MLISP: Machine Learning in Signal Processing

Lecture 21

Michael A. Nielsen, "Neural Networks and Deep Leanring, Chapters 3 and 5"

Scribe: Prof. V. I. Morgenshtern

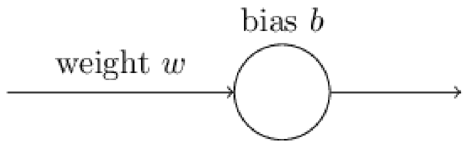
Agenda:

1. The cross-entropy cost function
2. Another model of artificial neuron: RELU
3. Why are deep neural networks hard to train?
4. The vanishing gradient problem

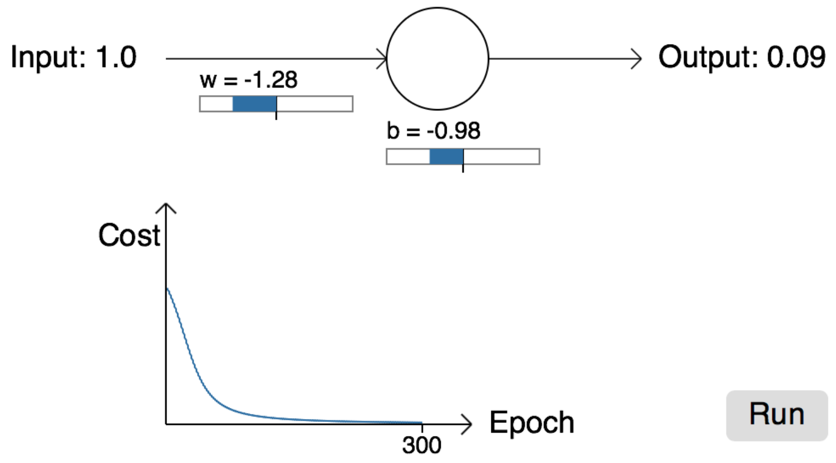
1 The cross-entropy cost function

When humans learn a new task, they usually progress very quickly in the beginning, when they perform the task in a decisively wrong way. When the learn to perform the task better they start to learn slower.

Ideally, we hope and expect that our neural networks will learn fast from their errors. Is this what happens in practice? To answer this question, let's look at a toy example. The example involves a neuron with just one input:

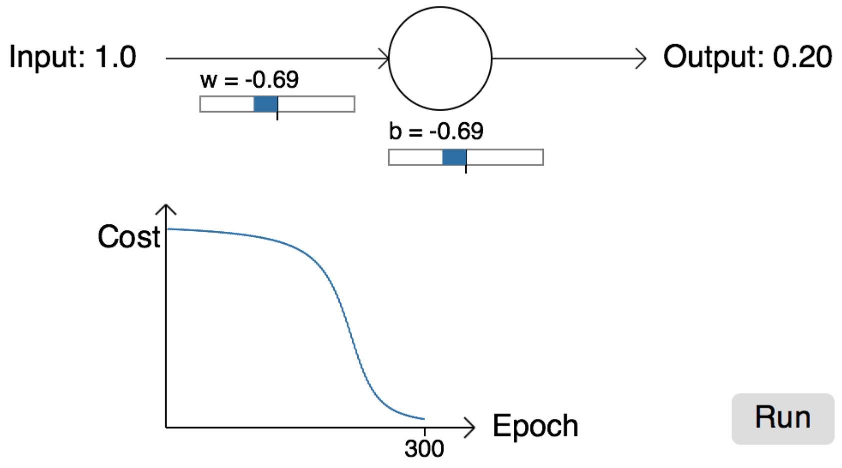


We'll train this neuron to do something ridiculously easy: take the input 1 to the output 0. To make things definite, we'll pick the initial weight to be 0.6 and the initial bias to be 0.9. The initial output from the neuron is 0.82, so quite a bit of learning will be needed before our neuron gets near the desired output, 0.0. Let's set the learning rate $\eta = 0.15$ and monitor the learning progress of the neuron:



As you can see, the neuron rapidly learns a weight and bias that drives down the cost, and gives an output from the neuron of about 0.09. That's not quite the desired output, 0.0, but it is pretty good.

Suppose, however, that we instead choose both the starting weight and the starting bias to be 2.0. In this case the initial output is 0.98, which is very badly wrong. Let's look at how the neuron learns to output 0 in this case:



Although this example uses the same learning rate $\eta = 0.15$, we can see that learning starts out much more slowly. Indeed, for the first 150 or so learning epochs, the weights and biases don't change much at all. Then the learning kicks in and, much as in our first example, the neuron's output rapidly moves closer to 0.0.

This behaviour is strange when contrasted to human learning. As I said at the beginning of this section, we often learn fastest when we're badly wrong about something. But we've just seen that our artificial neuron has a lot of difficulty learning when it's badly wrong – far more difficulty than when it's just a little wrong. What's more, it turns out that this behaviour occurs not just in this toy model, but in more general networks. Why is learning so slow? And can we find a way of avoiding this slowdown?

