# THE INCREDIBLE SHRINKING NEURAL NETWORK: NEW PERSPECTIVES ON LEARNING REPRESENTATIONS THROUGH THE LENS OF PRUNING

**Nikolas Wolfe, Aditya Sharma & Bhiksha Raj**
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
{nwolfe, bhiksha}@cs.cmu.edu, adityasharma@cmu.edu

**Lukas Drude**
Universitat Paderborn
drude@nt.upb.de

## ABSTRACT

How much can pruning algorithms teach us about the fundamentals of learning representations in neural networks? A lot, it turns out. Neural network model compression has become a topic of great interest in recent years, and many different techniques have been proposed to address this problem. In general, this is motivated by the idea that smaller models typically lead to better generalization. At the same time, the decision of what to prune and when to prune necessarily forces us to confront our assumptions about how neural networks actually learn to represent patterns in data. In this work we set out to test several long-held hypotheses about neural network learning representations and numerical approaches to pruning. To accomplish this we first reviewed the historical literature and derived a novel algorithm to prune whole neurons (as opposed to the traditional method of pruning weights) from optimally trained networks using a second-order Taylor method. We then set about testing the performance of our algorithm and analyzing the quality of the decisions it made. As a baseline for comparison we used a first-order Taylor method based on the Skeletonization algorithm and an exhaustive brute-force serial pruning algorithm. Our proposed algorithm worked well compared to a first-order method, but not nearly as well as the brute-force method. Our error analysis led us to question the validity of many widely-held assumptions behind pruning algorithms in general and the trade-offs we often make in the interest of reducing computational complexity. We discovered that there is a straightforward way, however expensive, to serially prune 40-60% of the neurons in a trained network with minimal effect on the learning representation and without any re-training.

## 1 INTRODUCTION

In this work we propose and evaluate a novel algorithm for pruning whole neurons from a trained neural network without any re-training and examine its performance compared to two simpler methods. We then analyze the kinds of errors made by our algorithm and use this as a stepping off point to launch an investigation into the fundamental nature of learning representations in neural networks. Our results corroborate an insightful though largely forgotten observation by Mozer & Smolensky (1989a) concerning the nature of neural network learning. This observation is best summarized in a quotation from Segee & Carter (1991) on the notion of fault-tolerance in multilayer perceptron networks:

> Contrary to the belief widely held, multilayer networks are *not* inherently fault tolerant. In fact, the loss of a single weight is frequently sufficient to completely

> disrupt a learned function approximation. Furthermore, having a large number of weights *does not seem* to improve fault tolerance. [Emphasis added]

Essentially, Mozer & Smolensky (1989b) observed that neural networks during training do *not* distribute the learning representation evenly or equitably across hidden units. What actually happens is that a few, elite neurons learn an approximation of the input-output function, and the remaining units must learn a complex interdependence function which cancels out their respective influence on the network output. Furthermore, assuming enough units exist to learn the function in question, increasing the number of parameters does not increase the richness or robustness of the learned approximation, but rather simply increases the likelihood of overfitting and the number of noisy parameters to be canceled during training. This is evinced by the fact that in many cases, multiple neurons can be removed from a network with no re-training and with negligible impact on the quality of the output approximation. In other words, there are few bipartisan units in a trained network. A unit is typically either part of the (possibly overfit) input-output function approximation, or it is part of an elaborate noise cancellation task force. Assuming this is the case, most of the compute-time spent training a neural network is likely occupied by this arguably wasteful procedure of silencing superfluous parameters, and pruning can be viewed as a necessary treatment procedure to "trim the fat."

We observed copious evidence of this phenomenon in our experiments, and this is the motivation behind our decision to evaluate the pruning algorithms in this study on the simple criteria of their ability to trim neurons *without* any re-training. If we were to employ re-training as part of our evaluation criteria, we would arguably *not* be evaluating the quality of our algorithm's pruning decisions per se but rather the ability of back-propagation trained networks to recover from faults caused by non-ideal pruning decisions, as suggested by the conclusions of Segee & Carter (1991) and Mozer & Smolensky (1989a). Moreover, as Fahlman & Lebiere (1989) discuss, due to the "herd effect" and "moving target" phenomena in back-propagation learning, the remaining units in a network will simply shift course to account for whatever error signal is re-introduced as a result of a bad pruning decision or network fault. So long as there are enough critical parameters to learn the function in question, a network can typically recover faults with additional training. This limits the conclusions we can draw about the quality of our pruning criteria when we employ re-training.

In terms of removing units without re-training, what we discovered is that predicting the behavior of a network when a unit is to be pruned is very difficult, and most of the approximation techniques put forth in existing pruning algorithms do not fare well at all when compared to a brute-force search. To begin our discussion of how we arrived at our algorithm and set up our experiments, we begin with a review of the existing literature.

## 2 Literature Review

Pruning algorithms, as comprehensively surveyed by Reed (1993), are a useful set of heuristics designed to identify and remove elements from a neural network which are either redundant or do not significantly contribute to the output of the network. This is motivated by the observed tendency of neural networks to overfit to the idiosyncrasies of their training data given too many trainable parameters or too few input patterns from which to generalize, as stated by Chauvin (1990).

Network architecture design and hyperparameter selection are inherently difficult tasks typically approached using a few well-known rules of thumb, e.g. various weight initialization procedures, choosing the width and number of layers, different activation functions, learning rates, momentum, etc. Some of this "black art" appears unavoidable. For problems which cannot be solved using linear threshold units alone, Baum & Haussler (1989) demonstrate that 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 seems to inhibit learning, and so in practice it is common to attempt to over-parameterize networks initially using a large number of hidden units and weights, and then prune or compress them afterwards if necessary. Of course, as the old saying goes, there's more than one way to skin a neural network.

### 2.1 NON-PRUNING BASED GENERALIZATION & COMPRESSION TECHNIQUES

The generalization behavior 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. Of course, while cascade correlation specifically tries to construct of minimal networks, many techniques to improve network generalization do not explicitly attempt to reduce the total number of parameters or the memory footprint of a trained network per se.

Model compression often has benefits with respect to generalization performance and the portability of neural networks to operate in memory-constrained or embedded environments. Without explicitly removing parameters from the network, weight quantization allows for a reduction in the number of bytes used to represent each weight parameter, as investigated by Balzer et al. (1991), Dundar & Rose (1994), and Hoehfeld & Fahlman (1992).

A recently proposed method for compressing recurrent neural networks (Prabhavalkar et al. (2016)) uses the singular values of a trained weight matrix as basis vectors from which to derive a compressed hidden layer. Øland & Raj (2015) successfully implemented network compression through weight quantization with an encoding step while others such as Han et al. (2016) have tried to expand on this by adding weight-pruning as a preceding step to quantization and encoding.

In summary, we can say that there are many different ways to improve network generalization by altering the training procedure, the objective error function, or by using compressed representations of the network parameters. But these are not, strictly speaking, examples of techniques to reduce the number of parameters in a network. For this we must employ some form of pruning criteria.

### 2.2 PRUNING TECHNIQUES

If we wanted to continually shrink a neural network down to minimum size, the most straightforward brute-force way to do it is to individually switch each element off and measure the increase in total error on the training set. We then pick the element which has the least impact on the total error, and remove it. Rinse and repeat. This is extremely computationally expensive, given a reasonably large neural network and training set. Alternatively, we might accomplish this using any number of much faster 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 algorithms, with one major variation: Instead of pruning individual weights, we prune entire neurons, thereby eliminating all of their incoming and outgoing weight parameters in one go, resulting in more memory saved, faster.

The algorithm developed for this paper is targeted at reducing the total number of neurons in a trained network, which is one way of reducing its computational memory footprint. This is often a desirable criteria to minimize in the case of resource-constrained or embedded devices, and also allows us to probe the limitations of pruning down to the very last essential network elements. In terms of generalization as well, we can measure the error of the network on the test set as each element is sequentially removed from the network. With an oracle pruning algorithm, what we expect to observe is that the output of the network remains stable as the first few superfluous neurons are removed, and as we start to bite into the more crucial members of the function approximation, the error should start to rise dramatically. In this paper, the brute-force approach described at the beginning of this section serves as a proxy for an oracle pruning algorithm.

One reason to choose to rank and prune individual neurons as opposed to weights is that there are far fewer elements to consider. Furthermore, 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. If we want to reduce the *size* of a network as efficiently as possible, we argue that pruning neurons instead of weights is more efficient computationally as well as practically in terms of quickly reaching a hypothetical target reduction in memory consumption. This approach also offers downstream applications a realistic expectation of the minimal increase in error resulting from the removal of a specified percentage of neurons. Such trade-offs are unavoidable, but performance impacts can be limited if a principled approach is used to find the best candidate neurons for removal.

