Report on the Individual Report

Mutian Wang (mw3386)

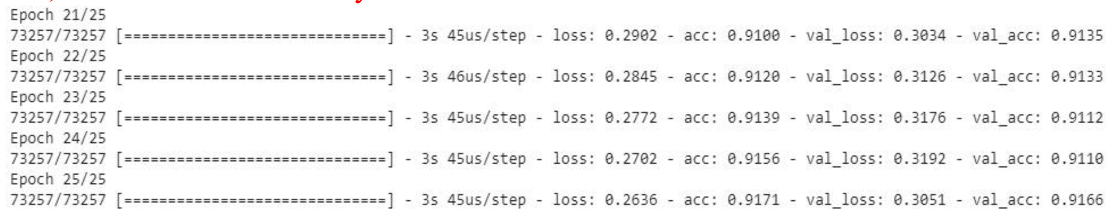
I did the whole project at Google Colab and downloaded the .py file (the path is changed to relative path). The code can be run at Google Colab without any problem. Link of my Google Colab script is as follows:

https://drive.google.com/open?id=1Y2PqxC016ijGEvyFTxRD_2PugL9--R8z

When working on this project, I used the Keras team's MNIST code as a reference, as suggested by the instructor.

Source: https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py

Below is a snippet of the result of my final code. The test accuracy can reach 91.66%, while the train accuracy is 91.71%.



```
Epoch 21/25
73257/73257 [==============================] - 3s 45us/step - loss: 0.2902 - acc: 0.9100 - val_loss: 0.3034 - val_acc: 0.9135
Epoch 22/25
73257/73257 [==============================] - 3s 46us/step - loss: 0.2845 - acc: 0.9120 - val_loss: 0.3126 - val_acc: 0.9133
Epoch 23/25
73257/73257 [==============================] - 3s 45us/step - loss: 0.2772 - acc: 0.9139 - val_loss: 0.3176 - val_acc: 0.9112
Epoch 24/25
73257/73257 [==============================] - 3s 45us/step - loss: 0.2702 - acc: 0.9156 - val_loss: 0.3192 - val_acc: 0.9110
Epoch 25/25
73257/73257 [==============================] - 3s 45us/step - loss: 0.2636 - acc: 0.9171 - val_loss: 0.3051 - val_acc: 0.9166
```

I. How CNN works?

CNN stands for Convolutional Neural Network. In deep learning, CNN is most frequently applied on the image data. It has some basic components: input layer, convolutional layer, pooling layer, fully connected layer, and output layer.

A typical CNN starts with an input layer and ends with an output layer. The layers in the middle are called hidden layers. In hidden layers, a pooling layer usually comes after one or more convolutional layers. The last layer of hidden layers is a fully connected layer.

Input Layer

The input layer is just the image data. If the data are in the format of RGB, then the input is an array with the dimension (#images, width, height, 3). If the data are in the format of grayscale, then the input is an array with the dimension (#images, width, height, 1).

Convolutional Layer

The convolutional layer is the most important component in CNN. It can extract features from the images. We use a kernel or filter to do the convolution operation. In one convolutional layer, there can be multiple kernels, and each kernel has the same size and stride.

One problem of convolution is that it cannot deal with the values on the margin. Hence, padding can be used. There are several types of padding, and the most popular one is zero padding.

After the convolution operation, an activation function will be applied. The most common activation functions include ReLu, Sigmoid and tanh. Currently, ReLu is the most popular one. An activation function is very important in that it can make the model non-linear, while convolution is just a linear transformation.

Pooling Layer

Usually the dimension of the data will get larger after the convolution layer, greatly increasing the computational cost - just think about how many parameters need to be trained! Fortunately, pooling can progressively reduce the size of the data.

There are multiple types of pooling, and the most common one is max pooling. We can set the size and stride of max pooling.

Fully Connected Layer

This layer is the same as the one in a regular neural network. It appears at the end of CNN. It also contains an activation function.

If the input of this layer is not a one-dimensional array, then we need to flatten the data in advance. After this layer, the data is still a one-dimensional array.

Output Layer

The number of neurons in this layer equals to the number of classes. A function called softmax is used for prediction.

II. Hypterparameters & Parameters of CNN

Hyperparameters

Hyperparameters are specified by people, and we need to tune them manually.

(1) Hyperparameters regarding to the hidden layers: number of convolutional layers, number of pooling layers, the order of hidden layers, kernel size, stride, padding, etc.

Function: These hyperparameters decide the general structure of CNN. A shallow CNN might cause underfitting, while a deep CNN will increase computational cost and cause ovefitting if the data set is not large enough.

(2) Number of epochs

Number of epochs is the number of times the training data is shown to CNN.

Function: It can increase the accuracy, because the model might not converge within a single epoch. Thus the model should run multiple epochs until the accuracy does not increase.

(3) Batch size

Batch size is the number of data given to CNN each time.

Function: Since the data set is usually very large, we cannot put all the data to CNN at one time, due to the limited memory. Setting an appropriate batch size can solve this problem.

