CSC4007 ADVANCED MACHINE LEARNING

Assignment 1 report

03/2020

This Report discusses experiment results obtain from task 1 and task2, working dataset is based on different features of different classes of hand-written digits which import from sklearn.datasets.load_digits

Baibing Ji
Bji01@qub.ac.uk

Task3 in Task1

In task1, An Artificial Neural Network has been implemented from scratch using linear algebra instead of third-party libraries, according to requirements of task1, at the beginning of implementation, load-digits dataset has been imported and split to training and testing datasets which testing dataset has 20% data of the whole dataset, then different parameters such as weights, minibatch size, learning rate and number of epochs are also be set according to task description.

During the implementation stage, according to the task description, this ANN has 3 layers: one input layer at start, one hidden layer in middle and one output layer at end. For input layer, there are 64 features of each sample, so there are 64 neurons in input layer which each neuron represents one feature. For hidden layer, number of neurons have been set to 10 and use ReLu as activation function. For output layer, since there are 10 different classes in this dataset, so number of output layer's neuron has been set to 10 and use softmax activation function for multi-class classification. Working process of this ANN will be: select a sample, input all 64 features it has, processed by hidden layer's ReLu activation function and processed by output layer's softmax activation function, the output of this ANN will give possibilities for 10 different classes, just like this:

```
[8.20790202e-02 8.40101192e-02 9.95752611e-06 3.09148495e-02
 2.89983435e-03 3.95267348e-01 8.77503123e-02 1.90870789e-01
 1.24176280e-01 2.02148917e-03]
```

Figure 1. This is a sample output of ANN, there are 10 different possibilities

Usually, the class with the highest possibility will be the prediction of current sample, in figure 1, highest possibility is 0.395 in position 6, so current prediction of this sample is digit 5.

After 500 epochs, weights and bias have been updated, and following accuracy and error figures has been created:

```
Final Error:  0.3429839205427245
Final Accuracy:  0.9138888888888889
```

Figure 2. Final error and accuracy after 500 epochs update

91.38% is an acceptable accuracy.

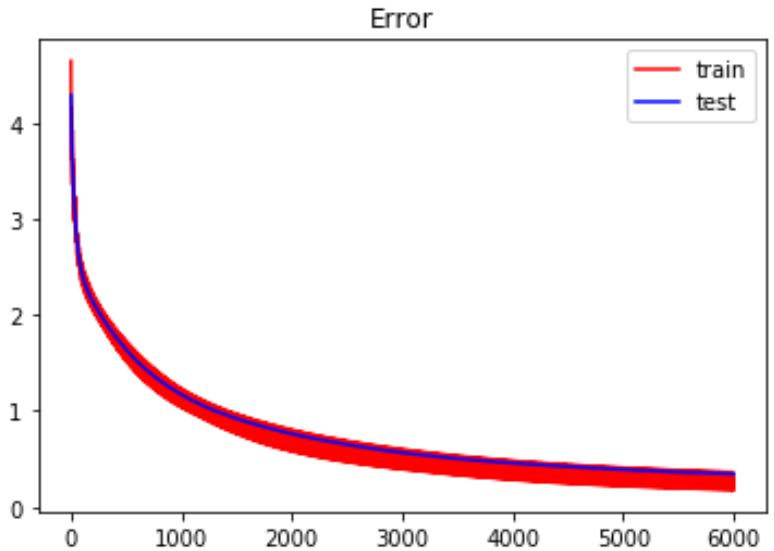


Figure 3. Error

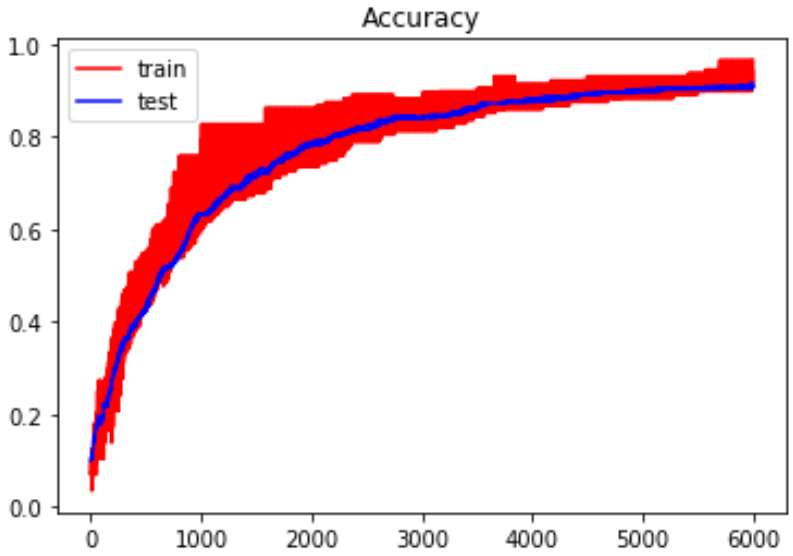


Figure 4. Accuracy

These two graphs show the training process is working since both training and testing error decrease but accuracy increase, but in both graphs, compare to error and accuracy of test data, the training error and accuracy are non-stable, if we training fewer epochs, this situation can be visualized clearer:

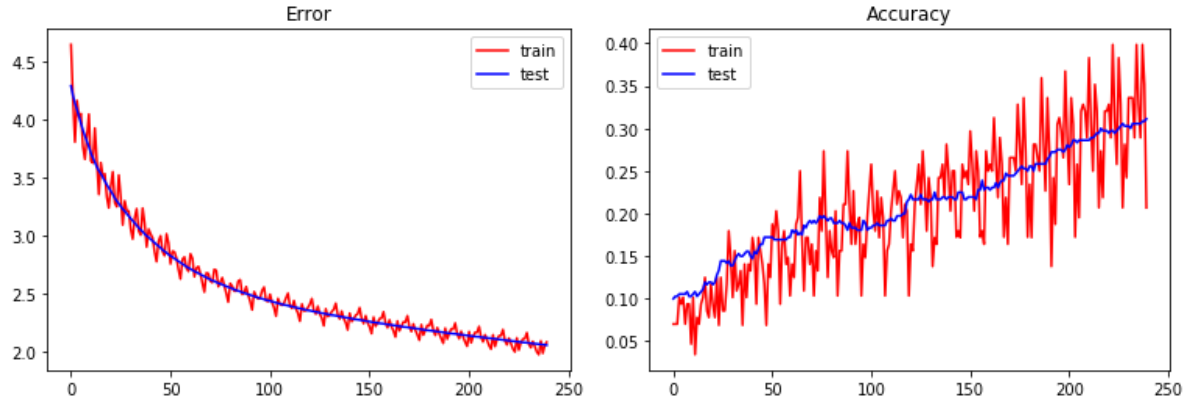


Figure 5. Error and Accuracy when epoch=50

It can be found that training error and accuracy fluctuated during the whole training process, this may because some portion of examples is classified randomly.

Also, it can be found that after calculating 3000-4000 times, the error and accuracy become stable, this may mean model are no longer be able to learn too much information from dataset and it can be predicted that as training times increase, the accuracy and error of this model will change in a very small interval.

