Lecture 16: Introduction to Neural Networks

Instructor: Aditya Bhaskara *Scribe: Philippe David*

CS 5966/6966: Theory of Machine Learning

March 20th, 2017

Abstract

In this lecture, we consider Backpropagation, a standard algorithm that's used to solve the ERM problem in neural networds. We also discuss the use of Stochastic Gradient Descent (SGD) in Backpropagation. The use of SGD in the neural network setting is motivated by the high cost of running back propagation over the full training set.

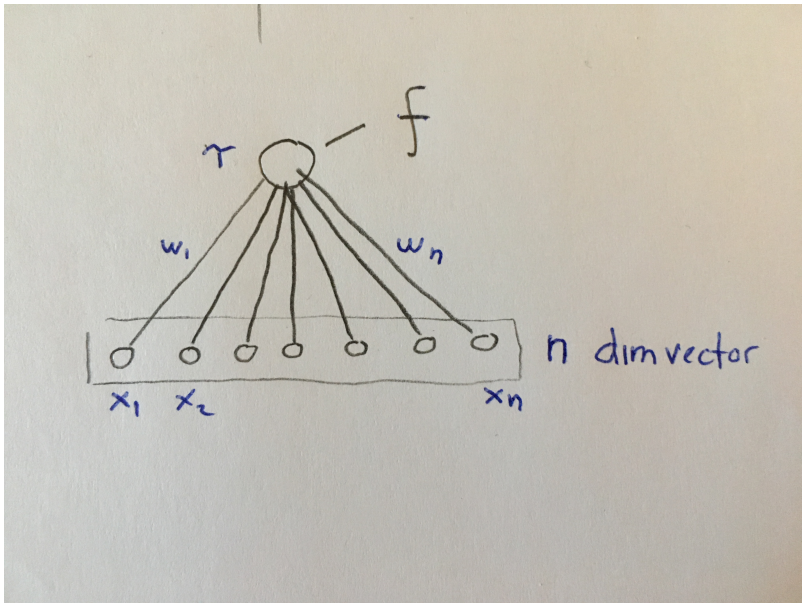
1 Recap: The Expressive Power of Neural Networks

In previous lecture, we started formalizing feed-forward neural networks. These networks take a collection of inputs, pass these inputs through a series of layers composed of neurons, and use this mapping to produce an output. We then defined what threshold networks are: they are basically a collection of neurons in every layer where each neuron computes a weighted threshold function. We then went on to study the expressive power of these neural networks. Namely, we looked at what type of functions can be implemented using a neural network. We looked at two theorems that related to this question. The first said that a feed-forward network with a single hidden layer can compute any arbitrary functions - this is known as universal approximation theorem. We then said that while this may look powerful, there are many functions that require an exponential number of inputs to be approximated. We proved this by saying VC dimension of the class of all such networks (all networks with m edges, and n neurons) is $O((m+n)log(m+n)$. While we didn't prove this theorem, we used it to conclude the previous theorem. What this theorem states is that if you are interested in learning neural networks that just have at most m edges, all you need is roughly so many samples. If a class of functions has VC dimension d, then to learn earn the best network need VC dimension/ϵ^2 samples. This is what is known as the Fundamental Theorem of Learning Theory. Nevertheless, these is a caveat: All this tells you is that if you had a bunch of samples and if you had a procedure that could find the best Empirical Risk Minimizer (ERM) for these examples you could also get the best network. VC theory tells us that VC dimension/ϵ^2 examples + an algorithm for solving ERM implies efficient learning. It turns out, however, for Neural Networks, this second part if very hard. We went on to show that the the ERM problem for a two-layer neural network with just 3 neurons in each layer is NP-Hard as it can be reduced to coloring a k-colorable graph with 2k - 1 colors for general k which is known to be NP-Hard.

We will prove a simplified version of this theorem in assignment.#3

2 The Backpropagation Algorithm

The backpropagation algorithm — the process of training a neural network. The key to training a neural network is finding the right set of weights for all of the connections to make the right decisions. This algorithm work to solves the above problem in a very similar manner to gradient descent as it iteratively progressively work their way towards the optimal solution. Clearly, there are no guarantees for it because the problem is NP-Hard, but this is a very good heuristic approach. We will start with a simple examples. Suppose you have a one layer neural network