(4) Dropout

Dropout means some neurons of a hidden layer are dropped out randomly in each iteration. We need to choose the dropout rate.

Function: It can avoid overfitting. Dropout rate is usually between 0.2 and 0.5. If it's too low, then it has little effects; if it's too high, then it might cause underfitting.

(5) Learning rate

Learning rate is a parameter in gradient descent, an optimization method.

Function: Learning rate is necessary in optimization like gradient descent. A small learning rate will make the model converge slowly, while a large learning rate will make the model fail to converge.

(6) Activation function

This concept has already been explained in the previous section.

Function: This is a non-linear transformation on the image data.

Parameters

Parameters are not specified by people. In fact, they are optimized by the model and they are estimated from the data set. For example, every value of a kernel or filter is a parameter.

Function: The model uses parameters to make predictions.

III. Code Logic

Firstly, the code reads in the data, and converts the RGB images to grayscale images. The data is also reshaped to have the right dimension. Then it corrects the label by changing 10 to 0. Next, CNN is built, trained and validated. Finally the accuracy on the test data is printed.

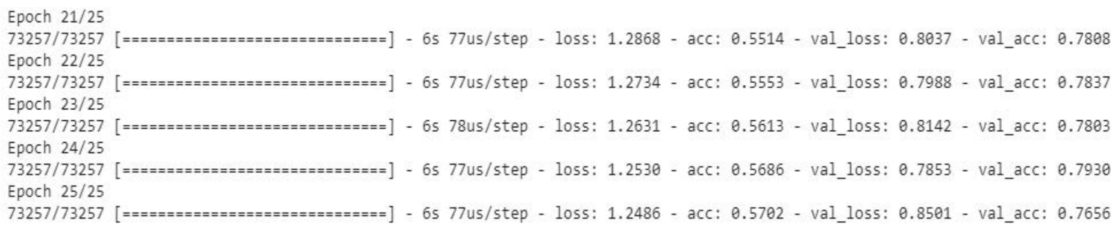
IV. Starting Model

Initially I used the classical LeNet-5 model that works well on the MNIST data set. I chose this model because it's classical and SVHN format 2 is quite similar to MNIST. The model is as follows:

Conv: 6 filters with the size 5*5 and ReLu function
Pooling: size and stride are 2, dropout rate is 0.3
Conv: 16 filters with the size 5*5 and ReLu function
Pooling: size and stride are 2, dropout rate is 0.3
Conv: 120 filters with the size 5*5 and ReLu function
FC: 84 units, dropout rate is 0.3

Epoch: 25
Learning rate: 0.005
Optimizer: Adam

The result is as follows:



```
Epoch 21/25
73257/73257 [==============================] - 6s 77us/step - loss: 1.2868 - acc: 0.5514 - val_loss: 0.8037 - val_acc: 0.7808
Epoch 22/25
73257/73257 [==============================] - 6s 77us/step - loss: 1.2734 - acc: 0.5553 - val_loss: 0.7988 - val_acc: 0.7837
Epoch 23/25
73257/73257 [==============================] - 6s 78us/step - loss: 1.2631 - acc: 0.5613 - val_loss: 0.8142 - val_acc: 0.7803
Epoch 24/25
73257/73257 [==============================] - 6s 77us/step - loss: 1.2530 - acc: 0.5686 - val_loss: 0.7853 - val_acc: 0.7930
Epoch 25/25
73257/73257 [==============================] - 6s 77us/step - loss: 1.2486 - acc: 0.5702 - val_loss: 0.8501 - val_acc: 0.7656
```

The best accuracy is 79.3% achieved in the 24th epoch. It's not good enough.

V. Intermediate Models

I tried to change the learning rate, and I found 0.005 or 0.001 the best. A larger learning rate, like 0.01, will make the model fail to converge; a smaller learning rate will slow down the training.

I also changed the optimizer, such as Adadelta, SGD and RMSprop, but they did not make a huge difference. Thus I decided to stick to Adam.

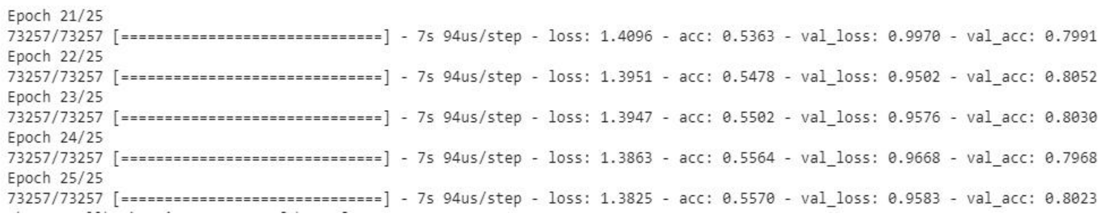
I left the kernel size unchanged because I think it's great: 32*32 will become 28*28 after the first convolution; it becomes 14*14 after a pooling layer; it becomes 10*10 after the second convolution; it becomes 5*5 after a second pooling; it becomes a 1*1 after the third convolution! It's a one dimensional array after the execution of all the hidden layers.

