Assignment #00010

Amirreza Bagherzadeh
PennStateUniversity
PSUID: 946204639

Abstract

The assignment's main focus is to interpret machine learning algorithms and the effect of different parameters on the performance. While there is no use of neural network in this assignment, Tensorflow was used to in the most simple form to perform simple machine learning algorithms.

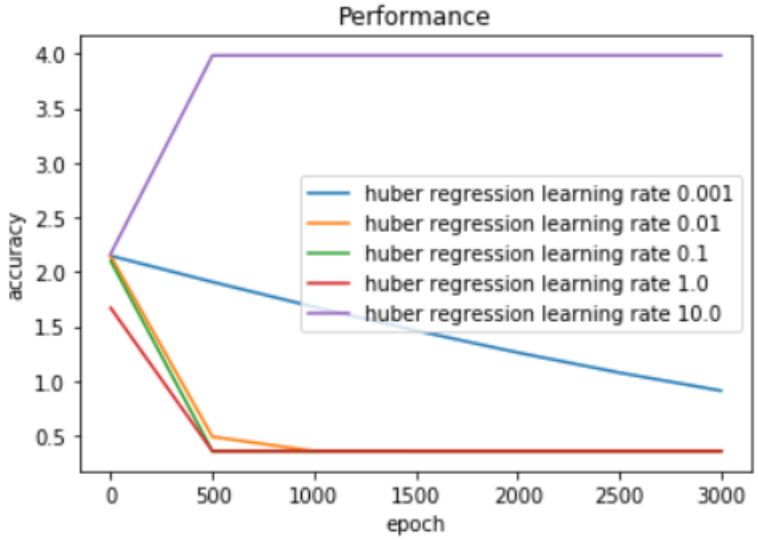
1 Problem 1: Linear Regression

1.1 Changing the loss function

In this part, 4 different loss function has been used: 1)Square mean 2)Huber loss 3)Pseudo Huber Loss 4) Hybrid L1+L2 loss. The last 2 had better result in comparison to the first two. While the huber loss performs better than Square mean, it is pretty slower due to the condition that needs to be checked and different operation on each part. Psudo huber loss holds the good properties of Huber loss and remains differentiable on all points and robust to noise. L1+L2 is also a good loss function. However, it is not differentiable at point "0". So in my opinion Psudo Huber loss is the best loss between the other three.

1.2 Changing The learning rate

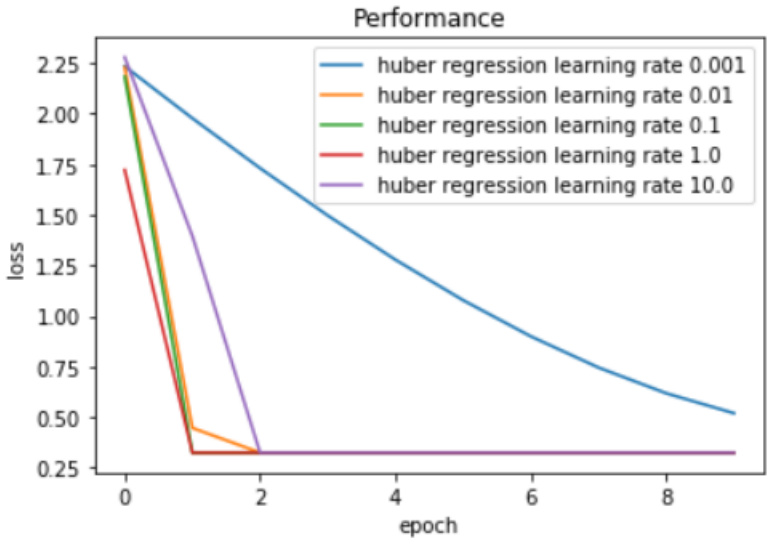
The learning rate has been changed from 0.001 to 10 each time increasing 10 times. the result is as follows:



As it is seen in the plot, the fastest learning rate was 1.0, 0,001 didn't manage to converge in 3000 epoch but it was in the path of converging while learning rate of 10 diverged and will never get to the optimum point.

