BP-Neural-Network-Exploration-Start-From-Scratch

Shen haowen

April 15, 2022

1 Introduction

In this project I used completely python to realize the BP neural network of hand-writing digits identification. Simply by virtue of BP Algorithm and sigmoid activation function I achieved an accuracy of 95.3% , which was already impressive(3 layers, 10 epochs, 0.2 learning rate, 50 hidden neurons). Also, I tried some other modifications like the options in question list to optimize the performance of my model and finally received an accuracy of 97% and cross entropy loss of 0.078. Further explanation is shown below.

2 Steps

2.1 Change the network structure

I changed the hidden units in the hidden layer with [25, 50, 100, 200, 300](50 as default). And I found that the accuracy was[0.951, 0.953, 0.961, 0.961 0.956] respectively. More hidden units indeed increased the accuracy. However, when hidden units exceeded the input units(256 in this case), I found that accuracy fell instead.

hidden units	accuracy
25	0.951
50	0.953
100	0.961
200	0.961
300	0.956

Table 1: Change hidden units in 3 layer network

Also, I tried to add one hidden layer in order to find whether more layers can lead to better accuracy. In this case, the hidden units of each hidden layer matter a lot. I tried the Rule of thumb to compute N_h(hidden units)

$$N_h = \frac{N_s}{\alpha * (N_i + N_o)}$$

where N_i represents the number of neurons in the input layer, N_o represents the number of neurons in the output layer and N_s represents the number of samples. To find the best structure, I fixed lr = 0.3 and tried several (hidden_unit1, hidden_unit2).

hidden units	accuracy
(300,200)	0.954
(200,200)	0.953
(200,100)	0.962
(180,120)	0.963
(150,150)	0.961
(100,100)	0.961

Table 2: Change hidden units in 4 layer network

As we can see in the table above, adding one hidden layer does improve the performance a little bit, but not much. I assume it is because of the network has captured most of the features in the first hidden layer.

2.2 Change step-sizes

Taking the above experiments into consideration, I used 4 layer neural network with epoch = 10 and hidden units = (180, 100). I changed the learning rates of 0.1, 0.2, 0.3, 0.5, 0.7. As we can see, in this case changing lr won't make much difference to accuracy.

learning rate	accuracy
0.1	0.962
0.2	0.961
0.3	0.964
0.5	0.962
0.7	0.963

Table 3: Change learning rate

Also I tried to add momentum in my weight optimization as follows.

$$v_{t+1} = (1 - momentum) * a^t \delta^{t+1} + momentum * v_t$$

$$w_{t+1} = w_t + lr * v_{t+1}$$

```
self.V_dW1 = self.momentum * self.V_dW1 + (1 - self.momentum) * dw1
self.V_db1 = self.momentum * self.V_db1 + (1 - self.momentum) * db1
self.V_dW2 = self.momentum * self.V_dW2 + (1 - self.momentum) * dw2
self.V_db2 = self.momentum * self.V_db2 + (1 - self.momentum) * db2

self.w1 += self.lr * (self.V_dW1 - self.lamda * self.w1) # lamda -> weight decay
self.b1 += self.lr * self.V_db1
self.w2 += self.lr * (self.V_dW2 - self.lamda * self.w2)
self.b2 += self.lr * self.V_db2
```

Figure 1: Add momentum

Momentum technique can help the network converge faster. After adding momentum, the accuracy didn't get much improvement with a highest accuracy of 0.964. I assume the reason is that the amount of train data is relatively small and the speed of fit convergence is already fast.

2.3 Vectorize loss function

As my whole neural network was built on numpy, vectorization is already implemented. Vectorization is significantly faster than loops.

