Designing a Neural Network

Joe Tilley

September 15, 2017

Abstract

This paper is based on Michael Nielsen's online book, "Neural Networks and Deep Learning" [1]. I will synthesise Nielsen's book, approaching from a more mathematical angle and using original codes and examples.

1 An Introduction to a basic Neural Network

The aim of a neural network is to act as a complicated function, taking many inputs and producing a variety of outputs. Nielsen uses the example of a neural network understanding handwritten digits, taking inputs of the darkness of each pixel in the picture of the digit and producing an output of the probability of it being each digit.

1.1 Setting up the Network

The network is made up of perceptrons. A *perceptron* a function of the form

$$p(x_1, x_2, ..., x_n) = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

where w_i, b are the *weights* and *bias*, chosen by the user. For notational purposes, we let $\mathbf{x}, \mathbf{w}$ be the column vectors of inputs and weights respectively. Hence,

$$p(\mathbf{x}) = f\left(\mathbf{w} \cdot \mathbf{x} + b\right)$$

The function f, called the *activation function*, can be chosen, but in this section, we will let f by the sigmoid function

$$\sigma(x) = \frac{1}{1 - e^{-x}}$$

We create a network of perceptron layers, each layer taking every input from the previous later and acting as an input to the next (i.e. given two layers next to each other, we form a complete bipartite graph). If the function f is continuous, then a small change in any single w_i or b will result in a small change in the output, since the network is a composition of continuous functions.

We can encode this system in terms of matrices. Suppose we are given layer ℓ with n perceptrons and layer $\ell + 1$ with m perceptrons. The n perceptrons will have an associated output $x_1, ..., x_n$, called the *activation output from layer* ℓ, and each of the n perceptrons will have m associated weights, perceptron $1 \leq i \leq n$ in level ℓ having weights $w_{1i}, ..., w_{mi}$. Then, the column vector of activation outputs from level $\ell + 1$ is given by

$$\mathbf{x}_{\ell+1} = \sigma\left(\mathbf{w}_{\ell+1} \cdot \mathbf{x}_\ell + \mathbf{b}_{\ell+1}\right)$$

where taking σ of a column vector is interpreted as taking σ of each of the values. Here, $\mathbf{x}_\ell$ is the column vector of n activation outputs from level ℓ, $\mathbf{b}_{\ell+1}$ is the column vector of biases for level $\ell+1$, and the weights

$\mathbf{w}_{\ell+1} = \left(w_{ij}^{(\ell+1)}\right)$ is an $m \times n$ matrix where $w_{ij}^{(\ell+1)}$ is the weight between perceptron j in level ℓ to perceptron i in level $\ell + 1$.

Based on Nielsen's code, I have written a class which builds the network with random, normally distributed weights and biases. Moreover, the function feedforward is shown, which will take the initial input vector $\mathbf{x}_1$ and compute the activation output vector $\mathbf{x}_\ell$ (where ℓ is the number of layers) given predetermined weights and biases. We compute this value through, what appears to be, quite a contrived value called z, which we also store and output by the function. Its importance will only be shown later in the section 'Backpropogation', so feel free to ignore it for now.

```
class Network(object):

    def __init__(self, sizes):
        self.sizes = sizes
        self.L = len(sizes)
        self.biases = [np.random.normal(0, 1, (sizes[n+1], 1)) for n in range(len(
            sizes)-1)]
        self.weights = [np.random.normal(0, 1, (sizes[n+1], sizes[n])) for n in
            range(len(sizes)-1)]

    def feedforward(self, input_vector):
        ''' Runs the input vector through the network'''
        z = [np.add(self.weights[0].dot(input_vector), self.biases[0])]
        for n in range(1,self.L-1):
            z.append(np.add(self.weights[n].dot(sigmoid(z[-1])), self.biases[n]))
        output_vector = sigmoid(z[-1])
        return [z,output_vector]

def sigmoid(x):
    return 1/(1+np.exp(-x))
```

1.2 Training the Network

The aim of our neural network is for it to learn from its previous attempts. We shall have some training data, T, from which it shall learn. T will contain tuples $(\mathbf{x}, \mathbf{y_x})$ where $\mathbf{x}$ is the input vector and $\mathbf{y_x}$ the true output vector associated to $\mathbf{x}$ (which we wish our network to produce). Our aim is to minimise the function

$$C(\mathbf{w}, \mathbf{b}) = \frac{1}{2n} \sum_{\mathbf{x} \in T} \|N(\mathbf{w}, \mathbf{b}, \mathbf{x}) - \mathbf{y_x}\|^2$$

where $n = |T|$ is the number of training tuples, and $N(\mathbf{w}, \mathbf{b}, \mathbf{x})$ is the output of the network given those weights, biases and input vector, i.e. $\mathbf{x}_\ell$ the activation output from the final layer ℓ. We minimise this function by our choice of $\mathbf{w}$ and $\mathbf{b}$. Other cost functions can be chosen, and we will explore this later.

