Hyperparameters and Network Tuning

Vinay Bhaip*

November 2018

1 Introduction

Neural networks are extremely powerful tools in machine learning that are applied to numerous problems. However, there are numerous variable hyperparameters, and there's no real rules on how to decide what they should be. For example, how many different layers should the network have? How many nodes in each layer? What should the learning rate be? Adjusting the hyperparameters is crucial in order to have a successful model.

2 Network Design

The first question that arises when making a model is how many layers and nodes in each layer there should be. To begin, we know there has to be an input layer and an output layer, which are constrained in the number nodes depending on the problem.

For the hidden layers, there is no real guideline to follow. Generally, more hidden layers allows the network to understand more advanced patterns, but takes more time and data. However, this could also mean that the network might be overfitting, which means fitting the model to the specifics of the dataset, not the underlying patterns. In other words, the model could just be memorizing the data. With good hyperparameters, we shouldn't need to worry about this, and as long as we have enough computing power, we should be able to add many layers.

3 Regularization

Now that we know we want to have many layers in our model to have deep learning, how do we stop overfitting? One way we can do this is through regularization. Regularization modifies the loss function and punishes models that get more and more complex. There's 2 main regularization techniques that are used.

*Based off Mihir Patel's lecture

1. L_1 Regularization - Adds the absolute value of all the weights in a network to the error function.

$$L_1 = Error + \lambda \sum_{i=1}^{k} |w_i|$$

2. L_2 Regularization - Adds the squares of all the weights in a network to the error function.

$$L_2 = Error + \lambda \sum_{i=1}^{k} w_i^2$$

Both of these regularization techniques minimize the complexity of the network. L_1 Regularization has been shown to be best at feature classification problems, and L_2 Regularization is good at essentially everything else.

4 Dropout

Another method to regularize networks is dropout. During backpropagation, some percentage of the nodes in each layer will be ignored. This helps to create a more robust network that isn't dependent on just one or a few number of nodes by making sure every node is learning some valuable information. Look at the figure below to see how dropout works.

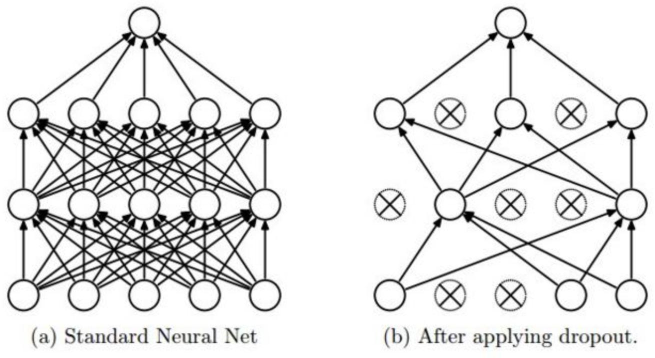

Figure 1: Dropout on a Network

5 Batch Size

Large datasets can obviously contain errors, like mislabeled images. If we backpropagate after every forward propagation, the network might move have a gradient that will shift it the wrong way. To combat this, we can adjust the batch size, the number of data points that the network will forward propagate before going through back propagation.

Having a small batch size means that the weights will update faster, but probably less accurately, whereas having a large batch size means the weights will update slower, but probably more accurately. Generally, we want to choose the largest batch size that our machine's RAM can handle.

6 Learning Rate

In gradient descent, we look for the direction of the steepest decline. Once we have that gradient, we need to decide how far we should go down in that direction. This value that is multiplied by the gradient is called the learning rate. If we choose a learning rate too small, it would take forever for the network to converge on the minimum error rate. Additionally, we could get stuck in local minima, points where it is the lowest point relative to its location but not relative to the entire function. See the figure below for examples of local minima. If we choose a learning rate too large, we can accidentally skip over the minimum (if the minimum is at 2.5 and we go from 2 to 3 with a step size of 1).

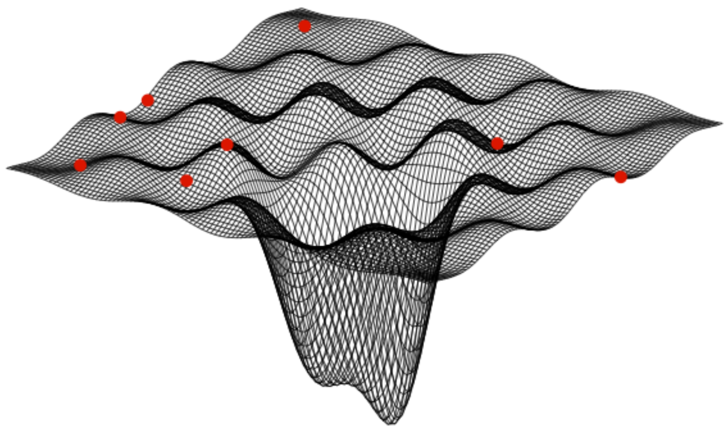

Figure 2: Minima of the Error of a Network

One possible way of addressing this is to use momentum. Momentum adds acceleration into gradient descent. Over iterations, as the steepest decline gradient is computed, the learning rate grows, picking up in "speed". A good way to think of this is as if a ball was put on the error function graph. The ball would gain speed as it goes downhill and lose speed as it goes uphill. This helps avoid local minima as there would a large enough step size to avoid them.