Task3 in Task2

In task2, datasets and parameters are same as task1, the only different is task 2 using 10 fold cross validation to evaluate model which created in task1, task2 also visualize error, but replace accuracy to f1-score, f1-score is used to balance the influence of precision and recall, by using f1-score people can evaluate a model (classifier) comprehensively. After performing cross validation, model's training error for every 100 epochs, each fold's f-score and final average f-score are printed out as following screenshots:

```
Training error of fold 1 epoch 0 is: 6.077740559982455
Training error of fold 1 epoch 100 is: 1.0594304190454154
Training error of fold 1 epoch 200 is: 0.6194675510749282
Training error of fold 1 epoch 300 is: 0.4177586361026791
Training error of fold 1 epoch 400 is: 0.31837559846245406
Fold 1 's f-score is: 0.8973113056336672
Training error of fold 2 epoch 0 is: 4.730870697892228
Training error of fold 2 epoch 100 is: 0.5822308047842655
Training error of fold 2 epoch 200 is: 0.317307547463822
Training error of fold 2 epoch 300 is: 0.24398369073080461
Training error of fold 2 epoch 400 is: 0.20093963401118486
Fold 2 's f-score is: 0.9339063410925121
Training error of fold 3 epoch 0 is: 4.862286853745861
Training error of fold 3 epoch 100 is: 1.068523774805214
Training error of fold 3 epoch 200 is: 0.4983904145978191
Training error of fold 3 epoch 300 is: 0.3382123281431242
Training error of fold 3 epoch 400 is: 0.27025811656532395
Fold 3 's f-score is: 0.9204012380501869
Training error of fold 4 epoch 0 is: 4.241345136733649
Training error of fold 4 epoch 100 is: 1.0483175948151502
Training error of fold 4 epoch 200 is: 0.6368537565335702
Training error of fold 4 epoch 300 is: 0.45966875957391945
Training error of fold 4 epoch 400 is: 0.37290699039544506
Fold 4 's f-score is: 0.8977879211193811
Training error of fold 5 epoch 0 is: 3.1629444735894285
Training error of fold 5 epoch 100 is: 1.0555945417075705
Training error of fold 5 epoch 200 is: 0.6107063025694747
Training error of fold 5 epoch 300 is: 0.42090132473991954
Training error of fold 5 epoch 400 is: 0.3364338808504123
Fold 5 's f-score is: 0.9269771916283546
Training error of fold 6 epoch 0 is: 4.2077820969113295
Training error of fold 6 epoch 100 is: 1.0823367869634555
Training error of fold 6 epoch 200 is: 0.5255940504531251
Training error of fold 6 epoch 300 is: 0.3330653871743918
Training error of fold 6 epoch 400 is: 0.247296426367889
Fold 6 's f-score is: 0.9053912915346801
Training error of fold 7 epoch 0 is: 4.098104708967188
Training error of fold 7 epoch 100 is: 1.5276457863453758
Training error of fold 7 epoch 200 is: 0.6405222402194652
Training error of fold 7 epoch 300 is: 0.40547220942744294
Training error of fold 7 epoch 400 is: 0.31270478751790365
Fold 7 's f-score is: 0.8926329296445645
Training error of fold 8 epoch 0 is: 3.706689434784735
Training error of fold 8 epoch 100 is: 0.8043388639967525
Training error of fold 8 epoch 200 is: 0.41871834413884174
Training error of fold 8 epoch 300 is: 0.2974351646870324
Training error of fold 8 epoch 400 is: 0.24252555996247632
Fold 8 's f-score is: 0.944541481909903
Training error of fold 9 epoch 0 is: 3.3378536180874243
Training error of fold 9 epoch 100 is: 1.0335712596848208
Training error of fold 9 epoch 200 is: 0.6485316573496713
Training error of fold 9 epoch 300 is: 0.5154714467774395
Training error of fold 9 epoch 400 is: 0.4290292886713275
Fold 9 's f-score is: 0.8198717627401839
Training error of fold 10 epoch 0 is: 3.1221874023226395
Training error of fold 10 epoch 100 is: 1.3270108205707538
Training error of fold 10 epoch 200 is: 0.8640324678526551
Training error of fold 10 epoch 300 is: 0.6597223810818137
Training error of fold 10 epoch 400 is: 0.5083393227152281
Fold 10 's f-score is: 0.8301318435606362
Average F-score across 10 folds is:  0.896895330691407
```

Figure 6. Every 100 epoch's training error change, each fold's f-score and average f-score

As we can see f score of each fold is changed between 0.82 and 0.93 which changing rate is about 13%, but the interval of different fold's f score can be different, in some aspects f1 score also shows imbalance between different classes in each fold. Last fold has the highest final training error is because last fold's dataset is less than other folds. However, at last the average f score across 10 folds is nearly 0.9, usually, f score reaches its best value at 1 and worst value at 0, this model's f score is 0.9 which is quite close to 1 also prove this model is relatively great.

For training error, different folds have different initial training error and final training error, but all of them are keep decreasing when training more times but the interval between two 100 epochs becomes smaller after each training.

Task4

In this part, I plan to test performance change in three different parts:

1. First I will change those parameters given by task description such as split ratio, mini batch size, learning rate and number of epochs etc. to test how these parameters affect performance. In this part, I will use given value in task 1 as default value, when I test one part I will keep other parts in default, at the same time, the structure of ANN will also use the same structure in previous tasks.
2. Second I will explore how structure change affects ANN performance such as change activation functions, change number of neurons in hidden layers etc.
3. Third, I will analyze different results in previous parts in task 4, combine different structures and parameters and try to get the best results.

I will use tables, graphs and screenshots to help explain different results.

Note: all the results are tested in this specific dataset may not be generalized to other datasets, if there's no specific comments, error and accuracy in following tables are test error and test accuracy.

Part 1

1. Weight

For weight initialization, we usually use uniform distribution, there are also other initialization methods such as He-et-al initialization etc. I planned to try three different initialization for weight: all zero, all one and normal distribution, some other initialization method will also be mentioned but won't be used in final model.

If initialize weight to all 0, we can get following tables and graphs:

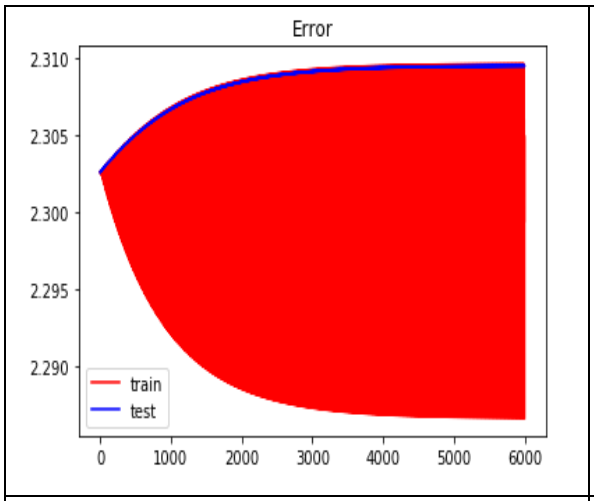
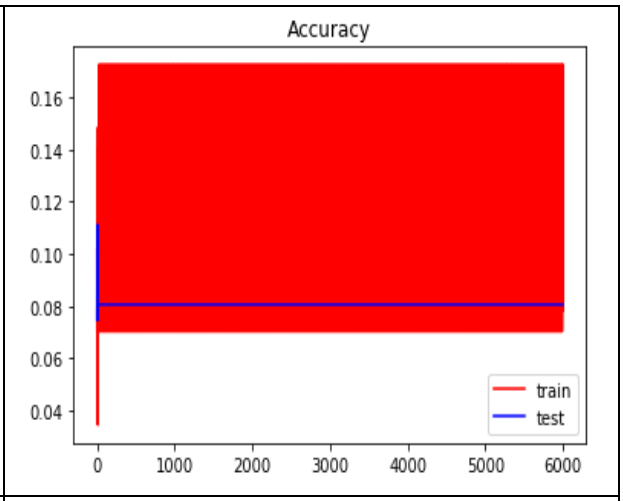
	
