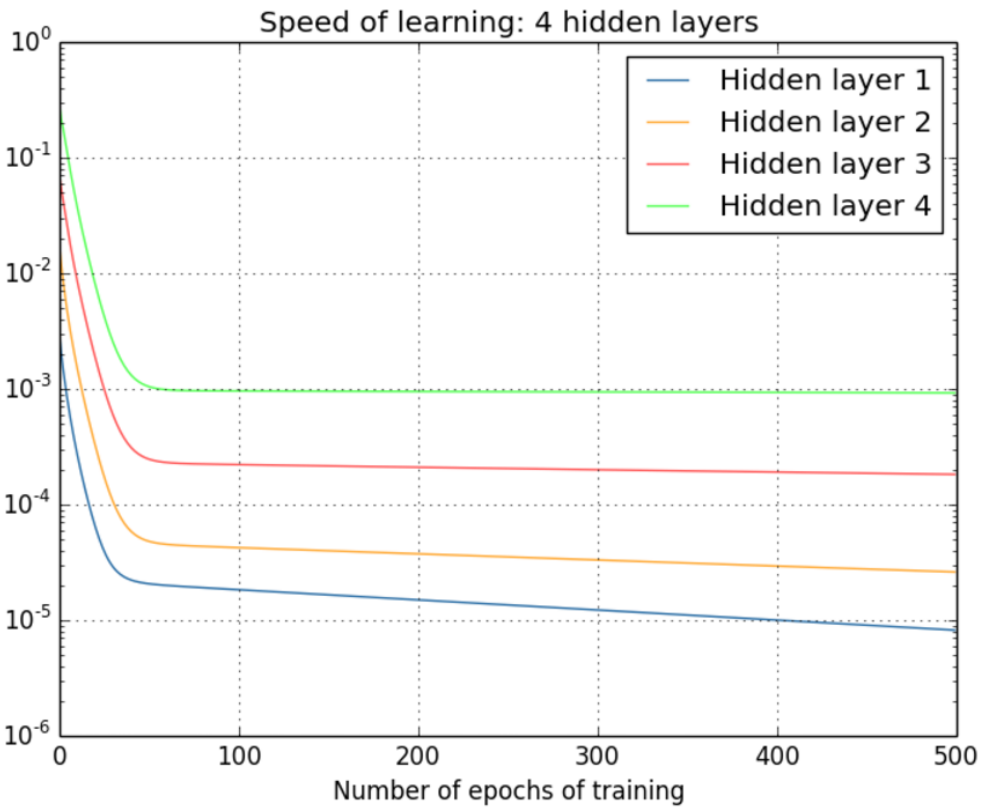
chapter 5: why are deep nets hard to train