I mainly focused on the number of filters in every convolutional layer. Below is one of the model I used:

Conv: 16 filters with the size 5*5 and ReLu function
Pooling: size and stride are 2, dropout rate is 0.3
Conv: 32 filters with the size 5*5 and ReLu function
Pooling: size and stride are 2, dropout rate is 0.3
Conv: 64 filters with the size 5*5 and ReLu function
FC: 128 units, dropout rate is 0.3

Epoch: 25
Learning rate: 0.005
Optimizer: Adam

The result is as follows:



```
Epoch 21/25
73257/73257 [==============================] - 7s 94us/step - loss: 1.4096 - acc: 0.5363 - val_loss: 0.9970 - val_acc: 0.7991
Epoch 22/25
73257/73257 [==============================] - 7s 94us/step - loss: 1.3951 - acc: 0.5478 - val_loss: 0.9502 - val_acc: 0.8052
Epoch 23/25
73257/73257 [==============================] - 7s 94us/step - loss: 1.3947 - acc: 0.5502 - val_loss: 0.9576 - val_acc: 0.8030
Epoch 24/25
73257/73257 [==============================] - 7s 94us/step - loss: 1.3863 - acc: 0.5564 - val_loss: 0.9668 - val_acc: 0.7968
Epoch 25/25
73257/73257 [==============================] - 7s 94us/step - loss: 1.3825 - acc: 0.5570 - val_loss: 0.9583 - val_acc: 0.8023
```

The accuracy is around 80%. It’s still not good enough.

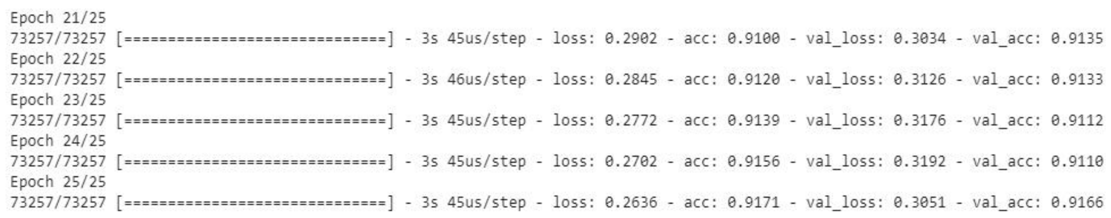
VI. Final Model

After many experiments, I decided my final model as follows:

Conv: 16 filters with the size 5*5 and ReLu function
Pooling: size and stride are 2, dropout rate is 0.3
Conv: 32 filters with the size 5*5 and ReLu function
Pooling: size and stride are 2, dropout rate is 0.3
Conv: 64 filters with the size 5*5 and ReLu function
FC: 512 units, dropout rate is 0.3

Epoch: 25
Learning rate: 0.001
Optimizer: Adam

The result is as follows:



```
Epoch 21/25
73257/73257 [==============================] - 3s 45us/step - loss: 0.2902 - acc: 0.9100 - val_loss: 0.3034 - val_acc: 0.9135
Epoch 22/25
73257/73257 [==============================] - 3s 46us/step - loss: 0.2845 - acc: 0.9120 - val_loss: 0.3126 - val_acc: 0.9133
Epoch 23/25
73257/73257 [==============================] - 3s 45us/step - loss: 0.2772 - acc: 0.9139 - val_loss: 0.3176 - val_acc: 0.9112
Epoch 24/25
73257/73257 [==============================] - 3s 45us/step - loss: 0.2702 - acc: 0.9156 - val_loss: 0.3192 - val_acc: 0.9110
Epoch 25/25
73257/73257 [==============================] - 3s 45us/step - loss: 0.2636 - acc: 0.9171 - val_loss: 0.3051 - val_acc: 0.9166
```

The test accuracy can reach 91.66% in the 25th epoch, and the train accuracy is 91.71%. It's quite a good result.

VII. Difficulties

At first, I used the learning rate 0.01, and my model could not converge however I tuned the hyperparameters. Then I changed the learning rate to 0.005 or 0.001, and the problem was solved.

The major difficulty is that the model ran quite slowly, so tuning hyperparameters was not convenient. Then I used Google Colab GPU, and the problem was solved.

Generally speaking, I did not encounter too many difficulties when implementing CNN.

VIII. Future Works

I have tried plenty of models with different hyperparameters, so maybe tuning hyperparameters is a dead end. I think more focus should be placed on the preprocessing part and the data itself.

Some of the original images contain multiple digits, but the label is just 0~9 (if 10 is transformed to 0). The redundant digit(s) might have a negative impact on the model.

Some images are very blurred, and even human beings cannot correctly recognize them, let alone the CNN model. Including these images in the training set or test set can decrease the accuracy. Thus we can do some quality check/control.