Neural Networks[*]

March 21, 2024

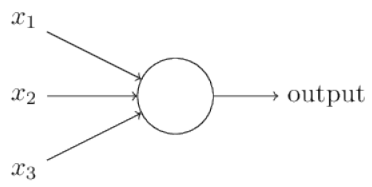
In this lecture notes, we introduce neural networks, and we use the recognization of handwritten digits as an example throughout this document.

1 Introduction

The human visual system is amazing, but recognizing handwritten digits is a difficult task for computers. Neural networks, which learn from training examples, offer a solution. This lecture notes will teach how to implement a neural network that recognizes handwritten digits with good accuracy, without human intervention. Handwriting recognition is an excellent prototype problem for learning about neural networks, and throughout the course, the lecturer will develop key ideas about neural networks, including two important types of artificial neuron (the perceptron and the sigmoid neuron), and the standard learning algorithm for neural networks, known as stochastic gradient descent. By the end of the lecture notes, students will be able to understand what deep learning is and why it matters.

2 Perceptrons

A perceptron is a type of artificial neuron that was developed in the 1950s and 1960s by the scientist Frank Rosenblatt. It takes several binary inputs and produces a single binary output based on a simple rule. Rosenblatt introduced weights to the inputs, which are 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 is less than or greater than some threshold value, which is also a parameter of the neuron. So how do perceptrons work? A perceptron takes several binary inputs, $x_1, x_2, \ldots$, and produces a single binary output:



In the example shown the perceptron has three inputs, x_1, x_2, x_3. In general it could have more or fewer inputs. Rosenblatt proposed a simple rule to compute the output. He

[*]References
- Mickael Nielsen. Neural Networks and Deep Learning

introduced *weights*, w_1,w_2, ..., 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 $\sum_j w_j x_j$ 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:

$$\text{output} = \begin{cases} 0 & \text{if} \quad \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if} \quad \sum_j w_j x_j > \text{threshold} \end{cases} \tag{1}$$

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. Suppose the weekend is coming up, and you've heard that there's going to be a cheese festival in Gouda. 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) Do your friends want to accompany you?

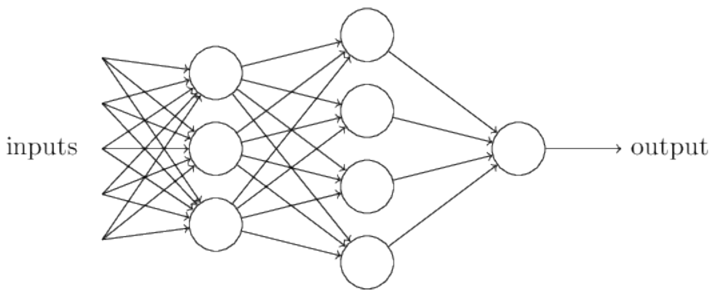
3) Is the place easy to commute?

We can represent these three factors by corresponding binary variables x_1, x_2 and x_3. For instance, we'd have $x_1 = 1$ if the weather is good, and $x_1 = 0$ if the weather is bad. Similarly, $x_2 = 1$ if your friends want to go, and $x_2 = 0$ if not. And similarly again for x_3 and public transit.

Now, suppose you deep into Dutch culture and adore cheese :), so much so that you're happy to go to the festival even if your friends are 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 $w_1 = 6$ for the weather, and $w_2 = 2$ and $w_3 = 2$ for the other conditions. The larger value of w_1 indicates that the weather matters a lot to you, much more than whether your friends, or the commute. 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 friends want 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 venue is easy to commute and your friends were 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.

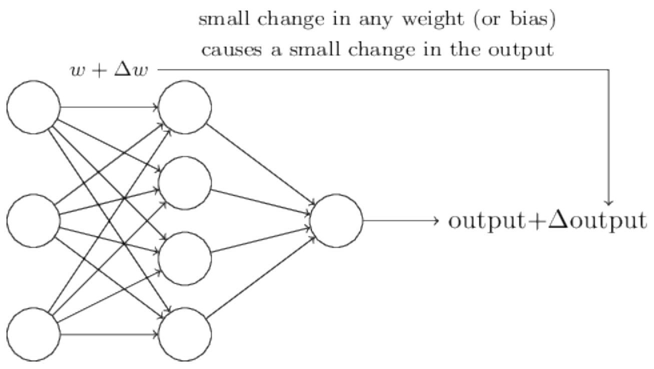
Let's simplify the way we describe perceptrons. The condition $\sum_j w_j x_j$ >threshold is cumbersome, and we can make two notational changes to simplify it. The first change is to write $\sum_j w_j x_j$ as a dot product, $w \cdot x = \sum_j w_j x_j$, where w and x are vectors whose components are 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≡ −threshold. Using the bias instead of the threshold, the perceptron rule can be rewritten:

$$\text{output} = \begin{cases} 0 & \text{if} \quad w \cdot x + b \leq 0 \\ 1 & \text{if} \quad w \cdot x + b > 0 \end{cases} \tag{2}$$

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 lecture notes we won't use the threshold, we'll always use the bias.

