Neural Networks and Deep Learning

Neurons and Neural Nets

The Perceptron

Definition. A perceptron is an algorithm whose inputs are n-dimensional vectors with binary outputs given by

$$p(x) = \begin{cases} 0, & w \cdot x + b \leq 0 \\ 1, & w \cdot x + b > 0 \end{cases}$$

where $x, w \in \mathbb{R}^n$, w is the vector of weights and $b \in \mathbb{R}$ is the bias.

The vector w tells us how "important" each component of x is, that is, how much each compent x_i contributes to the output. The magnitude of the bias term b tells us how hard it is to switch from one binary classification to another. For example, if b is very positive, then the output is likely to be 1 regardless of what the weights and inputs are.
With learning, we'd like a small change in inputs to cause only a small change in outputs. This is not possible with the perceptron since it takes on binary values. So we introduce the sigmoid neuron.

The Sigmoid Neuron

Definition. The sigmoid function $\sigma : \mathbb{R} \to (0, 1]$ is defined to be

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

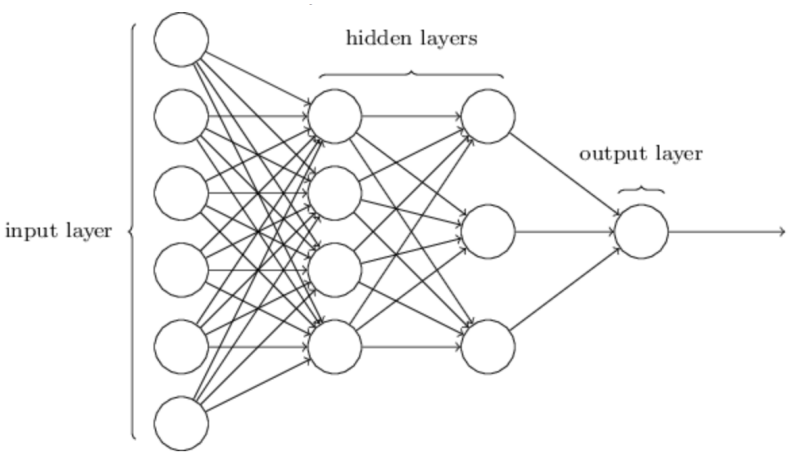
The output of a sigmoid neuron is

$$\sigma(w \cdot x + b) = \frac{1}{1 + \exp(-(w \cdot x + b))} \in (0, 1].$$

Note. A sigmoid neuron is simply a "smooth" perceptron. In fact, when $z = w \cdot x + b$ is large, $\sigma(z) \approx 1$ just like how a perceptron would behave. The same is also true for small negative z.

Structure of Neural Networks

Neural networks consists of three parts: the input layer, the hidden layer(s), and the output layer. Neural



nets are also sometimes referred to as "multilayered perceptrons" or MLP, though the layers consists of sigmoid neurons not perceptrons.

Example 1. Suppose we'd like to use a neural net to classify a hand written numeral 9. What we might consider is taking a 64×64 image and having our input layer as the intensity of each pixel. Thus our input layer would be a $4,096$-dimensional vector. The output layer would a single neuron with output value of 0.5 or less indicating "not a nine" and "a nine" otherwise.

So far, the neural nets we've been discussing takes as inputs the outputs from the previous layer. This is what we call *feed forward* neural nets. In such nets, the information is always fed forward, not back. Other networks allow for feedback loops and are called *recurrent neural networks*. Though recurrent nets are more akin to how human brains work, they are (to date) not as powerful as feed forward nets.

