mlp_week09

Alexandru Girban (S2148980), Hariaksh Pandya (S2692608)

March 20, 2025

1 Introduction to Neural network (Keras + MNIST)

1.0.1 Aims

The main concepts covered in this notebook are:

- getting familiar with basic keras
- input-output with keras
- construciton of neural network models with keras

[1]:
```
pip install tensorflow
```

```
Collecting tensorflow
  Downloading tensorflow-2.19.0-cp311-cp311-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (4.1 kB)
Collecting absl-py>=1.0.0 (from tensorflow)
  Downloading absl_py-2.1.0-py3-none-any.whl.metadata (2.3 kB)
Collecting astunparse>=1.6.0 (from tensorflow)
  Downloading astunparse-1.6.3-py2.py3-none-any.whl.metadata (4.4 kB)
Collecting flatbuffers>=24.3.25 (from tensorflow)
  Downloading flatbuffers-25.2.10-py2.py3-none-any.whl.metadata (875 bytes)
Collecting gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 (from tensorflow)
  Downloading gast-0.6.0-py3-none-any.whl.metadata (1.3 kB)
Collecting google-pasta>=0.1.1 (from tensorflow)
  Downloading google_pasta-0.2.0-py3-none-any.whl.metadata (814 bytes)
Collecting libclang>=13.0.0 (from tensorflow)
  Downloading libclang-18.1.1-py2.py3-none-manylinux2010_x86_64.whl.metadata
(5.2 kB)
Collecting opt-einsum>=2.3.2 (from tensorflow)
  Downloading opt_einsum-3.4.0-py3-none-any.whl.metadata (6.3 kB)
Requirement already satisfied: packaging in /opt/conda/lib/python3.11/site-
packages (from tensorflow) (24.0)
Requirement already satisfied:
protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.3
in /opt/conda/lib/python3.11/site-packages (from tensorflow) (4.25.3)
Requirement already satisfied: requests<3,>=2.21.0 in
/opt/conda/lib/python3.11/site-packages (from tensorflow) (2.32.3)
Requirement already satisfied: setuptools in /opt/conda/lib/python3.11/site-
packages (from tensorflow) (69.5.1)
```

```
Requirement already satisfied: six>=1.12.0 in /opt/conda/lib/python3.11/site-
packages (from tensorflow) (1.16.0)
Collecting termcolor>=1.1.0 (from tensorflow)
  Downloading termcolor-2.5.0-py3-none-any.whl.metadata (6.1 kB)
Requirement already satisfied: typing-extensions>=3.6.6 in
/opt/conda/lib/python3.11/site-packages (from tensorflow) (4.11.0)
Collecting wrapt>=1.11.0 (from tensorflow)
  Downloading wrapt-1.17.2-cp311-cp311-
manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_6
4.whl.metadata (6.4 kB)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in
/opt/conda/lib/python3.11/site-packages (from tensorflow) (1.62.2)
Collecting tensorboard~=2.19.0 (from tensorflow)
  Downloading tensorboard-2.19.0-py3-none-any.whl.metadata (1.8 kB)
Collecting keras>=3.5.0 (from tensorflow)
  Downloading keras-3.9.0-py3-none-any.whl.metadata (6.1 kB)
Requirement already satisfied: numpy<2.2.0,>=1.26.0 in
/opt/conda/lib/python3.11/site-packages (from tensorflow) (1.26.4)
Requirement already satisfied: h5py>=3.11.0 in /opt/conda/lib/python3.11/site-
packages (from tensorflow) (3.11.0)
Collecting ml-dtypes<1.0.0,>=0.5.1 (from tensorflow)
  Downloading ml_dtypes-0.5.1-cp311-cp311-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (21 kB)
Collecting tensorflow-io-gcs-filesystem>=0.23.1 (from tensorflow)
  Downloading tensorflow_io_gcs_filesystem-0.37.1-cp311-cp311-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (14 kB)
Requirement already satisfied: wheel<1.0,>=0.23.0 in
/opt/conda/lib/python3.11/site-packages (from astunparse>=1.6.0->tensorflow)
(0.43.0)
Requirement already satisfied: rich in /opt/conda/lib/python3.11/site-packages
(from keras>=3.5.0->tensorflow) (13.9.3)
Collecting namex (from keras>=3.5.0->tensorflow)
  Downloading namex-0.0.8-py3-none-any.whl.metadata (246 bytes)
Collecting optree (from keras>=3.5.0->tensorflow)
  Downloading optree-0.14.1-cp311-cp311-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (49 kB)
                           49.1/49.1 kB
1.1 MB/s eta 0:00:00a 0:00:01
Requirement already satisfied: charset-normalizer<4,>=2 in
/opt/conda/lib/python3.11/site-packages (from requests<3,>=2.21.0->tensorflow)
(3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /opt/conda/lib/python3.11/site-
packages (from requests<3,>=2.21.0->tensorflow) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/opt/conda/lib/python3.11/site-packages (from requests<3,>=2.21.0->tensorflow)
(2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in
/opt/conda/lib/python3.11/site-packages (from requests<3,>=2.21.0->tensorflow)
```

```
(2024.8.30)
Collecting markdown>=2.6.8 (from tensorboard~=2.19.0->tensorflow)
  Downloading Markdown-3.7-py3-none-any.whl.metadata (7.0 kB)
Collecting tensorboard-data-server<0.8.0,>=0.7.0 (from
tensorboard~=2.19.0->tensorflow)
  Downloading tensorboard_data_server-0.7.2-py3-none-
manylinux_2_31_x86_64.whl.metadata (1.1 kB)
Collecting werkzeug>=1.0.1 (from tensorboard~=2.19.0->tensorflow)
  Downloading werkzeug-3.1.3-py3-none-any.whl.metadata (3.7 kB)
Requirement already satisfied: MarkupSafe>=2.1.1 in
/opt/conda/lib/python3.11/site-packages (from
werkzeug>=1.0.1->tensorboard~=2.19.0->tensorflow) (2.1.5)
Requirement already satisfied: markdown-it-py>=2.2.0 in
/opt/conda/lib/python3.11/site-packages (from rich->keras>=3.5.0->tensorflow)
(3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in
/opt/conda/lib/python3.11/site-packages (from rich->keras>=3.5.0->tensorflow)
(2.17.2)
Requirement already satisfied: mdurl~=0.1 in /opt/conda/lib/python3.11/site-
packages (from markdown-it-py>=2.2.0->rich->keras>=3.5.0->tensorflow) (0.1.2)
Downloading
tensorflow-2.19.0-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
(644.9 MB)
                         644.9/644.9 MB
4.2 MB/s eta 0:00:00:00:0100:01
Downloading absl_py-2.1.0-py3-none-any.whl (133 kB)
                         133.7/133.7 kB
3.3 MB/s eta 0:00:00:00:01
Downloading astunparse-1.6.3-py2.py3-none-any.whl (12 kB)
Downloading flatbuffers-25.2.10-py2.py3-none-any.whl (30 kB)
Downloading gast-0.6.0-py3-none-any.whl (21 kB)
Downloading google_pasta-0.2.0-py3-none-any.whl (57 kB)
                         57.5/57.5 kB
1.6 MB/s eta 0:00:00
Downloading keras-3.9.0-py3-none-any.whl (1.3 MB)
                         1.3/1.3 MB
35.5 MB/s eta 0:00:00:00:01
Downloading libclang-18.1.1-py2.py3-none-manylinux2010_x86_64.whl (24.5
MB)
                         24.5/24.5 MB
64.4 MB/s eta 0:00:00:00:0100:01
Downloading
ml_dtypes-0.5.1-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (4.7
MB)
                         4.7/4.7 MB
54.1 MB/s eta 0:00:00:00:01
Downloading opt_einsum-3.4.0-py3-none-any.whl (71 kB)
                         71.9/71.9 kB
```

```
2.6 MB/s eta 0:00:00
Downloading tensorboard-2.19.0-py3-none-any.whl (5.5 MB)
                         5.5/5.5 MB
64.3 MB/s eta 0:00:00:00:0100:01
Downloading tensorflow_io_gcs_filesystem-0.37.1-cp311-cp311-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl (5.1 MB)
                         5.1/5.1 MB
76.3 MB/s eta 0:00:00:00:01
Downloading termcolor-2.5.0-py3-none-any.whl (7.8 kB)
Downloading wrapt-1.17.2-cp311-cp311-
manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_6
4.whl (83 kB)
                         83.2/83.2 kB
2.8 MB/s eta 0:00:00
Downloading Markdown-3.7-py3-none-any.whl (106 kB)
                         106.3/106.3 kB
3.9 MB/s eta 0:00:00
Downloading tensorboard_data_server-0.7.2-py3-none-
manylinux_2_31_x86_64.whl (6.6 MB)
                         6.6/6.6 MB
77.5 MB/s eta 0:00:00:00:0100:01
Downloading werkzeug-3.1.3-py3-none-any.whl (224 kB)
                         224.5/224.5 kB
8.0 MB/s eta 0:00:00
Downloading namex-0.0.8-py3-none-any.whl (5.8 kB)
Downloading
optree-0.14.1-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (408
kB)
                         408.9/408.9 kB
15.7 MB/s eta 0:00:00
Installing collected packages: namex, libclang, flatbuffers, wrapt,
werkzeug, termcolor, tensorflow-io-gcs-filesystem, tensorboard-data-server,
optree, opt-einsum, ml-dtypes, markdown, google-pasta, gast, astunparse, absl-
py, tensorboard, keras, tensorflow
Successfully installed absl-py-2.1.0 astunparse-1.6.3 flatbuffers-25.2.10
gast-0.6.0 google-pasta-0.2.0 keras-3.9.0 libclang-18.1.1 markdown-3.7 ml-
dtypes-0.5.1 namex-0.0.8 opt-einsum-3.4.0 optree-0.14.1 tensorboard-2.19.0
tensorboard-data-server-0.7.2 tensorflow-2.19.0 tensorflow-io-gcs-
filesystem-0.37.1 termcolor-2.5.0 werkzeug-3.1.3 wrapt-1.17.2
Note: you may need to restart the kernel to use updated packages.
```