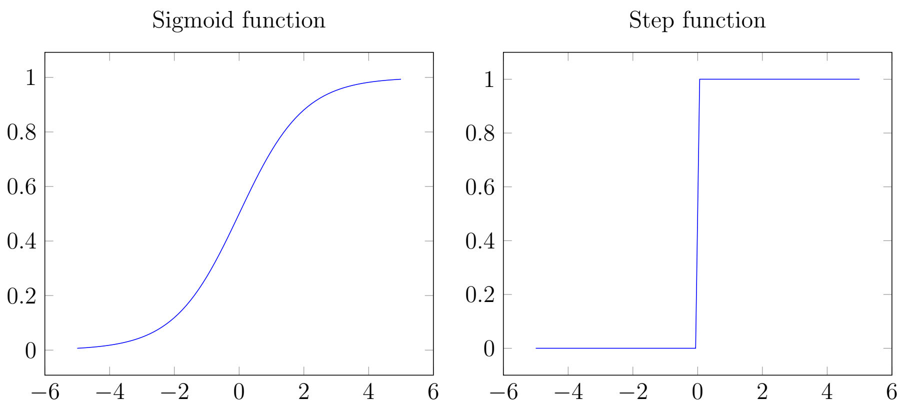
3 Sigmoid neurons

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!):

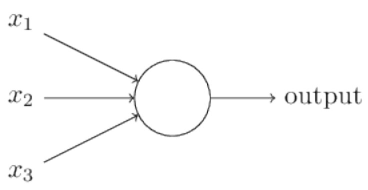

small change in any weight (or bias)
causes a small change in the output
$w + \Delta w$
output+Δoutput

We can modify the weights and biases of a neural network to get it to behave more like we want if a small change in weight or bias causes only a small change in output. However, this is not true when the network contains perceptrons. In fact, a slight change in weight or bias of any single perceptron can cause a complete flip in the output from 0 to 1, resulting in the change of the whole network's behavior in a hard-to-control manner. This makes it difficult to gradually modify the weights and biases to get the network closer to the desired behavior, and hence it's not immediately clear how to make a network of perceptrons learn.

A new type of artificial neuron called a sigmoid neuron can overcome this problem. Sigmoid neurons are similar to perceptrons, but with small changes in their weights and bias, causing only a small change in their output. The output of a sigmoid neuron is $\sigma(w \cdot x + b)$, where σ is the sigmoid function defined by $\sigma(z) = \frac{1}{1+e^{-z}}$. The output takes any value between 0 and 1 for inputs x_1, x_2, ..., with corresponding weights w_1, w_2, ..., and a bias b. The output function σ has a smoothed out version of a step function shape, which can be used to represent a continuous range of outputs.



Although the algebraic form of the sigmoid function may seem opaque and forbidding, 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. For example, when the input z to the sigmoid function is a large positive number, the output of the sigmoid neuron is approximately 1, just as it would have been for a perceptron. Similarly, when z is very negative, the output from a sigmoid neuron closely approximates a perceptron.

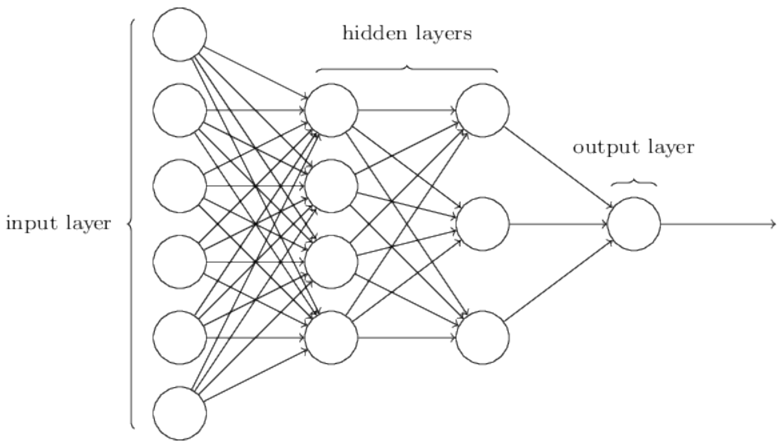


If σ had in fact been a step function, then the sigmoid neuron would be a perceptron, since the output would be 1 or 0 depending on whether $w \cdot x + b$ was positive or negative[1]. By using the actual σ function we get, as already implied above, a smoothed out perceptron. Indeed, it's the smoothness 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 Δoutput in the output from the neuron. In fact, calculus tells us that Δoutput is well approximated by

$$\Delta \text{output} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \text{output}}{\partial b} \Delta b \tag{3}$$

