Lecture #20: Optimization in Neural Nets, Feature Learning

Instructor: Aditya Bhaskara *Scribe: Jiayi Wang*

CS 5966/6966: Theory of Machine Learning

March 29th, 2022

Abstract

This is a summary of the day's lecture. In this lecture, two main topics are discussed, Gradient Descent (GD) algorithm for training neural networks, and the Feature Learning. For GD, its computation costs and resulted training loss are discussed. For Feature Learning, what is 'useful' feature embedding is discussed.

1 Some Preliminaries

Before we proceed to the main topics in this lecture, we will introduce some basics. The definition for neural network is as the following. **Definition** A layered 'circuit' that takes a vector of input feature x, produces $y = F_r \odot F_{r-1} \circ \ldots \circ F_1(x)$, where F_i is a function of the form $F_i(z) = \sigma(Az + b)$, for some activation function $\sigma()$, which acts coordinate-wise.

Common activation functions include Threshold function, Sigmoid function, ReLU and Tanh etc. To train a neural network is to find the parameters with given architectures which can minimize the empirical loss of the given dataset. However, learning weights is a NP-hard problem.

Theorem Given an architecture, it is NP-hard to learn weights, even if classification error is 0 and we just have 3 internal nodes.

But considering the worse case, where the input data points are bad to train, is not reflective of practice. However, if the width is large, with random inputs, we can train the neural network efficiently. One common algorithm used for training is the Gradient Descent, since gradients are easy to compute.

2 Is Gradient Descent (GD) Good?

Gradient descent is a common method to find the weights for neural network. Naive GD takes computation time proportional to $N|w|$ per iteration, where N is the number of data points and $|w|$ is the number of weights. This could be a huge cost since N and $|w|$ could be very large, i.e. thousands of. Therefore, in practice, we often use stochastic gradient descent (SGD) to save time. In SGD, we divide N data points into several batches and we only compute gradients on one batch in one iteration.

Another question is given the whole dataset, can GD result in training error as $\epsilon^* + f(N)$ after N iterations? Here, ϵ^* is the optimal training loss. For convex functions, GD do result in this results with $f(N) = \frac{1}{\sqrt{N}}$. Bur for neural

networks, which are highly non-IID, the answer is no. Then if the network architecture allows for zero error, does GD converge to zero error? The answer is no since this is an NP-hard problem. There is an alternative to GD, method of moments, which is used for shallow networks. It has been shown that this method can converge in exponential time with respect to the number of nodes, depth. And it can learn what GD can't.

However, in the over-parameterized case, where the width of the net is larger than the number of input, GD can result in a good training result. This can be shown in the following theorem.

Theorem [Jacot, Gabriel, Hongler 2018][Arora, et al. 2019] A width proportional to n^3 network with any number of layers trained via GD from random initialization achieves zeros training error.

The key idea behind this theorem is that since the width is so large, parameters do not change much during the training. GD-based training in neural networks could be equivalent to kernel methods (Neural Tangent Kernel) at least with infinite width.

3 Feature Learning

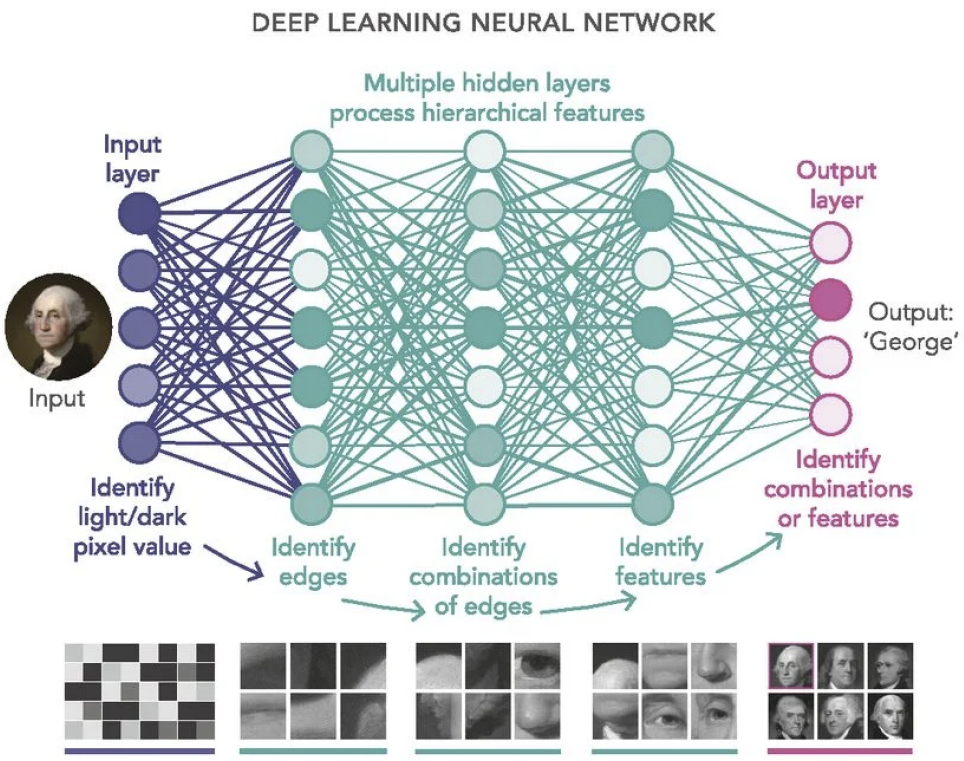

Figure 1: A neural network figure from `https://www.pnas.org/doi/10.1073/pnas.1821594116`.

We show a figure for a neural network for face identification in Figure 1. We can see that for neurons near to the input layer, they can identify some basic features such as the light/dark pixel value, edges. But neurons near the output

can identify some complicated facial features. The features in the output layer are used to output the final classification result. Therefore, we can regard this neural network as performing a linear classification with the final features. It can be seen that what the neural network is doing is 'embedding' the input to the feature space.

Then a question would be, given some input, can we extract some useful features? This is what feature learning does. There are some examples for extracting features. Contrastive learning extract features by comparing which data points are similar, which are different. Or we can discriminate different data points in terms of predicting their classes and classes are 'orthogonal'. There is also a method called Manifold learning. The basic idea is that to describe the dataset with some low-dimensional manifold. And every point on the manifold is defined by $k << N$ parameters.

Then how should we find the feature embedding and what do we expect? The feature embedding can be formalized as the following

$$x \to f(x)$$

where x is the raw data and $f(x)$ is the feature embedding. After discussions, we can summarize our expectations for feature embedding as followings.

- $f(x)$ should be useful to distinguish between classes.
- Coordinates of $f(x)$ should be 'independent' to one another.
- $f(x)$ should be as 'informative' about x as possible.

By 'independent', we mean features are nearly 'orthogonal' to each other. By 'informative', we mean the we recover the exact data by its feature embedding.