Learning Network Size While Training with ShrinkNets

Guillaume Leclerc, Raul Castro Fernandez, Samuel Madden
Massachussets Institute of Technology
[leclerc,raulcf,madden]@mit.edu

1 INTRODUCTION

When designing neural networks, one of the key parameters is the network size, i.e., the number of layers and neurons per layer. Choosing these parameters appropriately can dramatically affect performance, yet there is no reliable way to efficiently set them. Although many search strategies and heuristics [1] have been proposed, including random search, meta-gradient descent [11], Gaussian processes [2], and Parzen Estimators [2], they generally require a compute-intensive search of parameter space.

In this paper we present a method to automatically find an appropriate network size, drastically reducing optimization time. 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 learned to classify images with an accuracy similar to a the best manually engineered network. Our approach has two main benefits. First, we no longer need to choose a network size before training. Second, the final network size will tuned to be appropriate for the task at hand, and not larger. This is important because over-sized networks have a lower inference throughput and higher memory footprint.

Our approach has two main challenges. First, we need a way to dynamically size the network during training. Second, we need a loss function that optimizes for the additional task of sizing, without deteriorating the learning performance of the main task. Our approach, called ShrinkNets, copes with both challenges.

2 SHRINKNETS

During training, our approach starts with an explicitly over-sized network. As training progresses, we learn which neurons are not contributing to learning and remove them dynamically, effectively shrinking the network. This method requires two key components: first, we need a way to identify neurons that are not contributing to the learning process, and second we need a way to balance the network size and the generalization capability for the main task. We introduce a new Filter layer that takes care of *deactivating* neurons. We also modify existing loss functions to incorporate a new term that takes care of balancing network size and generalization capability appropriately.

Filter Layers: Filter layers have weights in the range $[0, +\infty]$ and are usually placed after linear and convolutional layers. 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, in the case of convolutional layers), 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 $\theta \in \mathbb{R}^C$. The output is defined as follows:

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

Where $\circ$ is the pointwise multiplication, and θ is expanded in all dimensions to match the input size (except the second one since they are equal by definition). It is easy to see that if for any k, if $\theta_k \leq 0$, the k^{th} input feature/channel is multiplied by zero and have no influence on the output. If this happens, we say the Filter layer deactivates the neuron. These disabled neurons/channels can be removed from the network without changing its output. Before explaining how that is achieved, we explain next how the weights of the Filter Layer are initialized and adjusted during training.

Training Procedure: Once Filter layers are placed in a network and initialized (sampled from the Uniform[0, 1] distribution), we could train the network directly using our standard loss function, and we could achieve performance 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*, thus forcing the deactivation of neurons. To obtain this sparsity, we simply redefine the loss function:

$$L'(x, y; \theta) = L(x, y) + \lambda |\max(0, \theta)| \tag{2}$$

The additional term $\lambda|\max(0, \theta)|$ introduces sparsity (see Lasso loss [14]). The second component of the loss increases the gradient with respect to θ, thus pushing its value towards zero. Neurons with little impact on the original loss (gradient lower than λ), will not be able to compete against this attraction towards zero. Because the entries in θ with a value of 0 or less correspond to dead neurons, λ effectively controls the number of neurons/channels in the entire network. We introduced the $\max(\dots)$ into the loss to make sure that neurons are permamently disabled when performing gradient descent based optimization. Next, we explain how to implement ShrinkNets efficiently.

Dynamic Network Resizing It is possible to reduce the overhead of the training process by removing neurons as soon as they become deactivated by θ going to 0. To do this, we implemented a *neural garbage collection* mechanism which prunes deactivated neurons on-the-fly, reducing the processing time and memory overhead. To support this feature, it is crucial to understand the information flow between neurons and layers in the neural network. We achieve this by representing such information flow as a graph. Vertices represent layers, and edges are event-hubs responsible for propagating information about disabled neurons to the relevant layers.

3 EVALUATION

We implemented ShrinkNets, including the *garbage collection* mechanism on top of *PyTorch* [10], and we have made the software open source.[1]. We used used our implementation to evaluate two crucial aspects of ShrinkNets, which we present next.

