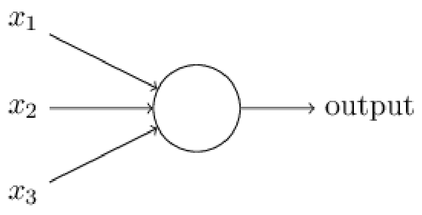
Credit goes to the original author Michael Nielsen. This work is just to keep track of my study. Here, I have also used resources from other media.

Perceptrons

What is a neural network? To get started, I'll explain a type of artificial neuron called a perceptron. Perceptrons were developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts. Today, it's more common to use other models of artificial neurons - in this book, and in much modern work on neural networks, the main neuron model used is one called the sigmoid neuron. We'll get to sigmoid neurons shortly. But to understand why sigmoid neurons are defined the way they are, it's worth taking the time to first understand perceptrons.

So how do perceptrons work? A perceptron takes several binary inputs, x1,x2,…, and produces a single binary output:



In the example shown the perceptron has three inputs, x1,x2,x3. In general it could have more or fewer inputs. Rosenblatt proposed a simple rule to compute the output. He introduced weights, w1,w2,…, real numbers expressing the importance of the respective inputs to the output. The neuron's output, 0 or 1, is determined by whether the weighted sum ∑j wjxj is less than or greater than some threshold value. Just like the weights, the threshold is a real number which is a parameter of the neuron. To put it in more precise algebraic terms:

0 if ∑j wjxj ≤ threshold

1 if ∑j wjxj > threshold

That's all there is to how a perceptron works!

That's the basic mathematical model. A way you can think about the perceptron is that it's a device that makes decisions by weighing up evidence. Let me give an example. It's not a very realistic example, but it's easy to understand, and we'll soon get to more realistic examples. Suppose the weekend is coming up, and you've heard that there's going to be a cheese festival in your city. You like cheese, and are trying to decide whether or not to go to the festival.

You might make your decision by weighing up three factors:

1. Is the weather good?
2. Does your boyfriend or girlfriend want to accompany you?
3. Is the festival near public transit? (You don't own a car).

We can represent these three factors by corresponding binary variables x1,x2, and x3. For instance, we'd have x1 = 1 if the weather is good, and x1 = 0 if the weather is bad. Similarly, x2 = 1 if your boyfriend or girlfriend wants to go, and x2 = 0 if not. And similarly again for x3 and public transit.

Now, suppose you absolutely adore cheese, so much so that you're happy to go to the festival even if your boyfriend or girlfriend is uninterested and the festival is hard to get to. But perhaps you really loathe bad weather, and there's no way you'd go to the festival if the weather is bad. You can use perceptrons to model this kind of decision-making. One way to do this is to choose a weight $w1 = 6$ for the weather, and $w2 = 2$ and $w3 = 2$ for the other conditions. The larger value of $w1$ indicates that the weather matters a lot to you, much more than whether your boyfriend or girlfriend joins you, or the nearness of public transit. Finally, suppose you choose a threshold of 5 for the perceptron. With these choices, the perceptron implements the desired decision-making model, outputting 1 whenever the weather is good, and 0 whenever the weather is bad. It makes no difference to the output whether your boyfriend or girlfriend wants to go, or whether public transit is nearby.

By varying the weights and the threshold, we can get different models of decision-making. For example, suppose we instead chose a threshold of 3. Then the perceptron would decide that you should go to the festival whenever the weather was good or when both the festival was near public transit and your boyfriend or girlfriend was willing to join you. In other words, it'd be a different model of decision-making. Dropping the threshold means you're more willing to go to the festival.

Obviously, the perceptron isn't a complete model of human decision-making! But what the example illustrates is how a perceptron can weigh up different kinds of evidence in order to make decisions. And it should seem plausible that a complex network of perceptrons could make quite subtle decisions:

In this network, the first column of perceptrons - what we'll call the first layer of perceptrons - is making three very simple decisions, by weighing the input evidence. What about the perceptrons in the second layer? Each of those perceptrons is making a decision by weighing up the results from the first layer of decision-making. In this way a perceptron in the second layer can make a decision at a more complex and more abstract level than perceptrons in the first layer. And even more complex decisions can be made by the perceptron in the third layer. In this way, a many-layer network of perceptrons can engage in sophisticated decision making.

Incidentally, when I defined perceptrons I said that a perceptron has just a single output. In the network above the perceptrons look like they have multiple outputs. In fact, they're still single output. The multiple output arrows are merely a useful way of indicating that the output from a perceptron is being used as the input to several other perceptrons. It's less unwieldy than drawing a single output line which then splits.

Let's simplify the way we describe perceptrons. The condition

$\sum j\ wjxj >$ threshold is cumbersome, and we can make two notational changes to simplify it. The first change is to write

$\sum j\ wjxj$ as a dot product, $w \cdot x \equiv \sum j\ wjxj$, where w and x are vectors whose components are the weights and inputs, respectively. The second change is to move the threshold to the other side of the inequality, and to replace it by what's known as the perceptron's bias, $b \equiv -$threshold. Using the bias instead of the threshold, the perceptron rule can be rewritten:

$$\begin{array}{ll} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{array}$$

Ow, wait. Can you explain bias a little more. It is still not clear what is meant by bias. What is it?

Ok, just think that bias is measurement of how much you are making your output relying on a particular input. In the example above, we are saying that if the weather is good then we will go to the festival, we will not even care the other factors if the weather is good. So, we are giving our threshold such a high value that only if there is a good weather then the value of the sum of all wjxj will surely cross the threshold value. Otherwise, we will never cross the threshold value according to the above example. How? Well if the weather is not good then x1 is 0. So w1x1 is also 0. Now if boyfriend or girlfriend wants to go and also transit is available then x2 and x3 both are 1. So w1x1 + w2x2 + w3x3 = 0+2+2 = 4. But our threshold value is 5. So, we will not go to the festival. So you see, we are biasing our output or our chance to go to the festival to the input x1 or perceptron of good weather by giving our threshold a high value. So, to simplify things we are using the term bias instead of threshold.

Ok, now deduce ∑j wjxj + b > 0 ?

∑j wjxj > threshold

Or, ∑j wjxj - threshold > threshold – threshold

Or, ∑j wjxj – threshold > 0

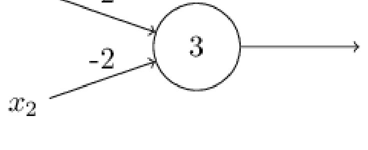
Or, ∑j wjxj + b > 0

Ok, got it. But why b ≡ −threshold ?

Now, as b ≡ −threshold, if ∑j wjxj is not greater then the threshold value, then ∑j wjxj + b or ∑j wjxj – threshold will give us a negative value or a zero value if ∑j wjxj is equal to the threshold. Otherwise, if ∑j wjxj is greater then the threshold then we will get a positive value.

You can think of the bias as a measure of how easy it is to get the perceptron to output a 1. Or to put it in more biological terms, the bias is a measure of how easy it is to get the perceptron to fire. For a perceptron with a really big bias, it's extremely easy for the perceptron to output a 1. But if the bias is very negative, then it's difficult for the perceptron to output a 1. Obviously, introducing the bias is only a small change in how we describe perceptrons, but we'll see later that it leads to further notational simplifications. Because of this, in the remainder of the book we won't use the threshold, we'll always use the bias.

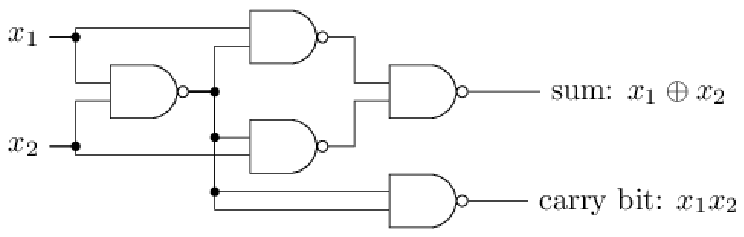
I've described perceptrons as a method for weighing evidence to make decisions. Another way perceptrons can be used is to compute the elementary logical functions we usually think of as underlying computation, functions such as AND, OR, and NAND. For example, suppose we have a perceptron with two inputs, each with weight −2, and an overall bias of 3. Here's our perceptron:



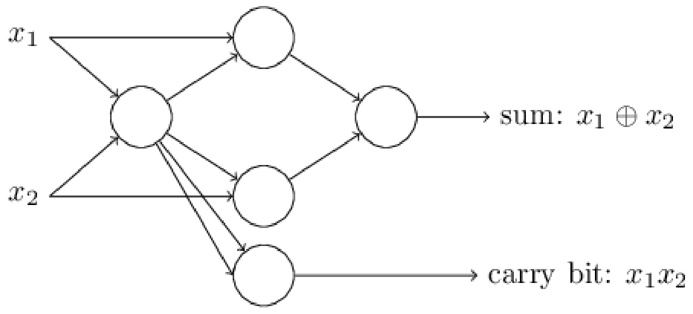
Then we see that input 00 produces output 1, since

$(-2) * 0 + (-2) * 0 + 3 = 3$ is positive. Here, I've introduced the $*$ symbol to make the multiplications explicit. Similar calculations show that the inputs 01 and 10 produce output 1. But the input 11 produces output 0, since $(-2) * 1 + (-2) * 1 + 3 = -1$ is negative. And so our perceptron implements a NAND gate!

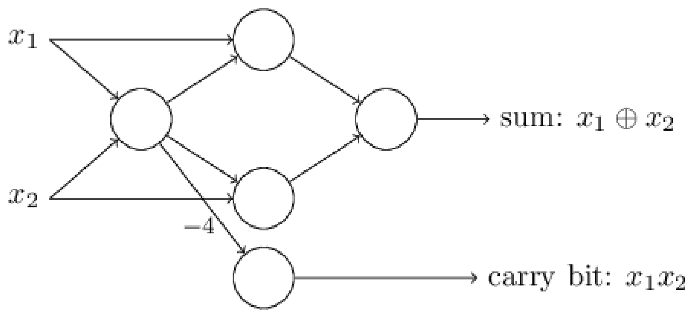
The NAND example shows that we can use perceptrons to compute simple logical functions. In fact, we can use networks of perceptrons to compute any logical function at all. The reason is that the NAND gate is universal for computation, that is, we can build any computation up out of NAND gates. For example, we can use NAND gates to build a circuit which adds two bits, x1 and x2. This requires computing the bitwise sum, x1 ⊕ x2, as well as a carry bit which is set to 1 when both x1 and x2 are 1, i.e., the carry bit is just the bitwise product x1x2:



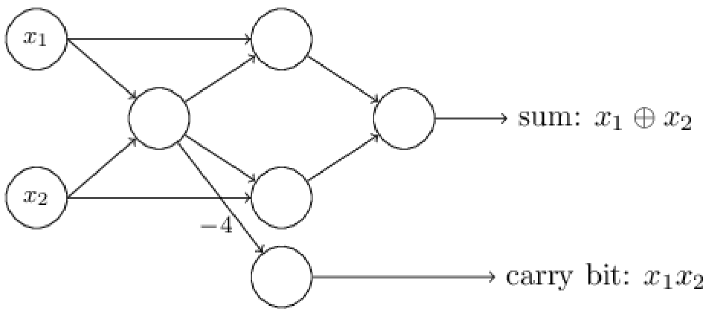
To get an equivalent network of perceptrons we replace all the NAND gates by perceptrons with two inputs, each with weight −2, and an overall bias of 3. Here's the resulting network. Note that I've moved the perceptron corresponding to the bottom right NAND gate a little, just to make it easier to draw the arrows on the diagram:



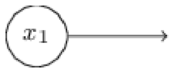
One notable aspect of this network of perceptrons is that the output from the leftmost perceptron is used twice as input to the bottommost perceptron. When I defined the perceptron model I didn't say whether this kind of double-output-to-the-same-place was allowed. Actually, it doesn't much matter. If we don't want to allow this kind of thing, then it's possible to simply merge the two lines, into a single connection with a weight of -4 instead of two connections with -2 weights. (If you don't find this obvious, you should stop and prove to yourself that this is equivalent.) With that change, the network looks as follows, with all unmarked weights equal to -2, all biases equal to 3, and a single weight of -4, as marked:



Up to now I've been drawing inputs like x1 and x2 as variables floating to the left of the network of perceptrons. In fact, it's conventional to draw an extra layer of perceptrons - the input layer - to encode the inputs:



This notation for input perceptrons, in which we have an output, but no inputs,



is a shorthand. It doesn't actually mean a perceptron with no inputs. To see this, suppose we did have a perceptron with no inputs. Then the weighted sum ∑j wjxj would always be zero, and so the perceptron would output 1 if b > 0, and 0 if b ≤ 0. That is, the perceptron would simply output a fixed value, not the desired value (x1, in the example above). It's better to think of the input perceptrons as not really being perceptrons at all, but rather special units which are simply defined to output the desired values, x1,x2,….

The adder example demonstrates how a network of perceptrons can be used to simulate a circuit containing many NAND gates. And because NAND gates are universal for computation, it follows that perceptrons are also universal for computation.

The computational universality of perceptrons is simultaneously reassuring and disappointing. It's reassuring because it tells us that networks of perceptrons can be as powerful as any other computing device. But it's also disappointing, because it makes it seem as though perceptrons are merely a new type of NAND gate. That's hardly big news!

