NEURAL NETWORK TRAINING AND GRADIENT DESCENT

Gradient descent is a powerful mathematical tool for narrowing in on a minimum value of a function. Basically it relies on the very familiar idea that at a minimum value, a (differentiable) function f has derivative of zero. If at some point x our function has a positive derivative, that suggests looking at a point a bit smaller than (i.e., to the left of) x, for a smaller value of f; for instance, x – alpha * f ’(x) where alpha is some small positive constant (the so-called learning rate). We then compute f at this new point, and repeat the process over and over until we are satisfied that we are close enough to a minimum. Notice that this works whether the derivative is positive or negative; in the latter case, the new point x – alpha * f ’(x) will fall to the right of x which again will tend to make f smaller.

Some complications:

1. We might get caught near a local minimum, not the global one; we will mostly ignore this issue here.
2. In many applications, f is a function of many (very very many) variables, so we are dealing with the gradient of f, made up of all the partial derivatives, not a single derivative. Thus the name: gradient descent.
3. We will need to use differentiable functions – or ones which tend to change gradually with small changes in x – which requires a slight change to how we formulate the “neurons”.

OK, fine. But what is this function that we would want to minimize? What has this to do with learning?

Let’s suppose we have a neural network N, and we want to “train” it to produce certain outputs y when given certain inputs x. That is, we want to adjust the synaptic weights w until N outputs y when x is input, for each of a set of training pairs (x,y) and also hopefully gives desired outputs for other inputs not in the training set. We could think of N as “learning” a function y=g(x). But in many cases of interest, there is no precise function such as g. For instance, inputs x could be images or sentences, and outputs y could be classifications (cat, human, etc) or responses (more sentences); these will not always have unambiguous “correct” answers. But if we have prepared training pairs that we are happy with, we can measure how well N does on those; that measurement (of how close an output for input x comes to the intended y) will be the function to minimize. Here is more explanation.

We start by running N on a particular training input x, seeing what output **a** it gives, compare that to the desired y, and call some measure of the difference the error (or cost or loss). Note that x and y could be single numbers, or vectors, etc. Now, the weights w are what we are interested in adjusting; the rest of the network is considered fixed. So a = a(w); here I write w to refer to the entire collection of weights, which we can regard as a list or vector (or matrix, tensor, etc., depending on how we conceptualize N).

It often simplifies calculations to look at (a(w) – y)^2 and we call this the cost function, C(w). (In the case of many (x,y) training pairs, we may instead examine the sum of all these individual costs, usually divided by how many such pairs there are.) And then this function C is what we want to minimize: we want adjusted weights w so that the actual outputs a(w) are very close to the desired outputs y, which happens if and only if C is small. And then it's just calculus! (Some ML specialists have complained that they started out wanting to understand the nature of mind, and find that they instead are doing curve-fitting!)

Just calculus? Well, slightly subtle calculus. It turns out that in most cases, there are LOTS of weights, and not only that: the connection between a typical weight and the network output is rather complicated, so that quite complicated partial derivatives arise. Basically, it turns out to be a real mess. So bad, in fact, that for a decade or more many researchers had basically given up on neural networks as being very useful – until the power of backpropagation was realized. And so we now turn to that.

Backprop is just a set of tricks to compute partial derivatives of a neural network cost function more efficiently, thereby making gradient descent much more practical for large networks. And once that – along with much faster computers – became available, networks with many layers (i.e., "deep") could be trained in reasonable amounts of time. This has led to an explosion of research in the last ten years.

OK, let's look at some details. First, the reformulation of how a neural unit works. We'd like the output signal value to be vary gradually as a function of the input and the weights. So instead of the discontinuous Heaviside step function, we choose one of several alternates; these functions are called activation functions: they specify the resulting "activity" of the units.

A common activation function – and the one we will use – is the (logistic) sigmoid function: 1/(1 + exp(-x)). This actually looks a lot like the Heaviside function: it is essentially zero for x < -4, and essentially 1 for x > +4, and it rises smoothly between 0 and 1 for -4 < x < 4, crossing the y-axis at y=1/2.

Another common choice is the ReLU (rectified linear unit) function, which is simply zero for x<0 and x for x>=0; that is, ReLU(x) = max(x,0). While behaving quite differently from the Heaviside function (it is in effect the integral of the Heaviside function), it is nevertheless in wide use, especially for deep networks. To be sure, it is not differentiable at x=0, but this is easily dealt with. Still others are tanh(x) and log(1+e^x); the latter is called the softplus function, and is basically a smoothed version of ReLU.