4 The architecture of sigmoid neural networks

In the following section, I will introduce a neural network that can classify handwritten digits effectively. To facilitate understanding, let me first explain some relevant terminology used to identify different parts of a network. Consider the network diagram below:



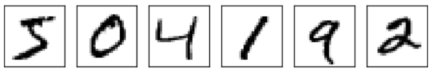
As previously mentioned, the leftmost layer in the diagram is the input layer, consisting of *input neurons*. The rightmost layer is the *output layer*, containing a single *output neuron*. The middle layers are the *hidden layer*, where the neurons are neither inputs nor outputs.

5 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

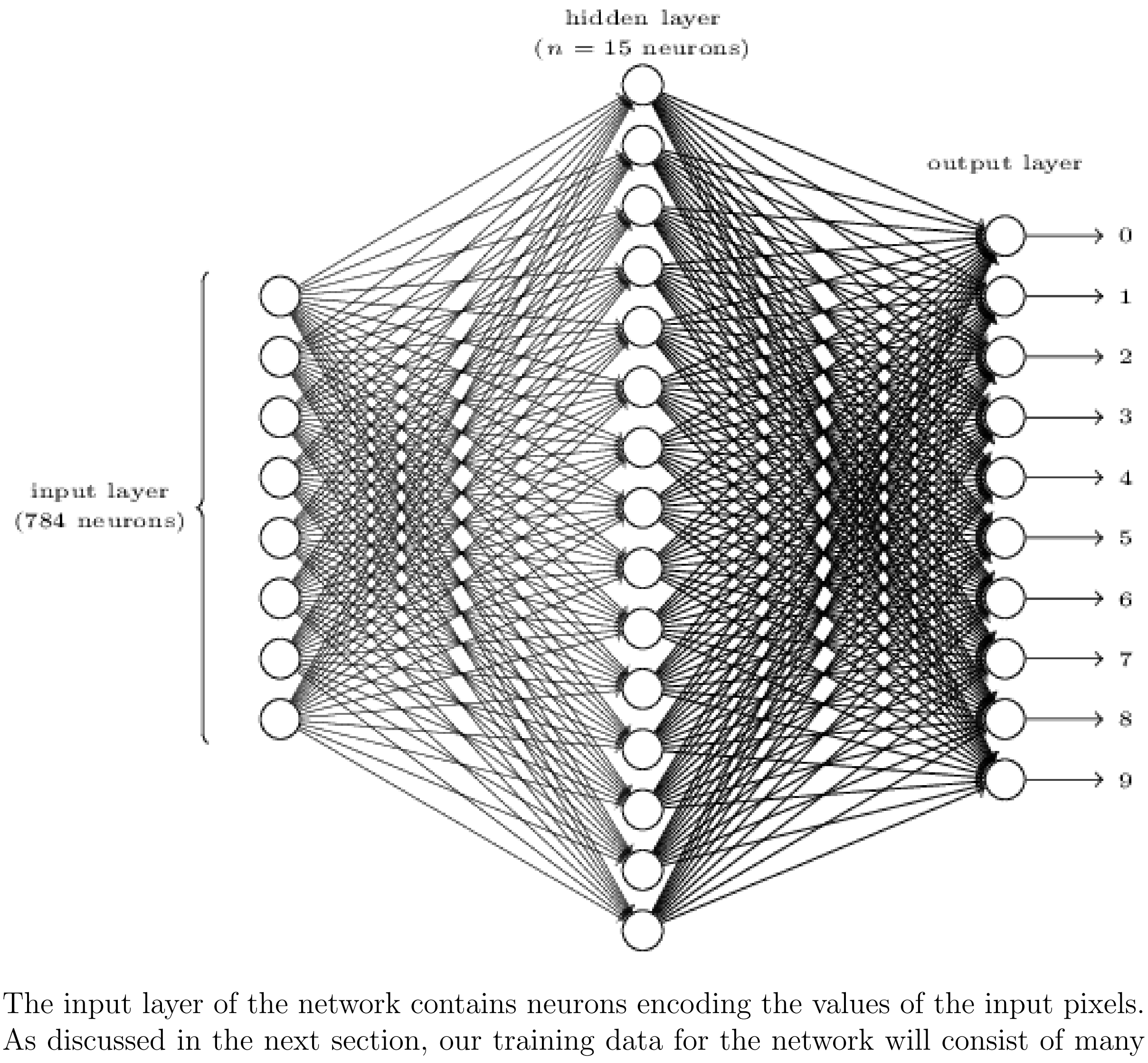
[1]Actually, when $w \cdot x + b = 0$ the perceptron outputs 0, while the step function outputs 1. So, strictly speaking, we'd need to modify the step function at that one point. But you get the idea.