It is well known that too many free parameters in a neural network can lead to overfitting. 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 thus the removal of any single hidden unit at random can actually result in a network fault. Mozer & Smolensky (1989b) argue 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's influence or begin over-fitting to the noise in the data. We leverage this idea in the current work to rank all neurons in pre-trained networks based on their effective contributions to the overall performance. We then remove the unnecessary neurons to reduce the network's footprint. Through our experiments we not only concretely validate the theory put forth by Mozer & Smolensky (1989b) but we also successfully build on it to prune networks to 40 to 60 % of their original size without any major loss in performance.

## 3 Pruning Neurons to Shrink Neural Networks

As discussed in Section 1 our aim here is to leverage the non-uniform distribution of the learning representation in pre-trained neural networks to eliminate excess neurons in their entirety, without focusing on individual weight parameters. Taking this approach enables us to remove all the weights (incoming and outgoing) associated with a non-contributing neuron in one go. We would like to note here that in an ideal scenario, based on the neuron interdependency theory put forward by Mozer & Smolensky (1989a), one would evaluate all possible combinations of neurons to remove (one at a time, two at a time, three at a time and so forth) to find the optimal subset of neurons to keep. This however is not feasible practically, which is why we will only focus on removing one neuron at a time and explore "greedy" algorithms to do this in an efficient manner.

The general approach taken to prune an optimally trained neural network here is to create a ranked list of all the neurons in the network based off of one of the 3 proposed ranking criteria: Brute Force approximation (which we use as our ground truth), linear approximation and quadratic approximation of the neuron's impact on the overall performance of the network. We then test the effects of removing neurons on the accuracy and error of the network. All the algorithms and methods presented here are easily parallelizable.

One last thing to note here before moving forward is that the methods discussed in this section involve some non-trivial mathematics, discussing all of which is beyond the scope of this paper as we wish to keep the math to a minimum in the main body of the paper. However, a complete step-by-step derivation and proof of all the results presented is provided in the Supplementary Material as an Appendix.

### 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 possibly unusable on large-scale neural networks with thousands of neurons. This method explicitly evaluates each neuron in the network. 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.

### 3.2 Taylor Series Representation of Error

Let us denote the total error from the optimally trained neural network for any given validation dataset by $E$. $E$ can be seen as a function of $O$, where $O$ is the output of any general neuron in the network. This error can be approximated at a particular neuron's output (say $O_k$) by using the 2nd order Taylor Series as,

$$\hat{E}(O) \approx E(O_k) + (O - O_k) \cdot \left.\frac{\partial E}{\partial O}\right|_{O_k} + 0.5 \cdot (O - O_k)^2 \cdot \left.\frac{\partial^2 E}{\partial O^2}\right|_{O_k}, \tag{1}$$

When a neuron is pruned, its output $O$ becomes 0.

Replacing $O$ by $O_k$ in equation 1 shows us that the error is approximated perfectly by equation 1 at $O_k$. So:

$$\Delta E_k = \hat{E}(0) - \hat{E}(O_k) = -O_k \cdot \frac{\partial E}{\partial O}\bigg|_{O_k} + 0.5 \cdot O_k^2 \cdot \frac{\partial^2 E}{\partial O^2}\bigg|_{O_k}, \tag{2}$$

where $\Delta E_k$ is the change in the total error of the network when exactly one neuron ($k$) is turned off. Most of the terms in this equation are fairly easy to compute, as we have $O_k$ already from the activations of the hidden units and we already compute $\frac{\partial E}{\partial O}|_{O_k}$ for each training instance during backpropagation. The $\frac{\partial^2 E}{\partial O^2}|_{O_k}$ terms are a little more difficult to compute. This is derived in the appendix and summarized in the sections below.

#### 3.2.1 Linear Approximation Approach

We can use equation 2 to get the linear error approximation of the change in error due to the $k$th neuron being turned off and represent it as $\Delta E_k^1$ as follows:

$$\Delta E_k^1 = -O_k \cdot \frac{\partial E}{\partial O}\bigg|_{O_k}, \tag{3}$$

The derivative term above is the first-order gradient which represents the change in error with respect to the output a given neuron. This term can be collected during back-propagation. As we shall see further in this section, linear approximations are not reliable indicators of change in error but they provide us with an interesting basis for comparison with the other methods discussed in this paper.

#### 3.2.2 Quadratic Approximation Approach

As above, we can use equation 2 to get the quadratic error approximation of the change in error due to the $k$th neuron being turned off and represent it as $\Delta E_k^2$ as follows:

$$\Delta E_k^2 = -O_k \cdot \frac{\partial E}{\partial O}\bigg|_{O_k} + 0.5 \cdot O_k^2 \cdot \frac{\partial^2 E}{\partial O^2}\bigg|_{O_k}, \tag{4}$$

The additional second-order gradient term appearing above represents the quadratic change in error with respect to the output of a given neuron. This term can be generated by performing back-propagation using second order derivatives. Collecting these quadratic gradients involves some non-trivial mathematics, the entire step-by-step derivation procedure of which is provided in the Supplementary Material as an Appendix.

### 3.3 Proposed Pruning Algorithm

Figure 1 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 neuron output as can be seen in the figure. The process of training a neural network is essentially the process of finding these minimizing output values for all the neurons in the network. Pruning this particular neuron (which translates to getting a zero output from it will result in a change in the total overall error. This change in error is represented by distance between the original minimum error (shown by the dashed line) and the top red arrow. This neuron is clearly a bad candidate for removal since removing it will result in a huge error increase.

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.

One thing to note here is 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

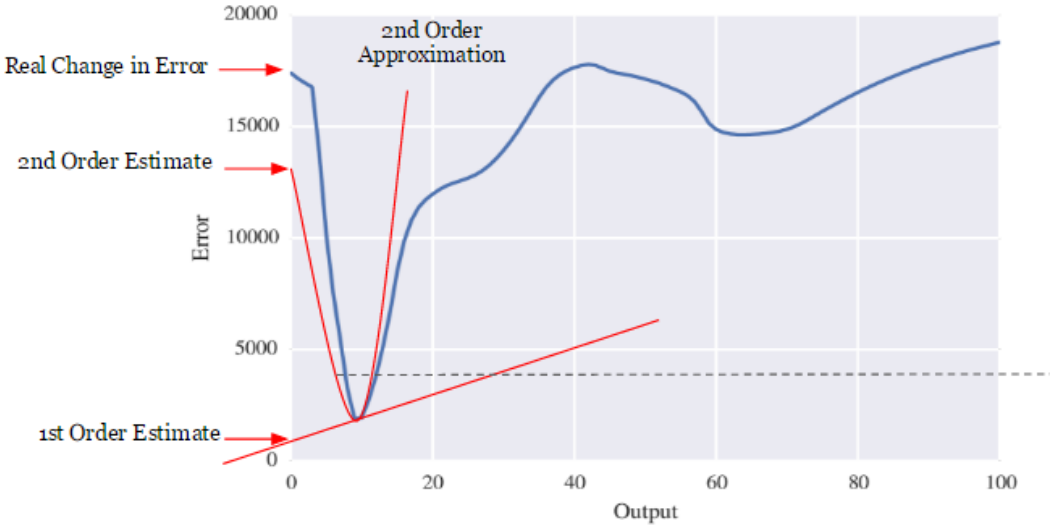


Figure 1: The intuition behind neuron pruning decision.

arise when the parabolic approximation undergoes a steep slope change. To take into account such cases, mean and median thresholding were employed, 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 $\Delta E_k$, the approximation of the change in error as the basis for the ranking. $\Delta E_k$ can be calculated using the Brute Force method, or one of the two Taylor Series approximations discussed previously.

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: maximum number of neurons to remove, percentage scaling needed, maximum allowable accuracy drop etc.

#### 3.3.1 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 $\Delta E_k$. This involves a single pass of second-order back-propagation (without weight updates) to collect the gradients for each neuron. The neurons from this rank-list (with the lowest values of $\Delta E_k$) are then pruned according to the stopping criterion decided. We note here that this algorithm is intentionally naive and is used for comparison only.

**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 $\Delta E_k$;
**while** *stopping criterion is not met* **do**
  remove the last ranked neuron ;
**end**

**Algorithm 1:** Single Overall Ranking

#### 3.3.2 Algorithm II: Iterative Re-Ranking

In this greedy variation of the algorithm (Algorithm 2), after each neuron removal, the remaining network undergoes a single forward and backward 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 on one another which would lead to a change in error contribution every time a dependent neuron is removed.

