The Incredible Shrinking Neural Network: Pruning to Operate in Constrained Memory Environments

Nikolas Wolfe, Aditya Sharma, Bhikhsa Raj
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA 15213, USA
{nwolfe, bhiksha}@cs.cmu.edu, adityasharma@cmu.edu

Abstract

We propose and evaluate a method for pruning neural networks to operate in constrained memory environments such as mobile or embedded devices. We evaluate a simple pruning technique using first-order derivative approximations of the gradient of each neuron in an optimally trained network, and turning off those neurons which contribute least to the output of the network. We then show the limitations of this type of approximation by comparing against the ground truth value for the change in error resulting from the removal of a given neuron. We attempt to improve on this using a second-order derivative approximation. We also explore the correlation between neurons in a trained network and attempt to improve our choice of candidate neurons for removal to account for faults that can occur from the removal of a single neuron at a time. We argue that this method of pruning allows for the optimal tradeoff in network size versus accuracy in order to operate within the memory constraints of a particular device or application environment.

1 Introduction

Neural network pruning algorithms were first popularized by Sietsma & Dow (1988) as a mechanism to determine the proper size network required to solve a particular problem. To this day, network design and optimal pruning remain inherently difficult tasks. For problems which cannot be solved using linear threshold units alone, Baum & Haussler (1989) demonstrate there is no way to precisely determine the appropriate size of a neural network a priori given any random set of training instances. Using too few neurons inhibits learning, and so in practice it is common to attempt to over-parameterize networks initially using a large number of hidden units and weights. However, as Chauvin (1990) writes, this approach can lead to over-fitting as the network's unnecessary free parameters start to latch on to idiosyncrasies in the training data.

Pruning algorithms, as comprehensively surveyed by Reed (1993), are a useful set of heruristics designed to identify and remove network parameters which do not contribute significantly to the output of the network and potentially inhibit generalization performance. At the time of Reed's writing, reducing network size was also a practical concern, as smaller networks are preferable in situations where computational resources are scarce. In this paper we are particularly concerned with application domains in which space is limited and network size constraints must be imposed with minimal impact on performance.

2 Related Work

Neural network over-fitting is fundamentally a problem arising from the use of too many free parameters. Regardless of the number of weights used in a given network, as Segee & Carter (1991) assert, the representation of a learned function approximation is almost never evenly distributed over the hidden units, and the removal of any single hidden unit at random can actually result in a total network fault. Mozer & Smolensky (1989b) suggest that only a subset of the hidden units in a neural network actually latch on to the invariant or generalizing properties of the training inputs, and the rest learn to either mutually cancel each other out or begin over-fitting to the noise in the data. Determining which elements are unnecessary and removing them outright is therefore a well-founded approach to improving network generalization, and simultaneously provides a way to reduce their size in memory.

The generalization performance of neural networks has been well studied, and apart from pruning algorithms many heuristics have been used to avoid overfitting, such as dropout (Srivastava et al. (2014)), maxout (Goodfellow et al. (2013)), and cascade correlation (Fahlman & Lebiere (1989)), among others. However, these algorithms do not explicitly prioritize the reduction of network memory footprint as a part of their optimization criteria per se, (although in the opinion of the authors Fahlman's cascade correlation architecture holds great promise in this regard.) Computer memory size and processing capabilities have improved so much since the introduction of pruning algorithms in the late 1980s that space complexity has become a relatively negligible concern. The proliferation of cloud-based computing services has furthermore enabled mobile and embedded devices to leverage the power of massive data and computing centers remotely. In this domain, however, it is also reasonable to suggest that certain performance-critical applications running on low-resource devices could benefit from the ability to use neural networks locally.

At present there are few (if any) mechanisms specifically designed to shrink neural networks down in order to meet an externally imposed constraint on byte-size in memory. Without explicitly removing parameters from the network, one could use weight quantization to reduce the number of bytes used to represent each weight parameter, as investigated by Balzer et al. (1991), Dundar & Rose (1994), and Hoehfeld & Fahlman (1992). Of course, this method can only reduce the size of the network by a factor proportional to the byte-size reduction of each weight parameter.

Another method which has recently gained popularity is using the singular values of a trained weight matrix as basis vectors from which to derive a compressed hidden layer.

* describe SVD stuff

If we wanted to continually shrink a network to its absolute minimal size in an optimal manner, we might accomplish this using any number of off-the-shelf pruning algorithms, such as Skeletonization (Mozer & Smolensky (1989a)), Optimal Brain Damage (LeCun et al. (1989)), or later variants such as Optimal Brain Surgeon (Hassibi & Stork (1993)). In fact, we borrow much of our inspiration from these antecedent algorithms, with one major variation.

The aforementioned strategies all focus on the targeting and removal of *weight* parameters. Scoring and ranking individual weight parameters in a large network computationally expensive, and generally speaking the removal of a single weight from a large network is a drop in the bucket in terms of reducing a network's core memory footprint. We therefore train our sights on the ranking and removal of entire neurons along with their associated weight parameters. We argue that this is more efficient computationally as well as practically in terms of quickly reaching a target reduction in memory size. Our approach also attacks the angle of giving downstream applications a realistic expectation of the minimal increase in error resulting from the removal of a specified percentage of neurons from a trained network. Such trade-offs are unavoidable, but performance impacts can be limited if a principled approach is used to find candidate neurons for removal.

* brief overview of approach

3 METHODOLOGY

The general approach taken to prune an optimally trained neural network in the present work is to create a ranked list of all the neurons in the network based off of one of the 3 ranking criteria: Brute Force approximation (which we use as our ground truth), linear approximation and quadratic approximation. We then test the effects of removing neurons sequentially on the accuracy of the network. These tests can be found in the Experiments section. Next, we propose our algorithm with 2 variants

3.1 Brute Force Removal Approach

This is perhaps the most naive yet the most accurate method for pruning the network. It is also the slowest and hence unusable on large-scale neural networks with thousands of neurons. The idea is to manually check the effect of every single neuron on the output. This is done by running a forward propagation on the validation set K times (where K is the total number of neurons in the network), turning off exactly one neuron each time (keeping all other neurons active) and noting down the change in error. Turning a neuron off can be achieved by simply setting its output to 0. This results in all the outgoing weights from that neuron being turned off. This change in error is then used to generate the ranked list.