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, the program then needs to classify each individual digit. So, for instance, we'd like our program to recognize that the first digit above,

is a 5. We'll focus on writing a program to solve the second problem, that is, classifying individual digits. To recognize individual digits we will use a three-layer neural network:



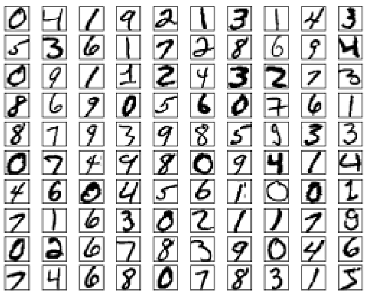
The input layer of the network contains neurons encoding the values of the input pixels. As discussed in the next section, our training data for the network will consist of many 28 by 28 pixel images of scanned handwritten digits, and so the input layer contains $784 = 28 \times 28$ neurons.

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.

6 Learning with gradient descent

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.



We'll use the notation x to denote a training input. It'll be convenient to regard each training input x as a $28 \times 28 = 784$-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.

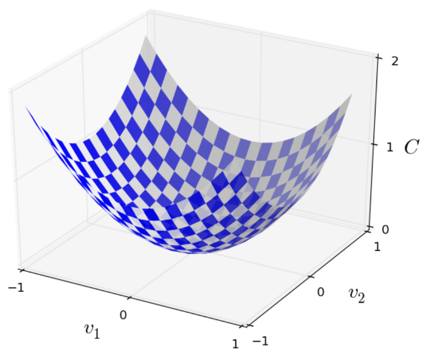
What we'd like is an algorithm which lets us find weights and biases so that the output from the network approximates $y(x)$ for all training inputs x. To quantify how well we're achieving this goal we define a cost function[2]:

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2 \tag{4}$$

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.

Our goal in training a neural network is to find weights and biases which minimize the quadratic cost function $C(w, b)$. Suppose we're trying to minimize some function, $C(v)$. This could be any real-valued function of many variables, $v = v_1, v_2, \ldots$. To minimize $C(v)$ it helps to imagine C as a function of just two variables, which we'll call v_1 and v_2:

[2]Sometimes referred to as a loss or objective function. We use the term cost function throughout this lecture notes, but you should note the other terminology, since it's often used in research papers and other discussions of neural networks.



What we'd like is to find where C achieves its global minimum. 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. 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:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2. \tag{5}$$

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, $\Delta v \equiv (\Delta v_1, \Delta v_2)^T$. We'll also define the *gradient* of C to be the vector of partial derivatives, $\left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$. We denote the gradient vector by ∇C, i.e.:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T. \tag{6}$$

In a moment we'll rewrite the change ΔC in terms of Δv and the gradient, ∇C. With these definitions, the expression (5) for ΔC can be rewritten as

$$\Delta C \approx \nabla C \cdot \Delta v \tag{7}$$

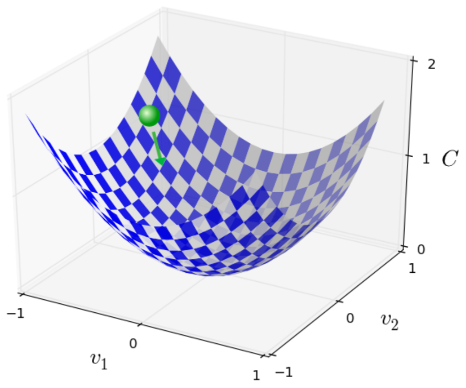
What's really exciting about the equation is that it lets us see how to choose Δv so as to make ΔC negative. In particular, suppose we choose

$$\Delta v = -\eta \nabla C, \tag{8}$$

where η is a small, positive parameter (known as the *learning rate*). Then Equation (7) 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 (8). (Within, of course, the limits of the approximation in Equation (7)). This is exactly the property we wanted! And so we'll take Equation (8) to define the "law of motion" for the ball in our gradient descent algorithm. That is, we'll use Equation (8) to compute a value for Δv, then move the ball's position v by that amount:

$$v \rightarrow v' = v - \eta \nabla C. \tag{11}$$

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:



To make gradient descent work correctly, we need to choose the learning rate η to be small enough that Equation (7) 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.