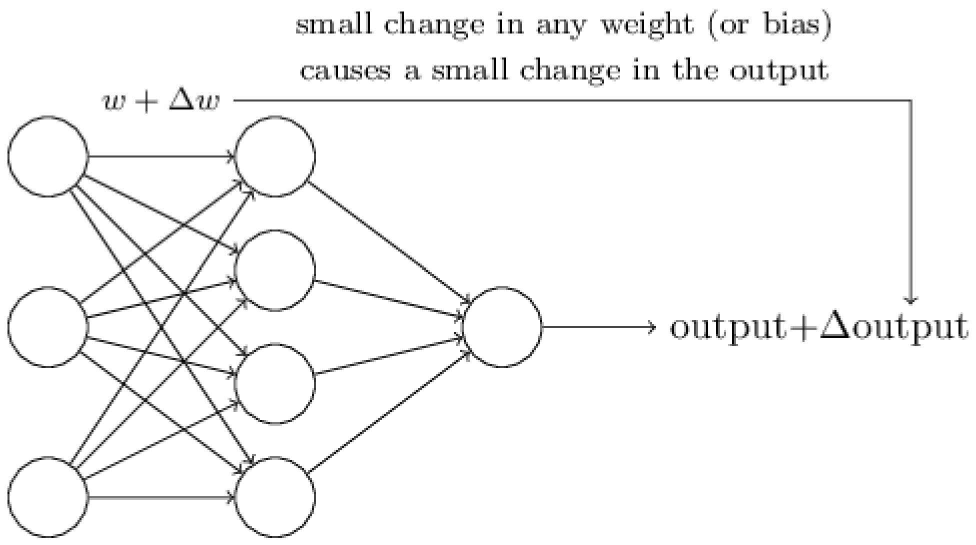
However, the situation is better than this view suggests. It turns out that we can devise learning algorithms which can automatically tune the weights and biases of a network of artificial neurons. This tuning happens in response to external stimuli, without direct intervention by a programmer. These learning algorithms enable us to use artificial neurons in a way which is radically different to conventional logic gates. Instead of explicitly laying out a circuit of NAND and other gates, our neural networks can simply learn to solve problems, sometimes problems where it would be extremely difficult to directly design a conventional circuit.

Ok, If we use bias then why using weight ?

If we don’t use weights, then the sum of other inputs may overcome the threshold value which is not as expected. We use weights to control the behavior of inputs so that the we exceed the threshold value only for our desired input value.

Sigmoid neurons

Learning algorithms sound terrific. But how can we devise such algorithms for a neural network? Suppose we have a network of perceptrons that we'd like to use to learn to solve some problem. For example, the inputs to the network might be the raw pixel data from a scanned, handwritten image of a digit. And we'd like the network to learn weights and biases so that the output from the network correctly classifies the digit. To see how learning might work, suppose we make a small change in some weight (or bias) in the network. What we'd like is for this small change in weight to cause only a small corresponding change in the output from the network. As we'll see in a moment, this property will make learning possible. Schematically, here's what we want (obviously this network is too simple to do handwriting recognition!):

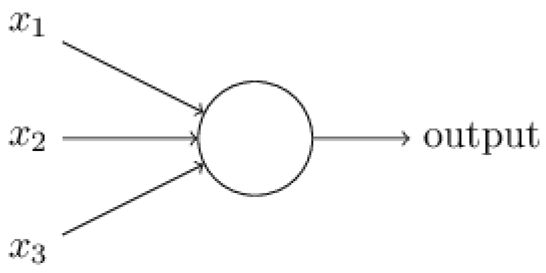

small change in any weight (or bias)
causes a small change in the output
$w + \Delta w$
output+Δoutput

If it were true that a small change in a weight (or bias) causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want. For example, suppose the network was mistakenly classifying an image as an "8" when it should be a "9". We could figure out how to make a small change in the weights and biases so the network gets a little closer to classifying the image as a "9". And then we'd repeat this, changing the weights and biases over and over to produce better and better output. The network would be learning.

The problem is that this isn't what happens when our network contains perceptrons. In fact, a small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 0 to 1. That flip may then cause the behaviour of the rest of the network to completely change in some very complicated way. So while your "9" might now be classified correctly, the behaviour of the network on all the other images is likely to have completely changed in some hard-to-control way. That makes it difficult to see how to gradually modify the weights and biases so that the network gets closer to the desired behaviour. Perhaps there's some clever way of getting around this problem. But it's not immediately obvious how we can get a network of perceptrons to learn.

We can overcome this problem by introducing a new type of artificial neuron called a *sigmoid* neuron. Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output. That's the crucial fact which will allow a network of sigmoid neurons to learn.

Okay, let me describe the sigmoid neuron. We'll depict sigmoid neurons in the same way we depicted perceptrons:



Just like a perceptron, the sigmoid neuron has inputs, $x1,x2,\ldots$. But instead of being just 0 or 1, these inputs can also take on any values *between* 0 and 1. So, for instance, $0.638\ldots$ is a valid input for a sigmoid neuron. Also just like a perceptron, the sigmoid neuron has weights for each input, $w1,w2,\ldots$, and an overall bias, b. But the output is not 0 or 1. Instead, it's $\sigma(w\cdot x+b)$, where σ is called the *sigmoid function.* Incidentally, σ is sometimes called the *logistic function*, and this new class of neurons called *logistic neurons*. It's useful to remember this terminology, since these terms are used by many people working with neural nets. However, we'll stick with the sigmoid terminology:

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}. \tag{3}$$

To put it all a little more explicitly, the output of a sigmoid neuron with inputs $x1,x2,\ldots$, weights $w1,w2,\ldots$, and bias b is

$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}. \tag{4}$$

I did not understand the equation? Explain it more

Ok, so here z = Σw.x + b. for simplicity, let us suppose we have only one neuron who’s input is x, weight is w and bias is b. And so, z = w.x + b

From perceptron rule we know that:

$0 \quad \text{if } w \cdot x + b \leq 0$

1 if w · x + b > 0

Now for perceptron, we could imagine the activation function to be something like this, if w · x + b > 0:

$\sigma(z) = (w.x + b) + (1 - (w.x + b))$.

So, if (w.x + b) > 0 for example if (w.x + b) is 0.65 then →

$\sigma(z) = 0.65 + (1 - 0.65)$

$= 0.65 + 0.25$

$= 1$

So, the neuron will be activated.

Again, we could imagine the activation function to be something like this, if w · x + b ≤ 0:

$\sigma(z) = (w.x + b)*0$

So, for example if (w.x + b) is - 0.65 then →

$\sigma(z) = - 0.65 * 0$

$= 0$

So, the neuron will not be activated.

Similarly, for sigmoid neurons, the activation funtion is :

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}. \qquad (3)$$

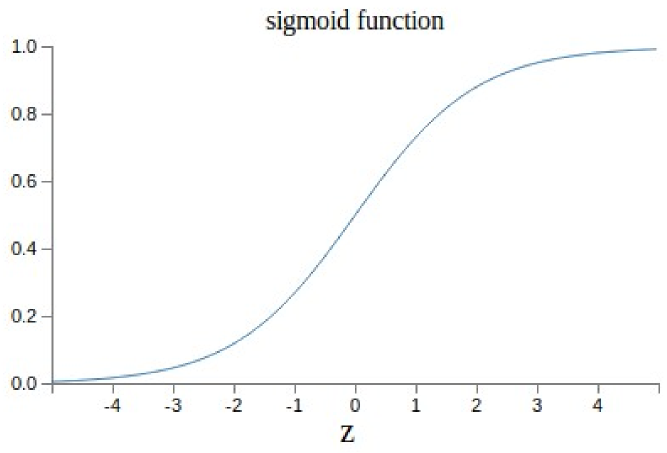
This sigmoid function is the activation function for sigmoid neurons. And also, do not actually think that sigmoid function is applicable for perceptrons. That is just for illustration for your better understanding. Actually, there is no need to have a sigmoid function as an activation function for perceptrons.

At first sight, sigmoid neurons appear very different to perceptrons. The algebraic form of the sigmoid function may seem opaque and forbidding if you're not already familiar with it. In fact, there are many similarities between perceptrons and sigmoid neurons, and the algebraic form of the sigmoid function turns out to be more of a technical detail than a true barrier to understanding.

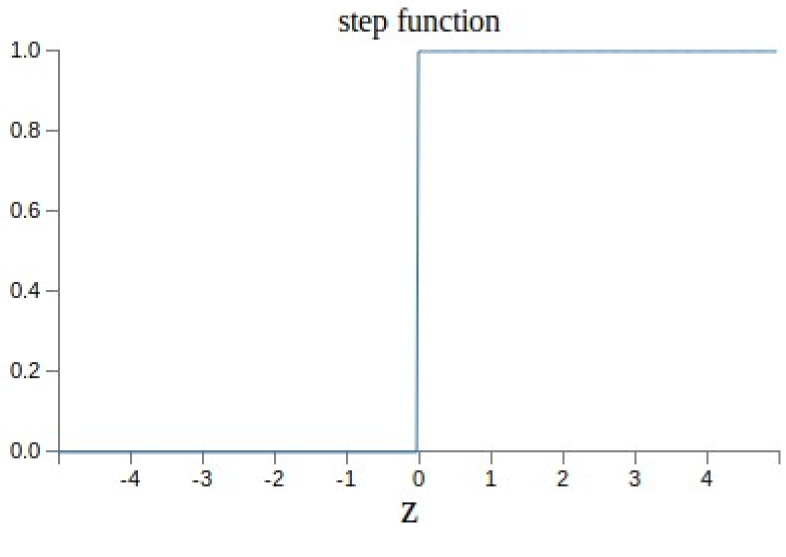
To understand the similarity to the perceptron model, suppose $z \equiv w \cdot x + b$ is a large positive number. Then $e^{-z} \approx 0$ and so $\sigma(z) \approx 1$. In other words, when $z = w \cdot x + b$ is large and positive, the output from the sigmoid neuron is approximately 1, just as it would have been for a perceptron. Notice carefully that I said approximately 1 like 0.65 or 0.97 or 0.911 or similar number between 0 and 1, not exactly 1. Suppose on the other hand that $z = w \cdot x + b$ is very negative. Then $e^{-z} \rightarrow \infty$, and $\sigma(z) \approx 0$. Again notice carefully that I said approximately 0 like 0.065 or 0.097 or 0.011 or similar number between 0 and 1, not exactly 0. So when $z = w \cdot x + b$ is very negative, the behaviour of a sigmoid neuron also closely approximates a perceptron. It's only when $w \cdot x + b$ is of modest size that there's much deviation from the perceptron model.

This also answer the question **why we choose the algebraic form in equation (3) for the sigmoid function.** Because incase of sigmoid neuron, we accept all the numbers between 0 and 1, unlike perceptrons where we just accept either 0 or 1.

In fact, the exact form of σ isn't so important - what really matters is the shape of the function when plotted. Here's the shape:



This shape is a smoothed out version of a step function:



If σ had in fact been a step function, then the sigmoid neuron would *be* a perceptron, since the output would be 1 or 0 depending on whether $w \cdot x+b$ was positive or negative .Actually, when $w \cdot x+b=0$ the perceptron outputs 0, while the step function outputs 1. So, strictly speaking, we'd need to modify the step function at that one point. But you get the idea. By using the actual σ function we get, as already implied above, a smoothed out perceptron. Indeed, it's the smoothness of the σ function that is the crucial fact, not its detailed form. The smoothness of σ means that small changes Δw_j in the weights and Δb in the bias will produce a small change $\Delta \mathrm{output}$ in the output from the neuron. In fact, calculus tells us that $\Delta \mathrm{output}$ is well approximated by:

$$\Delta \text{output} \approx \sum_j \frac{\partial\, \text{output}}{\partial w_j} \Delta w_j + \frac{\partial\, \text{output}}{\partial b} \Delta b, \qquad (5)$$

where the sum is over all the weights, w_j, and $\partial \text{output}/\partial w_j$ and $\partial \text{output}/\partial b$ denote partial derivatives of the output with respect to w_j and b, respectively.

I did not understand. Where is the ‘x’ of ‘w.x + b’ in equation (5) ?

Well, first of all we are here talking about what are the factors that actually changes the output. The output changes if weights w and bias b changes. So, we omit the symbol ‘x’ to illustrate how the output changes with the change in weights w and bias b. Read it like this, we are first determining how much the output changes with respect to very small change or unit change in weight w_j which is ‘∂w_j’. With respect to this, we are figuring out how much the output changes with our change in w_j (which is ‘ΔW_j’). We are doing this for all w_j and summing them up. We are adding the changes in output with respect to our change in bias b (which is ‘Δb’) to that result. Thus we are figuring out the actual approximation of the change in output with respect to our change in weight and bias.

Don't panic if you're not comfortable with partial derivatives! While the expression above looks complicated, with all the partial derivatives, it's actually saying something very simple (and which is very good news): Δoutput is a *linear function* of the changes Δw_j and Δb in the weights and bias. This linearity makes it easy to choose small changes in the weights and biases to achieve any desired small change in the output. So while sigmoid neurons have much of the same qualitative behaviour as perceptrons, they make it much easier to figure out how changing the weights and biases will change the output.

If it's the shape of σ which really matters, and not its exact form, then why use the particular form used for σ in Equation (3)? In fact, later in the book we will occasionally consider neurons where the output is $f(w \cdot x + b)$ for some other *activation function* $f(\cdot)$. The main thing that changes when we use a different activation function is that the particular values for the partial derivatives in Equation (5) change. It turns out that when we compute those partial derivatives later, using σ will simplify the algebra, simply because exponentials have lovely properties when differentiated. In any case, σ is commonly-used in work on neural nets, and is the activation function we'll use most often in this book.