[1]The code is available as Python/PyTorch library on http://github.com/mitdbg/fastdeepnets

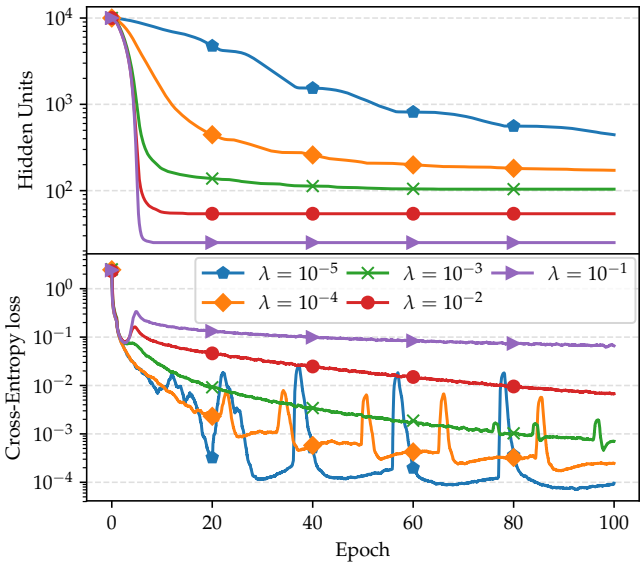

Figure 1: Evolution of the number of hidden units and loss over time on the `MNIST` dataset for different λ values

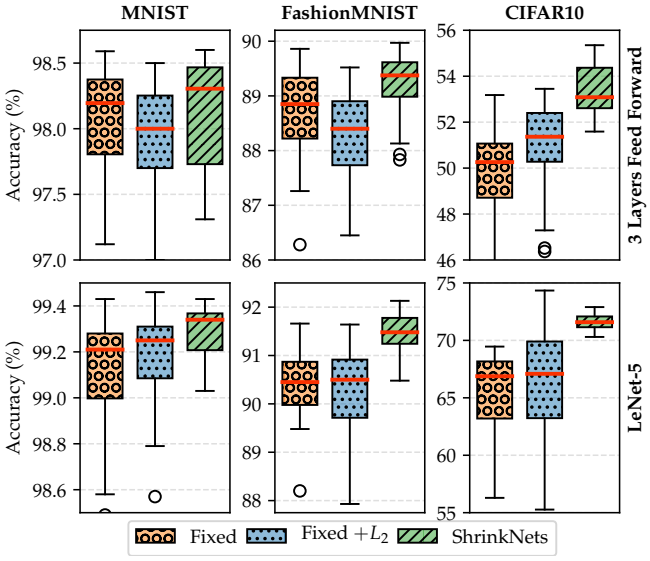

Figure 2: Distributions of testing accuracy for different training methods, datasets and architectures using random search

3.1 Does ShrinkNet converge?

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` [7] using different values for λ. Figure Figure 1 shows that λ works as a proxy for the network size, with bigger λ implying smaller networks. More importantly, the figure helps us confirm that despite the regular spikes—which are caused by the dynamic disabling of neurons—ShrinkNets eventually converge.

3.2 Does ShrinkNet find good networks?

In this experiment we want to find an appropriate network size for two network architectures, a multi-layer perceptron (MLP) and the *LeNet-5* model [7]. We have no prior information about the network size, but we set an upper bound of 50 channels per convolutional layer and 5000 neurons for fully connected layers. We then use random search to vary the parameters that determine the network size, i.e., λ in the case of ShrinkNets, the width of each layer in the case of `Fixed` and similar to previous but adding a L_2 penalty [9] (`Fixed-L2`). We train all variants on `MNIST` [7], `FashionMNIST` [15] and `CIFAR10` [6]. We use the same number of epochs, 100 for MLPs and 200 for *LeNet-5*. We select the epoch that performed the best on the validation set and evaluated it on the testing set. We repeat this process 50 times and show the distribution of the testing accuracy in Figure 2.

The results in the figure show that ShrinkNets finds better networks with fewer training iterations than the other methods, because it achieves higher median and less variable accuracies. Despite the simple nature of this experiment, we believe that the results generalize to more complex search methods; mainly because our solution reduces the dimensionality of the space they have to explore.