**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 $\Delta E_k$ ;
    remove the worst neuron based on the ranking ;
**end**

**Algorithm 2:** Iterative Re-Ranking

## 4 Experimental Results

For all the results presented in this section, the MNIST database of Handwritten Digits by LeCun & Cortes (2010) was used. It is worth noting that due to the time taken by the brute force algorithm we rather used a 5000 image subset of the MNIST database in which we have normalized the pixel values between 0 and 1.0, and compressed the image sizes to 20x20 images rather than 28x28, so the starting test accuracy reported here appears higher than those reported by LeCun et al. We do not believe that this affects the interpretation of the presented results because the basic learning problem does not change with a larger dataset or input dimension.

### 4.1 Pruning a 1-Layer Network

The network architecture in this case consisted of 1 layer, 100 neurons, 10 outputs, logistic sigmoid activations, and a starting test accuracy of 0.998.

#### 4.1.1 Single Overall Ranking Algorithm

We first present the results for a single-layer neural network in Figure 2, using the Single Overall algorithm (Algorithm 1) as proposed in Section 3. (We again note that this algorithm is intentionally naive and is used for comparison only. Its performance should be expected to be poor.) After training, each neuron is assigned its permanent ranking based on the three criteria discussed previously: A brute force "ground truth" ranking, and two approximations of this ranking using first and second order Taylor estimations of the change in network output error resulting from the removal of each neuron.

An interesting observation here is that with only a single layer, no criteria for ranking the neurons in the network (brute force or the two Taylor Series variants) using Algorithm 1 emerges superior, indicating that the 1st and 2nd order Taylor Series methods are actually reasonable approximations of the brute force method under certain conditions. Of course, this method is still quite bad in terms of the rate of degradation of the classification accuracy and in practice we would likely follow Algorithm 2 which is takes into account Mozer & Smolensky (1989a)'s observations stated in the Related Work section. The purpose of the present investigation, however, is to demonstrate how much of a trained network can be theoretically removed without altering the network's learned parameters in any way.

#### 4.1.2 Iterative Re-Ranking Algorithm

In Figure 3 we present our results using Algorithm 2 (The Iterative Re-Ranking Algorithm) in which all remaining neurons are re-ranked after each successive neuron is switched off. We compute the same brute force rankings and Taylor series approximations of error deltas over the remaining active

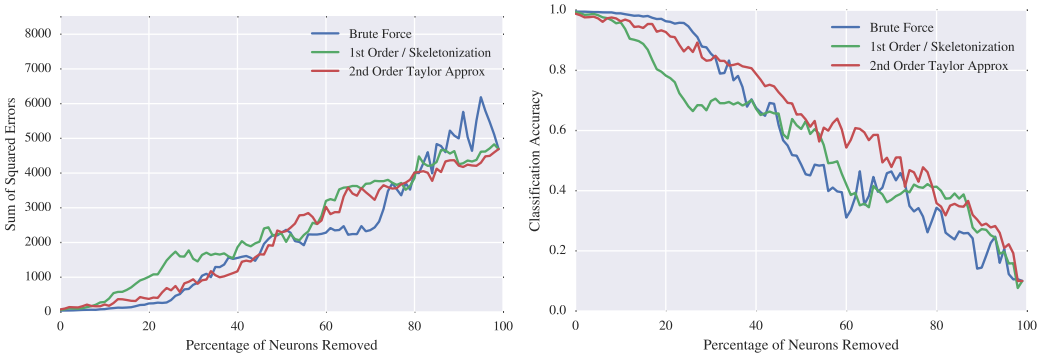


Figure 2: Degradation in squared error (left) and classification accuracy (right) after pruning a single-layer network using The Single Overall Ranking algorithm (**Network:** 1 layer, 100 neurons, 10 outputs, logistic sigmoid activation, starting test accuracy: 0.998)

neurons in the network after each pruning decision. This is intended to account for the effects of cancelling interactions between neurons.

There are 2 key observations here. Using the Brute Force ranking criteria, almost 60% of the neurons in the network can be pruned away without any major loss in performance. The other noteworthy observation here is that the 2nd order Taylor Series approximation of the error performs consistently better than its 1st order version.

#### 4.1.3 Visualization of Error Surface & Pruning Decisions

As explained in Section 3, these graphs are a visualization of the error surface of the network output with respect to the neurons chosen for removal using each of the 3 ranking criteria, represented in intervals of 10 neurons. In each graph, the error surface of the network output is displayed in log space (left) and in real space (right) with respect to each candidate neuron chosen for removal. We create these plots during the pruning exercise by picking a neuron to switch off, and then multiplying its output by a scalar gain value $\alpha$ which is adjusted from 0.0 to 10.0 with a step size of 0.001. When the value of $\alpha$ is 1.0, this represents the unperturbed neuron output learned during training. Between 0.0 and 1.0, we are graphing the literal effect of turning the neuron off ($\alpha = 0$), and when $\alpha > 1.0$ we are simulating a boosting of the neuron's influence in the network, i.e. inflating the value of its outgoing weight parameters.

We graph the effect of boosting the neuron's output to demonstrate that for certain neurons in the network, even doubling, tripling, or quadrupling the scalar output of the neuron has no effect on the

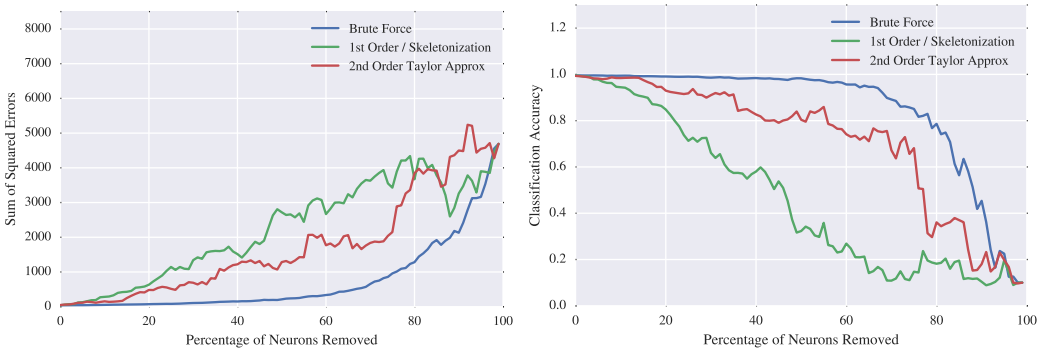


Figure 3: Degradation in squared error (left) and classification accuracy (right) after pruning a single-layer network the Iterative Re-ranking algorithm (**Network:** 1 layer, 100 neurons, 10 outputs, logistic sigmoid activation, starting test accuracy: 0.998)

overall error of the network, indicating the remarkable degree to which the network has learned to ignore the value of certain parameters. In other cases, we can get a sense of the sensitivity of the network's output to the value of a given neuron when the curve rises steeply after the red 1.0 line. This indicates that the learned value of the parameters emanating from a given neuron are relatively important, and this is why we should ideally see sharper upticks in the curves for the later-removed neurons in the network, that is, when the neurons crucial to the learning representation start to be picked off.

Some very interesting observations can be made in each of these graphs.

**Brute Force Criterion**

Notice how low to the floor and flat most of these curves are. It's only until the 90th removed neuron that we see a higher curve with a more convex shape (clearly a more influential piece of the network). This again validates the fact that a good majority of the neurons in a network do not contribute to the overall performance.

