Keras MNIST Digit Classification

Team members:
Jufeng Yang ID: 20125011
Xingda Zhou ID: 19107471
Zhongen Qin ID: 19107579

Table of Content

1. Introduction

MNIST handwritten number classification, a precursor project to deep learning, is a necessary practice for AI learners to get started. It is well known that one of the most efficient languages for writing deep learning or machine learning is Python. Python integrates a large number of libraries of deep learning algorithms that can be used directly without having to write the associated algorithms each time, which greatly improves the simplicity and readability of deep learning programs. Based on Python's tensorflow library, importing the keras library from tensorflow will allow programmer to write code directly to achieve a 99% accurate MNIST classification target in the validation dataset and test datasets. The platform chosen for programming the python code in this project will be Jupyter. Jupyter can run Python code in segments. This approach will therefore make it easier for programmers to debug their code.

```
Epoch 10/10
108/108 [==============================] - 31s 289ms/step - loss: 0.0239 - Accuracy: 0.9925 - val_loss: 0.0242 - val_Accuracy: 0.9933

10000/10000 - 12s - loss: 0.0214 - Accuracy: 0.9932
The loss is:  0.021367348730564117
The accuracy is : 0.9932000041007996
```

Figure.1: The validation accuracy and loss (Upper), test accuracy and loss (Down)

2. Data pre-processing

2.1.Import relative code libraries

```
#Import the code libaries.
import tensorflow as tf
from tensorflow.keras import layers, datasets, Sequential, optimizers, utils
import numpy as np
import matplotlib.pyplot as plt
```

Figure.2: The code of importing libraries.

As seen in Figure 2, the four libraries tensorflow, keras, numpy, and matplotlib are imported. These libraries will be used to import MNIST datasets, pre-process the datasets, build convolutional layers, and define optimizer algorithms. numpy will be used to perform the associated matrix calculations. matplotlib is used to draw graphs.

Run this code.

2.2.Load MNIST hand-write digit datasets

```
#Data preprocess.
#Load MNIST data into Variables
(x_train, y_train), (x_test, y_test) = datasets.mnist.load_data("MNIST_data")
for i in range(10):
    print('The %d train set label:'%(i),y_train[i])
    print('The %d test set label: '%(i),y_test[i])
```

Figure.3: Load the datasets and print the first ten sets.

The load_data statement uses the keras library of datasets and calls the method load_data to load MNIST datasets from the path MNIST_data. If there are no datasets in the path, the MNIST datasets will be downloaded automatically.

Check that the dataset is loaded successfully and use a loop to print out the first ten labels of the test and training sets. The results are shown in Figure 4.

```
The 0 train set label: 5
The 0 test set label:  7
The 1 train set label: 0
The 1 test set label:  2
The 2 train set label: 4
The 2 test set label:  1
The 3 train set label: 1
The 3 test set label:  0
The 4 train set label: 9
The 4 test set label:  4
The 5 train set label: 2
The 5 test set label:  1
The 6 train set label: 1
The 6 test set label:  4
The 7 train set label: 3
The 7 test set label:  9
The 8 train set label: 1
The 8 test set label:  5
The 9 train set label: 4
The 9 test set label:  9
```

Figure.4: The first 10 train and test labels.

The data has been exported successfully, the data has been loaded successfully, and the data will then need to be processed.

2.3.Reshape and Normalize the photo data

```
#Reshape the data into [-1,28,28,1] form, and Normalize the data
x_train = x_train.reshape(-1,28,28,1).astype('float32')/255
x_test = x_test.reshape(-1,28,28,1).astype('float32')/255
```

Figure.5: Reshape and normalize the digit data.

The reshaped code has [-1,28,28,1], where -1 allows the system to calculate the total number of images, in this case 60,000 training images and 10,000 test images. 28*28 is the size of the image (in pixels), one image contains 28*28 pieces of data. The last 1 is the chanel of the image, the handwritten numbers are grayscale images and therefore have only one dimension. The code used to check the shape of the datasets as shown in Figuire.6(Upper). The output of shape is shown in Figure.6(Down)

```
print("Train figures sets shape:", x_train.shape)
print("Test figures sets shape:", x_test.shape)
print("Train labels sets shape:", y_train.shape)
print("Test labels sets shape:", y_test.shape)
```

```
Train figures sets shape: (60000, 28, 28, 1)
Test figures sets shape: (10000, 28, 28, 1)
Train labels sets shape: (60000, 10)
Test labels sets shape: (10000, 10)
```

Figure.6: Code for printing shape of the data(Upper), Output(Down).