2.4 Add weight decay

I added l_2 weight decay in my neural network as Figure 1, but I found that the accuracy didn't improve. Instead, it decreased when increasing the value of λ.
I also added an early-stopping regulation which worked pretty well. After every epoch, I checked the accuracy of both training data and validation data. I set a patience of 5 round. When the accuracy of validation dataset no longer improves for 5 rounds, I break my training procedure and apply the best weight in my network. Part of the code is shown below.

```
for idx, data in enumerate(valid_data):
    data = (np.asfarray(data) / 255.0) * 0.99 + 0.01
    outputs = n.pre(data)
    label = np.argmax(outputs)  # argmax()函数用于找出数值最大的值所对应的标签
    if (label + 1 == dic["yvalid"][idx]):
        valid_score += 1
    else:
        # network's answer doesn't match correct answer, add 0 to scorecard
        valid_score += 0
    val_accuracy = valid_score/len(dic["yvalid"])
print(f"Training accuracy: {tr_accuracy:.4f}. Validation accuracy: {val_accuracy:.4f}")
if val_accuracy > best_acu:
    best_acu = val_accuracy
    best_w1 = n.weights()[0].copy()
    best_w2 = n.weights()[1].copy()
    best_b1 = n.weights()[2].copy()
    best_b2 = n.weights()[3].copy()
    stopping_rounds = 0
else:
    stopping_rounds += 1
if stopping_rounds >= patience:
    print(f"Validation accuracy has not improved for {patience} rounds. Stopping training.")
    n.weights_optim(best_w1, best_w2, best_b1, best_b2)
    break
```

Figure 2: early-stopping regulation

Also by virtue of large epochs and early-stopping which efficiently decreased overfitting, the accuracy of my network reached a new high of 0.967.

```
Training accuracy: 0.9512. Validation accuracy: 0.9574
Training accuracy: 0.9512. Validation accuracy: 0.9594
Training accuracy: 0.9494. Validation accuracy: 0.9604
Training accuracy: 0.9504. Validation accuracy: 0.9584
Training accuracy: 0.9522. Validation accuracy: 0.9576
Training accuracy: 0.9538. Validation accuracy: 0.9584
Training accuracy: 0.9552. Validation accuracy: 0.9612
Training accuracy: 0.9526. Validation accuracy: 0.9618
Training accuracy: 0.9520. Validation accuracy: 0.9612
Training accuracy: 0.9502. Validation accuracy: 0.9604
Training accuracy: 0.9508. Validation accuracy: 0.9618
Training accuracy: 0.9494. Validation accuracy: 0.9628
Training accuracy: 0.9546. Validation accuracy: 0.9632
Training accuracy: 0.9536. Validation accuracy: 0.9610
Training accuracy: 0.9546. Validation accuracy: 0.9626
Training accuracy: 0.9518. Validation accuracy: 0.9620
Training accuracy: 0.9500. Validation accuracy: 0.9630
Validation accuracy has not improved for 4 rounds. Stopping training.
performance =  0.967

进程已结束,退出代码0
```

Figure 3: accuracy improvement after early-stopping

2.5 Apply Softmax

My neural network implemented sigmoid activation function at beginning. Now I changed the last layer's activation function as softmax and implemented cross entropy loss function instead of MSE.

```python
def sigmoid_func(x):  # 解决了overflow问题后显著提高正确率
    return .5 * (1 + np.tanh(.5 * x))

def softmax(x):
    exps = np.exp(x- np.max(x, axis=0, keepdims=True))
    return exps / np.sum(exps, axis=0, keepdims=True)

def negative_log_likelihood(self, y_true, y_pred):
    return -np.mean(np.log(y_pred[range(len(y_true)), y_true]))

def compute_cost(prediction, label):
    m = label.shape[0]
    cost = -(np.sum(label * np.log(prediction))) / float(m)
    assert (cost.shape == ())

    return cost
```

Figure 4: softmax function

It is worth mentioning that overflow is a big problem when dealing with exp() values, which will severely influence model's performance. So when defining sigmoid and softmax function, I tried to avoid overflow by subtract the maximum on the power index.
As we can see, with learning rate = 0.1, cost decreased constantly as we expected. However, after implementing softmax as the last layer's activation function, the neural network's accuracy decreased a little bit.

