to the cost of performing convolutions – for example, we find that only evaluating $k$ layers out of $n$ on average takes roughly the same amount of time for a fixed $k$ and a varying $n$.

We otherwise use fairly standard methods for our neural networks. The weight matrices are initialized by normalizing the weights so that the activations of a sample minibatch has 0 mean and unit variance through the layers. We

---

1Implementations used in this paper are available at http://github.com/bengioe/condnet/
use a constant learning rate and use Adam (Kingma & Ba, 2014) with a momentum term of 0.9. Our fully connected models use tanh activation units (ReLU activations offer similar performance), while our convolutional models use leaky ReLUs (with a learnable negative slope).

**Convolutional network parameterization** Our current convolutional policy parameterization consists of two separate policy networks: one that predicts the dropout probabilities for the first \(N\) layers, and the second that predicts them for the final \(2N\) layers. Each network has the same architecture (detailed in figure 1), but learns different parameters. This allows for greater freedom in treating these layers separately, which is intuitively beneficial as lower layers often compute the same features (e.g. edges and corners) regardless of input, and thus need to be dropped out less often. This is partially motivated by previous work (Huang et al., 2016), where early layers are given a higher chance of being active than the later layers. The architecture of the recognition network is shown in figure 2.

### 4.2. Model evaluation

#### 4.2.1. FULLY-CONNECTED MODELS

A thorough evaluation of fully-connected models with block-policy-dropout is presented in (Bengio et al., 2015), which we summarize here.

![Figure 3. MNIST fully-connected results. (a,b,c), probability distribution of the policy, each example’s probability (y axis) of activating each unit (x axis) is plotted as a transparent red dot. Redder regions represent more examples having that probability. Plot (a) is for class ‘0’, (b) for class ‘1’, (c) for all classes. (d), weight matrix of the policy.](image)

We first evaluated the performance of our fully-connected model on the **MNIST** digit dataset. What we find is that the policies learned to separate the input space, i.e. activate different blocks given the nature of the input. This can be seen in figure 3.

Next, we considered the performance of our model on the **CIFAR-10** (Krizhevsky & Hinton, 2009) image dataset. Summarized in figure 4, we showed that it is possible to achieve similar performance with our model (denoted condnet) as with a normal neural network (denoted NN), yet using sensibly reduced computation time. However, the error is still quite high for all models compared to the state-of-the-art of .072 (Salimans & Kingma, 2016). Finally we tested our model on the Street View House Numbers (SVHN) (Netzer et al., 2011) dataset, which also yielded encouraging results (figure 5).

<table>
<thead>
<tr>
<th>model</th>
<th>test error</th>
<th>(\tau)</th>
<th>#blocks</th>
<th>(k)</th>
<th>test time</th>
</tr>
</thead>
<tbody>
<tr>
<td>condnet</td>
<td>.514</td>
<td>1/16</td>
<td>16,32</td>
<td>16</td>
<td>1.4s (5.7( \times ))</td>
</tr>
<tr>
<td>condnet</td>
<td>.497</td>
<td>1/16</td>
<td>10,10</td>
<td>64</td>
<td>2.0s (5.3( \times ))</td>
</tr>
<tr>
<td>NN</td>
<td>.546</td>
<td>-</td>
<td>128,128</td>
<td>1</td>
<td>2.31s</td>
</tr>
<tr>
<td>NN</td>
<td>.497</td>
<td>-</td>
<td>480,480</td>
<td>1</td>
<td>8.34s</td>
</tr>
</tbody>
</table>

**Figure 4.** CIFAR-10 fully-connected results. condnet: our approach, NN: Neural Network without the conditional activations. \(k\) is the block size. ‘test time’ is the time required to do a full pass over the test dataset using the implementation, on a CPU, running on a single core; in parenthesis is the speedup factor compared to runtime without the specialized implementation (3.3).

For all three datasets and all fully-connected condnet models presented, the required training time was slightly higher than non-conditional models, but still reasonable: on average 1.5 to 3 times longer (wall time). All computations were performed on a CPU, running on a single core, and thus not optimized for fast training time, since the motivation for our work is to achieve speed-ups in run-time deployment of neural networks.

#### 4.2.2. CONVOLUTIONAL MODELS

We conduct preliminary results for our convolutional model on the **CIFAR-10** dataset, shown in figure 6. We can first see that the sparse models can learn to be appreciably accurate compared to dense models for a much lighter computation time. While this is true for the fully-connected case as well, we now show that this principle holds in a setting where the results are much closer to the state-of-the-

