Notes from Neural Networks and Deep Learning

March 13, 2020

1 Using neural nets to recognize handwritten digits

1.1 The sigmoid function

The sigmoid function is defined as:

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{e^z+1} \tag{1}$$

1.2 The cost function

The cost function is defined as:

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

The sum over all the training data of the square of each difference between the correct (or desired) output and the network's output.

1.3 Gradient descent

Calculus dictates that C changes as follows:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 \tag{3}$$

We define the gradient vector as:

$$\nabla C \equiv (\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2})^T \tag{4}$$

And by defining the vector of changes to all the inputs:

$$\Delta v \equiv (\Delta v_1, \Delta v_2)^T \tag{5}$$

We can now rewrite Equation 3 as:

$$\Delta C \approx \nabla C \cdot \Delta v \tag{6}$$

Now we can define our movement of weights and biases to be in the opposite direction of the gradient of the cost function. This is the concept of the ball rolling down the hill. If we define:

$$\Delta v = -\eta \nabla C \tag{7}$$

Where η is the *learning rate*, we can rewrite Equation 6 as:

$$\Delta C = -\eta ||\nabla C||^2 \tag{8}$$

We know $||\nabla C||^2$ will always be positive, and have chosen η as a positive constant, so it is guaranteed that ΔC will always decrease, as desired.

Stochastic gradient descent is a cut down version of gradient descent, which reduces computation time by randomly choosing a *mini batch* from the training data, rather than using all of it.

- n.b. a mini batch of size 1 is equivalent to on-line learning.

1.4 Code

The code centres around the **Network** class, with the following variables:

- $\mathbf{num_{layers}}$: the number of layers the network has
- **sizes**: a list of integers, with the nth integer defining the number of nodes in the nth layer
- **biases**: the biases of the network
- **weights**: the weights of the links

For now we initialise the biases and weights to be completely random. Both biases and weights are numpy matrices, so weights[1] is a matrix storing the weights of the links between the second and third row of neurons.

Looking at that particular matrix, which we will name w, let us denote the weight of the link joining the k^{th} neuron in the second layer with the j^{th} neuron in the third layer: w_{jk}. While this notation seems to have j, and k around the wrong way, it allows for a very concise notation for the activations of the neurons in the 3rd layer:

$$a' = \sigma(wa + b) \tag{9}$$

where:

- a is the vector of activations from the second layer of neurons
- so a' is obtained by multiplying the previous layer's activations by the link weights, and adding the biases. Simple!

The **SGD** function does a lot of the work here. Each epoch it randomly shuffles the training data, then partitions it into mini batches of the specified size. For each mini-batch, it makes one gradient descent step by using the function $\text{update}_{\text{minibatch}}$, updating the network's weights and biases

The $\textbf{update}_{\textbf{minibatch}}$ function also does a lot of work, as it performs the actual update. Within this function, the **self.backrop** function does most of the actual work, as it computes the gradient of the cost function. $\textbf{update}_{\textbf{minibatch}}$ really works by computing the gradient for every training example in the mini-batch, and updatng self.weights and self.biases. The backprop function is detailed in the next chapter, for now, all that is necessary is an understanding of what it is doing.

2 How the back propagation algorithm works

2.1 Using matrices to compute the output of a neural network

In chapter 1 we introduced the notation w_{jk} leading up to equation 9. We now extend this slightly, such that w^l_{jk} refers to the weight of the link between the k^{th} neuron in the $(l-1)^{\text{th}}$ layer and the j^{th} neuron in the l^{th} layer. Similarly for biases, b^l_j is the bias of the j^{th} neuron in the l^{th} layer.

Using these notations, the activation a^l_j of the j^{th} neuron in the l^{th} layer is given by:

$$a^l_j = \sigma(\sum_k w^l_{jk} a^{l-1}_k + b^l_j) \tag{10}$$

The sum here is over all neurons, k, in the $(l-1)^{\text{th}}$ layer.

To utilise matrix form we define a **weights matrix**, w^l, for the weights connecting the neurons in the l^{th} layer to those in the $(l-1)^{\text{th}}$ layer. w^l will be defined such that the entry in the j^{th} row and the k^{th} column is w^l_{jk}.

Similarly, we define a **bias vector**, b^l, which follows naturally as the biases for each neuron in layer l. And finally, we define the **activation vector**, a^l, whose components are the activations a^l_j.

The final step towards writing Equation 10 in matrix form is vectorising a function, such as σ. This simply refers to performing the σ operation elementwise.

Equation 10 can now be written as:

$$a^l = \sigma(w^l a^{l-1} + b^l) \tag{11}$$

We will actually compute the intermediate quantity:

$$z^l = w^l a^{l-1} + b^l \tag{12}$$

It turns out that this quantity will be useful enough (in time) to be worth naming. We call z^l the *weighted input* to the neurons in layer l. Naturally, Equation 11 can be written as:

$$a^l = \sigma(z^l) \tag{13}$$

2.2 The two assumptions we need about the cost function

The goal, as discussed, of back propagation is to compute the partial derivatives $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ of the cost function C with respect to any weight or bias in the network.

As the title of this section suggests, we need to make two assumptions for back propagation to work. Before we state those assumptions, we should define a cost function. We return to the quadratic cost function defined in Equation 2:

$$C(w,b) \equiv \frac{1}{2n} \sum_{\mathrm{n}=1} ||y(x) - a^{\mathrm{L}}(x)||^2$$

We have made some small notational changes, such as the superscript L, but it means the same thing.

- $y(x)$ is the expected (or correct) outputs of the network
- $a^L(x)$ is the vector of activations of the output layer

Okay, on to the assumptions.

Assumption 1 is that the cost function is an average $C = \frac{1}{n} \sum_x C_x$ over cost functions C_x, for individul training examples, x. This is the case for the quadratic cost function, where the cost of a single training example is $C_x = \frac{1}{2}||y - a^L||^2$. This assumtion will hold true for all cost functions to be introduced.

This assumption is important, because the backpropagation algorithm will actually calculate $\frac{\partial C_x}{\partial w}$ and $\frac{\partial C_x}{\partial b}$ for a single training example. We then get $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ by averaging over training examples.

Assumption 2 is that the cost function can be written as a function of the outputs of the network. That is, for output layer activations a^L, the cost function can be expressed as:

$$costC = C(a^L) \tag{14}$$

This is true of the quadratic cost function, as the expected network outputs can be thought of as a fixed expression (i.e. not a variable because the weights and biases of the network do not change them), and as such it is a function only of a^L.

2.3 The Hadamard product

The Hadamard product ($\circ$) is like the dot product but it works for matrices of multiple dimensions rather than just vectors.

2.4 The four fundamental equations behind backpropagation

As has been previousy discussed, backpropagation is concerned with how a change in the weights and biases of the network affects the cost function. This ultimately means calculating $\frac{\partial C}{\partial w^l_{jk}}$ and $\frac{\partial C}{\partial b^l_j}$, but first we will define an intermediate quantity, δ^l_j, known as the *error* in the j^{th} neuron of the l^{th} layer.