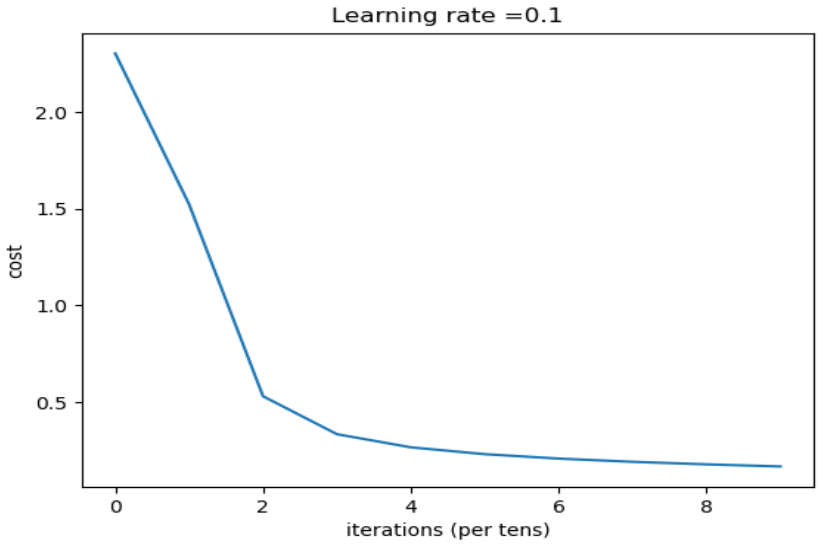

Figure 5: cost decrease

```
Train accuracy: 0.9618. Valid accuracy: 0.9436. Train loss: 0.1258. Valid loss: 0.1243
Train accuracy: 0.9618. Valid accuracy: 0.9408. Train loss: 0.1296. Valid loss: 0.1230
Train accuracy: 0.9644. Valid accuracy: 0.9450. Train loss: 0.1274. Valid loss: 0.1344
Train accuracy: 0.9658. Valid accuracy: 0.9476. Train loss: 0.1329. Valid loss: 0.1271
Train accuracy: 0.9686. Valid accuracy: 0.9510. Train loss: 0.1349. Valid loss: 0.1261
Train accuracy: 0.9712. Valid accuracy: 0.9506. Train loss: 0.1366. Valid loss: 0.1435
Train accuracy: 0.9720. Valid accuracy: 0.9520. Train loss: 0.1358. Valid loss: 0.1372
Train accuracy: 0.9728. Valid accuracy: 0.9480. Train loss: 0.1376. Valid loss: 0.1466
Train accuracy: 0.9744. Valid accuracy: 0.9508. Train loss: 0.1392. Valid loss: 0.1286
Train accuracy: 0.9746. Valid accuracy: 0.9554. Train loss: 0.1388. Valid loss: 0.1444
Train accuracy: 0.9766. Valid accuracy: 0.9526. Train loss: 0.1435. Valid loss: 0.1352
Train accuracy: 0.9772. Valid accuracy: 0.9530. Train loss: 0.1417. Valid loss: 0.1449
Train accuracy: 0.9790. Valid accuracy: 0.9510. Train loss: 0.1446. Valid loss: 0.1541
Train accuracy: 0.9772. Valid accuracy: 0.9524. Train loss: 0.1458. Valid loss: 0.1500
Train accuracy: 0.9790. Valid accuracy: 0.9498. Train loss: 0.1431. Valid loss: 0.1646
Validation accuracy has not improved for 5 rounds. Stopping training.
test accuracy :  0.9560 , test lost : 0.1445

进程已结束,退出代码0
```

