Lecture 19: Neural networks: Representation, Optimization

Instructor: Aditya Bhaskara *Scribe: Manoj Marneni*

CS 5966/6966: Theory of Machine Learning

April 3rd, 2022

Abstract

This lectures recaps Deep Neural Network functions and discusses how do you train Neural Networks using gradient descent algorithm. Also It is shown how complex calculating weights using gradient descent can get even for 2 layered networks, which was solved using Back propagation algorithm.

1 Artificical/Deep Neural Network Recap

Neural Networks are a class of functions that is inspired by how our brain perceive things. In the initial layer we'll have input and then this input feeds into detectors of basic features and more complex features are built on top of it.

Definition: A layered "circuit" that takes a vector of input features x, produces output y = $F_r \circ F_{r-1} \circ .... \circ F_1(x) \circ$, where each F_i is a function of the form $F_i(z)$ = $\sigma(Az + b)$, for some activation function $\sigma()$.

Some common activation functions are Threshold, ReLU, Tanh, Sigmoid, etc

2 Theory of Deep Learning - Recap

Most of the theory of deep learning can be divided into three kinds of studies. The three broad directions are

1. **Expressibility:** What kind of functions can be obtained using a Deep Neural Network (DNN). This is used to define the complexity of the functions and There is no focus on training in this.
2. **Training Complexity Training dynamics for Gradient Descent (GD) and variants:** The studies in this part is focused on how do you train GD type of algorithms and how do you analyse them. What of kind solutions does it converge to? In general, this involves studying, Can the Empirical Risk Minimization (ERM) for Neural Networks be solved efficiently? and what guarantees are possible?
3. **Generalization:** This branch is mainly focused on what kind of generalisation bounds can we prove? VC-Dimension is one way to capture generalisation. The question here is, if you get m training examples, can you ensure that the test error, assuming same distribution, is less than $f(m)$?, usually $f(m)$ will be $\frac{1}{\sqrt{m}}$.

A general thing to keep in mind is that you can have easy answers for all these questions but they are unsatisfactory for realistic settings.

In Expressibility basics, we saw Barron's theorem and Cybenco's Universality expression. If we have d dimensional function, Inorder to represent it using a neural network we need exponential in d width. This true for classical universal approximation result. But it is not needed for barron's theorem, the niceness condition allows it to not depend on dimensionality. Therefore the curse of dimensionality is only applicable for Cybenco's universality approximation.

3 Why Deep Networks? - Recap

Even very shallow networks like depth 2 networks are universal i.e, they can express any function given large enough width. But we don't use it because, if we don't use depth then the result requires extremely large width and we don't have very good generalisation properties either. That is why we need deep networks.

In previous lecture, we formalised this general practical intuition, that depth allows for more intuitive, more high level feature, while width is useful for brute force memorisation. Some additional motivation for depth is, if you want to work with high dimensional data then shallow networks are not very useful because if you will need extremely high width.

Power of Depth: In the previous lecture we saw a theorem, which states that there exists a network of depth k^2 and $O(1)$ width, that computes function f, with the property that any network of depth k that approximates function f requires width of $>\approx 2^k$. The proof of this relied on three basic observations, such as

1. A function with 2^{k^2} oscillations can be output by a depth k^2 network with 3 neurons per layer.
2. Any function that is the output of width w, depth k ReLU network is composed of atmost $(2w)^k$ piecewise linear "pieces".
3. A picewise linear function with fewer than $2^{\frac{k^2}{2}}$ pieces cannot even approximate the function from (1) to an error of $\approx \frac{1}{4}$.

This theorem restricted in applicability, this is not known to be true even for sigmoid or sinewave. It is only known to be true for piecewise polynomial functions. You can expect this to be true for other classes of functions as well.

Moral:

1. Depth allows to capture complex, highly oscillating patterns.
2. Width typically allow you to capture different behaviour in different regions of space.

Finding the right network for an application is a difficult problem to solve, and there is no general solution to that. Some attempts were made to solve this problem like network architecture search but they are applicable to very specific domains. There are areas like physics informed machine learning where you need different types of network architectures than what you would have

normally. People used Hebbian principle as a guiding principe to develop neural networks for a while, where you don't start with an architecture, but you build an architecture based on activations of the previous layer. You can show that this sort of thing works under some assumptions, but it works pretty sub-optimal compared to known networks.

