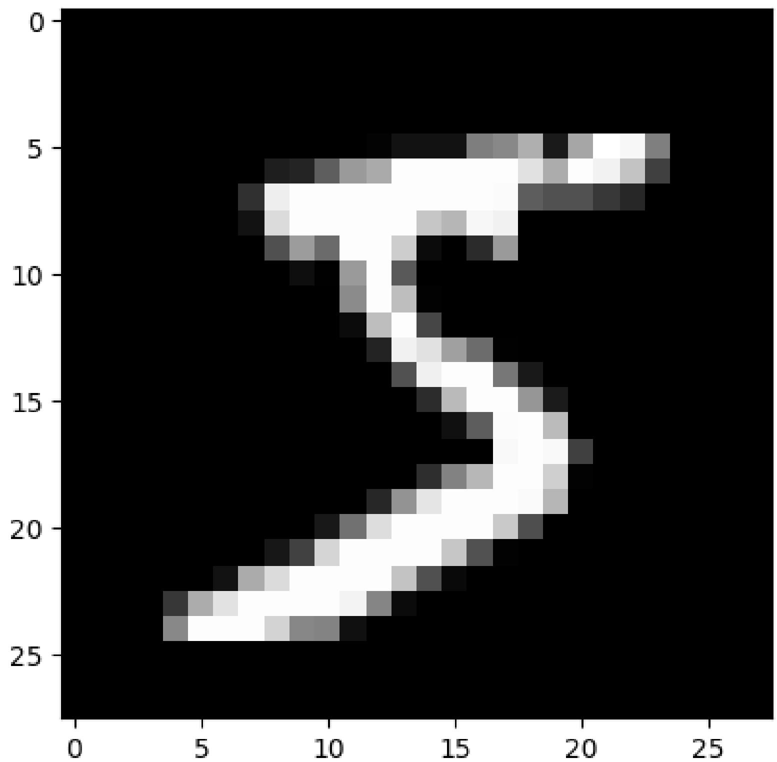
demo-mnist

April 22, 2025

[12]:
```python
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt
from sklearn import metrics

# Load the OCR dataset

# The MNIST dataset is a built-in dataset provided by Keras.
# It consists of 70,000 28x28 grayscale images, each of which displays a single␣
↪handwritten digit from 0 to 9.
# The training set consists of 60,000 images, while the test set has 10,000␣
↪images.

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

# X_train and X_test are our array of images while y_train and y_test are our␣
↪array of labels for each image.
# The first tuple contains the training set features (X_train) and the training␣
↪set labels (y_train).
# The second tuple contains the testing set features (X_test) and the testing␣
↪set labels (y_test).
# For example, if the image shows a handwritten 7, then the label will be the␣
↪intger 7.

plt.imshow(x_train[0], cmap='gray')  # imshow() function which simply displays␣
↪an image.
plt.show() # cmap is responsible for mapping a specific colormap to the values␣
↪found in the array that you passed as the first argument.

# image appears black and white and that each axis of the plot ranges from 0 to␣
↪28.

# This is because of the format that all the images in the dataset have:
```

```python
# 1. All the images are grayscale, meaning they only contain black, white and grey.
# 2. The images are 28 pixels by 25 pixels in size (28x28).

print(x_train[0])

# image data is just an array of digits. You can almost make out a 5 from the pattern of the digits in the array.
# Array of 28 values
# a grayscale pixel is stored as a digit between 0 and 255 where 0 is black, 255 is white and values in between are different shades of gray.
# Therefore, each value in the [28][28] array tells the computer which color to put in that position when we display the actual image.
```




```
[[  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   3  18  18  18 126 136
 175  26 166 255 247 127   0   0   0   0]
[  0   0   0   0   0   0   0   0  30  36  94 154 170 253 253 253 253 253
 225 172 253 242 195  64   0   0   0   0]
[  0   0   0   0   0   0   0  49 238 253 253 253 253 253 253 253 253 251
  93  82  82  56  39   0   0   0   0   0]
[  0   0   0   0   0   0   0  18 219 253 253 253 253 253 198 182 247 241
   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0  80 156 107 253 253 205  11   0  43 154
   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0  14   1 154 253  90   0   0   0   0
   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0 139 253 190   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  11 190 253  70   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  35 241 225 160 108   1
   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0  81 240 253 253 119
  25   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0  45 186 253 253
 150  27   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  16  93 252
 253 187   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 249
 253 249  64   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0  46 130 183 253
 253 207   2   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0  39 148 229 253 253 253
 250 182   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0  24 114 221 253 253 253 253 201
  78   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0  23  66 213 253 253 253 253 198  81   2
   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0  18 171 219 253 253 253 253 195  80   9   0   0
   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0  55 172 226 253 253 253 253 244 133  11   0   0   0   0
   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0 136 253 253 253 212 135 132  16   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]]
```