Figure 6: accuracy decreased a little

2.6 Add bias for each layer

Actually in every layer of my neural network, I implemented a constant bias in calculation. This helped the neural network to increase stability.

```python
hidden_inputs_1 = np.dot(self.w1, inputs) + self.b1 # bias 1
# calculate the signals emerging from hidden layer
# using sigmoid
hidden_outputs_1 = (sigmoid_func(hidden_inputs_1))  # (100*1)
# add dropout layer
drop_out1 = np.random.rand(*hidden_outputs_1.shape) < p
hidden_outputs_1 *= drop_out1

hidden_inputs_2 = np.dot(self.w2, hidden_outputs_1) + self.b2 # bias 2
hidden_outputs_2 = (sigmoid_func(hidden_inputs_2))
drop_out2 = np.random.rand(*hidden_outputs_2.shape) < p
hidden_outputs_2 *= drop_out2

final_inputs = np.dot(self.w3, hidden_outputs_2) + self.b3 # bias 3
```

Figure 7: bias for each layer

2.7 implement dropout

Implementing dropout help my network decrease the overfitting problem. I also found that with a deeper network(adding hidden layers) and more hidden units, dropout became more effective. I assume the reason is that with dropout hidden units won't get affected by each other.

```
hidden_inputs_1 = np.dot(self.w1, inputs) + self.b1 # dot()函数是指两个矩阵做点乘  (100*1)
# calculate the signals emerging from hidden layer
# using sigmoid
hidden_outputs_1 = (sigmoid_func(hidden_inputs_1))  # (100*1)
# add dropout layer
drop_out1 = np.random.rand(*hidden_outputs_1.shape) < p
hidden_outputs_1 *= drop_out1

hidden_inputs_2 = np.dot(self.w2, hidden_outputs_1) + self.b2
hidden_outputs_2 = (sigmoid_func(hidden_inputs_2))
drop_out2 = np.random.rand(*hidden_outputs_2.shape) < p
hidden_outputs_2 *= drop_out2
```

Figure 8: drop out in training function

Also, in the prediction procedure, we multiplied "p" back to keep the expectation the same.

```
def pre(self, inputs):
    hidden_inputs_1 = np.dot(self.w1, inputs) + self.b1  # dot()函数是指两个矩阵做点乘  (100*1)
    # calculate the signals emerging from hidden layer
    # using sigmoid
    hidden_outputs_1 = (sigmoid_func(hidden_inputs_1)) * p  # (100*1)

    hidden_inputs_2 = np.dot(self.w2, hidden_outputs_1) + self.b2
    hidden_outputs_2 = (sigmoid_func(hidden_inputs_2)) * p

    final_inputs = np.dot(self.w3, hidden_outputs_2) + self.b3  # (10*1)
    # calculate the signals emerging from final output layer
    final_outputs = (sigmoid_func(final_inputs))  # (10*1)

    return final_outputs
```

Figure 9: drop out in prediction function

With the help of dropout, we are happy to see that accuracy has reached a new high of 0.97, which is quite inspiring.

2.8 implement fine-tuning

Fine-tuning is a very practical way of improve the model's performance towards specific type of tasks, especially for nowadays' large pre-trained models. In this case first I tried to train my network to a highest accuracy. Then I saved the trained weights and bias(except the last layer)and fixed them in my fine-tuning. All I need to do now is to feed the model with training data and update the last layer's weight and bias.

```
Training accuracy: 0.9596. Validation accuracy: 0.9580
Training accuracy: 0.9630. Validation accuracy: 0.9610
Training accuracy: 0.9664. Validation accuracy: 0.9608
Training accuracy: 0.9672. Validation accuracy: 0.9612
Training accuracy: 0.9658. Validation accuracy: 0.9618
Training accuracy: 0.9692. Validation accuracy: 0.9624
Training accuracy: 0.9698. Validation accuracy: 0.9632
Training accuracy: 0.9718. Validation accuracy: 0.9638
Training accuracy: 0.9726. Validation accuracy: 0.9648
Training accuracy: 0.9738. Validation accuracy: 0.9660
Training accuracy: 0.9752. Validation accuracy: 0.9658
Training accuracy: 0.9752. Validation accuracy: 0.9658
Training accuracy: 0.9766. Validation accuracy: 0.9658
Training accuracy: 0.9760. Validation accuracy: 0.9666
Training accuracy: 0.9750. Validation accuracy: 0.9666
Training accuracy: 0.9748. Validation accuracy: 0.9670
Training accuracy: 0.9758. Validation accuracy: 0.9662
Training accuracy: 0.9766. Validation accuracy: 0.9658
Training accuracy: 0.9768. Validation accuracy: 0.9666
Training accuracy: 0.9784. Validation accuracy: 0.9668
Validation accuracy has not improved for 3 rounds. Stopping training.
performance =  0.97

进程已结束,退出代码0
```