Final error: 2.31	Final Accuracy: 0.08

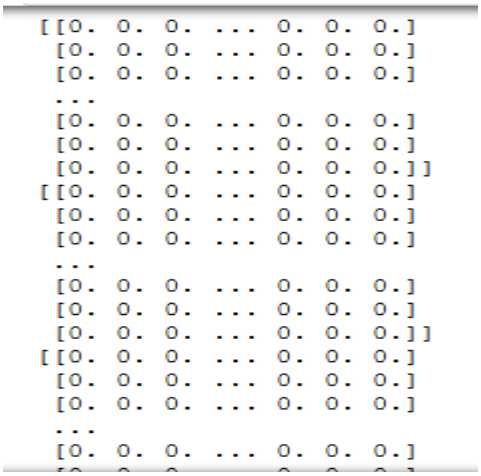
Table: Weight initialized to all 0

As we can see from previous table, the test error keep increasing and test accuracy is not change, this because weight initialized to 0, the activations in all hidden units are also the same.

Input contains NaN	Input contains NaN
N/A	N/A

Table: Weight initialized to all 1

As we can see from previous tables and graphs, if we initialized weights to all zero or all 1, there will be some error of results, if we initialize all weights to 0, every hidden layer's neuron will get 0 value as input just like the screenshot:



```
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
```

This means all of the neurons in hidden layer will follow the same gradient and always do same things between each other, this also makes bias can't affect the training.

The same situation happens when initialize all weights to 1, the input will become to this:

```
[[21.7168956  21.7168956  21.7168956  ... 21.7168956  21.7168956
  21.7168956 ]
 [16.85912698 16.85912698 16.85912698 ... 16.85912698 16.85912698
  16.85912698]
 [17.1875     17.1875     17.1875     ... 17.1875     17.1875
  17.1875    ]
 ...
 [19.8125     19.8125     19.8125     ... 19.8125     19.8125
  19.8125    ]
 [17.8125     17.8125     17.8125     ... 17.8125     17.8125
  17.8125    ]
```

All of the input will become a quite large value and for single epoch input values are the same. So initialize weights to all 0 or all 1 may not a good idea because it can't break symmetry, all 0 or all 1 weights will stop network's learning process and may get unexpected results.

I also tried random normal distribution, I set mean to 0 to make generated numbers around 0, and I tried different standard deviation and got following tables:

Standard Deviation	Error	Accuracy
0.01	0.503	0.858
0.1	0.41	0.886
0.5	0.52	0.828

Table: different error and accuracy for different standard deviation

As experiments, when standard deviation become larger, training error will decrease faster but will get higher overall error, however, as an example, when standard deviation set to 0.1, I got following graph:

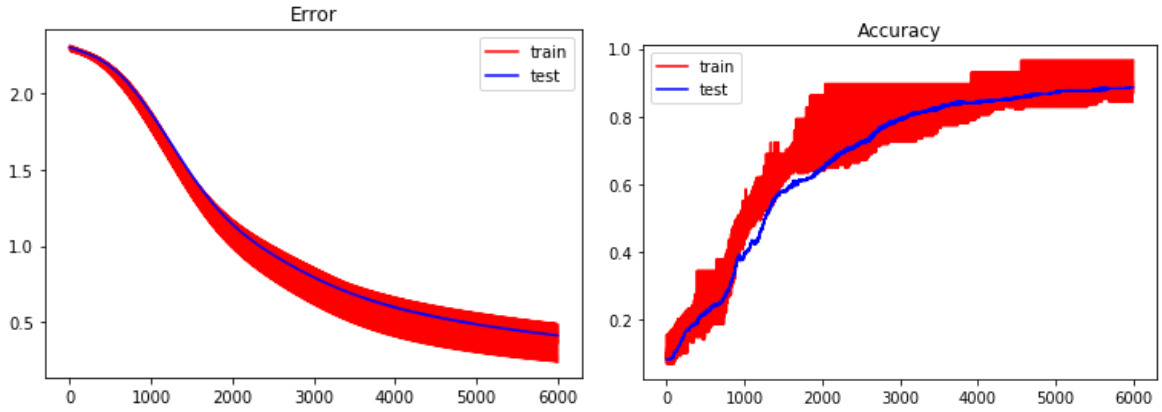


Figure: Error and accuracy graph for normal distribution when mean=0 sd=0.1

As we can see from previous graph, the training process are slower than uniform distribution, fluctuate of error and accuracy also become larger, final accuracy and error are also worse than uniform distribution. So, for ReLu activation function, use uniform distribution may help model get better performance.

Different random distributions can be considered as a not too bad but relatively easier way to make ANN work, however, random distributions also have some problems, sometimes random initialization will make training use longer time, at the same time, specific combinations of weight initialization method and specific activation functions may cause vanishing/exploding gradient problem.

There are still many other great weight initialization methods such as Glorot and Kaiming He initialization method, but these methods won't be tested in this report since they need to import third-party library Keras which not match assessment criteria.

2. Bias

For bias, initialize bias to 0 is possible and useful, other than initialize bias to 0, some people like to initialize bias to 0.01 since this ensure all ReLu unit can be activated at first, after testing this I got following tables and graphs:

```
Final Error:  0.33676478787188896
Final Accuracy:  0.9138888888888889
```