1.3 Patient Scheduling and longer training

In this part the learning rate was caught to half if the difference between accuracy in two steps was less than 0.00001. The results is as follows:

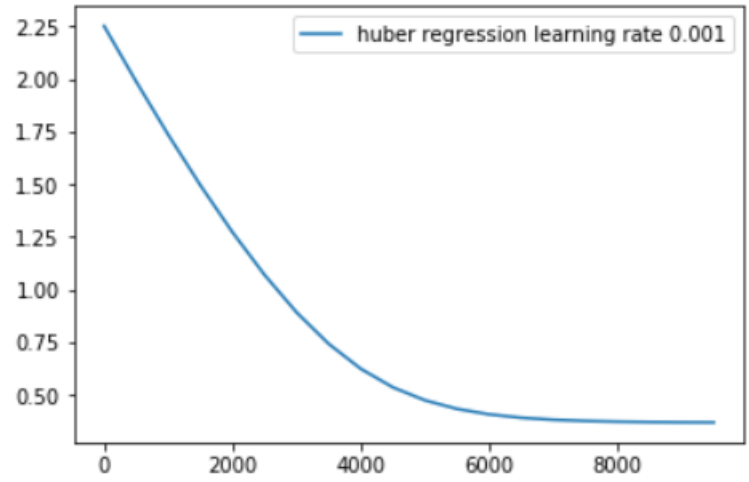


As far as the convergence rate is concern dividing by 2 didn't speed up any of them. However, learning rate of 10 was able to converge after 2000 epoch by this method. It worth mentioning that the convergence point didn't change too. Longer duration of training wasn't helpful and they stayed the same place as before. Just learning rate of 0.001 got nearer to optimal point.

1.4 Different Initial Value for W and B

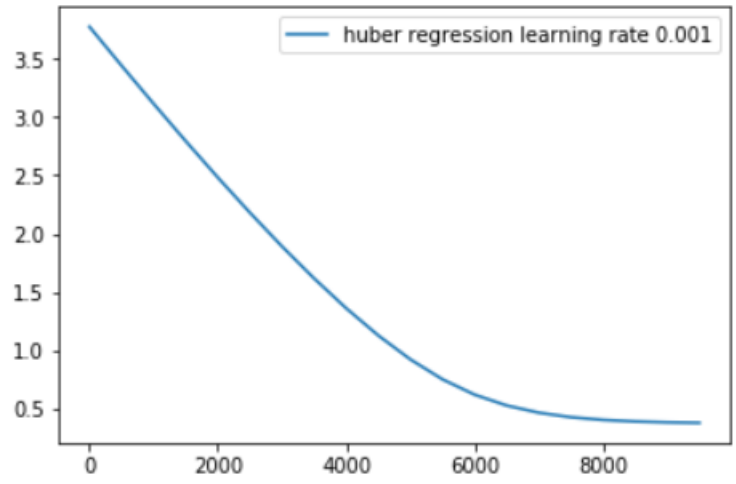
In this part, started from a different point where initial value of both W and B are generated by tf.random. It can be seen that they again converge to the same optimal point. Since we train for 10000 steps differences are not that big.

```
Loss at step 9500: 0.369
W : 2.936720132827759 , b = 1.9084995985031128
```



W and B as 0

```
Loss at step 9500: 0.375
W : [2.9154353] , b = [1.9746892]
```



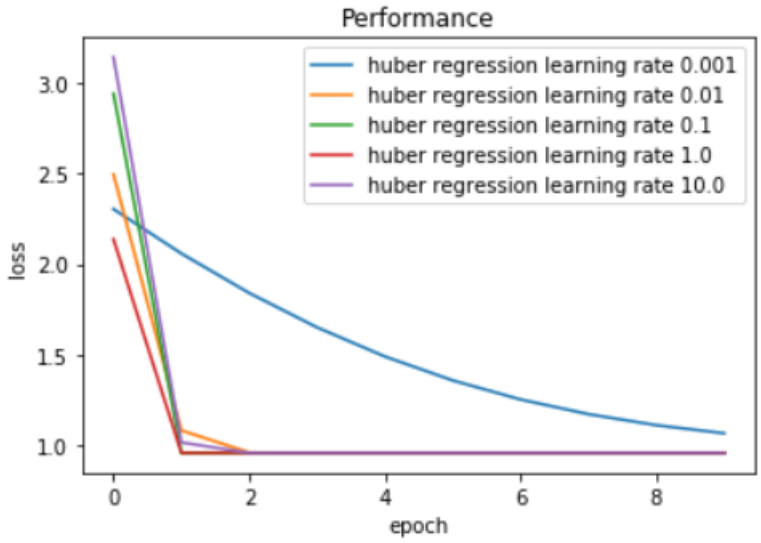
W and B as -1.016811 and -1.3725803

It is obvious that each time we run we would get a different result.