3.1.1 Taylor Series Representation of Error

Let us denote the total error from the optimally trained neural network for any given validation dataset with N instances as E_{total}. Then,

$$E_{\text{total}} = \sum_n E_n, \tag{1}$$

where E_n is the error from the network over one validation instance. E_n can be seen as a function O, where O is the output of any general neuron in the network (In reality this error depends on each neuron's output, but for the sake of simplicity we use O to represent that). This error can be approximated at a particular neuron's output (say O_k) by using the 2nd order Taylor Series as,

$$\hat{E}_n(O) \approx E_n(O_k) + (O - O_k) \cdot \left.\frac{\partial E_n}{\partial O}\right|_{O_k} + 0.5 \cdot (O - O_k)^2 \cdot \left.\frac{\partial^2 E_n}{\partial O^2}\right|_{O_k}, \tag{2}$$

where $\hat{E}_n(O_k)$ represents the contribution of a neuron k to the total error E_n of the network for any given validation instance n. When this neuron is pruned, its output O_k becomes 0. From equation 2, the contribution $E_n(0)$ of this neuron, then becomes:

$$\hat{E}_n(0) \approx E_n(O_k) - O_k \cdot \left.\frac{\partial E_n}{\partial O}\right|_{O_k} + 0.5 \cdot O_k^2 \cdot \left.\frac{\partial^2 E_n}{\partial O^2}\right|_{O_k} \tag{3}$$

Replacing O by O_k in equation 2 shows us that the error is approximated perfectly by equation 2 at O_k. Using this and equation 3 we get:

$$\Delta E_{n,k} = \hat{E}_n(0) - \hat{E}_n(O_k) = -O_k \cdot \left.\frac{\partial E_n}{\partial O}\right|_{O_k} + 0.5 \cdot O_k^2 \cdot \left.\frac{\partial^2 E_n}{\partial O^2}\right|_{O_k}, \tag{4}$$

where $\Delta E_{n,k}$ is the change in the total error of the network given a validation instance n, when exactly one neuron (k) is turned off.

3.2 Linear Approximation Approach

We define the following network terminology here which will be used in this section and all subsequent sections unless stated otherwise. Figure 1 can be used as a reference to the terminology defined here:

$$E = \frac{1}{2}\sum_i (o_i^{(0)} - t_i)^2 \quad o_i^{(m)} = \sigma(x_i^{(m)}) \quad x_i^{(m)} = \sum_j w_{ji}^{(m)} o_j^{(m+1)} \quad c_{ji}^{(m)} = w_{ji}^{(m)} o_j^{(m+1)} \tag{5}$$

Superscripts represent the index of the layer of the network in question, with 0 representing the output layer. E is the squared-error network cost function. Note that we are dropping the E_n

notation used previously as the subsequent discussion is insusceptible to the data instances. $o_i^{(m)}$ is the ith output in layer m generated by the activation function σ, which in this paper is is the standard logistic sigmoid. $x_i^{(m)}$ is the weighted sum of inputs to the ith neuron in the mth layer, and $c_{ji}^{(m)}$ is the contribution of the jth neuron in the $(m+1)$th layer to the input of the ith neuron in the mth layer. $w_{ji}^{(m)}$ is the weight between the jth neuron in the $(m+1)$th layer and the ith neuron in the mth layer.

We can use equation 4 to get the linear error approximation of the change in error due to the kth neuron being turned off and represent it as ΔE_k^1 as follows:

$$\Delta E_k^1 = -o_k \cdot \left. \frac{\partial E}{\partial o_j^{(m+1)}} \right|_{o_k} \tag{6}$$

The derivative term above is the first-order gradient which represents the change in error with respect to the output of a given neuron o_j in the $(m+1)$th layer. This term can be collected during back-propagation. The derivative term above can be calculated as follows:

$$\frac{\partial E}{\partial o_j^{(m+1)}} = \sum_i \frac{\partial E}{\partial x_i^{(m)}} \cdot w_{ji}^{(m)} \tag{7}$$

The full step-by-step mathematical derivation of the above equation can be found in the appendix.

3.3 QUADRATIC APPROXIMATION APPROACH

As seen in equation 4, $\Delta E_{n,k}$ which can now be represented as ΔE_k^2 is the quadratic approximation of the change in error due to the kth neuron being turned off. The quadratic term in equation 4 requires some discussion which we provide here. A more detailed and step-by-step mathematical derivation can be found in the appendix.

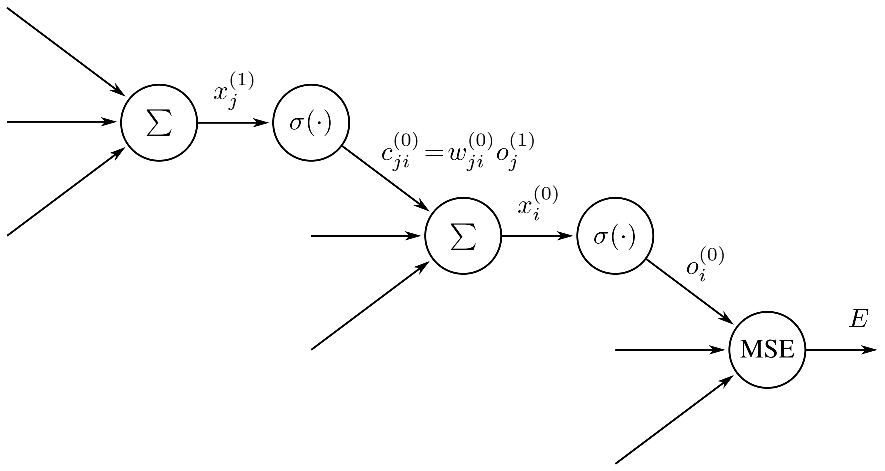

Figure 1: A computational graph of a simple feed-forward network illustrating the naming of different variables, where $\sigma(\cdot)$ is the nonlinearity, MSE is the mean-squared error cost function and E is the overall loss.

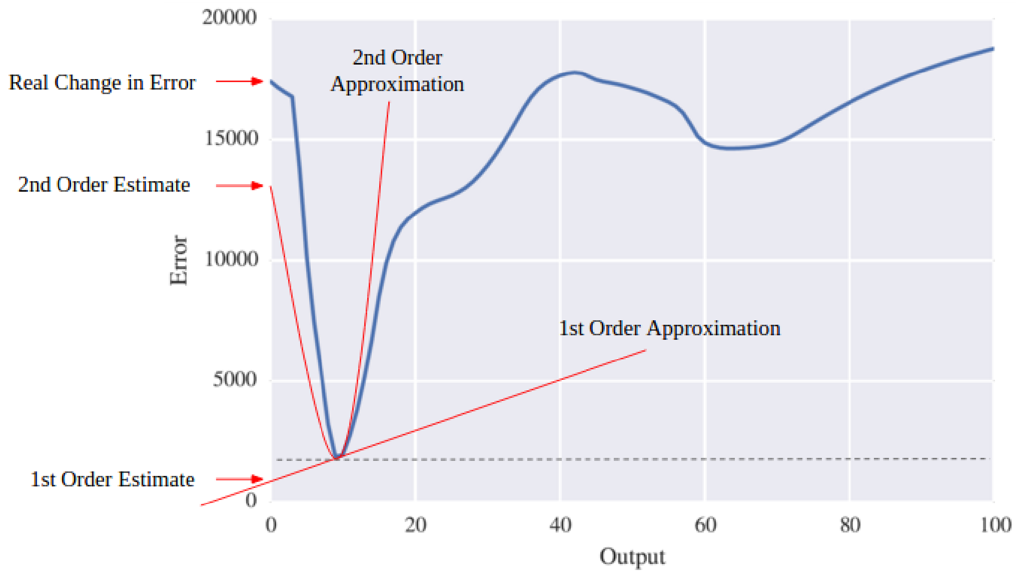

Figure 2: The intuition behind neuron pruning decision.