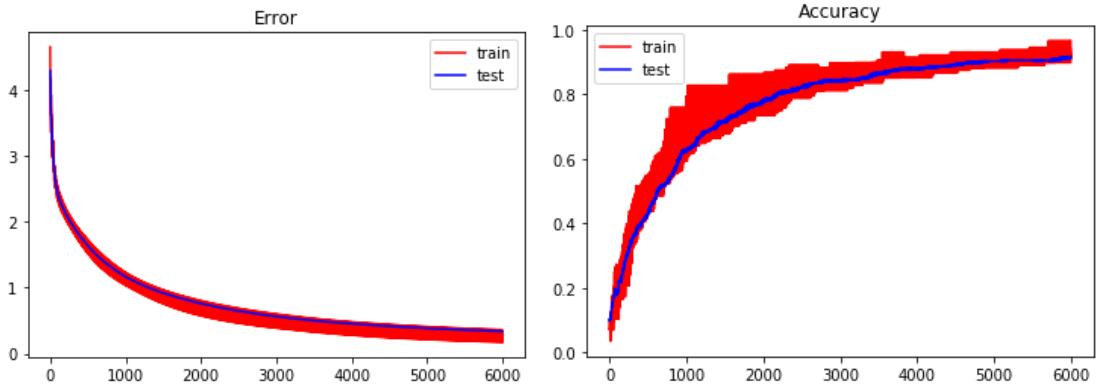


Figure: Error and accuracy when bias initialized to all 0.01

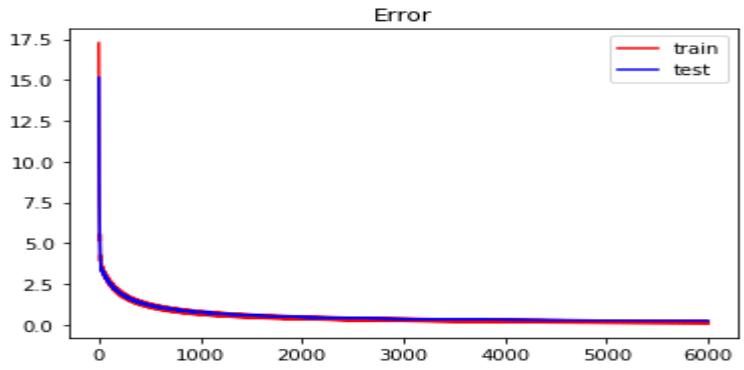
I can't find too much different from graphs, and accuracy also not changed, but the final error is dropped from 0.34 to 0.33.

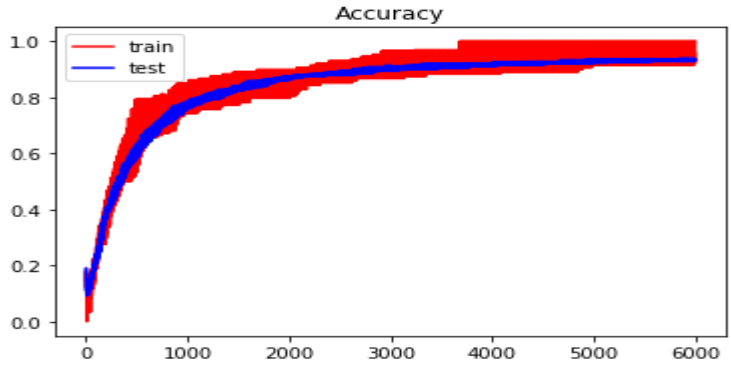
I try to initialize bias to all one:

```
Final Error:  0.26446824620703324
Final Accuracy:  0.9222222222222223
```

Initialize to 10:

```
Final Error:  0.24849299682943007
Final Accuracy:  0.9305555555555556
C:\ProgramData\Anaconda3\lib\site-packages\matplot
e://ipykernel.pylab.backend_inline, which is a no
  % get_backend())
```





As we can see, if we keep using larger number to initialize bias, training error and accuracy will reach nearly 0 error and 100% accuracy faster.

From previous experiments for bias initialization, I can't say how initialize bias to a larger number will affect network's training, according to Stanford CS231N notes [1], sometimes use smaller natural number such as 0.01 can ensure ReLu units will be activated at the beginning, but some results shows this may cause worse results, but it can be ensure that if initialize bias to a large value will cause unexpected affect to neural network.

3. Other parameters

Split ratio:

For split ratio, actually, this problem is same as: how increase or decrease size of training dataset will affect final result, but obviously, I cannot set test data ratio to 0 since there will be no data for testing, I experiment ratio decrease to 0.1 and increase to 0.3 and 0.5 to test influence:

Test data ratio	Error	Accuracy
0.2 (default)	0.343	0.914
0.1	0.316 (decrease)	0.917(increase)

0.3	0.362(increase)	0.904(decrease)
0.5	0.459(increase)	0.863(decrease)

Again, set test data ratio to 0.1 will increase size of training data, which may make network get more information from dataset and features, this will decrease error and increase accuracy, set test data ratio to 0.3 and 0.5 make performance worse since it make training set smaller, however, set lower test ratio is not always a good choice, especially in small dataset, for example, a small dataset contains 10 samples for 2 classes, 80/20 ratio for splitting, there are 8 samples for training 2 for testing, if a model doesn't trained but simply guess what class it should be, we have ¼ chance to have a right guess, if test set decrease to 1, the percentage for right guess increase to 50%. From previous example, use 20% or 10% test set ratio will not affect too much to large datasets but small test set is not a not good choice for small dataset.

80/20 is a common ratio which often referred to Pareto principle, however, Different ratio can be used according to different actual situation (overall dataset size).

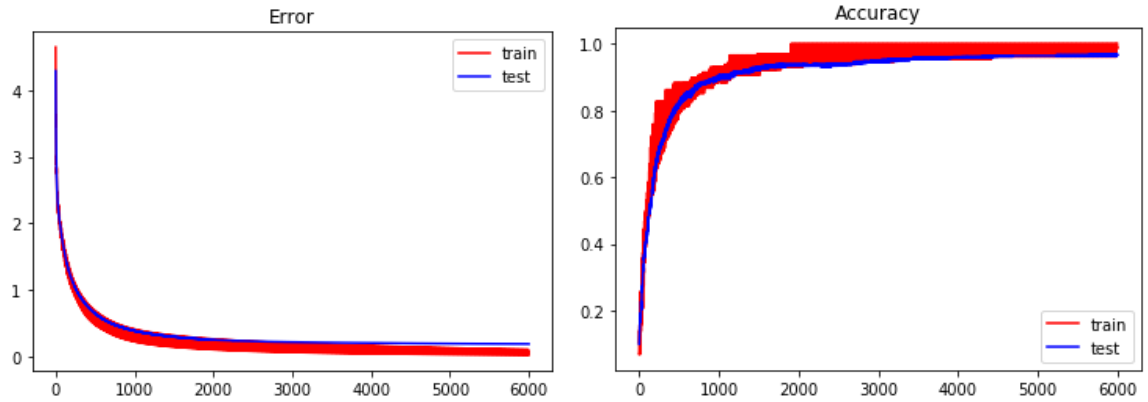
Mini-batch size:

Batch size	Error	Accuracy
64	0.240(decrease)	0936(increase)
128 (default)	0.343	0.914
256	0.547(increase)	0.861(decrease)

As we can see from previous table, smaller batch size will make training more stable, training will be faster, make quality of model better, and finally decrease error and increase accuracy, this maybe because smaller batch size are 'noisy' which will offer regularizing effect and lower generalization error. Large batch size tends to converge to sharp minimizers, however, sharp minima lead to poor generalization then make quality of model decrease and finally affect performance.

