Lab 1 - CS599 Deep Learning

Keegan Line
Northern Arizona University - School of Forestry
ktl56@nau.edu

1 Linear Regression

The goal of exercise one was to fit a linear regression model using learning and gradient descent. We start with our true model which is

$$y = 3x + 2 + \epsilon$$

where x is given by 500 points generated from a random normal distribution and epsilon is some random error. To start we have epsilon generated from a normal distribution as well but we will test different distributions later on. The two loss functions we will consider are the mean squared error (MSE) and the mean absolute error and they are defined by the functions below:

$$MSE = \frac{1}{n}\sum_{1}^{n}(\hat{y}_i - y_i)^2$$

$$MAE = \frac{1}{n}\sum_{i}^{n}|\hat{y}_i - y_i|$$

where $\hat{y}$ stands for our predicted "ith" value and y_i stands for the true "ith" value.

When starting both our weight and bias terms at zero we see that MAE and MSE converge to the true value at very different rates. After 2500 training steps, we see that the MSE gives us a weight of 2.9910 and bias of 1.9916 which is close to the true values of 3 and 2 respectively. The MAE error does noticeably worse and provides us with a weight of 1.5384 and a bias of 1.0134. Below figure one shows us the convergence of the weights and bias to the true values with the MAE loss function and figure two shows the same under the MSE loss function.

From the images above we see that the two models converge to the true answer at two different rates. The MAE does not get close to the correct answer and we can see still appears to be learning at the 2500th epoch. The MSE however gets close to the correct answer and the curve starts to smooth out as it approaches the 2500th epoch. Increasing the number of epochs allows for both models to learn more and get closer to the correct answer. After 5000 epochs, the MSE model does seem to stop learning but it takes the MAE model closer to 1300 epochs to reach the same state. We see that MAE takes longer than MSE and this is most likely due to one being squared. Since MSE is squared, it puts more emphasis on getting minimizing larger errors and that helps the model learn quicker. When running 2500 epochs patient scheduling doesn't help because both models are still learning and that would only take effect if the loss remains the same meaning the models are not learning anymore. What does help is increasing the learning rate. Increasing it 10-fold from 0.001 to 0.01 massively speeds up the convergence to the true answer for both loss functions. Both losses do not decrease after the 1000th epoch. The final weight is 2.9968 for MSE, 2.9886 for MAE and 2.9932 for the hybrid of both loss functions.

Where the first weight and bias initialization occur can have a very large impact on where the final model ends up. If the initialization randomly sets the values far away from the true value, it will take the model a longer time to get the correct value since it started farther away. If the starting value is closer, it will take less time. Initially all model parameters were started at zero and we could see how long it would take to learn. With the new initializations the time it takes could vary.