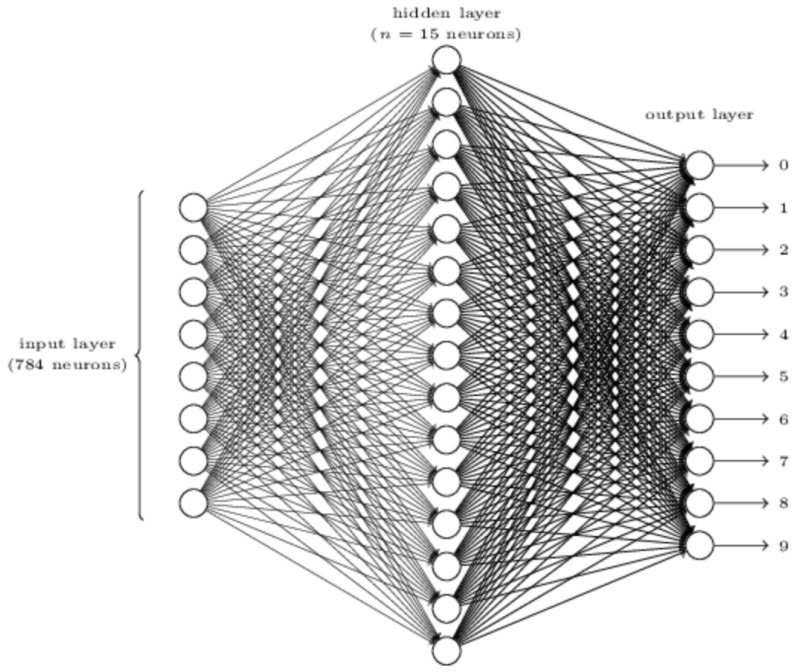
A Simple Classifier

We now turn our attention to classifying handwritten digits. To solve this problem, we define two sub-problems:

i We'd like to break an image containing many digits into a sequence of individual images.

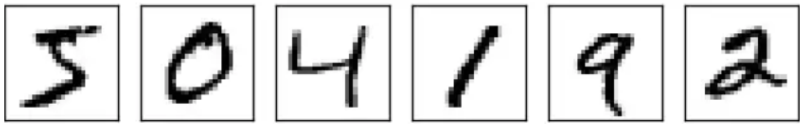
ii Then we'd like the program to correctly classify the digit.

It turns out the first sub-problem can be solved using the solution to the second sub-problem. Once we have a robust classifier, we can rate our segmentation of the string of digits based on how well our classifier performs on the segments. So we focus our efforts on developing a good classifier.
The classifier we'll develop will take in as input a 28×28 image of a handwritten digit. So the input layer will be a 784-dimensional vector. The pixels are greyscale, so the values of the input neurons will range from $[0, 1]$ where 0 represents white and 1 represents black. The hidden layer will consist of $n = 15$ neurons. We'll adjust this as we go along. The output layer consists of 10 neurons; one for each possible classification. The values of the individual output layer neuron represents what the algorithm thinks the classification should be. Here is a sketch of the net:



Gradient Descent

We'll be using the MNIST data set which contains tens of thousands of scanned hand written digits. Here are a few images from the data set: The data set can be split into two parts: training data and test



data. The training data consists of $60,000$ images which we'll use to train our algorithm. The remaining $10,000$ images will be used to evaluated how well our algorithm performs.
Let x denote an individual training instance. That is x is a 784-dimensional vector whose entries are greyscale values for each pixel of the image. The corresponding output will be a 10-dimensional vector. For example, if the images depicted is a 2, then $y = y(x) = (0, 0, 1, 0, 0, 0, 0, 0, 0, 0)^t$ is the desired output. Our goal is now to develop an algorithm which lets us find weights w and biases b such that the output from the network approximates $y(x)$ for all x in our training set.
We need a metric that tells us how well we are approximating the true solution. To that end, we define:

$$C(w, b) = \frac{1}{n} \sum_x \|y(x) - a\|_2^2 .$$

Here a denotes the vector of outputs from our network and $y(x)$ is the true solution, We call C the cost function or the mean squared error (MSE).

Note. The cost function is nonnegative for all w and b. Furthermore, $C \approx 0$ when $y(x)$ is close to the approximation a.

It's clear to see that when C is small, our network has done a good job. To find a set of weights and biases that minimizes the cost function, we'll use an algorithm called gradient descent.

Note. The reason we try to minimize the cost function as opposed to maximizing the number of correct classification is that the cost function is smooth. That is, a small change in inputs cause a small change in output. So using this, we can determine more easily how changing weights and biases change our output.