**1st Order Approximation Criterion**

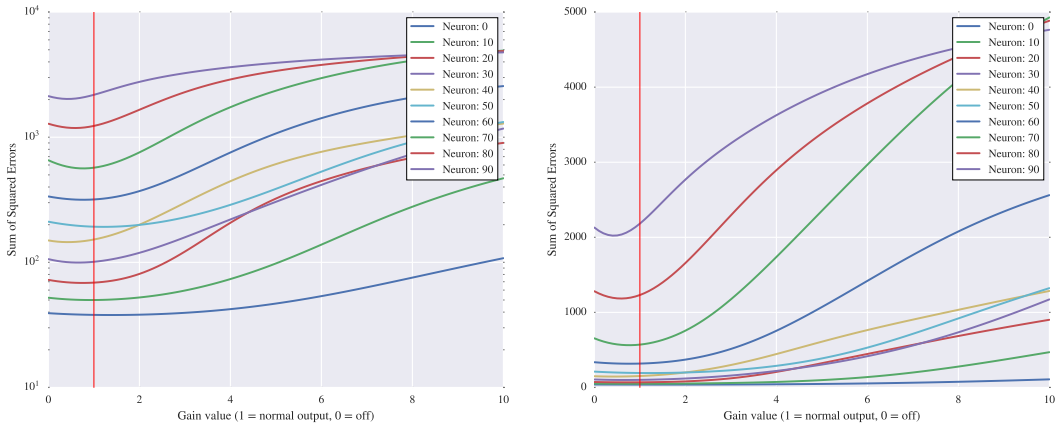


Figure 4: Error surface of the network output in log space (left) and in real space (right) with respect to each candidate neuron chosen for removal using the brute force criterion; (**Network:** 1 layer, 100 neurons, 10 outputs, logistic sigmoid activation, starting test accuracy: 0.998)

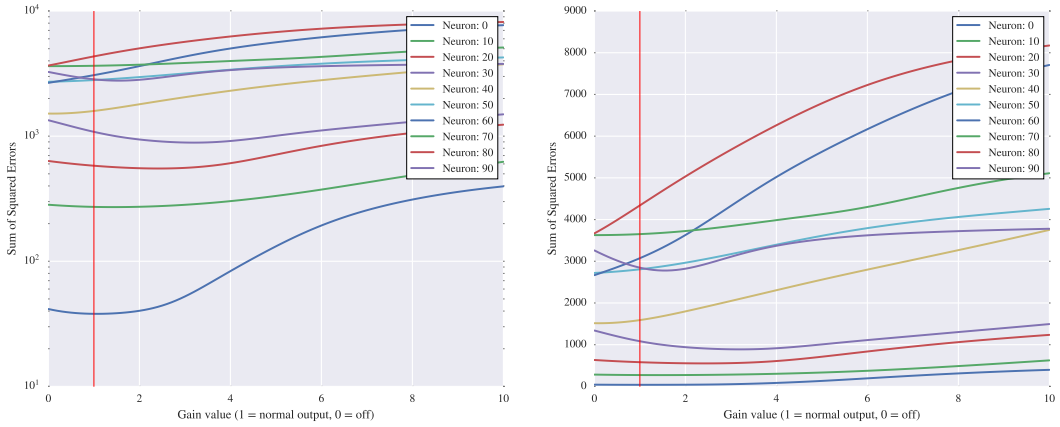


Figure 5: Error surface of the network output in log space (left) and in real space (right) with respect to each candidate neuron chosen for removal using the 1st order Taylor Series error approximation criterion; (**Network:** 1 layer, 100 neurons, 10 outputs, logistic sigmoid activation, starting test accuracy: 0.998)

It can be seen that most choices seem to have flat or negatively sloped curves, indicating that the first order approximation seems to be pretty good, but examining the brute force choices shows they could be better.

**2nd Order Approximation Criterion**

This method looks much more similar to the Brute Force method choices, though clearly not as good (they're more spread out). Notice the difference in convexity between the 2nd and 1st order method choices. It's clear that the first order method is fitting a line and the 2nd order method is fitting a parabola in their approximation.

### 4.2 Pruning A 2-Layer Network

The network architecture in this case consisted of 2 layers, 50 neurons per layer, 10 outputs, logistic sigmoid activations, and a starting test accuracy of 1.000.

#### 4.2.1 Single Overall Ranking Algorithm

Figure 7 shows the pruning results for Algorithm 1 on a 2-layer network. The ranking procedure is identical to the one used to generate Figure 2. (We again note that this algorithm is intentionally naive and is used for comparison only. Its performance should be expected to be poor.)

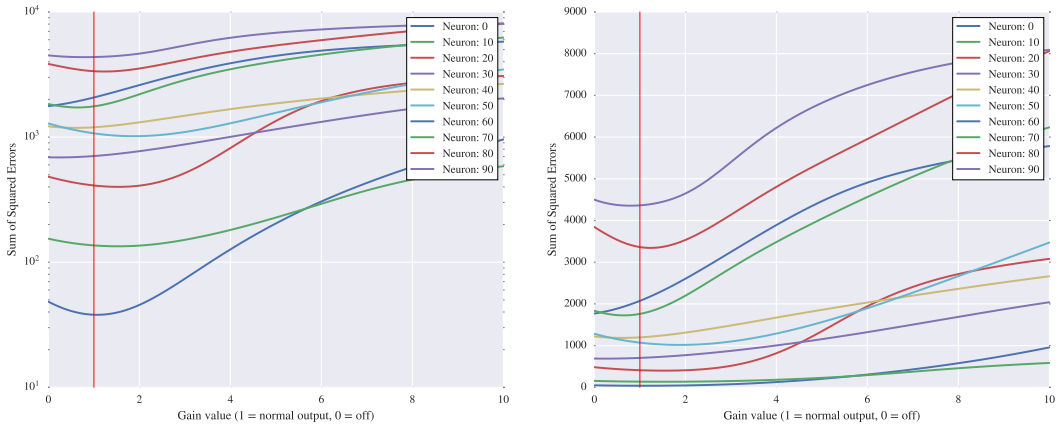


Figure 6: Error surface of the network output in log space (left) and in real space (right) with respect to each candidate neuron chosen for removal using the 2nd order Taylor Series error approximation criterion; (**Network:** 1 layer, 100 neurons, 10 outputs, logistic sigmoid activation, starting test accuracy: 0.998)

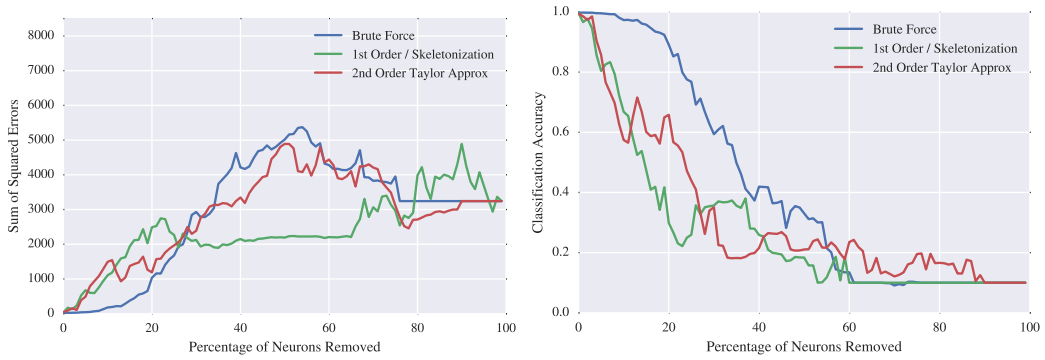


Figure 7: Degradation in squared error (left) and classification accuracy (right) after pruning a 2-layer network using the Single Overall Ranking algorithm; (**Network:** 2 layers, 50 neurons/layer, 10 outputs, logistic sigmoid activation, starting test accuracy: 1.000)

Unsurprisingly, a 2-layer network is harder to prune because a single overall ranking will never capture the interdependencies between neurons in different layers. It makes sense that this is much worse than the performance on the 1-layer network, even if this method is already known to be bad, and we'd likely never use it in practice.

#### 4.2.2 ITERATIVE RE-RANKING ALGORITHM

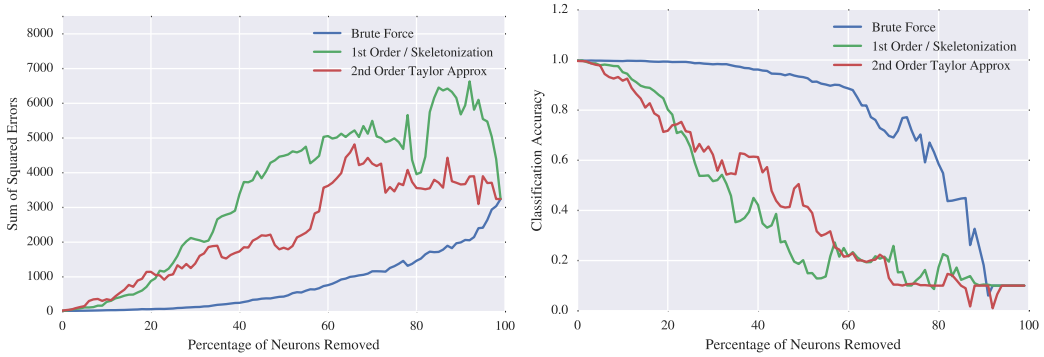


Figure 8: Degradation in squared error (left) and classification accuracy (right) after pruning a 2-layer network using the Iterative Re-ranking algorithm; (**Network:** 2 layers, 50 neurons/layer, 10 outputs, logistic sigmoid activation, starting test accuracy: 1.000)

Figure 8 shows the results from using Algorithm 2 on a 2-layer network. We compute the same brute force rankings and Taylor series approximations of error deltas over the remaining active neurons in the network after each pruning decision used to generate Figure 3. Again, this is intended to account for the effects of cancelling interactions between neurons.

