Hessian-Free Optimization and its applications to Neural Networks

Joseph Christianson

June 7, 2016

Abstract

Neural Networks are a simple model of biological cognitive processes. In this paper we introduce how the Neural Network operates, summarizing key sections of A.C.C. Coolen's large introductory paper. We examine their biological motivations, how they process data, and the default training method of gradient descent and backpropogation. We then analyze an optimization to the traditional structure provided by James Martens. His technique improves upon gradient descent by employing second order derivatives and he introduces a numerically stable way to calculate them.

1 Biological Motivation

Neural Networks are an abstraction of the way we understand brains process information. Brains are composed of huge networks of individual neurons. Brains neurons act as individual processors. They sit in a ion rich fluid that causes there to be an electrical gradient between the inside of the neuron cell and the outside. When a neuron is electrically stimulated, pumps open and the fluid enters the cell. This causes a generation of electrical current which spreads along the membrane. This action is called firing. In this way a signal can propagate from neuron to neuron.

Neurons are connected by dendrites, synapses, and an axon. Each neuron only as one axon. When a neuron fires, the signal travels down its membrane. Dendrites are thread-like membranes that bring signals to a neuron. They are connected to axons of other neurons via synapses. Synapses have a synaptic cleft, were the cell membranes interface without being contiguous. Each synapse stores a certain amount of positive and negatively charged fluid called neurotransmitter. When a signal reaches a synapse its neurotransmitter is released. Positively charged neurotransmitter provides an excitatory signal, while negatively charged provides an inhibitory one. The amount of neurotransmitter released modulates the strength of the signal passed along to the dendrite, and so is called a synaptic weight.

Overall brains are simply huge networks of these neurons. The network structure and synaptic weights store the "program" of the brain. This includes both behavior and memory. The brain continuously adapts by adjusting the weights and connections present. Neurons themselves are somewhat unreliable as they fire as a result of input, but also sometimes spontaneously. This results in a high degree of redundancy in structure and makes the brain more resilient and adaptable.

In drawing a comparison to digital computers, rather than having a few deterministic, powerful processors, a brain is a massively parallel network of low power stochastic processors. Additionally, where as computers generally keep program instructions separate from memory, brains do not differentiate.

2 Modelling

With the above description in mind, we now tackle creating a simple model of neural activity. Suppose we have N neurons. Then at time t_0 we express the input signal to the ith neuron as:

$$\text{input}_i = w_1 S_1 + ... + w_N S_N,$$

where w_j is the synaptic weight connecting the jth neuron to the ith neuron. S_j is the signal generated by the jth neuron. If the ith neuron has some signal threshold, θ_i over which it fires then we can create a function for activation:

$$S_i(t+1) = \begin{cases} 1 & : \quad w_1 S_1(t) + ... + w_N S_N > \theta_i \\ 0 & : \quad w_1 S_1 + ... + w_N S_N \leq \theta_i \end{cases}$$

The final step is to decide if we apply this function to all neurons at once in a single time step, to randomly choose neurons to update, or to update single neurons in a set sequence. A neural network then is just a set of binary values, updated by this rule.

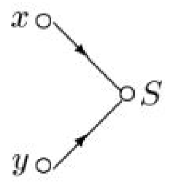
3 Computability

From this simple model we can glean a powerful piece of information. Neural networks as we have defined them are universal computing machines. Coolen's proof for this is to show that neural networks can encode the logical `AND`, `OR`, and `NOT`.

`AND` becomes two input dendrites with synapses of weight 1, connected to a single neuron with threshold $\theta = 3/2$. `OR` becomes two input dendrites with synapses of weight 1, connected to a single neuron with threshold $\theta = 1/2$. `AND` becomes one input dendrite with synapse of weight -1, connected to a single neuron with threshold $\theta = -1/2$.