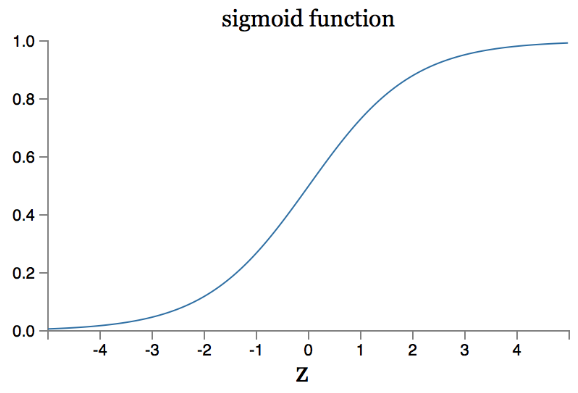
To understand the origin of the problem, consider that our neuron learns by changing the weight and bias at a rate determined by the partial derivatives of the cost function, $\partial C/\partial w$ and $\partial C/\partial b$. So saying "learning is slow" is really the same as saying that those partial derivatives are small. The challenge is to understand why they are small. To understand that, let's compute the partial derivatives. Recall that we're using the quadratic cost function, which, is given by

$$C = \frac{(y-a)^2}{2},$$

where a is the neuron's output when the training input $x = 1$ is used, and $y = 0$ is the corresponding desired output. To write this more explicitly in terms of the weight and bias, recall that $a = \sigma(z)$, where $z = wx + b$. Using the chain rule to differentiate with respect to the weight and bias we get

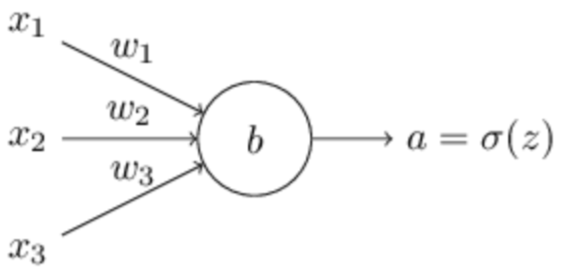
$$\begin{aligned}\frac{\partial C}{\partial w} &= (a-y)\sigma'(z)x = a\sigma'(z) \\ \frac{\partial C}{\partial b} &= (a-y)\sigma'(z) = a\sigma'(z),\end{aligned}$$

where we have substituted $x = 1$ and $y = 0$. To understand the behaviour of these expressions, let's look more closely at the $\sigma'(z)$ term on the right-hand side. Recall the shape of the σ function:



We can see from this graph that when the neuron's output is close to 1, the curve gets very flat, and so $\sigma'(z)$ gets very small. Equations above tell us that $\partial C/\partial w$ and $\partial C/\partial b$ get very small. This is the origin of the learning slowdown. What's more, as we shall see a little later, the learning slowdown occurs for essentially the same reason in more general neural networks, not just the toy example we've been playing with.

How can we address the learning slowdown? It turns out that we can solve the problem by replacing the quadratic cost with a different cost function, known as the cross-entropy. To understand the cross-entropy, let's move a little away from our super-simple toy model. We'll suppose instead that we're trying to train a neuron with several input variables, $x_1, x_2, \ldots$, corresponding weights $w_1, w_2, \ldots$, and a bias, b:



The output from the neuron is, of course, $a = \sigma(z)$, where $z = \sum_j w_j x_j + b$ is the weighted sum of the inputs. We define the cross-entropy cost function for this neuron by

$$C = -\frac{1}{n} \sum_x \left[y \ln a + (1-y) \ln(1-a) \right], \tag{1}$$

where n is the total number of items of training data, the sum is over all training inputs, x and y is the corresponding desired output.

It's not obvious that the expression above fixes the learning slowdown problem. In fact, frankly, it's not even obvious that it makes sense to call this a cost function! Before addressing the learning slowdown, let's see in what sense the cross-entropy can be interpreted as a cost function.

Two properties in particular make it reasonable to interpret the cross-entropy as a cost function. First, it's non-negative, that is, $C > 0$. To see this, notice that: (a) all the individual terms in the sum in (57) are negative, since both logarithms are of numbers in the range 0 to 1; and (b) there is a minus sign out the front of the sum.

Second, if the neuron's actual output is close to the desired output for all training inputs, x, then the cross-entropy will be close to zero. To see this, suppose for example that $y = 0$ and $a \approx 0$ for some input x. This is a case when the neuron is doing a good job on that input. We see that the first term in the expression (1) for the cost vanishes, since $y = 0$, while the second term is just $-\ln(1-a) \approx 0$. A similar analysis holds when $y = 1$ and $a \approx 1$. And so the contribution to the cost will be low provided the actual output is close to the desired output.