The astype after the code converts all 784 data into floating point data, dividing each data by 255, so that the data only exists in the interval 0-1. This helps to reduce overfitting.

2.4.Convert label into a vector.

```
#To convert the label data into a matricx.
#This step can transfer 0 to [1,0,0,0,0,0,0,0,0,0]. like this format.
y_train = utils.to_categorical(y_train)
y_test = utils.to_categorical(y_test)
for i in range(10):
    print('The %d train set label:'%(i),y_train[i])
    print('The %d test set label: '%(i),y_test[i])
```

Figure.7: Convert the label into a matrix vector.

The code in Figure 7 will transform the label into a matrix vector. For example, 1 -> [0,1,0,0,0,0,0,0,0,0]. Print out the first 10 training data labels and the test data labels.

The conversion result is shown in the Figure.8.

```
The 0 train set label: [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
The 0 test set label:  [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
The 1 train set label: [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
The 1 test set label:  [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
The 2 train set label: [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
The 2 test set label:  [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
The 3 train set label: [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
The 3 test set label:  [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
The 4 train set label: [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
The 4 test set label:  [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
The 5 train set label: [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
The 5 test set label:  [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
The 6 train set label: [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
The 6 test set label:  [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
The 7 train set label: [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
The 7 test set label:  [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
The 8 train set label: [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
The 8 test set label:  [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
The 9 train set label: [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
The 9 test set label:  [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
```

Figure.8: Output after conversion.

The label contains a total of 10 categories (0-9). By artificially converting the matrix vector output to decimal numbers we can obtain:

Train: 5, 0, 4, 1, 9, 2, 1, 3, 1, 4

Test: 7, 2, 1, 0, 4, 1, 4, 9, 5, 9

More obviously this gives the same result as before the transformation. Therefore the transformation process is successful.

3. CNN model

3.1.Create a CNN model

The CNN model is a model that is very good at processing images. CNNs have an efficient way of handling parameters (shared weights), which greatly reduces the number of parameters. The CNN model consists of a convolutional layer, a pooling layer, a smoothing layer, and a fully connected layer. In general, there can be multiple convolutional and pooling layers, and one fully-connected layer, and only one flattening layer. For simple classification, two fully-connected layers are sufficient. In this project code, the design of the convolutional model is shown in Figure.9.

```
#Creat a CNN layers
Conv_layers = [
    #First layer
    layers.Conv2D(filters=32, kernel_size = [3,3], padding = 'same', input_shape = [28,28,1], activation = tf.nn.relu),
    layers.MaxPool2D(pool_size=[2,2]),

    #Second layer
    layers.Conv2D(filters=64, kernel_size = [3,3], padding = 'same', input_shape = [28,28,1], activation = tf.nn.relu),
    layers.MaxPool2D(pool_size=[2,2]),

    #Third layer
    layers.Conv2D(filters=128, kernel_size = [3,3], padding = 'same', input_shape = [28,28,1], activation = tf.nn.relu),
    layers.MaxPool2D(pool_size=[2,2]),

    #fLatten the all parameters
    layers.Flatten(),

    #Full connection layer with dropout 0.25.
    layers.Dense(128, activation = tf.nn.relu),
    layers.Dropout(0.25),
    layers.Dense(10, activation = tf.nn.softmax)
]
```

Figure.9: The CNN layers design.

There are three convolutional layers, three pooling layers, one flattening layer, two fully-connected layers, and one dropout. The first convolutional layer uses 32 convolutional kernels, each of size 3*3, a padding type of same, the shape of the input data as before, and an activation function of "relu". The 32 convolution kernels give 32 results showing different features, and the padding ensures that the output image is the same size as the input. The second convolutional layer only increases the number of convolutional kernels compared to the first convolutional layer. The third convolutional layer also increases the number of convolutional kernels.

The pooling matrices of the three pooling layers are all of size 2*2. After processing the three convolutional layers and the pooling layer, a flattening layer is added. The flattening layer puts all the parameters in one dimension, so that it can be connected directly to the fully-connected layer, which passes all the parameters through the fully-connected layer (the number of neurons in the fully-connected layer can be arbitrarily designed, the dense method automatically calculates and sets the matrix parameters of the neurons), followed by a Dropout layer, which randomly "rests" some of the neurons. This can effectively reduce overfitting. Finally, a fully connected layer (output layer) with an output of 10 is connected.