Let us reproduce equation 4 in our new terminology here:

$$\Delta E_k^2 = -o_k \cdot \left.\frac{\partial E}{\partial o_j^{(m+1)}}\right|_{o_k} + 0.5 \cdot o_k^2 \cdot \left.\frac{\partial^2 E}{\partial {o_j^{(m+1)}}^2}\right|_{o_k} \tag{8}$$

The second term here involves the second-order gradient which represents the second-order change in error with respect to the output of a given neuron o_j in the $(m+1)$th layer. This term can be generated by performing back-propagation using second derivatives. A full derivation of the second derivative back-propagation can be found in the appendix. We will quote some results from the derivation here. The second-order derivative term can be represented as:

$$\frac{\partial^2 E}{\partial {o_j^{(m+1)}}^2} = \sum_i \frac{\partial^2 E}{\partial {c_{ji}^{(m)}}^2} \left(w_{ji}^{(m)}\right)^2 \tag{9}$$

Here,$c_{ji}^{(m)}$ is one of the component terms of $x_i^{(m)}$, as follows from the equations in 5. Hence, it can be easily proved that (full proof in appendix):

$$\frac{\partial^2 E}{\partial {c_{ji}^{(m)}}^2} = \frac{\partial^2 E}{\partial {x_i^{(m)}}^2} \tag{10}$$

Now, the value of $x_i^{(m)}$ can be easily calculated through the steps of the second-order back-propagation using Chain Rule. The full derivation can again, be found in the appendix.

$$\frac{\partial^2 E}{\partial {x_i^{(m)}}^2} = \frac{\partial^2 E}{\partial {o_i^{(m)}}^2} \left(\sigma'\left(x_i^{(m)}\right)\right)^2 + \frac{\partial E}{\partial o_i^{(m)}} \sigma''\left(x_i^{(m)}\right) \tag{11}$$

3.4 Proposed Pruning Algorithm

Figure 2 shows a random error function plotted against the output of any given neuron. Note that this figure is for illustration purposes only. The error function is minimized at a particular value of the output as can be seen in the figure. This output is fixed during training as decided by back propagation. Pruning this particular neuron would result in a change in the total error that would be equal to the increase shown in the figure. This neuron is clearly a bad candidate for removal. The straight red line in the figure represents the first-order approximation of the error using Taylor Series as described before while the parabola represents a second-order approximation. It can be clearly seen that the second-order approximation is a much better estimate of the change in error.

We would also like to point out here that it is possible in some cases that there is some thresholding required when trying to approximate the error using the 2nd order Taylor Series expansion. These cases might arise when the parabolic approximation undergoes a steep slope change. To take into account such cases, we employed mean and median thresholding, where any change above a certain threshold was assigned a mean or median value respectively.

Two pruning algorithms are proposed here. They are different in the way the neurons are ranked but both of them use ΔE_k^2, the second-order approximation of the error function we got from the Taylor Series, as the basis for the ranking.

The first step in both the algorithms is to decide a stopping criterion. This can vary depending on the application but some intuitive stopping criteria can be the maximum number of neurons to remove, percentage scaling needed, maximum allowable accuracy drop etc.

3.4.1 Algorithm I: Single Overall Ranking

The complete algorithm is shown in Algorithm 1. The idea here is to generate a single ranked list based on the values of ΔE_k^2. This involves a single pass of second-order back-propagation (without weight updates) to collect the gradients for each neuron. The neurons from the formed rank-list (with the least value of ΔE_k^2) are then pruned according to the stopping criterion decided.

```
Data: optimally trained network, training set
Result: A pruned network
initialize and define stopping criterion ;
perform forward propagation over the training set ;
perform second-order back-propagation without updating weights and collect linear and quadratic
  gradients ;
rank the remaining neurons based on $\Delta E_k^2$;
while stopping criterion is not met do
    remove the last ranked neuron ;
end
```

Algorithm 1: Single Overall Ranking

3.4.2 Algorithm II: Iterative Re-Ranking

In this greedy variation of the algorithm (Algorithm 2), after each neuron is pruned, the remaining network is made to undergo a single pass of second-order back-propagation (without weight updates) and the rank list is formed again. Hence, each removal involves a new pass through the network. This method is computationally more expensive but takes into account the dependencies the neurons might have with one another which would lead to a change in error contribution every time a dependent neuron is removed.

4 Experiments

We propose several experiments to compare our discussed algorithm for determining which neurons to switch off with the empirically determined best ordering. First, we do brute force ordering. Using the training data, we get the sum of squared errors for the unaltered network. Then, we switch off one neuron at a time, and do one forward pass through the training data to get the change in the output error. We use this value to rank the neurons. This is the ground truth.

```
Data: optimally trained network, training set
Result: A pruned network
initialize and define stopping criterion ;
while stopping criterion is not met do
    perform forward propagation over the training set ;
    perform second-order back-propagation without updating weights and collect linear and
      quadratic gradients ;
    rank the remaining neurons based on $\Delta E_k^2$ ;
    remove the worst neuron based on the ranking ;
end
```

Algorithm 2: Iterative Re-Ranking

4.1 Correlation of Ranking vs. Ground Truth

4.2 Change in Ground Truth Error vs. Expected Improvement

4.3 Sequential Turn-Off

4.4 Iterative Re-Estimation of Next Best Neuron

5 Experimental Results & Conclusions

Both versions of the proposed algorithm were run on 4 pattern recognition datasets, starting from a cosine wave and increasing in complexity to a distributed random shape pattern. These patterns are shown in Figure **??**. In all cases, the models were trained on a 2-hidden layer network with 50 neurons in each layer. Median thresholding produced much better results in all the cases and hence is used for all the results presented here.

The results of pruning using ground truth, first-order Taylor Series approximation and second-order Taylor Series approximation for both the variations of the proposed algorithm are shown in Figures, **??**, 5, 6 and 7.

It can be seen in all the cases that first-order Taylor Series expansion based pruning is a very bad approximation of the ground truth while ranking based on second-order expansion approximates the ground truth with much more accuracy. Also, it can be observed that the Iterative Re-Ranking version of the algorithm always performs better that the Single Overall Ranking version. All of these observations are in consistence with the intuitive assumptions from the Methodology section.

Table 1 provides an overall summary of the experiments carried out. It can be seen that all of the experiments were carried out on optimally trained networks. The percentages shown represent the amount of pruning possible without a significant drop in performance.

In conclusion, it can be said that using the second-order Taylor Series expansion of the error function to rank individual neurons is an encouragingly accurate method of pruning networks to save memory. However it is definitely not the most accurate representation of the ranking based on the actual contribution of neurons to the error function. Another key take-away is that there are dependencies between individual neurons which might not be apparent from the outside. The better success of the Iterative Re-Ranking algorithm validates this. Perhaps a different approach like using Legendre Polynomials instead of the Taylor Series expansion would result in a better approximation of the neuron contributions.