Consider an arbitrary function of two variables $f(x_1, x_2)$. Then f will be some curve in $\mathbb{R}^3$. To find the minimum of the function we can start at some point (x_1, x_2) and see where we go when we move Δx_1 in the x_1 direction and Δx_2 in the x_2 direction. Vector calculus tells us that

$$\Delta f = \frac{\partial f}{\partial x_1} \Delta x_1 + \frac{\partial f}{\partial x_2} \Delta x_2$$

or more succinctly

$$\Delta f = \nabla f \cdot (\Delta x_1, \Delta x_2) = \nabla f \cdot \Delta x.$$

Since we are minimizing f, we'd like Δf to be negative and so we choose x so that this happens. In particular, let

$$\Delta x = -\eta \nabla f$$

for some small η. Then we have that

$$\Delta f = \nabla f \cdot -\eta \nabla f = -\eta \|\nabla f\|^2$$

which is guaranteed to be negative. The goal of this process is we iteratively move x in such a way so that f reaches its minimum:

$$x \to x' = x - \eta \nabla f.$$

To make gradient descent work, we need to choose η small enough so that the approximation is good, but nor too small so that the algorithm runs slowly. More generally, given a function of n variables, $f(x_1, \ldots, x_n)$, we have

$$\begin{aligned} \Delta f &= \frac{\partial f}{\partial x_1} \Delta x_1 + \cdots + \frac{\partial f}{\partial x_n} \Delta x_n \\ &= \sum_{j=1}^{n} \frac{\partial f}{\partial x_j} \Delta x_j = \nabla f \cdot \Delta x. \end{aligned}$$

From here, we set $\Delta x = -\eta \nabla f$ and proceed as in the 2-dimensional case. As for our problem, the function f we are minimizing will be the cost function C. We update the weights and biases as such:

$$w_i \to w_i' = w_i - \eta \frac{\partial C}{\partial w_i}$$

$$b_k \to b_k' = b_k - \eta \frac{\partial C}{\partial b_k}.$$

The cost function is an average of *individual* training examples. This is an issue since in practice, we need to compute the gradient ∇C for each individual training instance and then average them. This can be costly when we have a large number of training examples. The solution to this is *stochastic gradient descent*. The idea is to take random subsets of the data and compute the gradient on the smaller subset. The random subsets are called *mini-batches*. Provided the subset is chosen large enough the average of the smaller set should approximate the average of the whole set:

$$\frac{\sum_{j=1}^{m} \nabla C_{x_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} \approx \nabla C$$

So given weights w and biases b,

$$w_i \to w_i' = w_i - \frac{\eta}{m} \sum_j \frac{\partial C_{x_j}}{\partial w_i}$$

$$b_k \to b_k' = b_k - \frac{\eta}{m} \sum_j \frac{\partial C_{x_j}}{\partial b_k}$$

where the sum over j indicates we are summing over the training examples in our current mini-batch. Then we choose another mini-batch until the training set is exhausted. Once we have reached that point we way an *epoch* of training has been completed and move on to the next training epoch.
Though using stochastic gradient descent won't be totally accurate in terms of the gradient, it really doesn't matter! The whole point of gradient descent is to find the minimum of the cost function not accurately computing the gradient.

Implementation

We will be implementing our neural net in Python 2.7. The main piece of the neural net is the `Network` class. Here's the code for initializing a `Network` object:

```
class Networks(object):
        def __init__(self, sizes):
                self.num_layers = len.(sizes)
                self.sizes = sizes
                self.biases = [np.random.randn(y,1) for y in sizes[1:]]
                self.weights = [np.random.randn(x,y) for x,y in zip(sizes[:-1],sizes[1,:]]
```