AND:

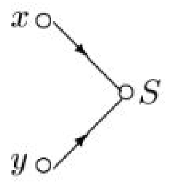
x	y	$x \wedge y$	$x+y-\frac{3}{2}$	S
0	0	0	$-3/2$	0
0	1	0	$-1/2$	0
1	0	0	$-1/2$	0
1	1	1	$1/2$	1



$w_1 = w_2 = 1$
$\theta = \frac{3}{2}$

OR:

x	y	$x \vee y$	$x+y-\frac{1}{2}$	S
0	0	0	$-1/2$	0
0	1	1	$1/2$	1
1	0	1	$1/2$	1
1	1	1	$3/2$	1



$w_1 = w_2 = 1$
$\theta = \frac{1}{2}$

NOT:

x	$\neg x$	$-x+\frac{1}{2}$	S
0	1	$1/2$	1
1	0	$-1/2$	0

$w_1 = -1$
$\theta = -\frac{1}{2}$

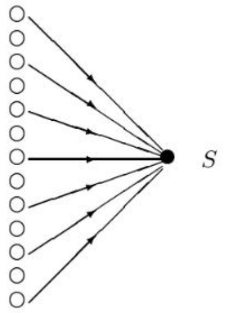
4 Architecture

There are many ways to organize neurons into neural networks with each being suited to different tasks. We will focus on one called “feed forward.” Here neurons are organized into layers. Each neuron in a layer receives input from every neuron in the layer before it, and outputs to every neuron in the layer after. Neurons are not connected to other neurons in their layer, nor to layers beyond those immediately adjacent. The neural network can then be operated by updating every neuron in a layer simultaneously, and the layers sequentially.

Successfully operating a feed forward network then becomes a matter of figuring out which weights and thresholds produce the desired result. This process is called learning because the network is physically storing information. Learning methods can be divided into two categories, supervised and unsupervised. We will focus on the former.

Supervised learning has three main steps. First, acquire a dataset of questions and answers in the form of bit arrays. Then, set the initial signals of the first layer, the input layer, to the bit array of a question. Perform the neural network operation and observe the difference between the output (the signals of the last layer) and the given answer. Then by some rule we adjust the weights to reduce future error. This process repeats until the network does well enough on all the questions.

Coolen presents a basic example of this in a network with a single neuron.



question :	$(x_1, \ldots, x_N)$
teacher's answer :	T (0 or 1)
student's answer :	S (0 or 1)
input $> \theta$:	$S = 1$
input $< \theta$:	$S = 0$

input $= w_1x_1 + \ldots + w_Nx_N$

Given the above definitions, the learning rule is as follows: for a given question $(x_1, ..., x_N)$ compare $S = S(x_1, ..., x_N)$ and $T = T(x_1, ..., x_N)$. Proceed as follows: Such a neuron with a

if $S = T$:	do nothing
if $S = 1$, $T = 0$:	Set $w_i = w_i - x_i$ and $\theta = \theta + 1$
if $S = 0$, $T = 1$:	Set $w_i = w_i + x_i$ and $\theta = \theta - 1$

learning rule is called a perceptron. Coolen asserts that this rule comes with a proof of convergence. Formally he states:

If values for the parameters $\{w_l\}$ and θ exists, such that $S = T$ for each question $(x_1, ..., x_N)$, then the perceptron learning rule will find these, or equivalent ones, in a finite number of modification steps.

His proof follows:

Assume Ω is a finite and discrete set of questions. Rather than keeping track of the threshold separately, we let $x_0 = -1$, and say $w_0 = \theta$. Now we can establish a weight vector $\boldsymbol{w} = (w_0, w_1, ..., w_N)$ and a question vector as $\boldsymbol{x} = (x_0, x_1, ..., x_N)$. Operation now simply becomes:

$$\boldsymbol{w} \cdot \boldsymbol{x} > 0 \rightarrow S = 1, \qquad \boldsymbol{w} \cdot \boldsymbol{x} \leq 0 \rightarrow S = 0$$

and the learning rule becomes

$$\boldsymbol{w} = \boldsymbol{w} + [T(\boldsymbol{x}) - S(\boldsymbol{x})]\boldsymbol{x}$$

We have assumed that some weight vector $\boldsymbol{w}'$ exists such that

$$T(\boldsymbol{x}) = 1 \implies \boldsymbol{w}' \cdot \boldsymbol{x} > 0, \qquad T(\boldsymbol{x}) = 0 \implies \boldsymbol{w}' \cdot \boldsymbol{x} \leq 0$$