6 Cosine Function

7 Discussion of Results

* first or second layer? * cascade correlation in reverse? * retraining? * investigation of the true independence of the elements

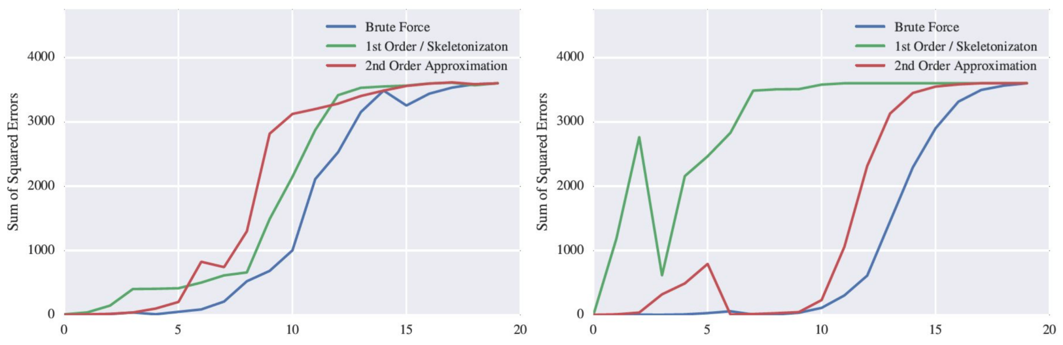

Figure 3: (Starting training accuracy: 0.9999993) Performance degradation using a small network with two layers of ten neurons each. Vertical axes represent the sum of squared errors on the training set, and horizontal axes represent the number of neurons removed. The left graph represents the single-pass overall ranking procedure (Algorithm 1) and the right represents continual re-estimation procedure (Algorithm 2).

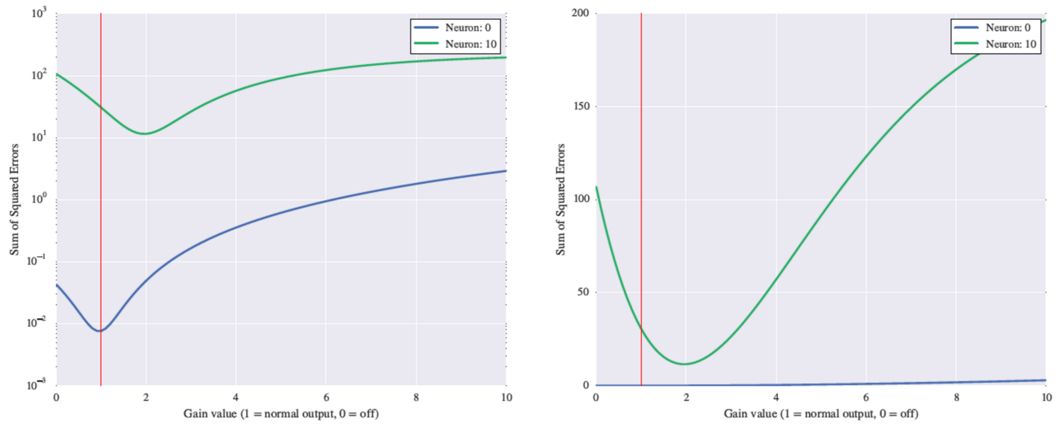

Figure 4: (Starting training accuracy: 0.9999993) Performance degradation using a small network with two layers of ten neurons each. Vertical axes represent the sum of squared errors on the training set, and horizontal axes represent the number of neurons removed. The left graph represents the single-pass overall ranking procedure (Algorithm 1) and the right represents continual re-estimation procedure (Algorithm 2).

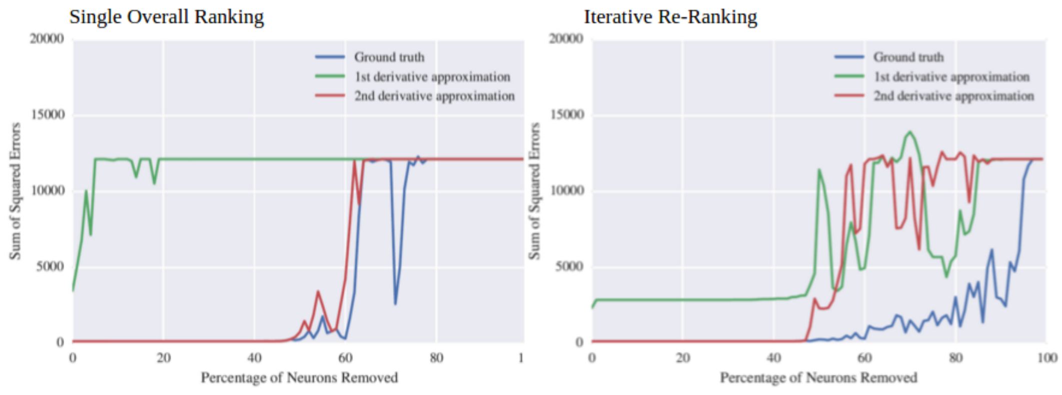

Figure 5: Performance on a diamond pattern dataset for both variations of the proposed algorithm. The blue curve shows the ground truth estimated from brute force pruning, the green and the red curves show pruning based on the approximation from first and second-order expansion of the Taylor Series.

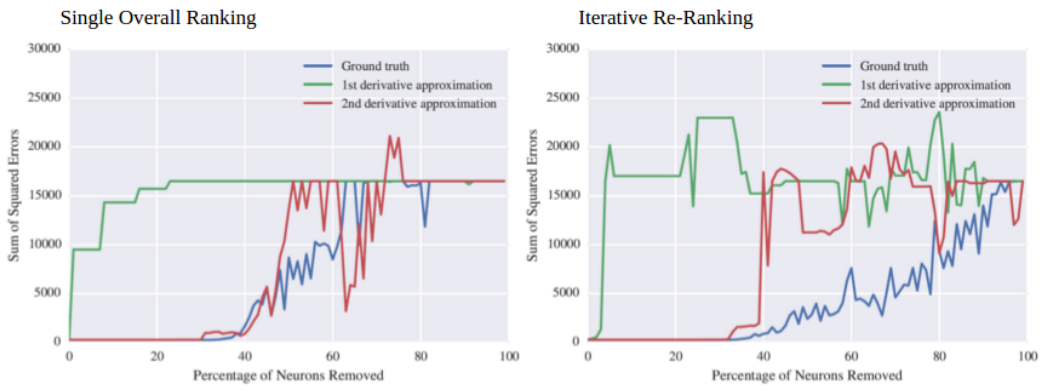

Figure 6: Performance on a random shape pattern dataset for both variations of the proposed algorithm. The blue curve shows the ground truth estimated from brute force pruning, the green and the red curves show pruning based on the approximation from first and second-order expansion of the Taylor Series.

