Recognizing handwritten digits with convolution layers

1. Introduction

In lab4, we use Gradient Descent to implement a function for recognizing handwritten digits. It has 1000 gradient descent iterations with a step size of 0.01. At the end of lab4, we achieved 85 percent accuracy.

This project is to implement a new and advanced algorithm and hopefully to get much better accuracy on the result. We choose to implement CNN(Convolutional Neural Network) as the instructor advised to solve the problem here, since CNN is one of the most popular models used today, and it has very high accuracy in image recognition problems. In the meantime, we believe that our data is large enough to feed it.

2. Method

CNN is a recognition deep learning method that was developed in recent years. It is a type of feedforward Neural network including convolution calculation. It can be used in a very wide range such as facial recognition or object detection. The CNN is more commonly used to get a high accuracy when the dataset concludes images.

Compared to other image classification algorithms, this method uses relatively little pre-processing for the dataset which means it can directly input the original image. In addition, CNN can effectively reduce the number of dimensions from clear images into vague images with important features of the images remaining. This characteristic satisfied the aim of the data processing. For this project, since the dataset is images, it is hard to retain the features of images using other methods.

In this section, we use a CNN model which is designed using Keras with Tensorflow backend for the well-known MNIST digit recognition. Here are the details below:

Firstly, I used the code given by my professor to load data and from the MNIST and define x_train and x_test to indicate images and y_train and y_test to be the character labels. There are 60000 data inside the training dataset and 10000 in the testing dataset. Also, each image dimension is 28 *28 pixels.

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Then, I will reshape the dataset. To check K's format, I will use the image_data_format() function to determine if the format is channels_first or channels_last. And then input the channel dimension inside in the correct location inside the NumPy array.

```
if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, 28, 28)
    x_test = x_test.reshape(x_test.shape[0], 1, 28, 28)
    input_shape = (1, 28, 28)
else:
    x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
    x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
    input_shape = (28, 28, 1)
```

After that, I will use Keras library to build a CNN model:

```
##model building
model = tf.keras.models.Sequential()
```

Below are the steps of adding hidden layers. I choose to use one convolutional layer since the handwriting number is not complicated enough to use more layers. The result will be shown in the evaluation part. Also adding the convolution layer will need more compute time without improving efficiency. Then using maxpooling() to reduce the compute time again. After that, since there are too many dimensions and I only need the classification output, I use the Flatten() function. The reason to use the softmax activation function is that the softmax is mainly for the classification problem and can easily get the output from different categories.

```
#add a hidden convolutional layer
model.add(tf.keras.layers.Conv2D(4, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))

#this line of code is alternative
#model.add(tf.keras.layers.Conv2D(8, (3, 3), activation='relu'))

#using maxpooling
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))

#randomly turn neurons on and off to improve convergence
model.add(tf.keras.layers.Dropout(0.25))

#only want a classification output here
model.add(tf.keras.layers.Flatten())

#fully connected to get all relevant data
model.add(tf.keras.layers.Dense(128, activation='relu'))

#improve convergence again
model.add(tf.keras.layers.Dropout(0.5))

#output a softmax to squash the matrix into output probabilities
model.add(tf.keras.layers.Dense(10, activation='softmax'))
```

After building the model, I will compile it using Adam optimizer and cross-entropy to compute loss, since adam is the popular method for the optimizer nowadays.

```
model.compile(loss=tf.keras.losses.categorical_crossentropy,
              optimizer='adam',
              metrics=['accuracy'])
```

Then I use batch size=6 and epoch=3 to get my final result:

```
batch_size = 6
num_epoch = 3
#model training
model_log = model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=num_epoch,
          verbose=1,
          validation_data=(x_test, y_test))
```

Here is the final output: since I used the test dataset to be the validation_data, I will not use the testing dataset and treat the validation_data as the testing data.

```
Epoch 1/3
10000/10000 [==============================] - 34s 3ms/step - loss: 0.6039 - accuracy: 0.8057 - val_loss: 0.1302 - val_accuracy: 0.9602
Epoch 2/3
10000/10000 [==============================] - 33s 3ms/step - loss: 0.2468 - accuracy: 0.9252 - val_loss: 0.0936 - val_accuracy: 0.9724
Epoch 3/3
10000/10000 [==============================] - 35s 4ms/step - loss: 0.2001 - accuracy: 0.9368 - val_loss: 0.0804 - val_accuracy: 0.9759
```

The testing dataset should not from the training dataset or validation dataset. If there is some new data, I will use the below code to test.

```
test_loss,test_acc = model.evaluate(x=x_test,y=y_test)
print('\nTest Accuracy:',test_acc)
print('\nTest Loss',test_loss)
```

Finally, I got the result by using one convolutional layer, batch size=6 and epoch =3 to get the best performance.

```
Epoch 1/3
10000/10000 [==============================] - 34s 3ms/step - loss: 0.6039 - accuracy: 0.8057 - val_loss: 0.1302 - val_accuracy: 0.9602
Epoch 2/3
10000/10000 [==============================] - 33s 3ms/step - loss: 0.2468 - accuracy: 0.9252 - val_loss: 0.0936 - val_accuracy: 0.9724
Epoch 3/3
10000/10000 [==============================] - 35s 4ms/step - loss: 0.2001 - accuracy: 0.9368 - val_loss: 0.0804 - val_accuracy: 0.9759
```

Here are some trying before getting the best performance:

(1) For two convolutional layers, the accuracy does not have a huge difference with one convolutional layer. However, the compute time is almost two times of the one convolutional layer.

```
Epoch 1/6
10000/10000 [==============================] - 63s 6ms/step - loss: 0.4427 - accuracy: 0.8605 - val_loss: 0.0677 - val_accuracy: 0.9789
Epoch 2/6
10000/10000 [==============================] - 65s 7ms/step - loss: 0.1397 - accuracy: 0.9583 - val_loss: 0.0530 - val_accuracy: 0.9817
Epoch 3/6
10000/10000 [==============================] - 57s 6ms/step - loss: 0.1052 - accuracy: 0.9679 - val_loss: 0.0447 - val_accuracy: 0.9849
Epoch 4/6
10000/10000 [==============================] - 57s 6ms/step - loss: 0.0912 - accuracy: 0.9726 - val_loss: 0.0484 - val_accuracy: 0.9855
Epoch 5/6
10000/10000 [==============================] - 57s 6ms/step - loss: 0.0798 - accuracy: 0.9755 - val_loss: 0.0450 - val_accuracy: 0.9852
Epoch 6/6
10000/10000 [==============================] - 58s 6ms/step - loss: 0.0783 - accuracy: 0.9759 - val_loss: 0.0472 - val_accuracy: 0.9849
```

(2) If I tried to use a larger batchsize：27, then I noticed the time would be shorter; However, I didn't see a big difference in accuracy.

```
Epoch 1/6
2223/2223 [==============================] - 21s 9ms/step - loss: 0.6570 - accuracy: 0.7937 - val_loss: 0.1165 - val_accuracy: 0.9650
Epoch 2/6
2223/2223 [==============================] - 20s 9ms/step - loss: 0.2115 - accuracy: 0.9338 - val_loss: 0.0904 - val_accuracy: 0.9723
Epoch 3/6
2223/2223 [==============================] - 20s 9ms/step - loss: 0.1739 - accuracy: 0.9474 - val_loss: 0.0775 - val_accuracy: 0.9749
Epoch 4/6
2223/2223 [==============================] - 20s 9ms/step - loss: 0.1479 - accuracy: 0.9542 - val_loss: 0.0693 - val_accuracy: 0.9774
Epoch 5/6
2223/2223 [==============================] - 20s 9ms/step - loss: 0.1389 - accuracy: 0.9560 - val_loss: 0.0618 - val_accuracy: 0.9801
Epoch 6/6
2223/2223 [==============================] - 20s 9ms/step - loss: 0.1268 - accuracy: 0.9603 - val_loss: 0.0585 - val_accuracy: 0.9815
```

(3) So I increased the batch size again to 128, and the epoch changed to 3. I noticed the loss becomes greater and acc becomes lower, which is not good.

```
Epoch 1/3
469/469 [==============================] - 13s 27ms/step - loss: 0.9072 - accuracy: 0.7121 - val_loss: 0.2004 - val_accuracy: 0.9420
Epoch 2/3
469/469 [==============================] - 12s 26ms/step - loss: 0.3141 - accuracy: 0.9050 - val_loss: 0.1433 - val_accuracy: 0.9572
Epoch 3/3
469/469 [==============================] - 12s 26ms/step - loss: 0.2406 - accuracy: 0.9268 - val_loss: 0.1081 - val_accuracy: 0.9674
```

3.Evaluation

Compared the result we got from previous with the original gradient descent method with step size=0.01 and iteration=1000. The output is shown as below:

```
Running time =  157.15616083145142
Rate of success: 85.74000000000001 %
```

Here is the result of CNN method from last part as a reminder:

```
Epoch 1/3
10000/10000 [==============================] - 34s 3ms/step - loss: 0.6039 - accuracy: 0.8057 - val_loss: 0.1302 - val_accuracy: 0.9602
Epoch 2/3
10000/10000 [==============================] - 33s 3ms/step - loss: 0.2468 - accuracy: 0.9252 - val_loss: 0.0936 - val_accuracy: 0.9724
Epoch 3/3
10000/10000 [==============================] - 35s 4ms/step - loss: 0.2001 - accuracy: 0.9368 - val_loss: 0.0804 - val_accuracy: 0.9759
```

In Conclusion, with the CNN I implemented here, we got an obvious increase of accuracy with an even better running time, which has reached our expectation.

Furthermore, it is not easy to control the step size to reach good performance in the Gradient descent method. If the step size is too small, then it will need more computation time and may not reach the best performance; On the other hand, the extreme value might be passed if the step size is too large. However, with tuning hyperparameters inside of the CNN model such as batch size and epochs, I can control better for the performance and the final answer is almost 98% without overfitting.

Reference:

Adhikari, Anmol. “Hand-Written Digit Recognition Using CNN Classification(Process Explanation).” *Medium*, 23 Aug. 2020, medium.com/analytics-vidhya/hand-written-digit-recognition-using-cnn-classification-process-explanation-f6d75dcb72bb.

Mahapatra, Sambit. “A Simple 2D CNN for MNIST Digit Recognition - Towards Data Science.” *Medium*, 15 June 2018, towardsdatascience.com/a-simple-2d-cnn-for-mnist-digit-recognition-a998dbc1e79a.

Wikipedia contributors. “Convolutional Neural Network.” *Wikipedia*, 18 Apr. 2021, en.wikipedia.org/wiki/Convolutional_neural_network.