1.5 The level of noise

In this part we changed the level of noise by changing the variance of the noise:

```
W : [3.057939] , b = [1.8673719]
<function matplotlib.pyplot.show>
```



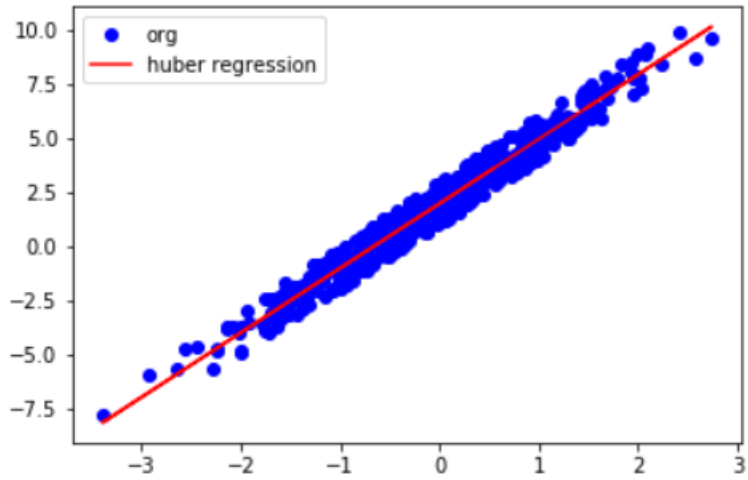
$\sigma = 2$

As it is seen the function which as the Pseudo huber loss is shows good robustness to the noise and the W are close enough to 3. However, b is getting far from 2.

1.6 Various type of noise

In this section, we changed the type of noise to uniform between -1 and 1. The most important thing in choosing the type of noise is that one should concern that the mean of the noise should be zero or else a right regression get b as big as $2 + \mu$. The result of uniform noise is as follows:

```
Loss at step 9500: 0.163
W : 2.9856982231140137 , b = 1.9807374477386475
tf.Tensor(0.16304027, shape=(), dtype=float32)
<function matplotlib.pyplot.show>
```



uniform noise

1.6.1 Headings: third level

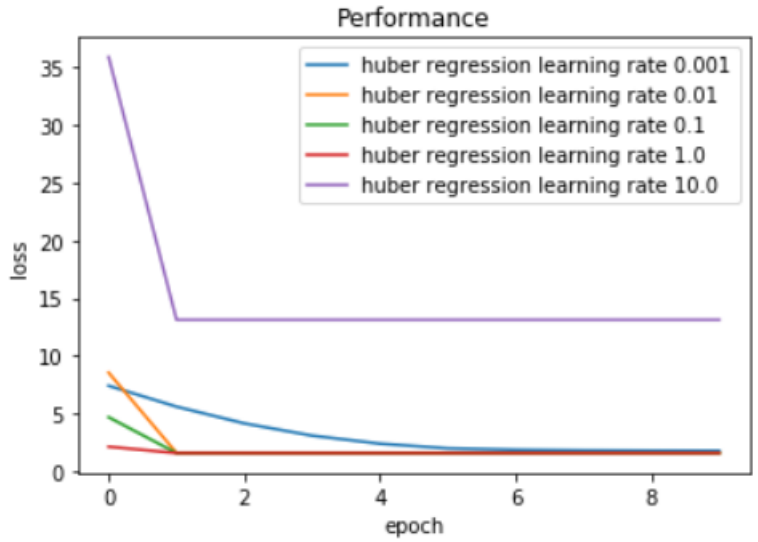
Third-level headings should be in 10-point type.

1.7 Noise in weights

In this section, In each step we add the noise to the weights and monitor the effect. There exists two scenarios:

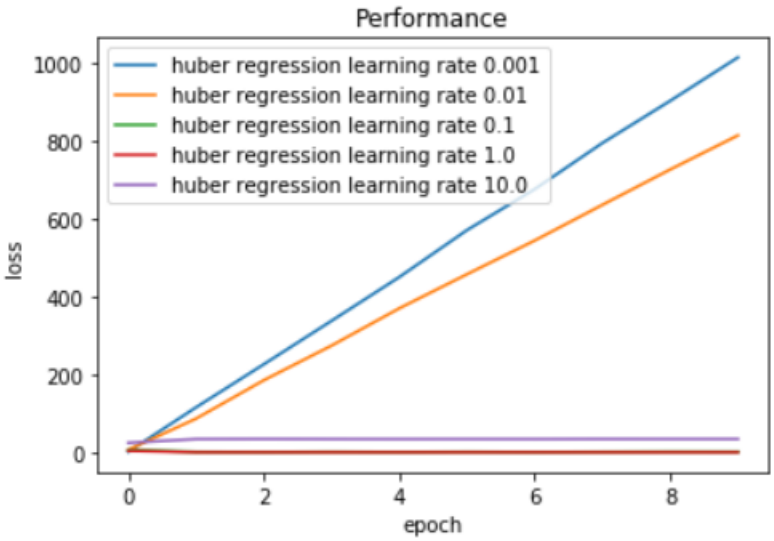
1. The amount of noise we add is too small that it does not changes the convergence:

```
W : [3.2102368] , b = [8.753334]
64.805395819
```



0.00001*np.random.normal([1])

2. We add too much noise that we never converge



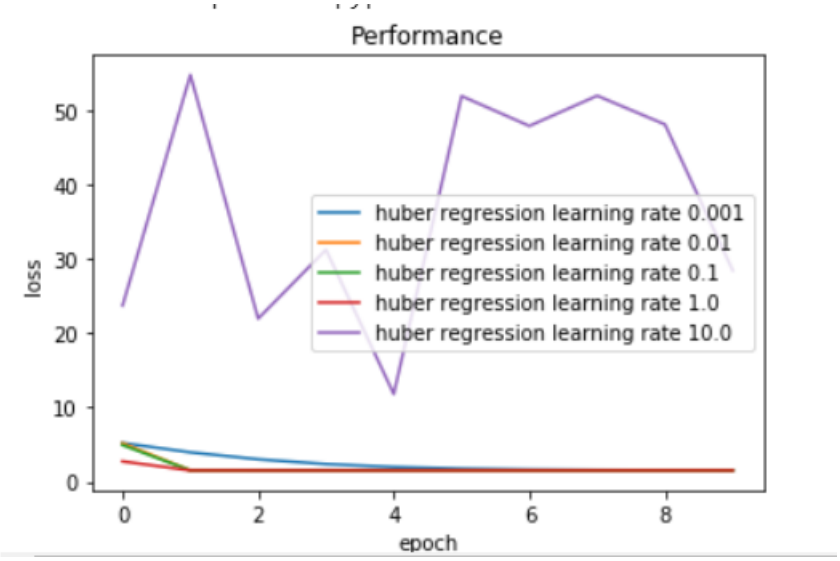
0.1*np.random.normal([1])

As it can be seen if the amount of noise is small in comparison to the learning rate the model would be robust to this kind of noise cause if $dw * \lambda$ where lamda is the learning rate big enough in comparision

to noise we can have hope to converge. However, as dw shrinks we reach to a point where even big learning rate won't help that much

1.8 Noise to learning rate

Very similar to what we had above cause $dw * (\lambda + noise) = dw * \lambda + dw * noise$ where we can substitute $dw * noise$ which just noise. However we won't have the problem we had above where near optimum point we had the problem of noise but in here since the noise is multiplied by derivative, it won't have the effect:



noise = LR*0.01*np.random.normal([1])

1.9 Why we get different result each time?

Each time we run the model different noise is being generated. The best line that fit to this data is not $3x + b$ we only get to this if we generate infinite samples. In case of limited sample each time something different will fit our data.

1.10 Can you get an model which is robust to noise?Does model lead to faster convergence?Do you get better local minima?Is noise beneficial?

As it was discussed in the top pseudo huber loss is more robust to noise than the others but we never can find a model that is completely robust to noise cause the model can only learn so much from the data and can not differentiate between noise and real value. Usually models which are more robust to noise are slower cause there must be some modifications to lost function in a noise range zone so that it fits the data not the noise. and as it was seen in the assignment, noise is not beneficial in this problem. **However, in some cases noise in weight or learning rate may take us out of a bad local minima and lead us to a better one**