Summing up, the cross-entropy is positive, and tends toward zero as the neuron gets better at computing the desired output, y, for all training inputs, x. These are both properties we'd intuitively expect for a cost function. Indeed, both properties are also satisfied by the quadratic cost. So that's good news for the cross-entropy. But the cross-entropy cost function has the benefit that, unlike the quadratic cost, it avoids the problem of learning slowing down. To see this, let's compute the partial derivative of the cross-entropy cost with respect to the weights. We substitute $a = \sigma(z)$ into (1), and apply the chain rule twice, obtaining:

$$\begin{aligned} \frac{\partial C}{\partial w_j} &= -\frac{1}{n} \sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)} \right) \frac{\partial \sigma}{\partial w_j} \\ &= -\frac{1}{n} \sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)} \right) \sigma'(z) x_j. \end{aligned}$$

Putting everything over a common denominator and simplifying this becomes:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x \frac{\sigma'(z)x_j}{\sigma(z)(1-\sigma(z))}(\sigma(z)-y).$$

Using the definition of the sigmoid function, $\sigma(z) = 1/(1+e^{-z})$ and a little algebra we can show that $\sigma'(z) = \sigma(z)(1-\sigma(z))$. We see that the $\sigma'(x)$ and $\sigma(z)(1-\sigma(z))$ terms cancel in the equation just above, and it simplifies to become:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x x_j(\sigma(z)-y).$$

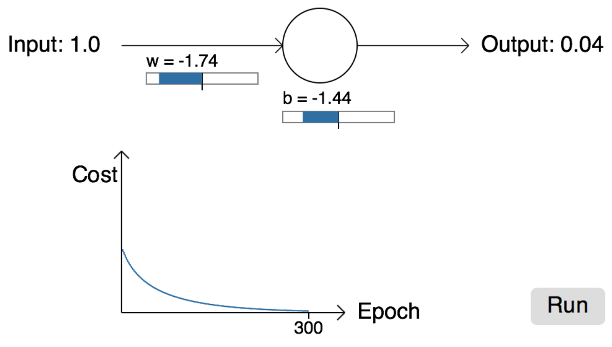
This is a beautiful expression. It tells us that the rate at which the weight learns is controlled by $\sigma(z)-y$, i.e., by the error in the output. The larger the error, the faster the neuron will learn. This is just what we'd intuitively expect. In particular, it avoids the learning slowdown. caused by the $\sigma'(z)$ term in the analogous equation for the quadratic cost. When we use the cross-entropy, the $\sigma'(z)$ term gets canceled out, and we no longer need worry about it being small. This cancellation is the special miracle ensured by the cross-entropy cost function.

In a similar way, we can compute the partial derivative for the bias. I won't go through all the details again, but you can easily verify that

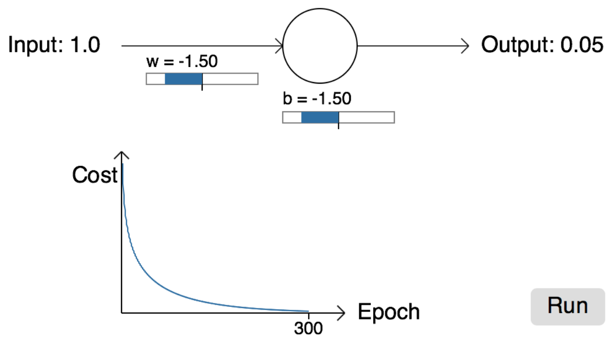
$$\frac{\partial C}{\partial b} = \frac{1}{n}\sum_x(\sigma(z)-y). \tag{2}$$

Again, this avoids the learning slowdown caused by the $\sigma'(z)$ term in the analogous equation for the quadratic cost.

Let's return to the toy example we played with earlier, and explore what happens when we use the cross-entropy instead of the quadratic cost. To re-orient ourselves, we'll begin with the case where the quadratic cost did just fine, with starting weight 0.6 and starting bias 0.9:



Unsurprisingly, the neuron learns perfectly well in this instance, just as it did earlier. And now let's look at the case where our neuron got stuck before, with the weight and bias both starting at 2.0:



Success! This time the neuron learned quickly, just as we hoped. If you observe closely you can see that the slope of the cost curve was much steeper initially than the initial flat region on the corresponding curve for the quadratic cost. It's that steepness which the cross-entropy buys us, preventing us from getting stuck just when we'd expect our neuron to learn fastest, i.e., when the neuron starts out badly wrong.

We've been studying the cross-entropy for a single neuron. However, it's easy to generalize the cross-entropy to many-neuron multi-layer networks. In particular, suppose $y = y_1, y_2, \ldots$ are the desired values at the output neurons, i.e., the neurons in the final layer, while $a_1^L, a_2^L, \ldots$ are the actual output values. Then we define the cross-entropy by

$$C = -\frac{1}{n}\sum_x \sum_j \left[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\right].$$

Problem set 5, problem 3-2: Cross entropy as a cost function In the single-neuron discussion at the start of this section, I argued that the cross-entropy is small if $\sigma(z) \approx y$ for all training inputs. The argument relied on y being equal to either 0 or 1. This is usually true in classification problems, but for other problems (e.g., regression problems) y can sometimes take values intermediate between 0 and 1. Show that the cross-entropy is still minimized when $\sigma(z) = y$ for all training inputs. When this is the case the cross-entropy has the value:

$$C = -\frac{1}{n}\sum_x [y \ln y + (1-y)\ln(1-y)].$$

The quantity $-[y \ln y + (1-y)\ln(1-y)]$ is sometimes known as the binary entropy.