How should we interpret the output from a sigmoid neuron? Obviously, one big difference between perceptrons and sigmoid neurons is that sigmoid neurons don't just output 0 or 1. They can have as output any real number between 0 and 1, so values such as $0.173\ldots$ and $0.689\ldots$ are legitimate outputs. This can be useful, for example, if we want to use the output value to represent the average intensity of the pixels in an image input to a neural network. But sometimes it can be a nuisance. Suppose we want the output from the network to indicate either "the input image is a 9" or "the input image is not a 9". Obviously, it'd be easiest to do this if the output was a 0 or a 1, as in a perceptron. But in practice we can set up a convention to deal with this, for example, by deciding to interpret any output of at least 0.5 as indicating a "9", and any output less than 0.5 as indicating "not a 9". I'll always explicitly state when we're using such a convention, so it shouldn't cause any confusion.

Exercises

Sigmoid neurons simulating perceptrons, part I

Suppose we take all the weights and biases in a network of perceptrons, and multiply them by a positive constant, $c > 0$. Show that the behaviour of the network doesn't change.

Ok, lets again look at the perceptron rule. From perceptron rule we know that:

$$\begin{array}{ll} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{array}$$

Let’s assume a scenario where $x = 2$, $w = 2$, Threshold = 2. So,

b = - threshold

or, $b = -2$

So, from our perceptron rule we get $\rightarrow$

$(2 * 2) - 2 = 2$

Now, let us assume $c = 2$ and multiply weight and bias by c. So $\rightarrow$

$(4 * 2) – 4 = 8 – 4 = 4$.

Thus if you notice you will see that if $w \cdot x + b > 0$ then multiplying it’s weights and biases by a positive constant $c > 0$ will also always result in $w \cdot x + b > 0$ and so the output will be always 1 .

If $w \cdot x + b \leq 0$ then multiplying it’s weights and biases by a positive constant $c > 0$ will also always result in $w \cdot x + b \leq 0$ and so the output will be always 0 .

So if we take all the weights and biases in a network of perceptrons, and multiply them by a positive constant, $c > 0$, the behaviour of the network doesn't change.

But if we take all the weights and biases in a network of perceptrons, and multiply them by a negative constant, $c < 0$, the behaviour of the network will change.

Sigmoid neurons simulating perceptrons, part II

Suppose we have the same setup as the last problem – a network of perceptrons. Suppose also that the overall input to the network of perceptrons has been chosen. We won't need the actual input value, we just need the input to have been fixed. Suppose the weights and biases are such that $w \cdot x + b \neq 0$ for the input x to any particular perceptron in the network. Now replace all the perceptrons in the network by sigmoid neurons, and multiply the weights and biases by a positive constant $c > 0$. Show that in the limit as $c \rightarrow \infty$ the behaviour of this network of sigmoid neurons is exactly the same as the network of perceptrons. How can this fail when $w \cdot x + b = 0$ for one of the perceptrons?

Let’s assume a scenario where $x = 2$, $w = 2$, Threshold = 2. So,

b = - threshold

or, $b = -2$

We know that $z = w.x + b$. So $\rightarrow$

$z = w.x + b$
or, $z = (2 * 2) - 2$
or, $z = 4 – 2$
or, $z = 2$

So, putting this value of z in the activation function for the sigmoid neuron we get $\rightarrow$

$\sigma(z) = 1 / (1 + e^{-z})$
or, $\sigma(w.x+b) = 1 / (1 + e^{-(w.x+b)})$
or, $\sigma(w.x+b) = 1 / (1 + e^{-2})$
or, $\sigma(w.x+b) = 1 / (1 + 0.1353)$
or, $\sigma(w.x+b) = 1 / 1.1353$
or, $\sigma(w.x+b) = 0.8808$

Now suppose we mulitply weight w and bias b by a positive constant c. Also suppose that mulitplying weight w and bias b by that positive constant c results in z = 3. Then,

$\sigma(w.x+b) = 0.9525$

Now suppose we increase the positive constant c and mulitply weight w and bias b by that positive constant c. Also suppose that mulitplying weight w and bias b by that positive constant c results in z = 4. Then,

$\sigma(w.x+b) = 0.9820$

We know from perceptron rule for perceptrons that if $(w.x + b) > 0$ then the output of the activation function for perceptrons will be 1 .

Here for sigmoid neurons, we see that if $(w.x + b) > 0$ then for $c \rightarrow \infty$ then $\sigma(w.x+b) \rightarrow 1$.

Now again suppose we mulitply weight w and bias b by a positive constant c. Also suppose that mulitplying weight w and bias b by that positive constant c results in z = -2. Then,

$\sigma(w.x+b) = 0.1192$

Now suppose we increase the positive constant c and mulitply weight w and bias b by that positive constant c. Also suppose that mulitplying weight w and bias b by that positive constant c results in z = -3. Then,

$\sigma(w.x+b) = 0.0474$

Now suppose we increase the positive constant c and mulitply weight w and bias b by that positive constant c. Also suppose that mulitplying weight w and bias b by that positive constant c results in z = -4. Then,

$\sigma(w.x+b) = 0.0179$

We know from perceptron rule for perceptrons that if $(w.x + b) < 0$ then the output of the activation function for perceptrons will be 0 .

Here for sigmoid neurons, we see that if $(w.x + b) < 0$ then for $c \to \infty$ then $\sigma(w.x+b) \to 0$.

So we have shown that in the limit as $c \to \infty$ the behaviour of this network of sigmoid neurons is exactly the same as the network of perceptrons if $w \cdot x + b \neq 0$.

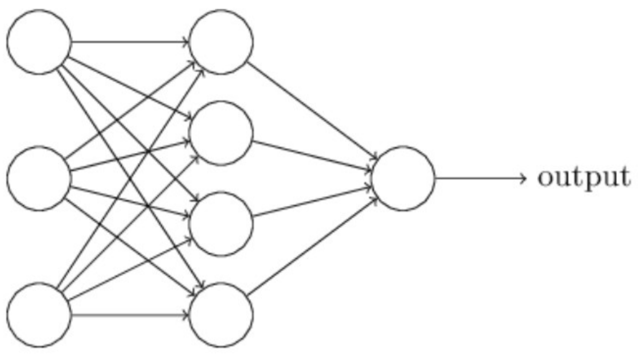
If $w \cdot x + b = 0$ then what happens ? Say $w \cdot x + b = 0$ for some weight w, bias b and input x. So,

$\sigma(z) = 1 / (1 + e^{-z})$
or, $\sigma(z) = 1 / (1 + e^{-0})$
or, $\sigma(z) = 1 / (1 + 1)$
or, $\sigma(z) = 1 / 2$
or, $\sigma(z) = 0.5$

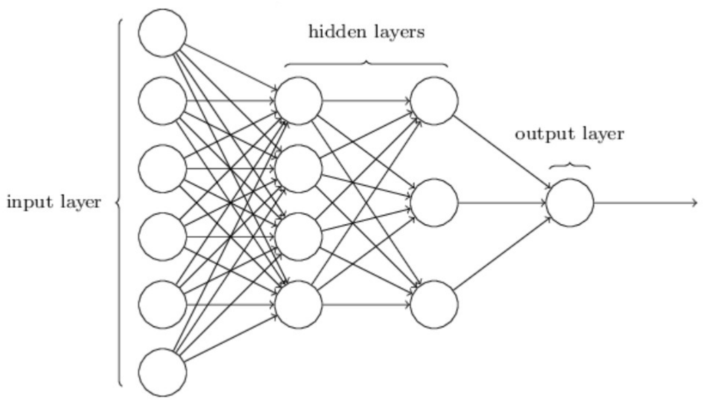
But for perceptrons, we know that if $w \cdot x + b = 0$ then the output is also 0. So we have got the answer for the second part of the question by showing why this fail when $w \cdot x + b = 0$ for one of the perceptrons.

The architecture of neural networks

In the next section I'll introduce a neural network that can do a pretty good job classifying handwritten digits. In preparation for that, it helps to explain some terminology that lets us name different parts of a network. Suppose we have the network:



As mentioned earlier, the leftmost layer in this network is called the input layer, and the neurons within the layer are called input neurons. The rightmost or output layer contains the output neurons, or, as in this case, a single output neuron. The middle layer is called a hidden layer, since the neurons in this layer are neither inputs nor outputs. The term "hidden" perhaps sounds a little mysterious - the first time I heard the term I thought it must have some deep philosophical or mathematical significance - but it really means nothing more than "not an input or an output". The network above has just a single hidden layer, but some networks have multiple hidden layers. For example, the following four-layer network has two hidden layers:



Somewhat confusingly, and for historical reasons, such multiple layer networks are sometimes called multilayer perceptrons or MLPs, despite being made up of sigmoid neurons, not perceptrons. I'm not going to use the MLP terminology in this book, since I think it's confusing, but wanted to warn you of its existence.

The design of the input and output layers in a network is often straightforward. For example, suppose we're trying to determine whether a handwritten image depicts a "9" or not. A natural way to design the network is to encode the intensities of the image pixels into the input neurons. If the image is a 64 by 64 greyscale image, then we'd have $4,096 = 64 \times 64$ input neurons, with the intensities scaled appropriately between 0 and 1 . The output layer will contain just a single neuron, with output values of less than 0.5 indicating "input image is not a 9", and values greater than 0.5 indicating "input image is a 9 ".

While the design of the input and output layers of a neural network is often straightforward, there can be quite an art to the design of the hidden layers. In particular, it's not possible to sum up the design process for the hidden layers with a few simple rules of thumb. Instead, neural networks researchers have developed many design heuristics for the hidden layers, which help people get the behaviour they want out of their nets. For example, such heuristics can be used to help determine how to trade off the number of hidden layers against the time required to train the network. We'll meet several such design heuristics later in this book.

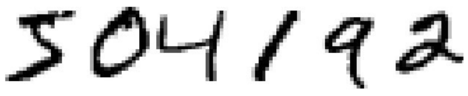
Up to now, we've been discussing neural networks where the output from one layer is used as input to the next layer. Such networks are called **feedforward neural networks**. This means there are no loops in the network - information is always fed forward, never fed back. If we did have loops, we'd end up with situations where the input to the σ function depended on the output of that σ function. Which means the input of the activation function of a sigmoid neuron is depended on the output of that same sigmoid neuron. That'd be hard to make sense of, and so we don't allow such loops.

However, there are other models of artificial neural networks in which feedback loops are possible. These models are called recurrent neural networks. The idea in these models is to have neurons which fire for some limited duration of time, before becoming quiescent. That firing can stimulate other neurons, which may fire a little while later, also for a limited duration. That causes still more neurons to fire, and so over time we get a cascade of neurons firing. Loops don't cause problems in such a model, since a neuron's output only affects its input at some later time, not instantaneously.

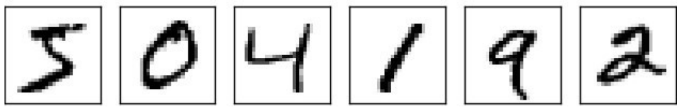
Recurrent neural nets have been less influential than feedforward networks, in part because the learning algorithms for recurrent nets are (at least to date) less powerful. But recurrent networks are still extremely interesting. They're much closer in spirit to how our brains work than feedforward networks. And it's possible that recurrent networks can solve important problems which can only be solved with great difficulty by feedforward networks. However, to limit our scope, in this book we're going to concentrate on the more widely-used feedforward networks.

A simple network to classify handwritten digits

Having defined neural networks, let's return to handwriting recognition. We can split the problem of recognizing handwritten digits into two sub-problems. First, we'd like a way of breaking an image containing many digits into a sequence of separate images, each containing a single digit. For example, we'd like to break the image

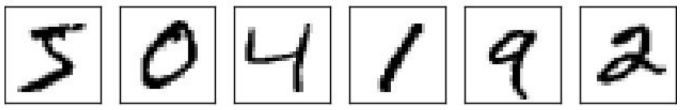


into six separate images,

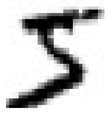


We humans solve this segmentation problem with ease, but it's challenging for a computer program to correctly break up the image. Once the image:

has been segmented into six different digits:



Here each digit is a segment. Then the program then needs to classify each individual digit. So, for instance, we'd like our program to recognize that the first segment or the first digit above,



Which is a 5.

Do we always need to segment an image into images or segments of single digit ?