1.11 GPU VS CPU performace

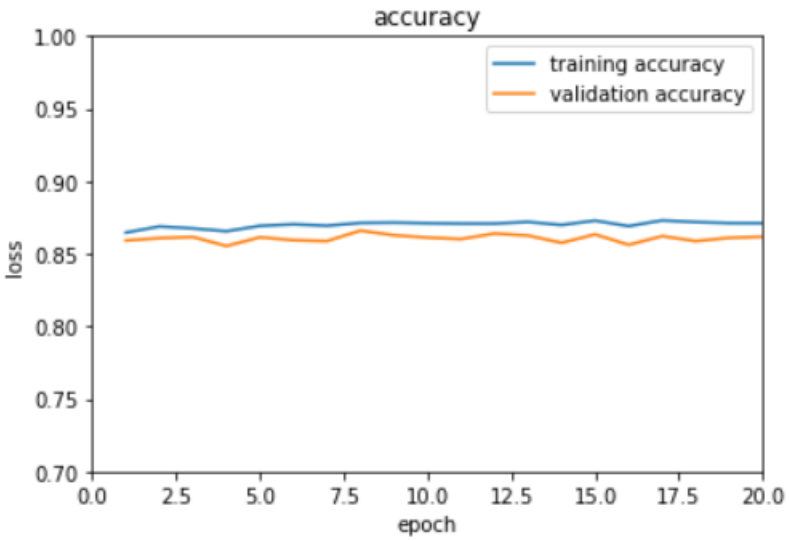
For this part we did the Weight_noise.py once with cpu and once with gpu we used this model because runs model 5 times with different learning rate and it has more computation than other methods.

The result was rather surprising: the cpu time was 64.61265681399982 and gpu time was 264.46867406. This result although intresting is reasonable. Because this was a single parameter tuning and wasn't paraleled in gpu and since cpu has more powerful cores it out-performed gpu.

