Report of Connectionist Computing

MLP network

YiKai. Wang

15206086

1. Introduction of the code structure:

Class MLP parameters:
self.Num_of_Inputs = 0 # number of inputs.
self.Num_of_Hidden = 0 # number of hidden layers.
self.Num_of_Outputs = 0 # number of outputs.
self.Input_Neurons = [] # Neurons number of input.
self.Hidden_Neurons = [] # Neurons number of hidden.
self.Output_Neurons = [] # # Neurons number of output.
self.Input_Weights = [] # weights from input > hidden.
self.Output_Weights = [] # weights from hidden > output.
self.Prediction = [] # a copy of output, used in mlp.predict.
self.ACTIVATE_FUNCTION = ***'sigmoid'*** # active function to be used.
other options: linear, relu (leaky relu), tanh.

Class MLP functions:

forward (inputs): # Propagating parameters forward.
backwards (label, learning_rate): # Backpropagate the parameters, update the weights at the same time, and calculate the error. (dw1,dw2,z1,z2 are merged inside)
train (data, label, learning_rate, epoch): # start the training, return the average error of each epoch.
predict (input): get the output of this input, do one time forward.
build(self, n_input, n_hidden, n_output, ACTIVATE_FUNCTION='sigmoid', initial_value=1.0, fill_w1=0.2, fill_w2=2.0):: setup the network with given parameters.
In train():
For every epoch:
    Do forward(data);
    Do backwards(label):
    Average(error);
    Return error;

Files inside the package:

mlp.py : the program
Q1_error_sigmoid.json : the print out of errors per epoch in Question 1 (the XOR problem), 10000 epochs in total.
Q1_error_sigmoid.png : the line graph showing the trend of error.
Q1_result.json : the output result of Q1
(all Q1 are using sigmoid active function)
Q3_error_relu.json : the print out of errors per epoch in Question 3
Q3_error_relu.png : the line graph showing the trend of error.
Q3_result_test.json : the output result of Q3 on test set
Q3_result_train.json : the output result of Q3 on test set
Q3_test_result.png : the graph showing the output and label on test set
Q3_train_result.png : the graph showing the output and label on train set
test_ERROR_Q3 : the final error on the test set.
Special1: (for the first run)

- Accuracy.txt: the accuracy of letter recognition.
- Error.txt: the overall error
- relu.json: the error output per epoch
- special_error_relu.png: the line graph showing the trend of error.
- Special_test_result.txt: the output on the test set.
- Special_train_result.txt: the output on the training set.

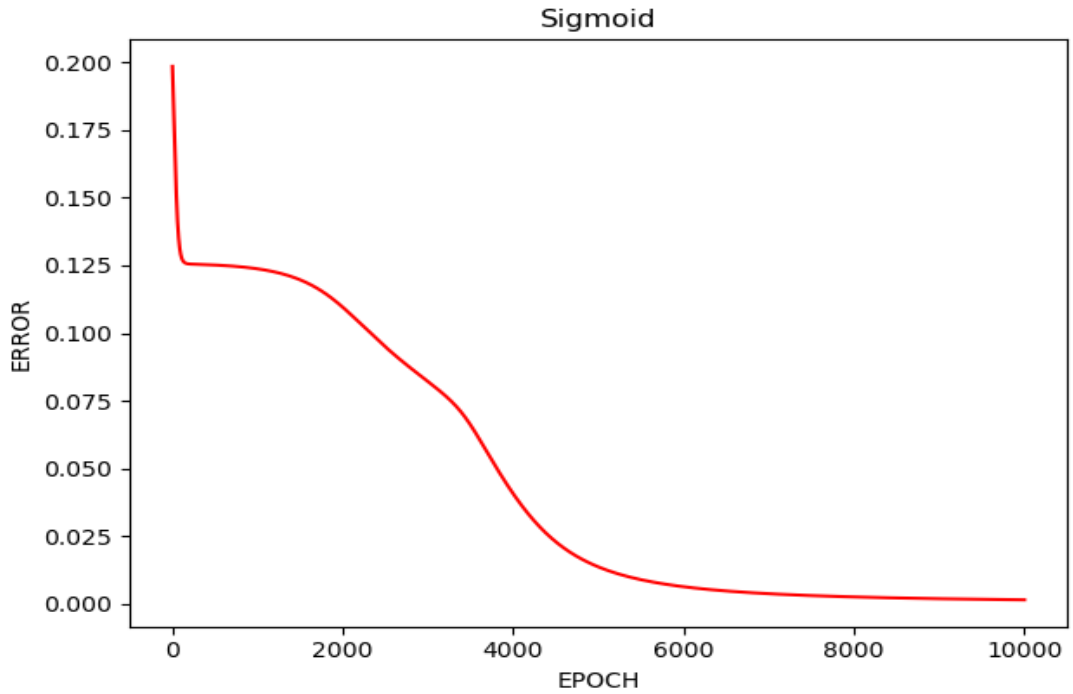
Special2: (for the second run)