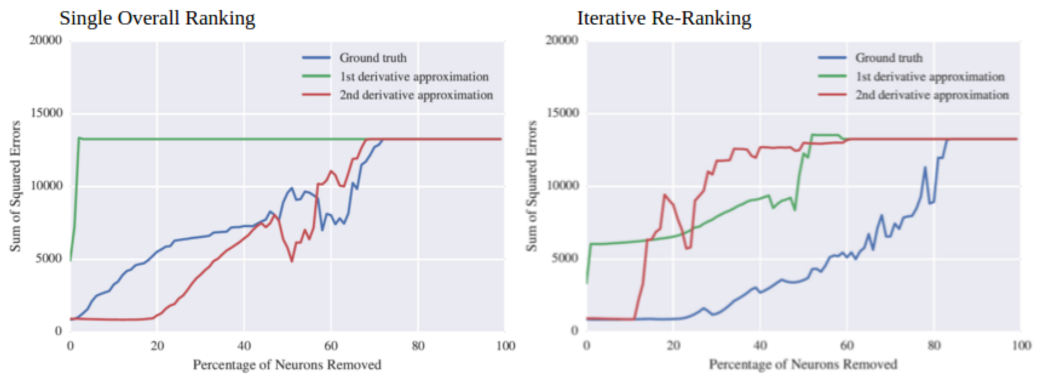

Figure 7: Performance on a distributed random shape pattern dataset for both variations of the proposed algorithm. The blue curve shows the ground truth estimated from brute force pruning, the green and the red curves show pruning based on the approximation from first and second-order expansion of the Taylor Series.

Pattern	Test Acc.	Ground Truth	Proposed Algorithm
Cosine Wave	0.9999	90%	70%
Diamond	0.9921	48%	48%
Random	0.9861	38%	35%
Dist. Random	0.9601	20%	10%
Circle	0.9968	40%	0%

Table 1: A comparison of the percentage pruning achieved using brute force and the proposed algorithm on optimally trained networks. The percentages shown represent the amount of pruning possible without a significant drop in performance.

8 Conclusions

* second derivative method works better * still not very good * retraining is a good idea * speedup of brute force method * which layer gets pruned first

Acknowledgments

* thank: lukas drude abelino jiminez thomas schaaf don wolfe

9 Appendix A: Second Derivative Back-Propagation

Name and network definitions:

$$E = \frac{1}{2}\sum_i (o_i^{(0)} - t_i)^2 \quad o_i^{(m)} = \sigma(x_i^{(m)}) \quad x_i^{(m)} = \sum_j w_{ji}^{(m)} o_j^{(m+1)} \quad c_{ji}^{(m)} = w_{ji}^{(m)} o_j^{(m+1)} \tag{12}$$

Superscripts represent the index of the layer of the network in question, with 0 representing the output layer. E is the squared-error network cost function. $o_i^{(m)}$ is the ith output in layer m generated by the activation function σ, which in this paper is is the standard logistic sigmoid. $x_i^{(m)}$ is the weighted sum of inputs to the ith neuron in the mth layer, and $c_{ji}^{(m)}$ is the contribution of the jth neuron in the $m+1$ layer to the input of the ith neuron in the mth layer.

9.1 First and Second Derivatives

The first and second derivatives of the cost function with respect to the outputs:

$$\frac{\partial E}{\partial o_i^{(0)}} = o_i^{(0)} - t_i \tag{13}$$

$$\frac{\partial^2 E}{\partial {o_i^{(0)}}^2} = 1 \tag{14}$$

The first and second derivatives of the sigmoid function in forms depending only on the output:

$$\sigma'(x) = \sigma(x)\,(1 - \sigma(x)) \tag{15}$$
$$\sigma''(x) = \sigma'(x)\,(1 - 2\sigma(x)) \tag{16}$$

The second derivative of the sigmoid is easily derived from the first derivative:

$$\sigma'(x) = \sigma(x)\,(1 - \sigma(x)) \tag{17}$$
$$\sigma''(x) = \frac{\mathrm{d}}{\mathrm{d}x}\,\underbrace{\sigma(x)}_{f(x)}\,\underbrace{(1 - \sigma(x))}_{g(x)} \tag{18}$$
$$\sigma''(x) = f'(x)g(x) + f(x)g'(x) \tag{19}$$
$$\sigma''(x) = \sigma'(x)(1 - \sigma(x)) - \sigma(x)\sigma'(x) \tag{20}$$
$$\sigma''(x) = \sigma'(x) - 2\sigma(x)\sigma'(x) \tag{21}$$
$$\sigma''(x) = \sigma'(x)(1 - 2\sigma(x)) \tag{22}$$

And for future convenience:

$$\frac{\mathrm{d}o_i^{(m)}}{\mathrm{d}x_i^{(m)}} = \frac{\mathrm{d}}{\mathrm{d}x_i^{(m)}}\left(o_i^{(m)} = \sigma(x_i^{(m)})\right) \tag{23}$$
$$= \left(o_i^{(m)}\right)\left(1 - o_i^{(m)}\right) \tag{24}$$
$$= \sigma'\left(x_i^{(m)}\right) \tag{25}$$
$$\frac{\mathrm{d}^2 o_i^{(m)}}{\mathrm{d}{x_i^{(m)}}^2} = \frac{\mathrm{d}}{\mathrm{d}x_i^{(m)}}\left(\frac{\mathrm{d}o_i^{(m)}}{\mathrm{d}x_i^{(m)}} = \left(o_i^{(m)}\right)\left(1 - o_i^{(m)}\right)\right) \tag{26}$$
$$= \left(o_i^{(m)}\left(1 - o_i^{(m)}\right)\right)\left(1 - 2o_i^{(m)}\right) \tag{27}$$
$$= \sigma''\left(x_i^{(m)}\right) \tag{28}$$

Derivative of the error with respect to the ith neuron's input $x_i^{(0)}$ in the output layer:

$$\frac{\partial E}{\partial x_i^{(0)}} = \frac{\partial E}{\partial o_i^{(0)}} \frac{\partial o_i^{(0)}}{\partial x_i^{(0)}} \tag{29}$$

$$= \underbrace{\left(o_i^{(0)} - t_i\right)}_{\text{from (13)}} \underbrace{\sigma\left(x_i^{(0)}\right)\left(1 - \sigma\left(x_i^{(0)}\right)\right)}_{\text{from (15)}} \tag{30}$$

$$= \left(o_i^{(0)} - t_i\right)\left(o_i^{(0)}\left(1 - o_i^{(0)}\right)\right) \tag{31}$$

$$= \left(o_i^{(0)} - t_i\right)\sigma'\left(x_i^{(0)}\right) \tag{32}$$

Second derivative of the error with respect to the ith neuron's input $x_i^{(0)}$ in the output layer:

$$\frac{\partial^2 E}{\partial {x_i^{(0)}}^2} = \frac{\partial}{\partial x_i^{(0)}}\left(\frac{\partial E}{\partial o_i^{(0)}} \frac{\partial o_i^{(0)}}{\partial x_i^{(0)}}\right) \tag{33}$$

