Program Code: J620-002-4:2020

Program Name: FRONT-END SOFTWARE DEVELOPMENT

Title : Exe31 - MNIST Handwriting Exercise

Name: Chong Mun Chen

IC Number: 960327-07-5097

Date : 1/8/2023

Introduction : Practising with Neural Network learning model on the MNIST handwritten digit dataset using Tensorflow's Sequential model.

Conclusion : Succeeded in achieving a highly accurate model that predicted the right number from the image.

The Problem: MNIST digit classification

We're going to tackle a classic machine learning problem: MNIST handwritten digit classification. It's simple: given an image, classify it as a digit.

In [1]:
```
import tensorflow as tf
print(tf.__version__)
```

```
2.13.0
```

In [2]:
```
#import all the required libraries
import numpy as np
import mnist
import keras

# The first time you run this might be a bit slow, since the
# mnist package has to download and cache the data.
train_images = mnist.train_images()
train_labels = mnist.train_labels()
test_images = mnist.test_images()
test_labels = mnist.test_labels()
```

Q: What's the dimension of the images data?

In [3]:
```
train_images.shape
```

Out[3]: (60000, 28, 28)

In [4]:
```
test_images.shape
```

Out[4]: (10000, 28, 28)

Q: What's the dimension of the label data?

In [5]:
```
train_labels.size
```

Out[5]: 60000

In [6]:
```
test_labels.size
```
Out[6]:
```
10000
```

Note: Curious about the dataset? try the following code. You can play around with the image_index value.

In [7]:
```
import matplotlib.pyplot as plt

image_index = 101 # You may select anything up to 60,000

print(train_labels[image_index]) # The label is 8
print(train_images[image_index])

plt.imshow(train_images[image_index], cmap='Greys')
```
```
 [  0   0   0   0   0   0   0   0   0   0   0   0  54 246 254 185   7
0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0 124 254 254  64   0
0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0  52 237 254 126   5   0
0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 118 254 254 119   0   0
0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 118 254 254  61   0   0
0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   2   6   6   2   0   0
0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0
```

2. Preparing the Data

As mentioned earlier, we need to flatten each image before we can pass it into our neural network. We'll also normalize the pixel values from [0, 255] to [-0.5, 0.5] to make our network easier to train (using smaller, centered values is often better).

In [8]:
```
# Normalize the images.
train_images = (train_images / 255) - 0.5
test_images = (test_images / 255) - 0.5

# Flatten the images.
train_images = train_images.reshape((-1, 784))
test_images = test_images.reshape((-1, 784))
```

Q: What's the dimension of the training and test images data?

In [9]:
```
train_images.shape
```

Out[9]:
```
(60000, 784)
```

In [10]:
```
test_images.shape
```

Out[10]:
```
(10000, 784)
```

3. Building the Model

Every Keras model is either built using the Sequential class, which represents a linear stack of layers, or the functional Model class, which is more customizeable. We'll be using the simpler Sequential model, since our network is indeed a linear stack of layers.

Step: Start by instantiating a Sequential model.

- The first two layers have 64 nodes each and use the ReLU activation function.
- The last layer is a Softmax output layer with 10 nodes, one for each class.

Q: what's the correct input shape for your input layer?

In [11]:
```
from keras.models import Sequential
from keras.layers import Dense

# Define the model
model = Sequential([
  Dense(64, activation='relu'),
  Dense(64, activation='relu'),
  Dense(10, activation='softmax'),
])
```

4. Compiling the Model

Before we can begin training, we need to configure the training process. We decide 3 key factors during the compilation step:

- The optimizer. We'll stick with a pretty good default: the Adam gradient-based optimizer. Keras has many other optimizers you can look into as well.
- The loss function. Since we're using a Softmax output layer, we'll use the Cross-Entropy loss. Keras distinguishes between binary_crossentropy (2 classes) and categorical_crossentropy (>2 classes), so we'll use the latter
- A list of metrics. Since this is a classification problem, we'll just have Keras report on the accuracy metric.

Step: Compile the model using the above options - adam, categorical_crossentropy, accuracy as metrics

In [12]:
```
model.compile(
  optimizer='adam',
  loss='categorical_crossentropy',
  metrics=['accuracy'],
)
```

5. Training the Model

Training a model in Keras literally consists only of calling fit() and specifying some parameters. There are a lot of possible parameters, but we'll only manually supply a few:

- The training data (images and labels), commonly known as X and Y, respectively.
- The number of epochs (iterations over the entire dataset) to train for.
- The batch size (number of samples per gradient update) to use when training.

Step: set epochs to a suitable number, and batch_size = 32

In [13]:
```
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical

# Train the model.
model.fit(
  train_images,
  to_categorical(train_labels),
  epochs=5,
  batch_size=32,
)
```

```
Epoch 1/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3581 -
accuracy: 0.8921
Epoch 2/5
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1923 -
accuracy: 0.9419
Epoch 3/5
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1525 -
accuracy: 0.9522
Epoch 4/5
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1278 -
accuracy: 0.9594
Epoch 5/5
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1106 -
accuracy: 0.9649
```

Out[13]:
```
<keras.src.callbacks.History at 0x1a20624de70>
```

Q: Do you run into any problem? Why?