Now define $X = \max\{|\boldsymbol{x}| : \boldsymbol{x} \in \Omega\} > 0$ and $\delta = \min\{|\boldsymbol{w}' \cdot \boldsymbol{x}| : \boldsymbol{x} \in \Omega\} > 0$. Then for all $\boldsymbol{x} \in \Omega, |\boldsymbol{x}| \leq X$ and $|\boldsymbol{w}' \cdot \boldsymbol{x}| \geq \delta$. Next we examine the modification step $w \to w_1$. We know $S = 1 - T$, otherwise there would be no modification. We examine two quantities:

$$\boldsymbol{w}_1 \cdot \boldsymbol{w}' = \boldsymbol{w} \cdot \boldsymbol{w}' + [2T(\boldsymbol{x}) - 1]\boldsymbol{x} \cdot \boldsymbol{w}' \qquad |\boldsymbol{w}_1|^2 = |\boldsymbol{w}|^2 + 2[2T(\boldsymbol{x}) - 1]\boldsymbol{x} \cdot \boldsymbol{w} + [2T(\boldsymbol{x}) - 1]^2 |\boldsymbol{x}|^2$$

If $\boldsymbol{w}' \cdot \boldsymbol{x} \leq 0$ then $2T(\boldsymbol{x}) - 1 = -1$ and if $\boldsymbol{w}'\boldsymbol{x} > 0$, and that therefore $[2T(\boldsymbol{x}) - 1]\boldsymbol{x} \cdot \boldsymbol{x} < 0$, so the above quantities simplify to:

$$\boldsymbol{w}_1 \cdot \boldsymbol{w}' = \boldsymbol{w} \cdot \boldsymbol{w}' + |\boldsymbol{x} \cdot \boldsymbol{w}'| \geq \boldsymbol{w} \cdot \boldsymbol{w}' + \delta \qquad |\boldsymbol{w_1}|^2 < |\boldsymbol{w}|^2 + |\boldsymbol{x}|^2 \leq |\boldsymbol{w}|^2 + X^2$$

After n iterations, we find:

$$\boldsymbol{w}_n \cdot \boldsymbol{w}' \geq \boldsymbol{w} \cdot \boldsymbol{w}' + n\delta \qquad |\boldsymbol{w_n}|^2 \leq |\boldsymbol{w}|^2 + nX^2$$

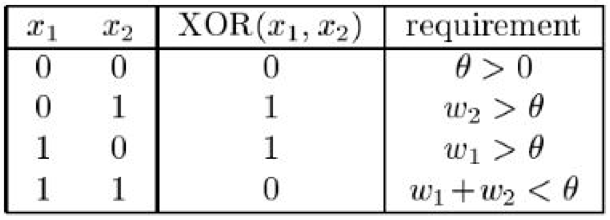
Yields:
$$\frac{\boldsymbol{w}_n \cdot \boldsymbol{w}'}{|\boldsymbol{w}'|\,|\boldsymbol{w}_n|} \geq \frac{\boldsymbol{w} \cdot \boldsymbol{w}' + n\delta}{|\boldsymbol{w}'|\sqrt{|\boldsymbol{w}|^2 + nX^2}} \implies \lim_{n\to\infty} \frac{1}{\sqrt{n}} \frac{\boldsymbol{w}_n \cdot \boldsymbol{w}'}{|\boldsymbol{w}'|\,|\boldsymbol{w}_n|} \geq \frac{\delta}{|\boldsymbol{w}'|\,X} > 0$$

Therefore the number of iterations must be bounded, other this is a contradiction of the Schwarz inequality $|\boldsymbol{w} \cdot \boldsymbol{w}'| \leq |\boldsymbol{w}|\,|\boldsymbol{w}'|$. If no more modifications can be made then $S(\boldsymbol{x}) = T(\boldsymbol{x})$ for every $\boldsymbol{x} \in \Omega$. $\square$

This theorem relied on the assumption that a solution exists. The binary operation `XOR` is a simple example that not every operation $\{0,1\}^N \to \{0,1\}$ is computable by a single perceptron. The requirement is that

$$\texttt{XOR}(x_1, x_2) = 1 \implies w_1x_1 + w_2x_2 > \theta \qquad \text{and} \qquad \texttt{XOR}(x_1, x_2) = 0 \implies w_1x_1 + w_2x_2 \leq \theta$$

The truth table: show that the requirements are impossible.



x_1	x_2	$\text{XOR}(x_1, x_2)$	requirement
0	0	0	$\theta > 0$
0	1	1	$w_2 > \theta$
1	0	1	$w_1 > \theta$
1	1	0	$w_1 + w_2 < \theta$

The solution to this problem is coupling perceptron's together, to form networks. Networks can tackle binary or real valued problems. Coolen provides a proof of computability for binary transformations. Regular (usually meaning bounded domain), real functions are modeled by using a real valued activation function, rather than the step function we previously introduced. These functions are continuous, non-decreasing, and bounded. Common choices are $\tanh(s)$ or $\operatorname{erf}(s)$, where s is the dot product of signals and weights.