Put all the CNN layers into the Sequential list of keras, and the CNN layers will be run sequentially. Output the CNN model as a text report. The code shows in the Figure.10(Upper), the text report shows in Figure.10(Down).

```
#Put those layer in to the Sequential list.
model = Sequential(Conv_layers)

#Output a report of the CNN layer.
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 28, 28, 32)        320
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 14, 14, 64)        18496
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 7, 7, 128)         73856
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 3, 3, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1152)              0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               147584
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1290
=================================================================
Total params: 241,546
Trainable params: 241,546
Non-trainable params: 0
_________________________________________________________________
```

Figure.10: Code for Sequential (Upper), Text report (Down).

The data in the report shows that the more convolutional layers there are, the more parameters there are, and the more convolutional cores there are, the more parameters there are. In general, 128 convolutional kernels is already a lot, and each doubling of the convolutional kernels results in a large increase in parameters. Therefore, it is not possible to use too many convolutional layers and kernels. If you are not satisfied with the test results, you can add more convolutional kernels and try to adjust the other parameters (eg. optimizer function, learning rate).

3.2.Model compile and train

After the convolutional model has been designed, the next step is to compile it, passing some parameters into the compiled method. The relevant code is shown in

Figure 11.

```
#Training model

#The optimizer is Adam
#The loss function is categorical_crossentrop
#The matricx information is accuracy
model.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = 1e-3),loss = 'categorical_crossentropy',metrics = ['Accuracy'])

#Traning setting, define the epochs, batch_size. Save the hissory to a variable
History = model.fit(x_train,y_train, epochs = 10, batch_size = 500,validation_split = 0.1)
```

Figure.11: Compile and Train code.

As shown in the figure, the optimizer algorithm is Adam, the loss function is categorical_crossentropy, and the matrix parameter is Accuracy. Adam is a parameter adaptive algorithm that combines Adagrid and momentum to form a very good algorithm. categorical_crossentropy is a loss function that works well in general. The matrix parameter is Accuracy, which is recommended by the tensorflow guidelines.

The next step is to train the model, passing the images and labels from the training dataset into the training process. The training epoches are set to 10, each Batch_size is set to 500, and the ratio of the training set is set to 0.1. So, in this case, a total of 60,000*0.9 = 54,000 data are used for training and 6,000 data are used for validation. In a batch with 500 images per input, there will be 108 times inputs in a batch. The same process can be run 10 times. All training history data is stored in the History variable, which is used to plot the accuracy and loss graphs.

3.3.Training results and plots

All the parameters of the CNN model were designed, compiled and passed through the training process. The next step will be to execute the code. The training results are shown in Figure.12.

```
Epoch 1/10
108/108 [==============================] - 33s 306ms/step - loss: 0.5250 - Accuracy: 0.8378 - val_loss: 0.0867 - val_Accuracy: 0.9740
Epoch 2/10
108/108 [==============================] - 35s 321ms/step - loss: 0.1054 - Accuracy: 0.9675 - val_loss: 0.0528 - val_Accuracy: 0.9850
Epoch 3/10
108/108 [==============================] - 34s 312ms/step - loss: 0.0722 - Accuracy: 0.9780 - val_loss: 0.0426 - val_Accuracy: 0.9882
Epoch 4/10
108/108 [==============================] - 31s 285ms/step - loss: 0.0577 - Accuracy: 0.9815 - val_loss: 0.0339 - val_Accuracy: 0.9903
Epoch 5/10
108/108 [==============================] - 31s 291ms/step - loss: 0.0488 - Accuracy: 0.9852 - val_loss: 0.0358 - val_Accuracy: 0.9908
Epoch 6/10
108/108 [==============================] - 31s 283ms/step - loss: 0.0414 - Accuracy: 0.9869 - val_loss: 0.0319 - val_Accuracy: 0.9910
Epoch 7/10
108/108 [==============================] - 30s 282ms/step - loss: 0.0350 - Accuracy: 0.9890 - val_loss: 0.0286 - val_Accuracy: 0.9923
Epoch 8/10
108/108 [==============================] - 30s 281ms/step - loss: 0.0309 - Accuracy: 0.9905 - val_loss: 0.0320 - val_Accuracy: 0.9917
Epoch 9/10
108/108 [==============================] - 31s 285ms/step - loss: 0.0273 - Accuracy: 0.9917 - val_loss: 0.0290 - val_Accuracy: 0.9922
Epoch 10/10
108/108 [==============================] - 31s 289ms/step - loss: 0.0239 - Accuracy: 0.9925 - val_loss: 0.0242 - val_Accuracy: 0.9933
```

Figure.12: The training result.