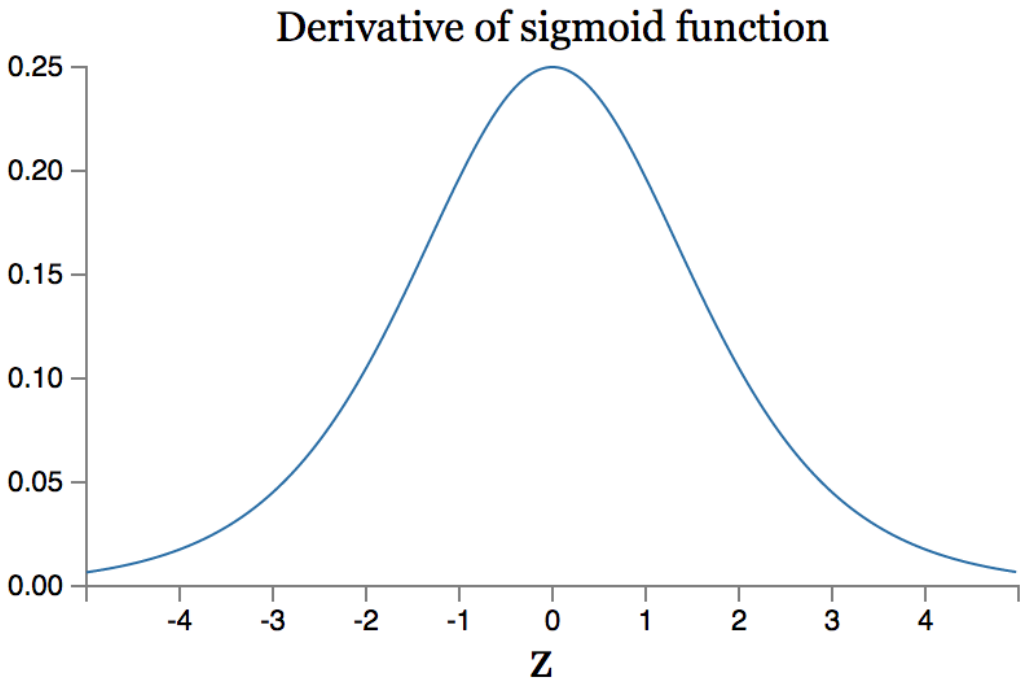
- Michael starts off drawing a comparison to logic circuits and building processing hardware.
- With some special gates it turns out to be possible to compute any function at all using a circuit that's just two layers deep.
- But just because something is possible doesn't make it a good idea. In practice, 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.
- 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. Deep circuits thus can be intrinsically much more powerful than shallow circuits.
- 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.
- there are theoretical results suggesting that deep networks are intrinsically more powerful than shallow networks, which Michael doesn’t go into.
- 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. by delving into these difficulties, we can begin to gain insight into what's required to train deep networks effectively.
- 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 yet, we are not improving.
- so what’s going wrong? Let's assume that the extra hidden layers really could help in principle, and the problem is that our learning algorithm isn't finding the right weights and biases.
- Are the neurons in the initial layers learning more slowly? lets take a look. below is visualized the training on a $[784, 30, 30, 30, 30, 10]$ network → we’ve only shown the hidden layer learning rates here, not input and output layer rates:



- Note that the above is actually a plot of the magnitudes of 4 vectors over epochs, each of whose elements is $\frac{\partial C}{\partial b_j^l}$(rate of bias learning). Note that the scale is logarithmic! So the earlier layers are learning *a lot slower.*
- 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.*
- 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.
- one response to this empirical evidence of the vanishing gradient problem is to say that maybe for the earlier layers we are already near an extrema so why is this really a problem: note that this is highly unlikely since the vanishing gradient problem repeatedly appears despite the random initialization for weights. Also, The random initialization means the first layer throws away most information about the input image. Even if later layers have been extensively trained, they will still find it extremely difficult to identify the input image, simply because they don't have enough information. And so it can't possibly be the case that not much learning needs to be done in the first layer.
- more concrete evidence for the vanishing gradient problem:
  - lets take the simplistic case of a network with just one neuron in each layer.

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

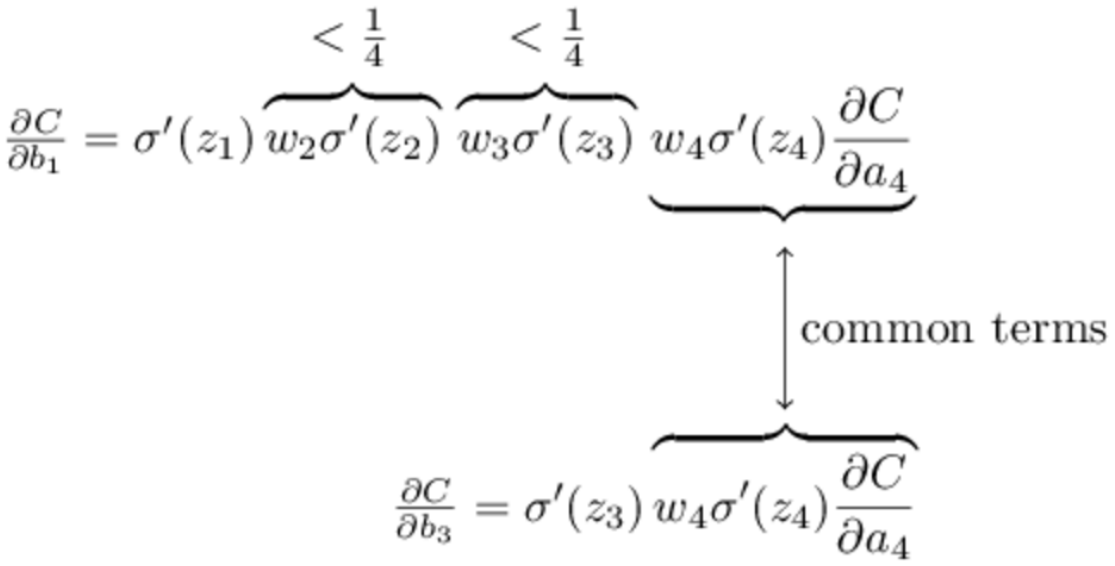
- note the shape of $\sigma'(z)$. its max value is just $1/4$



