

## DL\_Practical\_2

November 12, 2025

```
[1]: # a) Import the necessary packages
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

from tensorflow import keras
from tensorflow.keras import layers
print("Imports success!")
```

Imports success!

```
[2]: # b) Load the training and testing data (MNIST)

# (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

with np.load(r"C:\Users\kusha\Desktop\mnist_dataset.npz") as data:
    x_train = data["X_train"]
    y_train = data["y_train"]
    x_test = data["X_test"]
    y_test = data["y_test"]

    print(
        f"x_train:\n"
        f"  Data type: {x_train.dtype}\n"
        f"  Shape     : {x_train.shape}\n"
        f"  Pixel value range: {x_train.min()} to {x_train.max()}\n\n"
    )

    print(
        f"y_train:\n"
        f"  Data type: {y_train.dtype}\n"
        f"  Shape     : {y_train.shape}\n"
        f"  First 10 labels: {y_train[:10]}\n"
    )

    print(
        f"x_test:\n"
        f"  Data type: {x_test.dtype}\n"
```

```

        f" Shape      : {x_test.shape}\n"
        f" Pixel value range: {x_test.min()} to {x_test.max()}\n\n"
    )

    print(
        f"y_test:\n"
        f" Data type: {y_test.dtype}\n"
        f" Shape      : {y_test.shape}\n"
        f" First 10 labels: {y_test[:10]}\n"
    )

print(f"First image sample of x_train (pixel values):\n{x_train[0]}\n")

```

x\_train:  
 Data type: uint8  
 Shape : (60000, 28, 28)  
 Pixel value range: 0 to 255

y\_train:  
 Data type: uint8  
 Shape : (60000,)  
 First 10 labels: [5 0 4 1 9 2 1 3 1 4]

x\_test:  
 Data type: uint8  
 Shape : (10000, 28, 28)  
 Pixel value range: 0 to 255

y\_test:  
 Data type: uint8  
 Shape : (10000,)  
 First 10 labels: [7 2 1 0 4 1 4 9 5 9]

First image sample of x\_train (pixel values):

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
175	26	166	255	247	127	0	0	0	0	0	0	0	0	3	18	18	18	126	136



```
[3]: # c) Define the network architecture using Keras
# Preprocess - normalize pixel values to [0, 1]
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

print(
    f"\nAfter normalization:\n"
    f"x_train:\n"
    f" Data type: {x_train.dtype}\n"
    f" Shape : {x_train.shape}\n"
    f" Pixel value range: {x_train.min()} to {x_train.max()}\n\n"

    f"x_test:\n"
    f" Data type: {x_test.dtype}\n"
    f" Shape : {x_test.shape}\n"
    f" Pixel value range: {x_test.min()} to {x_test.max()}\n\n"

    f" Sample normalized image of x_train (first image pixels):\n"
    f"\n{x_train[0]}\n"
)
```

After normalization:

x\_train:

  Data type: float32  
  Shape : (60000, 28, 28)  
  Pixel value range: 0.0 to 1.0

x\_test:

  Data type: float32  
  Shape : (10000, 28, 28)  
  Pixel value range: 0.0 to 1.0

  Sample normalized image of x\_train (first image pixels):

[[0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	]
[0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	]
[0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.

0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.01176471	0.07058824	0.07058824	0.07058824	0.49411765	0.53333336
0.6862745	0.10196079	0.6509804	1.	0.96862745	0.49803922
0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.	0.11764706	0.14117648	0.36862746	0.6039216
0.6666667	0.99215686	0.99215686	0.99215686	0.99215686	0.99215686
0.88235295	0.6745098	0.99215686	0.9490196	0.7647059	0.2509804
0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.19215687	0.93333334	0.99215686	0.99215686	0.99215686
0.99215686	0.99215686	0.99215686	0.99215686	0.99215686	0.9843137
0.3647059	0.32156864	0.32156864	0.21960784	0.15294118	0.
0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.07058824	0.85882354	0.99215686	0.99215686	0.99215686
0.99215686	0.99215686	0.7764706	0.7137255	0.96862745	0.94509804
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.	0.3137255	0.6117647	0.41960785	0.99215686
0.99215686	0.8039216	0.04313726	0.	0.16862746	0.6039216
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.	0.	0.05490196	0.00392157	0.6039216
0.99215686	0.3529412	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.54509807
0.99215686	0.74509805	0.00784314	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.04313726