It is clear that it becomes harder to remove neurons 1-by-1 with a deeper network (which makes sense because the neurons have more interdependencies in a deeper network), but we see an overall better performance with 2nd order method vs. 1st order, except for the first 20% of the neurons (but this doesn't seem to make much difference for classification accuracy.)

Perhaps a more important observation here is that even with a more complex network, it is possible to remove up to 40% of the neurons with no major loss in performance which is clearly illustrated by the brute force curve. This shows the clear potential of an ideal pruning technique and also shows how inconsistent 1st and 2nd order Taylor Series approximations of the error can be as ranking criteria.

#### 4.2.3 VISUALIZATION OF ERROR SURFACE & PRUNING DECISIONS

As seen in the case of a single layered network, these graphs are a visualization the error surface of the network output with respect to the neurons chosen for removal using each algorithm, represented in intervals of 10 neurons.

**Brute Force Criterion**

It is clear why these neurons got chosen, their graphs clearly show little change when neuron is removed, are mostly near the floor, and show convex behaviour of error surface, which argues for the rationalization of using 2nd order methods to estimate difference in error when they are turned off.

**1st Order Approximation Criterion**

Drawing a flat line at the point of each neurons intersection with the red vertical line (no change in gain) shows that the 1st derivative method is actually accurate for estimation of change in error in these cases, but still ultimately leads to poor decisions.

**2nd Order Approximation Criterion**

Clearly these neurons are not overtly poor candidates for removal (error doesn't change much between 1.0 & zero-crossing left-hand-side), but could be better (as described above in the Brute Force Criterion discussion).

### 4.3 Experiments on Toy Datasets

As can be seen from the experiments on MNIST, even though the 2nd-order approximation criterion is consistently better than 1st-order, it's performance is not nearly as good as brute force based ranking, especially beyond the first layer. What is interesting to note is that from some other experiments conducted on toy datasets (predicting whether a given point would lie inside a given shape on the Cartesian plane), the performance of the 2nd-order method was found to be exceptionally good and produced results very close to the brute force method. The 1st-order method, as expected, performed poorly here as well. Some of these results are illustrated in Figure 12. This huge variation in results from MNIST might be attributed to the fact that these toy datasets had only 2 output classes (as opposed to 10 classes in MNIST), but it certainly warrants further investigation.

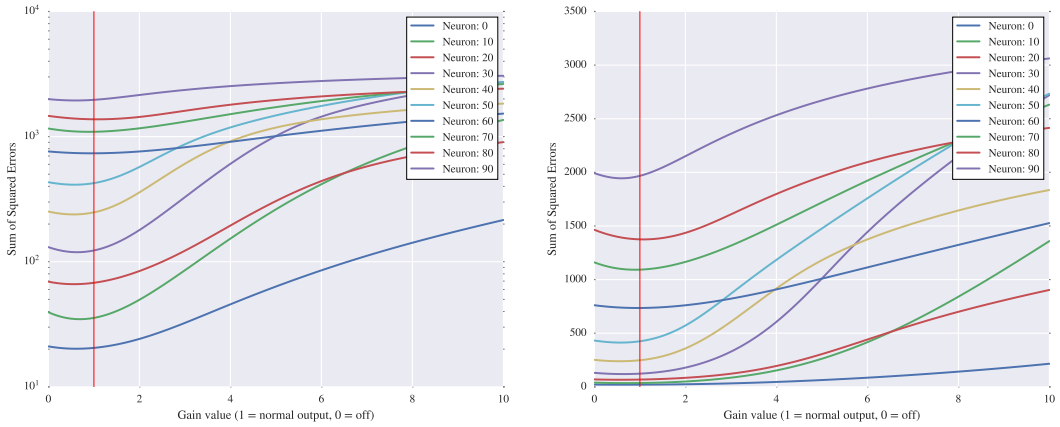


Figure 9: Error surface of the network output in log space (left) and in real space (right) with respect to each candidate neuron chosen for removal using the brute force criterion; (**Network:** 2 layers, 50 neurons/layer, 10 outputs, logistic sigmoid activation, starting test accuracy: 1.000)

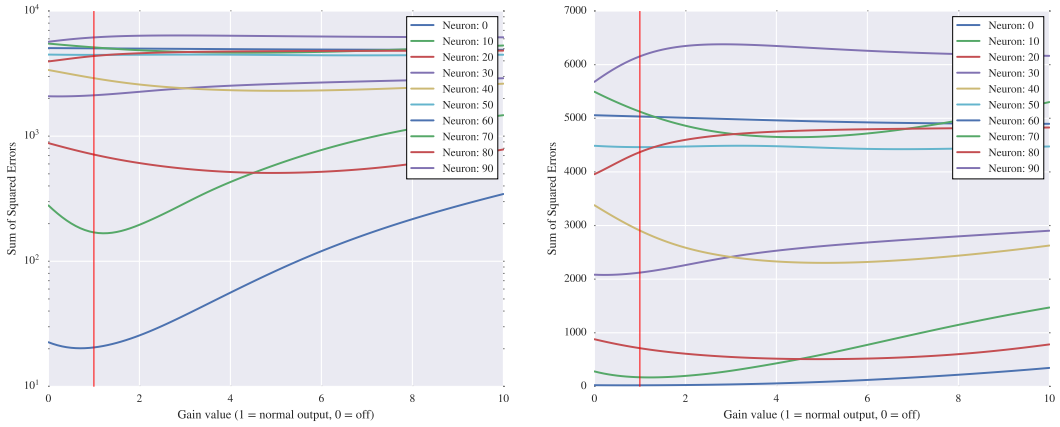


Figure 10: Error surface of the network output in log space (left) and in real space (right) with respect to each candidate neuron chosen for removal using the 1st order Taylor Series error approximation criterion; (**Network:** 2 layers, 50 neurons/layer, 10 outputs, logistic sigmoid activation, starting test accuracy: 1.000)

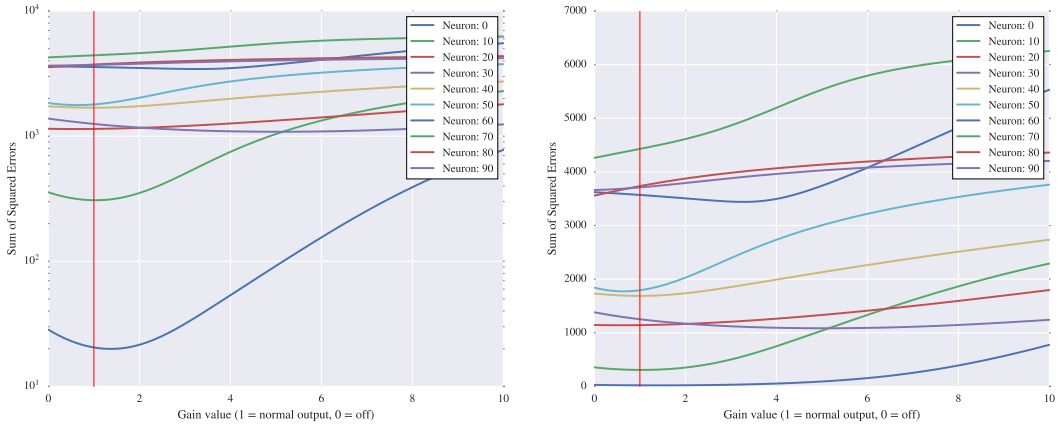


Figure 11: Error surface of the network output in log space (left) and in real space (right) with respect to each candidate neuron chosen for removal using the 2nd order Taylor Series error approximation criterion; (**Network:** 2 layers, 50 neurons/layer, 10 outputs, logistic sigmoid activation, starting test accuracy: 1.000)