Problem set 5, problem 3-3: Cost function and learning slowdown In the notation introduced in the last chapter, show that for the quadratic cost the partial derivative with respect to weights in the output layer is

$$\frac{\partial C}{\partial w_{jk}^L} = \frac{1}{n}\sum_x a_k^{L-1}(a_j^L - y_j)\sigma'(z_j^L).$$

The term $\sigma'(z_j^L)$ causes a learning slowdown whenever an output neuron saturates on the wrong value. Show that for the cross-entropy cost the output error δ^L for a single training example x is given by

$$\delta^L = a^L - y.$$

Use this expression to show that the partial derivative with respect to the weights in the output layer is given by

$$\frac{\partial C}{\partial w_{jk}^L} = \frac{1}{n}\sum_x a_k^{L-1}(a_j^L - y_j).$$

The $\sigma'(z_j^L)$ term has vanished, and so the cross-entropy avoids the problem of learning slowdown, not just when used with a single neuron, as we saw earlier, but also in many-layer multi-neuron networks. A simple variation on this analysis holds also for the biases. If this is not obvious to you, then you should work through that analysis as well.

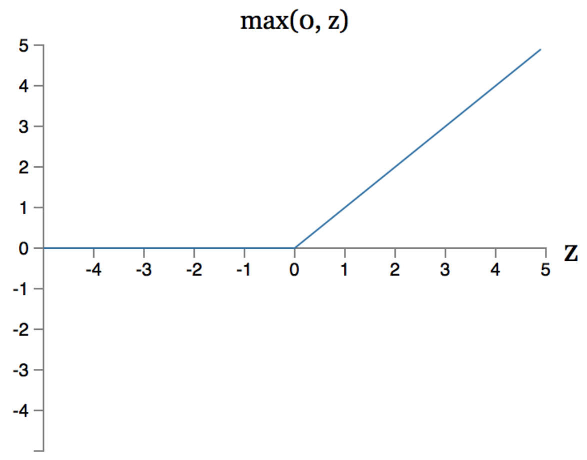
2 Another model of artificial neuron: RELU

Up to now we've built our neural networks using sigmoid neurons. In principle, a network built from sigmoid neurons can compute any function. In practice, however, networks built using other model neurons sometimes outperform sigmoid networks. Depending on the application, networks based on such alternate models may learn faster, generalize better to test data, or perhaps do both.

In modern deep neural networks, the most popular choice for the nonlinearity is the rectified linear neuron or rectified linear unit. The output of a rectified linear unit with input x, weight vector w, and bias b is given by

$$\max(0, w \cdot x + b).$$

Graphically, the rectifying function $\max(0, z)$ looks like this:

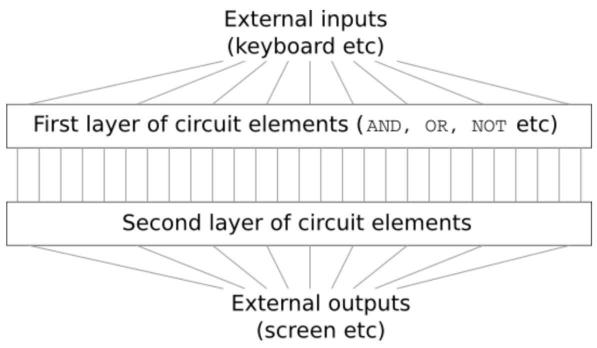


Obviously such neurons are quite different from the sigmoid neurons. However, like the sigmoid neurons, rectified linear units can be used to compute any function, and they can be trained using ideas such as backpropagation and stochastic gradient descent.

When should you use rectified linear units instead of sigmoid neurons? Some recent work on image recognition. has found considerable benefit in using rectified linear units through much of the network. However, as with tanh neurons, we do not yet have a really deep understanding of when, exactly, rectified linear units are preferable, nor why. To give you the flavor of some of the issues, recall that sigmoid neurons stop learning when they saturate, i.e., when their output is near either 0 or 1. As we've seen repeatedly in this chapter, the problem is that σ' terms reduce the gradient, and that slows down learning. By contrast, increasing the weighted input to a rectified linear unit will never cause it to saturate, and so there is no corresponding learning slowdown. On the other hand, when the weighted input to a rectified linear unit is negative, the gradient vanishes, and so the neuron stops learning entirely. These are just two of the many issues that make it non-trivial to understand when and why rectified linear units perform better than sigmoid neurons.

3 Why are deep neural networks hard to train?

Imagine you're an engineer who has been asked to design a computer from scratch. One day you're working away in your office, designing logical circuits, setting out AND gates, OR gates, and so on, when your boss walks in with bad news. The customer has just added a surprising design requirement: the circuit for the entire computer must be just two layers deep:


External inputs
(keyboard etc)
First layer of circuit elements (AND, OR, NOT etc)
Second layer of circuit elements
External outputs
(screen etc)

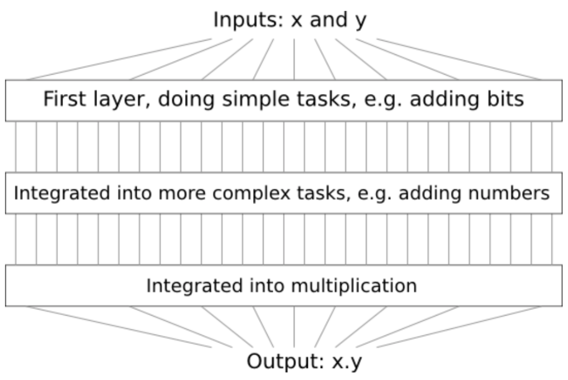
You're dumbfounded, and tell your boss: "The customer is crazy!"

