Statistical Machine Learning Notes 7

Automatic Differentiation and Neural Networks

Instructor: Justin Domke

1 Introduction

The name "neural network" is sometimes used to refer to many things (e.g. Hopfield networks, self-organizing maps). In these notes, we are only interested in the most common type of neural network, the **multi-layer perceptron**.

A basic problem in machine learning is function approximation. We have some inputs $\hat{\mathbf{x}}$ and some outputs $\hat{\mathbf{y}}$, and we want to fit some function $\mathbf{f}$ such that it predicts $\hat{\mathbf{y}}$ well. (For some definition of "well".) The basic idea of neural networks is very simple. We will make up a big nonlinear function $\mathbf{f}(\mathbf{x}, \boldsymbol{\theta})$, parameterized by some vector $\boldsymbol{\theta}$. Next, we pick some loss function, measuring how well $\mathbf{f}$ predicts $\mathbf{y}$. Finally, we use some local optimization algorithm to minimize the empirical risk

$$\sum_{\{\hat{\mathbf{x}}, \hat{\mathbf{y}}\}} L\big(\mathbf{f}(\hat{\mathbf{x}}; \boldsymbol{\theta}), \hat{\mathbf{y}}\big). \tag{1.1}$$

We can also, of course, add a regularization penalty to $\boldsymbol{\theta}$.

As we will see below, multi-layer perceptrons use a quite powerful class of functions $\mathbf{f}$, and so could approximate many mappings between $\mathbf{x}$ and $\mathbf{y}$. The price we pay for this is that the empirical risk is almost always non-convex. Thus, local minima are a fact of life with neural networks. How we should react to this fact of this is an issue of debate.

Understanding the particular class of functions $\mathbf{f}$ used in neural networks is not too hard. The main technical problem is this: how do we compute the derivative of the loss function for some particular input $\hat{\mathbf{x}}$? What is

$$\frac{d}{d\boldsymbol{\theta}} L\big(\mathbf{f}(\hat{\mathbf{x}}; \boldsymbol{\theta})\big)?$$

You may wonder what the problem is. We have a function... We want its derivatives... Isn't the answer just a matter of calculus? Unfortunately, basic calculus will not get the job done. (At least, not quite.) The problem is that $\mathbf{f}$ is very large, so much so that we won't generally write it down in any closed-form. Even if we did, we would find that just applying standard

calculus rules would cause it to "explode". That is, the "closed-form" for the derivatives would be gigantic, compared to the (already huge) form of $\mathbf{f}$. The practical meaning of this is that, with out being careful, it would be much more computationally expensive to compute the gradient than to compute $\mathbf{f}$.

Luckily, if we are a little bit clever, we can compute the gradient in the *same* time complexity as computing $\mathbf{f}$. The method for doing this is called the "backpropagation" algorithm. It turns out, however, that backpropagation is really a special case of of a technique from numerical analysis known as **automatic differentiation**. It may be somewhat easier to understand the basic idea of backprop by seeing the more general algorithm first. For that reason, we will first talk about autodiff.

2 Automatic Differentiation

Let's start with an example. Consider the scalar function

$$f = \exp\big(\exp(x) + \exp(x)^2\big) + \sin\big(\exp(x) + \exp(x)^2\big) \tag{2.1}$$

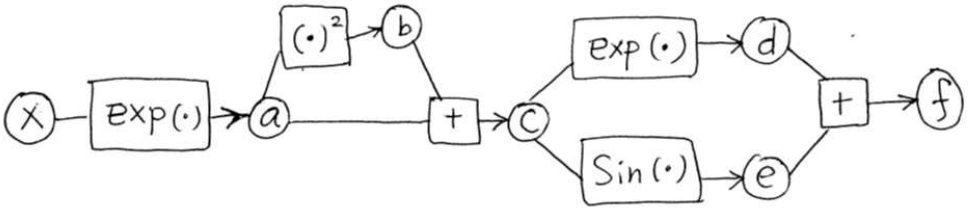
It isn't *too* hard to explicitly write down an expression for the derivative of this.