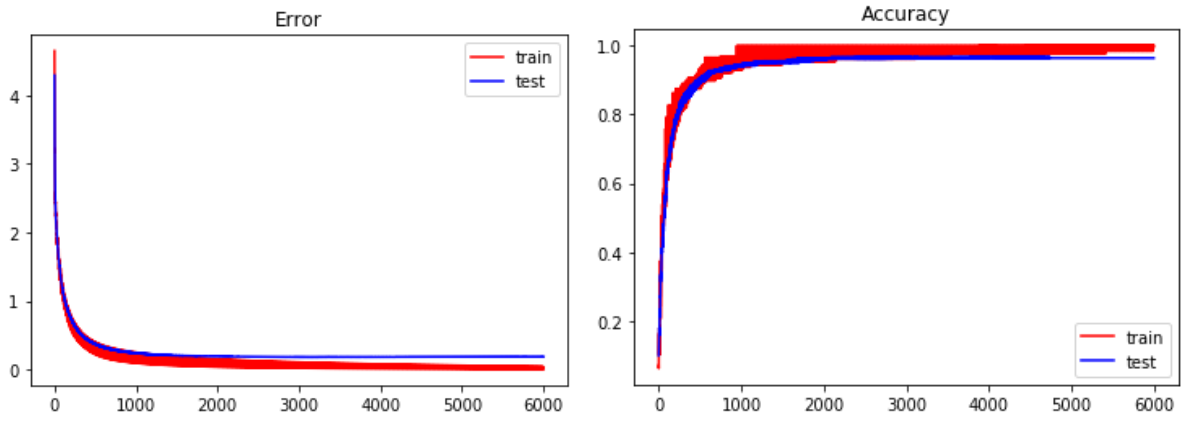
Learning rate:

As we know, higher learning rate sometimes can make training faster, and lower learning rate will make training slower, here's some experiments:




```
Final Error:  0.18940628313988117
Final Accuracy:  0.9666666666666667
```

Graph: learning rate set to 0.05




```
Final Error:  0.19162535714901854
Final Accuracy:  0.9638888888888889
```

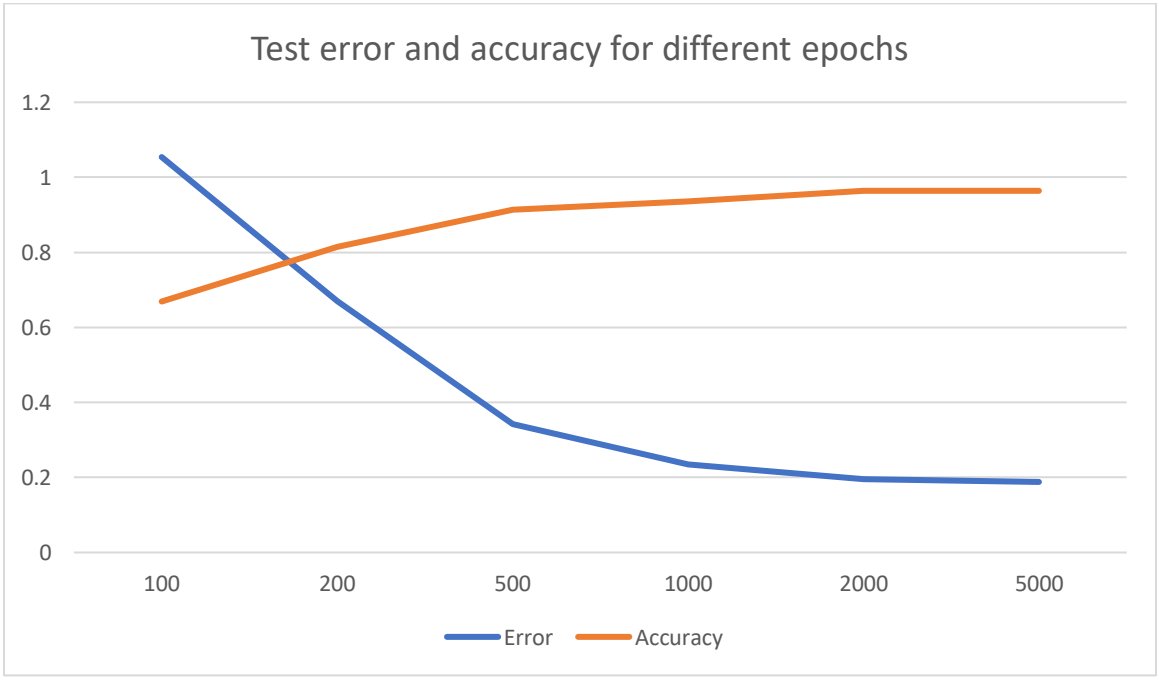
Graph: learning rate set to 0.1

As we can see from previous graphs, compare to default 0.01 learning rate, higher learning rate can make training faster, however, graphs' training error and accuracy decrease to nearly 0 and 100% rapidly which shows the learning rate is too high.

Training epochs:

More epochs means more training times, usually when number of epochs increased, model can learn more from input data, and I got following table:

Epochs	Error	Accuracy
100	1.054	0.669
200	0.67	0.814
500	0.343	0.914
1000	0.235	0.936
2000	0.195	0.964
5000	0.188	0.964



As we can see from previous charts, increase number of epochs can increase overall accuracy and decrease error. In early stage 100, 200 and 500 epochs, increase training times can make performance increase faster, after 1000 epochs, model cannot learn too much from dataset and the trends of error and accuracy become more stable. However, increase training times also have its drawback, time period for training model will become longer, especially for larger dataset (bigger model), sometimes need a balance between the time consumption and performance.

Conclusion for part 1:

Experiments above told us set parameters to extreme high or very low are not always a good idea for neuron network's performance:

Weights and bias have some agreements of initialization, initialization of these two parameters will change the whole model's training and final performance, each of them has their own initialization method in most of the time: normal distribution or uniform distribution for weight initialization, all 0 or 0.01 for bias initialization, they have a name 'parameter' which means they will be updated during learning process.

Other parameters are hyper parameters: the value of them are always set according to experience or different situation (dataset size etc.), model can get more information from larger training set, but smaller test set cannot fully prove model's performance, smaller batch size will make model better, increase learning rate is a good idea to make training faster but large learning rate will also make model lose more information during training, more training epochs can increase final performance but will also increase training time period, hyper parameters will also affect weight and bias's value after training, different hyper parameters settlement has their own advantages and disadvantages.

Also, it is worth to mention that there exist hyper-parameter interactions, for example, batch size may interact with other hyper parameters such as learning rate.

Overall, people should be careful about parameter and hyper parameter setting since they can make model better, but inappropriate parameter setting will also destroy neuron network, more experiments are encouraged because it can help get better parameters.

Part 2

In this part, I will try different activation functions and different number of neurons in hidden layer, because of this dataset refers to multiclass classification, I will keep softmax as output layer's activation function but try different number and structure of hidden layer.

According to online resources, very few experiments will adding more hidden layer (second or third) to try to increase performance, one hidden layer can solve most of the problems, so this report won't change the amount of hidden layers.