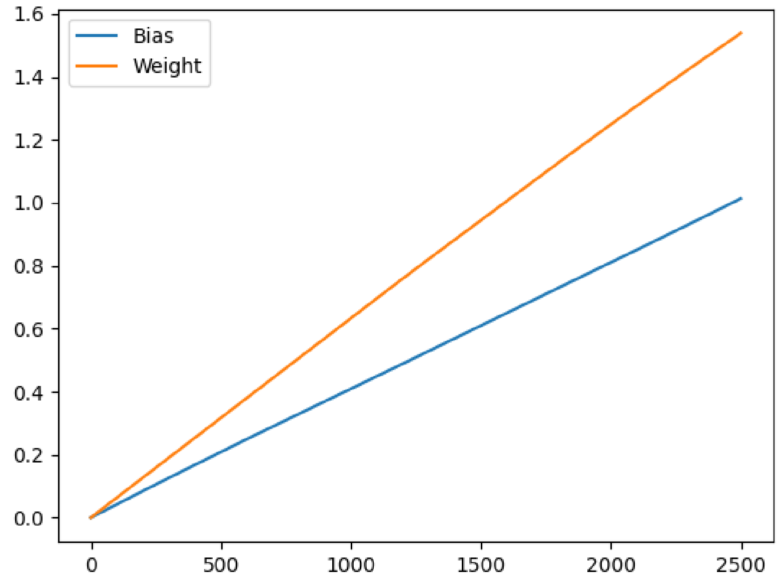

Figure 1: Training of Bias and Weights using MAE with 2500 epochs

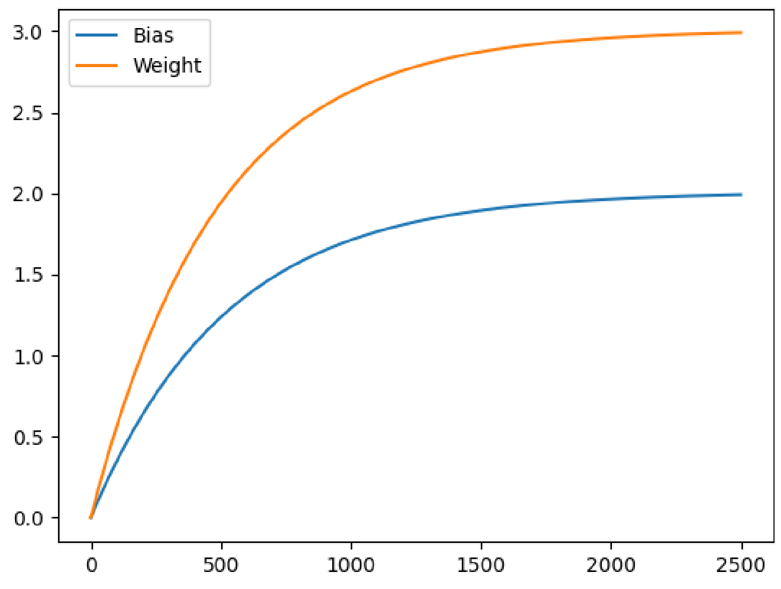

Figure 2: Training of Bias and Weights using MSE with 2500 epochs

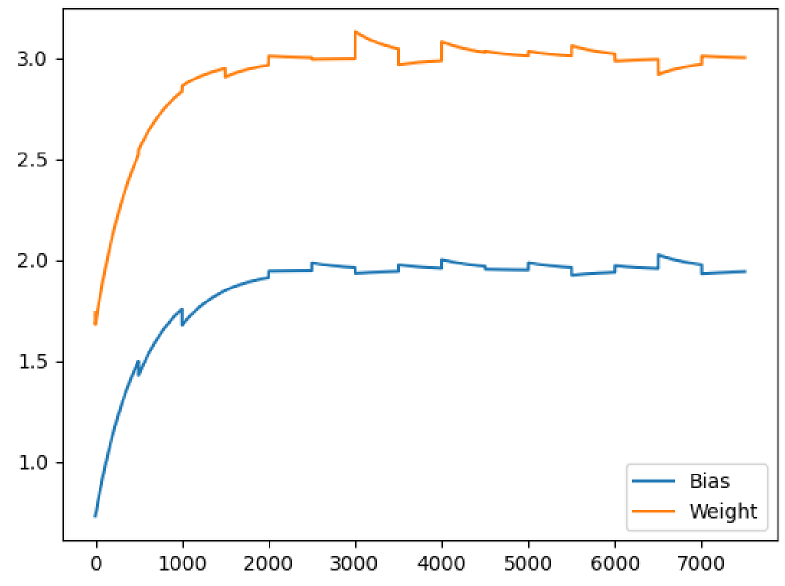

Figure 3: Training of Bias and Weights with noise added over 7500 epochs

Changing the amount of noise in the original data set means it will take the model a little longer to learn. One major change that can be seen is in the bias term. Our true weight of the model will remain constant at 3 regardless of what the noise is. Our true bias will remain constant as well at 2. What does change is our models ability to pick up on that bias with the larger amount of noise. If we double the normally distributed noise in the true model, after 5000 epochs the predicted model computes the bias to be 1.8955. With five times the noise, the bias is 1.7387. During training, our predicted model is picking up something in the noise and is starting to model that with the bias instead of the true model.

Adding noise to the weights and bias is a good way to help the model when it is stuck. I tested out adding 0.05 times a random error from a standard normal distribution and it lead to the best estimate of the weight at 3.0009. The bias was a little more off than before at 1.9417 but is still very close to the true value. One benefit to this is that it allows for the predicted model to cross that threshold of the true value. Before if the model was initialized below three, it could get close to three but would never be above three. If it was initialized above three, it again could get close to three but would never go below. This random noise allows for it to cross the barrier and get an even better estimate. One thing to note is that you must pay attention to your learning rate scheduling if you are using it. If your patience steps is less than your random noise steps, it could lead to you reducing your learning rate and not being able to correct anymore for the random noise that is introduced. Figure 3 above shows how the weights and biases change over time with a small amount of random noise introduced.