Suppose a demon sits at the j^{th} neuron in the l^{th} layer, and makes a small change Δz^l_j to the neuron's weighted input (z^l_j), such that the neuron now outputs $\sigma(z^l_j + \Delta z^l_j)$ instead of $\sigma(z^l_j)$. This change then propagates through the network, producing a final **change** in cost of $\frac{\partial C}{\partial z^l_j}\Delta z^l_j$.

The demon is our friend, however, and is trying to make a change that will minimise the cost function. Suppose $\frac{\partial C}{\partial z^l_j}$ has a large magnitude (positive or negative), the demon would then choose a value of Δz^l_j that has the opposite sign of $\frac{\partial C}{\partial z^l_j}$. In contrast, if $\frac{\partial C}{\partial z^l_j}$ small in magnitude, the demon would assume the neuron is near optimal, as he cannot easily change the cost function by perturbing the weighted input. From this we can derive a heuristic sense in which $\frac{\partial C}{\partial z^l_j}$ is a measure of error in the neuron.

Motivated by the demon's story, we will define the error of neuron j in layer l as:

$$\delta^l_j \equiv \frac{\partial C}{\partial z^l_j} \tag{15}$$

And like we have done before, we will use δ^l to define the vector of errors in layer l. Backpropagation then gives us a way of computing δ^l in each layer, and relating those vectors to $\frac{\partial C}{\partial w^l_{jk}}$ and $\frac{\partial C}{\partial b^l_j}$.

2.4.1 Error in the output layer, δ^L

The components of δ^L are given by:

$$\delta^L_j = \frac{\partial C}{\partial a^L_j}\sigma'(z^L_j) \tag{16}$$

- $\frac{\partial C}{\partial a^L_j}$ measures how fast the cost is changing as a function of the j^{th} output activation.

- $\sigma'(z_j^L)$ measures how fast the activation function is changing at z_j^L

To rewrite Equation 16 in matrix form:

$$\delta^L = \nabla_a C \circ \sigma'(z^L) \tag{17}$$

- $\nabla_a C$ is a vector whose components are the partials $\frac{\partial C}{\partial a_j^L}$

In the case of the quadratic cost function, $C = \frac{1}{2}\sum_j (y_j - a_j)^2$:

$$\begin{aligned}\frac{\partial C}{\partial a_j^L} &= 2 \times \frac{1}{2} \times (y_j - a_j) \times -1 \\ &= (a_j - y_j) \\ \nabla_a C &= (a^L - y)\end{aligned}$$

So now we can rewrite Equation 17 as:

$$\delta^L = (a^L - y) \circ \sigma'(z^L) \tag{18}$$

2.4.2 Error δ^l in terms of error in the next layer, δ^{l+1}

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \circ \sigma'(z^l) \tag{19}$$

- $(w^{l+1})^T$ is the transpose of the weights matrix for the $l + 1^{\text{th}}$ layer.

Suppose we know the error δ^{l+1} at the $l + 1^{\text{th}}$ layer. We can think of applying the transpose matrix to this error as moving the error backward through the network, giving us a sort of measure of the error at the l^{th} layer of the network. We then take the hadamard product $\circ\sigma'(z^l)$, which moves the error back through the activation function in layer l, giving us the error in layer l, δ^l.

It follows then, that by applying Equation 17 and then repeatedly applying Equation 19 for the remaining layers in the network, that we can compute the error for any layer in the network.

2.4.3 The rate of change of cost with respect to any bias in the network

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{20}$$

That is, the error δ_j^l is *exactly* the rate of change we are after. Naturally we can write this shorthand as:

$$\frac{\partial C}{\partial b} = \delta \tag{21}$$

where it is understood that δ is being evaluated at the same neuron as b.

2.4.4 The rate of change of cost with respect to any weight in the network

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{22}$$

We already know how to compute a^{l-1}_k and δ^l_j, yay!

We can write Equation 22 in a less index-intensive way as:

$$\frac{\partial C}{\partial w} = a_{in} \delta_{out} \tag{23}$$

where it is understood that:

- a_{in} is the activation of the neuron input to the weight w, and δ_{out} is the error in the neuron output from the weight w. One interesting conclusion from this result is that weights output from low activation neurons learn more slowly than those output from higher activation neurons.

2.5 Proofs for the four fundamental equations

The proofs for the four fundamental equations are mostly derived from the chain rule.

2.5.1 Error in the output layer

Recall that:

$$\delta^L_j = \frac{\partial C}{\partial z^L_j}$$

As earlier discussed, the cost function is a function only of output activations a^L, so by using the chain rule to break this down we get:

$$\delta^L_j = \frac{\partial C}{\partial a^L_j} \frac{\partial a^L_j}{\partial z^L_j}$$

We can break the first partial down quite easily when using the quadratic cost function, but we have already done this in Section 2.4.1. The second partial is, by definition:

$$a^l = \sigma(z^l)$$
$$\frac{\partial a^L_j}{\partial z^L_j} = \sigma'(z^l)$$

Naturally this leaves us with the desired result:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$$

2.5.2 Error in layer l, in terms of error in layer $l+1$

To formulate this equation, we want to write $\delta_j^l = \frac{\partial C}{\partial z_j^l}$ in terms of $\delta_k^{l+1} = \frac{\partial C}{\partial z_k^{l+1}}$. We do this using the chain rule:

$$\begin{aligned}\delta_j^l &= \frac{\partial C}{\partial z_j^l} \\ &= \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} \\ &= \sum_k \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_j^l}\end{aligned}$$

Now to evaluate the remaining partial term, we use the definition that:

$$z_k^{l+1} = \sum_j w_{kj}^{l+1}a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1}\sigma(z_j^l) + b_k^{l+1}$$

2.5.3 The rate of change of cost with respect to any bias in the netowrk

Recalling the demon example, he was perturbing the **weighted input** to the system. The weighted input is given by:

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$$

We have defined the error as:

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$$

Now we use partials:

$$\begin{aligned}\frac{\partial C}{\partial b_j^l} &= \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} \\ &= \delta_j^l \times 1\end{aligned}$$

As required.

2.5.4 The rate of change of cost with respect to any weight in the network

Using the same approach as in Section 2.5.3, we will break down the desired product using partials.

$$\begin{aligned}\frac{\partial C}{\partial w^l_{jk}} &= \frac{\partial C}{\partial z^l_j}\frac{\partial z^l_j}{\partial w^l_{jk}} \\ &= \delta^l_j \frac{\partial z^l_j}{\partial w^l_{jk}}\end{aligned}$$

Now we break down the second partial

$$\begin{aligned}z^l_j &= \sum_k w^l_{jk} a^{l-1}_k + b^l_j \\ \frac{\partial z^l_j}{\partial w^l_{jk}} &= a^{l-1}_k\end{aligned}$$

Therefore we get, as desired:

$$\frac{\partial C}{\partial w^l_{jk}} = \delta^l_j a^{l-1}_k$$

2.6 The backpropagation algorithm

2.6.1 For an individual training example

Now that the equations are established, we can define an algorithm that conducts backpropagation.