To initiate a `Network` object, we feed in a list indicating the size of each layer. For example `Network([3,2,1])` creates a network with 3 neurons in the first layer, 2 neurons in the second and 1 in the last. This code assumes that the first number in the list is the size of the input layer. The weights and biases are randomly generated using `np.random.randn` which returns normally distributed numbers with mean 0 and standard deviation 1. We should note that there are no biases for the first layer hence we use `for y in sizes[1:]` since biases are only used to compute other output layers.
Also note that weights and biases are stored as *lists* of Numpy matrices. For example, `net.weights[1]` is the Numpy matrix of weights that connects the *second layer of neurons to the third layer*. We can denote this matrix as w, where w_{jk} is the weight of the connection between the j^{th} neuron in the second layer and the k^{th} neuron in the third layer. So if there were 4 neurons in the second layer and 5 neurons in the third layer, $w \in \mathbb{R}^{4\times5}$.
Using this notation, the vector of activations is given by

$$a_{i+1} = \sigma(wa_i + b)$$

where a_i is the vector of activations from the i^{th} layer. To obtain the activations for the the $i+1^{th}$ layer, we multiply the previous activations by w, add the bias b and finally applying σ element-wise.
Now we write code to compute the output of a `Network` instance. We'll start by computing σ:

```
def sigmoid(z):
        return 1.0/(1.0+np.exp(-z))
```

`sigma` takes in a Numpy array (or vector) and applies the σ to each element of z. Next we'll define a method for the `Network` class to compute the output for the corresponding input:

```
def feedforward(self, a):
        for b,w in zip(self.biases, self.weights):
                a = sigmoid(np.dot(w,a)+b)
        return a
```

Now we define a method for stochastic gradient descent:

```
def SGD(self, training_data, epochs, batch_size, eta, test_data = None):
        if test_data: n_test = len(test_data)
        n=len(training_data)
        for j in xrange(epochs):
                random.shuffle(training_data)
                mini_batches = [training_data[k:k+batch_size] for k in xrange(0,n,batch_size)]
        for batch in mini_batches:
                self.update_mini_batch(batch, eta)
```

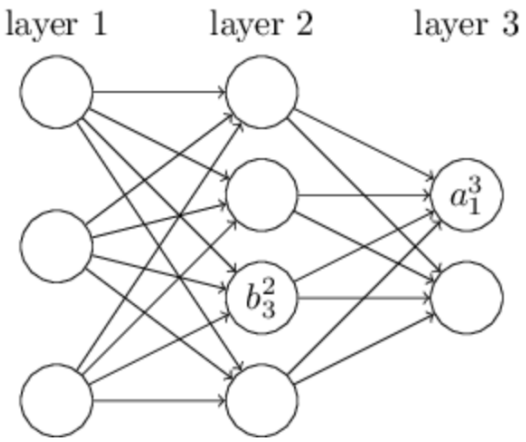
`training_data` is a list of tuples (x, y) representing training inputs and the desired output. In each epoch, the code starts by randomly shuffling the training data and partitions it into appropriately sized mini batches. Then for each batch, we apply one step of stochastic gradient descent. This is done by the function `update_mini_batch` which updates the weights and biases according to a single iteration of stochastic gradient descent using just the data in `mini_batch`. Here's the code for `update_mini_batch`:

```
def update_mini_batch(mini_batch, eta):
        nabla_b = [np.zeros(b.size) for b in self.biases]
        nabla_w = [np.zeros(w.size) for w in self.weights]
        for x,y in mini_batch:
                delta_nabla_b , delta_nabla_w = self.backprop(x,y)
                nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
                nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
                self.biases = [b-(eta/len(mini_batch))*nb for b,nb in zip(self.biases, nabla_b]
                self.weights = [w-(eta/len(mini_batch))*nw for w,nw in zip(self.weights, nabla_w]
```

The `backprop` method is an algorithm that computes the gradient of the cost function. So `update_mini_batch` works by computing the gradient for each example in `mini_baatch` and updating `self.weights` and `self.biases` appropriately.

