Computer Vision
Homework 10: Linear Classifiers and Neural Networks
CS 670, Fall 2019

Name: Kunjal Panchal
Student ID: 32126469
Email: kpanchal@umass.edu

November 22, 2019

Contents

1 Problem 1: Weight Initialization

1.1 Some characteristics of a Two-Layer Neural Network

To optimize a neural network, we can use gradient descent technique where we gradually descent to a local (and hopefully global) minima with some fixed or time-dependent learning rate/ step-size by taking gradient of the point we are currently sitting at.

And that point is initialized by parameters we choose to fire up our neural network with.

Those parameters generally are: weights and biases.

And the objective function we are trying to find minima of, is a loss function which we would want to be minimum and the place where it is minimum, we call parameter values at that point to be optimal.

We can also add regularization, to get a good generalization.

So, in a fee-forward stage, neural networks will make a prediction/-classification based on the current parameter value, in a two-layer case, it depends on parameters of input and the only hidden layer present.

And at the output stage, we would have both original label and predicted label, thus, we can minimize the loss by back-propagating through the layers with the gradient of the objective function. That way, we would be descending towards local minima till the gradient becomes almost zero.

MakeUppercasegradients are computed by chain rule of derivatives, which is given as follows:

$$\frac{dL_n}{dw_i} = \sum_j \frac{dL_n}{dh_j}\frac{dh_j}{dw_i} = -(y_n - v^T h_n)v_i f'(w_i^T x_n)x_n$$

which is dependent on input layer weights $\mathbf{w}$ and hidden layer weights $\mathbf{v}$.

1.2 Initializing all the Weights to Zero

There are two reasons why weights shouldn't be initialized to all zeros, or all same numbers in general:

- Neural Networks tend to GET STUCK AT LOCAL MINIMA, therefore, it is a better idea to initialize them with many different parameter values with enough variance. That "jumping out of a local minima" is not possible if we have initialized all weights to the same value: zeros.

- If the learning process starts with the same weights, then all the neurons will follow the same gradient, same descent and same weight (parameter value) updates. All the neurons in a layer will end up doing the same thing as/ imitating each other. That, of course, wouldn't train the neural network to classify between different inputs.

To elaborate on the second point, if we think about weights as priors in a Bayesian Network, then we will be multiplying them for getting a posterior distribution probability. That multiplication will result in all zeros, if priors are all zeros.

Back-prop identifies the set of updated weights/parameters that minimizes loss with prediction and actual output/objective function; with all weights to zero, we cannot orient gradient descent algorithm in terms of determining the direction of the system as WE WILL BE PLACING OURSELVES ON A SADDLE POINT OF THE PARAMETER SPACE.

Back-propagation based optimization algorithms will usually stop immediately at saddle point. In order to calculate the gradient we multiply deltas with weights and the result will always be zero.

In each iteration of back-propagation algorithm, we will update the weights by multiplying the existing weight by a delta determined by back-propagation. If the initial weight value is 0, multiplying it by any value for delta will change the weights, but they all will change together, the same; which means each iteration has no effect on the descent.

Initializing a network with all 0's in basic fully connected, feed-forward network, is equivalent to having 1 node per layer. Because in hidden

layer, all the nodes in would have exactly the same inputs and would therefore stay the same as each other.

1.3 To Avoid Getting Stuck

Basically, different weights/ parameter values; can break the hidden layer neuron symmetry. There are few techniques which might give satisfactory convergence:

- In RANDOM INITIALIZATION, one thing to note is that if the weights are very high, WX becomes high too and activation function like sigmoid can potentially map it to/ or close to 1, where slope of gradient changes slowly. Too low weights, and the same problem but this time they can get mapped to 0. That's VANISHING GRADIENT problem. Thus, random initialization for [-1, 1] range can obviate exploding or vanishing gradient problem.

- HE INITIALIZATION multiplies random initialization with $\sqrt{\frac{2}{\text{length of weight vector}}}$.

- XAVIER INITIALIZATION multiplies random initialization with $\sqrt{\frac{1}{\text{length of weight vector}}}$. This and above methods serve as good starting points for initialization and mitigate the chances of exploding or vanishing gradients. They set the weights neither too much bigger that 1, nor too much less than 1.

- There are also activation function specific methods:

    - For *tanh()*,
      <Uniform Distribution between [-1, 1]>* $\sqrt{\frac{6}{\#inputs+\#outputs}}$
    - For *sigmoid()*,
      <Uniform Distribution between [-1, 1]>* $4\sqrt{\frac{6}{\#inputs+\#outputs}}$

The idea is that you want to initialize the weights in a way that ensures good forward and backward data flow through the network. That is, you don't want the activations to be consistently shrinking or increasing as you progress through the network.