[2]:
```python
# Import necessary libraries
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
```

The following code boxes will allow you to visualise your model training. Scroll back up to take a look once you get to a "model.fit" statement! (You'll need to refresh the dashboard with the refresh button on the top right)

[3]:
```
import os, datetime
logdir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
```

[5]:
```
%load_ext tensorboard
%tensorboard --port=5036 --logdir $logdir
tensorboard_callback = keras.callbacks.TensorBoard(logdir, histogram_freq=1)
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

```
Reusing TensorBoard on port 5036 (pid 646), started 0:00:14 ago. (Use '!kill␣
↪646' to kill it.)
```

```
<IPython.core.display.HTML object>
```

2 Part 1: Sythetic Data

In the first part of this workshop we will work with a "clean" dataset, generating data from purely deterministic functions.

First, generate data according to

$$x_2 = \cos(x_1), \tag{1}$$

with $x_1 \in [-\pi, \pi]$. This data will be labelled as belonging to class $y = 0$.

For data in class $y = 1$, generate

$$x_2 = a + \cos(x_1), \tag{2}$$

where $a = 1$ for now.

You should generate ~2000 samples for x_1 (uniform distribution over $x_1 \sim U[-\pi, \pi]$).

For use in keras, it helps to build numpy arrays of following shape

$$X.\text{shape} = (N, D)$$

with a corresponding set of labels

$$y.\text{shape} = (N,)$$

where N is the number of samples and D the number of features ($D = 2$ here).

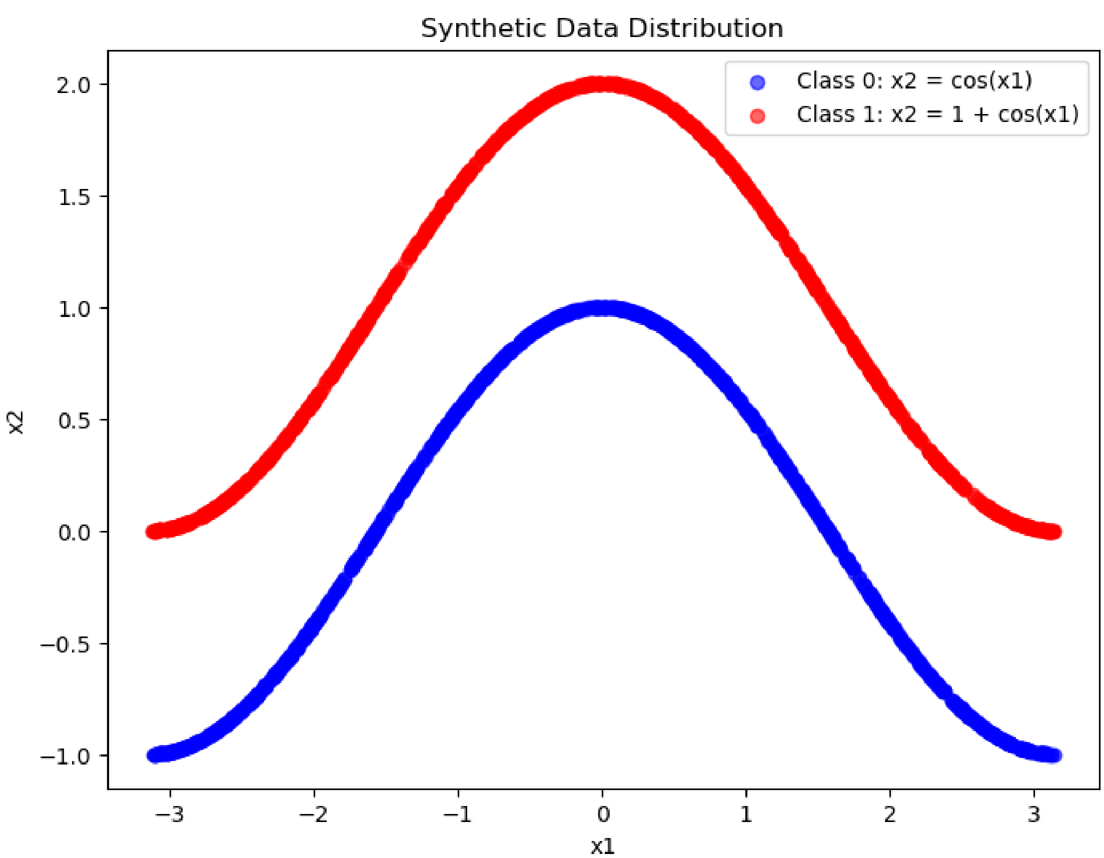
[6]:
```
N = 1000
np.random.seed(42)
```

[7]:
```
x1_0 = np.random.uniform(-np.pi, np.pi, N)
x2_0 = np.cos(x1_0)
y_0  = np.zeros(N)
```

[8]:
```
x1_1 = np.random.uniform(-np.pi, np.pi, N)
x2_1 = 1 + np.cos(x1_1)
y_1  = np.ones(N)
```

[9]:
```
X_0 = np.column_stack([x1_0, x2_0])
X_1 = np.column_stack([x1_1, x2_1])
X = np.concatenate([X_0, X_1], axis=0)
y = np.concatenate([y_0, y_1], axis=0)
```

[10]:
```
plt.figure(figsize=(8,6))
plt.scatter(X_0[:, 0], X_0[:, 1], color='blue', alpha=0.6, label='Class 0: x2 =␣
↪cos(x1)')
plt.scatter(X_1[:, 0], X_1[:, 1], color='red', alpha=0.6, label='Class 1: x2 =␣
↪1 + cos(x1)')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Synthetic Data Distribution')
plt.legend()
plt.show()
```



2.0.1 Exercise 1 (CORE): Building a Baseline Model

a) Build a logistic regression model in keras. The model should consist of an input layer and a fully-connected output layer. No hideen layer for now. See lecture notes for details of how to create these objects, or ask your tutors.

b) Compile the model. At this stage you need to select a loss function (specified via the "loss" keyword) and an optimizer. Any optimizer will do – you could use one of the "exciting" ones, e.g. Adam.

c. Train the model with model.fit. Pass the keyword argument