[13]:
```
# reformat our X_train array and our X_test array because they do not have the␣
↪correct shape.
# Reshape the data to fit the model
print("X_train shape", x_train.shape)
print("y_train shape", y_train.shape)
print("X_test shape", x_test.shape)
print("y_test shape", y_test.shape)


# Here you can see that for the training sets we have 60,000 elements and the␣
↪testing sets have 10,000 elements.

# y_train and y_test only have 1 dimensional shapes because they are just the␣
↪labels of each element.

# x_train and x_test have 3 dimensional shapes because they have a width and␣
↪height (28x28 pixels) for each element.

# (60000, 28, 28) 1st parameter in the tuple shows us how much image we have␣
↪2nd and 3rd parameters are the pixel values from x to y (28x28)
# The pixel value varies between 0 to 255.
# (60000,) Training labels with integers from 0-9 with dtype of uint8. It has␣
↪the shape (60000,).
# (10000, 28, 28) Testing data that consists of grayscale images. It has the␣
↪shape (10000, 28, 28) and the dtype of uint8. The pixel value varies between␣
↪0 to 255.
# (10000,) Testing labels that consist of integers from 0-9 with dtype uint8.␣
↪It has the shape (10000,).
```

```
X_train shape (60000, 28, 28)
y_train shape (60000,)
X_test shape (10000, 28, 28)
y_test shape (10000,)
```

[14]:
```
# X: Training data of shape (n_samples, n_features)
# y: Training label values of shape (n_samples, n_labels)
# 2D array of height and width, 28 pixels by 28 pixels will just become 784␣
↪pixels (28 squared).
# Remember that X_train has 60,000 elemenets, each with 784 total pixels so␣
↪will become shape (60000, 784).
# Whereas X_test has 10,000 elements, each with each with 784 total pixels so␣
↪will become shape (10000, 784).

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
```

```
x_train = x_train.astype('float32') # use 32-bit precision when training a␣
↪neural network, so at one point the training data will have to be converted␣
↪to 32 bit floats. Since the dataset fits easily in RAM, we might as well␣
↪convert to float immediately.
x_test = x_test.astype('float32')
x_train /= 255  # Each image has Intensity from 0 to 255
x_test /= 255

# Regarding the division by 255, this is the maximum value of a byte (the input␣
↪feature's type before the conversion to float32),
# so this will ensure that the input features are scaled between 0.0 and 1.0.
# USING svm-https://mgta.gmu.edu/courses/ml-with-python/
↪handwrittenDigitRecognition.php#:~:
↪text=Remember%20that%20X_train%20has%2060%2C000,keras.
```

[15]:
```
# Convert class vectors to binary class matrices
num_classes = 10
y_train = np.eye(num_classes)[y_train]  # Return a 2-D array with ones on the␣
↪diagonal and zeros elsewhere.
y_test = np.eye(num_classes)[y_test] # f your particular categories is present␣
↪then it mark as 1 else 0 in remain row
```

[16]:
```
# Define the model architecture
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))  # The input_shape␣
↪argument is passed to the foremost layer. It comprises of a tuple shape,
model.add(Dropout(0.2)) # DROP OUT RATIO 20%
model.add(Dense(512, activation='relu')) #returns a sequence of vectors of␣
↪dimension 512
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation='softmax'))
```