What is the training problem? The training problem is that you want to minimize the empirical risk given by the following equation:

$$\sum_i (f(x^{(i)}) \neq y_i) \tag{1}$$

where the inputs are $x^{(i)}$ and y_i correspond the labels for these inputs. As this is just an indicator function, and we want something to be smooth, the above is not the optimal. Nonetheless, there are various proxies for this and a commonly used one is the squared loss:

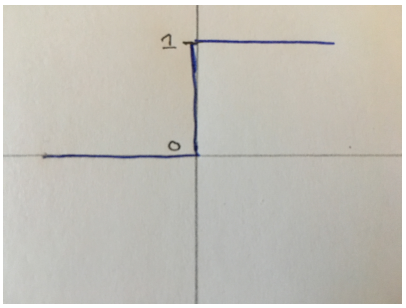
$$loss = \sum_i (f(x^{(i)}) - y_i)^2 \tag{2}$$

The network is now very simple, we have $w_1, w_2, \ldots, w_n$. f is a threshold function:

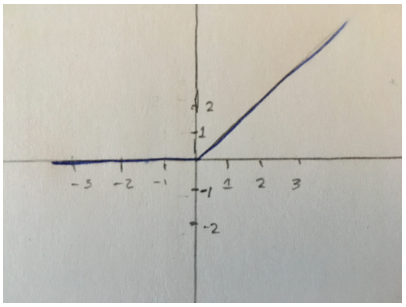
$$f = \sigma(\sum_j (w_j x_j^{(i)})) \tag{3}$$

where i corresponds to the input point and j is j_{th} index of the weights. From here, gradient descent appears to be the natural idea to find the best. To do this you start with some w^0 and set $w^{t+1} = w^{(t)} - \gamma \bigtriangledown f(w^{(t)})$. The issue with

this is with computing the gradient of $f(w^{(t)})$. We don't have a contentious gradient as the gradient of a simple threshold function is undefined.



A commonly used alternative is the rectified linear unit activation function which grows linearly to 1. This function is also non-linear (despite its name). If we only use linear activation functions in a neural network, the output will just be a linear transformation of the input, which is not enough to form a universal function approximator.



Now we know what f is, we can run gradient descent. Using the chain rule:

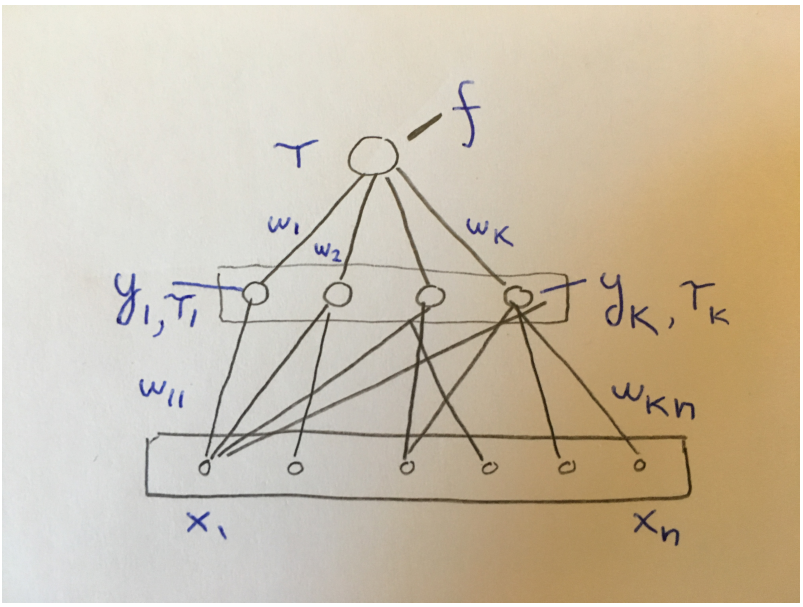
$$\frac{\partial loss}{\partial w_j} = \sum_j 2(f(x^{(i)}) - y_i)(\sigma'(\sum_j w_j x_j^{(i)})x_j^{(i)} \tag{4}$$

Now explore an stochastic gradient descent (SGD) version of this. What do you mean by this? Remember that the loss being a sum of terms is something we often encounter. In such a setting, computing the loss gradients is complicated so we will just take a random example and move along the gradient for that example. This is the common way of training neural networks, you don't look at the loss summed over everything. But you take a random training examples and you update according to the gradient for that example. This gives us the following algorithm:

- For $t = 1...T$.
- Pick an index $i_t within N$.
- $w^{(t)} = w^{(t+1)} - \gamma \bigtriangledown l_{i_t}(w^{(t)})$.

where $l(w^{(t)}) = \sum_i l_i(w^{(t)}) = \sum_i (f(x^i) - y_i)^2$. In SGD we iteratively update our weight parameters in the direction of the gradient of the loss function until we have reached a minimum. Unlike traditional gradient descent, we do not use the entire dataset to compute the gradient at each iteration. Instead, at each iteration we randomly select a single data point from our dataset and move in the direction of the gradient with respect to that data point.