No, that totally depends on our choice. Above we have segmented the image:

504192

into six segments like:

5,0,4,1,9,2

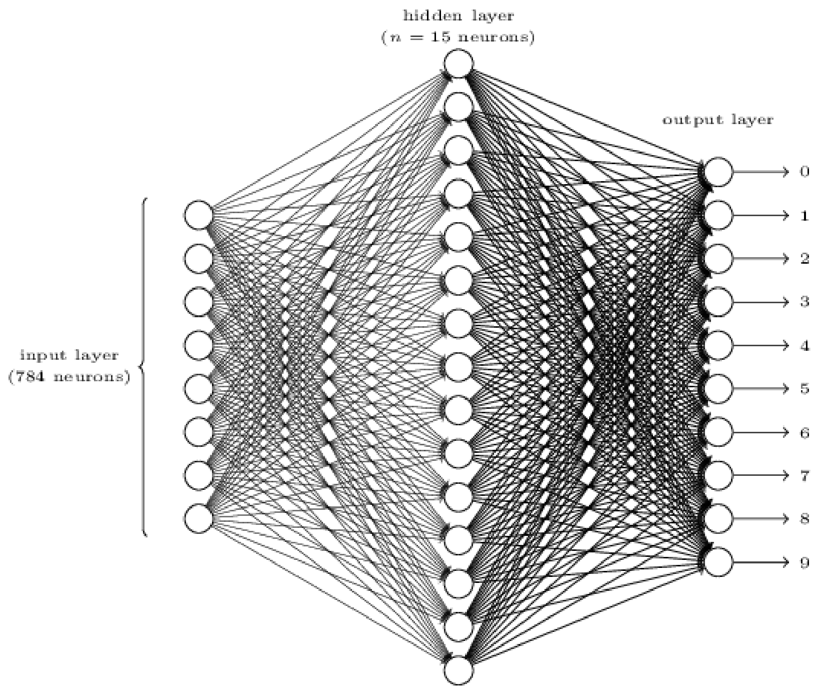
But if we want we can segment the image into i.e. three segments like:

504, 1, 92

How we want to segment an image totally depends on us. But a good segmentation is necessary to have correct output.

We'll focus on writing a program to solve the second problem, that is, classifying individual digits. We do this because it turns out that the segmentation problem is not so difficult to solve, once you have a good way of classifying individual digits. There are many approaches to solving the segmentation problem. One approach is to trial many different ways of segmenting the image, using the individual digit classifier to score each trial segmentation. A trial segmentation gets a high score if the individual digit classifier is confident of its classification in all segments, and a low score if the classifier is having a lot of trouble to tell which digit is it in one or more segments. The idea is that if the classifier is having trouble somewhere, then it's probably having trouble because the segmentation has been chosen incorrectly. This idea and other variations can be used to solve the segmentation problem quite well. So instead of worrying about segmentation we'll concentrate on developing a neural network which can solve the more interesting and difficult problem, namely, recognizing individual handwritten digits.

To recognize individual digits we will use a three-layer neural network:



The input layer of the network contains neurons encoding the values of the input pixels. As discussed in the next section, our training data for the network will consist of many 28 by 28 pixel images of scanned handwritten digits, and so the input layer contains $784 = 28 \times 28$ neurons. For simplicity I've omitted most of the 784 input neurons in the diagram above. The input pixels are greyscale, with a value of 0.0 representing white, a value of 1.0 representing black, and in between values representing gradually darkening shades of grey. Notice we are here accepting values in between 0.0 and 1.0 because these are not perceptrons, these are sigmoid neurons.

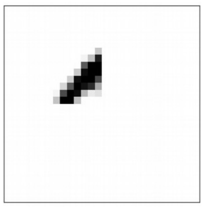
The second layer of the network is a hidden layer. We denote the number of neurons in this hidden layer by n, and we'll experiment with different values for n. The example shown illustrates a small hidden layer, containing just $n = 15$ neurons.

The output layer of the network contains 10 neurons. If the first neuron fires, i.e., has an output ≈ 1, then that will indicate that the network thinks the digit is a 0. If the second neuron fires then that will indicate that the network thinks the digit is a 1. And so on. A little more precisely, we number the output neurons from 0 through 9, and figure out which neuron has the highest activation value. If that neuron is, say, neuron number 6, then our network will guess that the input digit was a 6. And so on for the other output neurons.

You might wonder why we use 10 output neurons. After all, the goal of the network is to tell us which digit ($0,1,2,\ldots,9$) corresponds to the input image. A seemingly natural way of doing that is to use just 4 output neurons, treating each neuron as taking on a binary value, depending on whether the neuron's output is closer to 0 or to 1. Four neurons are enough to encode the answer, since $2^4 = 16$ is more than the 10 possible values for the input digit. Why should our network use 10 neurons instead? Isn't that inefficient? The ultimate justification is empirical: we can try out both network designs, and it turns out that, for this particular problem, the network with 10 output neurons learns to recognize digits better than the network with 4 output neurons. But that leaves us wondering why using 10

output neurons works better. Is there some heuristic that would tell us in advance that we should use the 10-output encoding instead of the 4-output encoding?

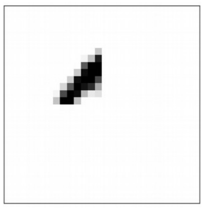
To understand why we do this, it helps to think about what the neural network is doing from first principles. Consider first the case where we use 10 output neurons. Let's concentrate on the first output neuron, the one that's trying to decide whether or not the digit is a 0. It does this by weighing up evidence from the hidden layer of neurons. What are those hidden neurons doing? Well, just suppose for the sake of argument that the first neuron in the hidden layer detects whether or not an image like the following is present:



It can do this by heavily weighting input pixels which overlap with the image, and only lightly weighting the other inputs.

What do you mean? I can not understand, please explain.

Ok, suppose we are giving an image depicting ‘0’ as an input to our neural network. So, as we said earlier we have assumed that our input image is a 28 by 28 pixels image. So, we need 28 * 28 = 784 input neurons in our neural network. Now as we said earlier, the first neuron in the hidden layer detects whether or not an image like the following is present:



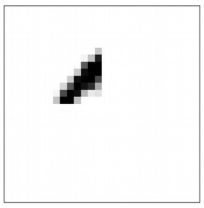
But how ? Well lets see the above image. We can see that some of the pixels in that image are black and the rest are white. Suppose in the above image, pixel number 50,51,52,53,54 are black among those 784 pixels and the rest of the pixels of those 784 pixels are white.

Now, first we need to set up a threshold value for the first neuron in the hidden layer. Remember w.x + b ? where b = - threshold ? Yes we need to set up the value of ‘b’ for the first sigmoid neuron in the hidden layer of our neural network. So, suppose we have set up a threshold value for the first neuron in the hidden layer. Ok? Lets move on.

Now we will increase the weight of pixels 50,51,52,53,54 of our input image.

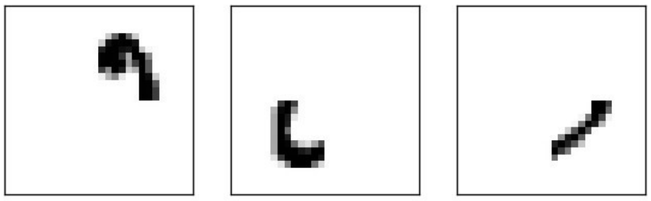
Our input image is not the above image. Don’t get confused. The above image depicts a section which the first neuron in the hidden layer is looking for in our input image. If the black section containing pixels 50,51,52,53,54 in the above image is present in our input image, which means if the pixels 50,51,52,53,54 of our input image are also black, only then the first neuron in the hidden layer will get activated.

So, where were we ? We were going to increase the weight of pixels 50,51,52,53,54 of our input image and decrease the weights of the rest of the pixels of our input image. Lets do that. After doing that, lets compute Σw.x. We have already set up the value of ‘b’ previously. After getting the value of Σw.x , lets compute Σw.x + b. Now if we see that Σw.x + b > 0, then we know that the activation function of the first sigmoid neuron in the hidden layer which is $\sigma(z) = 1/(1 + e^{-z})$ will be positive and if it is positive enough then our neural network will say that the black section in the image:

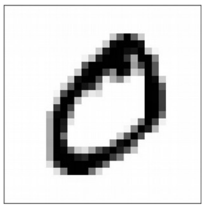


is present in our input image and so the first sigmoid neuron in the hidden layer will get activated. Otherwise it will not get activated.

In a similar way, let's suppose for the sake of argument that the second, third, and fourth neurons in the hidden layer detect whether or not the following images are present:



As you may have guessed, these four images together make up the 0 image that we saw in the line of digits shown earlier:



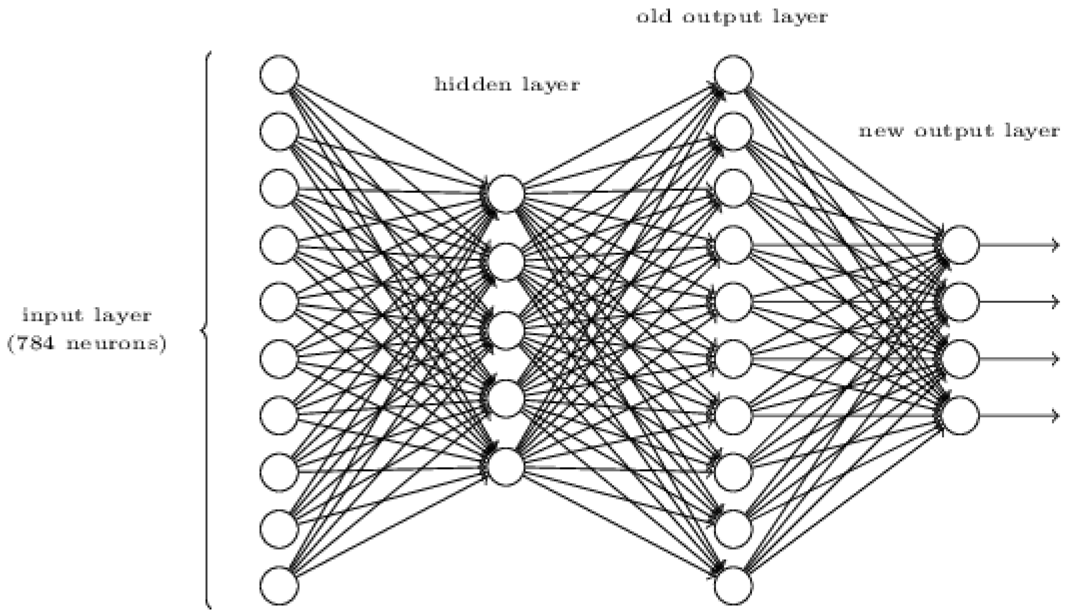
So if all four of these hidden neurons are firing then we can conclude that the digit is a 0. Of course, that's not the only sort of evidence we can use to conclude that the image was a 0 - we could legitimately get a 0 in many other ways (say, through translations of the above images, or slight distortions). But it seems safe to say that at least in this case we'd conclude that the input was a 0.

Supposing the neural network functions in this way, we can give a plausible explanation for why it's better to have 10 outputs from the network, rather than 4. If we had 4 outputs, then the first output neuron would be trying to decide what the most significant bit of the digit was. And there's no easy way to relate that most significant bit to simple shapes like those shown above. It's hard to imagine that there's any good historical reason the component shapes of the digit will be closely related to (say) the most significant bit in the output.

Now, with all that said, this is all just a heuristic. Nothing says that the three-layer neural network has to operate in the way I described, with the hidden neurons detecting simple component shapes. Maybe a clever learning algorithm will find some assignment of weights that lets us use only 4 output neurons. But as a heuristic the way of thinking I've described works pretty well, and can save you a lot of time in designing good neural network architectures.

Exercise

- There is a way of determining the bitwise representation of a digit by adding an extra layer to the three-layer network above. The extra layer converts the output from the previous layer into a binary representation, as illustrated in the figure below. Find a set of weights and biases for the new output layer. Assume that the first 3 layers of neurons are such that the correct output in the third layer (i.e., the old output layer) has activation at least 0.99, and incorrect outputs have activation less than 0.01.



Ok, let us look at a case where we suppose that we have given an image of 1 as an input to the input layer and the second sigmoid neuron of the old output layer of the image above gets fired. We know that the binary representation of 1 is 0001.

We can see from the image above that the output of the second sigmoid neuron works as an input to the four sigmoid neuron in the new output layer. Suppose threshold =1. So, bias b = -threshold = -1. Now suppose, for the second sigmoid neuron in the old output layer, we take a weight w= 0.01. We supposed earlier that the second sigmoid neuron of the old output layer is 0.99. So, for the second sigmoid neuron of the old output layer, w.x = (0.01*0.99) = 0.0099. In our assumed case, all the other sigmoid neurons except the first sigmoid neuron of the old output layer have activation less than 0.01. Let us assume that all the other sigmoid neurons except the first sigmoid neuron of the old output layer have activation 0.001. Suppose all those other sigmoid neurons have weight 0.01. So, for all those other sigmoid neurons, w.x is (0.01*0.001) = 0.00001.

So, for the first sigmoid neuron in the new output layer,

Σw.x = 0.00001+0.0099+0.00001+ 0.00001+ 0.00001+ 0.00001+ 0.00001+ 0.00001+ 0.00001+ 0.00001

or, Σw.x = 0.00999

Now, suppose for the first sigmoid neuron of the new output layer, threshold is 1. So, for the first sigmoid neuron of the new output layer, bias = -threshold = -1.

So, for the first sigmoid neuron of the new output layer, Σw.x+b = 0.00999 + (-1) = −0.99001 which is < 0 . Also, we know that, $\sigma(z) = 1 ⁄ (1 + e^{-z})$, where z = Σw.x+b. So, $\sigma(z) = 1 ⁄ (1 + e^{-z})$ will be < 0 for the first sigmoid neuron of the new output layer.

Let us assume that all the sigmoid neurons of the old output layer have this set of weights and this bias for the second and third sigmoid neurons of the new output layer. So, $\sigma(z) = 1 ⁄ (1 + e^{-z})$ will be < 0 for the second and third sigmoid neurons of the new output layer.