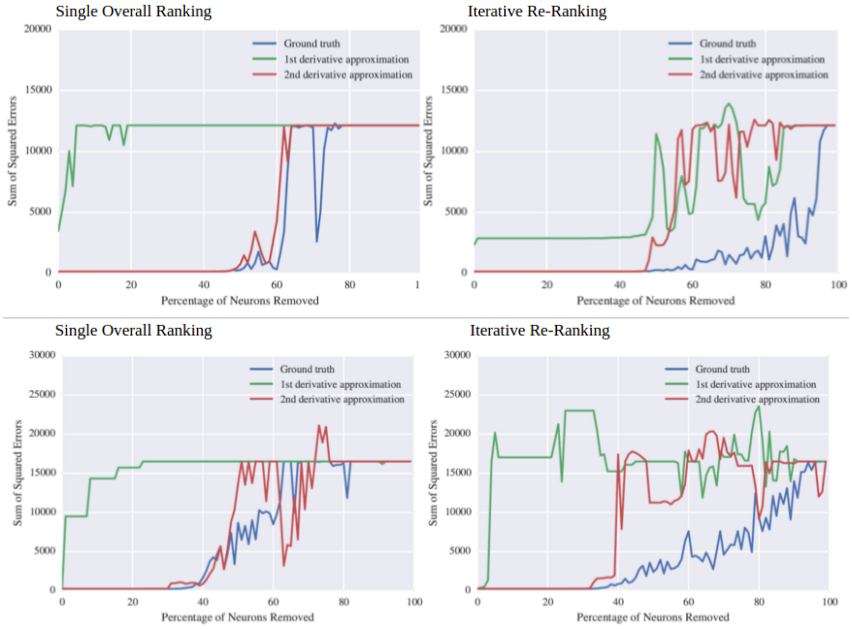


Figure 12: Degradation in squared error after pruning a 2-layer network using the Single Pass Algorithm (left) and the Iterative Re-ranking algorithm(right) on toy "diamond" shape dataset (above) and toy "random shape" dataset (below); (**Network:** 2 layers, 50 neurons/layer, 10 outputs, logistic sigmoid activation, starting test accuracy: 0.992(diamond); 0.986(random shape)

## 5 CONCLUSIONS & FUTURE WORK

Pruning neurons (instead of pruning individual weights) in a pre-trained neural network without seeing a major loss in performance is not only possible but also enables compressing networks to 40-80% of their original size, which is of great importance in constrained memory environments like embedded devices. This fact is established through the experiments using the brute force criterion, which if made computationally viable (through parallelization) or approximated more efficiently than the Taylor Series based methods discussed in this paper, can prove to be a useful compression tool. It would also be interesting to see how these methods perform on deeper networks and on some other popular and real world datasets. Also, as mentioned in Section 3, we have not considered all possible combinations of neuron interdependence in this work due to the in-feasibility of implementation and have always pruned one neuron at a time. Even though the greedy evaluation of such combinations is highly prohibitive, it is not entirely implausible to think that algorithms can be developed for it in the future. That would help in truly tapping into the power of these interconnections and hopefully lead to impressive performance results.

The experiments on the visualization of error surfaces and pruning decisions concretely establish the fact that not all neurons in a network contribute to its performance in the same way. This confirms the idea put forth by Mozer & Smolensky (1989b) that learning representation is not distributed uniformly across neurons. Neural networks use a few neurons to learn the function approximation, and the remaining neurons cooperate to cancel out each other's effects. This is also a strong indication of the fact that once training is done, bigger networks do not hold an advantage over smaller ones, which is similar to the idea put forth by Hinton et al. (2015) in their work on ensemble learning techniques.

## REFERENCES

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

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

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

Gunhan Dundar and Kenneth Rose. 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.

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

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

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149v5*, 2016.

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

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

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

Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

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

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

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

Anders Øland and Bhiksha Raj. Reducing communication overhead in distributed learning by an order of magnitude (almost). In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 2219–2223, 2015.

Rohit Prabhavalkar, Ouais Alsharif, Antoine Bruguier, and Lan McGraw. On the compression of recurrent neural networks with an application to lvcsr acoustic modeling for embedded speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5970–5974. IEEE, 2016.

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

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

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

APPENDIX

# A SECOND DERIVATIVE BACK-PROPAGATION

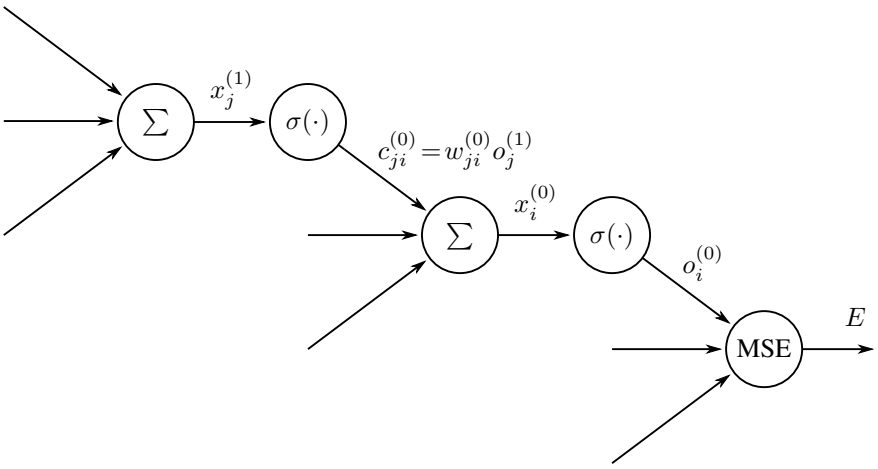


Figure 13: A computational graph of a simple feed-forward network illustrating the naming of different variables, where $\sigma(\cdot)$ is the nonlinearity, MSE is the mean-squared error cost function and $E$ is the overall loss.

Name and network definitions:

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

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. $o_i^{(m)}$ is the $i$th output in layer $m$ generated by the activation function $\sigma$, which in this paper is is the standard logistic sigmoid. $x_i^{(m)}$ is the weighted sum of inputs to the $i$th neuron in the $m$th layer, and $c_{ji}^{(m)}$ is the contribution of the $j$th neuron in the $m+1$ layer to the input of the $i$th neuron in the $m$th layer.

## A.1 FIRST AND SECOND DERIVATIVES

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

$$\frac{\partial E}{\partial o_i^{(0)}} = o_i^{(0)} - t_i \tag{6}$$

$$\frac{\partial^2 E}{\partial {o_i^{(0)}}^2} = 1 \tag{7}$$

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

$$\sigma'(x) = \sigma(x)\left(1 - \sigma(x)\right) \tag{8}$$

$$\sigma''(x) = \sigma'(x)\left(1 - 2\sigma(x)\right) \tag{9}$$

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

$$\sigma'(x) = \sigma(x)\left(1 - \sigma(x)\right) \tag{10}$$

$$\sigma''(x) = \frac{\mathrm{d}}{\mathrm{d}x} \underbrace{\sigma(x)}_{f(x)} \underbrace{(1 - \sigma(x))}_{g(x)} \tag{11}$$

$$\sigma''(x) = f'(x)g(x) + f(x)g'(x) \tag{12}$$

$$\sigma''(x) = \sigma'(x)(1 - \sigma(x)) - \sigma(x)\sigma'(x) \tag{13}$$

$$\sigma''(x) = \sigma'(x) - 2\sigma(x)\sigma'(x) \tag{14}$$

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

And for future convenience:

$$\frac{\mathrm{d}o_i^{(m)}}{\mathrm{d}x_i^{(m)}} = \frac{\mathrm{d}}{\mathrm{d}x_i^{(m)}} \left( o_i^{(m)} = \sigma(x_i^{(m)}) \right) \tag{16}$$

$$= \left( o_i^{(m)} \right) \left( 1 - o_i^{(m)} \right) \tag{17}$$

$$= \sigma' \left( x_i^{(m)} \right) \tag{18}$$

$$\frac{\mathrm{d}^2 o_i^{(m)}}{\mathrm{d}{x_i^{(m)}}^2} = \frac{\mathrm{d}}{\mathrm{d}x_i^{(m)}} \left( \frac{\mathrm{d}o_i^{(m)}}{\mathrm{d}x_i^{(m)}} = \left( o_i^{(m)} \right) \left( 1 - o_i^{(m)} \right) \right) \tag{19}$$

$$= \left( o_i^{(m)} \left( 1 - o_i^{(m)} \right) \right) \left( 1 - 2o_i^{(m)} \right) \tag{20}$$

$$= \sigma'' \left( x_i^{(m)} \right) \tag{21}$$

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

$$\frac{\partial E}{\partial x_i^{(0)}} = \frac{\partial E}{\partial o_i^{(0)}} \frac{\partial o_i^{(0)}}{\partial x_i^{(0)}} \tag{22}$$

$$= \underbrace{\left( o_i^{(0)} - t_i \right)}_{\text{from } (6)} \underbrace{\sigma \left( x_i^{(0)} \right) \left( 1 - \sigma \left( x_i^{(0)} \right) \right)}_{\text{from } (8)} \tag{23}$$

$$= \left( o_i^{(0)} - t_i \right) \left( o_i^{(0)} \left( 1 - o_i^{(0)} \right) \right) \tag{24}$$