```
# callbacks=[tensorboard_callback]
```

to visualise above.

You might also want to split the dataset into a training and validation component via

```
# validation_split=0.X
```

[12]:
```python
model_baseline = keras.Sequential([
    layers.Dense(1, activation='sigmoid', input_shape=(2,))
])
```

[13]:
```python
model_baseline.compile(optimizer='adam',
                       loss='binary_crossentropy',
                       metrics=['accuracy'])
```

[14]:
```python
history_baseline = model_baseline.fit(
    X,
    y,
    epochs=10,
    batch_size=32,
    validation_split=0.2,
    callbacks=[tensorboard_callback]
)
```

```
Epoch 1/10
50/50            1s 8ms/step -
accuracy: 0.5067 - loss: 0.6562 - val_accuracy: 0.7675 - val_loss: 0.4559
Epoch 2/10
50/50            0s 4ms/step -
accuracy: 0.5040 - loss: 0.6412 - val_accuracy: 0.7750 - val_loss: 0.4560
Epoch 3/10
50/50            0s 4ms/step -
accuracy: 0.5712 - loss: 0.6305 - val_accuracy: 0.7775 - val_loss: 0.4565
Epoch 4/10
50/50            0s 4ms/step -
accuracy: 0.6131 - loss: 0.6133 - val_accuracy: 0.7850 - val_loss: 0.4579
Epoch 5/10
50/50            0s 4ms/step -
```

```
accuracy: 0.6563 - loss: 0.5801 - val_accuracy: 0.7875 - val_loss: 0.4610
Epoch 6/10
50/50                0s 4ms/step -
accuracy: 0.6802 - loss: 0.5622 - val_accuracy: 0.7900 - val_loss: 0.4659
Epoch 7/10
50/50                0s 5ms/step -
accuracy: 0.6610 - loss: 0.5717 - val_accuracy: 0.7925 - val_loss: 0.4719
Epoch 8/10
50/50                0s 4ms/step -
accuracy: 0.6822 - loss: 0.5599 - val_accuracy: 0.7950 - val_loss: 0.4775
Epoch 9/10
50/50                0s 4ms/step -
accuracy: 0.6805 - loss: 0.5611 - val_accuracy: 0.7950 - val_loss: 0.4844
Epoch 10/10
50/50                0s 3ms/step -
accuracy: 0.7101 - loss: 0.5374 - val_accuracy: 0.7300 - val_loss: 0.4898
```

2.0.2 Exercise 2 (CORE): Testing your model

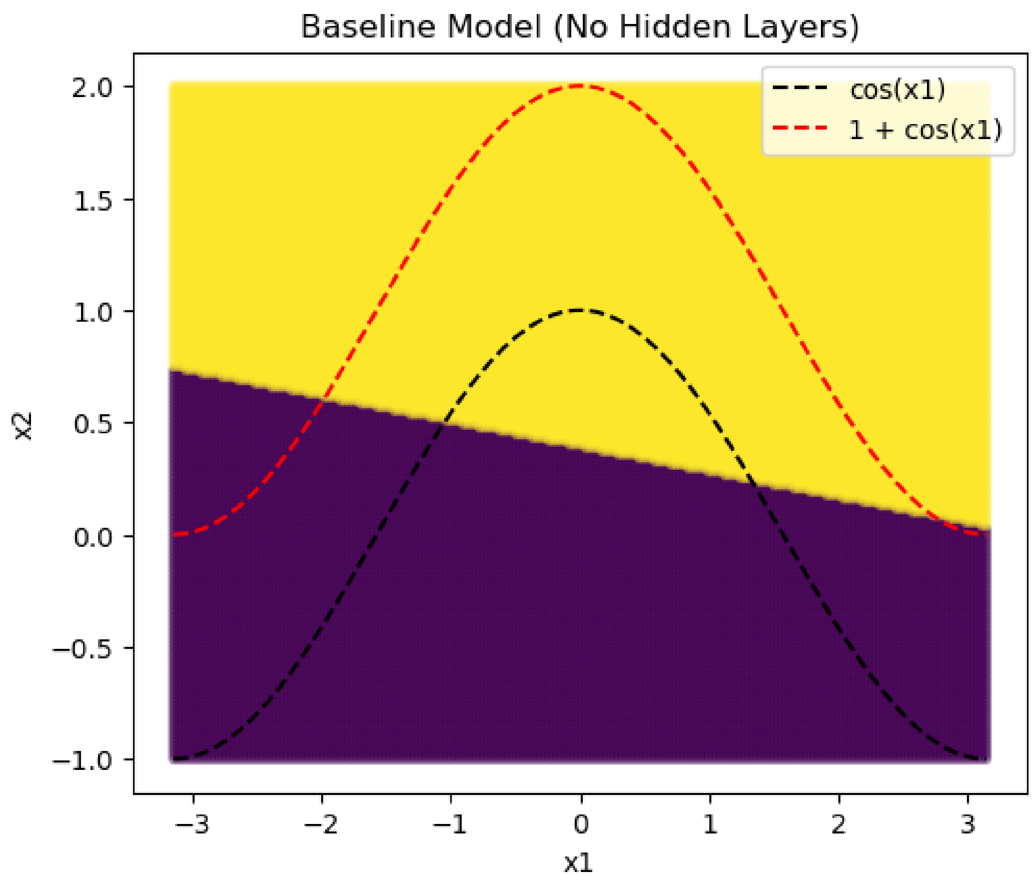
Generate "test" data uniformly over $x_1 \in [-\pi, \pi]$ and $x_2 \in [-1, 1 + a]$. Use your trained model to predict the y labels for this data and visualise the results. Overlay the original curves on the output – is the result what you expect, and why?

[15]:
```
test_x1 = np.linspace(-np.pi, np.pi, 200)
test_x2 = np.linspace(-1, 2, 200)
xx1, xx2 = np.meshgrid(test_x1, test_x2)
grid_points = np.column_stack([xx1.ravel(), xx2.ravel()])
```

[16]:
```
preds = model_baseline.predict(grid_points)
preds_class = (preds > 0.5).astype(int).ravel()
```