Now we need to construct a learning rule. We quantify the error as:

$$E = \frac{1}{2} \sum_{\boldsymbol{x} \in \Omega} p(\boldsymbol{x}) \left[T(\boldsymbol{x}) - S(\boldsymbol{x}) \right]^2,$$

where $p(\boldsymbol{x})$ is the proportion of the sample data that is question $\boldsymbol{x}$, assuming no duplicates, this is $1/N$, Ω is the set of questions, and $T(\boldsymbol{x})$ and $S(\boldsymbol{x})$ are the teacher and student's answers respectively. We can minimize this using gradient descent. Nielsen has a better description of how this process actually works, so we'll follow his lead for the moment.

We seek to minimize the error using the gradient descent method. This means we visualize all the possible weights as a high dimensional space, then taking the gradient of the cost function will point us in the optimum direction to maximize the cost function. We therefore choose a small positive scalar η and define an update step to be

$$w_{i+1} = w_i - \eta \nabla E.$$

We can then iterate on the weights as a whole until the error is small enough.

Computing the gradient of the error function can be quite difficult. The common approach is to use the backpropogation algorithm, which Coolen does not detail. For an explanation we turn to Michael Nielsen's book on deep learning.

4.1 Nielsen's Description of Backpropogation

Notation is difficult to keep track, so we will introduce Nielsen's carefully. He denotes the weight connecting the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer with: w^l_{jk}. Each neuron's bias is denoted by b^l_j and the activation is a^l_j where l and j are still layer and neuron number respectively. Using these definitions we can write the activation of any neuron as

$$a^l_j = \sigma\left(\sum k w^l_{jk} a^{l-1}_k + b^l_j \right)$$

where σ is the chosen activation function (Nielsen, eq 23). We adapt this notation to write an expression for all of the activations by forming matrices and vectors of like terms. Each layer has an associated weight matrix w^l, where w^l_{jk} is the k^{th} element in the j^{th} row. Likewise, we can form bias and activation vectors b^l and a^l respectively. Now we can describe the activation of an entire layer at once

$$a^l = \sigma(w^l a^{l-1} + b^l).$$

In this form sigma is vectorized so it maps from $\mathbb{R}^n$ to $\mathbb{R}^n$. A final note on notation, $z^l = w^l a^{l-1} + b^l$ is called the weighted input.

Backpropagation is an efficient and clear way of computing the partial derivatives of the error function, which we can then feed into gradient descent. Recall we have written the error function as

$$E = \frac{1}{2n} \sum_{\boldsymbol{x} \in \Omega} \left[T(\boldsymbol{x}) - a^L(\boldsymbol{x}) \right]^2,$$

where n is the number of questions in some set Ω, $T(\boldsymbol{x})$ is the correct answer, L is the number of layers in the network and $a^L(\boldsymbol{x})$ is the activation of the final layer given $\boldsymbol{x}$ as input to the network.

The way backpropagation works is by computing the partial derivatives $\partial E/\partial w^l_{jk}$ and $\partial E/\partial b^l_j$ for individual training samples and then average over the whole set.

Let δ^l_j be the individual error for each neuron, defined as

$$\delta^l_j = \frac{\partial E}{\partial z^l_j}$$

Backpropogation can be decomposed into four fundamental equations:

$$\delta^L_j = \frac{\partial E}{\partial a^L_j} \sigma' \left(z^L_j \right) \qquad \text{or in matrix form} \qquad \delta^L = \nabla_{a^L} E \odot \sigma' \left(z^L \right)$$

Using the square error function defined above, we can further simplify this to:

$$\delta^L = (a^L - T(\boldsymbol{x})) \odot \sigma' \left(z^L \right).$$

Next we formulate an expression for the error δ^l in terms of the error δ^{l+1}. This is where the name backpropogation starts to make sense, because such as equation will allow us to move backwards through the network. Such an equation is

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l).$$

Now that we have the tools to compute the error for any neuron at any layer, we want to compute the partial derivatives.

$$\frac{\partial E}{\partial b^l_j} = \delta^l_j \qquad \text{and} \qquad \frac{\partial E}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

We can now proceed using the gradient descent method previously described. Backpropogation simply evaluates the gradient for us.