<table>
<thead>
<tr>
<th>model</th>
<th>error</th>
<th>(\tau)</th>
<th>#blocks</th>
<th>(k)</th>
<th>test time</th>
</tr>
</thead>
<tbody>
<tr>
<td>condnet</td>
<td>.183</td>
<td>1/11</td>
<td>13,8</td>
<td>16</td>
<td>1.5s (1.4( \times ))</td>
</tr>
<tr>
<td>condnet</td>
<td>.139</td>
<td>.04/17</td>
<td>27,7</td>
<td>16</td>
<td>2.8s (1.6( \times ))</td>
</tr>
<tr>
<td>condnet</td>
<td>.073</td>
<td>1/22</td>
<td>25,22</td>
<td>32</td>
<td>10s (1.4( \times ))</td>
</tr>
<tr>
<td>NN</td>
<td>.116</td>
<td>-</td>
<td>288,928</td>
<td>1</td>
<td>4.8s</td>
</tr>
<tr>
<td>NN</td>
<td>.100</td>
<td>-</td>
<td>800,736</td>
<td>1</td>
<td>10.7s</td>
</tr>
<tr>
<td>NN</td>
<td>.091</td>
<td>-</td>
<td>1280,1056</td>
<td>1</td>
<td>16.8s</td>
</tr>
</tbody>
</table>

**Figure 5.** SVHN fully-connected results.
Reinforcement learning of conditional computation policies for neural networks

<table>
<thead>
<tr>
<th>model</th>
<th>test error</th>
<th>N</th>
<th>$\tau$</th>
<th>test time</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv-condnet</td>
<td>.157</td>
<td>4</td>
<td>0.5</td>
<td>1.03s</td>
</tr>
<tr>
<td>conv-condnet</td>
<td>.167</td>
<td>4</td>
<td>0.3</td>
<td>0.84s</td>
</tr>
<tr>
<td>conv-condnet</td>
<td>.176</td>
<td>4</td>
<td>0.2</td>
<td>0.66s</td>
</tr>
<tr>
<td>conv-condnet</td>
<td>.173</td>
<td>2</td>
<td>0.5</td>
<td>0.58s</td>
</tr>
<tr>
<td>conv-NN</td>
<td>.159</td>
<td>4</td>
<td>-</td>
<td>1.07s</td>
</tr>
</tbody>
</table>

Figure 6. CIFAR-10 convolutional model results. conv-condnet is our approach, conv-NN is a convolutional network with no layers dropped out. Time measured on a NVIDIA Titan X GPU.

We can also see that as we limit the capacity of the models (by increasing sparsity $\tau$ or decreasing number of units), conv-condnets retain acceptable performance with low run times.

We examine the policy activation patterns for two different policy learning rates in figure 7. The small policy networks are capable of separating the input space and activating different layers for different inputs, as shown by the different mean activations for different classes (represented by colour). We also found that early layers tended to be active more often, which squares with our earlier intuitions and partially justifies our dual policy networks.

### 4.2.3. Regularization

We notice that regularization plays a strong role in our results, and that it helps REINFORCE in finding suitable policies. Using only the sparsity regularization, or only REINFORCE, leads to policies that perform poorly.

The particular choice of penalty, $L_s$, also seems to lead to more stable policies (less saturated and uniform), compared to penalizing the overall distance between the average policy and $\tau$, or to adding computation time (or some proxy of it) in the cost used in REINFORCE to train the policies.

### 5. Related work

There exist many approaches that achieve some form of input-dependent decision-making that impacts the model’s computation (Ba & Frey, 2013; Bengio et al., 2013; Stollenga et al., 2014), however only few of them actually make potential improvements towards computation time (Denoyer & Gallinari, 2014; Léon & Denoyer, 2015; Ioannou et al., 2016). As far as we are aware, we are the first to report significant computation-time improvements over both CPU and GPU.

Another point of comparison for our work are attention models (Mnih et al., 2014; Gregor et al., 2015; Xu et al., 2015). These models typically learn a policy, or a form of policy, that allows them to selectively attend to parts of their input sequentially, in a visual 2D environment. Both attention and conditional computation share a goal of reducing computation times, although the overhead of visual attention as it is currently is significant.

### 6. Conclusion

We tackle the problem of conditional computation in deep networks by using reinforcement learning. We present methods for learning policies that drop blocks of nodes in fully-connected networks, and propose novel dropout policy parameterizations that apply to convolutional networks. While both fully-connected and convolutional models achieve test time speedups with comparable or improved test error compared to their dense counterparts, the convolutional models achieve results much closer to the state-of-the-art on CIFAR-10. Further, in all models we allow for an error/runtime trade-off by changing the value of a single sparsity parameter $\tau$. More interestingly still, it seems that the policy networks accurately learn to exploit the structure of the primary recognition networks, and distribute computations patterns over the input space.

The use of REINFORCE could be replaced by a more efficient policy search algorithm, or by one where rewards (or costs) as described above are replaced by a sequential variant. The more direct use of computation time as a cost may prove beneficial. In general, we consider conditional computation to be an area in which reinforcement learning could be very useful, and deserves further study.

Figure 7. CIFAR-10. Similarly to figure 3, the activation probability of a layer (y axis) plotted against the layer number (x axis). Each colour denotes a different input object class, and each faded point of that colour denotes an example from that class (mean class probabilities shown as bold dots). The policy learns different layer dropout probabilities for different classes, and becomes quite confident for larger learning rates.
References


Xu, Kelvin, Ba, Jimmy, Kiros, Ryan, Courville, Aaron, Salakhutdinov, Ruslan, Zemel, Richard, and Bengio,