1. **Input** x: get the activations of the first layer, a^1
2. **Feedforward**: For each $l = 2, 3, ..., L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$
3. **Output error** δ^L: Compute the vector $\delta^L = \nabla_a C \circ \sigma'(z^L)$
4. **Backpropagate the error**: For each $l = L-1, L-2, ..., 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \circ \sigma'(z^l)$
5. **Output**: The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$

It is clear now why this is called *backpropagation*. The desired products are the partial derivatives of cost with respect to weights and biases, which we can find from the layer errors, which we find by propagating backward through the network from the output layer.

2.6.2 For a mini-batch of training examples

The steps above outline the algorithm for a single training example, however in practice it is common to combine backpropagation with a learning algorithm such as stochastic gradient descent, in which the gradient is computed for a mini-batch of training examples at a time, and then a learning step is taken. The **SGD** algorithm is detailed below, for a mini-batch of m training examples.

1. **Input** a set of training examples
2. **For each training example x:** set the corresponding input activation $a^{x,1}$, and perform the following steps:
   (a) **Feedforward:** for each $l = 2, 3, ..., L$ compute $z^{x,l} = w^l a^{x,l-1} + b^l$ and $a^{x,l} = \sigma(z^{x,l})$
   (b) **Output error $\delta^{\mathbf{x},\mathbf{L}}$:** compute the vector $\delta^{x,L} = \nabla_a C_x \circ \sigma'(z^{x,L})$
   (c) **Backpropagate the error:** for each $l = L-1, L-2, ..., 2$ compute $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \circ \sigma'(z^{x,l})$
3. **Gradient descent:** For each $l = L, L-1, ..., 2$ update the weights according to the rule $w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$, and the biases according to the rule $b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$

2.7 Code

Following on from the code from the last chapter, we add the `update_mini_batch` and `backprop` methods.

The `update_mini_batch` method, as its name suggests, updates the network's weights and biases for a single mini-batch of training examples, by computing the gradient for the mini-batch. The work is mostly done by calling the `backprop` method, which computes $\frac{\partial C}{\partial w^l_{jk}}$ and $\frac{\partial C}{\partial b^l_j}$. **Note: The algorithm exploits the useful feature of Python where a negative index in a list gives that index counting from the end of the list.**

2.7.1 Modification to use matrix input

To leverage efficient linear algebra methods that are present in most programming languages, we will modify `network.py` to apply an activation matrix whose columns are mini-batches. This script is called `network_matrix.py`. It involves relativey few changes:

- Multiplying the weights matrix by the activations matrix automatically gives us a matrix of the correct number of rows, and n columns for n training examples
- The bias vector must become a bias matrix, with n columns (each a copy of the bias vector) for n training examples.
- The sigmoid function and cost derivative work without any modification as they can both be applied elementwise to both vectors and matrices
- δ^l is now a matrix, but the operations still work the same way
- **The only difference outside the `backprop_matrix` function is that we sum each row of $\text{nabla}_\mathbf{b}$ as it is returned currently.** This could be done within the $\text{backprop}_{\text{matrix}}$ function too

3 Improving the way neural networks learn

This chapter introduces some new concepts that will help our networks learn faster and be more applicable to data outside our training data, as well as a better way of initialising weights in the network, and heuristics for choosing the network's hyperparameters.

3.1 The cross-entropy cost function

When one is learning something new, being badly wrong about it tends to lead to very rapid learning. We touched on this in the last chapter, and of course we want the neurons in our network to act the same way. When they are far off producing the correct output, we would like them to rapidly correct themselves. This, however, does not always happen when using the quadratic cost function. The cost derivative, used to modify the weights and biases of a network, hinges upon the sigmoid function, which is nearly flat for a big chunk of its domain. The derivative in these areas is naturally flat, so the learning rate is slow. If the optimal weight or bias is far from the initial value, this can mean training takes a long time.

Let us consider a very basic model network: a single neuron with multiple inputs, $x_1, x_2, ...x_k$ and bias b. The output of the neuron is still $a = \sigma(z)$, where $z = \sum_j w_j x_j + b$. We define the *cross-entropy cost function* as:

$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)] \tag{24}$$

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

It is not immediately clear that this expression is even a valid cost function, let alone one that fixes the learning slowdown problem! Let us first discuss the former point.

Firstly, this cost function is non-negative. Both of the expressions in the summed term are negative by definition, as both y and a are bound to the interval $[0, 1]$. Paired with the negative sign at the beginning of the expression, this cost function must only give positive values.

Secondly, if the neuron's actual output is close to the desired output for all training inputs, x, then the cross-entropy cost will be close to zero. Suppose we have a training example with $y = 0$ and $a \approx 0$. The first term disappears as $y = 0$, and the second term does too as $\ln(1) = 0$. A similar

logic holds for $y = 1$ and $a \approx 1$. This does assume that the the desired putputs are 0 or 1, as is often the case in classification problems. With that said, at intermediate values of y (and a) between 0 and 1, the cross-entropy cost function is still minimised. This can be proven with calculus, or simply by plugging the cross-entropy cost function into a graphing calculator such as geogebra.

We will now discuss the learning slowdown problem. We substitute $a = \sigma(z)$ into Equation 24, and apply the chain rule twice.

$$\begin{aligned} C &= -\frac{1}{n}\sum_x [y \ln \sigma(z) + (1-y)\ln(1-\sigma(z))] \\ \frac{\partial C}{\partial w_j} &= \frac{\partial C}{\partial \sigma}\frac{\partial \sigma}{\partial w_j} \\ &= \frac{\partial C}{\partial \sigma}\frac{\partial \sigma}{\partial z}\frac{\partial z}{\partial w_j} \\ &= -\frac{1}{n}\sum_x [\frac{y}{\sigma(z)} + \frac{1-y}{1-\sigma(z)}](-1)\sigma'(z)x_j \\ \frac{\partial C}{\partial w_j} &= \frac{1}{n}\sum_x \frac{\sigma'(z)x_j}{\sigma(z)(1-\sigma(z))}(\sigma(z)-y) \end{aligned}$$

Using the very helpful fact that $\sigma'(z) = \sigma(z)(1-\sigma(z))$ (relatively easily proven using the quotient rule and Equation 1), we can simplify this experssion to the very neat:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x x_j(\sigma(z)-y) \tag{25}$$

This expression tells us that the cost is controlled **only** by the error in the output, where the analogous expression for the quadratic cost function also relates to $\sigma'(z)$, which caused the learning slowdown. Yay!

