Forward School

Program Code: J620-002-4:2020

Program Name: FRONT-END SOFTWARE DEVELOPMENT

Title : Exe31 - MNIST Handwriting Exercise

Name: Chuay Xiang Ze

IC Number: 021224070255

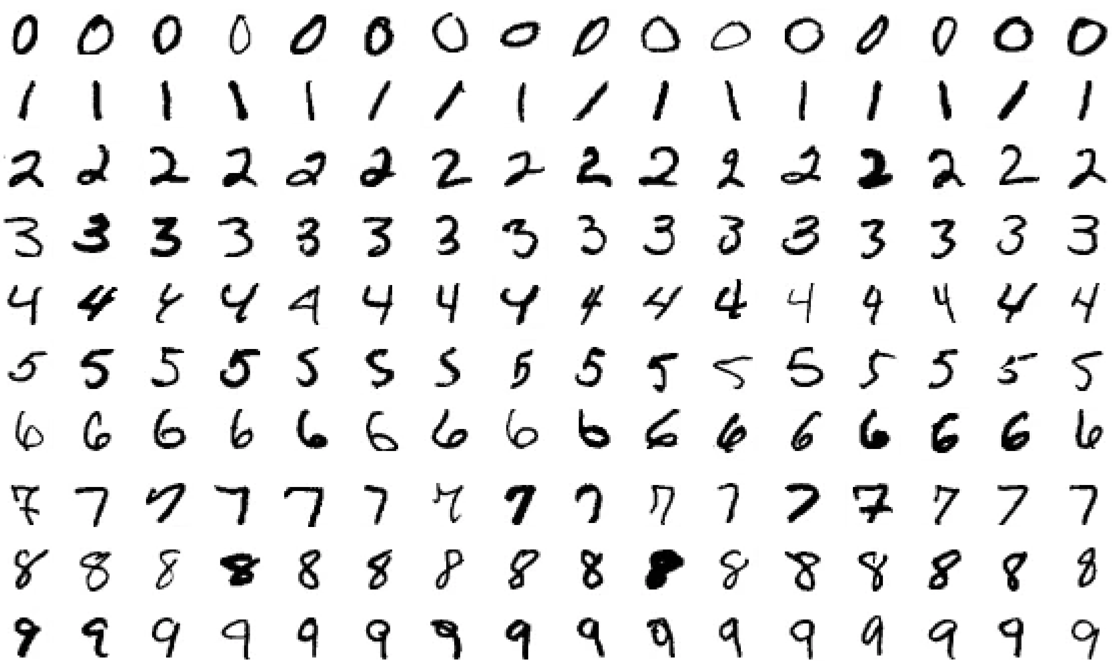
Date : 1/8/2023

Introduction : Learning how to do MNIST digit classification

Conclusion : Managed to complete tasks related to the topic

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.



Each image in the MNIST dataset is 28x28 and contains a centered, grayscale digit. We’ll flatten each 28x28 into a 784 dimensional vector, which we’ll use is input to our neural network. Our output will be one of 10 possible classes: one for each digit.

Please check that you have the following packages installed (via conda or pip) keras tensorflow numpy mnist

1. Setup

In [1]:

```
#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 [2]:

```
import tensorflow as tf
print(tf.__version__)
```

```
2.10.0
```

In [3]:

```
print('Train Images', train_images.shape)
print('Test Images', test_images.shape)
```

```
Train Images (60000, 28, 28)
Test Images (10000, 28, 28)
```

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

In [4]:

```
print('Train Labels', train_labels.shape)
print('Test Labels', test_labels.shape)
```

```
Train Labels (60000,)
Test Labels (10000,)
```

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

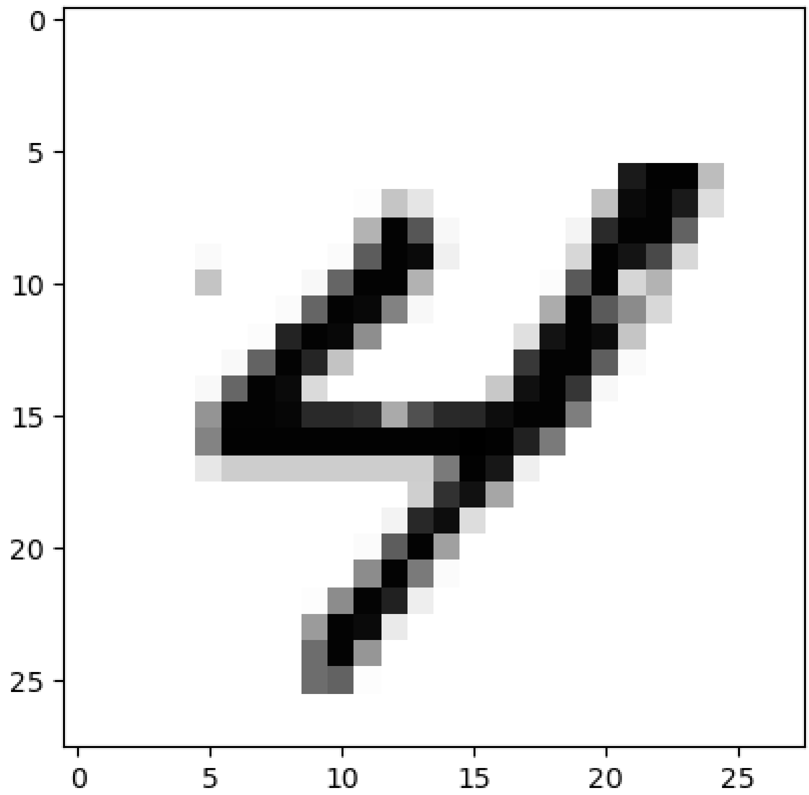
In [5]:

```
import matplotlib.pyplot as plt

image_index = 89 # 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')
```

```
4
[[  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   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   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   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   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   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
    0   0   0 232 253 253  95   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   3  86  46   0   0   0   0
    0   0  91 246 252 232  57   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 103 252 187  13   0   0   0
    0  22 219 252 252 175   0   0   0   0]
 [  0   0   0   0   0  10   0   0   0   0   8 181 252 246  30   0   0   0
    0  65 252 237 197  64   0   0   0   0]
 [  0   0   0   0   0  87   0   0   0  13 172 252 252 104   0   0   0   0
    5 184 252  67 103   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   8 172 252 248 145  14   0   0   0   0
  109 252 183 137  64   0   0   0   0   0]
 [  0   0   0   0   0   0   0   5 224 252 248 134   0   0   0   0   0  53
  238 252 245  86   0   0   0   0   0   0]
 [  0   0   0   0   0   0  12 174 252 223  88   0   0   0   0   0   0 209
  252 252 179   9   0   0   0   0   0   0]
 [  0   0   0   0   0  11 171 252 246  61   0   0   0   0   0   0  83 241
  252 211  14   0   0   0   0   0   0   0]
 [  0   0   0   0   0 129 252 252 249 220 220 215 111 192 220 221 243 252
  252 149   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0 144 253 253 253 253 253 253 253 253 253 255 253 226
  153   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0  44  77  77  77  77  77  77  77  77 153 253 235  32
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  74 214 240 114   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  24 221 243  57   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   8 180 252 119   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 136 252 153   7   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   3 136 251 226  34   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0 123 252 246  39   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0 165 252 127   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0 165 175   3   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   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
    0   0   0   0   0   0   0   0   0   0]]
```

Out[5]:

```
<matplotlib.image.AxesImage at 0x226013e7370>
```



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 [6]:

```
# 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 [7]:

```
print(train_images.shape)
print(test_images.shape)
```

```
(60000, 784)
(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 [8]:

```
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 [9]:

```
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 [10]:

```
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 [==============================] - 3s 1ms/step - loss: 0.3484 - 
accuracy: 0.8946
Epoch 2/5
1875/1875 [==============================] - 3s 1ms/step - loss: 0.1775 - 
accuracy: 0.9467
Epoch 3/5
1875/1875 [==============================] - 3s 1ms/step - loss: 0.1361 - 
accuracy: 0.9584
Epoch 4/5
1875/1875 [==============================] - 3s 1ms/step - loss: 0.1130 - 
accuracy: 0.9655
Epoch 5/5
1875/1875 [==============================] - 3s 1ms/step - loss: 0.0979 - 
accuracy: 0.9689
```

Out[10]:

```
<keras.callbacks.History at 0x226003e3d30>
```

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 [11]:

```
model.evaluate(
  test_images,
  to_categorical(test_labels)
)
```

```
313/313 [==============================] - 1s 1ms/step - loss: 0.1124 - ac
curacy: 0.9650
```

Out[11]:

```
[0.11238660663366318, 0.9649999737739563]
```

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 [12]:

```
model.save_weights('model.h5')
```

In [13]:

```
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 [23]:

```
# 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]

print(test_images[:2])
```

```
1/1 [==============================] - 0s 17ms/step
[7 2 1 0 4]
[7 2 1 0 4]
[[-0.5 -0.5 -0.5 ... -0.5 -0.5 -0.5]
 [-0.5 -0.5 -0.5 ... -0.5 -0.5 -0.5]]
```

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 [ ]:

```

```

Challenge 2:

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

In [15]:

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

Challenge 3:

Fit your model using validation_data option - What differences will that bring?

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))
)
```

```
Epoch 1/5

C:\Anaconda\envs\python-dscourse\lib\site-packages\keras\optimizers\optimi
zer_v2\adam.py:114: UserWarning: The `lr` argument is deprecated, use `lea
rning_rate` instead.
  super().__init__(name, **kwargs)

1875/1875 [==============================] - 4s 2ms/step - loss: 0.3765 -
accuracy: 0.8890 - val_loss: 0.2153 - val_accuracy: 0.9341
Epoch 2/5
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2210 -
accuracy: 0.9329 - val_loss: 0.1964 - val_accuracy: 0.9412
Epoch 3/5
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1897 -
accuracy: 0.9427 - val_loss: 0.2151 - val_accuracy: 0.9351
Epoch 4/5
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1714 -
accuracy: 0.9485 - val_loss: 0.1644 - val_accuracy: 0.9518
Epoch 5/5
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1587 -
accuracy: 0.9511 - val_loss: 0.1497 - val_accuracy: 0.9540
```

Out[20]:

```
<keras.callbacks.History at 0x22605debe50>
```

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 49ms/step
[7 2 1 0 4]
[7 2 1 0 4]
```

In [ ]:

```
```