The function is too complicated to find the global derivative of, so we use a method of steepest gradient descent. Suppose we are given weights and biases $(\mathbf{w}_n, \mathbf{b}_n)$, and we want to iterate to find a new set of weights and biases such that

$$C(\mathbf{w}_{k+1}, \mathbf{b}_{k+1}) \leq C(\mathbf{w}_k, \mathbf{b}_k)$$

Gradient descent tell us that if we let

$$(\mathbf{w}_{k+1}, \mathbf{b}_{k+1}) = (\mathbf{w}_k, \mathbf{b}_k) - \eta \nabla C(\mathbf{w}_k, \mathbf{b}_k)$$

for sufficiently small $\eta > 0$, the *learning rate*. This will work, since

$$\triangle C = \nabla C \cdot \mathbf{x} = \nabla C \cdot (-\eta \nabla C) = -\eta \|\nabla C\|^2 \leq 0$$

Our task now is to compute ∇C, the gradient of C. However, this is no easy task, since we will typically have large T. Instead, we will complete this by a method of averaging stochastic gradient descent. We take a small sample $T' \subset T$ (this sample could even be size of just one), to get

$$\nabla C \approx \frac{1}{|T'|} \sum_{\mathbf{x} \in T'} \nabla C_{\mathbf{x}}$$

where $C_{\mathbf{x}} = \frac{1}{2}\|N(\mathbf{w}, \mathbf{b}, \mathbf{x}) - \mathbf{y}_{\mathbf{x}}\|^2$. In particular, the iteration for any given weight w_n from the weights $\mathbf{w}_n$ or bias b_n from the biases $\mathbf{b}_n$ is given by

$$w_{k+1} = w_k - \frac{\eta}{|T'|} \sum_{\mathbf{x} \in T'} \frac{\partial C_{\mathbf{x}}}{\partial w_n}$$

and

$$b_{k+1} = b_k - \frac{\eta}{|T'|} \sum_{\mathbf{x} \in T'} \frac{\partial C_{\mathbf{x}}}{\partial b_n}$$

The question now is: how do we calculate the values of $\frac{\partial C_{\mathbf{x}}}{\partial w_n}$ and $\frac{\partial C_{\mathbf{x}}}{\partial b_n}$? This will introduce an entirely new chapter, so we shall present the code for stochastic gradient descent, leaving blank these values which we will fill in later.

```
def SGD(self, training_data, epochs, mini_batch_size, eta, test_data):
    '''Completes the method of stochastic gradient descent'''
    for n in range(epochs):
        mini_batch = random.sample(training_data, mini_batch_size)
        '''We need to replace (?) with the average (over minibatch) derivative of C
            wrt each term in bias/weight'''
        [dCdb, dCdw] = (?)
        self.biases = [np.add(b,np.multiply(-(eta/mini_batch_size),Cb)) for b, Cb in
             zip(self.biases, dCdb)]
        self.weights = [np.add(w,np.multiply(-(eta/mini_batch_size),Cw)) for w, Cw
            in zip(self.weights, dCdw)]
        print("Epoch {0}: {1} / {2}".format(n, self.evaluate(test_data), len(
            test_data)))

def evaluate(self, test_data):
    '''Calculates number of correct answers network produces for the test data'''
    correct = 0
    for (x,y) in test_data:
        network_ans = self.feedforward(x)[1]
        real_ans = y
        if np.argmax(network_ans) == np.argmax(real_ans):
            xy_works = 1
        else:
            xy_works = 0
        correct += xy_works
    return correct
```

Note I have also included an evaluate function, which allows you to compare the correct answer against what the neural network has output. The truth condition can be changed depending on the expected output. Nielsen calls an answer 'correct' if the highest value of the output vector is in the same index as the ideal output vector, which seems suitable for many contexts, hence I have used this, though it need not always be the case and any condition that the user wants can be chosen depending on the context.

1.3 Backpropagation

In this section, we will find a way to compute the partial derivatives $\frac{\partial C_{\mathbf{x}}}{\partial w}$ and $\frac{\partial C_{\mathbf{x}}}{\partial b}$. For notational purposes, we will write $C_{\mathbf{x}}$ simply as C. Recall that given layer ℓ with n perceptrons and layer $\ell+1$ with m perceptrons, the neural network works by the system of equations

$$\mathbf{x}_\ell = \sigma\left(\mathbf{w}_\ell \cdot \mathbf{x}_{\ell-\mathbf{1}} + \mathbf{b}_\ell\right)$$

We define the quantity

$$\mathbf{z}_\ell = \mathbf{w}_\ell \cdot \mathbf{x}_{\ell-\mathbf{1}} + \mathbf{b}_\ell$$

and call this the *weight input to layer* ℓ. This was used in the programming of the feedforward algorithm. This definition will play an important role in finding the derivatives required. Furthermore, we define the *error of layer* ℓ by

$$\boldsymbol{\delta}_\ell = \left(\delta_i^\ell\right) \qquad \text{where} \qquad \delta_i^\ell = \frac{\partial C}{\partial z_i^\ell}$$

where z_i^ℓ is the i^{th} entry of $\mathbf{z}_\ell$, i.e. the weight input to neuron i of layer ℓ. So,

$$\mathbf{x}_\ell = \sigma\left(\mathbf{z}_\ell\right)$$

Firstly, we will find a formula for $\boldsymbol{\delta}_L$ the error of the final layer L, and then use this to compute all other errors, and finally the required partial derivatives. Using multivariate chain rule, we see

$$\delta_i^L = \frac{\partial C}{\partial z_i^L} = \sum_k \frac{\partial C}{\partial x_k^L}\frac{\partial x_k^L}{\partial z_i^L}$$