Above gradient descent when C is a function of just two variables was explained. But, in fact, everything works just as well even when C is a function of many more variables. Suppose in particular that C is a function of m variables, $v_1, \ldots, v_m$. Then the change ΔC in C produced by a small change $\Delta v = (\Delta v_1, \ldots, \Delta v_m)^T$ is

$$\Delta C \approx \nabla C \cdot \Delta v, \tag{9}$$

where the gradient ∇C is the vector

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_m} \right)^T. \tag{10}$$

Just as for the two variable case, we can choose

$$\Delta v = -\eta \nabla C, \tag{11}$$

and we’re guaranteed that our (approximate) expression (9) for ΔC will be negative. This gives us a way of following the gradient to a minimum, even when C is a function of many variables, by repeatedly applying the update rule

$$v \rightarrow v' = v - \eta \nabla C. \tag{12}$$

How can we apply gradient descent to learn in a neural network? The idea is to use gradient descent to find the weights w_k and biases b_l which minimize the cost in Equation (4). To see how this works, let’s restate the gradient descent update rule, with the weights and biases replacing the variables v_j. In other words, our “position” now has components w_k and b_l, and the gradient vector ∇C has corresponding components $\partial C/\partial w_k$

and $\partial C/\partial b_l$. Writing out the gradient descent update rule in terms of components, we have

$$w_k \rightarrow w_k' = w_k - \eta \frac{\partial C}{\partial w_k} \tag{13}$$

$$b_l \rightarrow b_l' = b_l - \eta \frac{\partial C}{\partial b_l}. \tag{14}$$

By repeatedly applying this update rule we can "roll down the hill", and hopefully find a minimum of the cost function. In other words, this is a rule which can be used to learn in a neural network.

There are a number of challenges in applying the gradient descent rule. We'll look into those in depth in later lecture notes. But for now I just want to mention one problem. To understand what the problem is, let's look back at the quadratic cost in Equation (4). Notice that this cost function has the form $C = \frac{1}{n}\sum_x C_x$, that is, it's an average over costs $C_x \equiv \frac{\|y(x)-a\|^2}{2}$ for individual training examples. In practice, to compute the gradient ∇C we need to compute the gradients ∇C_x separately for each training input, x, and then average them, $\nabla C = \frac{1}{n}\sum_x \nabla C_x$. Unfortunately, when the number of training inputs is very large this can take a long time, and learning thus occurs slowly.

An idea called *stochastic gradient descent* can be used to speed up learning. The idea is to estimate the gradient ∇C by computing ∇C_x for a small sample of randomly chosen training inputs. By averaging over this small sample it turns out that we can quickly get a good estimate of the true gradient ∇C, and this helps speed up gradient descent, and thus learning.

To make these ideas more precise, stochastic gradient descent works by randomly picking out a small number m of randomly chosen training inputs. We'll label those random training inputs $X_1, X_2, \ldots, X_m$, and refer to them as a mini-batch. Provided the sample size m is large enough we expect that the average value of the ∇C_{X_j} will be roughly equal to the average over all ∇C_x, that is,

$$\frac{\sum_{j=1}^m \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C, \tag{15}$$

where the second sum is over the entire set of training data. Swapping sides we get

$$\nabla C \approx \frac{1}{m}\sum_{j=1}^{m} \nabla C_{X_j}, \tag{16}$$

confirming that we can estimate the overall gradient by computing gradients just for the randomly chosen mini-batch.

To connect this explicitly to learning in neural networks, suppose w_k and b_l denote the weights and biases in our neural network. Then stochastic gradient descent works by picking out a randomly chosen mini-batch of training inputs, and training with those,

$$w_k \rightarrow w_k' = w_k - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial w_k} \tag{17}$$

$$b_l \rightarrow b_l' = b_l - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial b_l}, \tag{18}$$

where the sums are over all the training examples X_j in the current mini-batch. Then we pick out another randomly chosen mini-batch and train with those. And so on, until we've exhausted the training inputs, which is said to complete an *epoch* of training. At that point we start over with a new training epoch.