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

ABSTRACT

Write the abstract at the end !

1 INTRODUCTION

When designing Neural Netowrks, finding the apropriate size (width and depth) is key. Indeed, these hyper parameters have a strong correlation with over/underfitting. 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, but we are still bound to use costly and sometimes complex methods to find reasonable values for these hyper-parameters.

2 RELATED WORK

In the litterature we can see different ways of approaching this issue. The first one is to select parameters, train the model, evaluate it, repeat and pick the best one. There are many different ways of selecting parameters: random search [ref], grid search [ref], meta gradient descent [ref], parsen trees[ref], gaussian processes [ref] etc... but they all suffer from one major drawback: we have to train and evaluate countless models; and even if some algorithms allow parallel evaluations [ref] to reduce the overall training time, the hardware and energy cost is still important.

To reduce the number of models trained, another approach emerged: train a slightly bigger model and after convergence, remove as many parameters as possible without impacting the performance of the model. Notable contributions are Optimal Brain Dammage [ref], Deep Compression [ref], and Group sparsity [ref] (this one is especially related to this article). These techniques are very interesting but they still require a reasonable network size to start with, so they usually have to be combined with classic hyper-optimization techniques.

However, some recent contributions like Non-Parametric Neural Netowrks [ref] and [ref] try to learn the network size (width for the former and depth for the latter) during the training process and without any prior assumption about the network size.

3 THE FILTER LAYER

3.1 Motivation

The method described in [ref] grows and shrink the network over time. Though it seems to be an attractive property to have, during our experiments and according to their results, models are very slow to train and sometimes converge to suboptimal solutions. Their method also required a new optimizer *AdaRad*. Our goal was to provide a solution that can easily be integrated in existing machine learning systems and provide similar convergence speed and accuracy. Therefore, designing a new layer that only allow seemed to be best approach.

3.2 Definition

In this section we define the *Filter Layer*. It takes an input of size $(B \times C \times D_1 \times \cdots \times D_n)$. So it is compatible with fully connected layers with $n = 0$ or convolutional layers with $n = 2$. This layer has a parameter $\theta \in \mathbb{R}^C$ and is defined the following way.

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

Where $\circ$ is the Hadamard product (pointwise multiplication), 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. We can use this property to devise a training procedure.

3.3 The training procedure

To train networks we need start with a clearly oversized network, then we insert *Filter Layers* in the architecture (usually after every Linear or Convolutional layer except the last one) and we sample their weight from the Uniform$(0,1)$ distribution. We could train this network as it is and it would be equivalent to a normal neural netork. However, our goal is to find the smallest network with reasonable performance. We can achieve that by introducing sparsity in the parameters of the *Filter Layers*. Indeed, having a negative value is equivalent to zeroing an entire row and column in the surrounding layers. These disabled neurons/channels can be removed from the network without changing its output. To obtain this sparsity we simply redefine the loss function:

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

The λ parameter controls the tradeof between optimizing the original loss or the size of the network.

4 SOFTWARE ARCHITECTURE

To obtain good performance using this approach, it is key to remove neurons and channels from the network as soon as possible. This task usually is cubbersome using existing machine learning frameworks. To enable the reproducibility of our results and let people use this method on their own dataset and architectures, we implemented a small library on top of *PyTorch*. It makes it very easy to design and train networks that contains *Filter Layers*. We will briefly describe the software architecture.

The key operation we want to perform is warn layers when a feature is removed so they can resize themselves. Therefore, we need to know what are the child and parents of a given layers. Since there is no concept of computation graph in *PyTorch* we had to