Similarly to the expression for w_j, the partial derivative of cost with respect to the bias is found as follows (skipping some steps, the derivation

is quite simple and can easily be computed in full on paper):

$$\begin{aligned}\frac{\partial C}{\partial b} &= \frac{\partial C}{\partial \sigma}\frac{\partial \sigma}{\partial z}\frac{\partial z}{\partial b} \\ &= -\frac{1}{n}\sum_x [\frac{\sigma(z)-y}{\sigma(z)(\sigma(z)-1)}]\sigma'(z)\end{aligned}$$

Which simplifies nicely down to:

$$\frac{\partial C}{\partial b} = \frac{1}{n}\sum_{\mathrm{x}}[\sigma(z)-y]$$

It is also worth mentioning that the learning rate (η) can be much lower for the cross-entropy cost function than for the quadratic cost function, but this doesn't really mean much.

3.1.1 Cross-entropy cost for a network of many neurons

Equation 24 is relevant to a network of one neuron, but can easily be generalised for networks of many neurons. With $y = y_1, y_2, ...y_k$ as the desired outputs of the system, and $a^L = a_1^L, a_2^L, ..., a_k^L$ as the final layer activations (actual outputs) of the system, we define cross entropy as:

$$C = -\frac{1}{n}\sum_x \sum_j [y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)] \tag{27}$$

This is the same as equation 24, just summing over the output neurons.

3.1.2 When should I use cross-entropy cost?

Cross-entropy cost is better than the quadratic cost function for almost every network, so long as **the sigmoid function is used**.

3.2 Softmax

While we will not discuss softmax layers again until chapter 6, they are an interesting aside. A softmax layer replaces the sigmoid activation function with the so-called *softmax function*. The weighted input is still calculated the same way. According to this function, the activation a_j^L is given by:

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \tag{28}$$

Where the denominator is summed over all output neurons.

The point of a softmax layer is that it makes the output of the neurons form a kind of probability distribution. That is to say, their activations sum to 1. This can be verified algebraically but can also be seen intuitively.

Softmax layers can also be used to address the learning slowdown problem. To understand this, we define the *log-likelihood* cost function. Using the usual notation where x is a training input to the network and y is the corresponding desired output, we define the log-likelihood cost as:

$$C \equiv -\ln a_y^L \tag{29}$$

Note that a_y^L is the output **only from the neuron we want to fire**. Using our MNIST classification as an example, suppose we want to classify a 7, then a_y^L will be the activation of the 7^{th} neuron. If the network is doing well and $a_7^L \approx 1$, then the cost will be low. We need not consider the other activations in the cost function, as the softmax function already does this. If $a_7^L \approx 1$, then by definition, all other activations will be quite low.

Now, if we find the partial derivatives of the log-likelihood cost with respect to individual weights and biases in the network, we will find that they, like those of the cross-entropy cost function, are only related to the difference between desired output and the output layer activations.

$$\frac{\partial C}{\partial b_j^L} = a_j^L - y_j \tag{30}$$

$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1}(a_j^L - y_j) \tag{31}$$

Note that here y refers to the vector of output activations, where above we were using it to refer to a single output activation.

3.3 Overfitting

Overfitting is a phenomenon where the network becomes very good at classifying the training data, but worse at generalising to test data or real world data. It is often referred to as *overtraining*, as it is caused by training for too long and/or on a dataset that is too small. In general the best way to avoid overfitting is to use as big a training dataset as possible, however this is not always pratical.

3.3.1 Detecting overfitting

There are some telltale signs of overfitting that we can use to help us detect it. Given our propensity for understanding data when visualised, most of them involve plotting accuracy.

- **Plotting the accuracy on the test data** - if we see a plateau of acuracy, this usually suggests the network has stopped learning. The cost on training data plotted over the same range of epochs will likely continue to decrease, a clear sign of overfitting.
- **Plotting accuracy on the training data** - Overfitting is characterised by accuracy on training data reaching (or getting very close to) 100
- **Plotting both of these together** - A network that has been overfitted to the training data will show a divergence in the previous two graphs when overfitting really kicks in.

3.3.2 An aside on the MNIST validation data

Until now we have not used the `validation_data` that the MNIST data loading function provides. The intention of this dataset is to be used to tune the network's hyper-parameters (network structure, no. of epochs, mini-batch size, etc.) without testing on the test data. That is to say, we use the validation data as test data while we tune the hyper-parameters, and only once we have finished tuning do we test on the test data. This prevents us from having a network that is biased toward the test data.

3.4 Regularization

There are ways to prevent overfitting other than just having a bigger dataset. We can always reduce the size of our network, though this reduces the power of the network as well, so we would only do this as a last resort.

Fortunately we can employ *regularization* techniques to help reduce the instance of overfitting. We will now intrduce *weight decay*, also known as *L2 regularization*. It works by adding a term to the cost function, called the regularization term. The regularized cross entropy is:

$$C = -\frac{1}{n}\sum_{xj}[y_j \ln a_j^L + (1-y)\ln(1-a_j^L)] + \frac{\lambda}{2n}\sum_w w^2 \tag{32}$$

The regularization term here is the sum of the squares of all the weights in the network, scaled by $\frac{\lambda}{2n}$, where $\lambda > 0$ is the *regularization parameter*, and n (as usual) is the number of training examples. Note here that the regularization term does not include biases.

The same thing can be done for the quadratic cost function, using exactly the same regularization term. We can use this fact to write the regularized cost function as:

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2 \tag{33}$$

where C_0 is the original cost function.

Intuitively, regularization makes the network prefer to learn small weights. Large weigts will only be allowed if they considerably reduce the cost function. The relative importance of the two terms in the cost function is varied with the regularization parameter, λ. Small λ conveys a preference for minimised cost, and large λ conveys a preference for minimised weights.

Let us now show algebraically that regularization works. As usual, we are required to compute $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ for each weight and bias in the network. We will take the partial derivatives of Equation 3.4:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w$$
$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}$$

We have already computed the intermediate terms $\frac{\partial C_0}{\partial w}$ and $\frac{\partial C_0}{\partial b}$, so we simply add $\frac{\lambda}{n} w$ to the partial derivative of all the weight terms.

The partial derivatives with respect to the biases have not changed, so the gradient descent learning rule for biases remains:

$$b \to b - \eta \frac{\partial C_0}{\partial b}$$

and the rule for weights becomes:

$$w \to w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta \lambda}{n} w \tag{34}$$
$$= (1 - \frac{\eta \lambda}{n}) w - \eta \frac{\partial C_0}{\partial w} \tag{35}$$

Exactly the same as the usual rule, except we first rescale the weight by a factor $1 - \frac{\eta \lambda}{n}$. This rescaling is referred to as /weight decay/. It appears

from the equation that this will drive the weights unstoppably toward 0, but this is not the case. If increasing the weight will reduce the cost function C_0 sufficiently, this term will be overpowered.

