CS 189 Project T - Hyperparameters

Daniel Abraham, Ganesh Vurimi,
Matthew Bronars, Warren Deng, Rasika Iyer

November 2020

1 Introduction

As an aspiring Machine Learning computer scientist, I hope it is becoming clear that your job is as much of an art as it is a science. There is not a "correct" way to extract information from your data, you must develop intuition about how to pick the best models, techniques, and loss functions for the problem at hand. In making these decisions, there are many different dials you can turn to get at better results. Some of these parameters unfold in the process of fitting your data, but others must be chosen by you, the model architect, and we call these hyperparameters. In this note we will describe some of the common hyperparameters, the effect they can have on models, and methods you can use to settle on their optimal values.

1.1 Parameters vs. Hyperparameters

Parameters simply refer to the coefficients or weights of the model that are choose by the model itself. This is usually done as the computer optimizes them to minimize the error/loss function (or perhaps to maximize a cost function). A simple example is if model is trying to fit a parabola of the form

$$y = ax^2 + bx + c$$

to a set of points, then a, b and c are the parameters that the model will try to learn. Hyperparameters on the other hand are various parameters set by the model architect to dictate how the model goes about the learning process. These can affect the accuracy of the model, the training time and the types of data that is learned. You have already seen some hyperparameters before such as the regularization constant in L1 and L2 regularization, the activation function used by MLP's and neural nets in general as well as the learning rate for neural networks.

1.2 Types of Hyperparameters

By and large, hyperparameters will fall into two different categories; optimization hyperparameters and model hyperparameters. In Machine Learning we are generally picking a model that represents our data and then optimizing that model based on a loss function. Model hyperparameters are the ones that determine the structure of the function our computer it trying to learn. Are we trying to fit linear data, quadratic data, cubic data? Optimization parameters on the other hand control how the computer is going about this optimization. How much should the computer be penalized picking a solution with a large magnitude? It should be clear that these are questions the model architect has to answer, but hopefully any confusion will be cleared up in this next section as we go over some common hyperparameters you have likely seen before.

2 Common Hyperparameters

2.1 λ: The Regularization Term of Ridge Regression

Perhaps the hyperparameter that you have you seen the most in your coursework thus far is the regularization term λ in ridge regression. Recall the ridge regression solution:

$$(\mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi} + \lambda\mathbf{I})^{-1}\mathbf{\Phi}^{\mathrm{T}}\mathbf{y}$$

λ is chosen to be a non-negative value that penalizes the norm of w. Also note that with a λ value 0, the above expression simplifies into the traditional least squares formula. This penalty term grows as the weights increase in size, having a potentially substantial effect on the error of the model. As seen in figure 1, when the regularization term is 0, the training error is at its minimum. This is because as mentioned, $\lambda = 0$ is the same as using OLS, which has the sole purpose of finding the best fit. However, introducing and increasing the regularization term will penalize the magnitude of the weights so the model will put more emphasis on keeping them small. While increasing error may seem counter intuitive, this regularization term can help with numerical stability and combat overfitting. This is because by penalizing the norm of w, we are encouraging the model to cut out weights of features that only serving to fit our noise while keeping the ones that capture the underlying structure. However, you should be able to see that if our λ is too high, all the weights may be forced to 0 in order to avoid the large penalty. There is no perfect work around to this, the model architect must be careful when setting their hyperparameters for incorrect values may give you unintended results. The important takeaway is that λ is a hyperparameter since there is no exact formula to choose it or to learn it, but is rather set by the designer of the model based on the context of the problem and possibly through some trial and error.