2 Problem 2: Logistic Regression

2.1 Result of the log reg

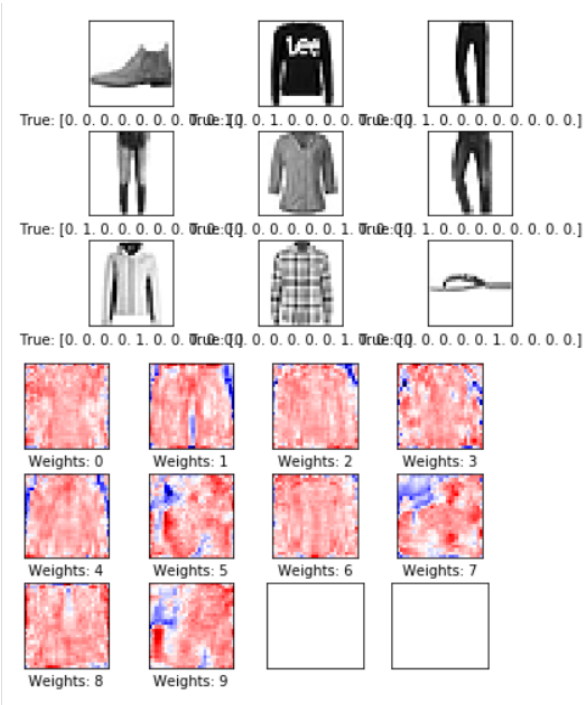
2.1.1 train/val accuracy graph



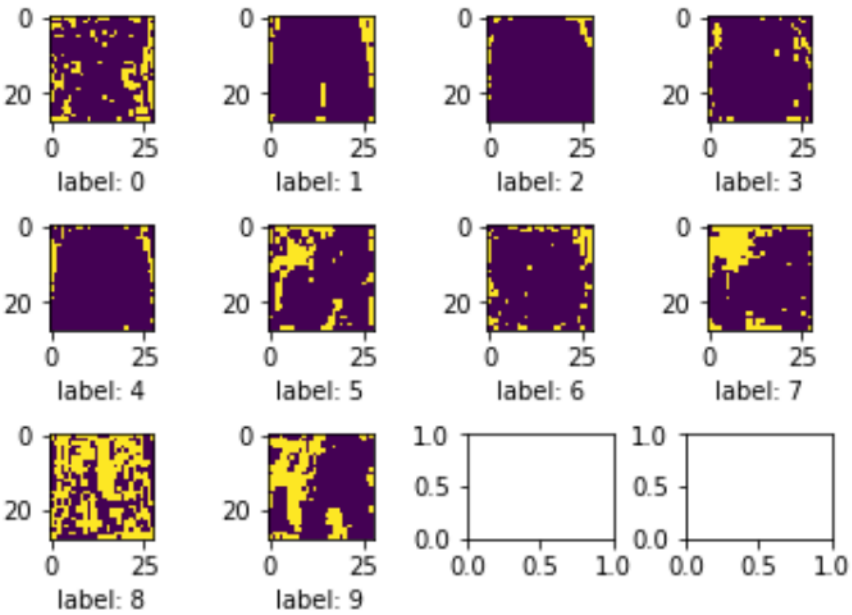
2.1.2 table of Train Val Test accuracy

Train	Test	Val
0.8711454	0.8424	0.8618

2.1.3 The plot of Effect of weights on each label



2.1.4 Weight Clusterings

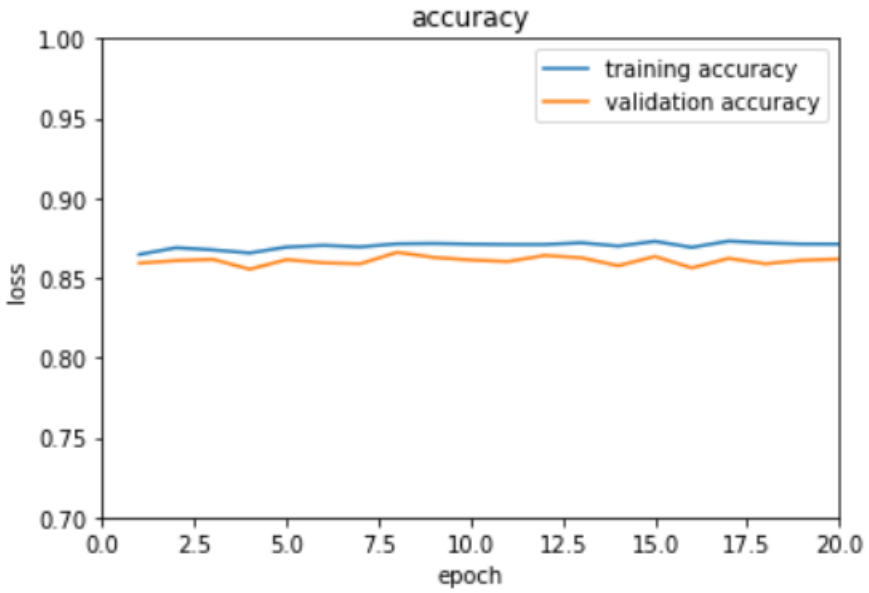


We choose 2 clusters to be able to differentiate between large value weights and low value or negative weights. As it can be seen in label 1 which is pants, it is obvious that it emphasis on dark pixels that has been shown in the plot and they actually look like pants. By doing 3 cluster it is expected to cluster weights into 3 clusters: high value weights, low value weights and negative weights which means that in case of presence of a color in that point it lower the probability of the label though as we drew the three cluster there was nothing interpretable and seems like 3 clusters are not what we expect them to be.