In [ ]:
```

```

Q: what's your achieved accuracy?

6. Testing the Model

Step: Evaluating the model by testing against the test data

In [14]:
```
model.evaluate(
  test_images,
  to_categorical(test_labels)
)
```

```
313/313 [==============================] - 1s 1ms/step - loss: 0.1349 - a
ccuracy: 0.9598
```

Out[14]:
```
[0.1348690688610077, 0.9598000049591064]
```

7. Using the Model

Now that we have a working, trained model, let's put it to use. The first thing we'll do is save it to disk so we can load it back up anytime.

Step: save the model using the save_weights function

In [15]:
```
model.save_weights('model.h5')
```

In [16]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Build the model.
model = Sequential([
  Dense(64, activation='relu', input_shape=(784,)),
  Dense(64, activation='relu'),
  Dense(10, activation='softmax'),
])

# Load the model's saved weights.
model.load_weights('model.h5')
```

8. Predict

Using the trained model to make predictions is easy: we pass an array of inputs to predict() and it returns an array of outputs. Keep in mind that the output of our network is 10 probabilities (because of softmax), so we'll use np.argmax() to turn those into actual digits.

In [17]:
```
# Predict on the first 5 test images.
predictions = model.predict(test_images[:5])

# Print our model's predictions.
print(np.argmax(predictions, axis=1)) # [7, 2, 1, 0, 4]

# Check our predictions against the ground truths.
print(test_labels[:5]) # [7, 2, 1, 0, 4]
```

```
1/1 [==============================] - 0s 197ms/step
[7 2 1 0 4]
[7 2 1 0 4]
```

Note: What's the difference between model.save_weights and model.save? - https://stackoverflow.com/questions/42621864/difference-between-keras-model-save-and-model-save-weights#:~:text=save()%20saves%20the%20weights,to%20HDF5%20and%20nothing%20else (https://stackoverflow.com/questions/42621864/difference-between-keras-model-save-and-model-save-weights#:~:text=save()%20saves%20the%20weights,to%20HDF5%20and%20nothing%20else).

This exercise is adapted from https://victorzhou.com/blog/keras-neural-network-tutorial/ (https://victorzhou.com/blog/keras-neural-network-tutorial/)

Challenge 1:

Retrain your model by using different network depths - what will you conclude?

In [18]:
```
model = Sequential([
    Dense(64, activation = 'relu', input_shape = (784,)),
    Dense(64, activation = 'relu'),
    Dense(64, activation = 'relu'),
    Dense(64, activation = 'relu'),
    Dense(10, activation = 'softmax')
])
```

Challenge 2:

Retrain your model by using different activation (other than ReLU) - what differences does it make?

In [19]:
```
model = Sequential([
    Dense(64, activation = 'sigmoid', input_shape = (784,)),
    Dense(64, activation = 'sigmoid'),
    Dense(10, activation = 'softmax'),
])
```

Challenge 3:

In [20]:
```
from tensorflow.keras.optimizers import Adam

model.compile(
    optimizer = Adam(lr = 0.005),
    loss = 'categorical_crossentropy',
    metrics = ['accuracy']
)

model.fit(
    train_images,
    to_categorical(train_labels),
    epochs = 5,
    batch_size = 32,
    validation_data = (test_images, to_categorical(test_labels))
)
```

```
WARNING:absl:`lr` is deprecated in Keras optimizer, please use `learning_
rate` or use the legacy optimizer, e.g.,tf.keras.optimizers.legacy.Adam.

Epoch 1/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5713 -
accuracy: 0.8494 - val_loss: 0.2594 - val_accuracy: 0.9247
Epoch 2/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2316 -
accuracy: 0.9315 - val_loss: 0.1908 - val_accuracy: 0.9434
Epoch 3/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1694 -
accuracy: 0.9492 - val_loss: 0.1792 - val_accuracy: 0.9443
Epoch 4/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1345 -
accuracy: 0.9601 - val_loss: 0.1415 - val_accuracy: 0.9581
Epoch 5/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1133 -
accuracy: 0.9661 - val_loss: 0.1169 - val_accuracy: 0.9631
```

Out[20]:
```
<keras.src.callbacks.History at 0x1a206a210c0>
```

Challenge 4:

How will you load your saved weights to use it in a separate code? Upload your saved model/weights, and compare your model/weights with a model/weights from one of your classmate's.

In [21]:
```
model.save_weights('model.h5')
```

Challenge 5:

How can you load any image from the data set and let your model (or your classmate's) to predict the image?

In [22]:
```
predictions = model.predict(test_images[:5])

print(np.argmax(predictions, axis = 1))
print(test_labels[:5])
```

```
1/1 [==============================] - 0s 48ms/step
[7 2 1 0 4]
[7 2 1 0 4]
```