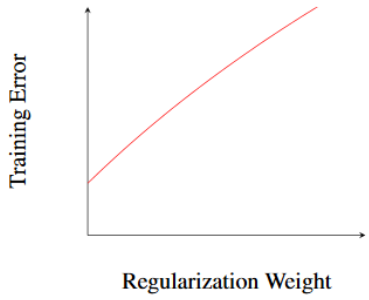
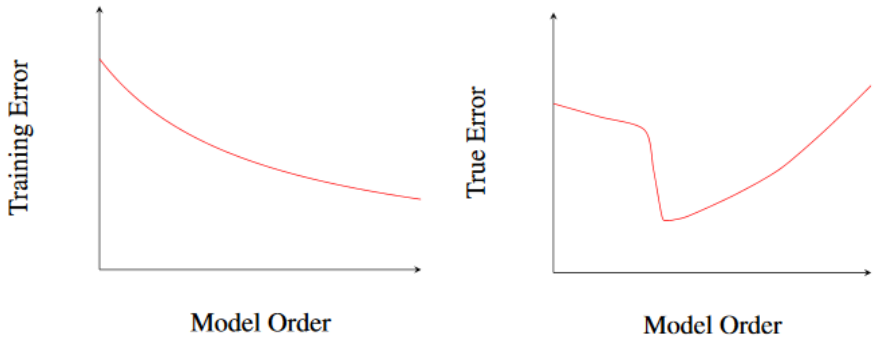


Figure 1: Training Error vs. Regularization Weight [1]

2.2 d: Degree/Type of Approximating Function

Another set of hyperparameters that can be customized is the degree and type of approximating function. You may recall from the feature engineering unit that the features used should belong to a family of universal approximators, such as polynomials or the Fourier series. Both of these can be used to approximate different types of underlying models. The feature space can also be elevated as well to allow for better approximations. For example if we are trying to approximate a circle with only x and y coordinates, we may not get the best results but if we elevate these to x^2 and y^2, we have a much better approximation. One key point to remember is that as the order of the model increases, the training error will keep decreasing due to the fact that a more complex model can approximate the data better and better. However, this will result in *overfitting* because the model will be over-precisely fit on the training data that it will not be able to account for variations within the actual data. See figure 2.

[1] Taken from fa20 CS 189 note 2
[2] Taken From fa20 CS 189 Note 2



(a) (b)

Figure 2: (a) Training error will always decrease as the model becomes more complex. (b) However, a complex model is prone to overfitting and cause the true error to increase [2]

2.3 k: PCA Projection

Recall that PCA is a method of reducing the dimensionality of the data matrix. Let us formally define the PCA projection and then highlight the relevant hyper parameter. Say that we have a data matrix with n data points and d dimensions per data point:

$$X \in \mathbb{R}^{n \times d}$$

Recall that PCA requires de-meaned data. Or, mathematically:

$$\hat{\vec{x}} = \frac{1}{n} \sum_{i=1}^{n} \vec{x_i}$$

$$\hat{X} = X - \hat{\vec{x}}$$

We have slightly abused notation. When we write $X - \hat{\vec{x}}$, we are referring to subtracting the row average of X from each row of X. This will give us a zero mean version of X, which we call $\hat{X}$. Using this new data matrix $\hat{X}$, let us define the k-dimensional PCA in terms of the SVD of $\hat{X}$:

$$\hat{X} = U \Sigma V^T$$

We will define V_K as the first k columns of V. These first k vectors are in fact our first k principal components! So, we define X_k as:

$$X_k = PCA_k(X) = \hat{X} V_k$$

Notice that $X_k \in \mathbb{R}^{n \times k}$. That is, we have reduced the number of dimensions from d dimensional data to k dimensional data where $k < d$. It is also useful to define the reconstructed version of the k-PCA of X:

$$\text{Reconstructed X} = \bar{X_k} = X_k V_k^T$$

Notice that $\bar{X_k} \in \mathbb{R}^{n \times d}$, however $rank(\bar{X_k}) \leq k$ because our reconstruction did not add any new information. That is, the dimensionality that was lost in our column space upon calculating X_k was not returned.

We bring up PCA because it is a strong method of dimensionality reduction. It helps us tackle problems that have over-parameterized data. However, we do not always know which value of k we must reduce our dimension to. That is where hyper-parameter optimization comes to play. We can grid search over all possible values of k to find the reduced dimension of our data matrix that best suits our model.

