Self-Selecting Neural Networks

Ryan Kingery, Chidubem Arachie, Ahmadreza Azizi, and Ali Taleb Zadeh Kasgari
Virginia Tech, Blacksburg, VA
Emails:{rkingery,achid17,reza321,alitk}@vt.edu.

***Abstract*—In this proposal we describe the idea and motivation behind a greedy-based approach to neural network architecture selection that we call self-selecting neural networks. We explore various ways of doing neural net model selection via this approach on multilayer perceptrons with a single hidden layer, observing that it can in fact be feasible and efficient to do so via neuron addition and deletion in the hidden layer, though more work must be done before it becomes viable.**

I. Introduction

In recent years, the surprising success of deep learning has made many problems that were thought to be hard to solve using a computer much more tractable. The most remarkable successes thus far have been in the fields of computer vision, speech, natural language processing, and medicine. In these fields, deep learning techniques have started to compete with humans at performing many common tasks well, including image recognition, speech translation, and medical diagnosis.

The backbone of deep learning is the neural network. Neural networks are complex hierarchical models that come in many flavors, and can be used for just about any machine learning task conceivable. It is this complexity and versatility, however, that often makes them quite difficult to train. Relative to other machine learning models, neural networks have a huge number of hyperparameters that must be tuned in advance.

Some of the more crucial and annoying hyperparameters to tune deal with the network architecture. Depending on the specific implementation, such architecture-related hyperparameters may include the number of hidden layers, number of units per layer, types for each layer (e.g. dense, convolutional, max pooling, LSTM), types of activation functions, filter sizes, and, of course, the order in which each of these things should be put together within the neural network. Clearly, expecting practitioners to efficiently choose which network architecture to use given a particular situation is asking a lot. It often requires them to have a great deal of domain specific expertise on top of experience training neural networks.

The goal of this research is to try to address this neural architecture selection problem. Namely, our goal is to create a routine for performing model selection on a class of neural networks. To do so we initially focus on performing model selection on the class of multilayer perceptron (MLP) models with a single hidden layer, leaving further extensions for future work.

II. Background

A. Basics

The L-layer MLP can be expressed mathematically as an equation consisting of L groups of simple functional compositions. Each layer l consists of a composition of some activation function σ_l with an affine function of the activation vector $a^{(l-1)} \in \mathbb{R}^{n_{l-1}}$ from the previous layer,

$$a^{(l)} = \sigma_l(W^{(l)} a^{(l-1)} + b^{(l)}),$$

where $a^{(0)} = x$ is defined to be the input features.

The goal is to set up this series of non-linear transformations such that for any input vector $x \in \mathbb{R}^{n_0}$ of features and output vector $y \in \mathbb{R}^{n_L}$ of labels one gets a new vector

$$\hat{y} = \sigma_L(W^{(L)} \sigma_{l-1}(\cdots \sigma_1(W^{(1)} x + b^{(1)}) \cdots) + b^{(L)}),$$

such that $\hat{y} \approx y$. To do so the parameters

$$\theta = \{W^{(1)}, \cdots, W^{(L)}, b^{(1)}, \cdots, b^{(L)}\}$$

are estimated by minimizing the empirical risk

$$\hat{R}(\theta) = \frac{1}{N} \sum_{i=1}^{N} L(y_i, \hat{y}(x_i|\theta))$$

over N training example pairs (x_i, y_i).

B. Model Selection

The task of model selection is to find a "best" model from some class of models using an efficient search algorithm. For our purposes, we define the best model as the model $M^* \in \mathcal{M}$ yielding the lowest overall empirical risk $\hat{R}(\theta|M)$ in a fixed, finite number of iterations, over all $\theta \in \Theta$ and all models $M \in \mathcal{M}$. That is, if it exists, we wish to find the model

$$M^* = \underset{M \in \mathcal{M}}{\text{argmin}} \left(\min_{\theta \in \Theta} \hat{R}(\theta|M) \right),$$

In the case we examine in this paper, the goal is to find the best MLP from the model class of all MLPs with a single hidden layer. Assuming the activation functions, parameter initializations, and other hyperparameters have been specified, our neural architecture selection problem reduces to finding the optimal number n^* of neurons in the hidden layer, i.e. to find

$$n^* = \underset{n \in \mathbb{N}}{\text{argmin}} \left(\min_{\theta \in \Theta} \hat{R}(\theta|n) \right).$$

For computational convenience, we impose a maximum size n_{max} for the hidden layer, thus constraining $n \leq n_{max}$.