Backpropagation

In our code above, we utilize an algorithm called *backpropagation*. This is the work horse in many neural networks and we'll take some time to discuss the details of it. First, some notation. Let w^l_{jk} denote the weight of the connection between the k^{th} neuron in the $l-1^{th}$ layer and the j^{th} neuron in the l^{th} layer. Note our choice of ordering here with j, k. Similarly, the bias of the j^{th} neuron in the l^{th} layer is denoted by b^l_j. And lastly, a^l_j is the activation of the j^{th} neuron in the l^{th} layer. An example: The activation of



the j^{th} neuron in the l^{th} layer is given by

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right).$$

We can write this expression using matrices. Let w^l be the matrix of weights for layer l, b^l be the vector of biases for layer l and a^l be the vector of activations for layer l. So we can write

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

The reason why we choose the ordering j, k is that if we were to use the more natural ordering, we would need to transpose the weight matrix. Let's also define $z = w^l a^{l-1} + b^l$ which we'll call the weighted inputs to neurons in layer l. This will be useful later.

Definition. Let $x, y \in R^n$. The Hadamard product of x and y, $x \odot y$, is defined to be

$$x \odot y = (x_1 y_1, x_2, y_2, \ldots, x_n y_n).$$

This is essentially element wise multiplication.

Imagine there is a demon in our network sitting at the j^{th} neuron in the l^{th} layer. This demon adds a little Δz_j^l to the neuron's weighted input. This little change will propagate through the rest of the network causing a total change of

$$\frac{\partial C}{\partial z_j^l} \Delta z_j^L.$$

Definition. We define the error of the j^{th} neuron in the l^{th} layer by

$$\epsilon_j^l = \frac{\partial C}{\partial z_j^l}.$$

We'll denote the vector of errors of neurons in layer l by ϵ^l.

We'll now introduce the four equations behind backpropagation.

1. **Equation for the error in the output layer**. The components of the vector of errors in the output layer is given by
$$\epsilon^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L) \tag{1}$$
Here $\frac{\partial C}{\partial a_j^L}$ measures how fast the cost is changing as a function of the j^{th} activation. For example, if C does not depend much on a particular neuron, the this quantity will be small. In matrix form we express (1) as
$$\epsilon^L = \nabla_a C \odot \sigma'(z^L)$$
where $\nabla_a C$ is the gradient of C with respect to the activation vector a^L. Recall the shape of the sigmoid function. The graph becomes very flat when $\sigma(z_j^L)$ is 1 or 0. When this happens, $\sigma'(z_j^L) \approx 0$. So the weight in the final layer will learn slowly when the activation of the output neuron is really low or really high. In this case, we say that the output neuron is saturated and the weight has stopped learning or is learning slowly.

2. **Recursive equation relating the errors in one layer to the errors in the next**. In particular,
$$\epsilon^l = ((w^l)^t \epsilon^{l+1}) \odot \sigma'(z^l) \tag{2}$$
Multiplication by the transpose of the weight function essentially moves us "backward" giving us a sense of the error of the output in layer l. Finally, the Hadamard product gives us the error of the weighted input in layer l. This combined with (1) allows us to compute the error of for any layer of the network. Note that ϵ^l is likely to be small when the neuron is near saturation.

3. **Equation for rate of change of cost with respect to any bias**. In particular,

$$\frac{\partial C}{\partial b_j^l} = \epsilon_j^l. \tag{3}$$

This value can be computed using (1) and (2). This is also denoted as

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

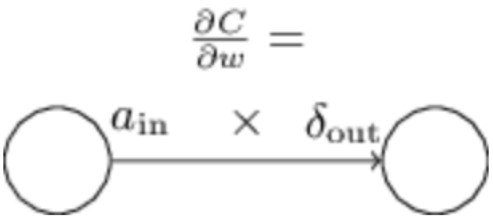
as long as it is understood that b and ϵ are the bias and error of the same neuron.

4. **Equation for rate of change of cost with respect to any weight**. In particular,

$$\frac{\partial C}{\partial w_{jk}^j} = a_k^{l-1}\epsilon_j^l \tag{4}$$

Alternatively, we can write (4) as

$$\frac{\partial C}{\partial w} = a_{in}\epsilon out.$$



One nice interpretation of this is that if the activation is small, the change in the cost is small. In this case, we say that the weight learns slowly.

Generally, we've seen that a weight will learn slowly if the input neuron is low-activation or the output neuron is saturated, i.e. is high or low activation. Note that these equations are not restricted to just σ activation functions. They can be generalized to other functions possibly with more desirable properties.