- its gradient is:

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

- If each $|w_i \sigma'(z_i)|$ term is small, then as the layer is deeper, we take more terms here and thus the gradient gets really really small.
- compare a shallower layer with a deeper layer:



  - this is the origin of the vanishing gradient problem: The derivative reaches a maximum at σ′(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 00 and standard deviation 11. 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 an informal argument, not a rigorous proof that the vanishing gradient problem will occur. There are several possible escape clauses. In particular, we might wonder whether the weights w_j could grow during training. If they do, it's possible the terms $w_j\sigma'(z_j)$ in the product will no longer satisfy $w_j\sigma'(z_j) < 1/4$. Indeed, if the terms get large enough - greater than 1 - then we will no longer have a vanishing gradient problem. Instead, the gradient will actually grow exponentially as we move backward through the layers.
  - The fundamental problem here isn't so much the vanishing gradient problem or the exploding gradient problem. It's that the gradient in early layers is the product of terms from all the later layers. When there are many layers, that's an intrinsically unstable situation. The only way all layers can learn at close to the same speed is if all those products of terms come close to balancing out. Without some mechanism or underlying reason for that balancing to occur, it's highly unlikely to happen simply by chance. In short, the real problem here is that neural networks suffer from an *unstable gradient problem*. → note the meaning of numerical instability here.
- exploding gradients: “The example is somewhat contrived. I'm going to fix parameters in the network in just the right way to ensure we get an exploding gradient. But even though the example is contrived, it has the virtue of firmly establishing that exploding gradients aren't merely a hypothetical possibility, they really can happen.”
- There are two steps to getting an exploding gradient. First, we choose all the weights in the network to be large, say $w_1 = w_2 = w_3 = w_4 = 100$. Second, we'll choose the biases so that the $\sigma'(z_j)$ terms are not too small. That's actually pretty easy to do: all we need do is choose the biases to ensure that the weighted input to each neuron is $z_j = 0$ (and so $\sigma'(z_j) = 1/4$). So, for instance, we want $z_1 = w_1a_0 + b_1 = 0$. We can achieve this by setting $b_1 = -100 * a_0$. We can use the same idea to select the other biases. When we do this, we see that all the terms $w_j\sigma'(z_j)$ are equal to some number > 1. With these choices we get an exploding gradient.

- Here we have used a super simple net. This argument can be extended to show why this is true in more realistic nets too.
- Note that if you're thinking "just multiply those weight changes by some reasonable number to magnify the amount of change", that won't work. The reason is because multiplying weights in different layers by different constants changes the overall gradient (the network has just one big gradient vector). changing the gradient arbitrarily no more guarantees we are moving in a good direction, let alone the best direction.
- Other challenges in training deep nets:
    - Researchers have found evidence suggesting that the use of sigmoid activation functions can cause problems training deep networks. In particular, they found evidence that the use of sigmoids will cause the activations in the final hidden layer to saturate near 0 early in training, substantially slowing down learning.
    - They have also studied the impact on deep learning of both the random weight initialization and the momentum schedule in momentum-based stochastic gradient descent. In both cases, making good choices made a substantial difference in the ability to train deep networks.
    - These examples suggest that "What makes deep networks hard to train?" is a complex question. In this chapter, we've focused on the instabilities associated to gradient-based learning in deep networks. The results in the last two paragraphs suggest that there is also a role played by the choice of activation function, the way weights are initialized, and even details of how learning by gradient descent is implemented. And, of course, choice of network architecture and other hyper-parameters is also important. Thus, many factors can play a role in making deep networks hard to train, and understanding all those factors is still a subject of ongoing research.