$$\frac{df}{dx} = \exp(\exp(x)+\exp(x)^2)(\exp(x)+2\exp(x)^2) + \cos(\exp(x)+\exp(x)^2)(\exp(x)+2\exp(x)^2) \tag{2.2}$$

Another way to attack this would be to just define some intermediate variables. Say

$$\begin{aligned} a &= \exp(x) \\ b &= a^2 \\ c &= a + b \\ d &= \exp(c) \\ e &= \sin(c) \\ f &= d + e. \end{aligned} \tag{2.3}$$

It is convenient to draw a little graph picturing the relationship between all the variables.



Then, we can mechanically write down the derivatives of the individual terms. Given all these, we can work backwards to compute the derivative of f with respect to each variable. This is just an application of the chain rule. We have the derivatives with respect to d and e above. Then, we can do

$$
\begin{aligned}
\frac{df}{dd} &= 1 & \frac{df}{dd} &= 1 \\
\frac{df}{de} &= 1 & \frac{df}{de} &= 1 \\
\frac{df}{dc} &= \frac{df}{dd}\frac{dd}{dc} + \frac{df}{de}\frac{de}{dc} & \frac{df}{dc} &= \frac{df}{dd}\exp(c) + \frac{df}{de}\cos(c) \\
\frac{df}{db} &= \frac{df}{dc}\frac{dc}{db} & \frac{df}{db} &= \frac{df}{dc} \\
\frac{df}{da} &= \frac{df}{dc}\frac{dc}{da} + \frac{df}{db}\frac{db}{da} & \frac{df}{da} &= \frac{df}{dc} + \frac{df}{db}2a \\
\frac{df}{dx} &= \frac{df}{da}\frac{da}{dx} & \frac{df}{dx} &= \frac{df}{da}\exp(x). \qquad (2.4)
\end{aligned}
$$

In this way, we can work backwards from the end of the graph, computing the derivative of each variable, making the use of the derivatives of the children of that variable.

Something important happened here. Notice that when we differentiated the original expression in Eq. 2.1, we obtained a significantly larger expression in Eq. 2.2. However, when we represent the function as a sequence of basic operations, as in Eq. 2.3, then we recover a very similar sequence of basic operations for computing the derivatives, as in Eq. 2.4.

Automatic differentiation is essentially just a formalization of what we did above. We can represent many functions $f : \Re^n \to \Re$ as expression graphs.

Forward Propagation

For $i = n+1,\ n+2 \ldots N$:

$x_i \leftarrow g_i(\mathbf{x}_{\mathrm{Pa}(i)})$

Here, we consider $x_1, ..., x_n$ to be the input, $x_{n+1}, ..., x_{N-1}$ to be the intermediate values, and x_N to be the final function value. The functions g_i are the elementary functions evaluated on the "parents" $\text{Pa}(i)$ of variable i.

Now, given a function represented in this way, we can just apply the chain rule step-by-step to compute derivatives. By definition, $f = x_N$, and so

$$\frac{df}{dx_N} = 1.$$

Meanwhile, for other values of x_i, we have

$$\begin{aligned} \frac{df}{dx_i} &= \sum_{j:i\in\text{Pa}(j)} \frac{df}{dx_j}\frac{dx_j}{dx_i} \\ &= \sum_{j:i\in\text{Pa}(j)} \frac{df}{dx_j}\frac{dg_j}{dx_i}. \end{aligned}$$

Thus, we can compute the derivatives by the following algorithm.

Back Propagation

(Do forward propagation)
$\frac{df}{dx_N} \leftarrow 1$
For $i = N-1, N-2, ...\, 1$:

$$x_i \leftarrow \sum_{j:i\in\text{Pa}(j)} \frac{df}{dx_j}\frac{dg_j}{dx_i}.$$

The wonderful thing about this is that it works for any differentiable function that can be phrased as an expression graph. One can really think of this as differentiating *programs*, rather than "functions". The back propagation step always has the same complexity as the original forward propagation step.

