Insights Into Machine Learning

Prathik Naidu
prathik.naidu@stanford.edu

June 10, 2017

1 Introduction and Motivation

Conventional computer programming is a straightforward, linear task. You take a problem, analyze that problem, understand its smaller components, and write code to tell the computer how to solve those components. While seemingly simple, this approach has helped speed up tasks that would normally be near impossible for a human to tackle. However, there were still numerous tasks that conventional programming methods weren't good at (computer vision, classification, pattern recognition, etc.) largely because these traditional coding approaches rely upon observational knowledge about a problem from a humans perspective. However, in many situations, datasets lack the immediate clarity and intuition by even the best computer programmers in order to create these "linear" solutions.

The advent of *machine learning* has changed how we think about solving challenging computational problems by teaching a computer how to recognize patterns and relationships that is normally difficult for a programmer to discover. The idea is that these machine learning algorithms can be deployed on large-scale datasets to rapidly determine these patterns and make predictions about a dataset giving numeric information.

The goal of this review/notes sheet/whatever you want to call it is to highlight the fundamentals and background of machine learning with a particular focus on neural networks. Plus, I have familiarity with machine learning concepts from previous projects, but I wanted to delve deeper into the math and theory while also exploring deep neural networks (because apparently their becoming super popular or something and I'm out here still going strong with my trusty random forest and ensemble methods).

2 Types of Neurons

2.1 Perceptron

The simplest (and generally unused) type of neuron for a neural network is a perceptron, which essentially takes a bunch of inputs, does some math to synthesize those inputs together, and then produces a single output of either 0 or 1. Pretty straightforward!

From a more mathematical perspective, the perceptron takes in various inputs, denoted as x_i, where $i \geq 0$. Each of these inputs into a single perceptron gets multiplied by a predefined weight denoted as w_i. This makes sense because certain inputs for a particular variable should be weighted more than others given a situation. For example, when trying to decide if I should play basketball, factors (inputs) that I would consider are the weather outside, if my friends are also coming, if my ankle isn't hurting anymore, etc. But for me, I would probably weigh if my friends are coming or not as the most important factor for playing basketball (because nobody likes to shoot hoops alone).

Ultimately what we end up doing with the perceptron is multiplying the predefined weights with the inputs into the perceptron and comparing that to a particular threshold. If the weighted sum is less than this threshold, the output is 0, but otherwise, the output is 1. This is the overall formula for a perceptron:

$$output = \begin{cases} 0, & \text{if } \sum w_i x_i \leq threshold \\ 1, & \text{if } \sum w_i x_i > threshold \end{cases}$$

Also, just to introduce some common terminology, we can rewrite the formula as follows and replace the threshold with what is known as a neuron's *bias* denoted by $b = -threshold$.

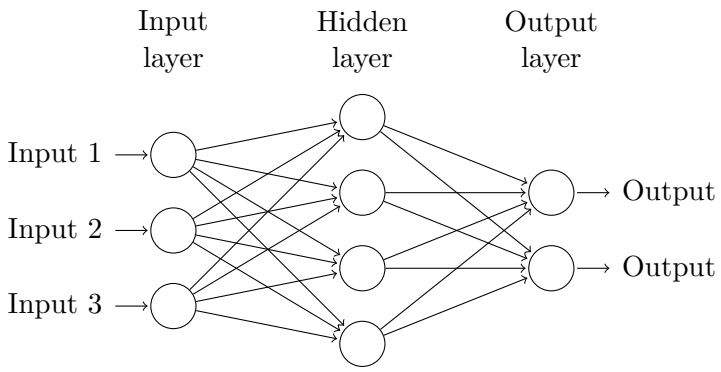
$$output = \begin{cases} 0, & \text{if } \sum w_i x_i + b \leq 0 \\ 1, & \text{if } \sum w_i x_i + b > 0 \end{cases}$$

2.2 Sigmoid Function

One of the key limitations of a perceptron is that its output is completely binary, meaning that despite any small changes in the inputs, the output will always be either 0 or 1. In practice, this wouldn't make sense especially since we want to account for these small changes in the inputs with changes in the output of a neuron. Instead of a perceptron, a sigmoid neuron can be more effective since it is able to output non binary values as outputs (range of numbers between 0 and 1).

The sigmoid functions takes in inputs x_1, x_2, ... and computes the following function: $\sigma(z) = \frac{1}{1+e^{-z}}$, where z is $\sum w_i x_i + b$. This application of the sigmoid function makes sense because as z becomes large, the output approaches 1 and when $z$becomes small, the output of the neuron approaches 0. The sigmoid function is known as an *activation function*.

Now that we've defined the types of neurons, we can combine this knowledge together into what is known as a neural network, where neurons are connected to one another and ultimately receive inputs and calculate outputs based on this activation function.



3 Determining the Weights and Biases for a Neuron

One key point about neural networks is that we don't automatically set the weights and biases for the neurons to their perfect values for optimal prediction. Instead, for a specific problem, we want the neural network to automatically adjust and recalculate the weights and biases to minimize the difference between the true outputs versus the predicted values. Based on this idea, we can define what is known as a cost (or loss) function:

$$C(w,b) = \frac{1}{2n} \sum \parallel y(x) - a \parallel^2$$

In this function C(w,b), w is the collections of weights, b is the biases, n is the number of inputs, a is the outputs from the network, and $y(x)$ is the true values for the outputs. By using this cost function, it is simpler to determine how changes in weights and biases values can alter the quadratic difference (mean squared error) between predicted vs. true outputs.

Fundamentally, the concept behind this cost function, as mentioned before, is that we want to minimize this value. The cost function itself is dependent on any sort of variables in the dataset and in the case of these notes, we'll define those variables as $v_1, v_2, v_3, \ldots$.

Gradient descent works by finding the /textitgradient of the cost function, which essentially consists of the partial derivates for each variable used in the inputs to the neural network.

$$\nabla C = \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}, \ldots \right)$$

By finding this gradient vector, we can continuously update the point-based position and then recalculate the gradient until the cost function is minimized.

4 Backpropagation

Great, so now we know how to get our neural network to learn weights and biases by calculating a gradient vector, but I still haven't mentioned exactly how to do this step, known formally

as *backpropagation*. I'll probably come back and fill in this section later with the math behind backpropagation but here is a brief explanation of the fundamentals:

Without going into the specific calculus-level details, this algorithm functions by first doing a standard feedforward approach, which calculates the activation values for each layer of the neural network. However, one effect of this is that changes in the weights and biases of neurons in various layers can ultimately impact the overall cost function. As a result, an error is calculated at the last output layers, and this error is backpropagated for all the previous layers in order to determine the gradient of the cost function.

This might sound a bit confusing without seeing the math, but from a bigger picture, imagine that we change the weight of a single neuron in a layer. Ultimately, that changes the output activation of that neuron, which subsequently changes all the activations of neurons in the next layer, which then impacts all the neurons int he following layer, and so on. Intuitively, this algorithm can determine how certain changes in weights are carried through and impact subsequent layers of the neural network, which from there changes the cost function. By backpropagating, we can understand the impact of these changes, and calculate the gradient of the cost function with respect to alterations in the weights and biases of neurons in the network.