A very fine application of this is in image classification. Images have very large dimensionality. Often times, if we are trying to perform a simple classification technique, like binary classification of handwritten digits, then we may benefit off of choosing a subset of pixels to use for classification rather than all of them.

2.4 Architecture of Neural Network

Recall that multilayer perceptrons have three different types of layers: input, hidden, and output as depicted in figure 3. However, please note there can be multiple hidden layers. Each of the circles in figure 3 represents a neuron. In neural networks, there are several hyperparameters including the number of neurons in each layer, the number of hidden layers, the learning rate, and the activation function.

Using more neurons reduces training error, but can also lead to overfitting of the data. Therefore, the number of neurons will need to be balanced, so the generality is not lost.

Multilayer perceptrons with just one hidden layer are universal approximators. In other words, given a continuous function, there exists a network with one hidden layer such that the output of the circuit can be made arbitrarily close to the output of the given function for all inputs. Adding a layer increases the dimensional complexity of the training data. Multiple layers can be used to represent convex regions, so it is not limited to data that is linearly separable.

The learning rate is used to training a neural network through gradient descent. It scales the magnitude of the weight updates to minimize the loss function. If the learning rate is too low, it will take a very long time to train the model. If the learning rate is too high, the gradient descent algorithm can diverge and skip past minimum points.

Lastly, the activation function is a threshold function. It takes in the weighted sum of the input neuron, adds the bias, and decides whether the neuron gets activated or not.

Examples of activation functions include Sigmoid, Tanh, and ReLu. Sigmoid activation functions tend to work well with classification problems, which is clear from the graph in figure 4. There are more widely used activation functions and in fact, you can use an activation function of your choice.

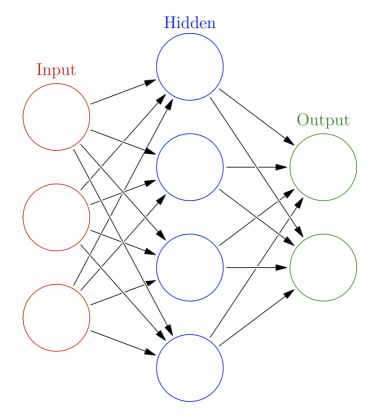


Figure 3: Example drawing of a multilayer perceptron [3]

[3]https://medium.com/@PABTennnis/how-to-create-a-neural-network-from-scratch-in-python-math-code-fd874168e955
[4]https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0

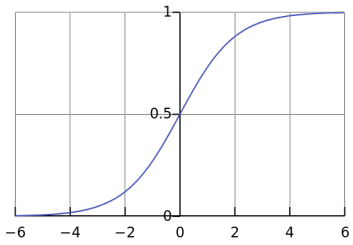


Figure 4: Sigmoid activation function [4]

2.5 α: Learning Rate/Step Size in Gradient Descent

Recall the update step in the gradient descent algorithm:

$$x_{k+1} = x_k - \alpha \nabla f(x_k)$$

The step size term, also known as a learning rate dictates how big of a 'step' the algorithm takes in the direction of the steepest gradient. The size of this step size can be crucial as one that is too large will continually overshoot the minimum and not converge while one that is too small can take extremely long to converge or may become trapped in a local minima. A helpful visualization is shown in figure 5.