- Accuracy.txt: the accuracy of letter recognition.
- Error.txt: the overall error
- sigmoid.json: the error output per epoch
- special_error_sigmoid.png: the line graph showing the trend of error.
- Special_test_result.txt: the output on the test set.
- Special_train_result.txt: the output on the training set.

2. Report of Q1&Q2 (XOR question):

The inputs are [0,0] [1,0] [0,1] [1,1]
The labels are [0] [1] [1] [0]
In this question, I chose the sigmoid function as the activation function, and looped the training 10000 times with learning rate = 0.05, and got the following results:
0.04949087562768775 --- 0
0.9502403444767942 --- 1
0.9502525640572013 --- 1
0.05701572518723778 --- 0

Judging from the results, this training is quite successful. The following figure is the error curve during training.



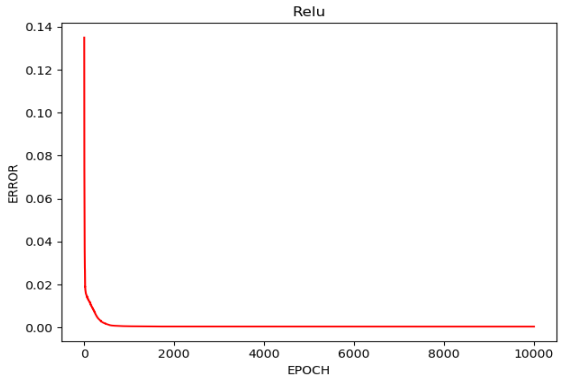
The erroneous decline curve is smooth. After a rapid decline in the early stage (correcting completely random weights) and then tending to decline gently. After more than 6000 cycles, the overall error rate tends to converge smoothly, indicating that the training has been completed.

Problems encountered during training:
Random weight setting problem. In the initialization process, the internal value of all neurons is set to 1, but the weight value is taken as a random value within an interval. After many attempts, in Q1, the initial random range of the weights from the input layer to the hidden layer is -0.2 to 0.2, and the parameters from the hidden layer to the output layer are -2 to 2. Here, if the random value is too large or too small, it will occasionally cause the following three abnormalities in the training results:
First, the output value overflows. The output value becomes larger and larger during the loop and eventually exceeds the allowable range of python, resulting in the termination of the program.
It is worth mentioning that the problem of output value overflow in the special test was very serious at first. To this end, I put two lines of different parameter initialization code in the *mlp.build* method to ensure that the parameters run within a reasonable range.
Second, poor training results. The final output results all approached 0.5, and the error rate did not decrease significantly.
Third, some times the fall of error will encounter a plateau that lasts about 1000 to 3000 epochs. After my analysis, I think this is due to the excessive learning rate. The output fluctuates between both sides of the expected value (greater than and less than) causing this result. We can use a dynamic learning rate, and automatically reduce the learning

rate when error drops below a certain threshold.