7 Activation Functions

A major problem that arises in networks is the vanishing gradient problem. To understand this problem, lets assume we have 4 different layers: an input layer, 2 hidden layers, and an output layer. The gradient of the error with respect to the fourth layer bias is:

$$\frac{\partial E}{\partial w_4}$$

To calculate the gradient of the error with respect to the third layer, we use the chain rule and get:

$$w_4 \times \sigma'(w_4 \times a_3 + b_4) \times \frac{\partial E}{\partial w_4}$$

In these expressions, E denotes the error, w_j denotes the weight matrix of a layer, b_j denotes the bias of a layer, and a_{j-1} denotes the output from the previous layer. As you may notice, there is more and more stuff that is getting multiplied as we compute the gradient in earlier layers. As a node converges to its optimal value, the derivative approaches 0. Although this is helpful because we don't want to have to retrain this node, this affects previous layers too, by creating a smaller and smaller gradient.

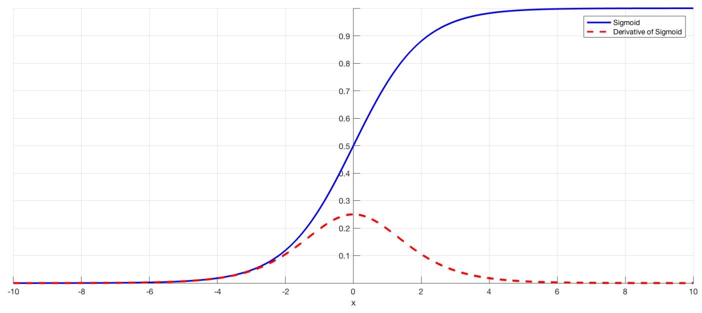

Figure 3: Sigmoid Function and its Derivative

Lets look at a list of activation functions to see if we can find one that avoids the vanishing gradient problem.

1. Sigmoid:

$$\frac{1}{1+e^-x}$$

   This is the function we've been using normally in our networks.

2. Tanh:

$$\frac{e^x - e^{-x}}{e^x + e^{-x}}$$

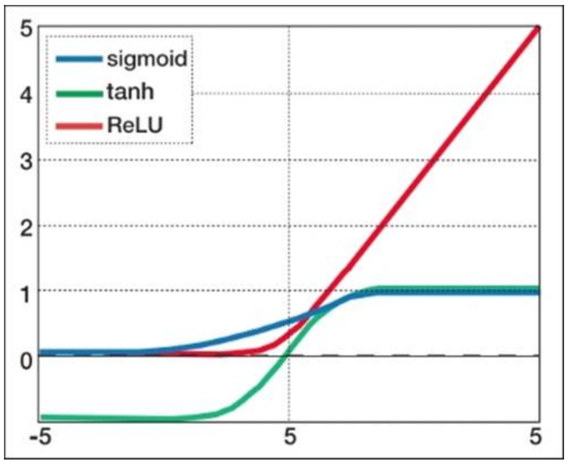

Figure 4: Different Activation Functions

This function looks more complicated and has a more complicated derivative, which means it'll take more time to compute. A key differentiation between hyperbolic tan and sigmoid is that tanh ranges from -1 to 1, rather than 0 to 1. To understand why this is important, imagine a network had inputs that were all positive. If we were to use sigmoid, the gradients would all have the same sign, meaning the weights can only increase or decrease together. Tanh allows different layers to have different signed gradients to update.

3. ReLU:

$$max(0, x)$$

This function looks a lot more simple. The derivative of ReLU, or the Rectified Linear Unit, is very easy and fast to compute. When x is positive, the deriviative is 1, otherwise, the derivative is 0. This avoids the vanishing gradient problem because the derivative is just 1, so when multiplied over and over again, the product won't tend towards 0.

4. Leaky ReLU:

$$(x < 0)\alpha x + (x >= 0)x$$

The problem with ReLU is that when the input value is negative, ReLU has a derivative of 0. If a node constantly passes in a negative value into the ReLU activation function, then the node will not update, as the derivative will make the gradient 0. This can occur, for example, if there is a large negative bias in a node. Since the node will never update, it is "dead". Sigmoid and Tanh can also suffer from this problem, but at least there's a little gradient flowing through to help it recover. Leaky ReLU seeks to solve the "dying ReLU problem" by having a small incline on the negative side of the function. In the above expression, α is a small value, commonly 0.01. This allows ReLU nodes the chance to recover.

8 Optimizers

We've seen that preventing getting stuck in local minima and efficiently updating weights is hard. To account for this, various modifications to the backpropagation algorithm have been proposed to produce more efficient weight updating and increased stabilization. We won't go into how they work because it's pretty complex. However, there are a few key features that we will highlight.

1. Learning Rate: Look at section 6.
2. Momentum: Momentum is a trick to prevent getting stuck in local minima. At each update, we factor in the change that we did the previous step, maintaining larger strides if our previous strides were big and vice-versa.
3. Decay: At each step, the weights are all multiplied by a constant less than one. This prevents exploding values just like L1 and L2 regularization.
4. Epsilon: At each step, some fuzz / noise is applied to the weights and biases, helping increase regularization.
5. Lots of other variables: Read the documentation for specific algorithms.

Here are a brief sample of some optimizers that are most commonly used. There are lots of other ones, though in general they are very ad-hoc or outdated.

1. (Stochastic/Mini-batch/Batch) Gradient Descent: Normal updates! Basically the run-of-the-mill algorithm that involves the least amount of computation.
2. RMSProp: Divides learning rate for a given weight by a running average the magnitude of recent gradients. Useful in RNNs.
3. Adam: Also adapts learning rates for given weights. Similar to RMSProp, but also includes momentum-like functionality. In general, this is the best one to use.