3 Computing weight vector derivative in 2-layer neural networks



As we have seen, for a one layer NN, computing this derivative is very simple. For the 2-Layer NN, this computation becomes more complicated. The first question that naturally arises are what parameter do we want learn if we want to learn the best neural network of this form? in order to estimate this loss function we need to have τs for every node, τ_i and $\tau(top - layer)$. Where τ represents the thresholds and the weights between nodes are represented by u_i and w_{ij}. To run SGD, you need to pick a random data point, and move along the gradient.

$$f = \sigma \sum_{j=1}^{k} u_j y_j \tag{5}$$

$$\frac{\partial f}{\partial u_j} = \sigma'(\sum_{j=1}^{k} u_j y_j) y_j \tag{6}$$

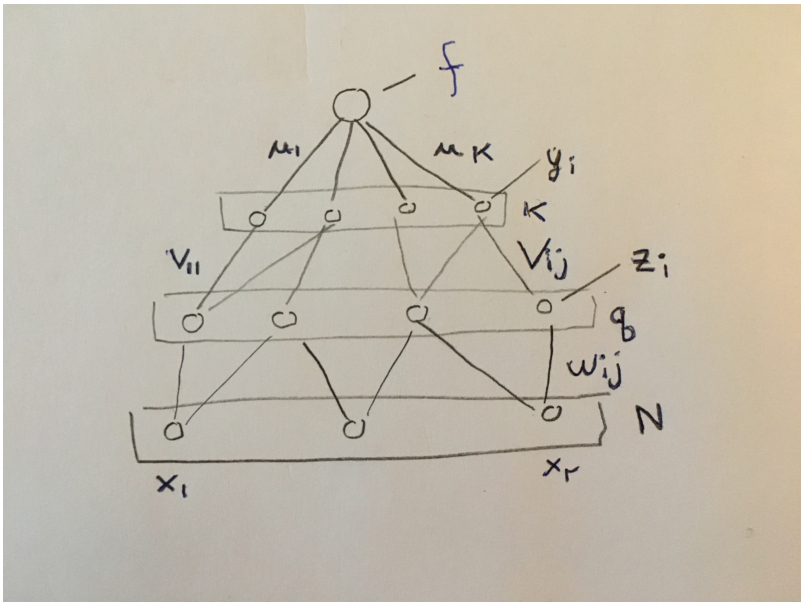
where σ is the threshold function. But what is

$$\frac{\partial f}{\partial w_{ij}} = \sigma'(\sum_{j=1}^{k} u_j y_j) \frac{\partial(\sum_{l=1}^{k} u_l y_l)}{\partial w_{ij}} = \sigma'(\sum_{j=1}^{k} u_j y_j) u_i \tag{7}$$

$$y_i = \prime(\sum_{l=1}^{n} w_{il} x_l), x_l = inputs \tag{8}$$

$$\frac{\partial f}{\partial w_{ij}} = \sigma'(\sum_{j=1}^{k} u_j y_j) u_i \sigma'(\sum_{l=1}^{n} w_{ij} x_j) x_j = \sigma'(f)\sigma'(y_i) u_i x_j \tag{9}$$

4 Computing weight vector derivative in 3-layer neural networks



(10) $$\frac{\partial f}{\partial w_{ij}} = \frac{\partial \sigma(\sum_{j=1}^{k} u_j y_j)}{\partial w_{ij}}$$

Taking this derivative will result in multiple summations.

(11) $$\frac{\partial f}{\partial w_{ij}} = \sigma'((\sum_{j=1}^{k} u_j y_j)) \sum_{l=1}^{k} u_j (\frac{\partial y_l}{\partial w_{ij}})$$

$\frac{\partial f}{\partial w_{ij}}$ is a weighted sum of these y_l's with respect to w_{ij} where $y_l = \sigma((\sum_{t=1}^{q} v_{lt} z_q))$ which implies that $\frac{\partial y_l}{\partial w_{ij}} = \sigma'(y_l) * v_{li} * \frac{\partial z_i}{\partial w_{ij}}$