Figure 10: accuracy after applying dropout

```
    if stopping_rounds >= patience:
        print(f"Validation accuracy has not improved for {patience} rounds. Stopping training.")
        n.weights_optim(best_w1, best_w2, best_w3, best_b1, best_b2, best_b3)
        trained_weight.append(best_w1)
        trained_weight.append(best_w2)
        trained_weight.append(best_b1)
        trained_weight.append(best_b2)
        break

# save the trained weight and bias
output = open('trained_weight.pkl', 'wb')
pickle.dump(trained_weight, output)
output.close()

pkl_file = open('trained_weight.pkl', 'rb')
data = pickle.load(pkl_file)
pkl_file.close()
```

Figure 11: fine-tuning function

Here I gave one sample of training network with and without fine-tuning. As we can see that both of the test accuracy and test lost were better. Test loss reached the lowest value of 0.085 by virtue of fine-tuning.

```
Train accuracy: 0.9598. Valid accuracy: 0.9212. Train loss: 0.1046. Valid loss: 0.0871
Train accuracy: 0.9598. Valid accuracy: 0.9046. Train loss: 0.1059. Valid loss: 0.0952
Train accuracy: 0.9614. Valid accuracy: 0.9268. Train loss: 0.1045. Valid loss: 0.0975
Train accuracy: 0.9638. Valid accuracy: 0.9306. Train loss: 0.1065. Valid loss: 0.0886
Train accuracy: 0.9645. Valid accuracy: 0.9228. Train loss: 0.1066. Valid loss: 0.0866
Train accuracy: 0.9654. Valid accuracy: 0.9340. Train loss: 0.1051. Valid loss: 0.0818
Train accuracy: 0.9659. Valid accuracy: 0.9354. Train loss: 0.1061. Valid loss: 0.0934
Train accuracy: 0.9683. Valid accuracy: 0.9308. Train loss: 0.1083. Valid loss: 0.0944
Train accuracy: 0.9679. Valid accuracy: 0.9370. Train loss: 0.1082. Valid loss: 0.0804
Train accuracy: 0.9686. Valid accuracy: 0.9344. Train loss: 0.1079. Valid loss: 0.0994
Train accuracy: 0.9693. Valid accuracy: 0.9398. Train loss: 0.1071. Valid loss: 0.0984
Train accuracy: 0.9708. Valid accuracy: 0.9420. Train loss: 0.1086. Valid loss: 0.0874
test accuracy :  0.9310 , test lost : 0.0877

进程已结束,退出代码0
```

Figure 12: without fine-tuning

```
Train accuracy: 0.9882. Valid accuracy: 0.9602. Train loss: 0.0831. Valid loss: 0.0928
Train accuracy: 0.9886. Valid accuracy: 0.9610. Train loss: 0.0845. Valid loss: 0.0941
Train accuracy: 0.9882. Valid accuracy: 0.9606. Train loss: 0.0850. Valid loss: 0.0945
Train accuracy: 0.9886. Valid accuracy: 0.9606. Train loss: 0.0859. Valid loss: 0.0943
Train accuracy: 0.9884. Valid accuracy: 0.9616. Train loss: 0.0869. Valid loss: 0.0943
Train accuracy: 0.9886. Valid accuracy: 0.9608. Train loss: 0.0877. Valid loss: 0.0953
Train accuracy: 0.9884. Valid accuracy: 0.9612. Train loss: 0.0877. Valid loss: 0.0962
Train accuracy: 0.9880. Valid accuracy: 0.9600. Train loss: 0.0889. Valid loss: 0.0973
Train accuracy: 0.9884. Valid accuracy: 0.9610. Train loss: 0.0894. Valid loss: 0.0997
Train accuracy: 0.9890. Valid accuracy: 0.9612. Train loss: 0.0909. Valid loss: 0.0978
Train accuracy: 0.9884. Valid accuracy: 0.9608. Train loss: 0.0910. Valid loss: 0.1006
Train accuracy: 0.9890. Valid accuracy: 0.9612. Train loss: 0.0914. Valid loss: 0.0988
Train accuracy: 0.9888. Valid accuracy: 0.9614. Train loss: 0.0922. Valid loss: 0.0988
Train accuracy: 0.9888. Valid accuracy: 0.9612. Train loss: 0.0927. Valid loss: 0.1043
Validation accuracy has not improved for 20 rounds. Stopping training.
test accuracy :  0.9560 , test lost : 0.0849

进程已结束,退出代码0
```