0.74509805	0.99215686	0.27450982	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.13725491	0.94509804	0.88235295	0.627451	0.42352942	0.00392157
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.31764707	0.9411765	0.99215686	0.99215686	0.46666667
0.09803922	0.	0.	0.	0.	0.
0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.1764706	0.7294118	0.99215686	0.99215686
0.5882353	0.10588235	0.	0.	0.	0.
0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.0627451	0.3647059	0.9882353
0.99215686	0.73333335	0.	0.	0.	0.
0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.9764706
0.99215686	0.9764706	0.2509804	0.	0.	0.
0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.	0.	0.18039216	0.50980395	0.7176471	0.99215686
0.99215686	0.8117647	0.00784314	0.	0.	0.
0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.
0.15294118	0.5803922	0.8980392	0.99215686	0.99215686	0.99215686
0.98039216	0.7137255	0.	0.	0.	0.
0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.09411765	0.44705883
0.86666667	0.99215686	0.99215686	0.99215686	0.99215686	0.7882353
0.30588236	0.	0.	0.	0.	0.
0.	0.	0.	0.	]	
[0.	0.	0.	0.	0.	0.
0.	0.	0.09019608	0.25882354	0.8352941	0.99215686
0.99215686	0.99215686	0.99215686	0.7764706	0.31764707	0.00784314
0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	]	

```
[0.          0.          0.          0.          0.          0.
 0.07058824 0.67058825 0.85882354 0.99215686 0.99215686 0.99215686
 0.99215686 0.7647059  0.3137255  0.03529412 0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          ]]
[0.          0.          0.          0.          0.21568628 0.6745098
 0.8862745  0.99215686 0.99215686 0.99215686 0.99215686 0.95686275
 0.52156866 0.04313726 0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          ]]
[0.          0.          0.          0.          0.53333336 0.99215686
 0.99215686 0.99215686 0.83137256 0.5294118  0.5176471  0.0627451
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          ]]
[0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          ]]
[0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          ]]
[0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          ]]
Model: "sequential"
```

Layer (type)	Output Shape	
Param #		
flatten (Flatten)	(None, 784)	
0		

```
dense (Dense)           (None, 128) □  
↳ 100,480
```

```
dense_1 (Dense)          (None, 10) □  
↳ 1,290
```

Total params: 101,770 (397.54 KB)

Trainable params: 101,770 (397.54 KB)

Non-trainable params: 0 (0.00 B)

```
[5]: # d) Train the model using SGD optimizer  
model.compile(  
    optimizer='sgd',  
    loss='sparse_categorical_crossentropy',  
    metrics=['accuracy'])  
)  
history = model.fit(  
    x_train, y_train,  
    validation_data=(x_test, y_test),  
    epochs=10,  
    verbose=2  
)
```

```
Epoch 1/10  
1875/1875 - 4s - 2ms/step - accuracy: 0.8299 - loss: 0.6760 - val_accuracy:  
0.9004 - val_loss: 0.3689  
Epoch 2/10  
1875/1875 - 3s - 2ms/step - accuracy: 0.9044 - loss: 0.3427 - val_accuracy:  
0.9159 - val_loss: 0.2990  
Epoch 3/10  
1875/1875 - 3s - 1ms/step - accuracy: 0.9182 - loss: 0.2915 - val_accuracy:  
0.9250 - val_loss: 0.2632  
Epoch 4/10  
1875/1875 - 3s - 1ms/step - accuracy: 0.9267 - loss: 0.2595 - val_accuracy:  
0.9314 - val_loss: 0.2392  
Epoch 5/10  
1875/1875 - 3s - 1ms/step - accuracy: 0.9341 - loss: 0.2355 - val_accuracy:  
0.9381 - val_loss: 0.2192  
Epoch 6/10  
1875/1875 - 3s - 2ms/step - accuracy: 0.9398 - loss: 0.2163 - val_accuracy:  
0.9422 - val_loss: 0.2026
```

```
Epoch 7/10
1875/1875 - 3s - 2ms/step - accuracy: 0.9442 - loss: 0.2004 - val_accuracy:
0.9461 - val_loss: 0.1895
Epoch 8/10
1875/1875 - 3s - 2ms/step - accuracy: 0.9474 - loss: 0.1864 - val_accuracy:
0.9492 - val_loss: 0.1781
Epoch 9/10
1875/1875 - 3s - 2ms/step - accuracy: 0.9512 - loss: 0.1745 - val_accuracy:
0.9510 - val_loss: 0.1692
Epoch 10/10
1875/1875 - 3s - 2ms/step - accuracy: 0.9541 - loss: 0.1637 - val_accuracy:
0.9538 - val_loss: 0.1607
```

```
[6]: # e) Evaluate the network
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=1)
print(f"\nTest accuracy: {test_acc:.4f}")
print(f"Test loss: {test_loss:.4f}")
```

```
313/313          0s 1ms/step -
accuracy: 0.9451 - loss: 0.1869
```

```
Test accuracy: 0.9538
Test loss: 0.1607
```

```
[7]: # Show predictions on test set samples
count = 9
predictions = model.predict(x_test[:count])
predicted_labels = np.argmax(predictions, axis=1)

plt.figure(figsize=(12, 5))
for i in range(count):
    plt.subplot(1, count, i + 1)
    plt.imshow(x_test[i], cmap='gray')
    plt.title(f"Pred: {predicted_labels[i]}\nTrue: {y_test[i]}")
    plt.axis('off')
plt.suptitle("Prediction vs Actual Labels on Test Samples")
plt.show()
```

```
1/1          0s 67ms/step
```

### Prediction vs Actual Labels on Test Samples

Pred: 7 True: 7	Pred: 2 True: 2	Pred: 1 True: 1	Pred: 0 True: 0	Pred: 4 True: 4	Pred: 1 True: 1	Pred: 4 True: 4	Pred: 9 True: 9	Pred: 6 True: 5
7	2	1	0	4	1	4	9	5

```
[8]: # f) Plot the training loss and accuracy
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss During Training')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy During Training')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```