However, we know that

$$x_k^L = \sigma\left(z_k^L\right) \qquad \text{so} \qquad \frac{\partial x_k^L}{\partial z_i^L} = \begin{cases} \sigma'\left(z_i^L\right) & \text{for } i = k \\ 0 & \text{else} \end{cases}$$

Hence

$$\delta_i^L = \frac{\partial C}{\partial x_i^L}\sigma'\left(z_i^L\right)$$

We can simplify this further, since

$$C = \frac{1}{2}\|N(\mathbf{w},\mathbf{b},\mathbf{x}) - \mathbf{y}_{\mathbf{x}}\|^2 = \frac{1}{2}\|\mathbf{x}_L - \mathbf{y}_{\mathbf{x}}\|^2 \qquad \text{so} \qquad \frac{\partial C}{\partial x_k^L} = x_k^L - y_k$$

and in vector form,

$$\boldsymbol{\delta}_L = (\mathbf{x}_L - \mathbf{y}_{\mathbf{x}}) \circ \sigma'(\mathbf{z}_L)$$

where $\circ$ represents entrywise product. We now form an iteration to compute the other errors. Using chain rule again, we see

$$\delta_i^\ell = \frac{\partial C}{\partial z_i^\ell} = \sum_k \frac{\partial C}{\partial z_k^{\ell+1}}\frac{\partial z_k^{\ell+1}}{\partial z_i^\ell} = \sum_k \delta_k^{\ell+1}\frac{\partial z_k^{\ell+1}}{\partial z_i^\ell}$$

We can compute this derivative, since, after unfolding the matrix definition of weight input into index terms, we know

$$z_k^{\ell+1} = \sum_j w_{kj}^{\ell+1}x_j^\ell + b_k^{\ell+1} = \sum_j w_{kj}^{\ell+1}\sigma\left(z_j^\ell\right) + b_k^{\ell+1}$$

so taking derivatives, we get

$$\frac{\partial z_k^{\ell+1}}{\partial z_i^\ell} = w_{ki}^{\ell+1}\sigma'\left(z_i^\ell\right)$$

And so,

$$\delta_i^\ell = \sum_k \delta_k^{\ell+1} w_{ki}^{\ell+1} \sigma'\left(z_i^\ell\right)$$

or in vector form,

$$\boldsymbol{\delta}_\ell = \left(\mathbf{w}_{\ell+1}^T \cdot \boldsymbol{\delta}_{\ell+1}\right) \circ \sigma'\left(\mathbf{z}_\ell\right)$$

where T represents the transpose. The last layer error and this reverse iteration will allow us to compute error in every layer, and this will allow us to compute the partial derivatives. Again, we use the chain rule and differentiating the index terms for the definition of weight input to get

$$\frac{\partial C}{\partial w_{ij}^\ell} = \sum_k \frac{\partial C}{\partial z_k^\ell} \frac{\partial z_k^\ell}{\partial w_{ij}^\ell} = \frac{\partial C}{\partial z_i^\ell} x_j^{\ell-1} = \delta_i^\ell x_j^{\ell-1}$$

and

$$\frac{\partial C}{\partial b_i^\ell} = \sum_k \frac{\partial C}{\partial z_k^\ell} \frac{\partial z_k^\ell}{\partial b_i^\ell} = \frac{\partial C}{\partial z_i^\ell} = \delta_i^\ell$$

Below, I have programmed the algorithm to compute these derivatives. In the previous section, we simply replace the (?) with this algorithm, and we have now completed our neural network. I have also included a sigmoid prime function (the derivative of sigmoid) and a derivative of the cost function with respect to the output vector. This will allow for ease of code modification if a different sigmoid and cost function were chosen.

```
    def backpropogation(self, mini_batch):
        '''Calculates sum of partial derivatives of C wrt weights and biases for a
            given minibatch'''
        dCdb = [np.zeros(b.shape) for b in self.biases]
        dCdw = [np.zeros(w.shape) for w in self.weights]
        for (x,y) in mini_batch:
            [z,x_L] = self.feedforward(x)
            activations = [sigmoid(zed) for zed in z]
            activations.insert(0,x)
            delta = [np.multiply(dCdx_L(x_L, y),sigmoid_prime(z[-1]))]
            for n in range(self.L-2,0,-1):
                delta.insert(0,np.multiply(((self.weights[n]).transpose()).dot(delta
                    [0]),sigmoid_prime(z[n-1])))
            db = delta
            dw = [np.zeros(w.shape) for w in self.weights]
            for l in range(self.L - 1):
                for i in range(self.sizes[l+1]):
                    for j in range(self.sizes[l]):
                        dw[l][i][j] = delta[l][i]*activations[l][j]
        dCdb = [np.add(Cb,b) for Cb, b in zip(dCdb, db)]
        dCdw = [np.add(Cw,w) for Cw, w in zip(dCdw, dw)]
        return [dCdb, dCdw]

def sigmoid_prime(x):
    return np.divide(np.exp(x),((np.exp(x)+1)**2))

def dCdx_L(x, y):
    ''' Calculates derivatives wrt x_L of C.
    x is column vector of activation outputs, x_L.
    y is column vector of true outputs.'''
    return np.add(x,-y)
```