Figure 13: with fine-tuning

2.9 data augmentation

Considered that we are training on hand_writing digits, not other complex images. So i decided to try rotation, translations and resize to change the input images. The data augmentation code is also attached to this project. However, I found that with 16*16 training data, translations and resize seemed not to be a good idea.
In my network, I used rotation to preprocess my training data and create 3000 new training data(after rotation)

```python
tr_data_rotation = []
for idx, data in enumerate(tr_data):
    img = data.reshape(16,16)
    if 0<= idx <= 1000:
        M = cv2.getRotationMatrix2D((8, 8), 45, 1.0)
        rotation_img = cv2.warpAffine(img, M, (16, 16))
        tr_data_rotation.append(rotation_img.reshape(256,))
    if 1000< idx <= 2000:
        M = cv2.getRotationMatrix2D((8, 8), 90, 1.0)
        rotation_img = cv2.warpAffine(img, M, (16, 16))
        tr_data_rotation.append(rotation_img.reshape(256,))
    if 2000< idx <= 3000:
        M = cv2.getRotationMatrix2D((8, 8), 180, 1.0)
        rotation_img = cv2.warpAffine(img, M, (16, 16))
        tr_data_rotation.append(rotation_img.reshape(256,))
    else:
        break
```

Figure 14: data augmentation

I found that test loss was impressively decreased. I achieved a approximately 0.02 decrease in test loss from 0.11 to 0.09. However, with more manually created training data, test accuracy didn't seem to have an impressive improvement. I find that with large amount of rotated training data, the network accuracy seemed to have some decrease. Both training accuracy and test accuracy were affected. I assume the reason is that more rotated data distracted the network's ability to identify the same digit.

```
Train accuracy: 0.8534. Valid accuracy: 0.9082. Train loss: 0.1222. Valid loss: 0.1095
Train accuracy: 0.8589. Valid accuracy: 0.8954. Train loss: 0.1216. Valid loss: 0.1268
Train accuracy: 0.8626. Valid accuracy: 0.9046. Train loss: 0.1222. Valid loss: 0.1118
Train accuracy: 0.8664. Valid accuracy: 0.9118. Train loss: 0.1207. Valid loss: 0.1053
Train accuracy: 0.8687. Valid accuracy: 0.9066. Train loss: 0.1212. Valid loss: 0.1062
Train accuracy: 0.8733. Valid accuracy: 0.9096. Train loss: 0.1200. Valid loss: 0.1008
Train accuracy: 0.8774. Valid accuracy: 0.9176. Train loss: 0.1195. Valid loss: 0.1073
Train accuracy: 0.8789. Valid accuracy: 0.8948. Train loss: 0.1199. Valid loss: 0.1170
Train accuracy: 0.8823. Valid accuracy: 0.9156. Train loss: 0.1195. Valid loss: 0.1031
Train accuracy: 0.8846. Valid accuracy: 0.9292. Train loss: 0.1184. Valid loss: 0.0964
Train accuracy: 0.8889. Valid accuracy: 0.9116. Train loss: 0.1188. Valid loss: 0.1060
Train accuracy: 0.8930. Valid accuracy: 0.9172. Train loss: 0.1182. Valid loss: 0.0975
Train accuracy: 0.8963. Valid accuracy: 0.9232. Train loss: 0.1190. Valid loss: 0.0991
Train accuracy: 0.8967. Valid accuracy: 0.9280. Train loss: 0.1177. Valid loss: 0.0933
Train accuracy: 0.9002. Valid accuracy: 0.9250. Train loss: 0.1187. Valid loss: 0.1004
Train accuracy: 0.9034. Valid accuracy: 0.9206. Train loss: 0.1171. Valid loss: 0.0940
Train accuracy: 0.9057. Valid accuracy: 0.9156. Train loss: 0.1180. Valid loss: 0.0995
Validation accuracy has not improved for 7 rounds. Stopping training.
test accuracy :  0.9230 , test lost : 0.0969

进程已结束,退出代码0
```