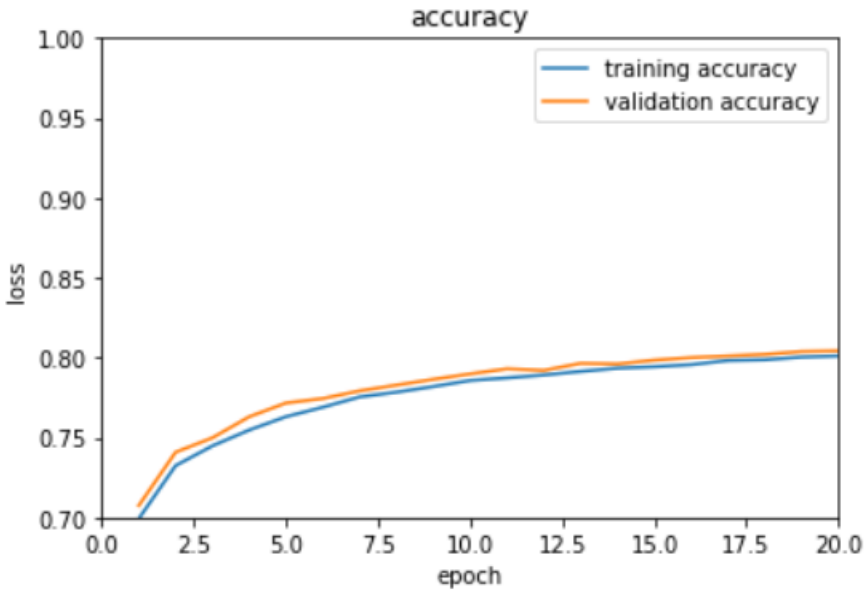
2.2 Effect of optimizer

Optimizer	time
ADAM	48.34
ADAGRAD	204.76
Gradientdcnt	210.27

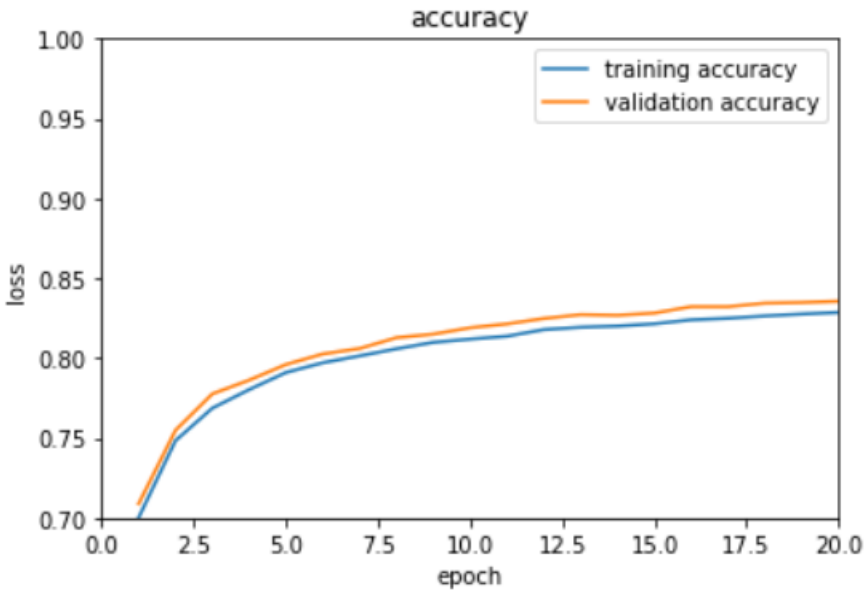
2.2.1 convergence graphs



adam graph



adagrad graph



gradient decent graph

2.2.2 Conclusion

As it can be seen for this specific problem ADAM optimizer converges with less epoch, each epoch takes less and it converges to a better point.

2.3 Longer epoch

In file *longer time(more epoch).ipynb* we run the optimization for 100 epoch instead of 20 and as we expected there is no improvement this could was expected just by looking at ADAMgraph from above. It has converges to in local minima with less than 10 epochs.

2.4 Train Val different split

In the file *changing train-val values.ipynb* the val dataset was shrinked from 5000 to 1000 and again no improvement was made. This was expected since we had 55k data and 4k is like 8 percnet increase in data. which is not much.

2.5 Effect of the batch size

For this part, in file *Batch size effect.ipynb* we went over three batch size: 200, 100 ,10 and the result was surprising. there was no performance difference between these three method while as the batches got smaller, the run time got smaller so the best choice would be the batch size of 10.

2.6 GPU VS CPU

for test the difference the ran cluster and GPU which are the base adam optimizer files with all alike parameters. by running on cpu we got 48.34s runtime while with gpu we got 101.25s of runtime. This result emphasises that GPUs are only useful in case of deep networks and in small scales, CPU is way better(in this case it was 2x faster)

2.7 Does the model overfit?

In this question we reached no point where trianing error decreases while the test error increases. So one can not be suspected to overfitting. However, *dropout.ipynb* addresses the potential problem by dropout. The only problem is that the best result we got in the test dataset is with no dropout.

This maybe due to insufficiency of training the dropout model cause this model need more training due to the fact that each time some nodes go inactive and wont get trained. Another this that was done is PCA. By doing so we shrinked the data space by a large margin so there won't be many parameters and although PCA is NOT a way to prevent overfitting, it may be helpful. in *PCA.ipynb* pca with 0.99% data reserve has been done and the result hasn't change by much so this method could be done to speed up the model. (PCA and dropout in the same model has been done in file $Dropput_PCA.ipynb$.

2.8 Random forest and SVM performance

In file random forest and SVM.ipynb the performance of best random forest with cross validation and the SVM with C = 1 has been calculated and the result is as follows:

Method	*accuracy*
Forest	*0.801*
SVM	*0.8398*

it can be seen that the logistic regression barely outperform these methods