III. Related Work

A. General Approaches

An older idea that has the affect of deleting neurons at training-time is LASSO [1]. The idea is to instead train by minimizing the regularized empirical risk

$$\hat{R}(\theta) + \lambda ||\theta||_1,$$

where $||\theta||_1$ is the flattened vector L_1 norm of all parameters in the model and $\lambda > 0$ is a hyperparameter that determines the tradeoff between model performance and model simplicity. One of the basic facts of LASSO is that it tends to shrink the parameters in a network to zero, effectively killing off neurons with smaller activations, though not explicitly removing them from the model.

Other techniques that have been tried in recent years include reinforcement learning based approaches [2], where the idea is to change the network by attempting to maximize some reward function, e.g. a function of how "happy" we are at the rate the loss is decreasing during training, and evolutionary approaches, where the idea is to allow the network evolve according to the parameters of some genetic algorithm. While these methods have been getting increasing focus lately, as of the date this was written such approaches are still largely impractical due to slower performance and difficulty to successfully implement without great computational resources.

B. Greedy Approaches

The inspiration for this work was inspired largely by [3]. In that paper the author chooses to focus on greedy-based model selection for simple recurrent neural networks (RNNs). In his model, the RNN is trained at each step using backpropagation-through-time followed by an Adam update, and then at each update the add-delete criteria are applied. The neural deletion is decided by minimizing the expected loss along with the L_1 norm of the *outgoing* weights for each neuron, which, similar to LASSO will tend to shrink the weights to zero, and then deleting neurons from the network that fall below some pre-defined deletion threshold. The neural addition is decided mainly by probability. With a certain probability, a neuron is added to the hidden layer provided the pre-specified max hidden layer size isn't exceeded. A simplified version of this algorithm is shown in algorithm 1.

Since our method was largely based on [3], we decided to put considerable effort up front into being able to replicate the author's results. He chose to test the above algorithm by using an RNN to train a character prediction model on the alphabet `a,b,),(`. The goal is to predict what the next character will be, given a simple input sequence of characters like `(ab)(ab)(ababab)....` Starting with an initial hidden layer size of $n = 1$, and hyperparameters $p_{add} = 0.01$, $p_{del} = 0.05$, and $\delta = 0.03$, we were able to reproduce his results, though not easily.

We can see from our own figure 2 that the number of neurons in the hidden layer of the RNN manages to stabilize to around 40 neurons after 100,000 iterations. Interestingly, we can also see from our own figure 2 that training actually performs better for the variable-size model than for the usual fixed-size

Algorithm 1 Simplified Miconi addition-deletion of neurons

1: **procedure** ADD_DELETE($W, b, p_{add}, p_{del}, \delta, M$)
2: $n =$ # neurons in hidden layer
3: **for** $i = 1, \cdots, n$ **do**
4: $w_i =$ outgoing weights of neuron i
5: **end for**
6: **for** each neuron i with $||w_i||_1 \leq \delta$ **do**
7: **if** $rand() < p_{del}$ **then**
8: Delete neuron i
9: $n = n - 1$
10: **end if**
11: **end for**
12: **if** $n < M$ and $rand() < p_{add}$ **then**
13: Add neuron j with $||w_j||_1 \geq \delta$, $b_j = 0$
14: $n = n + 1$
15: **end if**
16: Update W and b accordingly
17: **return** W, b
18: **end procedure**

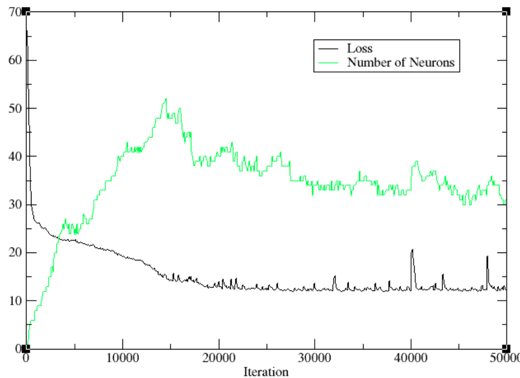

Fig. 1. Plot of training for the character prediction task performed in [3]. We can see the number of neurons stabilizes around 40.

models. Though the author didn't mention it, it turns out that his algorithm is highly sensitive to its choice of hyperparameters. This hyperparameter sensitivity, as well as the large number of hyperparameters needed to perform the model selection, is something we made it a point to address in our own method below.

