ShrinkNets

Guillaume Leclerc[*]
Massachussets Institute of Technology
Cambridge, Massachussets, USA
leclerc@mit.edu

Raul Castro Fernandez
Massachussets Institute of Technology
Cambridge, Massachussets, USA
raulcf@csail.mit.edu

Samuel Madden
Massachussets Institute of Technology
Cambridge, Massachussets, USA
madden@csail.mit.edu

1 INTRODUCTION

When designing Neural Networks, finding the appropriate size (width and depth) is key. In particular, these hyper parameters have a strong correlation with over/underfitting **Sam:** give reference **Gl:** I did some litterature review and actually this statement is quite controversial, should we rephrase ?. The main problem is that we have no reliable way to find them [ref]. Decades of experimentation led to some heuristics [ref] that try to prune that immense space of possible network sizes. For example, researchers have used strategies such as random search **Gl:** Should we also move the references here if we remove the pargraph from the related work ?, meta-gradient descent and parser trees to reduce find good parameters without exhaustively exploring the entire hyper-parameter space. Although these strategies help with finding a good set of hyper-parameters, they still require a compute-intensive search of the space.

In this paper we present a method to automatically find an appropriate network size from a single parameter, drastically reducing the hyper-parameter dimensionality. The key idea is to *learn* the right network size at the same time that the network is learning the main task. For example, for an image classification task, with our approach we can provide the training data to a network—without sizing it a priori—and expect to end up with a network that has learnt the task without overfitting **Gl:** too strong, we do not really have any guarantee on the generalization, how about changing it to: to end up with a network that learnt a tradeoff between size and accuracy. This approach has two main benefits. First, we do no longer need to choose a network size before training. Second, the final network size will be appropriate for the task at hand, and not larger **Gl:** Should we talk about the fact that it automatically gets rid of dead neurons ?. This is important because oversized networks have a lower inference throughput and higher memory footprint.

Our approach has two main challenges. First, on how to dynamically size the network during training. Second, on how to find a loss function that optimizes for the additional task of sizing, without deteriorating the learning performance of the main task. We describe next ShrinkNets, which cope with both challenges:

2 SHRINKNETS

Our approach consists of starting the training process for the task of interest with an explicitly oversized network. Then, as training progresses, we learn which neurons are not contributing to learning the main task and remove them dynamically, thus shrinking the network size. This method requires two building blocks. First, a way of identifying neurons that are not contributing to the learning process, and second a way of balancing the network size and the generalization capability for the main task. We introduce a new layer, called Filter, which takes care of *deactivating* neurons. We also modify existing loss functions to incorporate a new term that takes care of balancing network sizing and generalization capability appropriately. We explain them next:

Filter Layers: They have weights in the range $[0, +\infty]$ and are placed after linear and convolutional layers of traditional deep networks **Gl:** This is how I used it but with the new implementation we can be more creative and put them anywhere, should we say "usually placed" insdead ?. The *Filter Layer* takes an input of size $(B \times C \times D_1 \times \cdots \times D_n)$, where B is the batch size, C the number of features (or channels), and D any additional dimension. This structure makes it compatible with fully connected layers with $n = 0$ or convolutional layers with $n = 2$. Their crucial property is a parameter $\theta \in \mathbb{R}^C$, defined as follows:

$$Filter(I; \theta) = I \circ \max(0, \theta) \tag{1}$$

Where $\circ$ is the Hadamard product (pointwise multiplication **Gl:** Should we keep only one of them to be more concise ?), and θ is expanded in all dimensions except the second one to match the input size. It is easy to see that if for any k, if $\theta_k \leq 0$, the k^{th} feature/channel will forever be 0. When this happens, we say the Filter layer disables the neuron. These disabled neurons/channels can be removed from the network without changing its output (we describe how we perform this removal below). We explain next how the weights of the Filter Layer are adjusted during training.

Training Procedure: Once Filter layers are placed in a deep network, we could train it directly and it would be equivalent to a normal neural network. However, our goal is to find the smallest network with reasonable performance. We achieve that by introducing sparsity in the parameters of the *Filter Layers*. Indeed, having a negative component in the θ parameter of the filter layer permamently disable its associated feature **Gl:** Maybe redundant ? we talked about that in the previous paragraph . To obtain this sparsity, we simply redefine the loss function:

$$L'(x, y; \theta) = L(x, y) + \lambda|\theta| \tag{2}$$

This choice come from the Lasso loss [ref] and we use it for the same reason: it introduces sparsity. The bigger the λ the more entries in θ are set to zer, and since zeros entries in θ correspond to dead neurons, lambda effectively control the number of neurons/channels in the entire network. Next, we explain how to implement ShrinkNets efficiently.