5 Better Training through Hessian Free Optimization

Gradient descent scales well laterally, meaning it works well for layers with many neurons, however it does not work on deep nets that have many layers. It also exhibits issues with functions that have pathological curvature. These are functions where the direction of steepest descent is not a good approximation of the direction to a local minimum. An example of such pathological curvature is the Rosenbrock function. This function can be pictured as a horseshoe shaped valley. The minimum is located at the apex of the horseshoe. The walls of the valley are steep, but the floor only slopes gradually. The gradient descent algorithm does not perform well on this function because the direction of steepest descent points across the horseshoe rather than along it. This causes each iteration of the gradient descent to bounce back and forth across the horseshoe, only slowly converging to the true minimum.

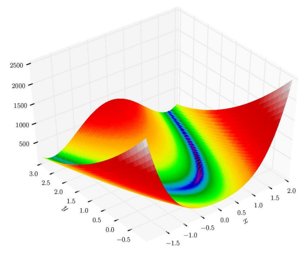


Figure 1: The Rosenbrock Function

A method of combating such issues in minimization is to employ a second-order technique. Gradient descent is a first-order technique because it only relies on the first derivatives, in the form of the gradient, of the function. Second-order techniques on the other hand, rely on second-order derivatives which provide information about the curvature.

The canonical example of a second order scheme is Newton's Method, which relies on the fact that for $\boldsymbol{\delta}$ where $|\boldsymbol{\delta}|$ is small:

$$f(\boldsymbol{x}_k + \boldsymbol{\delta}) \approx M_k(\boldsymbol{\delta}) = f(\boldsymbol{x}_k) + \nabla f(\boldsymbol{x}_k) \cdot \boldsymbol{\delta} + \frac{1}{2}\boldsymbol{\delta} \cdot H(\boldsymbol{x}_k)\boldsymbol{\delta}.$$

Where $\nabla f(\boldsymbol{x})$ is the gradient and $H(\boldsymbol{x})$ is the Hessian matrix. We then try to minimize $M(\boldsymbol{\delta})$ iteratively. For a given $\boldsymbol{\delta}_k$, we compute that $\boldsymbol{\delta}_{k+1} = \boldsymbol{\delta}_k + \alpha_k \boldsymbol{\delta}_k^*$, where $\boldsymbol{\delta}_k^*$ minimizes $M_k(\boldsymbol{\delta})$ and $\alpha_k \in [0, 1]$ is the step length. If $H(\boldsymbol{x_k})$ is positive definite, then $M_k(\delta)$ will be bounded below. This guarantees that $\boldsymbol{\delta}_k^*$ exists, and it can be shown that $\boldsymbol{\delta}_k^*$ is given by

$$\boldsymbol{\delta}_k^* = -H(\boldsymbol{x}_k)^{-1}\nabla f(\boldsymbol{x}_k) \qquad \text{or equivalently by solving} \qquad H(\boldsymbol{x}_k)\boldsymbol{\delta}_k^* = -\nabla f(\boldsymbol{x}_k)$$

The issue with this method is that even computing, must less inverting, the Hessian Matrix is prohibitively expensive for machine learning applications. Martens introduces two approximations into this method that allow for significant increases in efficiency while maintaining accuracy. The first is utilizing the conjugate gradient algorithm to minimize $M_k(\boldsymbol{\delta})$. The conjugate gradient is a powerful method in its own right, but a chief advantage in this context, is that it allows us to never form the Hessian matrix directly. This is where the Hessian Free name comes from. The second, is a numerically stable method of computing the Hessian.