$$= \frac{\partial^2 E}{\partial x_i^{(0)} \partial o_i^{(0)}} \frac{\partial o_i^{(0)}}{\partial x_i^{(0)}} + \frac{\partial E}{\partial o_i^{(0)}} \frac{\partial^2 o_i^{(0)}}{\partial {x_i^{(0)}}^2} \tag{34}$$

$$= \frac{\partial^2 E}{\partial x_i^{(0)} \partial o_i^{(0)}} \underbrace{\left(o_i^{(0)}\left(1 - o_i^{(0)}\right)\right)}_{\text{from (15)}} + \underbrace{\left(o_i^{(0)} - t_i\right)}_{\text{from (13)}} \underbrace{\left(o_i^{(0)}\left(1 - o_i^{(0)}\right)\right)\left(1 - 2o_i^{(0)}\right)}_{\text{from (16)}} \tag{35}$$

$$\left(\frac{\partial^2 E}{\partial x_i^{(0)} \partial o_i^{(0)}}\right) = \frac{\partial}{\partial x_i^{(0)}} \frac{\partial E}{\partial o_i^{(0)}} = \frac{\partial}{\partial x_i^{(0)}} \underbrace{\left(o_i^{(0)} - t_i\right)}_{\text{from (13)}} = \frac{\partial o_i^{(0)}}{\partial x_i^{(0)}} = \underbrace{\left(o_i^{(0)}\left(1 - o_i^{(0)}\right)\right)}_{\text{from (15)}} \tag{36}$$

$$\frac{\partial^2 E}{\partial {x_i^{(0)}}^2} = \left(o_i^{(0)}\left(1 - o_i^{(0)}\right)\right)^2 + \left(o_i^{(0)} - t_i\right)\left(o_i^{(0)}\left(1 - o_i^{(0)}\right)\right)\left(1 - 2o_i^{(0)}\right) \tag{37}$$

$$= \left(\sigma'\left(x_i^{(0)}\right)\right)^2 + \left(o_i^{(0)} - t_i\right)\sigma''\left(x_i^{(0)}\right) \tag{38}$$

First derivative of the error with respect to a single input contribution $c_{ji}^{(0)}$ from neuron j to neuron i with weight $w_{ji}^{(0)}$ in the output layer:

$$\frac{\partial E}{\partial c_{ji}^{(0)}} = \frac{\partial E}{\partial o_i^{(0)}} \frac{\partial o_i^{(0)}}{\partial x_i^{(0)}} \frac{\partial x_i^{(0)}}{\partial c_{ji}^{(0)}} \tag{39}$$

$$= \underbrace{\left(o_i^{(0)} - t_i\right)}_{\text{from (13)}} \underbrace{\left(o_i^{(0)}\left(1 - o_i^{(0)}\right)\right)}_{\text{from (15)}} \frac{\partial x_i^{(0)}}{\partial c_{ji}^{(0)}} \tag{40}$$

$$\left(\frac{\partial x_i^{(m)}}{\partial c_{ji}^{(m)}}\right) = \frac{\partial}{\partial c_{ji}^{(m)}}\left(x_i^{(m)} = \sum_j w_{ji}^{(m)} o_j^{(m+1)}\right) = \frac{\partial}{\partial c_{ji}^{(m)}}\left(c_{ji}^{(m)} + k\right) = 1 \tag{41}$$

$$\frac{\partial E}{\partial c_{ji}^{(0)}} = \left(o_i^{(0)} - t_i\right)\left(o_i^{(0)}\left(1 - o_i^{(0)}\right)\right) \tag{42}$$

$$= \underbrace{\left(o_i^{(0)} - t_i\right)\sigma'\left(x_i^{(0)}\right)}_{\text{from (32)}} \tag{43}$$

$$\frac{\partial E}{\partial c_{ji}^{(0)}} = \frac{\partial E}{\partial x_i^{(0)}} \tag{44}$$

Second derivative of the error with respect to a single input contribution $c_{ji}^{(0)}$:

$$\frac{\partial^2 E}{\partial {c_{ji}^{(0)}}^2} = \frac{\partial}{\partial c_{ji}^{(0)}} \left(\frac{\partial E}{\partial c_{ji}^{(0)}} = \underbrace{\left(o_i^{(0)} - t_i\right) \sigma'\left(x_i^{(0)}\right)}_{\text{from } (43)} \right) \tag{45}$$

$$= \frac{\partial}{\partial c_{ji}^{(0)}} \left(\sigma\left(x_i^{(0)}\right) - t_i \right) \sigma'\left(x_i^{(0)}\right) \tag{46}$$

$$= \frac{\partial}{\partial c_{ji}^{(0)}} \left(\sigma\left(\sum_j w_{ji}^{(m)} o_j^{(m+1)} \right) - t_i \right) \sigma'\left(\sum_j w_{ji}^{(m)} o_j^{(m+1)} \right) \tag{47}$$

$$= \frac{\partial}{\partial c_{ji}^{(0)}} \left(\sigma\left(\sum_j c_{ji}^{(0)} \right) - t_i \right) \sigma'\left(\sum_j c_{ji}^{(0)} \right) \tag{48}$$

$$= \frac{\partial}{\partial c_{ji}^{(0)}} \underbrace{\left(\sigma\left(c_{ji}^{(0)} + k\right) - t_i \right)}_{f\left(c_{ji}^{(0)}\right)} \underbrace{\sigma'\left(c_{ji}^{(0)} + k\right)}_{g\left(c_{ji}^{(0)}\right)} \tag{49}$$

We now make use of the abbreviations f and g:

$$= f'\left(c_{ji}^{(0)}\right) g\left(c_{ji}^{(0)}\right) + f\left(c_{ji}^{(0)}\right) g'\left(c_{ji}^{(0)}\right) \tag{50}$$

$$= \sigma'\left(c_{ji}^{(0)} + k\right) \sigma'\left(c_{ji}^{(0)} + k\right) + \left(\sigma\left(c_{ji}^{(0)} + k\right) - t_i\right) \sigma''\left(c_{ji}^{(0)} + k\right) \tag{51}$$

$$= \sigma'\left(c_{ji}^{(0)} + k\right)^2 + \left(o_i^{(0)} - t_i\right) \sigma''\left(c_{ji}^{(0)} + k\right) \tag{52}$$

$$\left(c_{ji}^{(0)} + k = \sum_j c_{ji}^{(0)} = \sum_j w_{ji}^{(m)} o_j^{(m+1)} = x_i^{(0)} \right) \tag{53}$$

$$\frac{\partial^2 E}{\partial {c_{ji}^{(0)}}^2} = \underbrace{\left(\sigma'\left(x_i^{(0)}\right)\right)^2 + \left(o_i^{(0)} - t_i\right) \sigma''\left(x_i^{(0)}\right)}_{\text{from } (38)} \tag{54}$$

$$\frac{\partial^2 E}{\partial {c_{ji}^{(0)}}^2} = \frac{\partial^2 E}{\partial {x_i^{(0)}}^2} \tag{55}$$