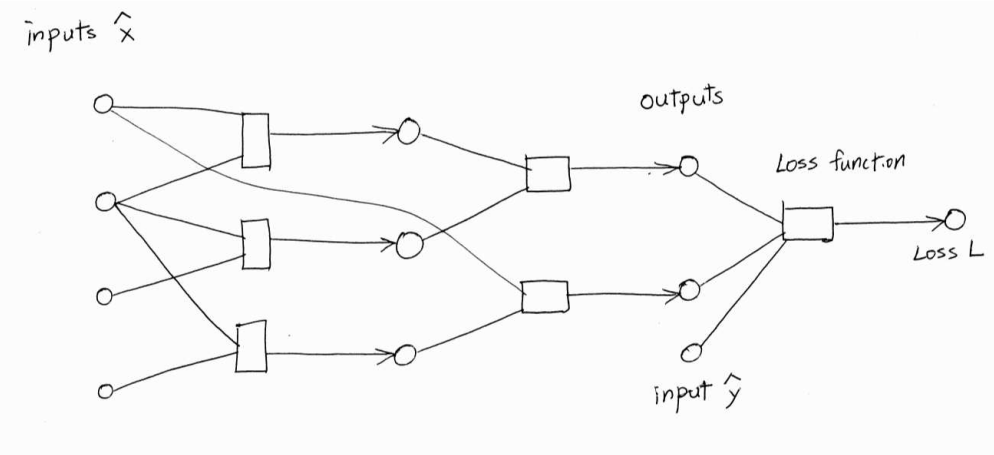
There are limitations, of course. Notably, if your program *isn't differentiable*, this can't work. There is also difficulty in handling conditionals like `if` statements.

Notice also that our entire discussion above was for computing derivatives of functions from some vector of inputs to a single output. That is, for differentiating functions $f : \Re^n \to \Re$. This is actually a special case of automatic differentiation, known as "reverse-mode". There exists another (simpler) "forward-mode" of automatic differentiation that can efficiently

compute the derivatives of a function $f : \Re \to \Re^m$ of a scalar input to a vector of outputs. There are also more general algorithms for computing the derivatives of functions from vector inputs to vector outputs $f : \Re^n \to \Re^m$. However, these algorithms are in general slower than just computing f, unlike reverse mode (forward mode) with a single output (input). In any case, the functions we are interested in in machine learning are from a vector of parameters to a scalar loss function value, so reverse-mode is all we really need.

3 Multi-Layer Perceptrons

A multi-layer perceptron is is just a big network of units connected together by simple functions.



We will denote the values computed in the network by v_i. We put the inputs $x_1, ..., x_n$ into the network by setting $v_1 = x_1$, $v_2 = x_2$, ..., $v_n = x_n$. Then, in a neural network, a new unit computes its value by (for $i > n$)

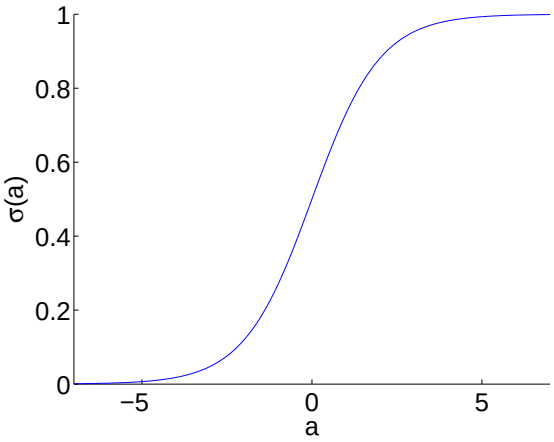
$$\boxed{v_i = \sigma_i\big(\mathbf{w}_i \cdot \mathbf{v}_{\mathrm{Pa}(i)}\big),}$$

where $\mathrm{Pa}(i)$ are the "parents" of node i. For example, if $\mathrm{Pa}(i) = (1, 3, 7)$, then

$$\mathbf{v}_{\mathrm{Pa}(i)} = (v_1, v_3, v_7).$$

Here, σ_i is a possibly nonlinear function. The most common functions are either $\sigma(a) = a$ (the identity function), or a "sigmoid" function like

$$\sigma(a) = \frac{1}{1 + \exp(-a)} \text{ or } \sigma(a) = \tanh(a).$$



(Note: there are several "sigmoid" functions in the literature that have a shape that generally looks like the above. In general, the exact choice of the function does not seem to be too important.) So, given the weights $\mathbf{w}_i$ for each variable, computing the network values are quite trivial.