2 More Advanced Training of Neural Network

2.1 Cross-Entropy Cost Function

One way of improving the speed at which a neural network learns is through the choice of the cost function. Previously, we used a quadratic cost function, however this certainly does not provide for the fastest learning. Looking at the derivatives of the cost function (for a single training value) with respect to the output layers weights and biases, we have

$$\frac{\partial C}{\partial w_{ij}^L} = \left(x_i^L - y_i\right) \sigma'\left(z_j^L\right) x_j^{L-1}$$

and

$$\frac{\partial C}{\partial b_i^L} = \left(x_i^L - y_i\right) \sigma'\left(z_i^L\right)$$

The problem here is due to the derivative of sigma. When $x_i^L = \sigma\left(z_i^L\right)$ becomes near 0 or 1 (which it often does, since outputs are often desired to be lists of 0's and 1's), then $\sigma'\left(z_i^L\right)$ becomes near 0, so a very slow rate of learning occurs. This is fine if are our network outputs 0 when we want 0, but, if by our random choice of weights and biases, our network outputs 1, the network will still learn very slowly and far from what we want.
To counter this, we introduce the cross-entropy cost function given by

$$C(\mathbf{w}, \mathbf{b}) = -\frac{1}{n} \sum_{\mathbf{x} \in T} \sum_i y_i \log x_i^L + (1 - y_i) \log\left(1 - x_i^L\right)$$

where $\sum_i$ is summing over the output neurons. It can be checked this makes sense as a cost function (for those with an ideal output of 0's and 1's). It will turn out this will solve our problem perfectly. See that, associated to this cost function (for a single training value), as before, we have

$$\delta_i^L = \frac{\partial C}{\partial x_i^L} \sigma'\left(z_i^L\right)$$

though now, recalling that $x_i^L = \sigma\left(z_i^L\right)$, we have

$$\frac{\partial C}{\partial x_i^L} = -\left(\frac{y_i}{x_i^L} - \frac{(1 - y_i)}{(1 - x_i^L)}\right) = -\left(\frac{y_i}{\sigma\left(z_i^L\right)} - \frac{(1 - y_i)}{\left(1 - \sigma\left(z_i^L\right)\right)}\right) = -\frac{y_i - \sigma\left(z_i^L\right)}{\sigma\left(z_i^L\right)\left(1 - \sigma\left(z_i^L\right)\right)}$$

However, it can be checked that

$$\sigma'(z) = \sigma(z)\left(1 - \sigma(z)\right)$$

so we get a cancellation and simply get

$$\delta_i^L = x_i^L - y_i$$

This makes more sense as an error. In fact, our partial differentials now become

$$\frac{\partial C}{\partial w_{ij}^L} = \delta_i^L x_j^{L-1} = \left(x_i^L - y_i\right) x_j^{L-1}$$

and

$$\frac{\partial C}{\partial b_i^L} = \delta_i^L = x_i^L - y_i$$

See that these are exactly the same as the quadratic cost function without the sigma prime term. This completely solves our problem, as this will no longer slow down learning rates when the output neuron gives a zero when a one is desired (or vice versa). In fact, when the output value is far away from the ideal value, the partial derivatives are large, just as we desire.

This might seem like a magical function to do such a thing, but what has really happened is that we have set the partial derivatives to be as we want and solved the partial differential equations to find what cost function will work, arriving at the cross-entropy function. To programme this in python, all that needs modifying is the choice of the function for $\frac{\partial C}{\partial x_L}$; everything else can remain the same. Admittedly this may not be the most computationally inexpensive method, though it certainly is the easiest to implement.

```
def dCdx_L(x, y):
    ''' Calculates derivatives wrt x_L of C.
    x is column vector of activation outputs, x_L.
    y is column vector of true outputs.'''
    return np.divide(np.add(x,-y),np.multiply(x,np.add(1,-x)))
```

A cheaper way would be to simply modify the error to be

in the backpropogation algorithm.

2.2 Softmax

Softmax is an alternative approach to solving the learning slowdown. We modify the output of the final layer of neurons by using the *softmax function* instead of the sigmoid function. We let the output of the i^{th} neuron of the final layer be

$$x_i^L = \frac{e^{z_i^L}}{\sum_k e^{z_k^L}}$$

where $\sum_k$ is the sum over all final layer neurons. Clearly, the sum of the output activations is 1. This makes using the softmax function most useful for cases where the outputs should be considered as a probability distribution, e.g. likelihood of horse to win a race. In this sense, we modify the cost function (for a single training value) to the log-likelihood cost function

$$C(\mathbf{w}, \mathbf{b}) = -\log x_*^L$$

where the ideal output for the training value is all 0's and a 1 on the output neuron x_*^L. Like the cross-entropy function, this solves our learning slowdown problem, since