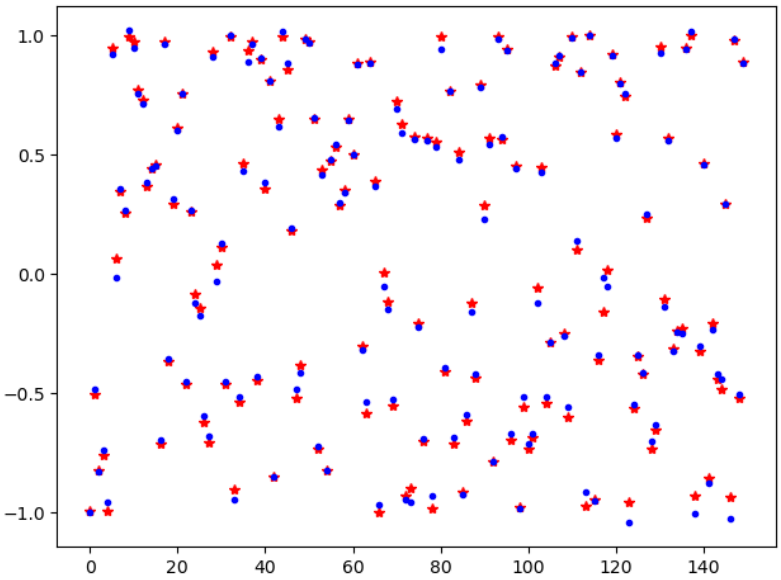
3. Report of Q3&Q4 (sin question):

Function ***data_process_q3()*** is used for generate the data. The result is saved as test_Q3.json and train_Q3.json
The size of train : test is 4:1
The graph below is the error change during the training:(use leaky relu as activation function, 10000 epochs, learning rate = 0.05):

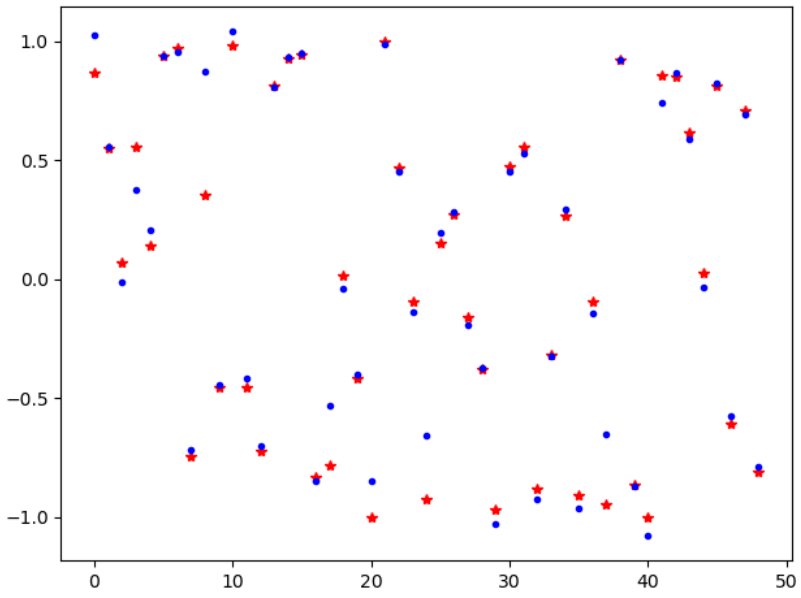


In the case of using leaky relu, the error drops extremely fast and enters the convergence phase within 2000 times.

I output label and output on the training set and test set (in the form of points, where the x-axis is the number of the data, and the y-axis is the corresponding output value of the data. Among them, the **blue points** are the output values, and the **red *** is its labeled value.)
TRAIN:



TEST:



The closer the red and blue points are, the better the prediction. We can see that the proximity on the training set is better than the test set. But the results on the test set are also very good. Except for the large gap between individual points, the prediction results of most points are very close to the labels.

New Idea here: leaky relu

Relu compared with the sigmoid / tanh function, when using the gradient descent method, the convergence speed is faster. Relu only needs a threshold value, that is, the activation value can be obtained, and the calculation speed is faster. The disadvantage is: When the input value of Relu is negative, the output is always 0, and its first derivative is always 0. This will cause the neuron to fail to update the parameters, that is, the neuron does not learn, resulting in "Dead Neuron". Therefore, a negative parameter (return 0.1x) is used here to circumvent this problem.
The error on the test set: 0.3218044757760008.