Forward Propagation (Compute $\mathbf{f}(\mathbf{x})$ and $L(\mathbf{f}(\mathbf{x}), y)$)

Input $\mathbf{x}$.
Set $\mathbf{v}_{1...,n} \leftarrow \mathbf{x}$
For $i = n+1$, $n+2$, ..., N:

$$v_i \leftarrow \sigma_i(\mathbf{w}_i \cdot \mathbf{v}_{\mathrm{Pa}(i)})$$

Set $\mathbf{f} \leftarrow (v_{N-M+1}, v_{N-M+2} \ldots v_N)$
Set $L \leftarrow L(\mathbf{f}, y)$.

We consider the last M values of $\mathbf{v}$ to be the output. So, for example, if $M = 1$, $f(\mathbf{x}) = v_N$. If $M = 3$, $\mathbf{f}(\mathbf{x}) = (v_{N-2}, v_{N-1}, v)$.

To do learning with neural networks, all that we need to do is fit the weights $\mathbf{w}_i$ to minimize the empirical risk. The technically difficult part of this is calculating $\{dL/d\mathbf{w}_i\}$. If we have those values, then we could apply any standard smooth unconstrained optimization method, such as BFGS, gradient descent, or stochastic gradient descent. Before worrying about how to calculate $\{dL/d\mathbf{w}_i\}$ do that, let's consider an example.

4 MNIST

We return to the MNIST database of handwritten digits, consisting of 6,000 samples of each of the digits 1, 2, ..., 9, 0. Each sample is a 28×28 grayscale image.

Here, we make use of the multiclass logistic loss.

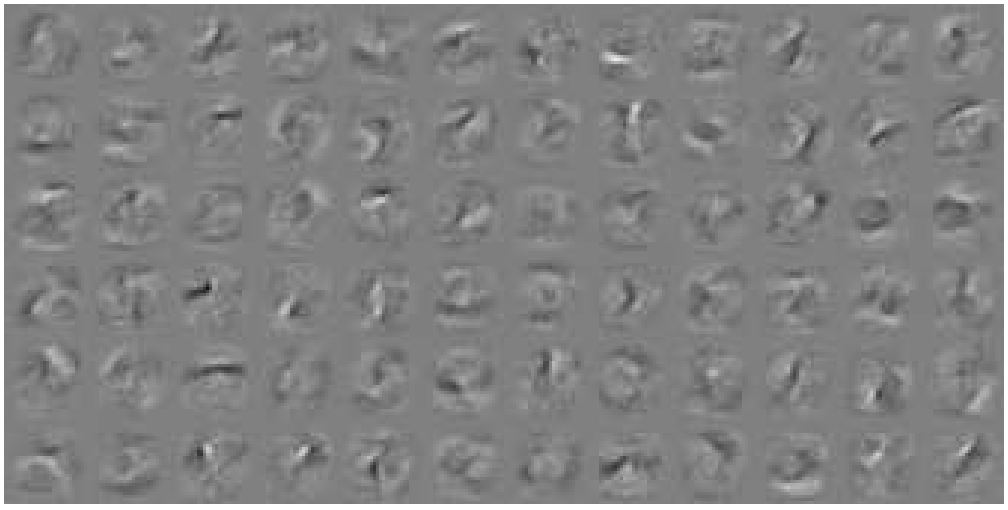
$$L(\mathbf{f}, \hat{y}) = \log \sum_y \exp f_y - f_{\hat{y}}$$

This loss can be understood intuitively as follows. given a vector $\mathbf{f}$, "log-sum-exp" can be understood as a "soft max". Very roughly speaking,[1]