```
1250/1250            1s 819us/step
```

[17]:
```
plt.figure(figsize=(6,5))
plt.title("Baseline Model (No Hidden Layers)")
plt.scatter(grid_points[:,0], grid_points[:,1], c=preds_class, alpha=0.3, s=10)
x_curve = np.linspace(-np.pi, np.pi, 200)
plt.plot(x_curve, np.cos(x_curve), 'k--', label='cos(x1)')
plt.plot(x_curve, 1 + np.cos(x_curve), 'r--', label='1 + cos(x1)')
plt.legend()
plt.xlabel('x1')
plt.ylabel('x2')
plt.show()
```



The preditcion is clearly wrong. We xan assume that the absence of hidden layers has left the system with too little complexity to recreate the true distribution. In toher word, the poool of functions in the form $G(x) = \sum_n \sigma(\alpha_n x_n + \theta_n)$ is too small to correctly approximate $f(x)$

2.0.3 Exercise 3 (CORE): Building a Baseline Model

Now create a new model by adding a fully-connected hidden layer with 2 neurons between your input and output above.

Train the new model and visualise the same test data from above.

[19]:
```
#making hidden layer
model_hidden = keras.Sequential([
    layers.Dense(2, activation='relu', input_shape=(2,)),
    layers.Dense(1, activation='sigmoid')
])
```

[20]:
```
model_hidden.compile(optimizer='adam',
                     loss='binary_crossentropy',
                     metrics=['accuracy'])
```

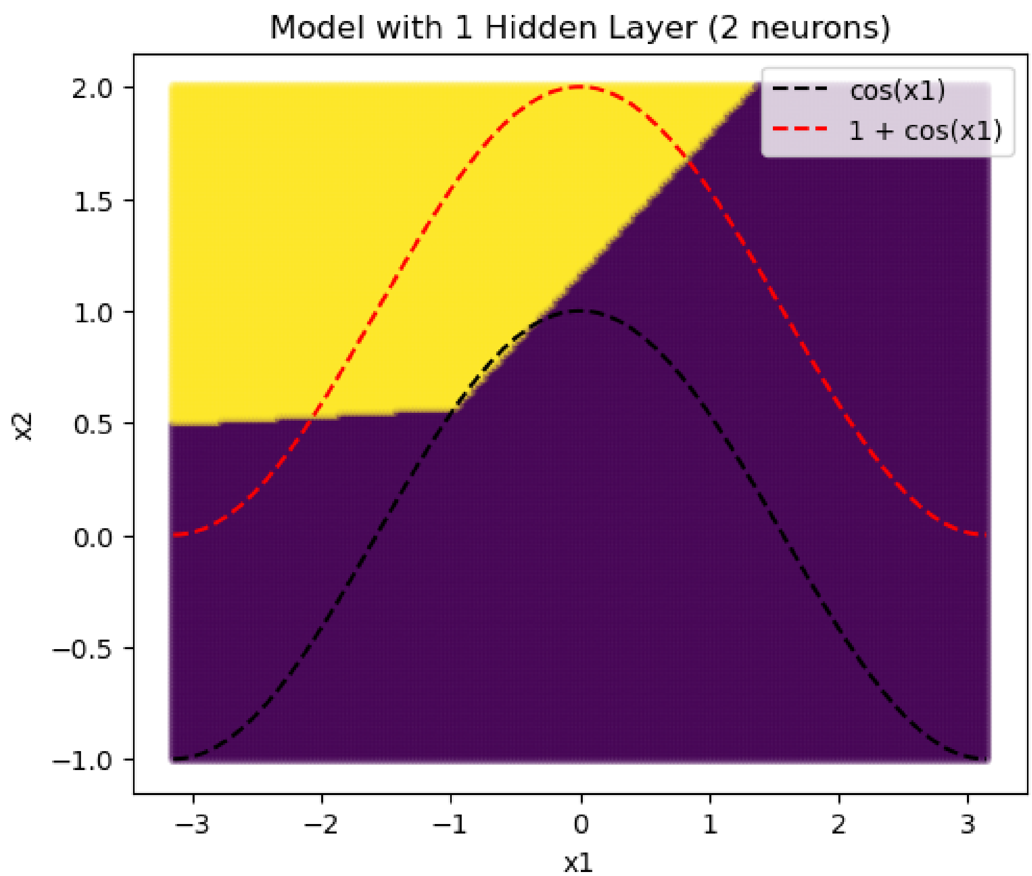
[21]:
```
history_hidden = model_hidden.fit(
    X,
    y,
    epochs=10,
    batch_size=32,
    validation_split=0.2,
    callbacks=[tensorboard_callback]
)
```

```
Epoch 1/10
50/50            1s 7ms/step -
accuracy: 0.4933 - loss: 0.7164 - val_accuracy: 0.6650 - val_loss: 0.6170
Epoch 2/10
50/50            0s 5ms/step -
accuracy: 0.5020 - loss: 0.6963 - val_accuracy: 0.6250 - val_loss: 0.6540
Epoch 3/10
50/50            0s 5ms/step -
accuracy: 0.5205 - loss: 0.6840 - val_accuracy: 0.5800 - val_loss: 0.6836
Epoch 4/10
50/50            0s 4ms/step -
accuracy: 0.6777 - loss: 0.6727 - val_accuracy: 0.5550 - val_loss: 0.7036
Epoch 5/10
50/50            0s 4ms/step -
accuracy: 0.8253 - loss: 0.6650 - val_accuracy: 0.4350 - val_loss: 0.7185
Epoch 6/10
50/50            0s 4ms/step -
accuracy: 0.7899 - loss: 0.6533 - val_accuracy: 0.3925 - val_loss: 0.7250
Epoch 7/10
50/50            0s 4ms/step -
accuracy: 0.7596 - loss: 0.6467 - val_accuracy: 0.3875 - val_loss: 0.7317
Epoch 8/10
50/50            0s 4ms/step -
accuracy: 0.7498 - loss: 0.6383 - val_accuracy: 0.4000 - val_loss: 0.7328
Epoch 9/10
50/50            0s 4ms/step -
accuracy: 0.7822 - loss: 0.6192 - val_accuracy: 0.4100 - val_loss: 0.7318
Epoch 10/10
50/50            0s 4ms/step -
accuracy: 0.7933 - loss: 0.6062 - val_accuracy: 0.4425 - val_loss: 0.7237
```

[22]:
```
#time to predict and see if it works as intended
preds_hidden = model_hidden.predict(grid_points)
preds_hidden_class = (preds_hidden > 0.5).astype(int).ravel()
```

```
1250/1250            1s 908us/step
```

[23]:
```
#time to visualise
plt.figure(figsize=(6,5))
plt.title("Model with 1 Hidden Layer (2 neurons)")
plt.scatter(grid_points[:,0], grid_points[:,1], c=preds_hidden_class, alpha=0.
 ↪3, s=10)
plt.plot(x_curve, np.cos(x_curve), 'k--', label='cos(x1)')
plt.plot(x_curve, 1 + np.cos(x_curve), 'r--', label='1 + cos(x1)')
plt.legend()
plt.xlabel('x1')
plt.ylabel('x2')
plt.show()
```



3 Part 2: MNIST Dataset and Study of the Hidden Layers

3.0.1 Loading,exploring, and preparing the data

For the second part, we are going to use the MNIST dataset, partly because you are already familiar with it, and partly because it comes with tensorflow, the most wideely used library for neural networks in python.

a. Load the MNIST dataset using `keras.datasets.mnist.load_data()` and split it into train and test.

b. Print the shapes of the training and testing data and labels.

c. Display the first image in the training set and its label.

d. Normalize the pixel values of the training and testing data to be between 0 and 1.

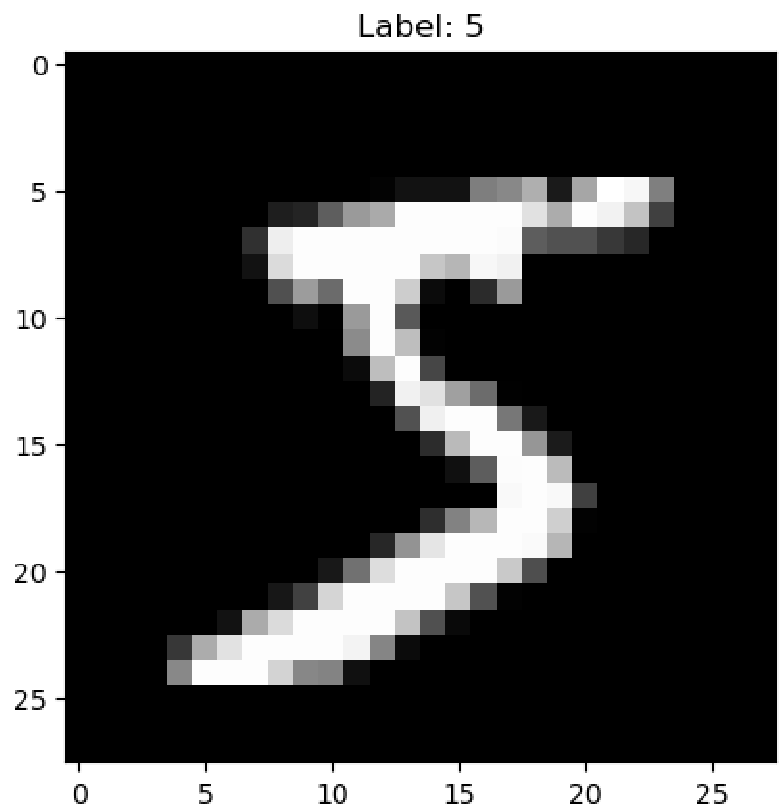
e. Convert the labels to one-hot encoded vectors using `keras.utils.to_categorical()`.

[24]:
```
#loading data set and doing the pre-reqs
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