4 RELATED WORK

There are many methods in the literature that aim to simplify network structure. Most of them focuses on removing connections (eg: [3], [4]). On the other side, ShrinkNets and some others [13] and [12] try to remove entire neurons instead of connections. This is very useful because it reduces the size of the matrices, producing speed-up even on devices/libraries that only support dense matrices . ShrinkNets improves on [13] because it removes neurons during training, speeding up the rest of the process. Further it outperforms [12] on convergence speed and because it does not need to change the optimizer. To the best of our knowledge this is also the first work that tries to learn the number of channels in a convolutional neural network.

5 CONCLUSION

In this paper we presented a novel technique to guess a reasonable network size based on single parameter λ that control the tradeoff between loss and network size. We demonstrated that ShrinkNets works both on fully connected networks and convolutional neural networks. Although the initial results are promising, there are many additional avenues to explore. In the current implementation we only "learn" the number of features (neurons or channels). We plan to augment this with a dynamic number of layers as seen in [8] to be able to determine the entire architecture. Further, as shown in 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 more slowly. For this reason we plan to explore proximal gradient methods to optimize the filter vectors and/or randomize neuron removals. Finally, during our evaluation we picked small datasets mainly to be able to train many models and have statistically significant distributions. We plan to verify that our approach see if it generalizes to bigger datasets and other architectures like `ResNet` [5], which is possible with small modifications to our existing code base.

REFERENCES

[1] Yoshua Bengio. 2012. Practical recommendations for gradient-based training of deep architectures. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7700 LECTU (2012), 437–478. https://doi.org/10.1007/978-3-642-35289-8-26 arXiv:1206.5533

[2] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for Hyper-Parameter Optimization. In *Advances in Neural Information Processing Systems (NIPS)*. 2546–2554. https://doi.org/2012arXiv1206.2944S arXiv:1206.2944

[3] Yann Le Cun, John S Denker, and Sara a Solla. 1990. Optimal Brain Damage. *Advances in Neural Information Processing Systems* 2, 1 (1990), 598–605. https://doi.org/10.1.1.32.7223 arXiv:arXiv:1011.1669v3

[4] Song Han, Huizi Mao, and William J Dally. 2015. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. (2015). https://doi.org/abs/1510.00149/1510.00149 arXiv:1510.00149

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90 arXiv:1512.03385

[6] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images. *… Science Department, University of Toronto, Tech. …* (2009), 1–60. https://doi.org/10.1.1.222.9220 arXiv:arXiv:1011.1669v3

[7] Y LeCun, L Bottou, Yoshua Bengio, and P Haffner. 2001. Gradient-Based Learning Applied to Document Recognition. In *Intelligent Signal Processing*. 306–351. https://doi.org/10.1109/5.726791 arXiv:1102.0183

[8] Benjamin Meier. [n. d.]. Going Deeper: Infinite Deep Neural Networks. ([n. d.]). https://github.com/kutoga/going_deeper/raw/master/doc/going_deeper.pdf

[9] Andrew Y Ng. 2004. Feature selection, \ell_1 vs. \ell_2 regularization, and rotational invariance.. In *ICML*. 78–85. http://www.machinelearning.org/proceedings/icml2004/papers/354.pdf

[10] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[11] Fabian Pedregosa. 2016. Hyperparameter optimization with approximate gradient. (2016). arXiv:1602.02355 https://arxiv.org/pdf/1602.02355.pdfhttp://arxiv.org/abs/1602.02355

[12] George Philipp and Jaime G Carbonell. 2017. Nonparametric Neural Network. In *Proc. International Conference on Learning Representations*. 1–27. arXiv:1712.05440 https://www.cs.cmu.edu/{~}jgc/publication/NonparametricNeuralNe

[13] Simone Scardapane, Danilo Comminiello, Amir Hussain, and Aurelio Uncini. 2017. Group sparse regularization for deep neural networks. *Neurocomputing* 241 (2017), 81–89. https://doi.org/10.1016/j.neucom.2017.02.029 arXiv:1607.00485

[14] Robert Tibshirani. 1996. Regression Selection and Shrinkage via the Lasso. (1996), 267–288 pages. https://doi.org/10.2307/2346178 arXiv:13697412/11/73273

[15] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. (2017). arXiv:1708.07747 https://arxiv.org/pdf/1708.07747.pdfhttp://arxiv.org/abs/1708.07747