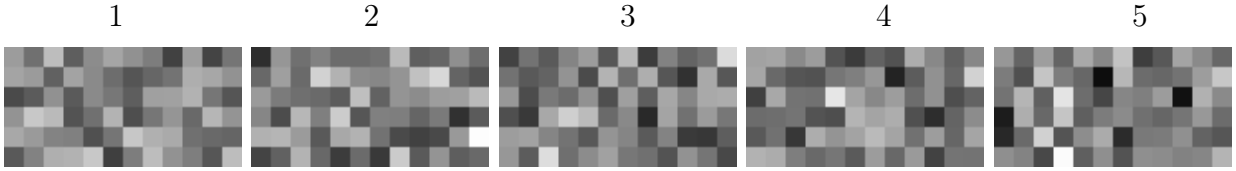
$$\log \sum_y \exp f_y \approx\approx \max_y f_y.$$

Thus, the logistic loss can be thought of as the difference between the maximum value f_y and the value $f_{\hat{y}}$. So, if $f_{\hat{y}}$ is much bigger than the other values, we have near zero loss. If there is some f_y, $y \neq \hat{y}$ much bigger than all the others, then we suffer approximately the loss $f_y - f_{\hat{y}}$.

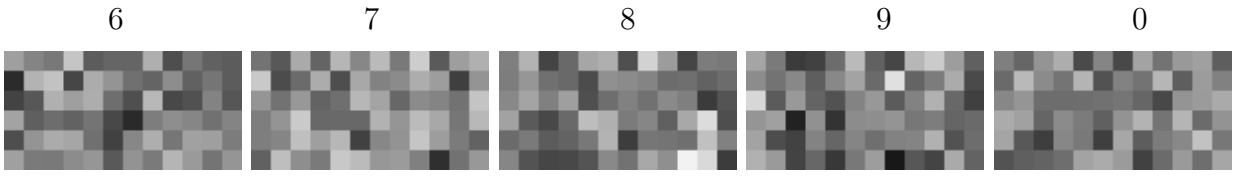
Here, a network was used with 72 "hidden" units fully connected to all inputs, with a tanh sigmoid function. The learned weights are shown below.



These 72 hidden units are then fully connected to 10 output units. These used an identiy function. The learned weights are shown below.



[1]The two '≈' symbols here are because we are being extra approximate.



We could also think about this in a vector form. If the input $\mathbf{x}$ is our 784 dimensional vector, we attain the hidden representation by doing