Figure 15: add 3000 rotated images as training data

I tried to use this exact trained weights and bias to do fine-tuning. It can be seen from below that after fine-tuning, both test accuracy and test loss were having better performance. This case showed the power of fine-tuning again.

```
Train accuracy: 0.9045. Valid accuracy: 0.9408. Train loss: 0.0981. Valid loss: 0.0926
Train accuracy: 0.9057. Valid accuracy: 0.9408. Train loss: 0.0995. Valid loss: 0.0918
Train accuracy: 0.9052. Valid accuracy: 0.9404. Train loss: 0.0999. Valid loss: 0.0923
Train accuracy: 0.9053. Valid accuracy: 0.9424. Train loss: 0.1009. Valid loss: 0.0952
Train accuracy: 0.9060. Valid accuracy: 0.9410. Train loss: 0.1020. Valid loss: 0.0998
Train accuracy: 0.9048. Valid accuracy: 0.9428. Train loss: 0.1019. Valid loss: 0.0951
Train accuracy: 0.9057. Valid accuracy: 0.9440. Train loss: 0.1024. Valid loss: 0.0962
Train accuracy: 0.9065. Valid accuracy: 0.9428. Train loss: 0.1029. Valid loss: 0.0959
Train accuracy: 0.9058. Valid accuracy: 0.9430. Train loss: 0.1040. Valid loss: 0.1020
Train accuracy: 0.9055. Valid accuracy: 0.9414. Train loss: 0.1046. Valid loss: 0.0969
Train accuracy: 0.9048. Valid accuracy: 0.9404. Train loss: 0.1054. Valid loss: 0.1019
Train accuracy: 0.9057. Valid accuracy: 0.9410. Train loss: 0.1050. Valid loss: 0.0987
Train accuracy: 0.9065. Valid accuracy: 0.9424. Train loss: 0.1055. Valid loss: 0.1022
Train accuracy: 0.9055. Valid accuracy: 0.9432. Train loss: 0.1058. Valid loss: 0.1017
Train accuracy: 0.9062. Valid accuracy: 0.9436. Train loss: 0.1070. Valid loss: 0.0979
Validation accuracy has not improved for 8 rounds. Stopping training.
test accuracy :  0.9430 , test lost : 0.0958

进程已结束,退出代码0
```

Figure 16: add 3000 rotated images after fine-tuning

2.10 Add 2d conv layer

At last I replaced the first layer with 2d convolutional layer. I reshaped the image size with 16*16 and used a 5*5 kernel. Here is part of my code extracted from convolutional layer.

```python
def forward(self, X):
    n_channels, input_height, input_width = X.shape
    # 计算输出张量的大小
    output_height = int((input_height + 2 * self.padding - self.kernel_size) / self.stride + 1)
    output_width = int((input_width + 2 * self.padding - self.kernel_size) / self.stride + 1)
    # 在输入张量的周围填充零
    X_pad = np.pad(X, ((0, 0), (self.padding, self.padding), (self.padding, self.padding)), 'constant')
    # 初始化输出张量
    out = np.zeros((self.out_channels, output_height, output_width))

    # 执行卷积运算
    for c_out in range(self.out_channels):
        for h_out in range(output_height):
            for w_out in range(output_width):
                h_start = h_out * self.stride
                h_end = h_start + self.kernel_size
                w_start = w_out * self.stride
                w_end = w_start + self.kernel_size

                X_slice = X_pad[:, h_start:h_end, w_start:w_end]
                out[c_out, h_out, w_out] = np.sum(X_slice * self.W[c_out]) + self.b[c_out]
```