4. Report of special test (letter question):

data_preprocess_special()* is used for preprocess the *letter-recognition.data
steps:
split the label and data (the first letter and rest 16 numbers)
trans the type of data into int.
trans the letter into a 1*26 sized array.
If the letter is ‘A’, then the array of A’s 1st value is 1.0, the rest are 0.0001.
If the letter is ‘B’, then the array of B’s 2nd value is 1.0, the rest are 0.0001.
If the letter is ‘C’, then the array of C’s 3rd value is 1.0, the rest are 0.0001.
……

If the letter is ‘Z’, then the array of ‘Z”s 26th value is 1.0, the rest are 0.0001.
Trans all alphabets into array by following the rule above.
When we need to transform the output into alphabets:
If the 1st value of this array is the maximum, then it stands for the ‘A’,
If the 2nd value of this array is the maximum, then it stands for the ‘B’,
If the 3rd value of this array is the maximum, then it stands for the ‘C’,
……
If the 26th value of this array is the maximum, then it stands for the ‘Z’,
Function ***get_letter(output)*** do the transform job.

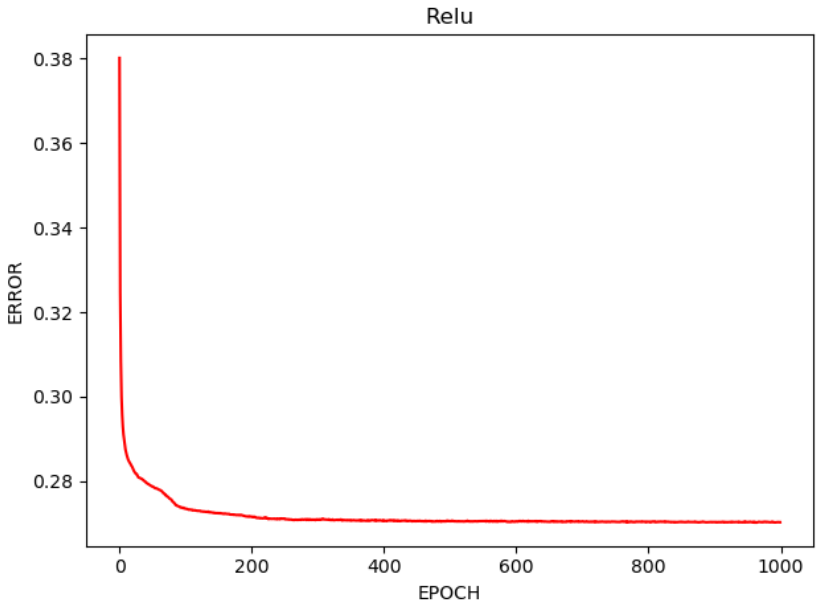
Why use 0.0001 here:

A fixed value is given here to prevent its output from overflowing (if it is 0, it will cause the error correction to be too fast, and the output value in the loop will continue to rise and eventually cause the program to be interrupted). At the same time, I also adjusted the range of weight initialization to ensure that the parameters run in a reasonable range.

First run:

Epoch = 1000, hidden layer = 12,
active_fuction = ‘relu’,
initial neuron value = 0.1,
w1 random range = [-0.1,0.1],
w2 random range = [-0.01,0.01],
learning_rate = 0.05

Result:
The following graph shows the trend of error during the training.