Now suppose, for the fourth sigmoid neuron in the old output layer, we take a weight w= 0.99. We supposed earlier that the second sigmoid neuron of the old output layer is 0.99. So, for the second sigmoid neuron of the old output layer, w.x = (0.99*0.99) = 0.9801. In our assumed case, all the other sigmoid neurons except the first sigmoid neuron of the old output layer have activation less than 0.01. Let us assume that all the other sigmoid neurons except the first sigmoid neuron of the old output layer have activation 0.001. Suppose all those other sigmoid neurons have weight 0.01. So, for all those other sigmoid neurons, w.x is (0.01*0.001) = 0.00001.

So, for the fourth sigmoid neuron in the new output layer,

Σw.x = 0.00001+0.9801 +0.00001+ 0.00001+ 0.00001+ 0.00001+ 0.00001+ 0.00001+ 0.00001+ 0.00001

or, Σw.x = 0.98019

Now, suppose for the fourth sigmoid neuron of the new output layer, threshold is 0.01. So, for the first sigmoid neuron of the new output layer, bias = -threshold = -0.01.

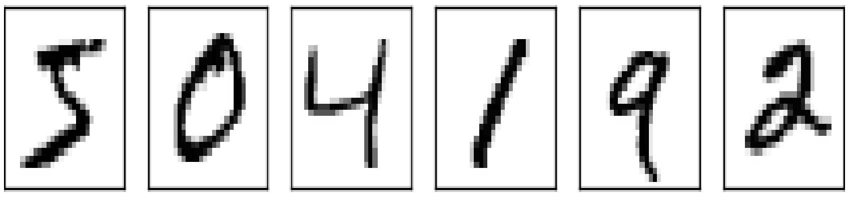
So, for the fourth sigmoid neuron of the new output layer, $\Sigma w.x+b = 0.98019 + (-0.01) = 97019$ which is > 0 . Also, we know that, $\sigma(z) = 1/(1 + e^{-z})$, where $z = \Sigma w.x+b$. So, $\sigma(z) = 1/(1 + e^{-z})$ will be > 0 for the fourth sigmoid neuron of the new output layer.

So. only the fourth sigmoid neuron of the new output layer will be activated when we give an image of 1 as an input to out input layer. Then the new output layer will look like 0001 for an image of 1 as an input to the input layer. So, our network have correctly identified the binary representation of our given input image.

Applying this technique we can figure out the sets of weights and biases to correctly generate the binary output of other inputs like 0,2,3,4,5,6,7 etc. So, we have seen that there is a way of determining the bitwise representation of a digit by adding an extra layer to the three-layer network described above.

Learning with gradient descent

Now that we have a design for our neural network, how can it learn to recognize digits? The first thing we'll need is a data set to learn from - a so-called training data set. We'll use the MNIST data set, which contains tens of thousands of scanned images of handwritten digits, together with their correct classifications. MNIST's name comes from the fact that it is a modified subset of two data sets collected by NIST, the United States' National Institute of Standards and Technology. Here's a few images from MNIST:



As you can see, these digits are, in fact, the same as those shown at the beginning of this chapter as a challenge to recognize. Of course, when testing our network we'll ask it to recognize images which aren't in the training set!

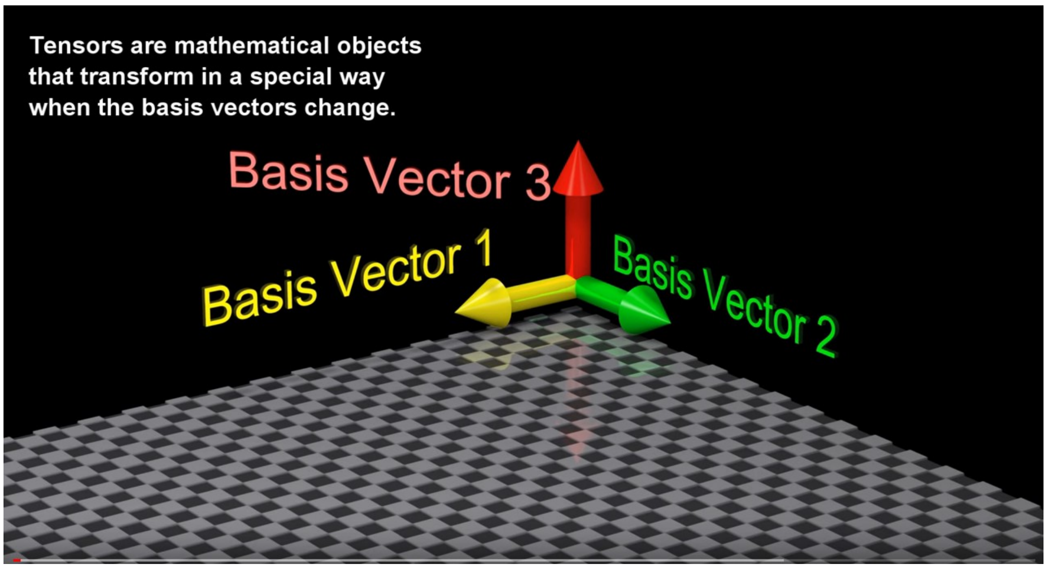
The MNIST data comes in two parts. The first part contains 60,000 images to be used as training data. These images are scanned handwriting samples from 250 people, half of whom were US Census Bureau employees, and half of whom were high school students. The images are greyscale and 28 by 28 pixels in size. The second part of the MNIST data set is 10,000 images to be used as test data. Again, these are 28 by 28 greyscale images. We'll use the test data to evaluate how well our neural network has learned to recognize digits. To make this a good test of performance, the test data was taken from a *different* set of 250 people than the original training data (albeit still a group split between Census Bureau employees and high school students). This helps give us confidence that our system can recognize digits from people whose writing it didn't see during training.
We'll use the notation x to denote a training input. It'll be convenient to regard each training input x as a $28\times28=784$-dimensional vector. Each entry in the vector represents the grey value for a single pixel in the image. We'll denote the corresponding desired output by $y=y(x)$, where y is a 10-dimensional vector. For example, if a particular training image, x, depicts a 6, then $y(x)=(0,0,0,0,0,0,1,0,0,0)^T$ is the desired output from the network. Note that T here is the transpose operation, turning a row vector into an ordinary (column) vector.

But what is a dimensional vector ?

In mathematics, a vector can be thought of as a combination of direction and magnitude. However, it can also be thought of as a coordinate. For example, a vector with magnitude 5 and an angle of about 37 degrees from the horizontal represents a point on a 2D plane. This point can also be represented with the Cartesian coordinate pair (3, 4). This pair (3, 4) is also a mathematical vector.

In programming, this name "vector" was originally used to describe any fixed-length sequence of scalar numbers. So, $y(x)=(0,0,0,0,0,0,1,0,0,0)^T$ is a vector and because the length of this vector is 10, so it is a 10-dimensional vector.

To understand vector in physics, we need to first look at this image below:

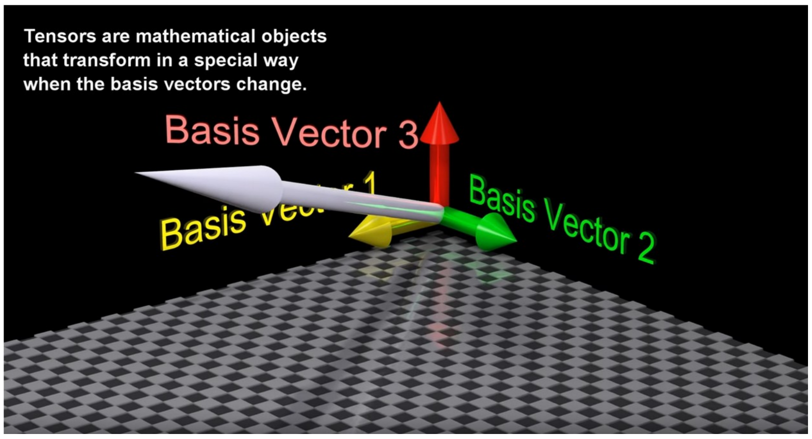

Tensors are mathematical objects
that transform in a special way
when the basis vectors change.
Basis Vector 3
Basis Vector 1
Basis Vector 2

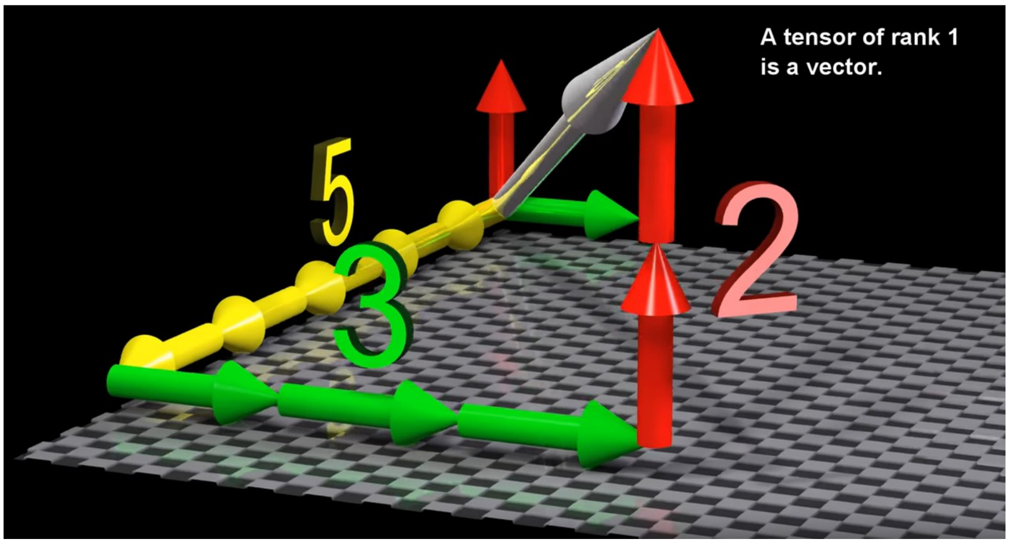
Let's forget about the term 'Tensor' in the above image. In the above image we can see that there is a fixed plane and there are three vectors over that plane named Basis Vector 1, Basis vector 2, Basis vector 3. We have a set of three vectors over a fixed plane in the above image. This is called a vector space.

A vector space over a field *F (Like the fixed plane in the above image)* is a set of vectors *(Like the set of three basis vectors in the above image)* on which you can execute two operations.

- The first operation, called **vector addition** or simply **addition.**
- The second operation, called **scalar multiplication** : takes any scalar *a* and any vector **v** and gives another vector *a***v**. (Similarly, the vector *a***v** is an element of the set *V*) .

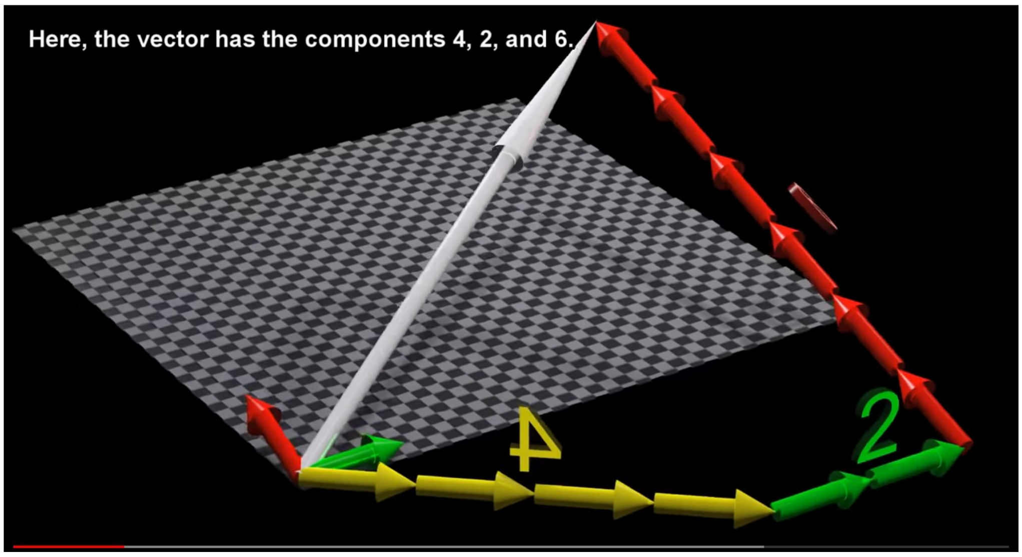
Now, lets get back to the term tensor. Tensors are mathematical objects that transform in a special way when the basis vectors change. In the below image, the white arrow is a tensor.


Tensors are mathematical objects
that transform in a special way
when the basis vectors change.
Basis Vector 3
Basis Vector 1
Basis Vector 2

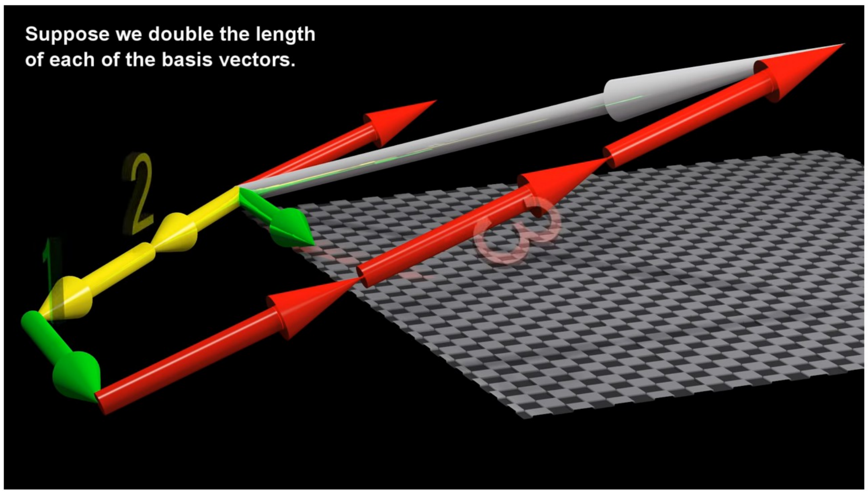


In the above image, the tensor (The white arrow) has components 5,3,2. Which means we need 5 ‘Basis vector 1’, 3 ‘Basis Vector 2’ and 2 ‘Basis Vector 3’ to create the tensor.