Backpropagation Algorithm

1. **Input:** Set the corresponding a^1 for the input layer.
2. **Feedforward:** For $l = 2, 3, \ldots, L$ compute the weighted input

$$z^l = w^l a^l + b^l$$

and activation

$$a^l = \sigma(z^l).$$

3. **Output Error:** Compute the vector

$$\epsilon^L = \nabla_a C \odot \sigma(a^L).$$

4. **Backpropagate:** For $l = L-1, L-2, \ldots, 2$ compute the error

$$\epsilon^l = ((w^{l+1})^t \epsilon^{l+1} \odot \sigma'(z^l).$$

5. **Output:** The gradient of the cost function is given by

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\epsilon_j^l$$

and

$$\frac{\partial C}{\partial b_j^l} = \epsilon_j^l.$$

We are now ready to implement `backprop`:

```
def backprop(self,x,y):
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        activation = x
        activations = [x]
        zs=[]
        for b,w in zip(self.biases,self.weights):
                z = np.dot(w,activation)+b
                zs.append(z)
                activation = sigmoid(z)
                activations.append(activation)
        epsilon = self.cost_derivative(activations[-1],y)*\sigmoid_prime(zs[-1])
        nabla_b[-1] = epsilon
        nabla_w[-1] = np.dot(epsilon,activations[-1].transpose())
        for l in xrange(2,self.num_layers):
                z = zs[-l]
                sp = sigmoid_prime(z)
                epsilon = np.dot(self.weights[-l+1].transpose(),epsilon)*sp
                nabla_b[-l] = delta
                nabla_w[-l] = np.dot(delta,activations[-l-1].transpose())
        return (nabla_b,nabla_w)
def cost_derivative(self,output_activations,y):
        return (output_activations-y)
def sigmoid(z):
        return 1.0/(1.0+np.exp(-z))
def sigmoid_prime(z)
        return sigmoid(z)*(1-sigmoid(z))
```

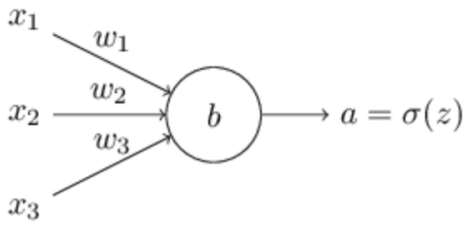
The code returns a tuple `(nalba_b,nabla_w)` which are layer-by-layer lists of Numpy arrays. This represents the gradient of the cost function.

Improving the Neural Net

Backpropagation provides us with a good way to compute the gradient of the cost function but it is just a basic first step. There are multiple ways to improve the learning of our network. These things include a different cost function, regularization methods, weight initialization, etc.

The Cross-Entropy Cost Function

Assume we have n training inputs $x_1, \ldots, x_n$ each with corresponding weights $w_1, \ldots, w_n$ and a bias b. Here, $z = \sum_j w_j x_j + b$ is the weighted sum of inputs.



Definition. The cross-entropy cost function for a single neuron is defined by

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

where y is the desired output of the individual neuron.

Note. The cross-entropy function is indeed a cost function since it is

1. non-negative; each individual term is non-negative and the negative in front makes the whole sum positive and
2. if the actual output is close to the desired output, the cost is close to zero.

One can show that

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

This tells us that the rate of learning is controlled by $\sigma(z) - y$, the error in the output. The larger the error, the faster the neuron will learn. This eliminates the slow-down in learning from $\sigma'(z)$ in the quadratic cost function. Similarly,

$$\frac{\partial C}{\partial b} = \frac{1}{n} \sum_x (\sigma(z) - y).$$

Put shortly, this allows our neuron to learn very fast when the output is very far from the desired output.

Definition. Let $y_1, \ldots, y_n$ denote the desired values of the output neurons, $a_1^L, \ldots, a_n^L$ be the actual output values and $x_1, \ldots, x_n$ be the inputs. Then the cross entropy is defined by

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

Cross-entropy is nearly always better than the quadratic cost function, provided the output neurons are sigmoid neurons.