$$\delta_i^L = \frac{\partial C}{\partial z_i^L} = \sum_k \frac{\partial C}{\partial x_k^L}\frac{\partial x_k^L}{\partial z_i^L} = -\frac{1}{x_*^L}\frac{\partial x_*^L}{\partial z_i^L}$$

but

$$\frac{\partial x_*^L}{\partial z_i^L} = \begin{cases} \frac{e^{z_*^L}\left(\sum_{k\neq *} e^{z_k^L}\right)}{\left(\sum_k e^{z_k^L}\right)\left(\sum_k e^{z_k^L}\right)} = -x_*^L\left(x_*^L - 1\right) & \text{for } i = * \\ \frac{-e^{z_*^L}e^{z_i^L}}{\left(\sum_k e^{z_k^L}\right)\left(\sum_k e^{z_k^L}\right)} = -x_*^L x_i^L & \text{for } i \neq * \end{cases}$$

Hence, remembering that the ideal output y_i is zero for $i \neq *$ and 1 for $i = *$, we can simplify to get

$$\delta_i^L = x_i^L - y_i$$

This is exactly the same as the cross-entropy function, and so the partial derivatives are also exactly the same, so solve our problem!

2.3 Overfitting/Overtraining

One problem we can encounter is choosing a training data set which is too small (this was slightly overlooked in the previous section: typically we have training data which the network learns from, and then we test the trained network on some test data which the network has never seen before and does not learn from, just to test how smart the network has become). Nielsen provides some interesting graphs using the handwritten digits example, which I'd suggest looking at here. When using a small training data set, whilst the cost of the training data decreases smoothly (as expected due to our gradient descent) and the classification accuracy increases smoothly (though reaching near 100% accuracy early on), the same cannot be said for the test data. Instead, the classification accuracy levels off after a point. Furthermore, the cost starts to increase even! Why does this happen? It's because the network has experienced *overfitting* or *overtraining* to the training data; intuitively speaking, it hasn't had enough exposure to the various situations (input types) that can occur to know how to generalise to give an accurate result to them.

To prevent overfitting, we keep another set of data called the *validation data*. We train the network on the training data until our accuracy has saturated on the validation data (which determining can be somewhat difficult in itself), where we stop training the network. We will end up testing all types of hyper-parameters such as mini batch size, learning rate, network structure, etc. We can choose the best ones by seeing which gives the highest accuracy (after saturation) on the validation data. So why use the validation data? If we used the test data, we could end up simply picking hyper-parameters that happen to work very well for the test data but don't in general, i.e. we have overfitted the hyper-parameters to the test data. Instead, we do this on the validation data, choose the hyper-parameters, then evaluate using the test data to see how well the network generalises to totally unseen data. This is called the *hold out method*.

2.4 Regularization

If you can't change the network structure (by reducing the number of neurons) or get more training data, we can reduce the overfitting by a method of *regularization*. There are many types of regularization, so we shall explore several of these (indeed, regularization serves an entire area of mathematics).

2.4.1 L_2 regularization

We add a new term to the cost function, the regularization term

$$\frac{\lambda}{2n}\|\mathbf{w}\|_{\ell^2}^2$$

where $\mathbf{w}$ is a vector of all weights (in any order). We call $\lambda > 0$ the regularization parameter. Other literature may not include the $2n$ in the denominator, and this is indeed unnecessary since it is captured by the choice of λ, though we shall follow Nielsen. You may notice that the name L_2 comes from the ℓ_2-norm for the sequence space (not to be confused with the L_2-norm for the function space). This regularisation term will make our network prefer smaller weights (though still allowing large weights provided they significantly reduce the original cost function), and our choice of λ will decide how much of preference towards smaller weights compared to reduction to the original cost function (which we now denote by the term C_0) value is made. Our partial derivatives now become

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

and

$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}$$

Note the bias term remains unchanged. Nielsen gives another set of fantastic graphs continuing his example, showing that adding this regularization term does indeed increase the accuracy and remove overfitting, which can again be seen in his book here.

Programming this isn't too difficult. Recalling section 1.2 and looking at how the stochastic gradient descent works, we only need to modify the iteration rules to match our new cost function and partial derivative with respect to weights. Now, we have

$$w_{k+1} = w_k - \frac{\eta}{|T'|}\sum_{\mathbf{x}\in T'}\frac{\partial C_{\mathbf{x}}}{\partial w_k} = \left(1 - \frac{\lambda\eta}{n}\right)w_k - \frac{\eta}{|T'|}\sum_{\mathbf{x}\in T'}\frac{\partial C_{0_{\mathbf{x}}}}{\partial w_k}$$

So, we just need to include the variable λ and change one line in the stochastic gradient descent function

```
def SGD(self, training_data, epochs, mini_batch_size, eta, lam, test_data):
    '''Completes the method of stochastic gradient descent'''
    for n in range(epochs):
    mini_batch = random.sample(training_data, mini_batch_size)
    [dCdb, dCdw] = self.backpropogation(mini_batch)
    self.biases = [np.add(b,np.multiply(-(eta/mini_batch_size),Cb)) for b, Cb in zip
        (self.biases, dCdb)]
    self.weights = [np.add(np.multiply(1-lam*eta/len(test_data),w),np.multiply(-(eta
        /mini_batch_size),Cw)) for w, Cw in zip(self.weights, dCdw)]
    print("Epoch {0}: {1} / {2}".format(n, self.evaluate(test_data), len(test_data))
        )
```