We can also standardize mean and variance by normalizing the weights.

2 Problem 2: Håstad Theorem

According to the theorem,
k is the depth of the circuit(neural network in our case)
N is the number of inputs
Computation Complexity will be $\exp(N^{\frac{1}{k-1}})$

Suppose, there are $\log(N)$ hidden layers. And one input layer.
$\boxed{\therefore k = \log(N) + 1}$
A binary tree of XOR will have two nodes in input layer as shown in Figure 1. $\boxed{\therefore N = 2}$ So, there will be $\exp(N^{\frac{1}{\log(N)+1-1}})$ nodes.

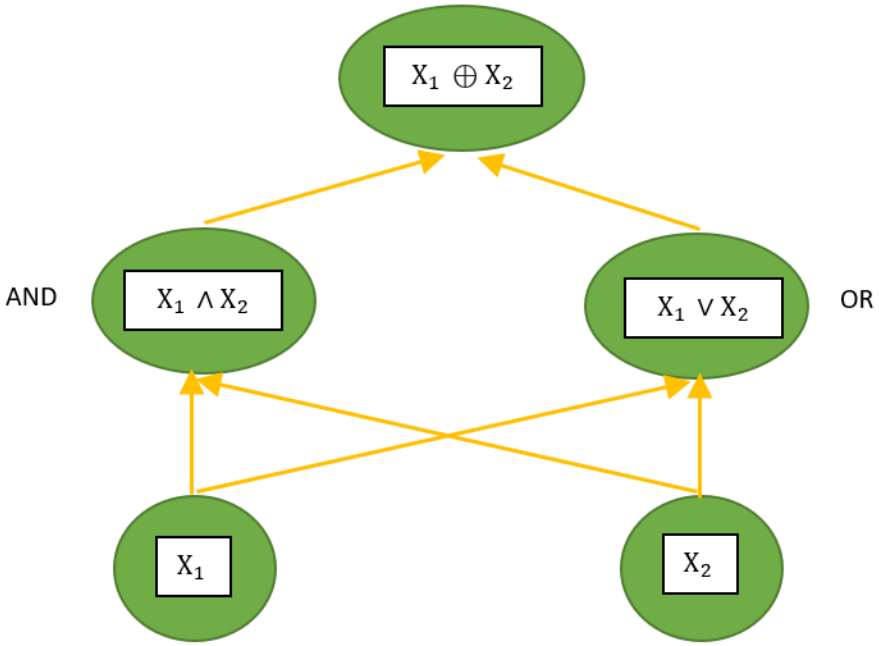

Figure 1: A Binary Tree of XOR

$$\begin{aligned}
\#\text{nodes} &= \exp(N^{\frac{1}{\log(N)+1-1}}) && (1)\\
&= \exp(N^{\frac{1}{\log(N)}}) && (2)\\
&= \exp(2^{\frac{1}{\log(2)}}) && (3)\\
&= \exp(10) && (4)\\
&\approx 28 && (5)
\end{aligned}$$

Hence, we need 28 nodes to compute an N-input, $\log(N)$-hidden layered parity function of a binary XOR tree.

3 Problem 3: Convolution Neural Networks

Input Layer: $N \times N \times 1$ [1-channel image]
Output Layer: $N \times N \times C$ [C-channel image]

3.1 (a) Fully Connected Layer

$$\#Parameters = \underbrace{(\#weights + \#biases)}_{\text{Of input layer. Each Input Layer Node will have one weight and each layer has 1 bias}} \times (\#outputs) \tag{6}$$

$$= (N \times N + 1) \times (N \times N \times C) \tag{7}$$

3.2 (b) Convolution Layer

C filters: $K \times K$
Each of C filters are convolved with the input to produce a feature map of the same size: $K \times K$

$$\#Parameters = \underbrace{(\#weights + \#biases)}_{\text{Of feature map. Each feature map pixel will have one weight and each layer of maps will have 1 bias}} \times (\#outputs) \tag{8}$$

$$= (K \times K + 1) \times (C) \tag{9}$$

References

[1] Subhransu Maji, *Linear Models* https://www.dropbox.com/s/cjl79dzn1cavfzn/lec17_linear_models.pdf?dl=0

[2] Subhransu Maji, *Neural Networks* https://www.dropbox.com/s/vkf5n46xqy7fzct/lec18\%2B19_neural_networks.pdf?dl=0