ShrinkNets

Guillaume Leclerc*
Massachussets Institute of Technology
32 Vassar Street
Cambridge, Massachussets, USA 02139
leclerc@mit.edu

Raul Castro Fernandez
Massachussets Institute of Technology
32 Vassar Street
Cambridge, Massachussets, USA 02139
raulcf@csail.mit.edu

Samuel Madden
Massachussets Institute of Technology
32 Vassar Street
Cambridge, Massachussets, USA 02139
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. 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, 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.

Sam: In this paper, we describe ... – need to give a high level idea of solution, and its benefits. Would be good to give your system/approach a name.

In this paper we present a method to automatically find an appropriate network size without paying the compute and time cost of searching through a large hyper-parameter space. 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. 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. This is important because oversized networks have a lower inference throughput.

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

Sam: Describe our key approach at a high level – i.e., add a new layer to the network, called the *filter layer*, which ...

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 Layer:

Filter layers have weights in the range $[0, 1]$ and are placed after linear and convolutional layers of traditional deep networks. The *Filter Layer* 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 $n = 2$. Their crucial property is a parameter $\theta \in \mathbb{R}^C$, defined as follows: **Sam:** Need to define B, C, D.

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

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. 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.

Sam: Give an English description of what this formalism achieves – what does the filter layer do, why is it significant.

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, **Sam:** having a negative value where? having a negative value is equivalent to zeroing an entire row and column in the surrounding layers. To obtain this sparsity, we simply redefinte the loss function:

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

The λ parameter controls the trade-off between optimizing the original loss or the size of the network. **Sam:** Again give some intuitive description of what is novel / important about this choice of loss function and learning process. It's hard to understand what is interesting/important about what you have done without you explicitly calling it out.

Next, we explain how to implement ShrinkNets efficiently.

3 SOFTWARE ARCHITECTURE

ra: this can be shorter. I'd add it as a third paragraph above, immediately above the eval

*Visiting Student

To obtain good performance using this approach, it is essential to be able to remove neurons and channels from the network as quickly as possible. This task usually is cumbersome using existing machine learning frameworks. To enable this, we implemented a small library on top of *PyTorch* which 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 to notify layers when a feature is removed so they can resize themselves. Therefore, we need to track the child and parents each layer. Since there is no concept of computation graph in *PyTorch* we had to build it; specifically, 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 notify layers of the change. We implemented a simple event-based architecture where edges send events about features to layers. **Sam:** Suggest cutting the following: Each edge maintains a list of features to keep, and called a `FeatureBag`. In many cases, layers do not change the number of features (e.g., *Dropout*, *BatchNorm*), therefore, event needs to propagate through them to notify 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-based 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. **Sam:** What does garbage collection do? During our tests we ran the garbage collection every epoch.

4 EVALUATION

4.1 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 in Figure 1. [Should we explain the results ?] **Sam:** Yes, you need to explain in detail – what does this show? How does it prove that your approach works / is a good idea?

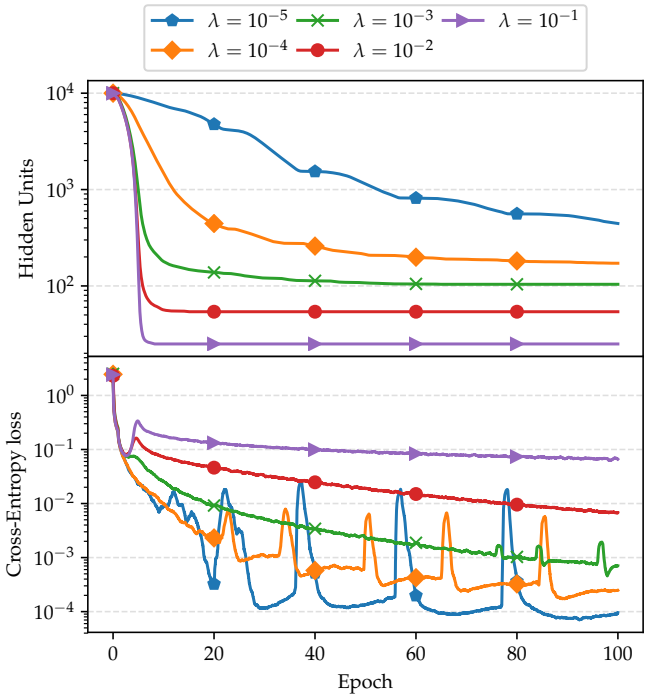

Figure 1: Evolution of the number of hidden units and loss over time on the `MNIST` dataset for different λ values

4.2 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 two different architectures, a three hidden layer feed forward network and a *LeNet-5* [ref]. We do not know the number of channels and neurons and we want 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 and/or neurons; for ShrinkNets there is only one parameter, λ. We sampled 50 models of each and trained them, picked the best epoch according to the validation accuracy and measured their performance 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) **Sam:** L2 regularization over what?. We summarized the accuracy we obtained on Figure 2. [Should we explain the results] **Sam:** Yes, need to describe in detail – how does this prove that your method works/ is a good idea? What are the key takeaways?

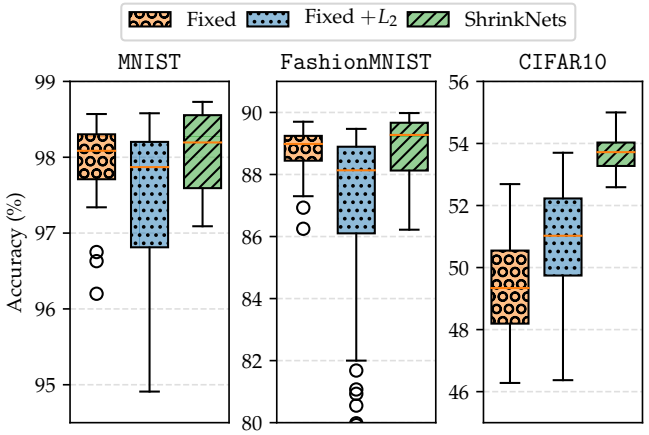

Figure 2: Distributions of the testing accuracy for different training methods, datasets and architectures using random search

5 RELATED WORK

ra: I'd remove this paragraph because we used it in the motivation already

In the literature 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 a very large number of models; and even if some algorithms allow parallel evaluations [ref] to reduce the overall training time, the hardware and energy cost is still very significant.

ra: I'd argue these techniques are aimed to improve inference time and not to avoid hyper-parameter optimization

To reduce the number of models trained, another approach is to 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 Damage [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.

Sam: How is this shrinking approach different than previous approaches like Optimal Brain Damage, Deep Compression,e tc?

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. **ra:** revise nextThis method dynamically grows and shrinks the network over time. Though this 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 requires a new optimizer *AdaRad*. **ra:** explain how difficult is to get the optimizer stable. that's a drawback of the method **ra:** Instead, ShrinkNets work out of the box with existing libraries and does not need any changes to the optimizer.

Sam: Give a short sentence about how our approach is different than these previous approaches.

6 FUTURE WORK

ra: this can be shorted, and used as conclusions

Even though the firsts results this technique 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 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)

7 CONCLUSION

Do the conclusion at the end