Lets look at another case:



Now, we can see from the above image that our tensor has components 4,2,6. If we double the length of each basis vector, we can see in the below image that components of the tensor has decreased to 2,1,3.



So increasing the length of the basis vector decreases the number of components. For this contradictory behavior, these components are called **contra-variant components of a tensor.**

When you say a vector is 1 dimensional is that a vector is a rank 1 tensor. Matrices are, on the other hand, rank 2 tensors, and scalar values are rank 0 tensors. A tensor of rank m can have dimensions d1×d2×…×dm, where di>0. What is rank of a tensor ? We will get back to it later. Let's now continue with our weights and biases.

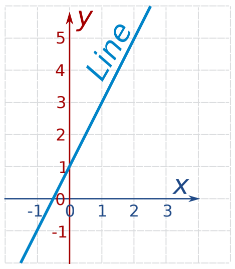
What we'd like is an algorithm which lets us find weights and biases so that the output from the network approximates y(x) for all training inputs x. To quantify how well we're achieving this goal we define a *cost function*. Sometimes referred to as a loss or objective function. We use the term cost function throughout this book, but you should note the other terminology, since it's often used in research papers and other discussions of neural networks. So, our defined cost function is:

$$C(w,b)\equiv(1/2n)\sum_x \| y(x)-a\|^2$$

Here, w denotes the collection of all weights in the network, b all the biases, n is the total number of training inputs, a is the vector of outputs from the network when x is input, and the sum is over all training inputs, x. Of course, the output a depends on x, w and b, but to keep the notation simple I haven't explicitly indicated this dependence. The notation ‖v‖ just denotes the usual length function for a vector v. We'll call C the *quadratic* cost function.

Wait, is it a good idea if we take some time to remember what is quadratic equation ?

Yes. No problem. To understand what is quadratic equation, let us first recall what is **linear equation**. When we hear the term linear equation the first thing that comes to our mind is a line. So lets draw a line in a graph.



Now we have drawn a line. But how do we represent this line with an equation. The equation that represents a line is linear equation. One type of equation that represents a line can be:

$$y=x$$

This equation can represent this line:

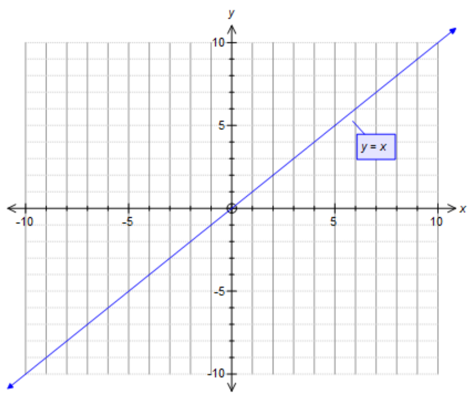


fig: Line 1

Let us call it line1. But what if the line cuts the y-axis in some point ? I mean what if the line is like this:

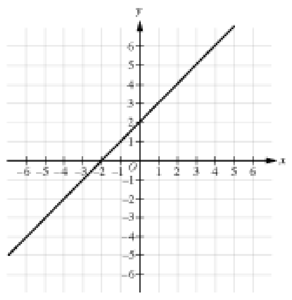


fig: Line 2

let us call it line2. You can not represent line2 with the equation y=x. So you need to modify the equation y=x into some equation that can represent both line1 and line2 shown above. In the above figure, line2 cuts y-axis at the point 2. So if you add 2 to x, then you will find y that will represent line2. So the equation that will represent line2 is:

$$y = x+2$$

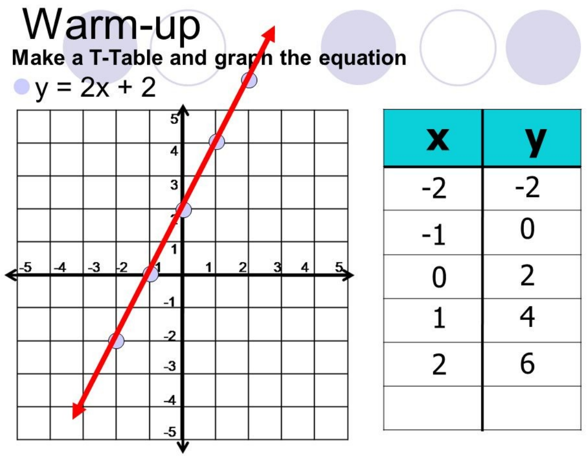
So, to represent any line like line2, we need to add a constant c to the equation y=x. So the modified equation becomes:

$$y=x+c$$

This equation can represent any line like line1 or line2. But what if line2 becomes a little bit steep. I mean what if the slope of line2 changes. Then you can not represent that line by the equation y=x+c. To represent that line you need to again modify the equation y=x+c by introducing another constant m and multiply x with that constant. So the more modified equation will be:

$$y = mx +c$$

Now this is an equation that can represent a line like:



Warm-up

Make a T-Table and graph the equation

- $y = 2x + 2$

x	y
-2	-2
-1	0
0	2
1	4
2	6

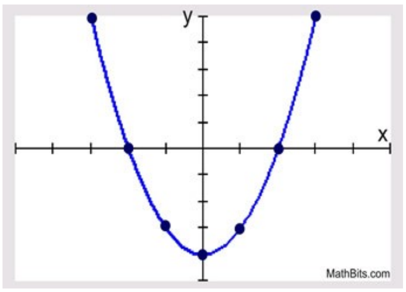
fig: Line 3

Let us call it line3. So by the equation y=mx+c, you can represent any line like line1,line2 or line3. Which means you can represent almost any line by the equation y=mx+c. So y=mx+c is our linear equation.

Is it correct to assume that all linear equations are just straight lines (with or with out slopes?)

Yes, it is correct to assume that all linear equations are just straight lines (with or with out slopes)

So we have recalled linear equation. Well, lets now recall quadratic equation. In linear equation, if we notice we can see that the line cuts x-axis at one point. But what if the line cuts x-axis in two points ? Something like this:



Well, you can not represent this line with the equation y=mx+c. You can not even draw half of this line by the equation y=mx+c. Because it is not a straight line. To represent this type of lines that cut x-axis at two different points you need a different equation. Let's try to make one.

So, looking at the above image we can see that there is one y value for two same but opposite x values. Right ? Which equation can generate this scenario ? We know this. The equation is:

$$y=x^2$$

Why? Suppose x1 = 2, x2 = -2. If we put x1 and x2 in the above equation, for both x1 and x2, we get y = 4. That is, one y value for two same but opposite x values. So, let's add x^2 to our equation y=mx+c. Then the equation will be:

$$y = x^2+mx+c$$

Now, what if the line is steep or you want to have more control over determining the steepness of the line ? Like before, just multiply x^2 by some constant a and you will have more accuracy to represent the line because you will get more control over the steepness of the line. So, modify the equation little bit to make it like this:

$$y = ax^2+mx+c$$

By this equation, you will be able to draw or represent any line like the image above that cuts x-axis in two different points. And this type of equation is called a **quadratic equation.** But remember, **by a quadratic equation you will be able to represent or draw any line like the image above where the line is symmetric**. This type of line is called a parabola. If the line is not symmetric but cuts x-axis in two different points, you can not represent or draw that line by the quadratic equation, for that you will need to make a different equation.

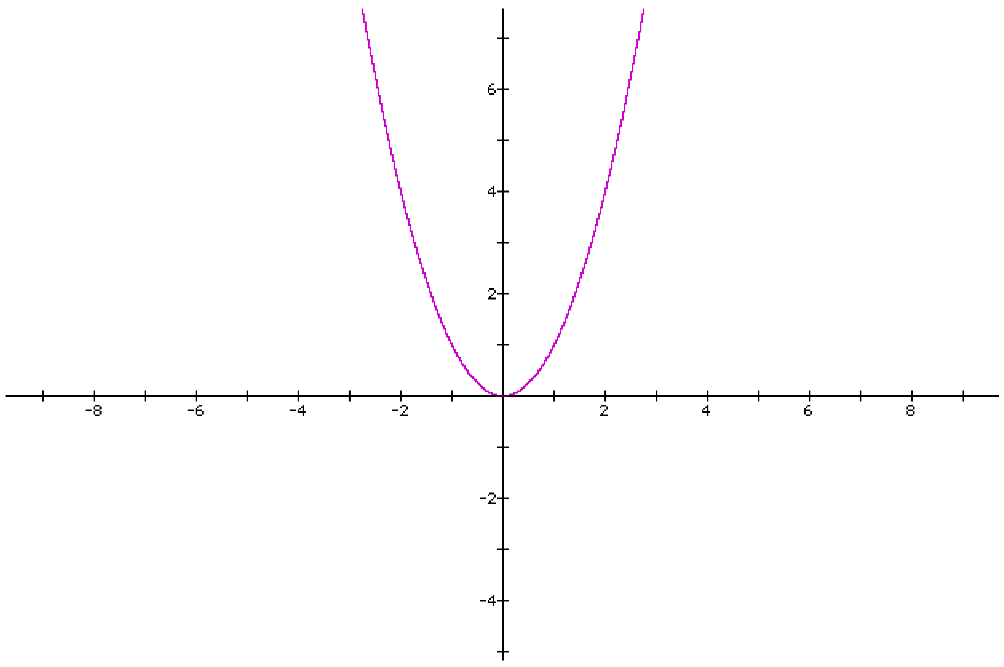
Let's see some quadratic functions:

1.

x	$y = x^2$	$y = x^2$	(x, y)
-3	$(-3)^2$	9	(-3, 9)
-2	$(-2)^2$	4	(-2, 4)
-1	$(-1)^2$	1	(-1, 1)
0	$(0)^2$	0	(0, 0)
1	$(1)^2$	1	(1, 1)
2	$(2)^2$	4	(2, 4)
3	$(3)^2$	9	(3, 9)

Plot the graph on your own graph paper and make sure that you get the same graph as depicted below.

Graph of $y = x^2$

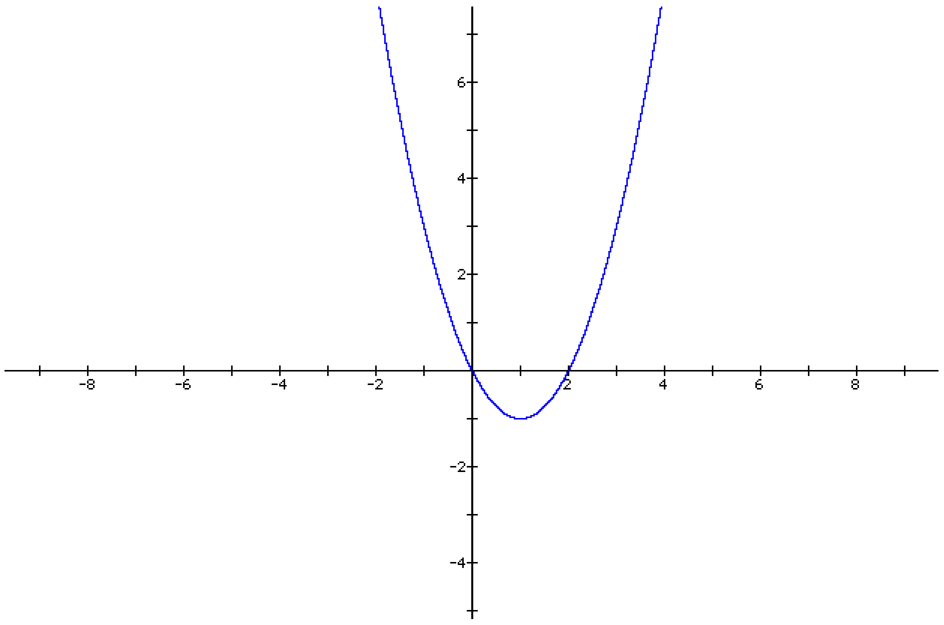


2.

x	$y = x^2$	$y = x^2 - 2x$	(x, y)
-3	$(-3)^2$-2x	15	(-3, 15)
-2	$(-2)^2$-2x	8	(-2, 8)
-1	$(-1)^2$-2x	3	(-1, 3)
0	$(0)^2$-2x	0	(0, 0)
1	$(1)^2$-2x	-1	(1, -1)
2	$(2)^2$-2x	0	(2, 0)
3	$(3)^2$-2x	3	(3, 3)

Let's graph this function.

Graph of the function $y = x^2 - 2x$



So, in our equation of the cost function which is:

$$C(w,b) \equiv (1/2n) \sum_x \| y(x) - a \|^2$$

Quadratic part is $\| y(x) - a \|^2$. So, our cost function $C(w,b)$ is a multiplication of the summation of the output of a quadratic function $\| y(x) - a \|^2$ for all x, by a constant $(1/2n)$.

