IST 597: Assignment 00010

Jooyoung Lee
Information Sciences and Technology
Pennsylvania University
920172979

1 Problem1. Linear Regression

To get you to think in terms of neural architectures, we will approach the problem of estimating good regression models from the perspective of incremental learning. In other words, instead of using the normal equations to directly solve for a closed-form solution, we will search for optima (particularly minima) by iteratively calculating partial derivatives (of some cost function with respect to parameters) and taking steps down the resultant error landscape. The ideas you will develop and implement in this assignment will apply to learning the parameters of more complex computation graphs, including those that define convolutional neural networks and recurrent neural net.

We create a complete example of using linear regression to predict the parameters of the function

f(x) = 3x + 2 + noise

Given a point x we want to predict the value of y. We train the model on 10000 data pairs (x,f(x)). The model to learn is a linear model

y = W x + b

1.1 Loss Function

We use a loss function to determine how far the predicted values deviate from the actual values in the training data. If the number is close to 0, it means the prediction is accurate. I implemented two different loss functions to compare their performances.

First, **Mean Square Error(MSE)** is one of basic loss functions that is easy to understand. In order to calculate it, you take the distances between target variable and predicted values, square it, and average it out across the whole dataset. If I put it in a mathematical formulation, it would be equivalent to

$$loss = (predictedY - Y)^2/numberofdataset$$

Second, **huber loss function** is is less sensitive to outliers in data than the squared error loss. We also have to assign delta value(d) to get the result. If I put it in a mathematical formulation, it would be equivalent to

$$\text{if } |predictedY - Y| <= d \text{ then } 0.5 * (predictedY - Y)^2$$
$$\text{if } |predictedY - Y| > d \text{ then } 0.5 * d^2 + d * (|predictedY - Y| - d)$$

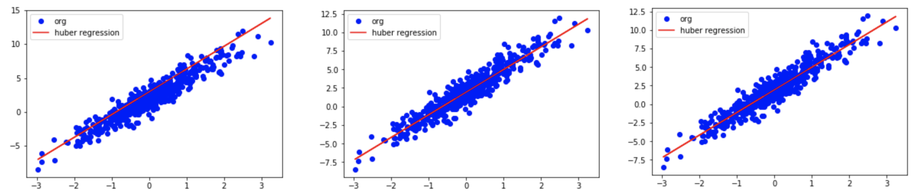
1.2 Results : MSE

First of all, I added random noise to inputs and outputs and created random weight and bias by implementing tf.random_uniform function. I assigned random float32 type numbers, less than 10 and greater than 0.

The total loss using MSE with 1000 training steps and 0.001 learning rate is 2.213 while weight is 3.3561802 and bias is 2.9786484. If I increase the learning rate to 0.01 or 0.1, computational speed increases significantly and the loss goes down to 1.054 which does not even require multiple training steps because it stops diminishing at certain point.

Let's change the initial value for W and B to see if it has effect on performance. If I use a random value between 0 and 50 as the initial value for W and B, loss surprisingly goes up to 49.501 with 6.217115 for the weight and 8.097692 for the bias. This is when learning rate is 0.001. If I decrease the maximum random value from 10 to 5, the loss still stays the highest.

What if we increase the learning rate? Surprisingly the final loss is goes back to 1.054 which is the original output I got in the first place. From these outcomes, I can conclude that lower learning rate(0.001) is more susceptible to the initial value of weight and bias.



[Fig1] output graphs using MSE loss function with learning rate 0.001, 0.01, 0.1 from left to right

1.3 Results : Huber Loss

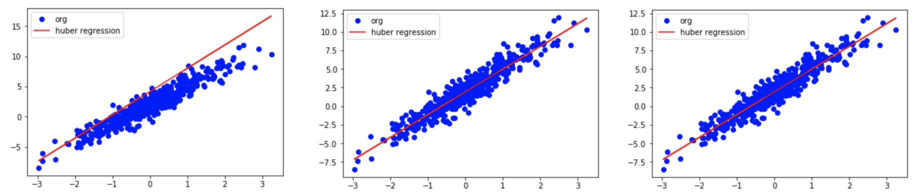
Initial data setting is the same with MSE example.

The total loss using huber loss with 1000 training steps and 0.001 learning rate is 15.261 while weight is 5.040479 and bias is 6.2451053. If I increase the learning rate to 0.01 or 0.1, the loss goes down to 1.054 like the prior example. Increasing training step won't contribute to reducing the loss because the loss stops dropping within 1000 steps.