9.1.1 Summary Of Output Layer Derivatives

$$\frac{\partial E}{\partial o_i^{(0)}} = o_i^{(0)} - t_i \qquad \frac{\partial^2 E}{\partial {o_i^{(0)}}^2} = 1 \tag{56}$$

$$\frac{\partial E}{\partial x_i^{(0)}} = \left(o_i^{(0)} - t_i\right) \sigma'\left(x_i^{(0)}\right) \qquad \frac{\partial^2 E}{\partial {x_i^{(0)}}^2} = \left(\sigma'\left(x_i^{(0)}\right)\right)^2 + \left(o_i^{(0)} - t_i\right) \sigma''\left(x_i^{(0)}\right) \tag{57}$$

$$\frac{\partial E}{\partial c_{ji}^{(0)}} = \frac{\partial E}{\partial x_i^{(0)}} \qquad \frac{\partial^2 E}{\partial {c_{ji}^{(0)}}^2} = \frac{\partial^2 E}{\partial {x_i^{(0)}}^2} \tag{58}$$

9.1.2 Hidden Layer Derivatives

The first derivative of the error with respect to a neuron with output $o_j^{(1)}$ in the first hidden layer, summing over all partial derivative contributions from the output layer:

$$\frac{\partial E}{\partial o_j^{(1)}} = \sum_i \frac{\partial E}{\partial o_i^{(0)}} \frac{\partial o_i^{(0)}}{\partial x_i^{(0)}} \frac{\partial x_i^{(0)}}{\partial c_{ji}^{(0)}} \frac{\partial c_{ji}^{(0)}}{\partial o_j^{(1)}} = \sum_i \underbrace{\left(o_i^{(0)} - t_i\right) \sigma'\left(x_i^{(0)}\right)}_{\text{from } (32)} w_{ji}^{(0)} \tag{59}$$

$$\frac{\partial c_{ji}^{(m)}}{\partial o_j^{(m+1)}} = \frac{\partial}{\partial o_j^{(m+1)}} \left(c_{ji}^{(m)} = w_{ji}^{(m)} o_j^{(m+1)}\right) = w_{ji}^{(m)} \tag{60}$$

$$\frac{\partial E}{\partial o_j^{(1)}} = \sum_i \frac{\partial E}{\partial x_i^{(0)}} w_{ji}^{(0)} \tag{61}$$

Note that this equation does not depend on the specific form of $\frac{\partial E}{\partial x_i^{(0)}}$, whether it involves a sigmoid or any other activation function. We can therefore replace the specific indexes with general ones, and use this equation in the future.

$$\frac{\partial E}{\partial o_j^{(m+1)}} = \sum_i \frac{\partial E}{\partial x_i^{(m)}} w_{ji}^{(m)} \tag{62}$$

The second derivative of the error with respect to a neuron with output $o_j^{(1)}$ in the first hidden layer:

$$\frac{\partial^2 E}{\partial {o_j^{(1)}}^2} = \frac{\partial}{\partial o_j^{(1)}} \frac{\partial E}{\partial o_j^{(1)}} \tag{63}$$

$$= \frac{\partial}{\partial o_j^{(1)}} \sum_i \frac{\partial E}{\partial x_i^{(0)}} w_{ji}^{(0)} \tag{64}$$

$$= \frac{\partial}{\partial o_j^{(1)}} \sum_i \left(o_i^{(0)} - t_i\right) \sigma'\left(x_i^{(0)}\right) w_{ji}^{(0)} \tag{65}$$

If we now make use of the fact, that $o_i^{(0)} = \sigma\left(x_i^{(0)}\right) = \sigma\left(\sum_j \left(w_{ji}^{(0)} o_j^{(1)}\right)\right)$, we can evaluate the expression further.

$$\frac{\partial^2 E}{\partial {o_j^{(1)}}^2} = \frac{\partial}{\partial o_j^{(1)}} \sum_i \underbrace{\left(\sigma\left(\sum_j w_{ji}^{(0)} o_j^{(1)}\right) - t_i\right)}_{f\left(o_j^{(1)}\right)} \underbrace{\sigma'\left(\sum_j w_{ji}^{(0)} o_j^{(1)}\right) w_{ji}^{(0)}}_{g\left(o_j^{(1)}\right)} \tag{66}$$

$$= \sum_i \left(f'\left(o_j^{(1)}\right) g\left(o_j^{(1)}\right) + f\left(o_j^{(1)}\right) g'\left(o_j^{(1)}\right)\right) \tag{67}$$

$$= \sum_i \sigma'\left(\sum_j w_{ji}^{(0)} o_j^{(1)}\right) w_{ji}^{(0)}\, \sigma'\left(\sum_j w_{ji}^{(0)} o_j^{(1)}\right) w_{ji}^{(0)} + \ldots \tag{68}$$

$$\sum_i \left(\sigma\left(\sum_j w_{ji}^{(0)} o_j^{(1)}\right) - t_i\right) \sigma''\left(\sum_j w_{ji}^{(0)} o_j^{(1)}\right) \left(w_{ji}^{(0)}\right)^2 \tag{69}$$

$$= \sum_i \left(\left(\sigma'\left(x_i^{(0)}\right)\right)^2 \left(w_{ji}^{(0)}\right)^2 + \left(o_i^{(0)} - t_i\right) \sigma''\left(x_i^{(0)}\right) \left(w_{ji}^{(0)}\right)^2\right) \tag{70}$$

$$= \sum_i \underbrace{\left(\left(\sigma'\left(x_i^{(0)}\right)\right)^2 + \left(o_i^{(0)} - t_i\right) \sigma''\left(x_i^{(0)}\right)\right)}_{\text{from } (38)} \left(w_{ji}^{(0)}\right)^2 \tag{71}$$

Summing up, we obtain the more general expression:

$$\frac{\partial^2 E}{\partial {o_j^{(1)}}^2} = \sum_i \frac{\partial^2 E}{\partial {x_i^{(0)}}^2} \left(w_{ji}^{(0)}\right)^2 \tag{72}$$

Note that the equation in (72) does not depend on the form of $\frac{\partial^2 E}{\partial {x_x^{(0)}}^2}$, which means we can replace the specific indexes with general ones:

$$\frac{\partial^2 E}{\partial {o_j^{(m+1)}}^2} = \sum_i \frac{\partial^2 E}{\partial {x_i^{(m)}}^2} \left(w_{ji}^{(m)}\right)^2 \tag{73}$$

At this point we are beginning to see the recursion in the form of the 2nd derivative terms which can be thought of analogously to the first derivative recursion which is central to the back-propagation algorithm. The formulation above which makes specific reference to layer indexes also works in the general case.