$$= \left( o_i^{(0)} - t_i \right) \sigma' \left( x_i^{(0)} \right) \tag{25}$$

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

$$\frac{\partial^2 E}{\partial {x_i^{(0)}}^2} = \frac{\partial}{\partial x_i^{(0)}} \left( \frac{\partial E}{\partial o_i^{(0)}} \frac{\partial o_i^{(0)}}{\partial x_i^{(0)}} \right) \tag{26}$$

$$= \frac{\partial^2 E}{\partial x_i^{(0)} \partial o_i^{(0)}} \frac{\partial o_i^{(0)}}{\partial x_i^{(0)}} + \frac{\partial E}{\partial o_i^{(0)}} \frac{\partial^2 o_i^{(0)}}{\partial {x_i^{(0)}}^2} \tag{27}$$

$$= \frac{\partial^2 E}{\partial x_i^{(0)} \partial o_i^{(0)}} \underbrace{\left( o_i^{(0)} \left( 1 - o_i^{(0)} \right) \right)}_{\text{from } (8)} + \underbrace{\left( o_i^{(0)} - t_i \right)}_{\text{from } (6)} \underbrace{\left( o_i^{(0)} \left( 1 - o_i^{(0)} \right) \right) \left( 1 - 2 o_i^{(0)} \right)}_{\text{from } (9)} \tag{28}$$

$$\left( \frac{\partial^2 E}{\partial x_i^{(0)} \partial o_i^{(0)}} \right) = \frac{\partial}{\partial x_i^{(0)}} \frac{\partial E}{\partial o_i^{(0)}} = \frac{\partial}{\partial x_i^{(0)}} \underbrace{\left( o_i^{(0)} - t_i \right)}_{\text{from } (6)} = \frac{\partial o_i^{(0)}}{\partial x_i^{(0)}} = \underbrace{\left( o_i^{(0)} \left( 1 - o_i^{(0)} \right) \right)}_{\text{from } (8)} \tag{29}$$

$$\frac{\partial^2 E}{\partial {x_i^{(0)}}^2} = \left( o_i^{(0)} \left( 1 - o_i^{(0)} \right) \right)^2 + \left( o_i^{(0)} - t_i \right) \left( o_i^{(0)} \left( 1 - o_i^{(0)} \right) \right) \left( 1 - 2 o_i^{(0)} \right) \tag{30}$$

$$= \left( \sigma' \left( x_i^{(0)} \right) \right)^2 + \left( o_i^{(0)} - t_i \right) \sigma'' \left( x_i^{(0)} \right) \tag{31}$$

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:

$$\frac{\partial E}{\partial c_{ji}^{(0)}} = \frac{\partial E}{\partial o_i^{(0)}} \frac{\partial o_i^{(0)}}{\partial x_i^{(0)}} \frac{\partial x_i^{(0)}}{\partial c_{ji}^{(0)}} \tag{32}$$

$$= \underbrace{\left( o_i^{(0)} - t_i \right)}_{\text{from } (6)} \underbrace{\left( o_i^{(0)} \left( 1 - o_i^{(0)} \right) \right)}_{\text{from } (8)} \frac{\partial x_i^{(0)}}{\partial c_{ji}^{(0)}} \tag{33}$$

$$\left( \frac{\partial x_i^{(m)}}{\partial c_{ji}^{(m)}} \right) = \frac{\partial}{\partial c_{ji}^{(m)}} \left( x_i^{(m)} = \sum_j w_{ji}^{(m)} o_j^{(m+1)} \right) = \frac{\partial}{\partial c_{ji}^{(m)}} \left( c_{ji}^{(m)} + k \right) = 1 \tag{34}$$

$$\frac{\partial E}{\partial c_{ji}^{(0)}} = \left( o_i^{(0)} - t_i \right) \left( o_i^{(0)} \left( 1 - o_i^{(0)} \right) \right) \tag{35}$$

$$= \underbrace{\left( o_i^{(0)} - t_i \right) \sigma' \left( x_i^{(0)} \right)}_{\text{from } (25)} \tag{36}$$

$$\frac{\partial E}{\partial c_{ji}^{(0)}} = \frac{\partial E}{\partial x_i^{(0)}} \tag{37}$$

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

$$\frac{\partial^2 E}{\partial {c_{ji}^{(0)}}^2} = \frac{\partial}{\partial c_{ji}^{(0)}} \left( \frac{\partial E}{\partial c_{ji}^{(0)}} = \underbrace{\left(o_i^{(0)} - t_i\right) \sigma'\left(x_i^{(0)}\right)}_{\text{from (36)}} \right) \tag{38}$$

$$= \frac{\partial}{\partial c_{ji}^{(0)}} \left(\sigma\left(x_i^{(0)}\right) - t_i\right) \sigma'\left(x_i^{(0)}\right) \tag{39}$$

$$= \frac{\partial}{\partial c_{ji}^{(0)}} \left(\sigma\left(\sum_j w_{ji}^{(m)} o_j^{(m+1)}\right) - t_i\right) \sigma'\left(\sum_j w_{ji}^{(m)} o_j^{(m+1)}\right) \tag{40}$$

$$= \frac{\partial}{\partial c_{ji}^{(0)}} \left(\sigma\left(\sum_j c_{ji}^{(0)}\right) - t_i\right) \sigma'\left(\sum_j c_{ji}^{(0)}\right) \tag{41}$$

$$= \frac{\partial}{\partial c_{ji}^{(0)}} \underbrace{\left(\sigma\left(c_{ji}^{(0)} + k\right) - t_i\right)}_{f\left(c_{ji}^{(0)}\right)} \underbrace{\sigma'\left(c_{ji}^{(0)} + k\right)}_{g\left(c_{ji}^{(0)}\right)} \tag{42}$$

We now make use of the abbreviations $f$ and $g$:

$$= f'\left(c_{ji}^{(0)}\right) g\left(c_{ji}^{(0)}\right) + f\left(c_{ji}^{(0)}\right) g'\left(c_{ji}^{(0)}\right) \tag{43}$$

$$= \sigma'\left(c_{ji}^{(0)} + k\right) \sigma'\left(c_{ji}^{(0)} + k\right) + \left(\sigma\left(c_{ji}^{(0)} + k\right) - t_i\right) \sigma''\left(c_{ji}^{(0)} + k\right) \tag{44}$$

$$= \sigma'\left(c_{ji}^{(0)} + k\right)^2 + \left(o_i^{(0)} - t_i\right) \sigma''\left(c_{ji}^{(0)} + k\right) \tag{45}$$

$$\left(c_{ji}^{(0)} + k = \sum_j c_{ji}^{(0)} = \sum_j w_{ji}^{(m)} o_j^{(m+1)} = x_i^{(0)}\right) \tag{46}$$

$$\frac{\partial^2 E}{\partial {c_{ji}^{(0)}}^2} = \underbrace{\left(\sigma'\left(x_i^{(0)}\right)\right)^2 + \left(o_i^{(0)} - t_i\right) \sigma''\left(x_i^{(0)}\right)}_{\text{from (31)}} \tag{47}$$

$$\frac{\partial^2 E}{\partial {c_{ji}^{(0)}}^2} = \frac{\partial^2 E}{\partial {x_i^{(0)}}^2} \tag{48}$$

### A.1.1 Summary Of Output Layer Derivatives

$$\frac{\partial E}{\partial o_i^{(0)}} = o_i^{(0)} - t_i \qquad \frac{\partial^2 E}{\partial {o_i^{(0)}}^2} = 1 \tag{49}$$

$$\frac{\partial E}{\partial x_i^{(0)}} = \left(o_i^{(0)} - t_i\right) \sigma'\left(x_i^{(0)}\right) \qquad \frac{\partial^2 E}{\partial {x_i^{(0)}}^2} = \left(\sigma'\left(x_i^{(0)}\right)\right)^2 + \left(o_i^{(0)} - t_i\right) \sigma''\left(x_i^{(0)}\right) \tag{50}$$

$$\frac{\partial E}{\partial c_{ji}^{(0)}} = \frac{\partial E}{\partial x_i^{(0)}} \qquad \frac{\partial^2 E}{\partial {c_{ji}^{(0)}}^2} = \frac{\partial^2 E}{\partial {x_i^{(0)}}^2} \tag{51}$$

### A.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:

$$\frac{\partial E}{\partial o_j^{(1)}} = \sum_i \frac{\partial E}{\partial o_i^{(0)}} \frac{\partial o_i^{(0)}}{\partial x_i^{(0)}} \frac{\partial x_i^{(0)}}{\partial c_{ji}^{(0)}} \frac{\partial c_{ji}^{(0)}}{\partial o_j^{(1)}} = \sum_i \underbrace{\left(o_i^{(0)} - t_i\right) \sigma'\left(x_i^{(0)}\right)}_{\text{from (25)}} w_{ji}^{(0)} \tag{52}$$

$$\frac{\partial c_{ji}^{(m)}}{\partial o_j^{(m+1)}} = \frac{\partial}{\partial o_j^{(m+1)}} \left(c_{ji}^{(m)} = w_{ji}^{(m)} o_j^{(m+1)}\right) = w_{ji}^{(m)} \tag{53}$$

$$\frac{\partial E}{\partial o_j^{(1)}} = \sum_i \frac{\partial E}{\partial x_i^{(0)}} w_{ji}^{(0)} \tag{54}$$