So, in this part, I will try following things:

2.1 keep number of neurons unchanged but try different activation functions

Result: how different activation function affect performance

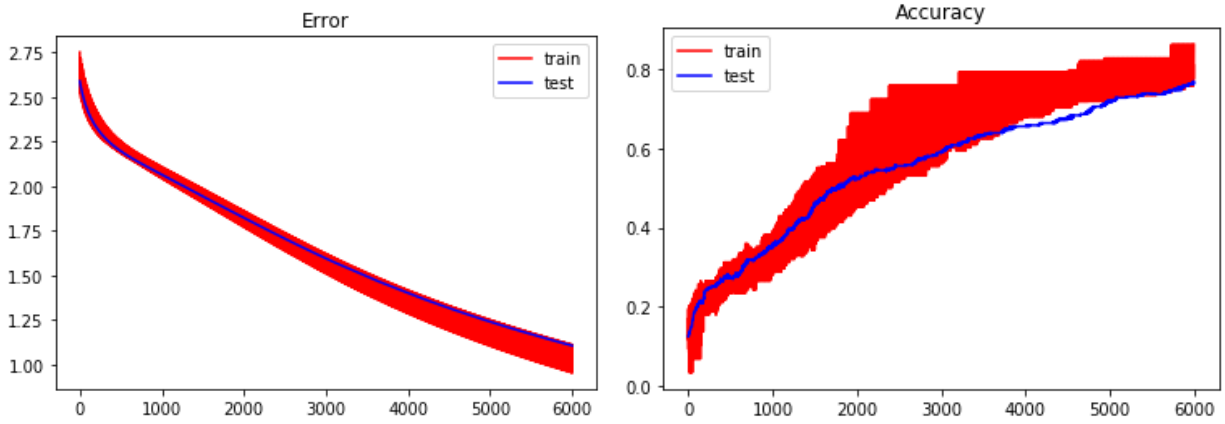
2.2 keep activation function as default (ReLu) but change number of neurons in hidden layer

Result: how different number of neurons in hidden layer affect performance

1. Change hidden layer's activation function to sigmoid

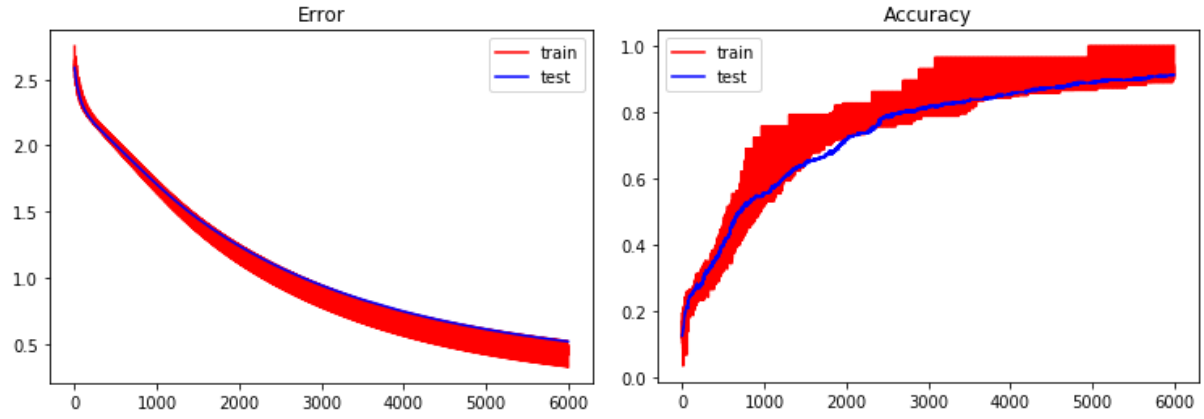
If I keep number of hidden layer's neuron unchanged but change activation function to sigmoid, and I got following graph and result:

```
Final Error:  1.1068059677678659
Final Accuracy:  0.7638888888888888
```



If we use default parameters, seems sigmoid 's performance are not great as ReLu's, but according to the error curve, seems we need to adjust learning rate because it learns too slow and not fully converged, so I adjust it and try to make the training accuracy reach 100% and I set learning rate to 0.025 got following results:

```
Final Error:  0.5179874460678847
Final Accuracy:  0.9138888888888889
```



After change the learning rate, the accuracy of test set become 91.38% which is same as default ReLu's accuracy, however, the test error is higher than default and both the training and test error curve are still not converge enough.

If I increase number of epochs to 1000, test error decrease to 0.30.

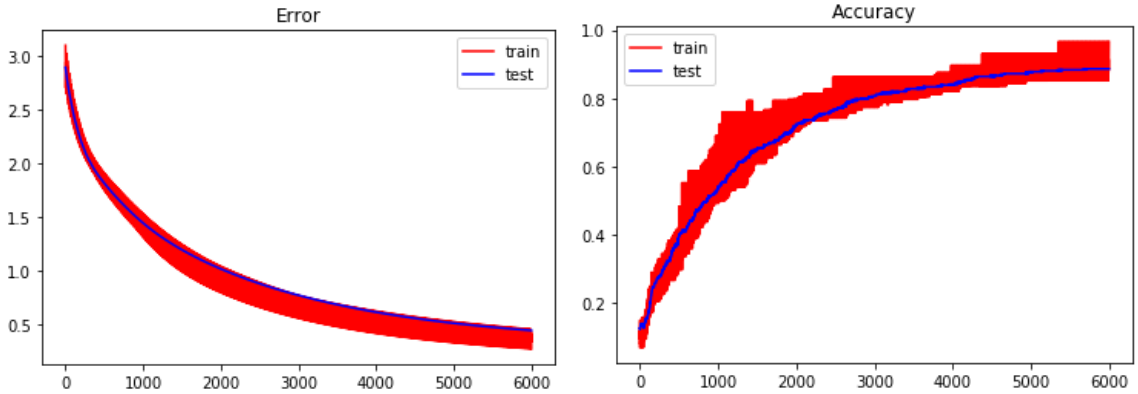
From previous test for sigmoid activation function, we can find that if we want to get similar result as ReLu, sigmoid activation function need more training, the reason that sigmoid activation function is slower than ReLu may because sigmoid activation function need to calculate e^x, it's compute-intensive, in order to improve model performance we won't choose sigmoid activation function for this dataset.

2. Change hidden layer's activation function to Tanh

For tanh activation function, compare to sigmoid, it overcomes non-zero centric issue of sigmoid activation function, the formula can simplify to: 2*sigmoid(2x)-1.

If I keep number of hidden layer's neuron unchanged but change activation function to Tanh, since tanh can be seen as optimized sigmoid function, so output is also similar:

```
Final Error:  0.4491310569532366
Final Accuracy:  0.8861111111111111
```



As we can see, in default parameter, tanh get better test error than sigmoid but worse accuracy, at the same time, tanh activation function in default parameter's curve is similar to sigmoid's curve after increasing learning rate, so we can get that tanh has better performance than sigmoid but still worse than ReLu.