[*]Visiting Student

Software architecture[1] **:***Filter Vectors* can easily be implemented in a few lines of code in many existing deep learning frameworks. However, in ShrinkNets we assume that we start with obviously oversized networks. If disabled neurons are not quickly removed, the overhead might caused the training process to be significantly slower than classic neural networks. This is why we want to support what we call garbage collection: removing the weights in the layers that are responsible(or using) features that are disabled.

To be able to catpure this relationship between layers we agumented *Pytorch* (the framework we used as the foundation of our implementation) with a graph structure very similar to the one available in Keras [ref]. In this graph edges are effectively event hubs responsible for propagating *feature removal* events to subscribing layers. ShrinkNet layers are agumented to emit and react to these events but also propagate the event further if has an impact at the other endpoint of the layer (input or output).

This event-based implementation coordinated by edges makes it very easy to integrate new layers in the library. We even provide automatic wrapping for layers from pytorch if they have no internal state. For more complex ones, we provide utilitaries that makes the implementation very concise.

3 EVALUATION

3.1 Convergence

To demonstrate that the approach is viable, we will first show that Shrinking Networks converge properly. For this experiment we trained a one hidden layer neural network with one filter layer to control the number of hidden units. We initialized the models with 10000 neurons and trained them on `MNIST` using different regularization factors (λ). We summarized the results in Figure 1. We can see on this plot is that for a given λ the number of hidden units seems to always converge to some value. It has two implications: it seems that λ is, as we would think, a proxy of the network size (bigger λ imply smaller networks). Secondly, it indicates that the spikes we see in the regular spikes we see in the loss that occur when neurons are disabled will eventually disapear because the number of neurons reach a plateau. **Gl:** I feel we should give a conclusion for this paragraph but I am not sure what to say

3.2 Relevance in the context of hyper-parameter optimization

One of our main goal is to help finding neural network architectures that perform reasonably well faster than existing techniques. For simplicity, we will try to see how ShrinkNets perform when doing random search. However, we believe that the results generalize to more complex methods, most of them might actually benefit a lot from the reduction in the dimensionality of the hyper-parameter space.

To isolate the effect of ShrinkNets we will consider very simple architectures We will be using multi-layer perceptrons (with three hidden layers) and convolutional neural networks (With the LeNet-5 architecture). For this experiment we will assume we have

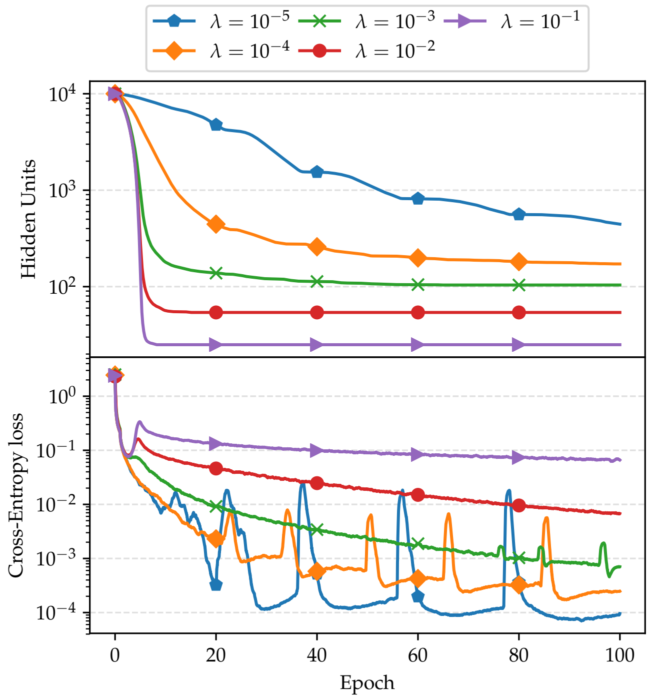

Figure 1: Evolution of the number of hidden units and loss over time on the `MNIST` dataset for different λ values

no information about the size except an upper bound of 50 channels per convolutional layer and 5000 neuron for fully connected layers. We try to find the appropriate size of the network using classical neural networks and ShrinkNets. To make sure that we are as fair as possible and since one might argue that ShrinkNets is also regularizing the network in addition to finding the size, we will also considered regualrized classic networks using the most commonly used technique the L_2 penalty [ref].