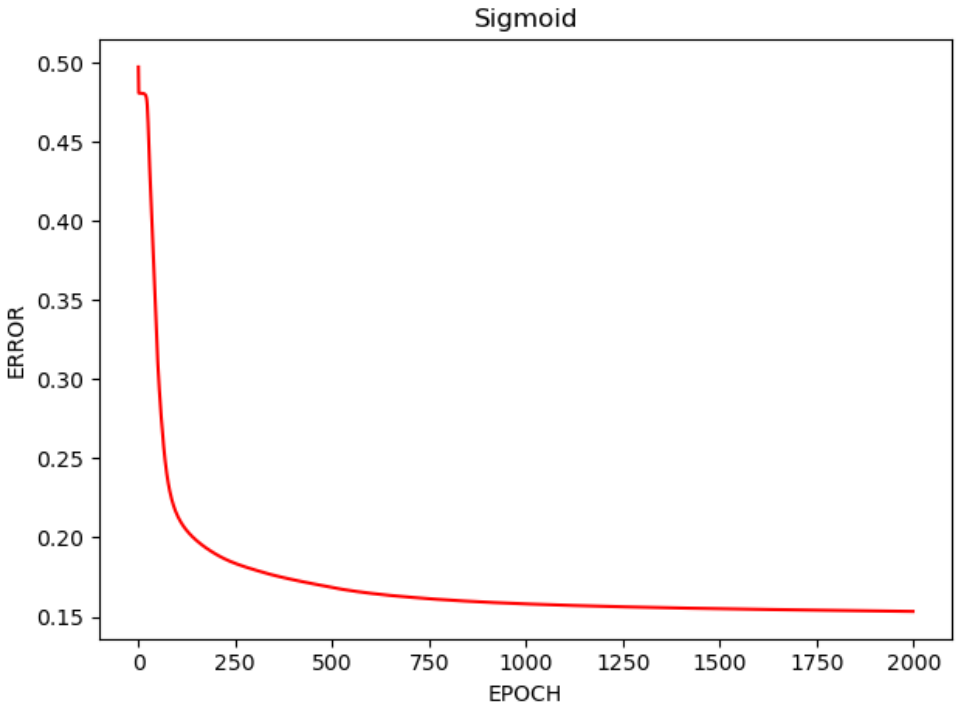
relu quickly reduced the overall error and reached convergence, and finally got relatively good results. In a sense, after 200 epochs, the overall improvement is obvious, it can be said that the next 800 trainings are a bit wasteful.
Overall error on test set:
ERROR on the test set:4117.696172240576
Above is the output of the program. We should divide it by its quantity here.
Average ERROR on the test set: 1.029424043060144
I also statistic the accuracy: (by function statistic_accuracy())
On training set, in 16000 samples, Correct = 11427 Wrong = 4573 Accuracy = 0.7141875%
On testing set, in 3999 samples, Correct = 2756 Wrong = 1243 Accuracy = 0.6891722930732683%

Second run:

Epoch = 2000, hidden layer = 12,
active_fuction = 'sigmoid',
initial neuron value = 0.1,
w1 random range = [-0.1,0.1],
w2 random range = [-0.01,0.01],
learning_rate = 0.025

Result:

The following graph shows the trend of error during the training.



This curve looks to me a very smooth curve. It perfectly reflects the overall changes in the training process. The rate of decline in error rate is high first and then low, which is very reasonable. The output is also very impressive, the overall effect is better than the first run.
Overall error on test set:
ERROR on the test set:3842.0759145253032
Above is the output of the program. We should divide it by its quantity here.
Average ERROR on the test set: 0.9605189786313258
I also statistic the accuracy: (by function statistic_accuracy())
On training set, in 16000 samples, Correct = 12452 Wrong = 3548 Accuracy = 0.77825%
On testing set, in 3999 samples, Correct = 3038 Wrong = 961 Accuracy = 0.7596899224806202%

5. Conclusion:

The first is the logic of the algorithm. The initial idea was to use matrix multiplication in np to achieve forward and backward. But it may be my personal reason. There will always be some strange bugs in the calculation, such as the output value is unchanged, the value overflows, etc. Finally, for the sake of safety, I use loops and double loops to calculate the calculation process of each neuron forward and backward one by one. This also leads to a slower training process. (The second special test ran for nearly 5 hours)

In the beginning, the initial value of the neuron and the random interval of the initial weight cannot be manually set in the build method. This did not cause problems in the first two questions. But in the third question, the input value is -10 ~ + 10 (may be greater) but the output value is between 0-1 (due to my design). The initial weight is between plus and minus 1. This caused the error to be too high at the beginning of the training. The gradient explosion during the training process eventually caused the value of error to overflow the upper limit of Python's calculation and the program was terminated.

After thinking and consulting the data, I determined the problem, and reset the random value range of the initialization parameters according to the input value interval to ensure the normal progress of training. What can be summarized here is that the output, input, hidden, and two sets of parameters between the three layers need to be properly matched (there should be no orders of magnitude difference in the calculation process) otherwise it will cause training failure.

- ***I upload this project onto the GitHub. If an unexpected error occurs during running, please clone from GitHub and try again.***
- ***https://github.com/AndyWang1996/COMP41390-15206086-MLP.git***