Let's change the initial value for W and B to see if it has effect on performance. The results are similar to MSE's results: loss goes up to 187,287 with 11.74782 as W value and 18.182787 as b value. However, there is a huge difference when I set random value from 0 to 5. This change results to the loss 1.049 whereas MSE's results did not show much difference in loss value.

What if we increase the learning rate? Huber loss function keeps the same loss value 0.705 just as MSE do.

To sum up, overall performance of MSE loss function surpasses huber loss function's performance and these functions are also vulnerable to change of initial value for W and B.



[Fig2] output graphs using huber loss function with learning rate 0.001, 0.01, 0.1 from left to right

1.4 Results : Hybrid Loss Function

In order to test the hybrid loss function, I added MSE loss function and huber loss function.

The total loss using hybrid loss function with 1000 training steps and 0.001 learning rate is 1.887 while weight is 3.1291955 and bias is 2.1987422. If I increase the learning rate to 0.01 or 0.1, the loss even goes down to 1.759. Unlike other loss functions, hybrid loss function is not susceptible to learning rate since it displays similar loss values after changing the learning rate.

1.5 Power of Unique Seed

$tf.random.set_random_seed(randomnumber)$ allows you to generate same random number every time with same seed value. It means that you can get the exact same results if you run the Notebook multiple times without changing any parameters.

2 Problem2. Logistic Regression

Logistic regression is used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal, interval or ratio-level independent variables. In this you will be using Fashion Mnist which is a dataset of Zalando's article images consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classe[https://github.com/zalandoresearch/fashion-mnist].FashionMnist is much harder than MNIST so getting accuracy in 90's is difficult. You can use whatever loss functions, optimizers, even models that you want, as long as your model is built in TensorFlow using eager execution.

2.1 Data Processing

The total number of train dataset is 60000 and test dataset's size is 10000. Since train dataset and test dataset is clearly divided in the first place, I decided to use 5 percent of train dataset as a validation set by implementing scikit-learn library. Scikit-learn offers `train_test_split` function to split train/validation/test dataset easily. I tested the effect of train/Val split on accuracy and here is the result : if the validation size grows, accuracy diminishes.

validation size	accuracy
0.05	86.14%
0.2	84.06%
0.3	83.4%

2.2 Model

Tensorflow provides a variety of models and it is important to remember that there is no right answer for certain problems. I used softmax model that is known for its effectiveness on a multi-class prediction problem.

2.3 Optimizer and Hyperparameter

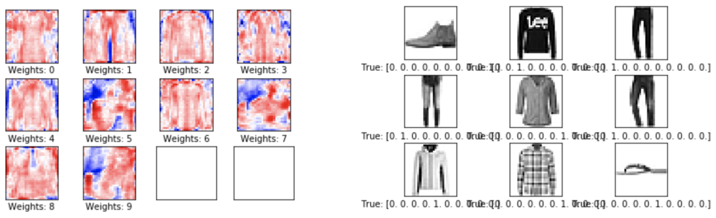
Optimizers shape and mold your model into its most accurate possible form by adjusting the weights. There are several optimizers that Tensorflow provides but I used three different optimizers in order to compare their efficiency.

conditions	AdagradOptimizer	AdamOptimizer	GradientDescent
lr=0.001/batch=64/n_epoch=5	72.89%	86.14%	71.82%
lr=0.01/batch=64/n_epoch=5	81.62%	10%(loss = nan)	80.65%
lr=0.001/batch=64/n_epoch=10	74.65%	84.4%	77.38%
lr=0.01/batch=64/n_epoch=10	82.66%	10%(loss = nan)	82.02%
lr=0.001/batch=128/n_epoch=5	70%	84.68%	67.32%
lr=0.001/batch=64/n_epoch=10	73.04%	84.22%	70.67%

From the table above, using AdamOptimizer with 0.001 learning rate, 64 batch, and 5 epochs is the most effective strategy. In terms of training and validation accuracy, the final accuracy after running 5 epochs are 86.14% and 84.83%.

2.4 Plot Images and Plot Weights

These are matplotlib plots from function plot_images and plot_weights. We can tell training is finished successfully by checking with plots.



[fig3]left is plot weights and right is plot images

2.5 Compare Performance

In order to compare the performance with random forest and svm, I utilized RandomForestClassifier and svm.SVC from scikit-learn. In terms of random forest with 50 max_depth and 100 estimators, the accuracy turned out to be 75.48%. For svm, it (kernel=poly) took a long time to train due to a large dataset and the accuracy was 71.55%.