Consider the ith neuron in any layer m with output $o_i^{(m)}$ and input $x_i^{(m)}$. The first and second derivatives of the error E with respect to this neuron's *input* are:

$$\frac{\partial E}{\partial x_i^{(m)}} = \frac{\partial E}{\partial o_i^{(m)}} \frac{\partial o_i^{(m)}}{\partial x_i^{(m)}} \tag{74}$$

$$\frac{\partial^2 E}{\partial {x_i^{(m)}}^2} = \frac{\partial}{\partial x_i^{(m)}} \frac{\partial E}{\partial x_i^{(m)}} \tag{75}$$

$$= \frac{\partial}{\partial x_i^{(m)}} \left(\frac{\partial E}{\partial o_i^{(m)}} \frac{\partial o_i^{(m)}}{\partial x_i^{(m)}} \right) \tag{76}$$

$$= \frac{\partial^2 E}{\partial x_i^{(m)} \partial o_i^{(m)}} \frac{\partial o_i^{(m)}}{\partial x_i^{(m)}} + \frac{\partial E}{\partial o_i^{(m)}} \frac{\partial^2 o_i^{(m)}}{\partial {x_i^{(m)}}^2} \tag{77}$$

$$= \frac{\partial}{\partial o_i^{(m)}} \left(\frac{\partial E}{\partial x_i^{(m)}} = \frac{\partial E}{\partial o_i^{(m)}} \frac{\partial o_i^{(m)}}{\partial x_i^{(m)}} \right) \frac{\partial o_i^{(m)}}{\partial x_i^{(m)}} + \frac{\partial E}{\partial o_i^{(m)}} \sigma'' \left(x_i^{(m)}\right) \tag{78}$$

$$= \frac{\partial^2 E}{\partial {o_i^{(m)}}^2} \left(\frac{\partial o_i^{(m)}}{\partial x_i^{(m)}} \frac{\partial o_i^{(m)}}{\partial x_i^{(m)}} \right) + \frac{\partial E}{\partial o_i^{(m)}} \sigma'' \left(x_i^{(m)}\right) \tag{79}$$

$$\frac{\partial^2 E}{\partial {x_i^{(m)}}^2} = \frac{\partial^2 E}{\partial {o_i^{(m)}}^2} \left(\sigma' \left(x_i^{(m)}\right)\right)^2 + \frac{\partial E}{\partial o_i^{(m)}} \sigma'' \left(x_i^{(m)}\right) \tag{80}$$

Note the form of this equation is the general form of what was derived for the output layer in (38). Both of the above first and second terms are easily computable and can be stored as we propagate back from the output of the network to the input. With respect to the output layer, the first and second derivative terms have already been derived above. In the case of the $m+1$ hidden layer during back propagation, there is a summation of terms calculated in the mth layer. For the first derivative, we have this from (62).

$$\frac{\partial E}{\partial o_j^{(m+1)}} = \sum_i \frac{\partial E}{\partial x_i^{(m)}} w_{ji}^{(m)} \tag{81}$$

And the second derivative for the jth neuron in the $m+1$ layer:

$$\frac{\partial^2 E}{\partial {x_j^{(m+1)}}^2} = \frac{\partial^2 E}{\partial {o_j^{(m+1)}}^2} \left(\sigma' \left(x_j^{(m+1)}\right)\right)^2 + \frac{\partial E}{\partial o_j^{(m+1)}} \sigma'' \left(x_j^{(m+1)}\right) \tag{82}$$

We can replace both derivative terms with the forms which depend on the previous layer:

$$\frac{\partial^2 E}{\partial {x_j^{(m+1)}}^2} = \underbrace{\sum_i \frac{\partial^2 E}{\partial {x_i^{(0)}}^2} \left(w_{ji}^{(0)}\right)^2}_{\text{from (73)}} \left(\sigma' \left(x_j^{(m+1)}\right)\right)^2 + \underbrace{\sum_i \frac{\partial E}{\partial x_i^{(m)}} w_{ji}^{(m)}}_{\text{from (62)}} \sigma'' \left(x_j^{(m+1)}\right) \tag{83}$$

And this horrible mouthful of an equation gives you a general form for any neuron in the jth position of the $m+1$ layer. Taking very careful note of the indexes, this can be more or less translated painlessly to code. You are welcome, world.

9.1.3 Summary Of Hidden Layer Derivatives

$$\frac{\partial E}{\partial o_j^{(m+1)}} = \sum_i \frac{\partial E}{\partial x_i^{(m)}} w_{ji}^{(m)} \qquad \frac{\partial^2 E}{\partial {o_j^{(m+1)}}^2} = \sum_i \frac{\partial^2 E}{\partial {x_i^{(m)}}^2} \left(w_{ji}^{(m)}\right)^2 \tag{84}$$

$$\frac{\partial E}{\partial x_i^{(m)}} = \frac{\partial E}{\partial o_i^{(m)}} \frac{\partial o_i^{(m)}}{\partial x_i^{(m)}} \tag{85}$$

$$\frac{\partial^2 E}{\partial {x_j^{(m+1)}}^2} = \frac{\partial^2 E}{\partial {o_j^{(m+1)}}^2} \left(\sigma'\left(x_j^{(m+1)}\right)\right)^2 + \frac{\partial E}{\partial o_j^{(m+1)}} \sigma''\left(x_j^{(m+1)}\right) \tag{86}$$

References

Balzer, Wolfgang, Takahashi, Masanobu, Ohta, Jun, and Kyuma, Kazuo. Weight quantization in boltzmann machines. *Neural Networks*, 4(3):405–409, 1991.

Baum, Eric B and Haussler, David. What size net gives valid generalization? *Neural computation*, 1(1):151–160, 1989.

Chauvin, Yves. Generalization performance of overtrained back-propagation networks. In *Neural Networks*, pp. 45–55. Springer, 1990.

Dundar, Gunhan and Rose, Kenneth. The effects of quantization on multilayer neural networks. *IEEE transactions on neural networks/a publication of the IEEE Neural Networks Council*, 6(6): 1446–1451, 1994.

Fahlman, Scott E and Lebiere, Christian. The cascade-correlation learning architecture. 1989.

Goodfellow, Ian J, Warde-Farley, David, Mirza, Mehdi, Courville, Aaron, and Bengio, Yoshua. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

Hassibi, Babak and Stork, David G. *Second order derivatives for network pruning: Optimal brain surgeon*. Morgan Kaufmann, 1993.

Hoehfeld, Markus and Fahlman, Scott E. Learning with limited numerical precision using the cascade-correlation algorithm. *IEEE Transactions on Neural Networks*, 3(4):602–611, 1992.

LeCun, Yann, Denker, John S, Solla, Sara A, Howard, Richard E, and Jackel, Lawrence D. Optimal brain damage. In *NIPs*, volume 89, 1989.

Mozer, Michael C and Smolensky, Paul. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In *Advances in neural information processing systems*, pp. 107–115, 1989a.

Mozer, Michael C and Smolensky, Paul. Using relevance to reduce network size automatically. *Connection Science*, 1(1):3–16, 1989b.

Reed, Russell. Pruning algorithms-a survey. *Neural Networks, IEEE Transactions on*, 4(5):740–747, 1993.

Segee, Bruce E and Carter, Michael J. Fault tolerance of pruned multilayer networks. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume 2, pp. 447–452. IEEE, 1991.

Sietsma, Jocelyn and Dow, Robert JF. Neural net pruning-why and how. In *Neural Networks, 1988., IEEE International Conference on*, pp. 325–333. IEEE, 1988.

Srivastava, Nitish, Hinton, Geoffrey, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.