The experiment consist in sampling parameters randomly for each class of network, train them for the same amount of epochs (100 for MLPs and 200 for CNNs). Pick the epoch that performed the best on the validation accuracy and evaluate it on the testing set. We repeat the process for 50 models. This evaluation was done on `MNIST` [ref], `FashionMNIST` [ref] and `CIFAR10` [ref]. The distribution of the testing accuracy we obtained is summarized on Figure 2.

We can see that the distributions for ShrinkNets are consistently better than the others for all datasets and both architectures. It shows that by training fewer networks, we are likely to obtain onpar or even better models which effectively reduces the cost of hyper-parameter optimization. In some cases the best ShrinkNets models outperforms the best model of the other techniques.

4 RELATED WORK

There are many methods in the litterature that try to simplify the neural network. Most of them focuses on removing connections

[1]The code is available as Python/PyTorch library on http://github.com/mitdbg/fastdeepnets **Gl:** Should we rename the repository ?

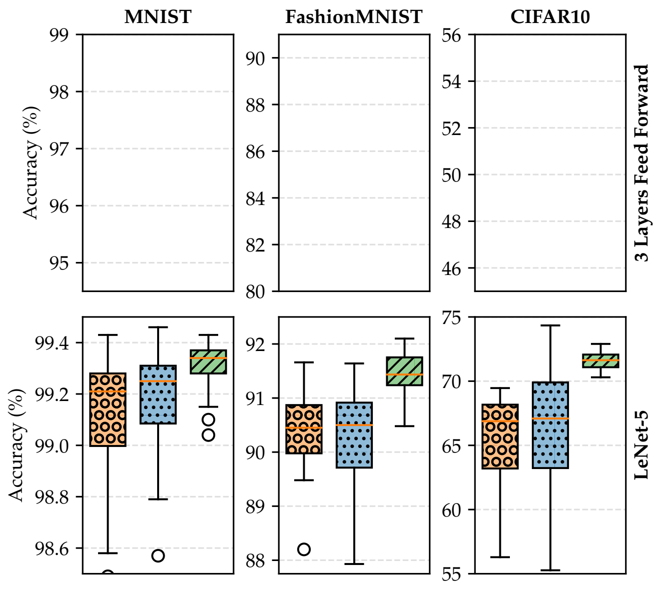

Figure 2: Distributions of the testing accuracy for different training methods, datasets and architectures using random search

(eg: [ref], [ref]). However, most hardware (GPU's, TPUS, TensorCores) do not really leverage sparsity in the connections efficiently. On the other side, ShrinkNets and some others [ref] and [ref] try to remove entire neuron instead of connections. This is very usefull because it reduce the size of the matrices, producing speed-up on every device. ShrinkNets improves on [ref] because it removes neurons during training, speeding up the rest of the process and beats [ref] on convergence speed and because it does not need to change the optimizer. To the best of our knowledge this is also the first contribution that tries to learn the number of channels of a convolutional neural network.

5 CONCLUSION

In this paper we presented a novel technique to guess a resonable network size based on single parameter λ that control the tradeof between loss and the size. We demonstrated that it works both on fully connected networks and convolutional neural networks. Even though the firsts results seem promising, there are many ways we could improve. In the current implementation we only "learn" the number of features (neurons or channels). We could try to augment it with dynamic number of layers as seen in [ref] to be able to determine the entire architecture.

We saw on Figure 1 that the loss temporarily suffers from the removal of neurons. It is likely that the loss would be more stable if the number of neurons converged faster or neurons disappeared slower. For this reason we plan to explore proximal gradient methods to optimize the filter vectors and/or randomize neuron removals.

During our evaluation we picked small datasets mainly to be able to train many models and have statistically significant distributions. With more computation resources and time, we could see if it generalizes to bigger datasets and other architectures like `ResNet` [ref] (small modifications to the existing code base are required to support them)