Figure 17: convolutional forward function

```python
def backward(self, X, dout):
    in_channels, input_height, input_width = X.shape
    _, output_height, output_width = dout.shape
    # 在输入张量的周围填充零
    X_pad = np.pad(X, ((0, 0), (self.padding, self.padding), (self.padding, self.padding)), 'constant')
    # 初始化梯度张量
    dX = np.zeros_like(X)
    dW = np.zeros_like(self.W)
    db = np.zeros_like(self.b)
    # 计算梯度
    for c_out in range(self.out_channels):
        for h_out in range(output_height):
            for w_out in range(output_width):
                h_start = h_out * self.stride
                h_end = h_start + self.kernel_size
                w_start = w_out * self.stride
                w_end = w_start + self.kernel_size
                X_slice = X_pad[:, h_start:h_end, w_start:w_end]
                dX[:, h_start:h_end, w_start:w_end] += self.W[c_out] * dout[c_out, h_out, w_out]
                dW += X_slice * dout[c_out, h_out, w_out]
                db[c_out] += dout[c_out, h_out, w_out]
                self.W -= self.lr * dW
                self.b -= self.lr * db
```

Figure 18: convolutional backward function

As we can see, after applying the conv2d layer, test loss has achieved a new lowest value of 0.078. However, test accuracy dropped around 0.95.

```
Train accuracy: 0.9346. Valid accuracy: 0.9288. Train loss: 0.0828. Valid loss: 0.0862
Train accuracy: 0.9426. Valid accuracy: 0.9336. Train loss: 0.0835. Valid loss: 0.0787
Train accuracy: 0.9474. Valid accuracy: 0.9378. Train loss: 0.0843. Valid loss: 0.0868
Train accuracy: 0.9526. Valid accuracy: 0.9408. Train loss: 0.0855. Valid loss: 0.0937
Train accuracy: 0.9556. Valid accuracy: 0.9402. Train loss: 0.0869. Valid loss: 0.0881
Train accuracy: 0.9602. Valid accuracy: 0.9488. Train loss: 0.0862. Valid loss: 0.1036
Train accuracy: 0.9638. Valid accuracy: 0.9460. Train loss: 0.0878. Valid loss: 0.0971
Train accuracy: 0.9660. Valid accuracy: 0.9478. Train loss: 0.0895. Valid loss: 0.0962
Train accuracy: 0.9678. Valid accuracy: 0.9488. Train loss: 0.0901. Valid loss: 0.0956
Train accuracy: 0.9710. Valid accuracy: 0.9516. Train loss: 0.0913. Valid loss: 0.1040
Train accuracy: 0.9730. Valid accuracy: 0.9526. Train loss: 0.0921. Valid loss: 0.0960
Train accuracy: 0.9736. Valid accuracy: 0.9530. Train loss: 0.0922. Valid loss: 0.0961
Validation accuracy has not improved for 10 rounds. Stopping training.
test accuracy =  0.936 , test lost =  0.07820809114571745

进程已结束,退出代码0
```

Figure 19: test loss decreased

3 Conclusion

With the implementation of above optimization and modification, the final neural network of mine consists of 4 layers, which begins with a conv2d layer, and two hidden layers with sigmoid activation function, and a final softmax layer. I was able to get the best performance of 0.97 accuracy and 0.078 cross entropy loss. Building a neural network with completely python and without the modern functions such as pytorch was not easy for me. However, I have also gained deeper understanding of neural network and these methods of optimization. It is pain and pleasure.