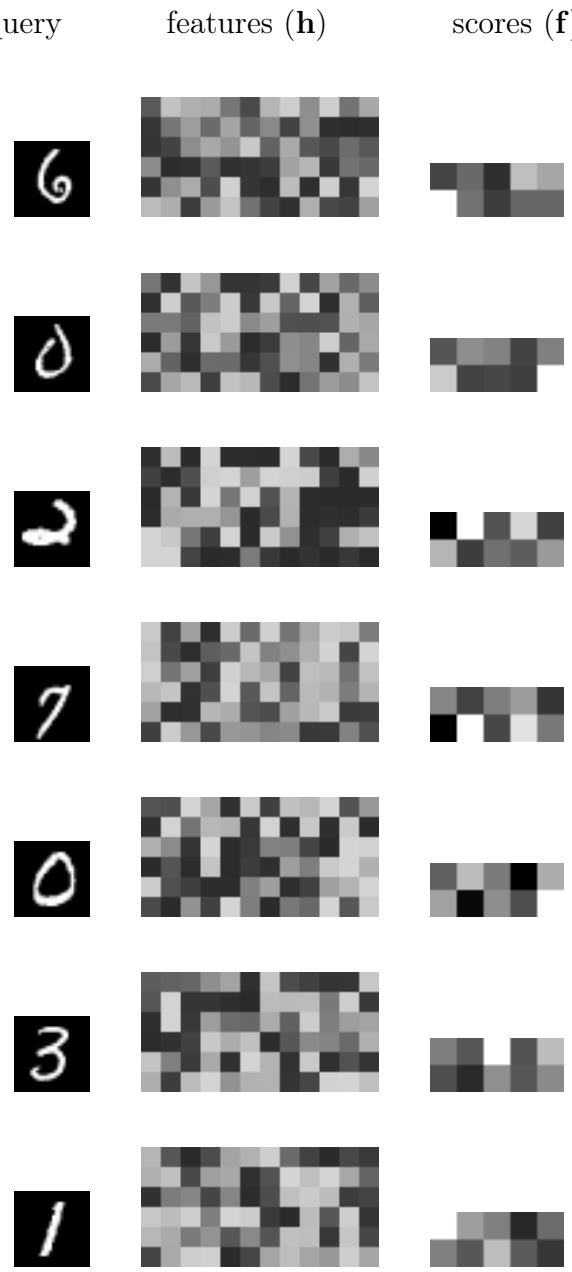
$$\mathbf{h} = \tanh(M\mathbf{x}),$$

where M is a 72×784 matrix. Then, we produce the output by

$$\mathbf{f} = W\mathbf{h},$$

where W is a 10×72 matrix. Thus, we are essentially computing a linear classifier on the basis expansion $\tanh(M\mathbf{x})$. The difference is this: we fit the basis expansion, as well as the linear classifier.

We can show the results of running the neural network on a variety of images. The features show the value computed by each of the hidden nodes, while the scores show the value computed by all the output nodes. The scores are organized as above, from 1 to 0.



5 Backpropagation

Fundamentally, backpropagation is just a special case of reverse-mode autodiff, applied to a neural network. We will derive the method here from first principles, but keep in mind that autodiff can generalize to essentially arbitrary expression graphs.

We can compute the derivatives of the loss with respect to **f** directly. Thus, we have

$$\frac{dL}{dv_i},\ i \geq N - M + 1.$$

Now, the good-old calculus 101 chain rule tells us that

$$\frac{dL}{dv_i} = \sum_{j:i\in \mathrm{Pa}(j)} \frac{dL}{dv_j}\frac{dv_j}{dv_i}.$$

Now, recall that we compute the value of v_i by

$$v_i = \sigma_i(\mathbf{w}_i \cdot \mathbf{v}_{\mathrm{Pa}(i)}).$$

From this, we can calculate that

$$\frac{dv_i}{dv_j} = \sigma_i'(\mathbf{w}_i \cdot \mathbf{v}_{\mathrm{Pa}(i)})w_{iq},\ \mathrm{Pa}(i)_q = j. \tag{5.1}$$

This is a little hard to understand in the abstract, but can be made clear by an example. Suppose $Pa(i) = (2, 7, 9)$. Then

$$v_i = \sigma_i(\mathbf{w}_i \cdot \mathbf{v}_{(2,7,9)}).$$

It is not hard to see that we have the three derivatives

$$\begin{aligned}
\frac{dv_i}{dv_2} &= \sigma_i'(\mathbf{w}_i \cdot \mathbf{v}_{(2,7,9)})w_{i1} \\
\frac{dv_i}{dv_7} &= \sigma_i'(\mathbf{w}_i \cdot \mathbf{v}_{(2,7,9)})w_{i2} \\
\frac{dv_i}{dv_9} &= \sigma_i'(\mathbf{w}_i \cdot \mathbf{v}_{(2,7,9)})w_{i3}.
\end{aligned}$$

Now, defining the notation $v_i' = \sigma_i'(\mathbf{w}_i \cdot \mathbf{v}_{\mathrm{Pa}(i)})w_{iq}$, we can rewrite Eq. 5.1 as

$$\frac{dv_i}{dv_j} = v_i' w_{iq},\ \mathrm{Pa}(i)_q = j. \tag{5.2}$$

Yet another way to write this, in vector notation, is

$$\frac{dv_i}{d\mathbf{v}_{\mathrm{Pa}(i)}} = v_i'\mathbf{w}_i.$$

Meanwhile, we can also easily derive that

$$\begin{aligned}\frac{dL}{d\mathbf{w}_i} &= \frac{dL}{dv_i}\frac{dv_i}{d\mathbf{w}_i} \\ &= \frac{dL}{dv_i}\sigma_i'(\mathbf{w}\cdot\mathbf{v}_{\mathrm{Pa}(i)})\mathbf{v}_{\mathrm{Pa}(i)} \\ &= \frac{dL}{dv_i}v_i'\mathbf{v}_{\mathrm{Pa}(i)}.\end{aligned}$$

Putting all the pieces together, we can derive the full Backpropagation algorithm.