IV. Method

Adapting the approach to [3], we decided to largely preserve the deletion step in algorithm 1, but change the addition step to take into account

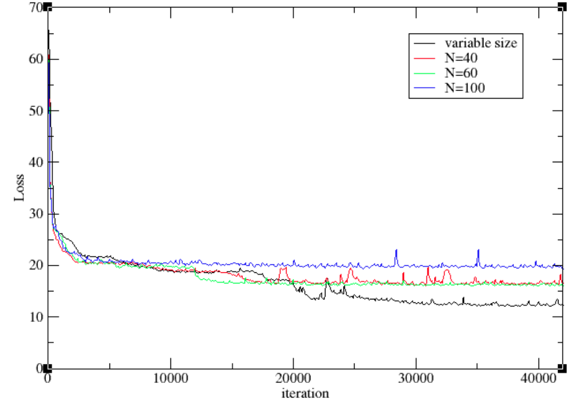

Fig. 2. Comparison of the variable-sized RNN model from [3] against several fixed sizes N. The plot qualitatively shows that the variable-sized network performs better on the character prediction task.

training behavior in the loss function. The deletion step is defined in algorithm 2, and the addition step is defined in algorithm 3.

Algorithm 2 Delete neurons in hidden layer

1: **procedure** DELETE(W, b, p, δ)
2: $n =$ # neurons in hidden layer
3: **for** $i = 1, \cdots, n$ **do**
4: $w_i =$ outgoing weights of neuron i
5: **end for**
6: **for** each neuron i with $||w_i||_1 \leq \delta$ **do**
7: **if** $rand() < p$ **then**
8: Delete neuron i
9: **end if**
10: **end for**
11: Update W and b accordingly
12: **return** W, b
13: **end procedure**

With the deletion step, we simply delete neurons in the hidden layer with near-zero outgoing weights, defined as weights with absolute value below some threshold δ, with some probability p. The rational for doing this is that such a neuron is effectively dead and not contributing to improving training, as its near-zero output will propagate to all future layers as well.

With the addition step, we check whether the training loss has stalled, and add a new neuron to the hidden layer with some probability when we decide training has stalled.

Algorithm 3 Add neurons in hidden layer

1: **procedure** ADD($W, b,$ losses$, p, \varepsilon, \tau, \delta, M$)
2: $n =$ # neurons in hidden layer
3: stalled = FALSE
4: **if** Last τ losses are within ε window **then**
5: stalled = TRUE
6: **end if**
7: **if** stalled and $n < M$ and $rand() < p$ **then**
8: Add neuron j s.t. $||w_j||_1 \geq \delta$, $b_j = 0$
9: **end if**
10: Update W and b accordingly
11: **return** W, b
12: **end procedure**

V. EXPERIMENTS

All code used to perform the following experiments is available on GitHub. Due to the fact that the above algorithms will tend to dynamically resize the parameters in the network we found it easier initially to implement all neural networks used in only base Python and NumPy. Due to the general lack of performance-boosting optimizations from doing this we largely confine the analysis in this section of the above algorithms to very simple datasets.

We focus here on a simple binary classification task, a simple regression task, and a moderate multiclass classification task. The hidden layer activation functions are all rectified linear units $ReLU(z) = \max(0, z)$, and the output activation functions and losses are determined by the task at hand. For simplicity, full-batch gradient descent is used to optimize each of the objectives. All tasks initialize the hidden layer size to $n = 1$ and allow the networks grow with training.