print("Train images shape:", x_train.shape)
print("Train labels shape:", y_train.shape)
print("Test images shape:", x_test.shape)
print("Test labels shape:", y_test.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/mnist.npz
11490434/11490434            0s
0us/step
Train images shape: (60000, 28, 28)
Train labels shape: (60000,)
Test images shape: (10000, 28, 28)
Test labels shape: (10000,)
```

[25]:
```
plt.figure()
plt.imshow(x_train[0], cmap='gray')
plt.title(f"Label: {y_train[0]}")
plt.show()
```



[26]:
```
x_train = x_train.reshape(-1, 28*28).astype('float32') / 255.0
x_test  = x_test.reshape(-1, 28*28).astype('float32') / 255.0
```

[27]:
```
num_classes = 10
y_train_oh = keras.utils.to_categorical(y_train, num_classes)
y_test_oh  = keras.utils.to_categorical(y_test,  num_classes)
```

3.0.2 Exercise 4 (CORE): Building a Baseline Model

a. Build a sequential neural network model with one hidden layer of 128 neurons and an output layer with 10 neurons. Let's try using different activation fucntions (there is no real need to do this here, except learn how to implement it in keras). For example, use ReLU activation for the hidden layer and softmax for the output layer.

b. Compile the model using the Adam optimizer, categorical crossentropy loss, and accuracy metric You can do this using the parameters

''' model.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["accuracy"]) '''

c. Train the model for 10 epochs with a batch size of 128 and a 10% validation split.

[28]:
```python
model_mnist = keras.Sequential([
    layers.Dense(128, activation='relu', input_shape=(784,)),
    layers.Dense(10, activation='softmax')
])
```

[29]:
```python
model_mnist.compile(optimizer='adam',
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])
```

[30]:
```python
history_mnist = model_mnist.fit(
    x_train,
    y_train_oh,
    validation_split=0.1,
    epochs=10,
    batch_size=128,
    callbacks=[tensorboard_callback]
)
```

```
Epoch 1/10
422/422              3s 6ms/step -
accuracy: 0.8276 - loss: 0.6384 - val_accuracy: 0.9543 - val_loss: 0.1751
Epoch 2/10
422/422              2s 6ms/step -
accuracy: 0.9428 - loss: 0.1991 - val_accuracy: 0.9638 - val_loss: 0.1304
Epoch 3/10
422/422              2s 6ms/step -
accuracy: 0.9606 - loss: 0.1396 - val_accuracy: 0.9667 - val_loss: 0.1114
Epoch 4/10
422/422              2s 6ms/step -
accuracy: 0.9697 - loss: 0.1038 - val_accuracy: 0.9723 - val_loss: 0.0971
Epoch 5/10
422/422              2s 6ms/step -
accuracy: 0.9755 - loss: 0.0835 - val_accuracy: 0.9742 - val_loss: 0.0849
Epoch 6/10
422/422              2s 6ms/step -
accuracy: 0.9816 - loss: 0.0664 - val_accuracy: 0.9760 - val_loss: 0.0785
Epoch 7/10
422/422              2s 6ms/step -
accuracy: 0.9842 - loss: 0.0569 - val_accuracy: 0.9752 - val_loss: 0.0803
Epoch 8/10
422/422              3s 6ms/step -
accuracy: 0.9863 - loss: 0.0482 - val_accuracy: 0.9782 - val_loss: 0.0790
Epoch 9/10
422/422              2s 6ms/step -
accuracy: 0.9887 - loss: 0.0406 - val_accuracy: 0.9772 - val_loss: 0.0775
Epoch 10/10
422/422              3s 6ms/step -
```

```
accuracy: 0.9912 - loss: 0.0334 - val_accuracy: 0.9792 - val_loss: 0.0745
```

3.0.3 Exercise 5 (CORE): Exploring Network Depth

a. Evaluate the model on the test set and print the test loss and accuracy.

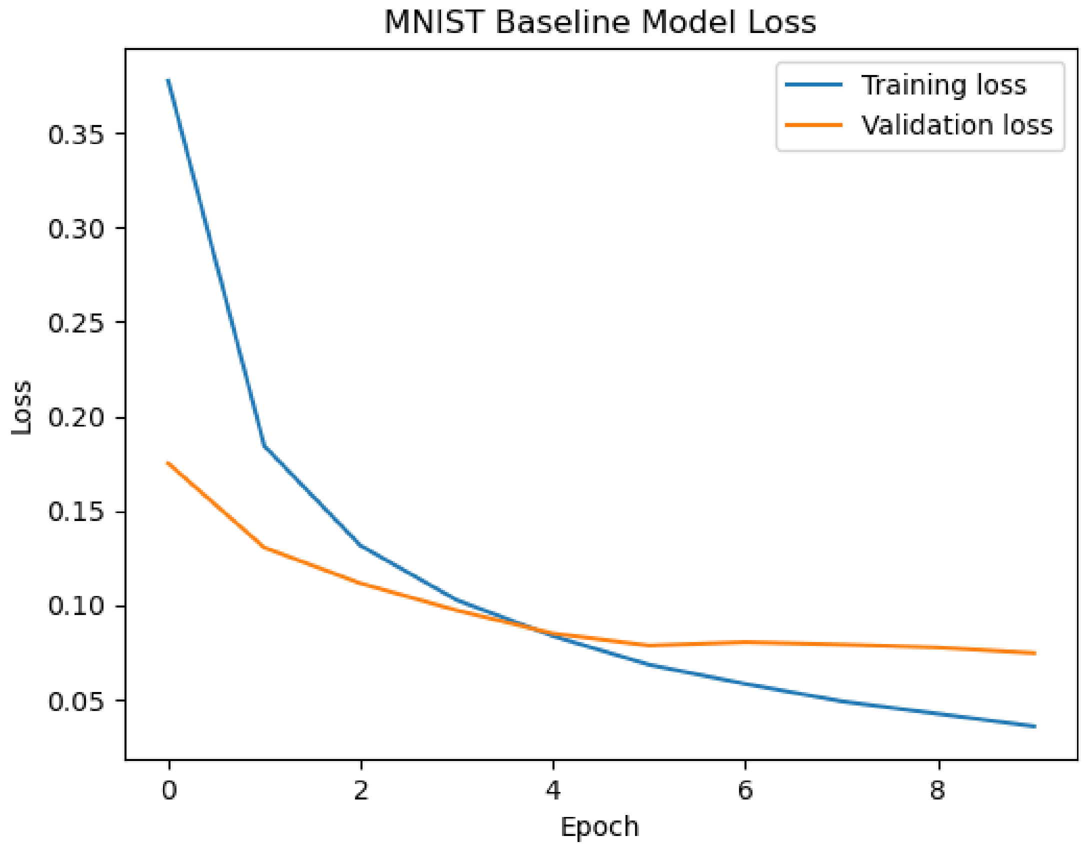
b. Plot the training and validation loss curves.

c. Generate and visualize a confusion matrix for the test set predictions.

[31]:
```
#model eval
test_loss, test_acc = model_mnist.evaluate(x_test, y_test_oh)
print("Test Loss:", test_loss)
print("Test Accuracy:", test_acc)
```

```
313/313              1s 2ms/step -
accuracy: 0.9727 - loss: 0.0878
Test Loss: 0.0775642916560173
Test Accuracy: 0.9761999845504761
```

[32]:
```
#training and validation loss curves
plt.figure()
plt.plot(history_mnist.history['loss'], label='Training loss')
plt.plot(history_mnist.history['val_loss'], label='Validation loss')
plt.title("MNIST Baseline Model Loss")
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



