Notes on Neural Networks

Trevin Gandhi

August 2015

1 Perceptrons

A "perceptron" is a very basic model of a neuron. It inherently has its own vector of weights w, where each element represents the weight associated with the corresponding decision. A perceptron also has a bias, b, used to make the decision. The perceptron takes as input a binary vector x representing the conditions relevant for each decision. Then the output of the perceptron, which is a single "yes" or "no" is as follows:

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \tag{1}$$

Cool things: You can layer perceptrons for more complex decision making. You can also use perceptrons to imitate logic gates.

2 Sigmoid Neurons

Sigmoid neurons are similar to perceptrons in that they take in a vector x as input, have weights and a bias, and produce an output decision. The key conceptual difference, however, is that sigmoid neurons are designed so that small changes in their weights and bias only have small effects on their output decision. This allows sigmoid neurons to be continuously "fine-tuned" towards optimal decision-making.

To manifest this conceptual change, sigmoid neurons can take a vector x where each element is in the interval $[0, 1]$, not just from the set $0, 1$ like it was with perceptrons. Additionally, sigmoid neurons do not output a binary decision - they output a sigmoid (or logistic) function.
The basic sigmoid function is defined by:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}} \tag{2}$$

meaning that a sigmoid neuron would output:

$$\text{output} = \frac{1}{1 + e^{-\sum_j w_j x_j - b}} \tag{3}$$

and in vectorized form:

$$\text{output } = \frac{1}{1+e^{-(w\cdot x)-b}} \tag{4}$$

Because the logistic function is a smooth function ranging from 0 to 1 (as opposed to a step function), it results in small changes being made to the output when small changes are made to the weights or bias. The change in output, Δoutput, can be approximated by:

$$\Delta\text{output} \approx \sum_j \frac{\partial\text{output}}{\partial w_j}\Delta w_j + \frac{\partial\text{output}}{\partial b}\Delta b \tag{5}$$

2.1 Exercises

1. Question: Suppose we take all the weights and biases in a network of perceptrons, and multiply them by a positive constant, $c > 0$. Show that the behaviour of the network doesn't change.
   Answer: The output for a perceptron is:

   $$\text{output } = \begin{cases} 0 & \text{if } w\cdot x + b \leq 0 \\ 1 & \text{if } w\cdot x + b > 0 \end{cases}$$

   and if scaled by c,

   $$\text{output } = \begin{cases} 0 & \text{if } c(w\cdot x + b) \leq 0 \\ 1 & \text{if } c(w\cdot x + b) > 0 \end{cases}$$

   and because $c > 0$, we can say

   $$\text{output } = \begin{cases} 0 & \text{if } w\cdot x + b \leq 0 \\ 1 & \text{if } w\cdot x + b > 0 \end{cases}$$

   and thus the behavior of the perceptron network does not change, as each perceptron is outputting the same thing as it did pre-scaling.

2. Question: Suppose we have the same setup as the last problem - a network of perceptrons. Suppose also that the overall input to the network of perceptrons has been chosen. We won't need the actual input value, we just need the input to have been fixed. Suppose the weights and biases are such that $w\cdot x + b \neq 0$ for the input x to any particular perceptron in the network. Now replace all the perceptrons in the network by sigmoid neurons, and multiply the weights and biases by a positive constant $c > 0$. Show that in the limit as $c \to \infty$ the behaviour of this network of sigmoid neurons is exactly the same as the network of perceptrons. How can this fail when $w\cdot x + b = 0$ for one of the perceptrons?

Answer: The output of a sigmoid neuron is given by:

$$\text{output } = \frac{1}{1 + e^{-(w \cdot x) - b}}$$

and thus the output of the scaled sigmoid neuron is:

$$\text{output } = \frac{1}{1 + e^{-c(w \cdot x + b)}}$$

Therefore, when we take $\lim_{c \to \infty} e^{-c(w \cdot x + b)}$, we can rewrite that as $e^{-\infty(w \cdot x + b)}$.
We now have two cases:
1) if $w \cdot x + b < 0$, then the negatives cancel and the term becomes ∞.
Thus, the output can be rewritten as:

$$\text{output } = \frac{1}{1 + \infty} \approx 0$$

just like with a perceptron.
2) if $w \cdot x + b > 0$, then the term becomes ≈ 0. Thus, the output can be rewritten as:

$$\text{output } = \frac{1}{1 + 0} = 1$$

again, just like with a perceptron.
And now we can see how this fails if $w \cdot x + b = 0$, because then the output becomes:

$$\text{output } = \frac{1}{1 + 1} = \frac{1}{2}$$

which is not part of the set $0, 1$ that a perceptron draws its output from.

3 The architecture of neural networks

In a neural network, the leftmost layer is called the input layer (containing *input neurons*), the middle layers are called the hidden layers, and the rightmost layer is called the output layer (containing *output neurons*).

4 A simple network to classify handwritten digits

The input layer consists of 784 neurons, each representing a pixel of the 28x28 pixel image of a handwritten digit. The hidden layer consists of a currently undecided n neurons, and the output layer contains 10 neurons - each representing what number the neural network has classified the digit as (i.e. the first output neuron firing would say that the digit is a 0).

5 Gradient Descent

We can define a cost function $C(w, b)$ that represents how close our approximation (what the neural network is outputting) is to the real deal. A simple quadratic cost function is:

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2 \tag{6}$$

In this case, y(x) is the function we are trying to approximate and a is our approximation (the output).
Now, to optimize our neural network, we clearly want to minimize our cost function. We can do this with a method called gradient descent, where we iteratively approach the minimum of a function. Recall the gradient function from multivariable calculus:

$$\nabla F(x_1, x_2, \ldots, x_n) = (\frac{\partial F}{\partial x_1}, \frac{\partial F}{\partial x_2}, \ldots, \frac{\partial F}{\partial x_n})^T \tag{7}$$

We can use this gradient function to approximate ΔF:

$$\Delta F \approx \nabla F \cdot \Delta x \tag{8}$$

To find the minimum, we can simply update x based on the negative gradient:

$$\Delta v = -\eta \nabla F \tag{9}$$

where η is the learning rate. To put this back in terms of our cost function C, we can say that:

$$\Delta w_k = -\eta * \frac{\partial C}{\partial w_k} \tag{10}$$

$$\Delta b_l = -\eta * \frac{\partial C}{\partial b_l} \tag{11}$$

However, there is an inherent flaw in this approach: because ∇C is defined as:

$$\nabla C = \frac{1}{n} sum_x \nabla C_x$$

∇C_x would need to be computed for each training input x - which can be computationally inefficient when x is very large.
Thus, we have the idea of *stochastic gradient descent*, which randomly selects a *mini-batch* of training inputs to calculate an approximation of ∇C. If m denotes the size of the mini-batch and n the number of training inputs, then we know:

$$\lim_{m \to n} \frac{sum_{j=1}^m \nabla C_{X_j}}{m} = \frac{\sum_x \nabla C_x}{n} = \nabla C \tag{12}$$

The process works by choosing a mini-batch, training the model, choosing another mini-batch, further training the model, and so on until all training inputs have been used (a *training epoch* has been completed).