For the binary classification task, we sampled 5000 points from a 2-dimensional Gaussian mixture model with 10 centroids each with variance 0.01, with 5 centroids assigned label 0 and the other 5 centroids assigned label 1. A plot of the dataset is shown in figure 3. We can see the results of training in figure 4. Observe that the network stabilized to 4 neurons over 10,000 iterations, which turned out to be the minimum number required to achieve 100% accuracy on the test set. We can also observe a characteristic behavior of dynamic training: when

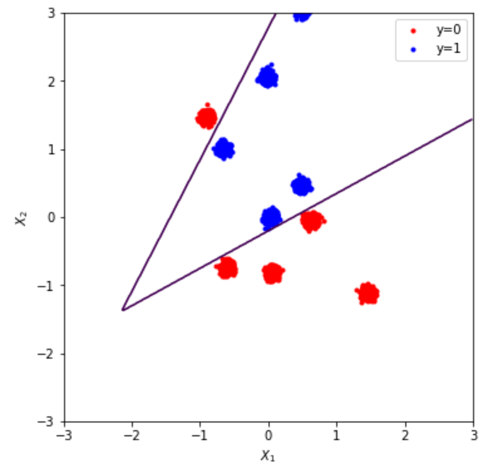

Fig. 3. Plot of the Gaussian mixture model data along with the decision boundary learned by the variable-sized model.

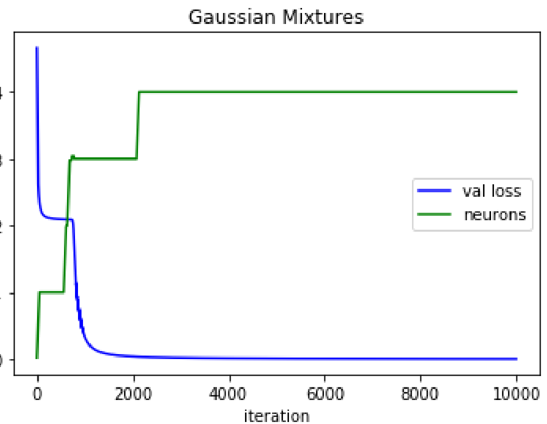

Fig. 4. Plot of (rescaled) validation loss and number of neurons over 10000 iterations of training an MLP on Gaussian mixture data. The number of neurons in the hidden layer is initialized at 1 and stabilizes at 4 neurons. The validation loss also decreases substantially as new neurons are added.

the training stalls and enough new neurons are added, training eventually drops again, in sort of an S shape. This is one of the main points of this approach. When training has stalled, add neurons to get it un-stalled.

For the regression task, we sampled 1000 points from the simple quadratic function $y = 10x^2 - 3 + \varepsilon$, where $\varepsilon \sim \mathcal{N}(0, 100)$. A plot of the dataset and the learned function approximation is shown in figure 5, and the results of training in figure 6. We can again see that the network does a pretty good job of fitting the data, with $R^2 = 0.997$. And while the number of neurons did increase, it did not manage

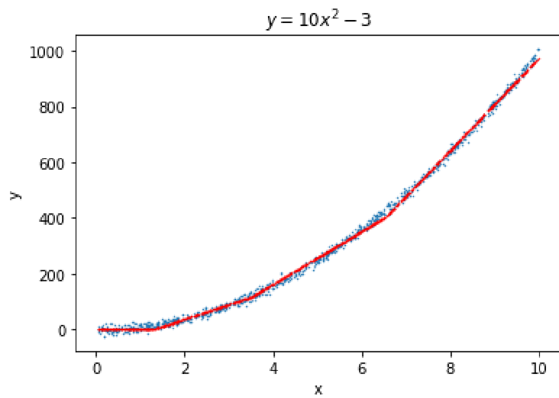

Fig. 5. Plot of the sampled regression data with mean $y = 10x^2 - 3$.