5.1 Conjugate Gradient

Here we provide an brief explanation and pseudo-code for the conjugate gradient algorithm. Conjugate gradient finds a solution to the linear system $A\boldsymbol{x} = b$, by finding a minimizer to the quadratic function $\phi : \mathbb{R}^n \to \mathbb{R}$. $\phi(\boldsymbol{x})$ is defined as $\phi(\boldsymbol{x}) = 1/2\boldsymbol{x}^T A\boldsymbol{x} - b^T\boldsymbol{x}$. Below we illustrate the process.

Conjugate Gradient

```
**Inputs**: $\boldsymbol{b}, A, \boldsymbol{x}_0$
$\boldsymbol{r}_0 \leftarrow \boldsymbol{b}$
$\boldsymbol{y}_0 \leftarrow \boldsymbol{r}_0$
$i \leftarrow 0$
**while** termination condition is false **do**
    $\alpha_i \leftarrow \frac{|\boldsymbol{r}_i|^2}{\boldsymbol{y}_i^T A\boldsymbol{y}_i}$
    $\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_{\boldsymbol{i}} + \alpha_i\boldsymbol{y}_i$
    $\boldsymbol{r}_{i+1} \leftarrow \boldsymbol{r}_i + \alpha_i A\boldsymbol{y}_i$
    $\beta_{i+1} \leftarrow \frac{|\boldsymbol{r}_{i+1}|^2}{|\boldsymbol{r}_i|^2}$
    $\boldsymbol{y}_{i+1} \leftarrow \boldsymbol{r}_{i+1} + \beta_{i+1}\boldsymbol{y}_i$        $i \leftarrow i + 1$
**end while**
**output:** $\boldsymbol{x}_i$
```

Marten's version is slightly more complicated, because he includes pre-conditioning. Pre-conditioning maps the entire process to a more favorable coordinate system, which allows for faster convergence. I do not discuss it in this paper, so I have omitted it from the algorithm.

The termination condition of the iteration requires its own discussion. One might propose conditions such as $|A\boldsymbol{x} - \boldsymbol{b}|_2 < \epsilon$. While this is in line with the goal of the iteration, it does not reflect what the method is actually doing. Because the conjugate gradient is minimizing $\phi(\boldsymbol{x}) = 1/2\boldsymbol{x}^T A\boldsymbol{x} - b^T\boldsymbol{x}$, it is not guaranteed that $|A\boldsymbol{x} - \boldsymbol{b}|_2$ will steadily decrease, even though they both have the same minimizer. Instead Marten's proposes a convergence condition based on the Cauchy condition for convergence. He prescribes terminating the loop if for:

$$i > k \quad \text{and} \quad \phi(\boldsymbol{x}_i) < 0 \quad \text{and} \quad \frac{\phi(\boldsymbol{x}_i) - \phi(\boldsymbol{x}_{i-k})}{\phi(\boldsymbol{x}_i)} < k\epsilon$$

where k is how many iterations in the past we look, an varies over time. Marten's choose to set $k = \max(10, 0.1i)$[1].

Martens has several further optimizations for the conjugate gradient, but the final one that we will mention here is sharing information across iterations. This is accomplished by clever choice of the $\boldsymbol{x}_0$ parameter. Rather follow the standard practice of setting $\boldsymbol{x}_0 = (0, ..., 0)^T$, we set it to the final value for y_i from the previous time the conjugate gradient was run. Call this value y_{n-1}. Martens reports experimental gains from this and hypothesizes that while $\phi(y_{n-1})$ may not be close to 0, the directions of y_{n-1} most responsible for the increase will be minimized first by the conjugate gradient, while preserving the most progress towards the minimum.

5.2 Computing the Hessian

The trick to avoid forming the Hessian completely is to form the product $H(\boldsymbol{x})\boldsymbol{\delta}$ directly. We can do this by recognizing that the Hessian is just the Jacobian of the gradient, so $H(\boldsymbol{x})\boldsymbol{\delta}$ is the directional derivative of the gradient $\nabla f(\boldsymbol{x})$ in the $\boldsymbol{v}$ direction. This gives us that

$$H(\boldsymbol{x})\boldsymbol{\delta} = \lim_{\epsilon \to 0} \frac{\nabla f(\boldsymbol{x} + \epsilon\boldsymbol{\delta}) - \nabla f(\boldsymbol{x})}{\epsilon}.$$

This is the intuition we will employ going forward, however, computing this limit as a finite difference is not numerically stable.