Why does regularization work? This is quite a difficult question to answer concisely. It should be obvious that adding the regularization term to the cost function will mean the network will have smaller weights than if the term were not there. Smaller weights means that the behaviour of the network will change less than larger weights, and so provides a simpler model which generalises better and avoids learning the peculiarities of the training data (c.f. a roughly linear set of data is modelled well by a simple line v.s. high order Lagrange polynomial which generalises badly). It is a bit of a leap of faith to suggest that simple models are the 'correct' models, since complex models sometimes are needed, though this is extremely unlikely in the case of a small training data set which suffers from overfitting. Note that the biases are not included in the regularisation term since these do not make neurons sensitive to their inputs, so it would be unnecessary.

2.4.2 L_1 regularization

As expected, we add the regularization term

$$\frac{\lambda}{n}\|\mathbf{w}\|_{\ell_1}$$

to the cost term. This will work in exactly the same way as L_2 regularization, however the partial derivative with respect to the weights now becomes

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

so gradient descent occurs the same for all weights irrespective of how large or small the weight is, unlike in L_2-regularization. This type of regularization generally gives lots of small and large weights, whereas L_2 regularization gives a more rounded set of weights. This should be expected: L_1 regularization will make the weights close to zero in a $\|\cdot\|_{\ell_1}$ sense and L_2 regularization will make the weights close to zero in a $\|\cdot\|_{\ell_2}$ sense, i.e. a Euclidean sense. Programming this is equally as easy, as we now have

$$w_{k+1} = w_k - \frac{\lambda\eta}{n}\text{sgn}(w_k) - \frac{\eta}{|T'|}\sum_{\mathbf{x}\in T'}\frac{\partial C_{0_{\mathbf{x}}}}{\partial w_k}$$

so

```
self.weights = [np.add(np.add(w,-lam*eta*np.sign(w)/len(test_data)),np.multiply(-(
    eta/mini_batch_size),Cw)) for w, Cw in zip(self.weights, dCdw)]
```

2.4.3 Dropout

I will very briefly introduce the idea of *dropout*. Given a neural network, we decide to randomly remove some of the hidden neurons and their connections, and train the network. We then place back in the neurons and their connections, remove others at random, and start the process again. Once we have done repeating this, we reduce all the weights and biases by a proportion of how many were removed (i.e. removing half the neurons means we half all the weights and biases).

This method seems completely random. The logic behind its reasoning is that it prevents neurons from relying on any other neurons, and forces them to learn more robustly in conjunction with other neurons. It can also be thought of as training various, random neural networks (which will train differently and overfit differently based on the random weights and biases which it starts with), and then averaging these, which should reduce any individual overfitting effects from a given neural network.

2.4.4 Artificially expanding the training data

This will again be a brief discussion, the title says it all really. We prevent overfitting by having a large set of training data, but this is often costly and not always possible. Instead, we can expand the training data artificially. Doing this depends entirely what the context of the training data is. In Nielsen's example, we use photos of handwritten digits, and these can be expanded by small rotations, translation, distortions (particularly those that mimic spasms in hands when writing) or adding background noise. We can quite easily expand the data set thousand-fold or even more by doing this. For other examples, you'll have to think for yourself! The key idea is expand the data set in a way which reflects real world operations.

2.5 Weight Initialization

We have been using randomly chosen weights and biases using a normal distribution with mean 0 and standard deviation of 1. However, this results in lots of saturated neurons. For example, if we have a binary input layer with n_{in} neurons on, using our current method, we end up with a weight input distributed normally with mean 0 and standard deviation $\sqrt{n_{in}+1}$, as we have summed all the distributions. This is quite a large standard deviation, so we can end up with extreme values for the weight input, which we know results in saturated neurons. To counter this, we initialize the weights as follows: $w_{ij}^{(\ell+1)}$, the weight between perceptron j in level ℓ to perceptron i in level $\ell+1$, is chosen from a normal distribution with mean 0 and standard deviation $\frac{1}{\sqrt{n_{in}}}$ where n_{in} is the number of weights into perceptron i in level $\ell+1$ (in a complete network, this is equal to the number of neurons in level ℓ). Now the weight input will be more like a normal distribution with standard deviation nearer to 1 (in fact, it would be exactly 1 if the outputs of level ℓ were all 1.) This isn't optimal, but does a good job of improving the issue, and can be experimentally shown to improve accuracy. Choosing the bias has negligible effect on the issue, and can actually be chosen as all zeroes.