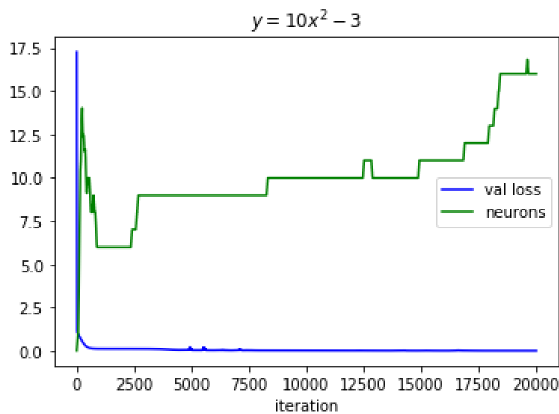

Fig. 6. Plot of (rescaled) validation loss and number of neurons over 20000 iterations of training a regression MLP on $y = 10x^2 - 3$.

to stabilize over 10,000 iterations for some reason. One possible reason for this is that the use of the squared loss results in a much sharper loss curve, which is much harder to measure stalling on.

For the multiclass classification task, we somewhat boldly attempted to perform classification on the MNIST dataset. Unfortunately, the full MNIST dataset proved much too ambitious for the first version of our code, so we instead trained on 5000 randomly-sampled images from the dataset. The results of this training is shown in figure 7. As one might expect, the high-dimensional feature space of a 28×28 image requires many more neurons to learn the underlying patterns. This is shown in our selection procedure by the fact that the network quickly grows to the max hidden layer size of 200 at around 4000 iterations, and stays there. One would assume that were the max hidden layer size higher

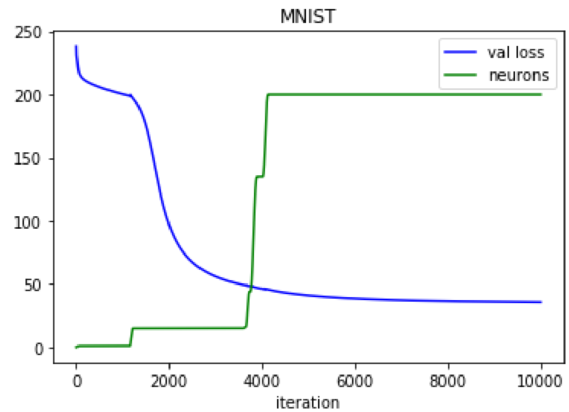

Fig. 7. Plot of (rescaled) validation loss and number of neurons over 10000 iterations of training MNIST, down-sampled to 5000. Note the max hidden size of 200 is reached here.

then the model would need many more neurons to train well on the data.

VI. Conclusions and Future Work

In conclusion, our greedy-based approach could prove viable as a form of neural architecture selection, though much has to be done to make it practical. For one, support for multiple layers needs to be added. This can be done, for example, by initializing a new layer once the max hidden size is reached, or perhaps by adding a new layer at each step with a certain small probability. Support for different types of neural networks other than MLPs would also be of value. For example, one could try this approach with convolutional neural networks by adding filters instead of neurons, or with resnets by adding resnet blocks instead of neurons.

Before any of this can be done, however, the code needs to be optimized for performance so that it can scale better to larger datasets. One immediate task will be to refactor the code using PyTorch and TensorFlow. Once this is complete more ambitious classification and regression tasks can be attempted.

On the theoretical side, more work needs to be done to analyze how well such a greedy-based model selection approach does in finding the optimal model for training over a fixed number of iterations. We suspect there to be possible local optima here to contend with, making this a somewhat challenging task. We've observed, for instance, that the number of neurons one initializes the hidden layer with does affect what number the network

finally stabilizes to. Such hurdles will have to be addressed before this technique can become a viable form of neural architecture selection.

References

[1] R. Tibshirani, “Regression Shrinkage and Selection Via the Lasso,” *Journal of the Royal Statistical Society, Series B*, vol. 58, pp. 267–288, 1994.

[2] B. Zoph and Q. V. Le, “Neural Architecture Search with Reinforcement Learning,” *arXiv:1611.01578 [cs]*, Nov 2016, arXiv: 1611.01578. [Online]. Available: http://arxiv.org/abs/1611.01578

[3] “Neural networks with differentiable structure.” [Online]. Available: http://arxiv.org/abs/1606.06216

[4] W. Pan, H. Dong, and Y. Guo, “DropNeuron: Simplifying the Structure of Deep Neural Networks,” *arXiv:1606.07326 [cs, stat]*, Jun 2016, arXiv: 1606.07326. [Online]. Available: http://arxiv.org/abs/1606.07326