Backpropagation (Compute $\mathbf{f}(\mathbf{x})$, $L(\mathbf{f}(\mathbf{x}), y)$, and $dL/d\mathbf{w}_i$)

Input $\mathbf{x}$.
Set $\mathbf{v}_{1...,n} \leftarrow \mathbf{x}$
For $i = n+1$, $n+2$, ..., N:

$$\begin{aligned}v_i &\leftarrow \sigma_i(\mathbf{w}_i\cdot\mathbf{v}_{\mathrm{Pa}(i)}) \\ v_i' &\leftarrow \sigma_i'(\mathbf{w}_i\cdot\mathbf{v}_{\mathrm{Pa}(i)})\end{aligned}$$

Set $\mathbf{f} \leftarrow (v_{N-M+1}, v_{N-M+2} \ldots v_N)$

Set $L \leftarrow L(\mathbf{f}, y)$

Compute $\dfrac{dL}{d\mathbf{f}}$

Initialize $\dfrac{dL}{dv_i} \leftarrow 0$

For $m = 1$, 2, ..., M:

$$\frac{dL}{dv_{N-M+m}} \leftarrow \frac{dL}{df_m}$$

For $i = N$, $N-1$, ..., $n+1$:

$$\begin{aligned}\frac{dL}{d\mathbf{w}_i} &\leftarrow \frac{dL}{dv_i}v_i'\mathbf{v}_{\mathrm{Pa}(i)} \\ \frac{dL}{d\mathbf{v}_{\mathrm{Pa}(i)}} &\leftarrow \frac{dL}{d\mathbf{v}_{\mathrm{Pa}(i)}} + \frac{dL}{dv_i}v_i'\mathbf{w}_i\end{aligned}$$

Note that if we didn't have Backpropagation, we could still calculate the derivatives dL/dw_{in}

with reasonable accuracy by finite differences[2]. This would require running the Forward propagation algorithm roughly P times if we have a total of P parameters. It would also be somewhat more numerically unstable.

6 Discussion

Neural networks are relatively fast, as classifiers go, as long as there aren't too many hidden units. They also seem to work reasonably well for many problems, and so seem to be the method of choice for a certain range of speed and accuracy requirements.

Perhaps the single biggest drawback of neural networks is the fact that their optimization is usually non-convex. Local minima are a fact of life. In practice, neural networks seem to usually find a reasonable solution when the number of layers is not too large, but find poor solutions when using more than, say, 2 hidden layers. (Convolutional neural networks, neural networks in which many weights are constrained to be the same in order to enforce translational invariance, are claimed to be somewhat immune to these problems.)

It is possible to find better minima by, e.g., searching from many initial solutions. If we were somehow able to identify the global optima, however, the results would not necessarily be better! Finding the global solution, effectively means searching over a large set of candidate functions. This decreases the bias of the method, but also increases the variance. This may or may not be beneficial.

[2]How would we do this? First, we would compute the loss function L_0 for the weights $\{\mathbf{w}_i\}$. Next, we would create weights $\{\mathbf{w}'_i\}$ where $w'_{jn} = w_{jn}$ for all j, n except for w'_{in} which we set to be equal to $w_{in} + \epsilon$ for some small constant ϵ. Then we compute the loss L_1 on the weights $\{\mathbf{w}_i\}$. Finally, we can approximate

$$\frac{dL}{dw_{in}} \approx \frac{1}{\epsilon}(L_1 - L_0).$$