To solve this problem Martens instead utilizes a technique called "forward differentiation." We denote the direction derivative operation of the gradient by

$$\mathcal{R}_{\boldsymbol{\delta}}\{\nabla f(\boldsymbol{x})\} = H(\boldsymbol{x})\boldsymbol{\delta} = \lim_{\epsilon \to 0} \frac{\nabla f(\boldsymbol{x} + \epsilon\boldsymbol{\delta}) - \nabla f(\boldsymbol{x})}{\epsilon}$$

First thing to note is that even though these are special kinds of derivatives, they still obey all the usual rules of derivatives.

$$\mathcal{R}_{\boldsymbol{v}}(x + y) = \mathcal{R}_{\boldsymbol{v}}x + \mathcal{R}_{\boldsymbol{v}}y$$

[1]Being in the complex analysis mindset, this confused me for longer than it should have.

$$\mathcal{R}_{\boldsymbol{v}}(xy) = (\mathcal{R}_{\boldsymbol{v}}x)\, y + (\mathcal{R}_{\boldsymbol{v}}y)\, x$$

$$\mathcal{R}_{\boldsymbol{v}}(\gamma(x)) = (\mathcal{R}_{\boldsymbol{v}}x)\, J_\gamma(x), \qquad \text{where } J_\gamma(x) \text{ is the Jacobian of } \gamma$$

In the same way that we had to use backpropogation to compute the gradient of the error function, we must break the computation of $H(\boldsymbol{x})\boldsymbol{\delta} = \mathcal{R}_{\boldsymbol{\delta}}\left(\nabla f(\boldsymbol{x})\right)$ into simpler parts to compute it. Here we follow Pearlmutter's explanation because it more closely follows Nielsen's description of the backpropogation method. Note, that for slightly more concise notation we let $R = \mathcal{R}_{\boldsymbol{\delta}}$

Computing $H(\boldsymbol{x})\boldsymbol{\delta}$

```
**input**: $\boldsymbol{\delta}$ unpacked as $(RW_1, ..., RW_L, Rb_1, ..., Rb_L)$
$Ra_0 \leftarrow 0$
**for** $i$ **from** 1 **to** $L$ **do:**
    $Rs_i \leftarrow RW_i a_{i-1} + W_i Ra_{i-1} + Rb_i$
    $Ra_i \leftarrow Rs_i \sigma_i'(s_i)$
**end for**
$RDa_L \leftarrow \left.\frac{\partial^2 E(y,z)}{\partial z^2}\right|_{z=a_L} (Ra_L)$
**for** $i$ **from** $L$ **downto** 1 **do**
    $RDs_i \leftarrow RDa_i \odot \sigma_i'(s_i) + Da_i \odot \sigma_i''(s_i) \odot Rs_i$
    $RDSW_i \leftarrow RDs_i a_{i-1}^T + Ds_i Ra^T i - 1$
    $RDb_i \leftarrow RDs_i$
    $RD_{a_{i-1}} \leftarrow RW_i^T Ds_i + W_i^T RDs_i$
**end for input**: $H(\boldsymbol{x})\boldsymbol{\delta}$ unpacked as $(RDW_1, ..., RDW_L, RDb_1, ..., RDb_L)$
```

Thus, as with backpropogation, we are left with the components of the Hessian vector product which we can utilize in our conjugate gradient minimization.

6 Conclusion

Neural networks are a powerful tool in data processing that also give us insights into the biology they model. Better training methods, such as Martens', greatly reduce training time. This allows for larger neural networks that can solve more complex problems.

7 References

Coolen, A.C.C. **A Beginner's Guide to the Mathematics of Neural Networks** Department of Mathematics, King's College London. 1998.

Martens, James. **Deep Learning via Hessian-free Optimization**. *Proceedings of the 27th International Conference on Machine Learning (ICML)*, 2010.

Martens, James; Sutskever, Ilya. **Training Deep and Recurrent Neural Networks with Hessian-Free Optimization**. *Neural Networks: Tricks of the Trade*, 2012.

Nielsen, Michael A. N**eural Networks and Deep Learning**. *Determination Press*, 2015.

Pearlmutter, Barak A. **Fast Exact Multiplication by the Hessian**. *Neural Computation.* 1993