Your boss replies: "I think they're crazy, too. But what the customer wants, they get."

In fact, there's a limited sense in which the customer isn't crazy. Suppose you're allowed to use a special logical gate which lets you AND together as many inputs as you want. And you're also allowed a many-input NAND gate, that is, a gate which can AND multiple inputs and then negate the output. With these special gates it turns out to be possible to compute any function at all using a circuit that's just two layers deep.

But just because something is possible doesn't make it a good idea. In practice, when solving circuit design problems (or most any kind of algorithmic problem), we usually start by figuring out how to solve sub-problems, and then gradually integrate the solutions. In other words, we build up to a solution through multiple layers of abstraction.

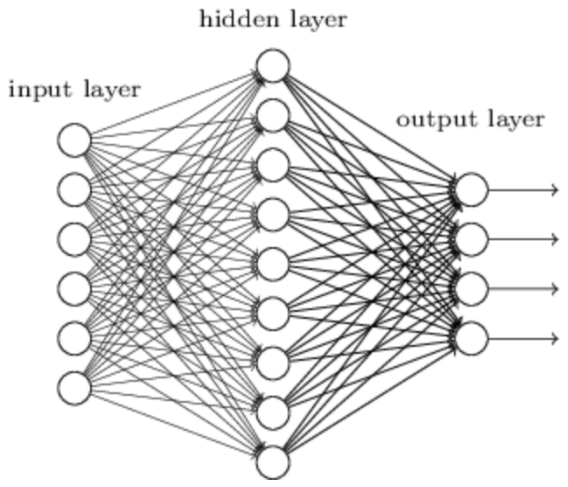
For instance, suppose we're designing a logical circuit to multiply two numbers. Chances are we want to build it up out of sub-circuits doing operations like adding two numbers. The sub-circuits for adding two numbers will, in turn, be built up out of sub-sub-circuits for adding two bits. Very roughly speaking our circuit will look like:


Inputs: x and y
First layer, doing simple tasks, e.g. adding bits
Integrated into more complex tasks, e.g. adding numbers
Integrated into multiplication
Output: x.y

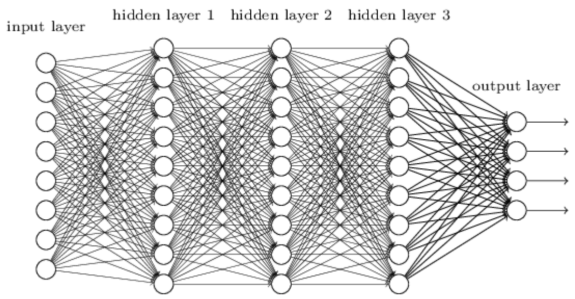
That is, our final circuit contains at least three layers of circuit elements. In fact, it'll probably contain more than three layers, as we break the sub-tasks down into smaller units than I've described. But you get the general idea.

So deep circuits make the process of design easier. But they're not just helpful for design. There are, in fact, mathematical proofs showing that for some functions very shallow circuits require exponentially more circuit elements to compute than do deep circuits. For instance, a famous series of papers in the early 1980s showed that computing the parity of a set of bits requires exponentially many gates, if done with a shallow circuit. On the other hand, if you use deeper circuits it's easy to compute the parity using a small circuit: you just compute the parity of pairs of bits, then use those results to compute the parity of pairs of pairs of bits, and so on, building up quickly to the overall parity. Deep circuits thus can be intrinsically much more powerful than shallow circuits.

Up to now, we have approached neural networks like the crazy customer. Almost all the networks we've worked with have just a single hidden layer of neurons (plus the input and output layers):



These simple networks have been remarkably useful: in earlier chapters we used networks like this to classify handwritten digits with better than 98 percent accuracy! Nonetheless, intuitively we'd expect networks with many more hidden layers to be more powerful:



Such networks could use the intermediate layers to build up multiple layers of abstraction, just as we do in Boolean circuits. For instance, if we're doing visual pattern recognition, then the neurons in the first layer might learn to recognize edges, the neurons in the second layer could learn to recognize more complex shapes, say triangle or rectangles, built up from edges. The third layer would then recognize still more complex shapes. And so on. These multiple layers of abstraction seem likely to give deep networks a compelling advantage in learning to solve complex pattern recognition problems.

How can we train such deep networks? We'll try training deep networks using our workhorse learning algorithm – stochastic gradient descent by backpropagation. But we'll run into trouble, with our deep networks not performing much (if at all) better than shallow networks.

When we look closely, we'll discover that the different layers in our deep network are learning at vastly different speeds. In particular, when later layers in the network are learning well, early layers often get stuck during training, learning almost nothing at all. This stuckness isn't simply due to bad luck. Rather, we'll discover there are fundamental reasons the learning slowdown occurs, connected to our use of gradient-based learning techniques.

As we delve into the problem more deeply, we'll learn that the opposite phenomenon can also occur: the early layers may be learning well, but later layers can become stuck. In fact, we'll find that there's an intrinsic instability associated to learning by gradient descent in deep, many-layer neural networks. This instability tends to result in either the early or the later layers getting stuck during training.