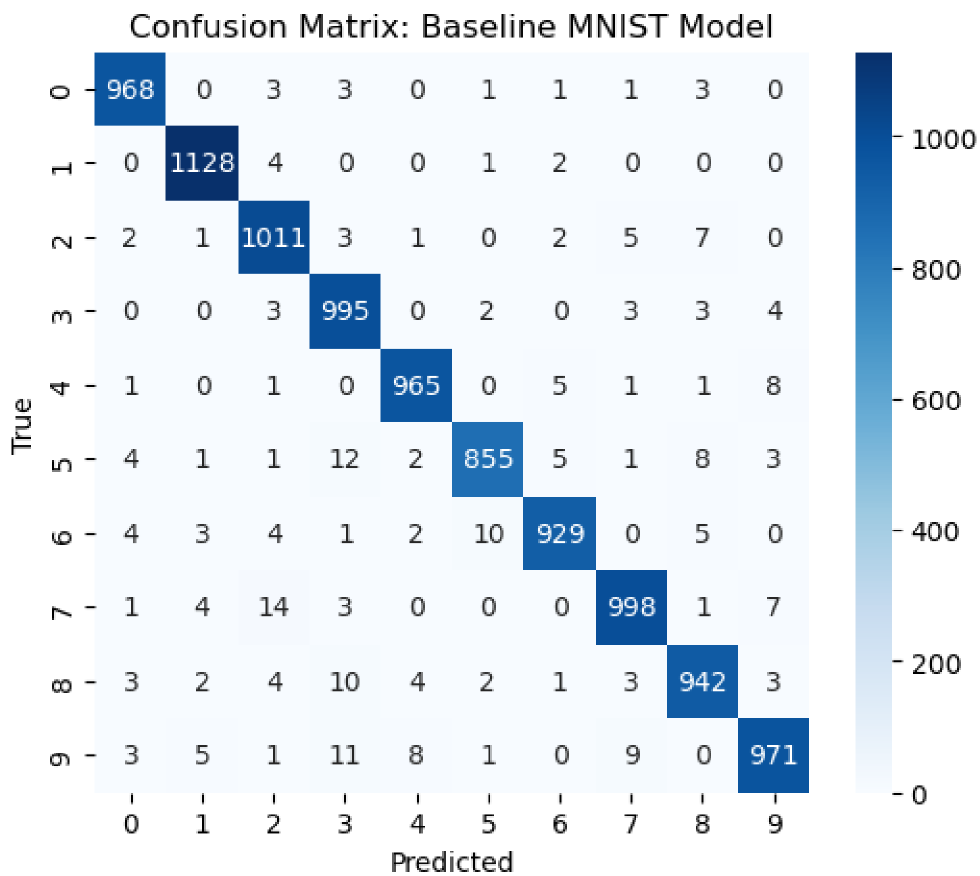
[33]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns

pred_mnist = model_mnist.predict(x_test)
pred_labels_mnist = np.argmax(pred_mnist, axis=1)
true_labels_mnist = np.argmax(y_test_oh, axis=1)

cm = confusion_matrix(true_labels_mnist, pred_labels_mnist)
```

```
313/313              1s 2ms/step
```

[34]:
```
plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title("Confusion Matrix: Baseline MNIST Model")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()
```



3.0.4 Exercise 6 (CORE): Exploring Network Depth

a. Build a deeper network with two hidden layers, each with 128 neurons and ReLU activation, and an output layer with 10 neurons and softmax activation.

b. Compile and train the model as in Exercise 1.

c. Evaluate the model on the test set and compare the results with the single-layer model.

d. Plot the training and validation loss curves for the deep network.

e. Generate and visualize a confusion matrix for the deep network's test set predictions.

[36]:
```
model_mnist_deep = keras.Sequential([
    layers.Dense(128, activation='relu', input_shape=(784,)),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

[37]:
```
model_mnist_deep.compile(optimizer='adam',
                         loss='categorical_crossentropy',
```

```
                                    metrics=['accuracy'])
```

[38]:
```
history_mnist_deep = model_mnist_deep.fit(
    x_train,
    y_train_oh,
    validation_split=0.1,
    epochs=10,
    batch_size=128,
    callbacks=[tensorboard_callback]
)
```

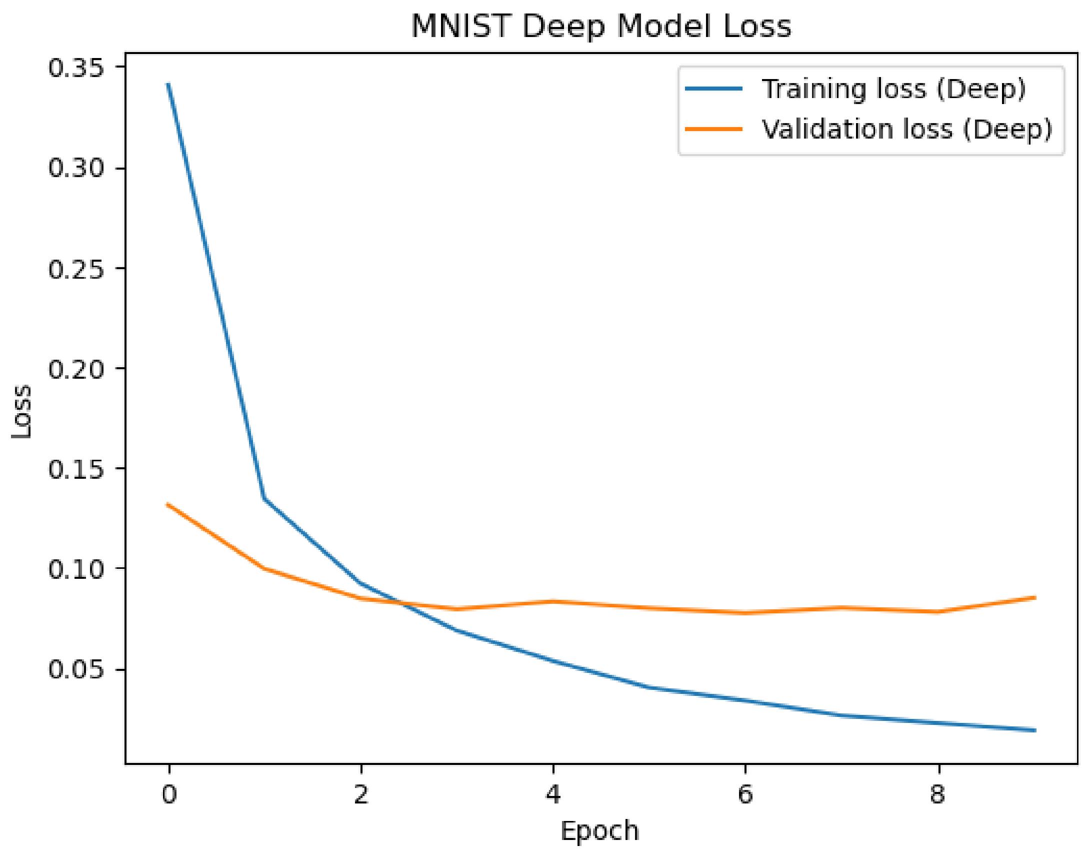
```
Epoch 1/10
422/422              4s 7ms/step -
accuracy: 0.8216 - loss: 0.6285 - val_accuracy: 0.9642 - val_loss: 0.1314
Epoch 2/10
422/422              3s 6ms/step -
accuracy: 0.9580 - loss: 0.1415 - val_accuracy: 0.9725 - val_loss: 0.0996
Epoch 3/10
422/422              3s 6ms/step -
accuracy: 0.9720 - loss: 0.0921 - val_accuracy: 0.9740 - val_loss: 0.0848
Epoch 4/10
422/422              3s 6ms/step -
accuracy: 0.9784 - loss: 0.0714 - val_accuracy: 0.9768 - val_loss: 0.0795
Epoch 5/10
422/422              3s 6ms/step -
accuracy: 0.9831 - loss: 0.0540 - val_accuracy: 0.9763 - val_loss: 0.0834
Epoch 6/10
422/422              3s 6ms/step -
accuracy: 0.9883 - loss: 0.0380 - val_accuracy: 0.9785 - val_loss: 0.0799
Epoch 7/10
422/422              3s 6ms/step -
accuracy: 0.9911 - loss: 0.0300 - val_accuracy: 0.9790 - val_loss: 0.0776
Epoch 8/10
422/422              3s 6ms/step -
accuracy: 0.9922 - loss: 0.0250 - val_accuracy: 0.9778 - val_loss: 0.0801
Epoch 9/10
422/422              3s 6ms/step -
accuracy: 0.9929 - loss: 0.0245 - val_accuracy: 0.9798 - val_loss: 0.0781
Epoch 10/10
422/422              3s 6ms/step -
accuracy: 0.9953 - loss: 0.0171 - val_accuracy: 0.9777 - val_loss: 0.0852
```