The learning rule for /stochastic/ gradient descent for biases remains:

$$b \to b - \frac{eta}{m} \sum_x \frac{\partial C_x}{\partial b}$$

And the equivalent for weights becomes:

$$w \to (1 - \frac{\eta\lambda}{n})w - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial w} \tag{36}$$

Where:

- m is the size of the mini-batch
- n is the size of the full training set
- The sum is over training examples x in the mini-batch

Interestingly, regularization also helps to avoid getting stuck in local minima of the cost function.

3.4.1 Why does regularization work?

The standard explanation for this (which is well explained in the online book with diagrams) involves Ockham's Razor. Suppose we have 9 data points in a mostly linear arrangement, and are trying to deduce the model of y in terms of x. We **can** perfectly fit a 9^{th} order polynomial to the data points, but it will become dominated by the x^9 term once we move far beyond the data points. A linear model, on the other hand, will not pass through all points, but logically provides a much better prediction of the model outside the range of the data points, if we assume the original data was polluted with some kind of noise. We "know" this because of Ockham's Razor.

Now let's think of this in terms of neural networks. Suppose we have a network with mostly small weights, as a regularized cost function generates. The smalless of the weights means that a few random inputs here and there (like noise) will have limited effect on the behaviour of the network. This makes it hard for the network to learn the effects of local noise in the data. It's analogous to think of this as a way of making single pieces of evidence matter less to the output of the network, while a type of evidence seen often across a network will elicit a greater response from the network.

3.4.2 Why not regularize biases?

We can regularize biases in our networks, but it often has little to no effect. This is partly because large biases do not make a neuron sensetive to its inputs in the same way as large weights. Large biases also allow neurons to saturate, which is actually desirable behaviour.

3.4.3 Other techniques for regularization

We will here discuss three other regularization techniques, though there are many, many more.

1. **L1 regularization** is similar to L2 regularization, but we add the sum of the absolute values of the weights rather than their squares.

$$C = C_0 + \frac{\lambda}{n} \sum_w |w| \tag{37}$$

While similar to L2 regularization, this will behave slightly differently. Let us look at the partial derivatives of the cost function now, to see if we can establish how differently it will behave.

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} sgn(w) \tag{38}$$

Where $sgn(w)$ is the sign of w, i.e. 1 if w is positive, and -1 if w is negative. Now we can consider the gradient descent learning rules for L1 regularization:

$$w \to w' - \frac{\eta\lambda}{n} sgn(w) - \eta \frac{\partial C_0}{\partial w} \tag{39}$$

As usual, we can replace the final term with an average over a mini-batch if we wish.

If we compare the gradient descent rules for L1 and L2 regularization, we see that L1 regularization drives the weights down by a constant amount, where L2 regularization drives them down by an amount proportional to the size of the weight. This means that when a particular weight is large, it will be driven down by L2 regularization faster than by L1 regularization; but if that weight is small it will be driven down by L1 regularization faster. This tends to concentrate the weight of the network in a small number of important connections.

We also must note that the derivative $\frac{\partial C}{\partial w}$ is not defined at $w = 0$, because the function $|w|$ has a sharp corner at $w = 0$. This is okay though, as we can simply define $sgn(0) = 0$, which will work because regularization is already trying to reduce weights, and it can't reduce a weight that is already 0.

2. **Dropout** has a very different mechanism of action to L1 and L2 regularization, in that it does not modify the cost function, but the network itself. For each mini-batch, half of the hidden neurons (selected at random) are temporarily removed. The training examples are fed through the stripped down network and then backpropagated, and the weights and biases (that have not been removed) are updated as usual. This is repeated, with a new random selection of weights and biases each time. This does mean that running the full network will result in twice the neurons activating, so we halve all weights outgoing from the hidden neurons. The mechanism by which dropout reduces overfitting is analogous to averaging the outputs from multiple networks. Actually doing that is unduly expensive, but does reduce the effect of overfitting. Empirically, dropout is very effective as a regularization tool, especially in deep neural networks.

3. **Atrificially expanding the training data** is a surprisingly effective tool for reducing overfitting. We know already that larger training datasets are less prone to overfitting than smaller ones, but collecting more data is expensive. We can, however, make small changes to our training data to make it look different on a pixel-by-pixel basis, while retaining the same desired network output. The simplest way to do this is by rotating the input image slightly. This changes the locations of the black and white pixels substantially, but is still the same image.

3.5 Weight Initialisation

In our work thus far we have initialised our networks' weights and biases with independent Gaussian random variables, normalised to have mean 0 and standard deviation 1. It turns out we can actually initialise our weights and biases wuite a bit better than this. Let us look at an example network to demonstrate the problem with our current initialisation.

Suppose we have a network with 1000 input neurons, with normalised Gaussians used to initialise the weights and biases. Let us focus on the 1000 weights connecting the input neurons and the first neuron in the first hidden

layer. Suppose also that we have an input where half of the input neurons are on and half are off. Now consider the weighted sum $z = \sum_j w_j x_j + b$ of inputs to our hidden neuron. Of course, 500 terms in this sum vanish, so we are left with 500 weight terms and a bias term. Therefore z itself is distributed as a Gaussian with mean 0 and standard deviation $\sqrt{501} \approx 22.4$. That is, z has a broad gaussian distribution, and it is highly likely that z wil be much greater than 1 or much less than -1. This means the chance of saturating our hidden neuron is alarmingly high, and as we know a saturated neuron will learn very slowly due to small changes in weights having almost no effect.

This is very similar to the problem we faced earlier with saturated output neurons, which we solved by implementing a better cost function. Unfortunately, a better choice of cost function does not help at all with saturated hidden neurons.

Considering the cause of the excessively large standard deviation in the example above, for a network with n_{in} input weights, we will initialise our weights as Gaussian random variables with mean 0 and standard deviation $\frac{1}{\sqrt{n_{in}}}$. We will still initialise our biases with mean 0 and standard deviation 1, because it really doesn't matter what biases start as. Some people initialise them all to 0.

3.6 Code

The code for our updated network, `network2.py`, is quite similar to `network.py`. We will cover the important changes here.

3.6.1 `default_weight_initializer`

This is the function that initialises the weights as discussed in Section 3.5. It is the same as the `large_weight_initializer`, except it divides the weights by the square root of the number of connections input to that neuron (i.e. the number of neurons in the previous layer).

3.6.2 `CrossEntropyCost`

The cost is now implemented as a class rather than a function. This is because different cost functions provide different δ functions, so each cost class (we also have `QuadraticCost`) has two functions within it. Two important notes about this class are:

- `np.nan_to_num()` ensures that we handle the log of numbers very close to 0 appropriately.
- `@staticmethod` tells the Python interpreter that the function that follows does not depend on the object in any way, and it is for this reason that both functions in the cost class do not take `self` as the first argument.

3.6.3 L2 Regularization

The change from L2 regularization is hard to detect, but it is there! In the 4^{th} last line of the `update_mini_batch` method, un updating the weights, we have the weight decay term.