From the results, the accuracy of the validation set was 97% by the end of the first epoch, again proving that the CNN is a very good model. After ten training epoches, the accuracy was 99.3%. I expect that if we increase the number of epoches to 20, the

accuracy can be increased by 0.1-0.2%.

All the training data has been stored in the variable History. The data is then represented as an image. The code is shown in Figure 13.

```
#Plot the training history in a diagram

#Plot the accuracy diagram
plt.figure(figsize = (8,8))
plt.subplot(1,2,1)
plt.plot(History.history['Accuracy'], label = 'Train')
plt.plot(History.history['val_Accuracy'], label = 'Validation')
plt.title('The Training Accuracy History')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc = 'lower right')

#Plot the loss diagram
plt.subplot(1,2,2)
plt.plot(History.history['loss'], label = 'Train')
plt.plot(History.history['val_loss'], label = 'Validation')
plt.title('The Training Loss History')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(loc = 'upper right')
plt.show()
```

Figure.13: Code for plot.

The data extraction function is already implemented in the code. The curve labels and axis labels have been set. Run the code. The graph looks like the Figure.14.

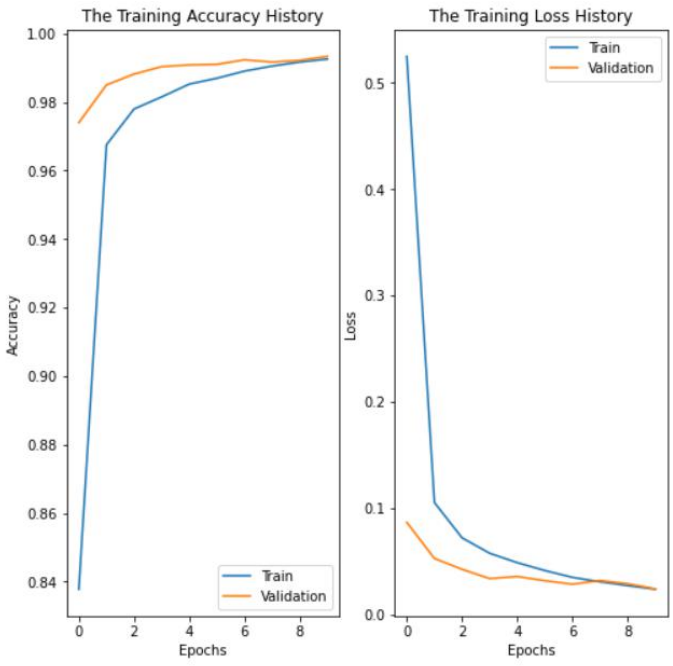

Figure.14: Training result in graph.

As can be seen from the training trend, the accuracy of the validation set is very close to that of the training set, with the final accuracy above 99%. The training process should be free of overfitting. This is already very close to the goal of this project.

4. Evaluation

Enter the final step of the project, evaluating the entire code. The code is shown in Figure 15:

```
#Test
Loss, accuracy = model.evaluate(x_test, y_test, batch_size=1, verbose=2)
#print the results out
print("The loss is: ", Loss)
print("The accuracy is :", accuracy)
```

Figure.15: Evaluation code.

The images and labels from the MNIST test set are passed into the model. Set the number of images per input batch_size to 1. verbose=2, in fact there will only be one epoch here, and verbose=2 will only output one log message. The evaluated accuracy and loss are returned and stored in accuracy and loss. The output prints the accuracy and loss.

The Final Test Result:

```
10000/10000 - 12s - loss: 0.0214 - Accuracy: 0.9932
The loss is:  0.021367348730564117
The accuracy is : 0.9932000041007996
```

Figure.16: The result of evaluation.

From Figure 16, the accuracy is 99.32%. The goal of the project has been achieved.

5. Conclusion

In such a small classification project, CNN has demonstrated its strengths in the field of image processing. Before adopting CNN, I tried to use a normal neural network for MNIST handwritten digit classification. However, when I increased the total number of neurons to 6000, my computer was already overwhelmed. It took 1 minute to get a result, and in that case, I often had to train more than 30 times to get a stable result, which meant that each training session would take more than 30 minutes. Even when I did this, the results were still only 97.8%. This is almost the starting accuracy of a CNN.