[17]:
```
# Compile the model
model.compile(loss='categorical_crossentropy',  # for a multi-class␣
↪classification problem
              optimizer=RMSprop(),
              metrics=['accuracy'])
```

[18]:
```
# Train the model
batch_size = 128 # batch_size argument is passed to the layer to define a batch␣
↪size for the inputs.
epochs = 20
history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
```

```
                    verbose=1, # verbose=1 will show you an animated progress␣
↪bar eg. [==========]
                    validation_data=(x_test, y_test)) # Using validation_data␣
↪means you are providing the training set and validation set yourself,
                                                # validation_split means␣
↪you only provide a training set and keras splits it into a training set and␣
↪a validation set
```

```
Epoch 1/20
469/469 [==============================] - 12s 24ms/step - loss: 0.2539 -
accuracy: 0.9216 - val_loss: 0.1443 - val_accuracy: 0.9547
Epoch 2/20
469/469 [==============================] - 9s 20ms/step - loss: 0.1039 -
accuracy: 0.9665 - val_loss: 0.0730 - val_accuracy: 0.9789
Epoch 3/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0742 -
accuracy: 0.9767 - val_loss: 0.0693 - val_accuracy: 0.9795
Epoch 4/20
469/469 [==============================] - 11s 22ms/step - loss: 0.0565 -
accuracy: 0.9827 - val_loss: 0.0669 - val_accuracy: 0.9811
Epoch 5/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0480 -
accuracy: 0.9850 - val_loss: 0.0747 - val_accuracy: 0.9790
Epoch 6/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0396 -
accuracy: 0.9872 - val_loss: 0.0699 - val_accuracy: 0.9817
Epoch 7/20
469/469 [==============================] - 10s 20ms/step - loss: 0.0342 -
accuracy: 0.9889 - val_loss: 0.0687 - val_accuracy: 0.9833
Epoch 8/20
469/469 [==============================] - 11s 22ms/step - loss: 0.0294 -
accuracy: 0.9903 - val_loss: 0.0661 - val_accuracy: 0.9836
Epoch 9/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0255 -
accuracy: 0.9923 - val_loss: 0.0745 - val_accuracy: 0.9822
Epoch 10/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0220 -
accuracy: 0.9930 - val_loss: 0.0734 - val_accuracy: 0.9836
Epoch 11/20
469/469 [==============================] - 9s 19ms/step - loss: 0.0202 -
accuracy: 0.9934 - val_loss: 0.0679 - val_accuracy: 0.9843
Epoch 12/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0179 -
accuracy: 0.9944 - val_loss: 0.0721 - val_accuracy: 0.9851
Epoch 13/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0170 -
accuracy: 0.9946 - val_loss: 0.0684 - val_accuracy: 0.9858
```

```
Epoch 14/20
469/469 [==============================] - 11s 23ms/step - loss: 0.0147 -
accuracy: 0.9949 - val_loss: 0.0841 - val_accuracy: 0.9822
Epoch 15/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0138 -
accuracy: 0.9956 - val_loss: 0.0802 - val_accuracy: 0.9835
Epoch 16/20
469/469 [==============================] - 10s 21ms/step - loss: 0.0139 -
accuracy: 0.9956 - val_loss: 0.0712 - val_accuracy: 0.9849
Epoch 17/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0124 -
accuracy: 0.9959 - val_loss: 0.0695 - val_accuracy: 0.9856
Epoch 18/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0120 -
accuracy: 0.9960 - val_loss: 0.0772 - val_accuracy: 0.9860
Epoch 19/20
469/469 [==============================] - 11s 23ms/step - loss: 0.0094 -
accuracy: 0.9968 - val_loss: 0.0829 - val_accuracy: 0.9851
Epoch 20/20
469/469 [==============================] - 9s 20ms/step - loss: 0.0095 -
accuracy: 0.9969 - val_loss: 0.0791 - val_accuracy: 0.9843
```

[19]:
```python
# Evaluate the model
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 0.07907029986381531
Test accuracy: 0.9843000173568726
```