Our quadratic cost function also sometimes known as the *mean squared error* or just *MSE*. Inspecting the form of the quadratic cost function, we see that $C(w,b)$ is non-negative, since every term in the sum is non-negative. Furthermore, the cost $C(w,b)$ becomes small, i.e., $C(w,b) \approx 0$, precisely when $y(x)$ is approximately equal to the output, a, for all training inputs, x. So our training algorithm has done a good job if it can find weights and biases so that $C(w,b) \approx 0$. By contrast, it's not doing so well when $C(w,b)$ is large - that would mean that $y(x)$ is not close to the output a for a large number of inputs. So the aim of our training algorithm will be to minimize the cost $C(w,b)$ as a function of the weights and biases. In other words, we want to find a set of weights and biases which make the cost as small as possible. We'll do that using an algorithm known as *gradient descent*.

Why introduce the quadratic cost? After all, aren't we primarily interested in the number of images correctly classified by the network? Why not try to maximize that number directly, rather than minimizing a proxy measure like the quadratic cost? The problem with that is that the number of images correctly classified is not a smooth function of the weights and biases in the network. For the most part, making small changes to the weights and biases won't cause any change at all in the number of training images classified correctly. That makes it difficult to figure out how to change the weights and biases to get improved performance. making small changes to the weights and biases won't cause any change at all in the number of training images classified correctly. If we instead use a smooth cost function like the quadratic cost it turns out to be easy to figure out how to make small changes in the weights and biases so as to get an improvement in the cost.

What do you mean by that ?

Earlier from cost function we came to know that, y(x) = expected output, a = actual output. making small changes to the weights and biases won't cause any change at all in the number of training images classified correctly which means the value of $\sum \| y(x) - a \|^2$ will not change that much. If the value of $\sum\| y(x) - a \|^2$ is largely positive, it will remain that if you make small changes to the weights and biases. Actually it will change very little which is negligible. Again If the value of $\sum\| y(x) - a \|^2$ is near to zero, then it means that the the number of training images classified correctly is very good and our neural network is performing very well. Now again making small changes to the weights and biases won't cause a big change in the value of $\sum\| y(x) - a \|^2$. But this time we can try to make those negligible changes in the value of $\sum\| y(x) - a \|^2$ by making small changes to the weights and biases because now those negligible changes will make our neural network even more accurate. But don't make the changes to the weights and biases big enough which will make the value of $\sum\| y(x) - a \|^2$ not close to zero. So we need cost function to figure out when we can make big changes to the weights and biases and when our neural network is performing well enough that we no longer need to make big changes to the weights and biases, when we can make small changes to the weights and biases to make our already performing well neural network perform more accurately.

That's why we focus first on minimizing the quadratic cost, and only after that will we examine the classification accuracy.

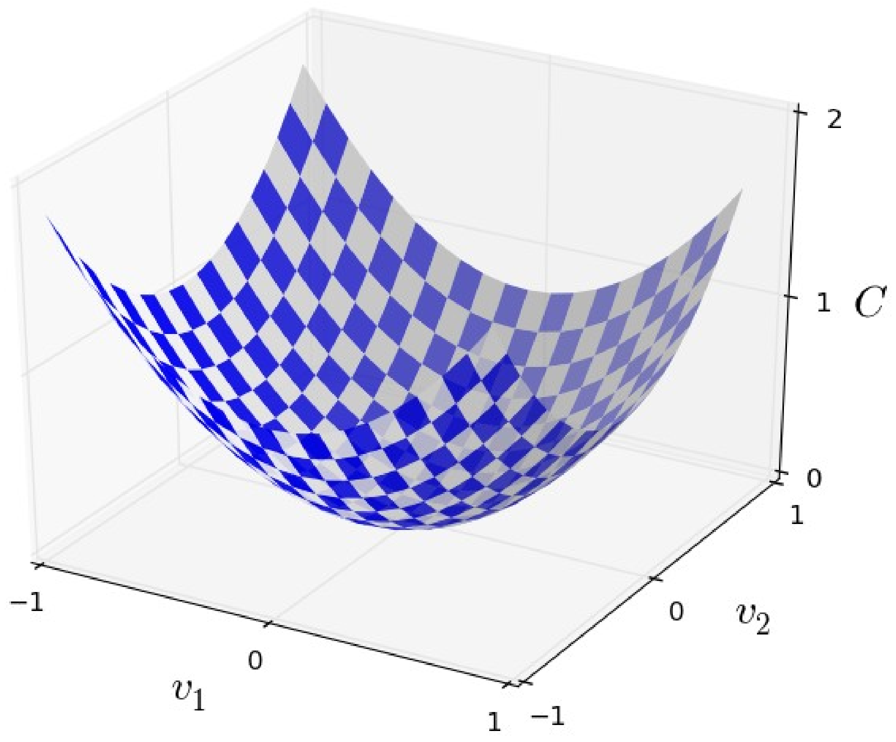
Even given that we want to use a smooth cost function, you may still wonder why we choose the quadratic function. Isn't this a rather *ad hoc* choice? Perhaps if we chose a different cost function we'd get a totally different set of minimizing weights and biases? This is a valid concern, and later we'll revisit the cost function, and make some modifications. However, the quadratic cost function works perfectly well for understanding the basics of learning in neural networks, so we'll stick with it for now. But If you are curious then here are other weight update methods that are widely used:

Neural Net Weight Update Methods

Adam	based on adaptive estimates of lower order moments
AdaGrad	Adagrad is an adaptive learning rate method
RMSProp	adaptive learning rate method, modification of Adagrad method
SGD	Stochastic gradient descent
AdaDelta	modification of Adagrad to reduce its aggressive, monotonically decreasing learning rate
Newton method	second order method, is not used in deep learning
Momentum	method that helps accelerate SGD in the relevant direction
Nesterov accelerated gradient	evaluate the gradient at next position instead of current

Recapping, our goal in training a neural network is to find weights and biases which minimize the quadratic cost function C(w,b). This is a well-posed problem, but it's got a lot of distracting structure as currently posed - the interpretation of w and b as weights and biases, the σ function lurking in the background, the choice of network architecture, MNIST, and so on. It turns out that we can understand a tremendous amount by ignoring most of that structure, and just concentrating on the minimization aspect. So for now we're going to forget all about the specific form of the cost function, the connection to neural networks, and so on. Instead, we're going to imagine that we've simply been given a function of many variables and we want to minimize that function. We're going to develop a technique called *gradient descent* which can be used to solve such minimization problems. Then we'll come back to the specific function we want to minimize for neural networks.

Okay, let's suppose we're trying to minimize some function, C(v). This function does not need to be quadratic function right now. This could be any real-valued function of many variables, v=v1,v2,…. Note that I've replaced the w and b notation by v to emphasize that this could be any function - we're not specifically thinking in the neural networks context any more. To minimize C(v) it helps to imagine C as a function of just two variables, which we'll call v1 and v2:



If you look carefully you can see that the above image looks like a 3D representation of the 2D graph of a quadratic function, or simply the above image looks like a 3D graph of some quadratic function. What we'd like is to find where C achieves its global minimum. Now, of course, for the function plotted above, we can eyeball the graph and find the minimum. In that sense, I've perhaps shown slightly *too* simple a function! A general function, C, may be a complicated function of many variables, and it won't usually be possible to just eyeball the graph to find the minimum.

Why should we emphasize on finding the minimum ?
Finding the minimum means figuring out the wights and biases for which the output of the cost function will be zero.
One way of attacking the problem is to use calculus to try to find the minimum analytically. We could compute derivatives and then try using them to find places where C is an extremum. With some luck that might work when C is a function of just one or a few variables. But it'll turn into a nightmare when we have many more variables. And for neural networks we'll often want *far* more variables - the biggest neural networks have cost functions which depend on billions of weights and biases in an extremely complicated way. Using calculus to minimize that just won't work!

(After asserting that we'll gain insight by imagining C as a function of just two variables, I've turned around twice in two paragraphs and said, "hey, but what if it's a function of many more than two variables?" Sorry about that. Please believe me when I say that it really does help to imagine C as a function of two variables. It just happens that sometimes that picture breaks down, and the last two paragraphs were dealing with such breakdowns. Good thinking about mathematics often involves juggling multiple intuitive pictures, learning when it's appropriate to use each picture, and when it's not.)

Okay, so calculus doesn't work. Fortunately, there is a beautiful analogy which suggests an algorithm which works pretty well. We start by thinking of our function as a kind of a valley. If you squint just a little at the plot above, that shouldn't be too hard. And we imagine a ball rolling down the slope of the valley. Our everyday experience tells us that the ball will eventually roll to the bottom of the valley. Perhaps we can use this idea as a way to find a minimum for the function? We'd randomly choose a starting point for an (imaginary) ball, and then simulate the motion of the ball as it rolled down to the bottom of the valley. We could do this simulation simply by computing derivatives (and perhaps some second derivatives) of C - those derivatives would tell us everything we need to know about the local "shape" of the valley, and therefore how our ball should roll.

Based on what I've just written, you might suppose that we'll be trying to write down Newton's equations of motion for the ball, considering the effects of friction and gravity, and so on. Actually, we're not going to take the ball-rolling analogy quite that seriously - we're devising an algorithm to minimize C, not developing an accurate simulation of the laws of physics! The ball's-eye view is meant to stimulate our imagination, not constrain our thinking. So rather than get into all the messy details of physics, let's simply ask ourselves: if we were declared God for a day, and could make up our own laws of physics, dictating to the ball how it should roll, what law or laws of motion could we pick that would make it so the ball always rolled to the bottom of the valley?

To make this question more precise, let's think about what happens when we move the ball a small amount Δv_1 in the v_1 direction, and a small amount Δv_2 in the v_2 direction. Calculus tells us that C changes as follows:

$$\Delta C \approx \{ (\partial C/\partial V_1)\, \Delta V_1 \} + \{ (\partial C/\partial V_2)\, \Delta V_2 \}$$

Well, I can not understand this equation, will you help me to understand this equation ?

Ok, let us recall primary school mathematics:

If you change V_1 by a small amount ∂V_1, C changes by ∂C
So, if you change V_1 by a 1 , C changes by $\partial C/\partial V_1$
So, if you change V_1 by a ΔV_1 , C changes by $(\partial C/\partial V_1)\, \Delta V_1$

Similarly,

If you change V_2 by a small amount ∂V_2, C changes by ∂C
So, if you change V_2 by a 1 , C changes by $\partial C/\partial V_2$
So, if you change V_2 by a ΔV_2 , C changes by $(\partial C/\partial V_2)\, \Delta V_2$

So, overall changes in C is measured by:

$$\{ (\partial C/\partial V_1)\, \Delta V_1 \} + \{ (\partial C/\partial V_2)\, \Delta V_2 \}$$

But why are we adding $\{ (\partial C/\partial V_1)\, \Delta V_1 \}$ and $\{ (\partial C/\partial V_2)\, \Delta V_2 \}$? Why not multiplying $\{ (\partial C/\partial V_1)\, \Delta V_1 \}$ and $\{ (\partial C/\partial V_2)\, \Delta V_2 \}$?

Let us suppose that the initial speed of a car is 0 km/h. Now suppose you put a force on the car. Now the speed of the car is 20 km/h. That means the force increases the speed by 20 km/h. How will you calculate the increase of the speed ? (0 km/h * 20 km/h) or (0 km/h + 20 km/h) ? (0 km/h * 20 km/h) is equal to 0 km/h which means that the car’s speed does not change after putting the force which is not right. (0 km/h + 20 km/h) is equal to 20 km/h which indicates the change of the speed of the car.

Now let us suppose that a car is moving at the speed of 60 km/h. Now suppose you put a force on the car. Now the speed of the car is 80 km/h. That means the force increases the speed by 20 km/h. How will you calculate the increase of the speed ? (60 km/h * 20 km/h) or (60 km/h + 20 km/h) ? (60 km/h * 20 km/h) is equal to 1200 km/h which means that the car's speed changes hugely after putting the force which is not right. (60 km/h + 20 km/h) is equal to 80 km/h which indicates the change of the speed of the car correctly.

I hope now you have understood why we did not multiply { $(\partial C/\partial V_1)\ \Delta V_1$ } and { $(\partial C/\partial V_2)\ \Delta V_2$ } . Because if we did then the equation would give us the wrong result.

We're going to find a way of choosing Δv_1 and Δv_2 so as to make ΔC negative; i.e., we'll choose them so the ball is rolling down into the valley. To figure out how to make such a choice it helps to define Δv to be the vector of changes in v, $\Delta v \equiv (\Delta v_1, \Delta v_2)^T$, where T is again the transpose operation, turning row vectors into column vectors. We'll also define the *gradient* of C to be the vector of partial derivatives, $(\partial C/\partial v_1, \partial C/\partial v_2)^T$. We denote the gradient vector by ∇C, i.e.:

$$\nabla C \equiv (\partial C/\partial v_1, \partial C/\partial v_2)^T$$

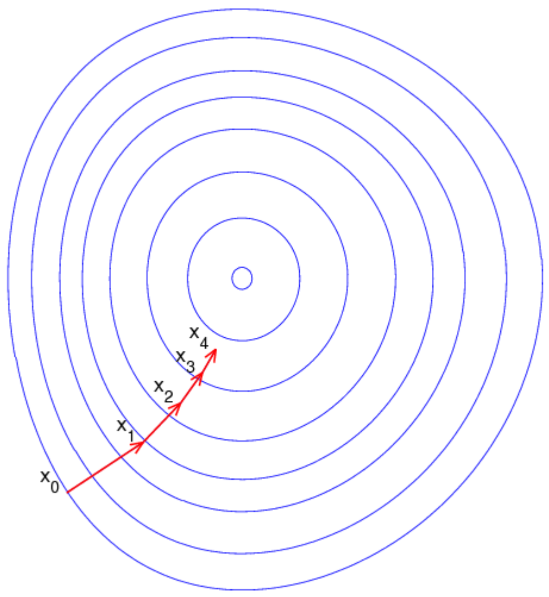
What does the term 'gradient' mean ?