3.7 How to choose hyper-parameters

Without an intuition for appropriate values for a neural network's hyper-parameters, it can be extremely difficult to just pull appropriate values out of a hat, so to speak.

3.7.1 Broad strategy

When using neural networks to attack a new problem, the first step is to achieve any non-trivial result, i.e. anything better than chance. This can be surprisingly difficult, especially when confronting a new kind of classification problem. There are some strategies we can adopt to help us overcome this, which mostly boil down to training faster so that we can try many different network hyper-parameters. Some techniques for this are covered below:

Remove all training examples that aren't ones or zeros, which will reduce our training and test sets to one fifth of their original size, which provides a training speedup by a factor of 5.

Strip the network down to the simplest network that will do meaningful learning. We may decide that by removing the hidden layer(s) or cutting down their size, our network can still learn in a meaningful way. This will be much faster, but of course this is inappropriate if we are trying to find the correct number of hidden layers or neurons for our network.

Increase the frequency of monitoring. Running `network2.py` on a laptop (without matrix optimisation) takes about 10 seconds per epoch. This isn't a huge issue, but when doing a lot of testing it can get very

tiresome. Monitoring every epoch means monitoring every 50000 images. We can monitor more frequently than every epoch, or we could also reduce the training set size to achieve a similar effect. Similarly, we could reduce the validation set size from 10000 to, say, 100. **Note:** If we decrease the number of training examples we should proportionally decrease λ.

It is worth mmentioning here that it can be tempting to discard these methods, on the assumption that we will get a result sooner or later, but this is not always true, and implementing these kinds of methods can save an immense amount of time.

We will now discuss some specific recommendations for setting parameters of our networks, focusing on the learning rate η, the L2 regularization parameter λ, and the mini-batch size. Many of the remarks will also apply to other hyperparameters, including those of network architecture, other regularization parameters, and some hyper-parameters we are yet to meet, such as momentum co-efficient.

3.7.2 Learning rate

As we have seen empirically so far, a value of η too low will cause very slow learning, while a value of η too high will cause no or very erratic learning. A process to determine a good value for η is:

1. Estimate a threshold value for η at which the cost on the training data immediately begins decreasing, rather than oscillating or increasing. This needs not be accurate, just as an order of magnitude. We could start at $\eta = 0.01$ and then increase to 0.1 and 1 in turn until we find a value for eta at which the cost oscillates or increases. The same applies in reverse if our initial guess is too high, and the cost oscillates or increases, we should decrease it to 0.001 and 0.0001 in turn until we find a value at which the cost decreases over the first few epochs. This procedure will give us an order of magnitude estimate for the threshold value of η.

2. Suppose we landed on $\eta = 0.1$ as our threshold value. We could then optionally bump up the value by 0.1 until we find the threshold at which the cost starts oscillating, let's say $\eta = 0.5$.

3. Of course, the actual value for η should be no greater than the threshold value, our network wouldn't learn that way. The ideal learning rate should be something like a factor of two below the threshold value, so

in our case this would be $\eta = 0.25$. In the case of the MNIST dataset, this strategy led to these exact figures for the learning rate. Over 30 eopchs, $\eta = 0.5$ works perfectly well also.

4. Holup, why are we measuring this based on the cost rather than accuracy on validation data? We will use accuracy on validation data to adjust all of the other parameters, and the choice to use cost to quantify this is really just a preference. This preference is rooted in the fact that learning rate is intended to control the step size in gradient descent, and only incidentally affects the classification accuracy of the network. The other parameters are directly intended to improve classification accuracy.

3.7.3 Use early stopping to determine the number of epochs to train for

Using early stopping eliminates the number of epochs parameter altogether, but necessarily introduces another parameter to determine when to stop. A common example, at least for MNIST, is the point at which no improvement is seen in 10 epochs. Even at this point, we could be missing future learning (if we are unlucky), but this helps to control the training time until the point at which we have come to know our network well. Once the other parameters are better set, we can relax this imposition to no improvement for, say, 20 epochs or even 50 epochs.

3.7.4 Learning rate schedule

So far we have been holding the learning rate η constant, but it is often desirable to vary the learning rate. Intuitively, we want a higher learning rate earlier on in our program when our weights are badly wrong, and later we want a lower learning rate to fie tune the weights, avoiding overshooting the local minima of the cost function.

A common way to implement this is to use a similar idea to early stopping, hold the learning rate constant until the validation accuracy gets worse. At this pint, we decrease the learning rate by, say, a factor of two or ten. We repeat this until our learning rate is a factor of 1024 or 1000 below its original value, then we terminate.

This can lead to a world of headaches, so it is usually best to start with a fixed learning rate while we get to know our network, then implement a learning rate schedule.

3.7.5 The regularization parameter

It is best to start with $\lambda = 0$ and find a good value for η first. Once this is done, start at $\lambda = 1.0$ (which is a completely arbitrary choice btw) and increase or decrease by factors of 10 as required to improve performance on validation data. Once the order of magnitude is found, λ can be fine tuned. Once that is done, return to η and make any necessary adjustments.

3.7.6 Mini-batch size

To answer the question of how to set mini-batch size, consider online learning (i.e. mini-batch size 1). A concern about online learning is that it will provide an inaccurate estimate of the gradient of the cost function. In reality this isn't all that important, so long as our gradient estimate still tends to decrease the cost. The positive aspect of online learning is that we are constantly updating our weights, so each new estimate is based on the previous, slightly improved one.

The optimal solution is, unsurprisingly, somewhere between online learning and learning with enormous mini-batches. A mini-batch size too small doesn't make good use of optimised matrix algebra libraries that speed up the backpropagation process, and one too big doesn't stop to learn often enough. Fortunately, mini-batch size is independent of other network hyper-parameters (except network architecture), so once acceptable values for the oher hyper-parameters have been found, a little bit of trial and error can be employed to optimise the mini-batch size.

3.8 Other techniques

3.8.1 Variations on stochastic gradient descent

1. **The Hessian Technique:** If we imagine our cost function as a function of the weights of the system (which it is), so $C = C(w)$ for $w = w_1, w_2, ..., w_n$, we can approximate the cost function near a point using a Taylor approximation:

$$C(w+\Delta w) = C(w)+\sum_j \frac{\partial C}{\partial w_j}\Delta w_j+\frac{1}{2}\sum_{jk}\Delta w_j\frac{\partial^2 C}{\partial w_j \partial w_k}\Delta w_k+... \quad (40)$$

Discarding any higher order terms, this can be written more compactly as:

$$C(w + \Delta w) = C(w) + \nabla C \dot{\Delta} w + \frac{1}{2} \Delta w^T H \Delta w \tag{41}$$

Where ∇C is the usual gradient vector, and H is a atrix known as the *Hessian matrix*, whose jk^{th} entry is $\frac{\partial^2 C}{\partial w_j \partial w_k}$. We can use calculus to show that the expression can be minimised by choosing

$$\Delta w = -H^{-1} \nabla C \tag{42}$$

Providing that Equation 41 is a good approximate expression for the cost function, we would expect that moving from point w to $w + \Delta w = w - H^{-1} \nabla C$ should significantly reduce the cost function. This suggests we can so something very similar to gradient descent, starting with random weights w, then updating the weights:

$$w' = w - \eta H^{-1} \nabla C \tag{43}$$

Where η is the learning rate.