Programming this is extremely simple, as

```
self.biases = [np.zeros(sizes[n+1], 1) for n in range(len(sizes)-1)]
self.weights = [np.random.normal(0, np.divide(1,np.sqrt(sizes[n])), (sizes[n+1],
    sizes[n])) for n in range(len(sizes)-1)]
```

2.6 Hyper Parameters

We have encountered many hyper parameters throughout our journey: choice of network structure, regularization parameter λ, mini batch size $|T'|$, learning rate η. Without good choices for these values, the network will never learn and will act randomly. The task is to find values that will have any kind of learning to start with which is better than pure chance, and then we can apply more specific methods to improve certain hyper parameters. The best method is to start with very basic structures and training values, saving a lot of time, and making things more complicated from there. This might seem strange, since adjusting one hyper parameter could have a dramatic impact on the others, and we begin a wild goose chase of tweaking hyper

parameters which change the optimal value of others, however this will essentially be the method. Finding optimal values is often too difficult, and getting good values is an achievement in itself.

2.6.1 Early stopping

Instead of worrying about choosing the number of epochs, we will stop the network from learning when the validation accuracy stops increasing. This will immediately prevent overfitting. The question becomes: how do we know when the validation accuracy has stopped improving? There are various methods which answer this. One is to terminate the algorithm if the validation accuracy does not increase after x epochs, where x is to be chosen (Nielsen uses $x = 10$). This introduces a new hyper parameter to optimise!

2.6.2 Learning rate

Thinking of learning rate as size of steps during a gradient descent, if we choose a rate too large, the descent will be chaotic. However a rate too large, whilst in general reducing the cost, will also act with slightly random behaviour and can sometimes increase the cost. Hence, we choose the initial rate to have the order of magnitude of the largest rate at which the cost (on the validation data) doesn't increase or oscillate. We can choose this more precisely if we wish to.

After the cost has reduced to a relatively small level, we will want to decrease the learning rate (imagine wanting to reach the precise bottom of a bunker but being forced to take metre steps). Hence, we will want a learning schedule, where we continually decrease the learning rate when the cost/accuracy no longer decreases. This schedule gives an infinite number of choices of hyper parameters, however a simple one could be to half the learning rate once the cost/accuracy no longer decreases (using early stopping).

2.6.3 Regularization parameter

Nielsen suggests using no regularisation parameter and determining the learning rate first. Then, much like the learning rate, we try to determine the order of magnitude of λ by increasing or decreasing by multiples of ten which improve the validation accuracy, and then fine tuning the value.

2.6.4 Mini batch size

Mini batch size is quite independent of other hyper parameters, so finding an optimal value for this is not too much of an issue. The only real issue is the time it takes to calculate the average derivative when using a large mini batch, so a compromise must be achieved. We should plot the validation accuracy versus real time elapsed, and choose the mini batch size which gives the best accuracy in the shortest time. Then, the remaining hyper parameters can be chosen.

2.6.5 Grid search

An expensive, but automated, way the find the optimal values is to do a simple grid search. We can try small intervals of all of the values for each parameter (presumably around some known 'good' values for each parameter) and trial them to get the validation accuracy. We should use the values for parameters with the largest validation accuracy.

2.7 Other learning methods

2.7.1 Hessian Technique

The cost C is a function of the weights and biases w_1, w_2, ..., which we write as a vector $\mathbf{w}$. Using multivariate Taylor's theorem, we get

$$C(\mathbf{w} + \Delta\mathbf{w}) = C(\mathbf{w}) + \sum_i \frac{\partial C}{\partial w_i}\Delta w_i + \frac{1}{2}\sum_j \sum_i \Delta w_j \frac{\partial C}{\partial w_j \partial w_i}\Delta w_i + \dots$$

Encoded as matrices and ignoring other terms, this can be written

$$C(\mathbf{w}+\Delta\mathbf{w}) \approx C(\mathbf{w}) + \nabla C \Delta\mathbf{w} + \frac{1}{2}\mathbf{w}^T H \mathbf{w}$$

where H is the Hessian matrix. Our aim is to minimise this expression through the choice of $\Delta\mathbf{w}$. This is done by letting

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

which makes the RHS be zero. Hence, our update is now

$$\mathbf{w}_{k+1} = \mathbf{w}_k - H^{-1}\nabla C(\mathbf{w}_k)$$

In practice, this is sometimes too big of a step due to our approximations, so we can still use the learning rate. The problem is that computing the Hessian can be incredibly expensive, taking $\Theta(n^2)$ memory, so isn't always the most practical method.

2.7.2 Moment based gradient descent

We will perform gradient descent using similar principles as in real life: thinking of a ball falling into a valley, the ball will fall quickly for steep gradients, encounter resistance, carry velocity and momentum which it will lose quickly when travelling uphill, etc. Hence, we introduce $\mathbf{v}$ with terms corresponding to each term in $\mathbf{w}$ (the weights and biases). We update them according to the rules

$$\mathbf{v}_{k+1} = \mu\mathbf{v}_k - \eta\nabla C(\mathbf{w}_k)$$

and

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \mathbf{v}_{k+1}$$

where μ is a hyper parameter of friction which is called the *momentum coefficient*. We think of $-\mu\nabla C$ as the force applied to the ball in the cost valley. It is relatively easy to see why this produces a system similar to our intuitive understanding of a ball in a valley. If $\mu = 0$, we are in the case of our usual gradient descent, with a lot of friction. Also, when $\mu = 1$, our system has no friction. We optimise μ as a hyper parameter just as we do with the others. To implement this, we introduce

```
self.Bvelocity = [np.zeros((sizes[n+1], 1)) for n in range(len(sizes)-1)]
self.Wvelocity = [np.zeros((sizes[n+1], sizes[n])) for n in range(len(sizes)-1)]
```

and in the SGD, after introducing the new parameter mu, we update according to the rules

```
self.Bvelocity = [mu*v-eta*Cb for v, Cb in zip(self.Bvelocity, dCdb)]
self.Wvelocity = [mu*v-eta*Cw for v, Cw in zip(self.Wvelocity, dCdw)]
self.biases = [b+vb for b, vb in zip(self.biases, self.Bvelocity)]
self.weights = [w+vw for w, vw in zip(self.weights, self.Wvelocity)]
```