4 Neural Network Training

Supervised Learning of NN: Given data $(x_1, y_1), (x_2, y_2).....(x_n, y_n)$ where x is the input and y is the label, from some distribution D, find a hypothesis h that minimizes the risk (empirical risk is what we care about).

We need a loss function to find such hypothesis, we can use standard metrics like sqaured loss, cross entropy, etc. Let $h_w(x)$ is the output of neural network, where w are the weights of NN. We need to find w, so as to minimise $\sum_{i=1}^{n}(h_w(x_i) - y_i)^2$. Even for this squared loss, ERM problem is NP-Hard because h is non-linear. Though it is NP-Hard it does not rule out the usefulness of NN. ERM assumes that you are given some data and you need to find the best possible solution, this problem is hard, but in practice, we can get more data if required. Therefore we need to consider NP-Hardness results of the problems in this space, with a pinch of salt.

5 Common Algorithm - Gradient Descent

We solved ERM problems with data $(x_1, y_1), (x_2, y_2).....(x_n, y_n)$ and a hypothesis class $h(w, x)$ defined by "weight vector" w and input x.
In linear hypothesis, $h(w, x) =< x, w > +c$
Let's say we have a 2 layered neural network, with $F_1(x) = \sigma(Ax + b)$ and the output $y = \sigma(v^T F_1(x) + b')$. The Weight vector now consists of, $w = (A, v, b, b')$. In general weights are set of all tuple of matrices, biases, etc. All the parameters that are involved in defining the model, these are often also called model parameters.

Now the goal of ERM is to find w, so that the sum of associated losses over training data is minimum (i.e find w such that $\sum_{i=1}^{n} l(h(w_i, x_i), y_i)$, say $G(W)$is minimum). This $G(w)$ is no longer a convex function.

1. If G were convex, Gradient Descent (GD) probably works.
2. In Neural Networks, G is usually not convex i.e, for most choices of activation functions. Minimisng for non-convex function is general a Hard problem, but you can hope that GD still works.
3. GD still finds local optimal. Hope is that if you have sufficient data, GD finds a "good" local optima. In most of modern NN, we try to use same tricks that are developed in the context of convex optimisation, like momentum, regularisation, etc.

5.1 *Landscape analysis of local optima:*

Quality of local optima: This refers to the difference between local optimum and global optimum. There is also a lot of work on finding the **Number of local optima**, it can be very large, but there is not much work on trying to

understand this quantity. Lot of people have tried to show you can have a lots of local optima and most of those local optima are reasonably good. **Lots of Global optima (or nearly global optima):** Some Global optima generalises way better than others. Initially, people used to focus more on the first two metrics, but recently people realised that even though a local optima is close to the global optima, some global optima are way better than others. There is a general belief that the process of gradient descent itself acts as some kind of regulariser. The solution you find by using Gradient Descent is better than other solutions that has zero loss but they don't generalise well.

5.2 *How to do Gradient Descent:*

We start with $w^{(0)}$, then

$$w^{(t+1)} = w^{(t)} - \eta \nabla G(w^{(t)})$$

Even for 2 layered neural network, the gradient descent algorithm can get messy. we have $G = \sum_{i=1}^{n} (h(w_i, x_i) - y_i)^2$. For simplicity let us say $G = (h(w_i, x_i) - y_i)^2$.

$$\nabla G = 2 * (h(w, x) - y) * \nabla h(w, x)$$

$\nabla h(w, x)$ captures how does h change with change in w. In 2-layered neural network we have

$$h(w, x) = \sigma(v^T(\sigma(Ax + b)) + b')$$

This implies ∇h has components along each entry of A, b, b', v. If σ is ReLU $\frac{\partial h}{\partial v_i}$ will be essentially y_i. Similarly you can differentiate with respect to A_{ij} using chain rule as follows,

$$\frac{\partial h}{\partial A_{ij}} = \sigma'(v^T(...)) * \frac{\partial y_j}{\partial A_{ij}} * v_j$$

If you have more than 2 -layers you'll start getting into quadratic computation, high complexity very easily. If you want to see the effect of a weight in layer 1 on the overall function, you can see it impacting all the layers, and its calculation getting very messy. The influence of the weight in layer 1, can be felt throughout the network. Now, if you want to calculate the derivative of h w.r.t weight in layer 1, it will become a messy problem involving all paths. There is a clean way of solving this problem using dynamic programming known as Back propagation algorithm. It introduces new variable that keeps track of gradient with respect to weights in each layer. These are not parameters of the network but some kind of accumulated totals. Using which you can find gradients very efficiently in linear time of the number of parameters.