In this project, I also tried to increase the number of convolutional kernels, increase the number of training batches, change the optimizer function, and change the loss function, but the highest accuracy was only 99.6%. My expertise is too limited to pinpoint the cause of the misclassification. I can only try to combine the parameters to achieve a higher accuracy. But this also shows the upper limit of the CNN (which may be my upper limit).

From both perspectives, CNN is indeed a very good model, but CNN alone is also very limited. For an AI project, choosing the right model algorithm is the key point to deal with the problem. Based on this, we can adjust the parameters to meet our expectations. In my opinion, there are not many rules to follow for parameter tuning, and the best way is to experiment.

Reference

[1] “Module: tf.keras” Accessed on: Nov. 3, 2021, Available: https://www.tensorflow.org/api_docs/python/tf/keras/

[2] “Image classification” Accessed on Nov 3 2021, Available: https://www.tensorflow.org/tutorials/images/classification?hl=zh-tw

6. Appendix A

The MNIST Classification program code:

```
#Import the code libaries.
import tensorflow as tf
from tensorflow.keras import layers, datasets, Sequential, optimizers, utils
import numpy as np
import matplotlib.pyplot as plt


#Data preprocess.
#Load MNIST data into Variables
(x_train, y_train), (x_test, y_test) = datasets.mnist.load_data("MNIST_data")
for i in range(10):
    print('The %d train set label:'%(i),y_train[i])
    print('The %d test set label: '%(i),y_test[i])

#Reshape the data into [-1,28,28,1] form, and Normalize the data
x_train = x_train.reshape(-1,28,28,1).astype('float32')/255
x_test = x_test.reshape(-1,28,28,1).astype('float32')/255
print("Train figures sets shape:", x_train.shape)
print("Test figures sets shape:", x_test.shape)
print("Train labels sets shape:", y_train.shape)
print("Test labels sets shape:", y_test.shape)

#To convert the label data into a matricx.
#This step can transfer 0 to [1,0,0,0,0,0,0,0,0,0]. like this format.
y_train = utils.to_categorical(y_train)
y_test = utils.to_categorical(y_test)
for i in range(10):
    print('The %d train set label:'%(i),y_train[i])
    print('The %d test set label: '%(i),y_test[i])



#Creat a CNN layers
Conv_layers = [
    #First layer
    layers.Conv2D(filters=32, kernel_size = [3,3], padding = 'same', input_shape = [28,28,1],
activation = tf.nn.relu),
    layers.MaxPool2D(pool_size=[2,2]),
```

```
        #Second layer
        layers.Conv2D(filters=64, kernel_size = [3,3], padding = 'same', input_shape = [28,28,1], activation = tf.nn.relu),
        layers.MaxPool2D(pool_size=[2,2]),

        #Third layer
        layers.Conv2D(filters=128, kernel_size = [3,3], padding = 'same', input_shape = [28,28,1], activation = tf.nn.relu),
        layers.MaxPool2D(pool_size=[2,2]),

        #fLatten the all parameters
        layers.Flatten(),

        #Full connection layer with dropout 0.25.
        layers.Dense(128, activation = tf.nn.relu),
        layers.Dropout(0.25),
        layers.Dense(10, activation = tf.nn.softmax)
]


#Put those layer in to the Sequential list.
model = Sequential(Conv_layers)

#Output a report of the CNN layer.
model.summary()

#Training model

#The optimizer is Adam
#The loss function is categorical_crossentrop
#The matricx information is accuracy
model.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = 1e-3),loss = 'categorical_crossentropy',metrics = ['Accuracy'])

#Traning setting, define the epochs, batch_size. Save the hissory to a variable
History = model.fit(x_train,y_train, epochs = 10, batch_size = 500,validation_split = 0.1)

#Plot the training history in a diagram

#Plot the accuracy diagram
plt.figure(figsize = (8,8))
plt.subplot(1,2,1)
plt.plot(History.history['Accuracy'], label = 'Train')
plt.plot(History.history['val_Accuracy'], label = 'Validation')
```

```
plt.title('The Training Accuracy History')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc = 'lower right')

#Plot the loss diagram
plt.subplot(1,2,2)
plt.plot(History.history['loss'], label = 'Train')
plt.plot(History.history['val_loss'], label = 'Validation')
plt.title('The Training Loss History')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(loc = 'upper right')
plt.show()


#Test
Loss, accuracy = model.evaluate(x_test,y_test,batch_size=1,verbose=2)
#print the results out
print("The loss is: ", Loss)
print("The accuracy is :", accuracy)
```