2.7.3 Other neuron functions

We need not always use the sigmoid neuron. Another choice is tanh neuron. This has a relationship with the sigmoid function as

$$\sigma(z) = \frac{1+\tanh(\frac{z}{2})}{2}$$

So, instead of having outputs in the range of 0 and 1, it has outputs in the range -1 and 1. There are some cases where this function will be learn quicker than the sigmoid neuron, though it is unknown when this will be the case.

Another neuron function is the *rectified linear neuron* given by

$$\max(0, \mathbf{w}\cdot\mathbf{x} + b)$$

This has some advantages, as increasing the weighted input to a rectified linear neuron will never cause it to saturated.

3 Attempted Examples

In this section, I attempt to use the neural network to learn the recognise the gender of a person by their voice. Though firstly, I try Nielsen's MNSIT digit set. Sadly, my network could not reach the accuracies of Nielsen - when introducing any hidden layers, the programme suffered errors from overflow using exp/log/sqrt which prevented learning. I'm unsure why my code suffered from this and Nielsen's did not. Furthermore, I copied and pasted Nielsen's code and tried using his parameters in my IDE and still encountered the errors and never achieved his accuracy! So, this section won't actually be able to obtain any results since making the network to complicated structurally encounters these frustrating errors preventing learning. I'm not entirely sure whether the errors prevented learning (since np.clip could easily solve the problem), though I'm also unsure why Nielsen never encountered the problem (or never mentioned it if he did) like I did when I ran his code, or whether there was something else wrong. Learning was certainly occuring, though never at the rapid pace others achieved and never reaching such high levels of accuracies, despite trying many values for hyper parameters. Instead, I'll discuss very briefly my attempts at an example, and perhaps return to it at a later date.

Using a Kaggle dataset, we have data on 3,168 recorded voice samples (we will keep 350 for test data, 350 for validation data and 2468 for training data), collected from male and female speakers; the voice samples are pre-processed by acoustic analysis in R using the seewave and tuneR packages, with an analyzed frequency range of 0Hz-280Hz [2]. The information includes things such as details about frequency, kurtosis, skewness, etc. Precisely, we have 20 input values (so will have 20 input neurons). We will only need a 2 output neurons, each indicating the probability of the voice being male or female. Firstly, we must import this csv of data into python. We must have a list of tuples, each tuple having a vector of input values and a second entry of [0,1] or [1,0]. The following code completes this method.

```
import csv
with open('voice.csv', 'rt') as f:
    reader = csv.reader(f)
    all_data = [row for row in reader]
    inputs = []
    outputs = []
    for i in range(len(all_data)):
        inputs.append(np.transpose(np.array([[float(j) for j in all_data[i][:-2]]]))
            )
        outputs.append(np.transpose(np.array([[float(j) for j in all_data[i][-2:]]])
            ))
    data = [(inp, out) for inp, out in zip(inputs, outputs)]

validation_data = data[0:350]
test_data = data[350:700]
training_data = data[700:]
```

Just as in Nielsen, we determine the learning rate using the training data and seeing which order of magnitude is largest and causes the cost of the training data to decrease smoothly. Using the cross entropy cost function and no regularisation, we acquire a reasonable value of $\eta = 0.01$ with this method. I also used the simplest network structure possible, 20 inputs and 2 output neurons with no hidden neurons, and mini batch size of the full training data.

With no regularization parameter, our validation data has an accuracy of 200/350 after 100 epochs. Introducing any type regularisation didn't seem to increase the classification accuracy.

At this point, I'd like to make the network have a more complicated structure, though this frequently gave the errors I discussed. I did see some marginal improvements using a hidden layer of 10 neurons, though the highest classification accuracy I ever managed to achieve was roughly 70%, nowhere near high classification accuracies (95% ore more) people have achieved.

Neural networks can certainly be used to classify genders by voice. There are interesting kernels on the Kaggle thread using Tensorflow to devise a neural network which classifies them, and achieves a classification

accuracy of 96%. Perhaps this exercise has been useful if only to highlight the frustrations of finding hyper parameters (sometimes being so frustratingly time consuming to find optimal values that other machine learning methods become more useful, just because of how much quicker they are) and that using something such as Tensorflow is probably a better idea and more structured.

References

[1] Michael Nielsen. "Neural Networks and Deep Learning", 2017. URL `http://neuralnetworksanddeeplearning.com/index.html`. [Accessed: 6th July 2017].

[2] Kaggle. URL `https://www.kaggle.com/primaryobjects/voicegender`. [Accessed: 6th July 2017].