3.1 The vanishing gradient problem

So, what goes wrong when we try to train a deep network?

To answer that question, let's first revisit the case of a network with just a single hidden layer.

1 hidden layer This network has 784 neurons in the input layer, corresponding to the $28 \times 28 = 784$ pixels in the input image. We use 30 hidden neurons, as well as 10 output neurons, corresponding to the 10 possible classifications for the MNIST digits ('0', '1', '2', ... ,'9').

Let's try training our network for 30 complete epochs, using mini-batches of 10 training examples at a time, a learning rate $\eta = 0.1$, and regularization parameter $\lambda = 5.0$. As we train we'll monitor the classification accuracy on the validation data. We get a classification accuracy of 96.48 percent (or thereabouts - it'll vary a bit from run to run), comparable to our earlier results with a similar configuration.

2 hidden layers Now, let's add another hidden layer, also with 30 neurons in it, and try training with the same hyper-parameters. This gives an improved classification accuracy, 96.90 percent. That's encouraging: a little more depth is helping.

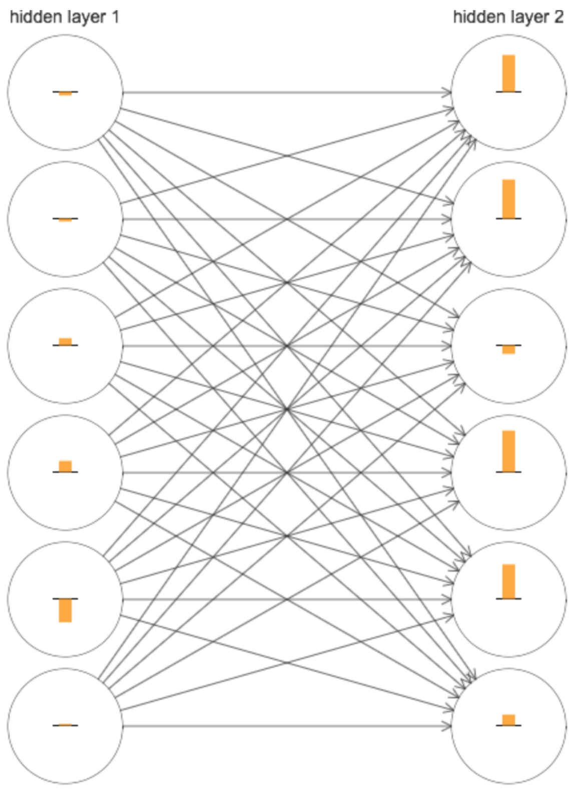
3 hidden layers Let's add another 30-neuron hidden layer. That doesn't help at all. In fact, the result drops back down to 96.57 percent, close to our original shallow network.

4 hidden layers And suppose we insert one further hidden layer. The classification accuracy drops again, to 96.53 percent. That's probably not a statistically significant drop, but it's not encouraging, either.

This behaviour seems strange. Intuitively, extra hidden layers ought to make the network able to learn more complex classification functions, and thus do a better job classifying. Certainly, things shouldn't get worse, since the extra layers can, in the worst case, simply do nothing. But that's not what's going on.

To get some insight into what's going wrong, let's visualize how the network learns. Below, we have plotted part of a [784,30,30,10] network, i.e., a network with two hidden layers, each containing 30 hidden neurons. Each neuron in the diagram has a little bar on it, representing how quickly that neuron is changing as the network learns. A big bar means the neuron's weights and bias are changing rapidly, while a small bar means the weights and bias are changing slowly. More precisely, the bars denote the gradient $\partial C/\partial b$ for each neuron, i.e., the rate of change of the cost with respect to the neuron's bias. We saw previously that this gradient quantity controlled not just how rapidly the bias changes during learning, but also how rapidly the weights input to the neuron change, too.

To keep the diagram simple, I've shown just the top six neurons in the two hidden layers. I've omitted the input neurons, since they've got no weights or biases to learn. I've also omitted the output neurons, since we're doing layer-wise comparisons, and it makes most sense to compare layers with the same number of neurons. The results are plotted at the very beginning of training, i.e., immediately after the network is initialized. Here they are:



The network was initialized randomly, and so it's not surprising that there's a lot of variation in how rapidly the neurons learn. Still, one thing that jumps out is that the bars in the second hidden layer are mostly much larger than the bars in the first hidden layer. As a result, the neurons in the second hidden layer will learn quite a bit faster than the neurons in the first hidden layer. Is this merely a coincidence, or are the neurons in the second hidden layer likely to learn faster than neurons in the first hidden layer in general?