[*]Visiting Student

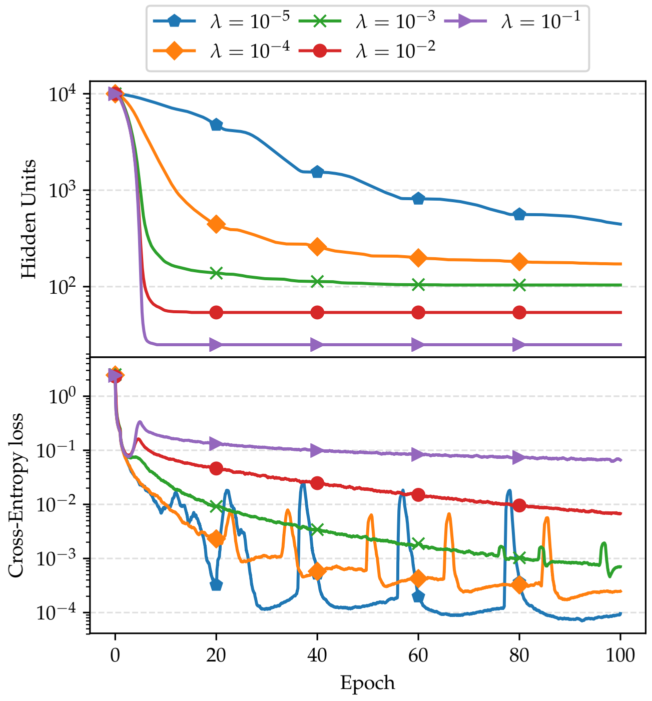

Figure 1: Evolution of the number of hidden units and loss over time on the `MNIST` dataset for different λ values

build it. Layers are augmented with a list of incoming modules (parents) and outgoing modules (children).

Edges in the graph carry features from parent to children. When a *Filter Layer* removes a feature, edges need to be able to warn layers so we implemented them as an event hubs that maintain the list of features to keep; we called them `FeatureBags`. In many cases, layers do not change the number of features (eg. *Dropout*, *BatchNorm*), therefore, events needs to propagate through them to warn all layers that share the same number of features. To solve this problem, such layers instantiate `MirrorFeatureBags` which role is to relay events to and from the root `FeatureBag`. With this event-base approach are able to keep all the layers in sync. Adding layers becomes straightforward: programmers just have to react to three events: input removal, output removal and garbage collection. During our tests we ran the garbage collection every epoch.

5 EVALUATION

5.1 Convergence

To demonstrate that the approach is viable, we will first show that Shrinking Networks converge. 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 on Figure 1. [Should we explain the results ?]

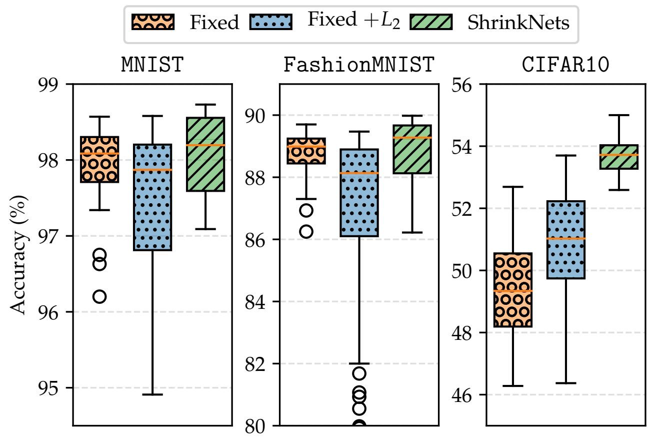

Figure 2: Distributions of the testing accuracy for different training methods, datasets and architectures using random search

5.2 Comparison between Shrinking and classic Networks

To demonstrate the value of ShrinkNets in the context of hyper-parameter optimization we set up the following experiment: We consider a *LeNet-5* architecture and assume we do not know the number of channels and neurons. We try to find the best architecture by performing random search on the parameters that define the size. For classic neural networks these parameters are the number of channels for the first two layers and the number of hidden units of the linear layer. For ShrinkNets there is only one parameter λ. We sampled 50 models of each and trained them, picked the best epoch using the validation accuracy and measured their accuracy on the testing set. Since ShrinkNets can be considered to be regularized in some sense, to make the comparison fair we also considered static networks with an $L2$ regularization factor (also drawn randomly). We summarized the accuracy we obtained on Figure 2. [Should we explain the results]

6 FUTURE WORK

Even though the firsts results this techique yields seem promising, there are many area that we could explore to improve it. 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 temporarly 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 disapeared more progressively. This is why we think we should 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 generalises to bigger datasets and other architectures like `ResNet` [ref] (small modifications to the existing code base are required to support them)

7 CONCLUSION

Do the conclusion at the end