The meaning of the term 'Gradient' is an increase or decrease in the magnitude of a property (e.g. temperature, pressure, or concentration) observed in passing from one point or moment to another.

"A gradient measures how much the output of a function changes if you change the inputs a little bit."—Lex Fridman (MIT)

The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops learning. Said it more mathematically, a gradient is a partial derivative with respect to its inputs.

Imagine a blindfolded man who wants to climb a hill, with the fewest steps possible. He just starts climbing the hill by taking really big steps in the steepest direction, which he can do, as long as he is not close to the top. As he comes further to the top, he will do smaller and smaller steps, since he doesn't want to overshoot it. This process can be described mathematically, using the gradient.
Just take a look at the picture below. Imagine it illustrates our hill from a top-down view, where the red arrows show the steps of our climber.Think of a gradient in this context as a vector that contains the direction of the steepest step the blindfolded man can go and also how long this step should be.



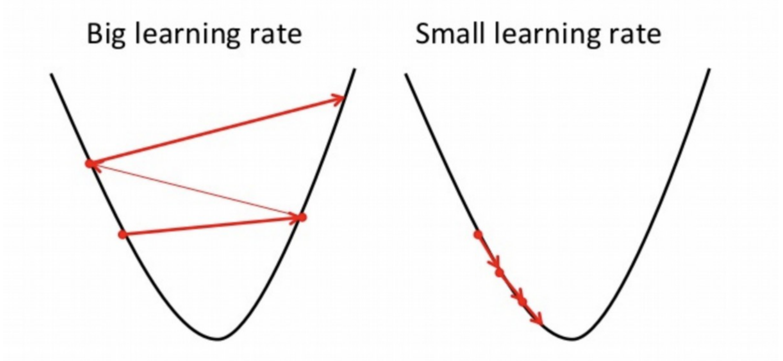
Note that the gradient ranging from X0 to X1 is much longer than the one reaching from X3 to X4. This is because the steepness/slope of the hill is less there, which determines the length of the vector. This perfectly represents the example of the hill, because the hill is getting less steep, the higher you climb it. Therefore a reduced gradient goes along with a reduced slope and a reduced step-size for the hill climber.

Gradient Descent can be thought of climbing down to the bottom of a valley, instead of climbing up a hill. This is because it is a minimization algorithm that minimizes a given function.

How big the steps are that Gradient Descent takes into the direction of the local minimum are determined by the so-called learning rate. It determines how fast or slow we will move towards the expected point.

In order for Gradient Descent to reach the local minimum, we have to set the learning rate to an appropriate value, which is neither too low nor too high.

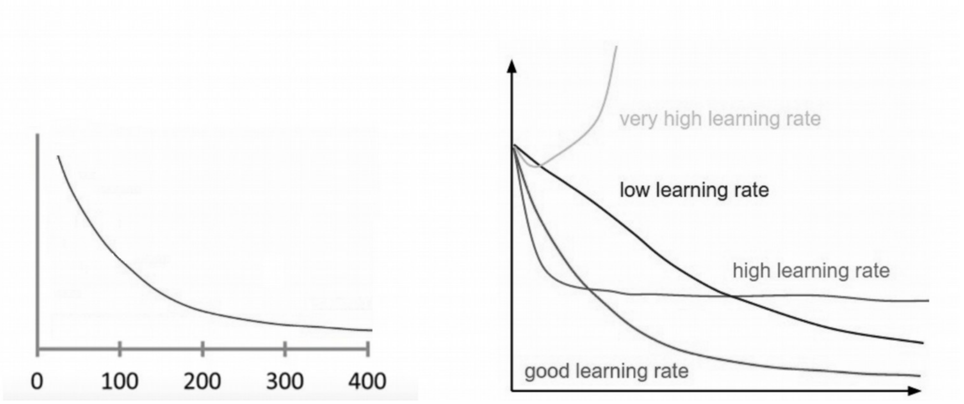
This is because if the steps it takes are too big, it maybe will not reach the local minimum because it just bounces back and forth between the convex function of gradient descent like you can see on the left side of the image below. If you set the learning rate to a very small value, gradient descent will eventually reach the local minimum but it will maybe take too much time like you can see on the right side of the image.



This is the reason why the learning rate should be neither too high nor too low. You can check if you’re learning rate is doing well by plotting the learning rate on a graph, which we will discuss in the section below.

A good way to make sure that Gradient Descent runs properly is by plotting the cost function as Gradient Descent runs. You put the number of iterations on the x-axes and the value of the cost-function at the y-axes. This enables you to see the value of your cost function after each iteration of gradient descent. This lets you easily spot how appropriate your learning rate is. You just try different values for it and plot them all together.

You can see such a plot below on the left and the image on the right shows the difference between good and bad learning rates:



If gradient descent is working properly, the cost function should decrease after every iteration. When Gradient Descent can’t decrease the cost-function anymore and remains more or less on the same level, we say it has converged. Note that the number of iterations that Gradient Descent needs to converge can sometimes vary a lot. It can take 50 iterations, 60,000 or maybe even 3 million. Therefore the number of iterations is hard to estimate in advance.

In a moment we'll rewrite the change ΔC in terms of Δv and the gradient, ∇C. Before getting to that, though, I want to clarify something that sometimes gets people hung up on the gradient. When meeting the ∇C notation for the first time, people sometimes wonder how they should think about the ∇ symbol. What, exactly, does ∇ mean? In fact, it's perfectly fine to think of ∇C as a single mathematical object - the vector defined above - which happens to be written using two symbols. In this point of view, ∇ is just a piece of notational flag-waving, telling you "hey, ∇C is a gradient vector". There are more advanced points of view where ∇ can be viewed as an independent mathematical entity in its own right (for example, as a differential operator), but we won't need such points of view.

With these definitions, the expression $\{ (\partial C/\partial V_1)\, \Delta V_1 \} + \{ (\partial C/\partial V_2)\, \Delta V_2 \}$ for ΔC can be rewritten as:

$\Delta C \approx \{ (\partial C/\partial V_1)\, \Delta V_1 \} + \{ (\partial C/\partial V_2)\, \Delta V_2 \}$
or, $\Delta C \approx (\partial C/\partial v_1, \partial C/\partial v_2)^T . (\Delta v_1, \Delta v_2)^T$
or, $\Delta C \approx \nabla C . \Delta v$

This equation helps explain why ∇C is called the gradient vector: ∇C relates changes in v to changes in C, just as we'd expect something called a gradient to do. But what's really exciting about the equation is that it lets us see how to choose Δv so as to make ΔC negative. In particular suppose we choose-

$$\Delta v = -\eta\nabla C$$

Ok, let us understand this deeply. What is Δv? It represents a value that indicates how much the position has changed. It does not represent the new position or where the new position is, instead it represents how much the position has changed from it's previous position to get to the new position. Now the question is why we are using the term 'learning rate' ? I mean we could just increase or decrease ∇C according to our need. Why multiplying ∇C by η ?

To answer this question lets look at what ' gradient' means ones again because ∇C is the gradient vector.

"A gradient measures how much the output of a function changes if you change the inputs a little bit."—Lex Fridman (MIT)

Gradient represents a value that indicates how much something has changed. It does not represent the new value of the output or where the new output is, instead it represents how much the output has changed from previous output.

What are the inputs that we are talking about to change ? Lets look at the equation of ∇C that we showed earlier.

$$\nabla C \equiv (\partial C/\partial v_1, \partial C/\partial v_2)^T$$

So, changing the inputs means changing the value of $\partial C/\partial v_1$ and $\partial C/\partial v_2$. If we change the inputs a little bit, the output will also change. And the amount of that change is called gradient.

So, suppose we have figured out that changing the inputs a little bit changes the output a little bit. So we know our gradient. But we are not satisfied by the change of our output. We want our output to be changed as much as we can to get as close as possible. What will we do ? Simple, change the inputs a little bit more. Right ? Ok, after changing the inputs a little bit more, suppose our output changes more than we have expected. Now what we will do ? Simple, again change the inputs. Thus continue changing the inputs until we get the desired change in our output. So can you spot the problem ? Just to find out how much can we change our output to get as much close as possible, we are changing our inputs every single time. Is it really necessary? We have already found the the small change in our output which is our gradient ∇C by changing our inputs a little bit. Why not just multiply the small change in our output which is our gradient ∇C by some number to find a bigger change or in other words, to get a bigger gradient which is close enough to our expected result. Thus we can find our actual change ΔV that is needed to be made in the output to get close to the result. If we can find ΔV, we can find ΔV_1 and ΔV_2. We have already knew $\partial C/\partial v_1, \partial C/\partial v_2$ when we figured out small change in output or gradient ∇C. So, we can now calculate the cost ΔC that we will need to get as much close to the result as possible from the equation:

$$\Delta C \approx \{ (\partial C/\partial V_1)\, \Delta V_1 \} + \{ (\partial C/\partial V_2)\, \Delta V_2 \}$$

Ok, so lets go back to our equation:

$$\Delta v = -\eta\nabla C$$

where η is a small, positive parameter (known as the *learning rate*). Then Equation tells us that $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$. Because $\|\nabla C\|^2 \geq 0$, this guarantees that $\Delta C \leq 0$, i.e., C will always decrease, never increase, if we change v according to the prescription in-

$$\Delta v = -\eta\nabla C$$

(Within, of course, the limits of the approximation in Equation: $\Delta C \approx \nabla C.\Delta v$). This is exactly the property we wanted! And so we'll take Equation-

$$\Delta v = -\eta \nabla C$$

to define the "law of motion" for the ball in our gradient descent algorithm. That is, we'll use Equation-

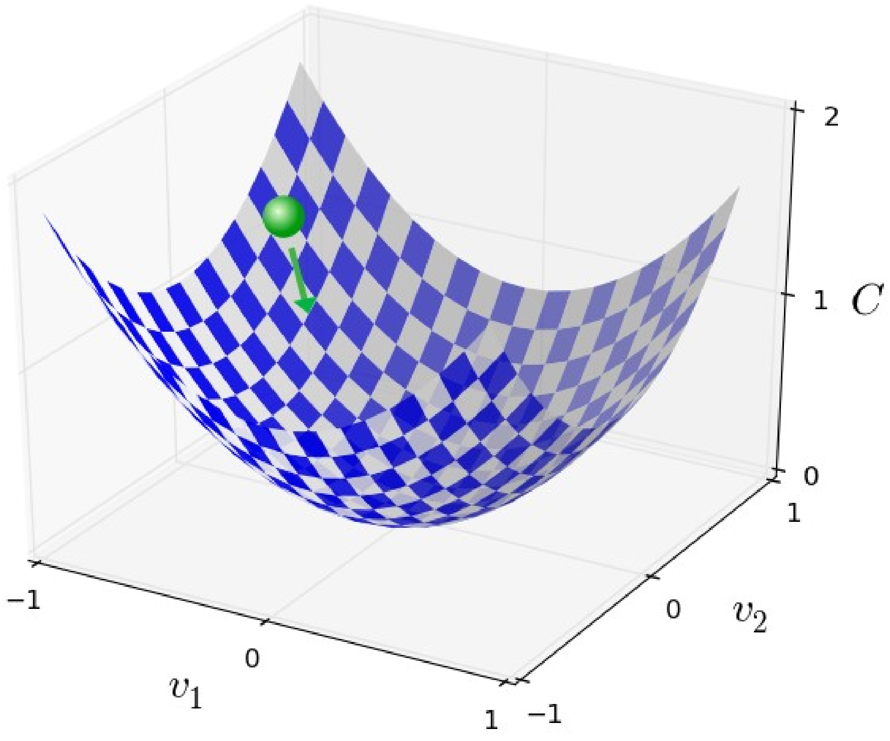
$$\Delta v = -\eta \nabla C$$

to compute a value for Δv, then move the ball's position v by that amount:

$$v \rightarrow v' = v - \eta \nabla C$$

Then we'll use this update rule again, to make another move. If we keep doing this, over and over, we'll keep decreasing C until - we hope - we reach a global minimum.

Summing up, the way the gradient descent algorithm works is to repeatedly compute the gradient ∇C, and then to move in the *opposite* direction, "falling down" the slope of the valley. We can visualize it like this:



Notice that with this rule gradient descent doesn't reproduce real physical motion. In real life a ball has momentum, and that momentum may allow it to roll across the slope, or even (momentarily) roll uphill. It's only after the effects of friction set in that the ball is guaranteed to roll down into the valley. By contrast, our rule for choosing Δv just says "go down, right now". That's still a pretty good rule for finding the minimum!

To make gradient descent work correctly, we need to choose the learning rate η to be small enough that Equation: $\Delta C \approx \nabla C.\Delta v$ is a good approximation. If we don't, we might end up with $\Delta C > 0$, which obviously would not be good! At the same time, we don't want η to be too small, since that will make the changes Δv tiny, and thus the gradient descent algorithm will work very slowly. In practical implementations, η is often varied so that Equation: $\Delta C \approx \nabla C.\Delta v$ remains a good approximation, but the algorithm isn't too slow. We'll see later how this works.