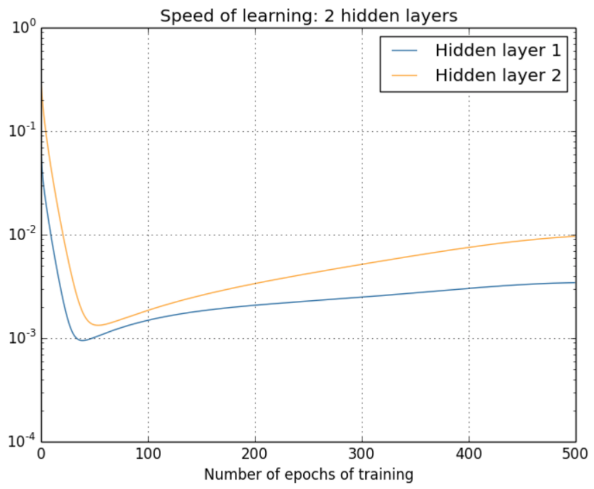
To determine whether this is the case, it helps to have a global way of comparing the speed of learning in the first and second hidden layers. To do this, let's denote the gradient as $\delta^l_j = \partial C/\partial b^l_j$ i.e., the gradient for the jth neuron in the lth layer. We can think of the gradient δ^1 as a vector whose entries determine how quickly the first hidden layer learns, and δ^2 as a vector whose entries determine how quickly the second hidden layer learns. We'll then use the lengths of these vectors as (rough!) global measures of the speed at which the layers are learning. So, for instance, the length $\|\delta^1\|$ measures the speed at which the first hidden layer is learning, while the length $\|\delta^2\|$ measures the speed at which the second hidden layer is learning.

With these definitions, and in the same configuration as was plotted above, we find $\|\delta^1\| = 0.07$ and $\|\delta^1\| = 0.31$. So this confirms our earlier suspicion: the neurons in the second hidden layer really are learning much faster than the neurons in the first hidden layer.

What happens if we add more hidden layers? If we have three hidden layers, in a [784,30,30,30,10] network, then the respective speeds of learning turn out to be 0.012, 0.060, and 0.283. Again, earlier hidden layers are learning much slower than later hidden layers. Suppose we add yet another layer with 30 hidden neurons. In that case, the respective speeds of learning are 0.003, 0.017, 0.070, and

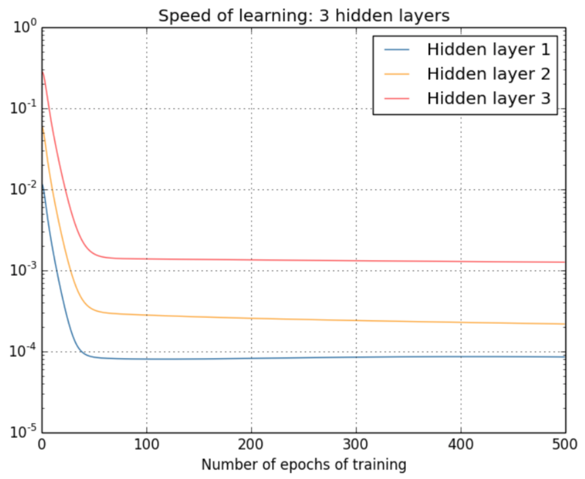
0.285. The pattern holds: early layers learn slower than later layers.

We've been looking at the speed of learning at the start of training, that is, just after the networks are initialized. How does the speed of learning change as we train our networks? Let's return to look at the network with just two hidden layers. The speed of learning changes as follows:

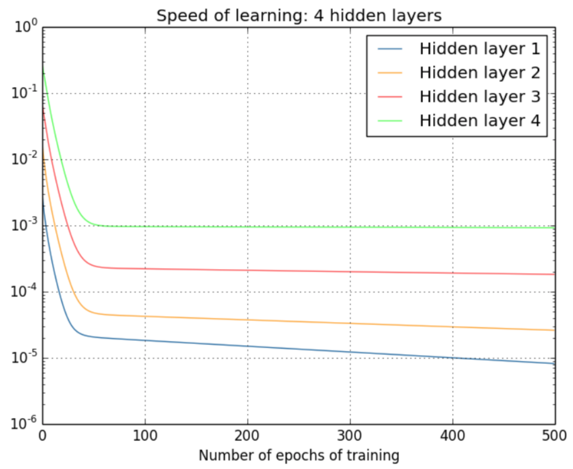


As you can see the two layers start out learning at very different speeds (as we already know). The speed in both layers then drops very quickly, before rebounding. But through it all, the first hidden layer learns much more slowly than the second hidden layer.

What about more complex networks? Here's the results of a similar experiment, but this time with three hidden layers (a [784,30,30,30,10] network):



Again, early hidden layers learn much more slowly than later hidden layers. Finally, let's add a fourth hidden layer (a [784,30,30,30,30,10] network), and see what happens when we train:

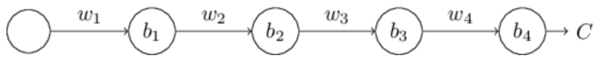


Again, early hidden layers learn much more slowly than later hidden layers. In this case, the first hidden layer is learning roughly 100 times slower than the final hidden layer. No wonder we were having trouble training these networks earlier!

We have here an important observation: in at least some deep neural networks, the gradient tends to get smaller as we move backward through the hidden layers. This means that neurons in the earlier layers learn much more slowly than neurons in later layers. And while we've seen this in just a single network, there are fundamental reasons why this happens in many neural networks. The phenomenon is known as the *vanishing gradient problem*.

Why does the vanishing gradient problem occur? Are there ways we can avoid it? And how should we deal with it in training deep neural networks? In fact, we'll learn shortly that it's not inevitable, although the alternative is not very attractive, either: sometimes the gradient gets much larger in earlier layers! This is the *exploding gradient problem*, and it's not much better news than the vanishing gradient problem. More generally, it turns out that the gradient in deep neural networks is unstable, tending to either explode or vanish in earlier layers. This instability is a fundamental problem for gradient-based learning in deep neural networks. It's something we need to understand, and, if possible, take steps to address.

