Notes from Neural Networks and Deep Learning

February 2, 2020

1 ToDos

1.1 Chapter 1

1.1.1 Install pip2.7 and install packages

1.1.2 Train the network using the full code

1.2 Chapter 2

1.2.1 Once all components are explained, convert the code to python3

2 Chapter 1: Using neural nets to recognize handwritten digits

2.1 The sigmoid function

The sigmoid function is defined as:

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

2.2 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.

2.3 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.

2.4 Code

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

- **num$_{\textbf{layers}}$**: 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.

3 Chapter 2: How the back propagation algorithm works

3.1 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:

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

3.2 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:

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

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.

3.3 The Hadamard product

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

3.4 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:

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

(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 $\frac{\partial C}{\partial b^l_j}$.

3.4.1 Error in the output layer, δ^L

The components of δ^L are given by:

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

- $\frac{\partial C}{\partial a^L_j}$ 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:

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

3.4.2 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.

3.4.3 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 $\delta_{\mathrm{j}}^{\mathrm{l}}$ 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.

3.4.4 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.

3.5 Proofs for the four fundamental equations

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

3.5.1 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 3.4.1. 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:

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

3.5.2 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:

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

3.5.3 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.

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

Using the same approach as in Section 3.5.3, 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:

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

3.6 The backpropagation algorithm

3.6.1 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.

3.6.2 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 $w^l \to w^l - \frac{\eta}{m}\sum_x \delta^{x,l}(a^{x,l-1})^T$, and the biases according to the rule $b^l \to b^l - \frac{\eta}{m}\sum_x \delta^{x,l}$

3.7 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.**

3.7.1 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 nabla$_\text{b}$ as it is returned currently.** This could be done within the backprop$_\text{matrix}$ function too

4 Chapter 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.

4.1 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$_2$, ... \$ 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.

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:

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