[39]:
```
#eval
test_loss_deep, test_acc_deep = model_mnist_deep.evaluate(x_test, y_test_oh)
print("Deep Model Test Loss:", test_loss_deep)
print("Deep Model Test Accuracy:", test_acc_deep)
```

```
313/313              1s 2ms/step -
accuracy: 0.9723 - loss: 0.0954
Deep Model Test Loss: 0.08118998259305954
Deep Model Test Accuracy: 0.9765999913215637
```

[40]:

```
#ploting and viz
plt.figure()
plt.plot(history_mnist_deep.history['loss'], label='Training loss (Deep)')
plt.plot(history_mnist_deep.history['val_loss'], label='Validation loss (Deep)')
plt.title("MNIST Deep Model Loss")
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

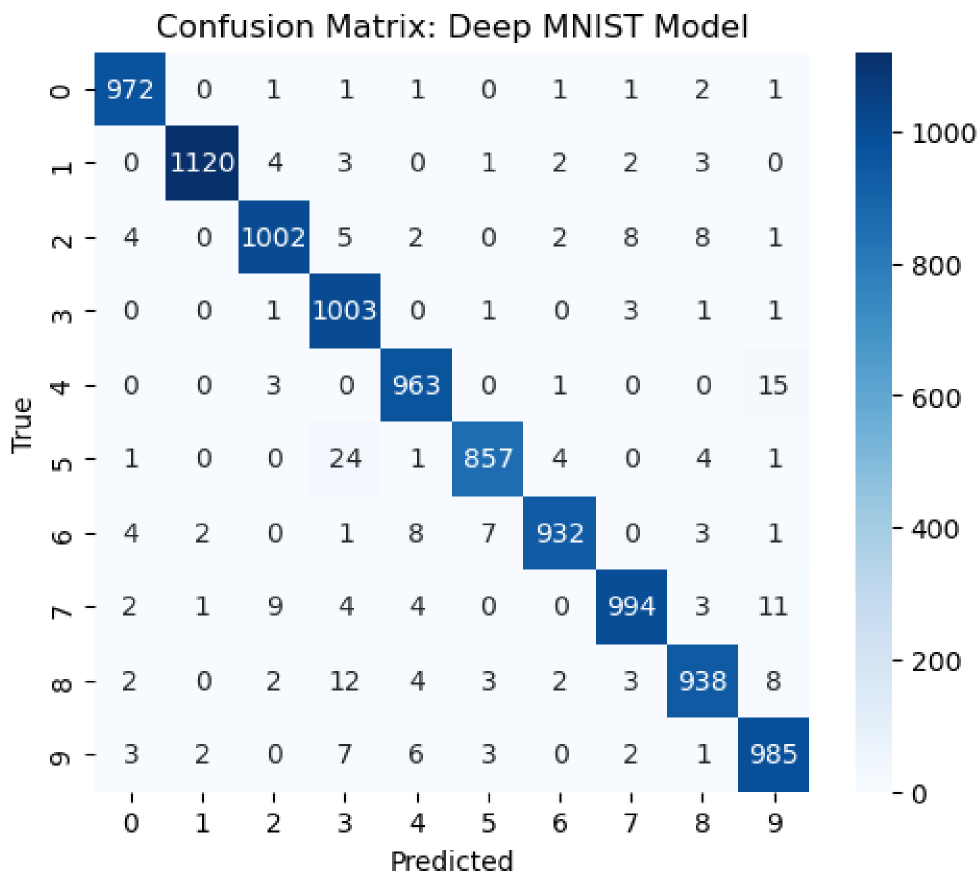


[41]:

```
#confus matrix
pred_mnist_deep = model_mnist_deep.predict(x_test)
pred_labels_mnist_deep = np.argmax(pred_mnist_deep, axis=1)
cm_deep = confusion_matrix(true_labels_mnist, pred_labels_mnist_deep)
```

```python
plt.figure(figsize=(6,5))
sns.heatmap(cm_deep, annot=True, fmt='d', cmap='Blues')
plt.title("Confusion Matrix: Deep MNIST Model")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()
```

```
313/313          1s 2ms/step
```



3.0.5 Exercise 3 (CORE): When to Stop Training?

Compare the training and validation loss curves for `model` (single hidden layer) and `model_deep` (two hidden layers).

a. In which scenario do you observe signs of overfitting? Explain your reasoning.

b. Based on these graphs, suggest a stopping criterion for training to prevent overfitting.

c. How does the depth of the network influence the point at which overfitting begins?

1. when we talk about overfitting it usually refers to the modelling learning the data a bit too well, when it happens the model fits perfectly to the data. This also happens when someone

accidentally uses the training data to train their model. In some cases the model learns the minute details and noises aswell. If this happens the model will fail when it deals with unknown data or essentially real world data. Here Overfitting appears when the training loss continues to decrease while the validation loss stops decreasing or even begins to rise, which can be observed in the deeper model as it shows a larger divergence between training and validation curves; hence we can say its overfitting.

2. It appears that we need to keep a track of the validation loss, since it seems to clearly affect the quality of our model, we can stop at earlier epochs espcially if the validation loss is not decreasing further or we can think that a point where the validatiion loss rather starts increasing that would the right time to stop or early stop our model.

3. It seems deeper graphs have bigger gap between validation and training losses and this can be considered as a sign of overfitting. While this cant be seen so strongly in the baseline model, hence they are less prone to overfitting. It seems that deeper graphs would do well with more complex dataset or functions.

3.0.6 Exercise 4 (CORE): Going Deep

Let's now validate the results in the previous question by increasing the number of hidden layers. We hope to see that the trends we observed when going from one to two hidden layers will be even more pronounced.

a. Build a neural network with 10 hidden layers, each with 128 neurons and ReLU activation, and an output layer with 10 neurons and softmax activation.

b. Compile and train the model for 20 epochs with a batch size of 128 and a 10% validation split.

c. Evaluate the model on the test set and plot the training and validation loss curves.

d. Discuss any challenges encountered during training and potential solutions.

[42]:
```
model_very_deep = keras.Sequential()

# First hidden layer + input
model_very_deep.add(layers.Dense(128, activation='relu', input_shape=(784,)))

# looping to add more layers
for _ in range(9):
    model_very_deep.add(layers.Dense(128, activation='relu'))

# our output layer
model_very_deep.add(layers.Dense(10, activation='softmax'))

model_very_deep.compile(optimizer='adam',
                        loss='categorical_crossentropy',
                        metrics=['accuracy'])

history_very_deep = model_very_deep.fit(
    x_train,
```

```
    y_train_oh,
    validation_split=0.1,
    epochs=20,
    batch_size=128,
    callbacks=[tensorboard_callback]
)
```

```
/opt/conda/lib/python3.11/site-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