These functions also require us to give a different interpretation to signals: a unit can't any longer simply fire (1) or not fire (0). Instead might think of the value as strength of firing; and one can arbitrarily pick an final output value – say 0.5 – above which (at the output layer) a yes-no classification, for instance, might be considered correct.

There also are alternative Cost functions one may use. One popular choice is the cross-entropy cost function: $Cost(w) = y \ln a + (1 - y) \ln(1 - a)$ where as before a (or h) is the output (resulting from weights w) that we want to be close to y.

Here is a summary of the overall picture, for supervised training of a feedforward (i.e., non-recursive) neural net.

1. Choose a set of training pairs (x,y).
2. Choose a network architecture (how many layers, etc)
3. Choose a learning rate (we'll get to this below).
4. Choose initial weight values w.
5. Choose an activation function.
6. Choose a cost function C.
7. Run the network on one or more training pairs.
8. Compute C for the results (compare outputs to y-values).
9. Alter the weight values to make C smaller.
10. Repeat until close enough (e.g., defined by the user).

It is step 9 above where gradient descent comes in. So let's return to that, with an example. Consider the function z=f(x,y) = 2x^2 + y^2 (which in an ML application would be the cost function, and x and y would be weights – try not to get confused about the variables here). The graph of f will be a surface. Now, we happen to know the true minimum: it's zero, and occurs at the point (0,0). But suppose we did not know this, and happened to be looking at the value of f at, say, (1,-1): f(1,-1) = 3. We can compute the partial derivatives at (1,-1), getting Grad_f(1,-1) = (4,-2). This means that if x increases a little, f increases roughly 4 times that amount; and if y increases a little, f decreases about twice the amount of increase in y. So, to decrease f, we might just want to increase y a little. But we can do better: increase y *and* decrease x together; this reduces f roughly by both twice the increase in y *and* four times the decrease in x. This is just what the gradient is telling us. So at the point (1,-1) x has a stronger capacity to reduce f than does y. So we are tempted to now look at a different point, close to (1,-1) but chosen to make f smaller. A simple-minded way would be to look at new x and y values as follows: new-x = old-x minus 4, new-y = old-y plus 2. Geometrically, this amounts to moving along the graph of f from the surface point corresponding the original (1,-1) to the new point (-3,1), where f has the value 19. Whoops! We moved too far, and f got bigger. So we introduce a smallish constant of proportionality, alpha, called the learning rate or step-size; that is we determine our new x and y values as new-x = old-x minus alpha*4, new-=y = old-y minus alpha*(-2). Notice that we used "minus" for both x and y, and the partial derivatives (4 and -2) take care of the rest. The gradient is a vector in the xy plane that points in the direction of maximum increase in f, so its negative points in the direction of maximum decrease.

To sum up: (x,y) → (x,y) – alpha*Grad_f(x,y)

Taking alpha = 0.1, for instance, we'd get the new x and y values of
x = 1 – 0.1*4 = 0.6, and y = -1 – 0.1*(-2) = -0.8. At these values, f has the value 2(0.6)^2 + (-0.8)^2 = 0.72 + 0.64 = 1.36, which is indeed smaller than our starting value of 3.

Then we could repeat this same process but now starting at these new x-y values and hope to get an even smaller value for f.

This may seem clumsy; but it can be extremely effective. It is not guaranteed to work; obviously things can go wrong; and choosing alpha can be tricky. But today there are excellent software packages that do much of this for you.

Ok, so what does a neural *unit* look like now? You may have noticed that we did not specify threshold values theta in our list of ten steps above. That is because in the modified version of a unit, we smuggle it into the network in so-called bias units. It works as follows:

Here again is the Mc-Cullogh-Pitts formulation –

> if weighted-sum of inputs > theta then signal = 1, else = 0

This can also be written as signal = H(weighted-sum – theta). And we will replace H with the sigmoid function. But we also will consider theta as just another element in the weighted sum, by introducing at every layer l one more unit whose output-signal is always fixed to be +1 and which (like all the other units at layer l) sends its signal to all units at layer l+1, with an associated weight w for each such connection (synapse). The w associated with such a "bias" unit plays the role of (the negative of) the old theta. So we don't need to think about threshholds (and how to adjust them) anymore, since this will happen as part of the adjusting of the weights.