Designing a Neural Network

Joe Tilley

August 25, 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

$$C(\mathbf{w}_{n+1}, \mathbf{b}_{n+1}) \leq C(\mathbf{w}_n, \mathbf{b}_n)$$

Gradient descent tell us that if we let

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

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

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

and

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

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 network_ans.index(max(network_ans)) == real_ans.index(max(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)))
```

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 $\sum_i$ 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, 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

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

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.

References

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