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.

$$\frac{\partial E}{\partial o_j^{(m+1)}} = \sum_i \frac{\partial E}{\partial x_i^{(m)}} w_{ji}^{(m)} \tag{55}$$

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

$$\frac{\partial^2 E}{\partial {o_j^{(1)}}^2} = \frac{\partial}{\partial o_j^{(1)}} \frac{\partial E}{\partial o_j^{(1)}} \tag{56}$$

$$= \frac{\partial}{\partial o_j^{(1)}} \sum_i \frac{\partial E}{\partial x_i^{(0)}} w_{ji}^{(0)} \tag{57}$$

$$= \frac{\partial}{\partial o_j^{(1)}} \sum_i \left(o_i^{(0)} - t_i\right) \sigma'\left(x_i^{(0)}\right) w_{ji}^{(0)} \tag{58}$$

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.

$$\frac{\partial^2 E}{\partial {o_j^{(1)}}^2} = \frac{\partial}{\partial o_j^{(1)}} \sum_i \underbrace{\left(\sigma\left(\sum_j w_{ji}^{(0)} o_j^{(1)}\right) - t_i\right)}_{f\left(o_j^{(1)}\right)} \underbrace{\sigma'\left(\sum_j w_{ji}^{(0)} o_j^{(1)}\right)}_{g\left(o_j^{(1)}\right)} w_{ji}^{(0)} \tag{59}$$

$$= \sum_i \left(f'\left(o_j^{(1)}\right) g\left(o_j^{(1)}\right) + f\left(o_j^{(1)}\right) g'\left(o_j^{(1)}\right)\right) \tag{60}$$

$$= \sum_i \sigma'\left(\sum_j w_{ji}^{(0)} o_j^{(1)}\right) w_{ji}^{(0)}\ \sigma'\left(\sum_j w_{ji}^{(0)} o_j^{(1)}\right) w_{ji}^{(0)} + \ldots \tag{61}$$

$$\sum_i \left(\sigma\left(\sum_j w_{ji}^{(0)} o_j^{(1)}\right) - t_i\right) \sigma''\left(\sum_j w_{ji}^{(0)} o_j^{(1)}\right) \left(w_{ji}^{(0)}\right)^2 \tag{62}$$

$$= \sum_i \left(\left(\sigma'\left(x_i^{(0)}\right)\right)^2 \left(w_{ji}^{(0)}\right)^2 + \left(o_i^{(0)} - t_i\right) \sigma''\left(x_i^{(0)}\right) \left(w_{ji}^{(0)}\right)^2\right) \tag{63}$$

$$= \sum_i \underbrace{\left(\left(\sigma'\left(x_i^{(0)}\right)\right)^2 + \left(o_i^{(0)} - t_i\right) \sigma''\left(x_i^{(0)}\right)\right)}_{\text{from (31)}} \left(w_{ji}^{(0)}\right)^2 \tag{64}$$

Summing up, we obtain the more general expression:

$$\frac{\partial^2 E}{\partial o_j^{(1)^2}} = \sum_i \frac{\partial^2 E}{\partial x_i^{(0)^2}} \left(w_{ji}^{(0)}\right)^2 \tag{65}$$

Note that the equation in (65) 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:

$$\frac{\partial^2 E}{\partial o_j^{(m+1)^2}} = \sum_i \frac{\partial^2 E}{\partial x_i^{(m)^2}} \left(w_{ji}^{(m)}\right)^2 \tag{66}$$

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 $i$th 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:

$$\frac{\partial E}{\partial x_i^{(m)}} = \frac{\partial E}{\partial o_i^{(m)}} \frac{\partial o_i^{(m)}}{\partial x_i^{(m)}} \tag{67}$$

$$\frac{\partial^2 E}{\partial x_i^{(m)^2}} = \frac{\partial}{\partial x_i^{(m)}} \frac{\partial E}{\partial x_i^{(m)}} \tag{68}$$

$$= \frac{\partial}{\partial x_i^{(m)}} \left( \frac{\partial E}{\partial o_i^{(m)}} \frac{\partial o_i^{(m)}}{\partial x_i^{(m)}} \right) \tag{69}$$

$$= \frac{\partial^2 E}{\partial x_i^{(m)} \partial o_i^{(m)}} \frac{\partial o_i^{(m)}}{\partial x_i^{(m)}} + \frac{\partial E}{\partial o_i^{(m)}} \frac{\partial^2 o_i^{(m)}}{\partial x_i^{(m)^2}} \tag{70}$$

$$= \frac{\partial}{\partial o_i^{(m)}} \left( \frac{\partial E}{\partial x_i^{(m)}} = \frac{\partial E}{\partial o_i^{(m)}} \frac{\partial o_i^{(m)}}{\partial x_i^{(m)}} \right) \frac{\partial o_i^{(m)}}{\partial x_i^{(m)}} + \frac{\partial E}{\partial o_i^{(m)}} \sigma'' \left( x_i^{(m)} \right) \tag{71}$$

$$= \frac{\partial^2 E}{\partial o_i^{(m)^2}} \left( \frac{\partial o_i^{(m)}}{\partial x_i^{(m)}} \frac{\partial o_i^{(m)}}{\partial x_i^{(m)}} \right) + \frac{\partial E}{\partial o_i^{(m)}} \sigma'' \left( x_i^{(m)} \right) \tag{72}$$

$$\frac{\partial^2 E}{\partial x_i^{(m)^2}} = \frac{\partial^2 E}{\partial o_i^{(m)^2}} \left( \sigma' \left( x_i^{(m)} \right) \right)^2 + \frac{\partial E}{\partial o_i^{(m)}} \sigma'' \left( x_i^{(m)} \right) \tag{73}$$

Note the form of this equation is the general form of what was derived for the output layer in (31). 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 $m$th layer. For the first derivative, we have this from (55).

$$\frac{\partial E}{\partial o_j^{(m+1)}} = \sum_i \frac{\partial E}{\partial x_i^{(m)}} w_{ji}^{(m)} \tag{74}$$

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

$$\frac{\partial^2 E}{\partial x_j^{(m+1)^2}} = \frac{\partial^2 E}{\partial o_j^{(m+1)^2}} \left( \sigma' \left( x_j^{(m+1)} \right) \right)^2 + \frac{\partial E}{\partial o_j^{(m+1)}} \sigma'' \left( x_j^{(m+1)} \right) \tag{75}$$

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

$$\frac{\partial^2 E}{\partial x_j^{(m+1)^2}} = \underbrace{\sum_i \frac{\partial^2 E}{\partial x_i^{(0)^2}} \left(w_{ji}^{(0)}\right)^2}_{\text{from (66)}} \left( \sigma' \left( x_j^{(m+1)} \right) \right)^2 + \underbrace{\sum_i \frac{\partial E}{\partial x_i^{(m)}} w_{ji}^{(m)}}_{\text{from (55)}} \sigma'' \left( x_j^{(m+1)} \right) \tag{76}$$

And this horrible mouthful of an equation gives you a general form for any neuron in the $j$th 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.

### A.1.3 SUMMARY OF HIDDEN LAYER DERIVATIVES

$$\frac{\partial E}{\partial o_j^{(m+1)}} = \sum_i \frac{\partial E}{\partial x_i^{(m)}} w_{ji}^{(m)} \qquad \frac{\partial^2 E}{\partial o_j^{(m+1)^2}} = \sum_i \frac{\partial^2 E}{\partial x_i^{(m)^2}} \left(w_{ji}^{(m)}\right)^2 \tag{77}$$

$$\frac{\partial E}{\partial x_i^{(m)}} = \frac{\partial E}{\partial o_i^{(m)}} \frac{\partial o_i^{(m)}}{\partial x_i^{(m)}} \tag{78}$$

$$\frac{\partial^2 E}{\partial x_j^{(m+1)^2}} = \frac{\partial^2 E}{\partial o_j^{(m+1)^2}} \left(\sigma'\left(x_j^{(m+1)}\right)\right)^2 + \frac{\partial E}{\partial o_j^{(m+1)}} \sigma''\left(x_j^{(m+1)}\right) \tag{79}$$