Noise in the learning rate does not seem to have much of an effect on model training performance. If we add large positive amounts of noise to the model early on, it might allow for the model to train faster since the learning rate will be higher. As the model continues to train and the gradient descent becomes less and less, this added noise will be less valuable as the gradient will be quite small.

Overall I think these changes would remain pretty consistent regardless of the classification problem or mathematical model. Each problem you come across is going to have problem specific things you will have to account for but in general I believe a lot of what I found is going to hold true. When encountering a new problem, you should test out several different examples from above and see what impact they have on the specific problem at hand.

Every time you run this model you should expect to get a different result. This is because of the inherent randomness of the code itself. Once you click run, there is random data being generated, random weights and biases being assigned and random noise being applied. Since all of these are random, it is expected that you will get slightly different results each time. A way to prevent that is to set your seed. When a seed is set, it means the randomness that comes after it is going to be the same each time. The number is still random but only random once. That means if you want to compare a change but you want the data to be the same between trials, you can use a set seed and it will ensure everything is equal except for that change.

The time for each epoch on CPU time 0.0012 seconds.

The model that lead to the fastest convergence was the model using the loss function MSE. No model was truly robust to noise but this model was also the best in that respect as it was able to correct more for it. Noise can be beneficial in certain aspects. Noise in the dataset and noise in the learning rate did not help the model at all. What helped the model the most was noise in the weights as it could help us get closer to the true value by getting us out of valleys in the loss function.

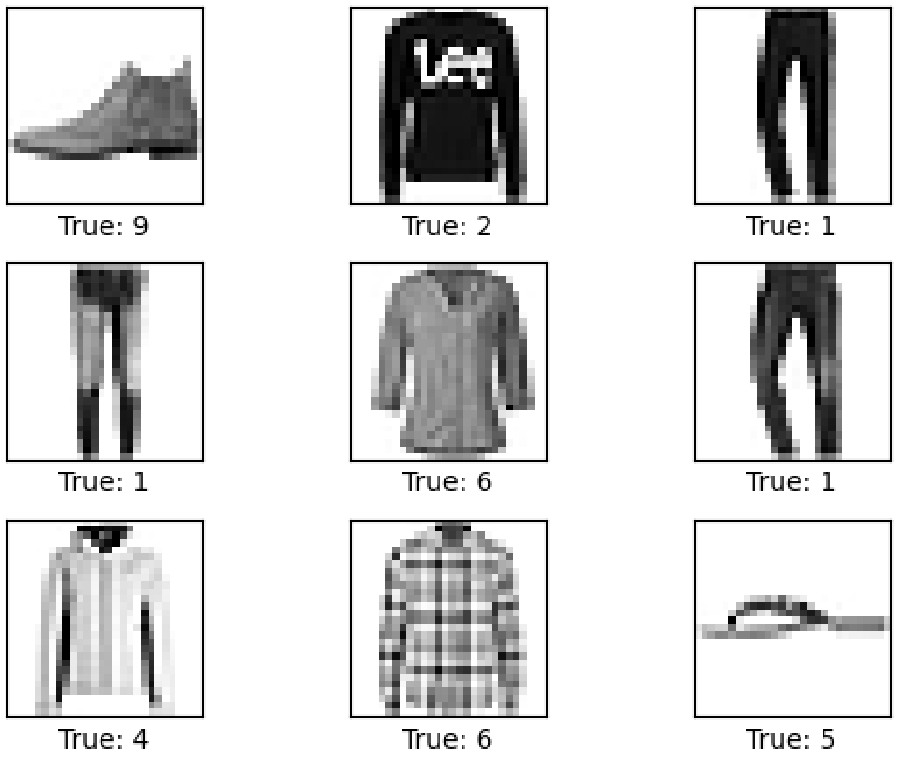

Figure 4: An example of nine images from the test set with their correct labels

2 Logistic Regression

The goal of this exercise was to be able to build a logistic regression model to be able to predict type of clothing based on a black and white image using deep learning. The training set contains 60,000 clothing images that can be categorized into 10 different categories. There is then a test set containing 10,000 labeled examples that were not used to train the model.