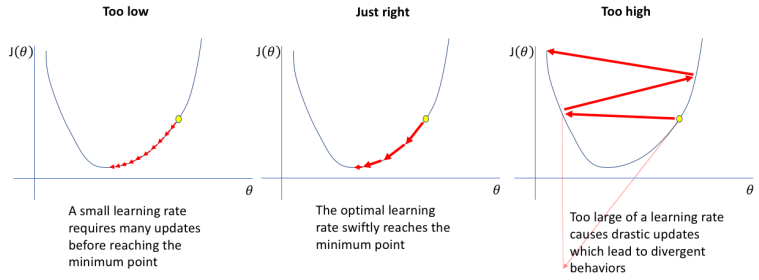


Figure 5: The consequences of different learning rates [5]

3 Hyperparameter Tuning

There are several ways to pick the combination of hyperparameters that will work best for the model being trained, but they are really all variations on guess and check. The general strategy is to choose values for your hyperparameters, train that model on training data, then compute error on validation data and compare to models defined by other hyperparameters. When doing this take care to avoid data incest, you need to evaluate your hyperparameters on different data than you used to train the model. Remember that you can use k-fold cross validation if you don't want to waist any data on a validation set. In this section we explore the ways that you can systematically search for optimal hyperparameter values, these are known as Random search and Grid search.

3.1 Random Search vs. Grid Search

Neither of these methodologies are very complicated and their names give away their secrets. Random search simply involves selecting combinations of hyperparameters randomly, often from predefined distribu-

[5] https://www.jeremyjordan.me/nn-learning-rate/

tions/ranges. Grid search on the other hand involves methodically checking every combination of hyperparameters from user defined ranges. In cases where you have a small range of values you are searching over, both methods may yield similar results in similar amounts of time. However, more consideration is needed when you don't have a good understanding of the model or you have a lot of hyperparameters to tune. A good example of this is the neural net application. As seen in section 2.5, there are multiple hyperparameters that are taken into consideration when designing a model. The number of hidden layers, number of neurons in each layer, the activation function used, learning rate can all vary. These are the only the ones we have discussed so far, there will be more that are discussed in the neural networks part of the course such as momentum and dropout.

The time complexity and computing power needed for both types of search must be considered. Random search is trivial - it will take $\mathcal{O}(n)$ time, where **n** is the number of iterations that will be run. For grid search, the resources needed increase in a multiplicative way based on the number of hyperparameters and their ranges. If there are two hyperparameters to optimize, each with α and unique values, then the resources needed will be $\mathcal{O}(\alpha\beta)$. If we add a third hyperparameter with γ unique values, then the resources will scale to be $\mathcal{O}(\alpha\beta\gamma)$, as one may infer.

Unfortunately, there is not yet a clear answer on when to use which method. If a similar model, based on similar data has been constructed before, then the model architect may have an idea of what range the hyperparameters should fall in. In such a case, a grid search may prove to be effective. In scenarios where the model may be a little more novel, random search may be more effective. A combination of the to may also be used, where random search is used to find a "ballpark" for an optimal hyperparameter combination and then grid search can be used to refine it. This technique may be better for problems with higher dimensionality. This is due to the fact that as seen in figure 6, grid search only tests three distinct values for each parameter while random search tests 9 distinct values for each. [6]

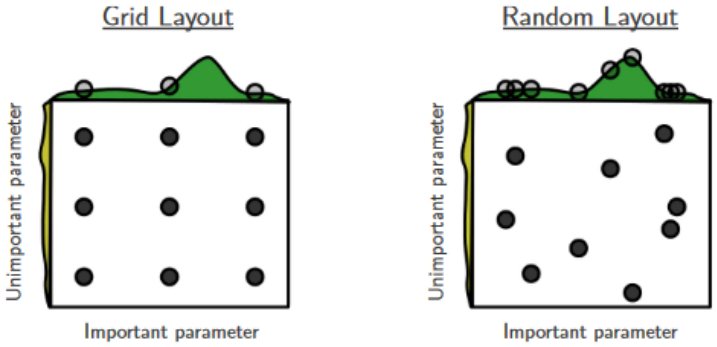


Figure 6: Visualization of Grid Search and Random Search [7]

There is still a lot of research going on in this area. In later classes, you will also be introduced to other hyperparameter optimization methods such as Bayesian Optimization, but that is currently out of scope for this class.

[6] More about this can be read in Bergstra and Bengio's paper.
[7] Image from Bergstra Bengio's paper