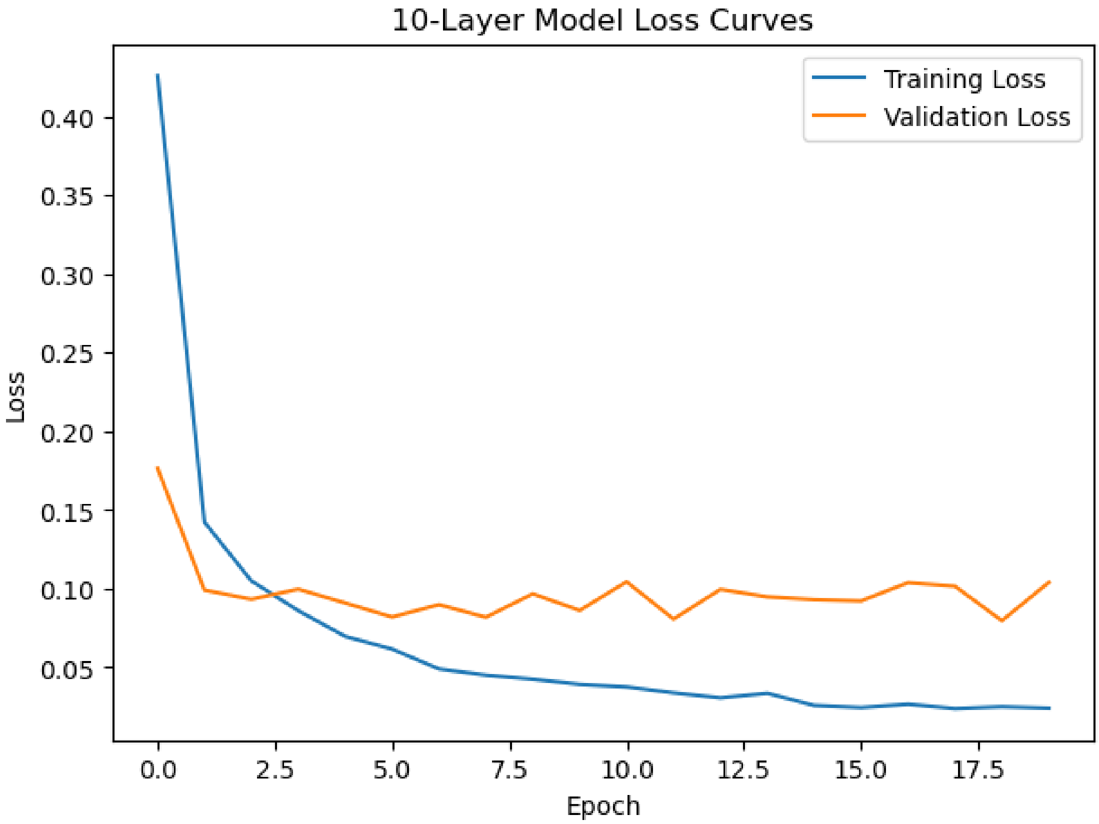
```
Epoch 1/20
422/422              7s 11ms/step -
accuracy: 0.7192 - loss: 0.8113 - val_accuracy: 0.9495 - val_loss: 0.1765
Epoch 2/20
422/422              4s 10ms/step -
accuracy: 0.9573 - loss: 0.1492 - val_accuracy: 0.9703 - val_loss: 0.0991
Epoch 3/20
422/422              4s 10ms/step -
accuracy: 0.9702 - loss: 0.1020 - val_accuracy: 0.9730 - val_loss: 0.0933
Epoch 4/20
422/422              4s 10ms/step -
accuracy: 0.9753 - loss: 0.0859 - val_accuracy: 0.9697 - val_loss: 0.0997
Epoch 5/20
422/422              4s 10ms/step -
accuracy: 0.9796 - loss: 0.0682 - val_accuracy: 0.9773 - val_loss: 0.0909
Epoch 6/20
422/422              4s 10ms/step -
accuracy: 0.9832 - loss: 0.0568 - val_accuracy: 0.9770 - val_loss: 0.0820
Epoch 7/20
422/422              4s 11ms/step -
accuracy: 0.9866 - loss: 0.0444 - val_accuracy: 0.9762 - val_loss: 0.0899
Epoch 8/20
422/422              4s 10ms/step -
accuracy: 0.9879 - loss: 0.0423 - val_accuracy: 0.9790 - val_loss: 0.0819
Epoch 9/20
422/422              4s 11ms/step -
accuracy: 0.9886 - loss: 0.0404 - val_accuracy: 0.9783 - val_loss: 0.0968
Epoch 10/20
422/422              5s 11ms/step -
accuracy: 0.9900 - loss: 0.0375 - val_accuracy: 0.9798 - val_loss: 0.0862
Epoch 11/20
422/422              4s 11ms/step -
accuracy: 0.9918 - loss: 0.0310 - val_accuracy: 0.9763 - val_loss: 0.1044
Epoch 12/20
422/422              4s 10ms/step -
```

```
accuracy: 0.9905 - loss: 0.0340 - val_accuracy: 0.9807 - val_loss: 0.0807
Epoch 13/20
422/422          4s 10ms/step -
accuracy: 0.9908 - loss: 0.0313 - val_accuracy: 0.9772 - val_loss: 0.0996
Epoch 14/20
422/422          4s 10ms/step -
accuracy: 0.9911 - loss: 0.0338 - val_accuracy: 0.9782 - val_loss: 0.0948
Epoch 15/20
422/422          4s 10ms/step -
accuracy: 0.9924 - loss: 0.0270 - val_accuracy: 0.9797 - val_loss: 0.0931
Epoch 16/20
422/422          4s 10ms/step -
accuracy: 0.9933 - loss: 0.0229 - val_accuracy: 0.9770 - val_loss: 0.0923
Epoch 17/20
422/422          4s 10ms/step -
accuracy: 0.9942 - loss: 0.0230 - val_accuracy: 0.9793 - val_loss: 0.1038
Epoch 18/20
422/422          4s 10ms/step -
accuracy: 0.9939 - loss: 0.0241 - val_accuracy: 0.9782 - val_loss: 0.1016
Epoch 19/20
422/422          4s 10ms/step -
accuracy: 0.9938 - loss: 0.0237 - val_accuracy: 0.9812 - val_loss: 0.0797
Epoch 20/20
422/422          4s 10ms/step -
accuracy: 0.9948 - loss: 0.0192 - val_accuracy: 0.9777 - val_loss: 0.1040
```

[43]:
```
test_loss_very_deep, test_acc_very_deep = model_very_deep.evaluate(x_test,␣
 ↪y_test_oh, verbose=0)
print("10-Layer Model Test Loss:", test_loss_very_deep)
print("10-Layer Model Test Accuracy:", test_acc_very_deep)

plt.figure(figsize=(7,5))
plt.plot(history_very_deep.history['loss'], label='Training Loss')
plt.plot(history_very_deep.history['val_loss'], label='Validation Loss')
plt.title("10-Layer Model Loss Curves")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

```
10-Layer Model Test Loss: 0.11166328191757202
10-Layer Model Test Accuracy: 0.978600025177002
```



It seems that such models are prone to overfitting, they are just memorising the data rather than generalising things, just before 5 epoch we see a spike which is still acceptable since the gap isnt that high, but as the epochs keep on increasing we can see the gap increase. Whats more it seems that the training loss stopped decreasing but rather started to increase after 10 epochs.

Furthermore, deeper models take quite a while to work. If the data was more complex or it had some other intricate details, it would take even longer. Another thing which should be noted is that deeper model means more number of hyperparameters to deal with.

Early stopping would have helped here.

3.0.7 Exercise (EXTRA): Regularisation Techniques (10 Layers Deep)

We have briefly touched on regularisation on Monday, which describes the process of removing complexity from an overfitting network. Here, let's try to implement dropout regularization, which is a technique that randomly ignores ("drops out") some layers when the network is overfitting. Look up the technique implementation before having a go. In keras, this is implemented using the "Dropout" method for the dropout layers, which accepts a parameter p between 0 and 1, and its effect is to randomly set input units for that layer to 0 with probability p at each step during training time.

a. Implement dropout regularization in the 10-layer deep network after each hidden layer with a dropout rate of $p = 0.2$.

b. Train the regularized model for 20 epochs and compare the training and validation loss curves with the original 10-layer deep model.

c. Discuss the impact of dropout regularization on the deep network's performance and generalization.

[ ]:
```

```

Discussion:

4 Competing the Worksheet

At this point you have hopefully been able to complete all the CORE exercises and attempted the EXTRA ones. Now is a good time to check the reproducibility of this document by restarting the notebook's kernel and rerunning all cells in order.

Before generating the PDF, please go to Edit -> Edit Notebook Metadata and change 'Student 1' and 'Student 2' in the **name** attribute to include your name. If you are unable to edit the Notebook Metadata, please add a Markdown cell at the top of the notebook with your name(s).

Once that is done and you are happy with everything, you can then run the following cell to generate your PDF. Once generated, please submit this PDF on Learn page by 16:00 PM on the Friday of the week the workshop was given.

[ ]:
```
!jupyter nbconvert --to pdf mlp_week09.ipynb
```

[ ]:
```

```