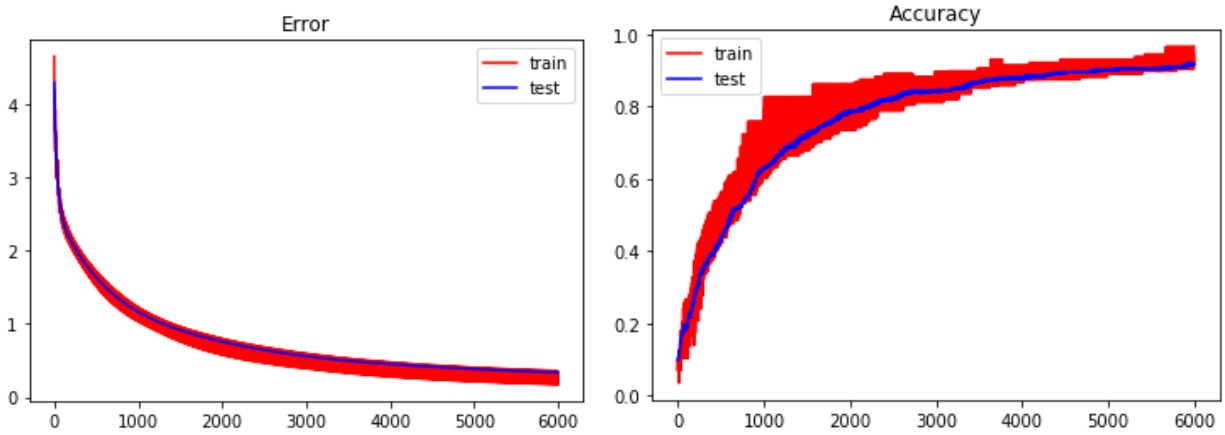
Previous two activation functions sigmoid and tanh are both non-linear function so they can capture more complex patterns and they are also continuously differentiable, there are some difference between output value of these two activation functions which sigmoid output is between 0 and 1, tanh is -1 to 1, they may better in binary classification but they may suffer from vanishing gradient, in our dataset, tanh's performance is better than sigmoid but they both worse than ReLu.

3. Change hidden layer's activation function to LReLu

Leaky ReLu is an optimized activation function of ReLu, it attampts to solve "dying ReLu problem", consider that bias takes or initialized to a large negative value, the weighted sum of inputs will close to 0 and neuron will not be activated, and this is neuron dies, as I mentioned above, initialize bias to 0.01 is one of the way to solve neuron dies problem, but LReLu is another way to avoid the same problem.

I use LReLu function with default parameters and I got following result and graphs:

```
Final Error:  0.33992672852304046
Final Accuracy:  0.9138888888888889
```



As we can see from previous results, in default parameters, Leaky ReLu can get same accuracy on test dataset but better error (ReLu is 0.343, LReLu IS 0.34).

I also use 10 folds cross validation to validate LReLu's performance is better than ReLu, the result shows LReLu always has better test error than ReLu, but sometimes ReLu's accuracy is better than LReLu and 10 folds' average f-score is slightly lower than ReLu.

There are also many advantages of Leaky ReLu such as no saturation problem in both positive and negative region, easy to compute and close to zero-centered functions.

According to result, My choice will between ReLu and Leaky ReLu for final model.

4. Change number of neurons in hidden layer but keep ReLu as activation function

In ANN, hidden layer is required if data can't be seperated linearly, however, it's not easy to choose number of hidden layer's neuron, usually it depends on type and size of datasets, in following experients, I will test how different numbers of neurons affect performance.

In default, there are 10 neurons in hidden layer, according to Jeff Heaton's article [2], there are some rules for choosing number of neurons in hidden layer:

4.1 The number of hidden neurons should be between the size of the input layer and the size of the output layer. (10-64)

4.2 The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer. (32-34)

4.3 The number of hidden neurons should be less than twice the size of the input layer. (less than 128)

According to these rules, the number of neurons should between 10 but less than 128, in our dataset, the default neuron numbers is 10, which is the lower bound of previous rules, however, I will try different numbers to show more comprehensive result, so I plan to try:

4.1.1 5 neurons (less than default, patialy prove rule 4.1)

4.2.2 32-34 neurons (rule 4.2)

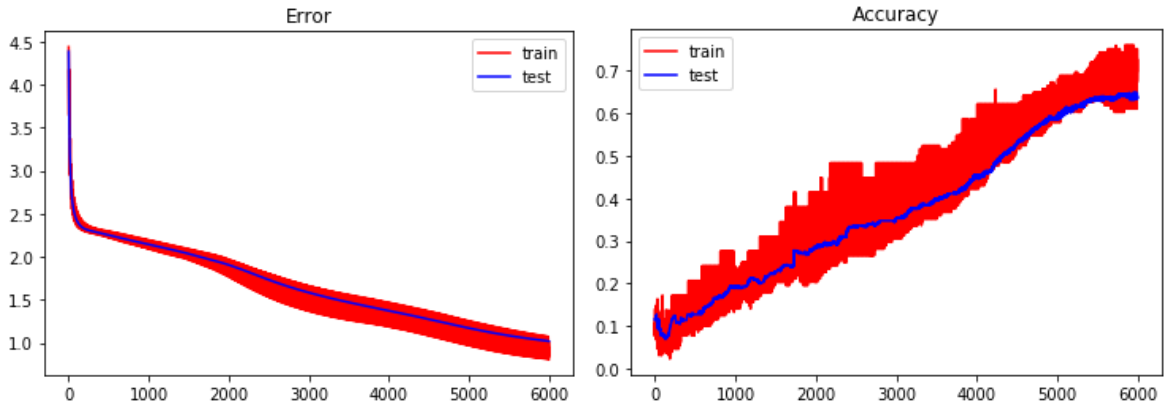
4.3.3 64 neurons (combine rule 4.1 and 4.3 and has the same as number of features)

Before experiment start, it is not hard to guess different number of neurons will cause different results, for example, there are 64 features of each sample, first, 64 features will scaled to 10 measurements, and since output layer has 10 neurons for 10 classes, hidden layer 10 neurons and output 10 neurons are equal, if hidden layer's neuron is less than output layer eg. 5, the progress will be 64-5-10, which will scale twice, seems more scale will lose more information. However, if hidden layer's neuron is more than 10, the total progress could be seen as keep shrinking's process, which may loss less information.

4.1.1 5 neurons

I use default paramters but change number of neurons in hidden layer to 5, I got following graph and results:

```
Final Error:  1.0208634322135828
Final Accuracy:  0.6416666666666667
```



As we can see from previous result, the performance decrease a lot, this proved if hidden layer's neuron less than output layer, it will make performance decrease.

4.2.2 32-34 neurons

I use default paramters but change number of neurons in hidden layer to 32-34, I got following graph and results:

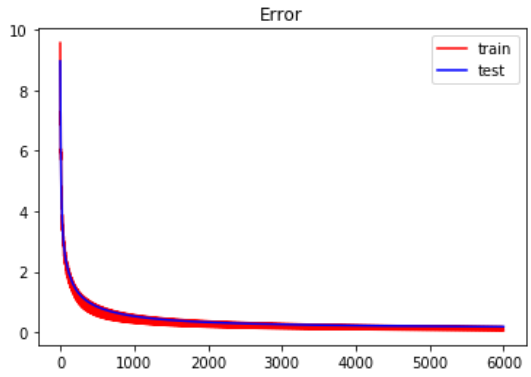
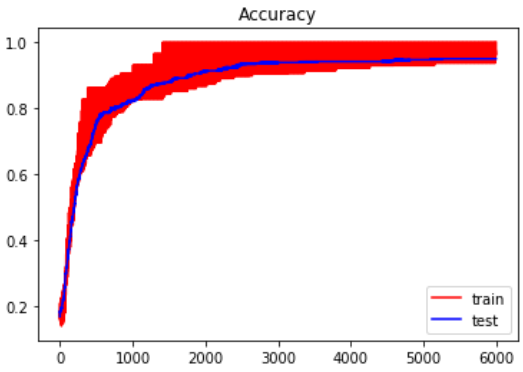
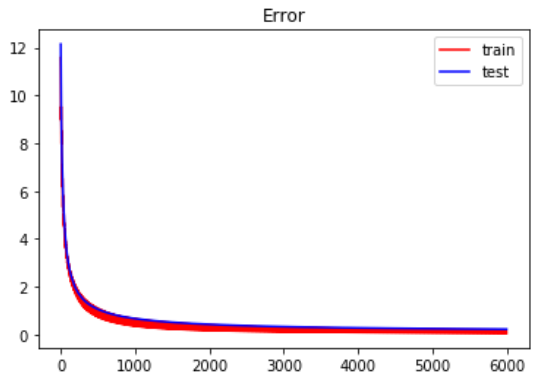
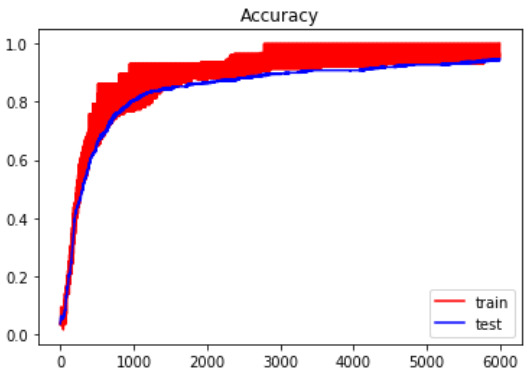
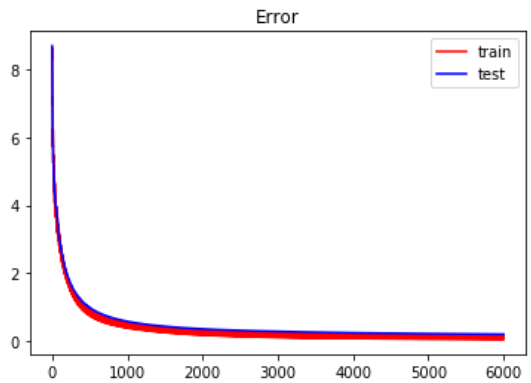
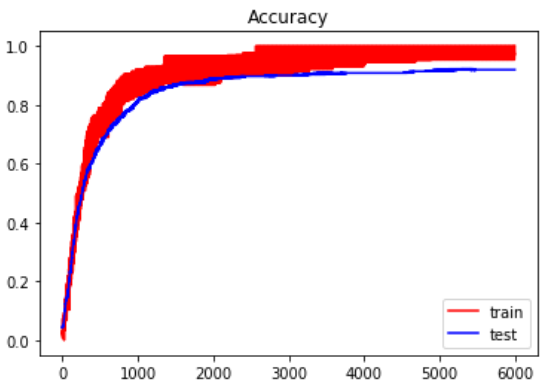
Number of neurons	Error	Accuracy
32		
33		
34		

Figure: 32-34 neurons' error and accuracy graph

Number of neurons	Error	Accuracy
32	0.185	0.95
33	0.224	0.947
34	0.190	0.919

Table: 32-34 neuron's error and accuracy table

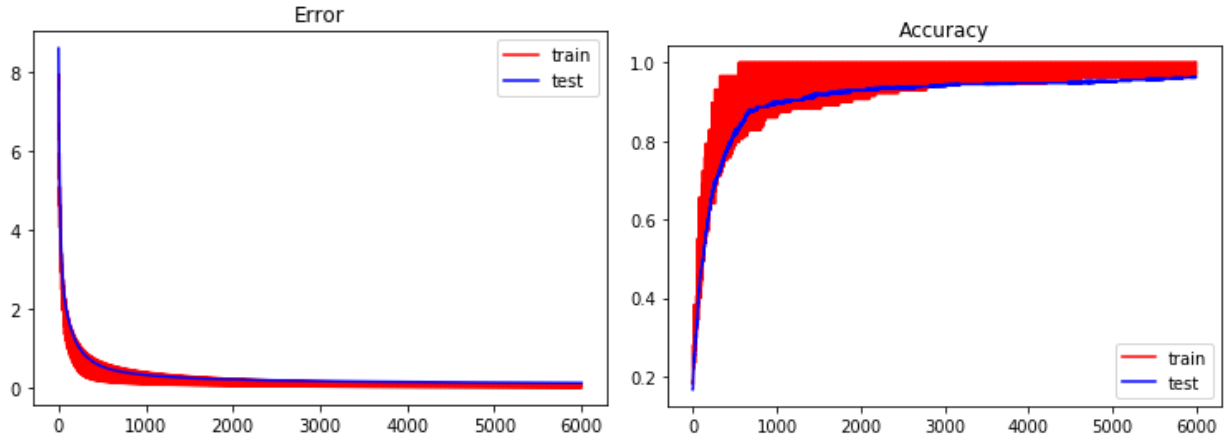
As we can see from previous tables and graphs, there are not too much difference between error graphs, but as number of neurons increase, the fluctuate of training accuracy beceme smaller, is this means more neurons in hidden layer will make model more stable? From

previous table, the test error and accuracy are slightly different, when number of neurons is 32, model has best performance in dataset.

4.2.2 64 neurons

I use default paramters but change number of neurons in hidden layer to 64, I got following graph and results:

```
Final Error:  0.136848749416088
Final Accuracy:  0.9638888888888889
```



As we can see from previous result, the performance keep increasing, but result is a little bit strange for me, so I decide to use 10 fold cross validation to test them again.

4.3.3 10 fold cross validation for different number of neurons

In this additional part, I will use average f1 score to validation last part's result:

Number of neurons	F1-score
5	0.813
32	0.95
33	0.949
34	0.948
64	0.955
128	0.948

According to table above, most of the f1 score prove results in last part may not have too much error, however, I add another amount of neuron 128 to test the upper bound of neurons according to rules, and finally I got 0.948 f1 score, compare to 64 neurons the f1 score decreased, as we can see, increase number of neurons may increase performance of neuron networks, but the amount of neurons in hidden layer should not be too large, number of neurons should reference actual situation so we can got better performance.

4.4 Final result

According to previous experiments, I will use normal distribution to initialize weight, initialize bias to all 0, split ratio 0.2, 64 as minibatch size (smaller mini batch size) , 1000 as training

epochs (more training times) and 64 neurons (more hidden layer's neuron) in hidden layer, after trying different learning rate and different activation functions, I got following results and graphs as final model:

Learning rate 0.1, activation function Leaky ReLu:

Error: 0.1006

Accuracy: 0.972

F1-score: 0.970

Conclusion

In this report, I discussed results obtained from basic model, and I also tried how different parameters, hyper-parameters, activation functions and neuron network structure will affect final performance, got experience about tuning different parameters. These experiments told me there's no fix method to tune different parameters, different settings should be used to difference situation. In final model, in order to get better performance, I increase the number of training times, this make training and validation times become longer but can get a little better performance. From this report, I also got some experience to help me saving time when trying different parameters such as I won't initialize weight to all 0 or all 1 because that is a wrong method to initialize weight. People should change parameters and training times according to specific requirements.

Bibliography

1. http://cs231n.github.io/neural-networks-2/
2. Heaton Research. (2020). *The Number of Hidden Layers*. [online] Available at: https://www.heatonresearch.com/2017/06/01/hidden-layers.html [Accessed 5 Mar. 2020].