There are theoretical and empirical results showing that this hessian technique converges in fewer steps than standard gradient descent, which is largely a result of incorporating second order changes in the cost function. So why aren't we using it? Despite its qualities, it is **very difficult to apply in practice**, partly due to the sheer size of the Hessian matrix. A neural network with 10^7 weights and biases will have a corresponding Hessian matrix with 10^{14} entries! There are, however, variations on gradient descent inspired by the Hessian technique, which avoid the problem of overly large matrices. One example is momentum-based gradient descent.

2. **Momentum-based gradient descent:** Thinking back to the notion of the ball rolling down the hill, it was important to be aware that gradient descent didn't behave exactly like a ball rolling down a hill. If we stick with this analogy, the advantage of the Hessian technique is its ability to capture the velocity of the ball, not just its position on the hill. Momentum- based gradient descent emulates this by adding an element of velocity to the parameters we're trying to optimise, as well as an element of friction, giving the ball a kind of momentum. The gradient acts to change the velocity, not (directly) the position,

and the friction gradually reduces the velocity.

For a more precise definition, we introduce the variables $v = v_1, v_2, ..., v_n$, one for each w_j variable. then we replace the gradient descent update rule $w \to w' = w - \eta \nabla C$ with:

$$v \to v' = \mu v - \eta \nabla C \tag{44}$$

$$w \to w' = w + v' \tag{45}$$

Where μ is a hyper-parameter which controls the amount of damping, or friction, in the system.

To build up an understanding of how this works, imagine the case where $\mu = 1$, which corresponds to no friction. The "force" term ∇C is modifying the velocities, and the velocities are controlling the rate of change of the weights. We build up the velocity by repeatedly adding gradient terms to it, which means if the gradient is roughly the same through several rounds of learning, we could build up quite a considerable velocity.

This enables us the momentum technique to work considerably faster than vanilla gradient descent, but what happens when we get to the bottom? With all that velocity we could easily overshoot. With $\mu = 0$ (maximum friction), the equations reduce to vanilla gradient descent. In practice, a value between 1 and 0 will have the optimal behaviour of building velocity while minimising overshoot.

The name, incidentally, of the hyper-parameter μ, is the poorly chosen *momentum coefficient*. Poorly chosen because it much more closel affects friction than momentum, but that's the name.

3.8.2 Other models of artificial neuron

In principle, a network built from sigmoid neurons can compute any function. In practice, however, networks built from different neurons can outperform sigmoid neuron networks. Let us look at some other neuron models in use today.

1. **tanh neuron:** The tanh neuron (pronounced "tanch") replaces the sigmoid function with the hyperbolic tangent function. It still takes in

the same input, $wx + b$, and looks very similar to the sigmoid function both graphically and algebraically. The tanh function is defined as:

$$\tanh(z) \equiv \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{46}$$

The tanh function is shaped very similarly to the sigmoid function, with the main difference being that its range is $[-1, 1]$ rather than $[0, 1]$. This means that the outputs of (and potentially inputs to) the system may have to be normalised differently to those with sigmoid neurons.

2. **Rectified linear neuron:** The output of a rectified linear neuron is given by:

$$\max(0, wx + b) \tag{47}$$

Plotting the activation a of a rectified linear neuron as a function of Z, it looks like the function $a = Z$, but only in the domain $[0, \infty)$, and is not defined in the domain $(-\infty, 0)$.

Clearly this is quite different to the sigmoid and tanh activation functions, but it can still be used to compute any function, and can still be trained using backpropagation and gradient descent. The primary advantage of the rectified linear neuron is that it doesn't saturate, but at the same time, any negative input to a rectified linear neuron won't induce any learning at all.

Both tanh neurons and rectified linear neurons suffer from a lack of research into the most appropriate circumstances in which to use them, and the same goes for all types of neuron. Generally speaking, sigmoid neurons will perform just fine, but it is worth knowing about the different alternatives that exist for developing our networks with.

4 A visual proof that neural nets can compute any functon

See the online book, this is all very straightforward. The takeaway is that a network with a single hidden layer can compute any continuous function to an arbitrary degree of accuracy.

5 Why are deep neural networks hard to train?

There is an intuitive sense in which deep networks should be able to learn better than shallow ones. The first layer can detect basic patterns, like edges and corners and such, with the second detecting compounds of these patterns and so on. Along with this are theoretical results showing that deep neural networks are intrinsically more powerful than shallow ones. Unfortunately, there are often situations where our network won't train as effectively as we hope, manifesting particularly in different layers training at different speeds. This is often due to our gradient based learning techniques.

5.1 The vanishing gradient problem

The vanishing gradient problem is the phenomenon where the last layer in a network trains the fastest, and the first layer in the network trains the slowest, with intermediate layers following in a linear fashion. The gradient tends to get smaller as we move backward through the hidden layers, which causes this learning slowdown.

5.2 The cause of the vanishing gradient problem

Consider a network with only one neuron in each layer, and three hidden layers. Using our regular nomenclature, we denote the weight of the link between the input neuron and the first hidden neuron w_1, with its bias b_1, and so on throughout the network. We will study the gradient $\frac{\partial C}{\partial b_1}$ associated with the first hidden neuron.

Suppose we make a small change Δb_1 in the bias b_1. That will trigger a cascade of changes in the rest of the network:

- First it will cause a change Δa_1 in the activation of the first hidden neuron
- Then it will cause a change Δz_2 in the weighted input to the second hidden neuron
- This will cause a change Δa_2 in the activation of the second hidden neuron
- etc.
- Finally, this will cause a change ΔC in the cost at the output.