From figure five below, we can see a couple of different things. Across 10 epochs we see a couple of different features. The first is that stochastic gradient descent performed the worst out of all three optimizers. It achieved around 70 percent accuracy whereas the other two optimizers averaged over 80 percent accuracy. We can also see that in at least two cases, the training set accuracy outperformed the validation set accuracy which is an indication that the model may be over fitting. To try and combat that, the regularization parameter was increased from 0.001 to 0.1 to try and mitigate the over fitting. The result was that there was a slight decrease in overall train and validation accuracy but the two numbers were closer together meaning it helped solve the issue.

When a longer number of epochs was trained for, we do see some increase in performance in all optimizers as we would expect. There does seem to be a limit with this specific data set in the increase in accuracy you can get from increasing the epochs before additional epochs provide no further benefit.

Changing the proportion of data that is in the training set versus the validation set does seem to have a large impact on the accuracy. The first initial models were trained with 90 percent of the data belonging to the training set and 10 percent to the validation set. Other splits that were tested were 80-20, 75-25, and 50-50. The split that appeared to perform the best was the 75-25 split. I believe this

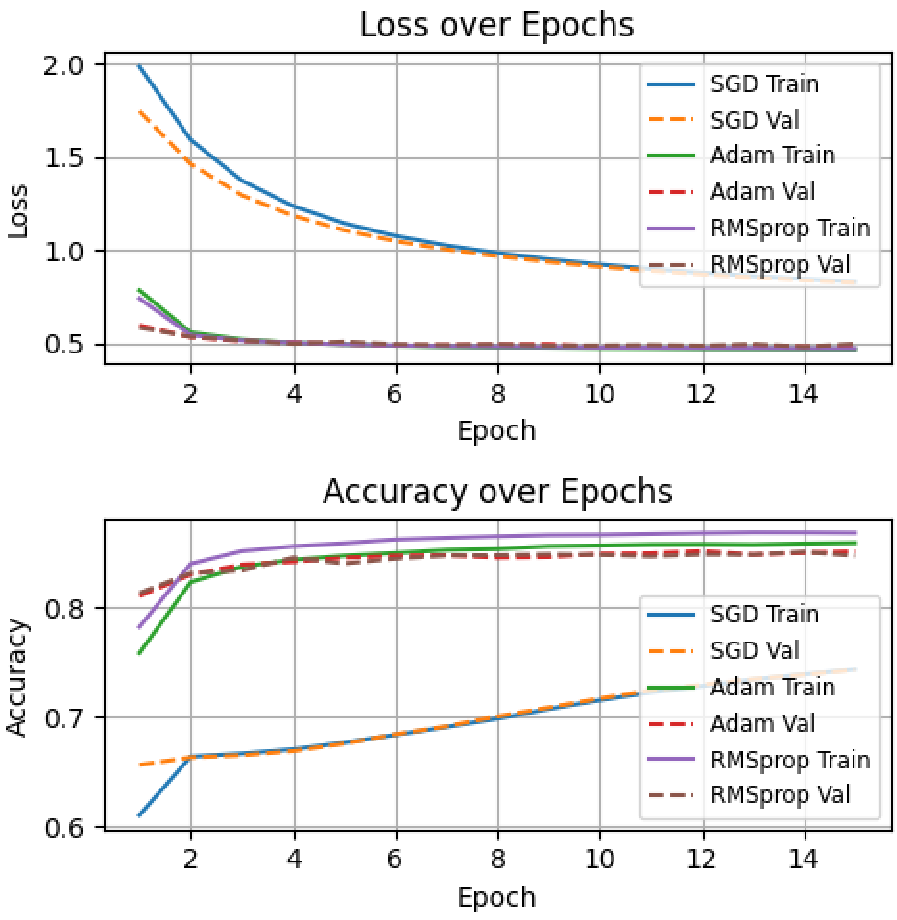

Figure 5: Accuracy of three different optimizers over 15 epochs

is because as you start to add more and more data to the validation set, you start to loose information about the subject from the training set and that hampers model performance.

Another model parameter we can investigate is the batch size. The default batch size was set to be 128 images at a time. Different batch sizes were tested such as 8, 16, 32, 64, and 128 with 64 found to be the optimal batch size among them. Using this batch size ensures that there are enough images and information to update the model parameters efficiently without slowing down model performance with too many images and a large computational drag. For this we found that each epoch took about 3.23 seconds averaged across multiple optimizers.