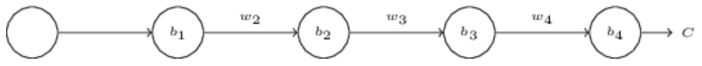
To get insight into why the vanishing gradient problem occurs, let's consider the simplest deep neural network: one with just a single neuron in each layer. Here's a network with three hidden layers:



Here, $w_1, w_2, \ldots$ are the weights, $b_1, b_2, \ldots$ are the biases, and C is some cost function. Just to remind you how this works, the output a_j from the jth neuron is $\sigma(z_j)$, where σ is the usual sigmoid activation function, and $z_j = w_j a_{j-1} + b_j$ is the weighted input to the neuron. I've drawn the cost C at the end to emphasize that the cost is a function of the network's output, a_4: if the actual output from the network is close to the desired output, then the cost will be low, while if it's far away, the cost will be high.

We are going to study the gradient $\partial C/\partial b_1$ It looks forbidding, but it's actually got a simple structure, which I'll describe in a moment. Here's the expression (ignore the network, for now, and note that σ' is just the derivative of the σ function):

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$



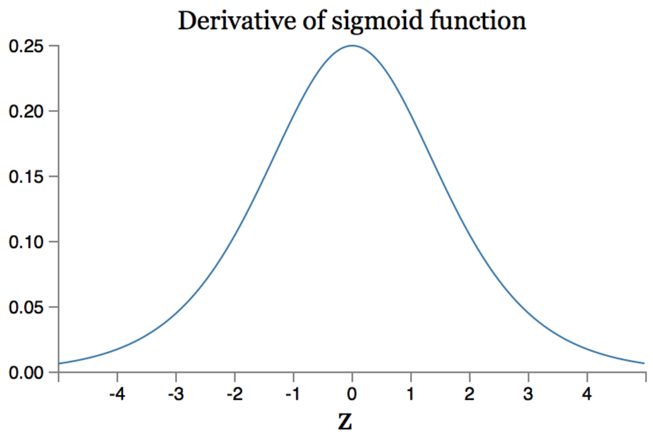
This expression is a direct consequence of the fundamental equations of backpropagation. You are welcome to rederive it for this specific network as an exercise.

The structure in the expression is as follows: there is a $\sigma'(z_j)$ term in the product for each neuron in the network; a weight w_j term for each weight in the network; and a final $\partial C/\partial a_4$ term, corresponding to the cost function at the end. Notice that we have placed each term in the expression above the corresponding part of the network. So the network itself is a mnemonic for the expression.

Excepting the very last term, the expression

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1)\, w_2\sigma'(z_2)\, w_3\sigma'(z_3)\, w_4\sigma'(z_4)\, \frac{\partial C}{\partial a_4}.$$

is a product of terms of the form $w_j\sigma'(z_j)$. To understand how each of those terms behave, let's look at a plot of the function $\sigma'(\cdot)$:



The derivative reaches a maximum at $\sigma'(0) = 1/4$. Now, if we use our standard approach to initializing the weights in the network, then we'll choose the weights using a Gaussian with mean 0 and standard deviation 1. So the weights will usually satisfy $|w_j| < 1$. Putting these observations together, we see that the terms $w_j\sigma'(z_j)$ will usually satisfy $|w_j\sigma'(z_j)| < 1/4$. And when we take a product of many such terms, the product will tend to exponentially decrease: the more terms, the smaller the product will be. This is starting to smell like a possible explanation for the vanishing gradient problem.

To make this all a bit more explicit, let's compare the expression for $\partial C/\partial b_1$ to an expression for the gradient with respect to a later bias, say $\partial C/\partial b_3$. Of course, we haven't explicitly worked out an expression for $\partial C/\partial b_3$, but it follows the same pattern described above for $\partial C/\partial b_1$. Here's the comparison of the two expressions:

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1)\,\overbrace{w_2\sigma'(z_2)}^{<\frac{1}{4}}\,\overbrace{w_3\sigma'(z_3)}^{<\frac{1}{4}}\,\underbrace{w_4\sigma'(z_4)\frac{\partial C}{\partial a_4}}_{\updownarrow\ \text{common terms}}$$

$$\frac{\partial C}{\partial b_3} = \sigma'(z_3)\,\overbrace{w_4\sigma'(z_4)\frac{\partial C}{\partial a_4}}$$

The two expressions share many terms. But the gradient $\partial C/\partial b_1$ includes two extra terms each of the form $w_j\sigma'(z_j)$. As we've seen, such terms are typically less than 1/4 in magnitude. And so the gradient $\partial C/\partial b_1$ will usually be a factor of 16 (or more) smaller than $\partial C/\partial b_3$. This is the essential origin of the vanishing gradient problem.

Problem set 5, problem 3-6: Vanishing gradients In our discussion of the vanishing gradient problem, we made use of the fact that $|\sigma'(z)| < 1/4$. Suppose we used a different activation function, one whose derivative could be much larger. Would that help us avoid the unstable gradient problem?