We can say:

$$\frac{\partial C}{\partial b_1} \approx \frac{\Delta C}{\Delta b_1} \tag{48}$$

Which suggests that we can find an expression for $\frac{\partial C}{\partial b_1}$ by carefully tracking the effect of each step in this cascade. Let us start by considering the first of the above dot points. We have $a_1 = \sigma(z_1) = \sigma(w_1 a_0 + b_1)$, so

$$\Delta a_1 \approx \frac{\partial \sigma(w_1 a_0 + b_1)}{\partial b_1} \Delta b_1 \tag{49}$$

$$= \sigma'(z_1) \Delta b_1 \tag{50}$$

This change Δa_1 then causes a change in the weighted input $z_2 = w_2 a_1 + b_2$ to the second hidden neuron:

$$\Delta z_2 \approx \frac{\partial z_2}{\partial a_1} \Delta a_1 \tag{51}$$

$$= w_2 \Delta a_1 \tag{52}$$

Combining the terms from the above two expressions, the change Δb_1 in b_1 causes z_2 to change as:

$$\Delta z_2 \approx \sigma'(z_1) w_2 \Delta b_1 \tag{53}$$

We can continue in this fashion, tracking the way changes propagate through the network. At each neuron we pick up a $\sigma'(z_j)$ term, and through each weight we pick up a w_j term. We also have the term $\frac{\partial C}{\partial a_4}$ relating the cost to the final neuron's activation at the end of the expression, because the cost is a function of the final activation. This leaves us with:

$$\Delta C \approx \sigma'(z_1) w_2 \sigma'(z_2) w_3 \sigma'(z_3) w_4 \sigma'(z_4) \frac{\partial C}{\partial a_4} \Delta b_1 \tag{54}$$

Dividing by Δb_1 and invoking Equation 48, we get:

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) w_2 \sigma'(z_2) w_3 \sigma'(z_3) w_4 \sigma'(z_4) \frac{\partial C}{\partial a_4} \tag{55}$$

5.2.1 Why the vanishing gradient problem occurs

With the exception of the final term, Equation 55 is a product of the terms $w_j \sigma'(z_j)$. Looking at this term's components:

- $\sigma'(z)$ is a normal (ish) distribution, which reaches a maximum at $z = 0$ of $\sigma'(0) = \frac{1}{4}$.

- With our new way of initialising weights and biases, weights are initialised as gaussian variables with mean 0 and standard deviation 1, so the weights will usually satisfy $|w| < 1$

Combining these two observations, it is clear that $w_j\sigma'(z_j)$ will usually satisfy $|w_j\sigma'(z_j)| < \frac{1}{4}$. When we take the product of many such terms, the product will tend to exponentially decrease.

For comparison, we will consider the equivalent of Equation 55 for b_3. We haven't explicitly calculated this expression, but we can just remove terms from 55.

$$\frac{\partial C}{\partial b_3} = \sigma'(z_3)w_4\sigma'(z_4)\frac{\partial C}{\partial a_4} \tag{56}$$

These two equations are very similar, but 56 has two fewer terms of the form $w_j\sigma'(z_j)$, and so the gradient $\frac{\partial C}{\partial b_1}$ will be a factor of 16 (or more) smaller than $\frac{\partial C}{\partial b_3}$. **This is essentially the origin of the vanishing gradient problem.**

This whole argument hingeson the fact that $|w_j\sigma'(z_j)| < \frac{1}{4}$, but if the weights grow during training (or are differently initialised), to the point that $|w_j\sigma'(z_j)| > 1$, then we will no longer have a vanishing gradient, rather the gradient will exlplode! This is called, of course, the *exploding gradient problem.*

6 Deep learning

6.1 Introducing convolutional networks

In our earlier networks, every neuron in layer n is connected to every neuron in layer $n+1$ and every neuron in layer $n-1$. At the input layer, this architecture treats pixels that are far apart the same as it treats adjacent pixels. In reality, it is much more likely that the relationship between adjacent pixels is more important than that of pixels on opposite ends of the image, when we are trying to classify images. Convolutional neural networks use a special architecture which is much better adapted to classifying images, and as such is the most common form of network used in image recognition.

Convolutional neural networks use three basic ideas: *local receptive fields*, *shared weights*, and *pooling.*

6.1.1 Local receptive fields

Previously we have visualised the input neurons in a straight vertical line, but let us now view them as a 28×28 grid. Rather than each neuron in the first hidden layer being connected to all of the input neurons, we will connect a small region, say a 5×5 square in the top left corner, corresponding to 25 input neurons, to the first neuron in the first hidden layer. This region is called the *local receptive field*, and we slide it one neuron across and connect all of the input neurons in our new region to the second neuron in the first hidden layer, and so on sliding across until we reach the other side of the image, then jump one input neuron down and repeat, until we have connected all local receptive fields to a neuron in the second hidden layer. This exact architecture (5×5 region jumping one neuron at a time) will lead to a hidden layer of size 24×24 The size of the region and the number of neurons to jump by can both be varied if desired.

6.1.2 Shared weights and biases

Naturally, each of the 24×24 neurons in the first hidden layer will have a bias and a 5×5 array of weights connecting to it. Unlike previous networks, however, we are going to use **the same** bias and array of weights for each neuron in the first hidden layer. This means that for the j, k^{th} hidden neuron, the output is:

$$\sigma(b + \sum_{\mathrm{l}=0}^{4} \sum_{\mathrm{m}=0}^{4} w_{\mathrm{l,m}} a_{\mathrm{j+l,k+m}}) \tag{57}$$

This means that all the neurons in the first hidden layer detect exactly the same feature, just at different locations across the image. For this reason, we somtimes call the map from the input layer to the hidden layer a *feature map*. We call the weights defining the feature map the *shared weights*, and the bias defining the feature map the *shared bias*. The shared weights and bias are often said to define a kernel, or filter.

Our hidden layer detects a feature in the image, but we will often want to detect many features, so most convolutional networks will have many hidden layers in paralel, all connected to the input layer. Because of the shared weights and biases, we have only 26 parameters per feature map, so having many in paralel isn't a big problem at all. By comparison, our original network has 784×30 weights and 30 biases, a total of 23550 parameters! Of course the direct comparison of both networks isn't really valid, but it's

good for framing the difference mentally.

The name *convolutional* comes from the fact that Equation 57 is sometimes known as a convolution. More concisely, people sometimes write that operation as $a^1 = \sigma(b + w * a^0)$, where a^1 denotes the set of output activations from a feature map, a^0 is the set of input activations, and $*$ is called the convolution operation.

6.1.3 Pooling layers

In addition to the convolutional layers, convolutional networks contain *pooling layers*, which are usually used immediately after convolutional layers. They exist to simplify the information in the output of the convolutional layer by condensing a region (say, 2×2) into a single neuron.

One common procedure for pooling is known as *max-pooling*, where the pooling unit simply outputs the maximum activation in the 2×2 input region. We apply *max-pooling* to each feature map separately, so there is a max-pooling layer for each one. We can think of it of a way of asking our network if there is a feature in the rough region, throwing away exact positional information but drastically cutting the number of parameters needed in later layers.

Max pooling isn't the only technique for pooling. Another common approach used is called *L2 pooling*, where we take the square root of the sum of the squares of the activations in the 2×2 region.

6.1.4 Putting it all together

We can now complete the construction of our convolutional neural network. We add to the end of our existing structure 10 output neurons, corresponding to the 10 possible values of a MNIST digit.

6.2 Convolutional networks in practice

We will now be itroducing `network3.py`, which we will use to build a convolutional neural network. Our earlier programs constructed neural networks from first principles, but this time we will use a library known as `Theano`. `Theano` makes it easy to implement backpropagation for convolutional neural networks, as it automatically computes all of the mappings involved, and

does so quite a bit faster than our easy to read code. It also allows us to use a GPU, if one is available.