Weekly Assessment

Akash T
ADIT | NSTI CALICUT

AIM

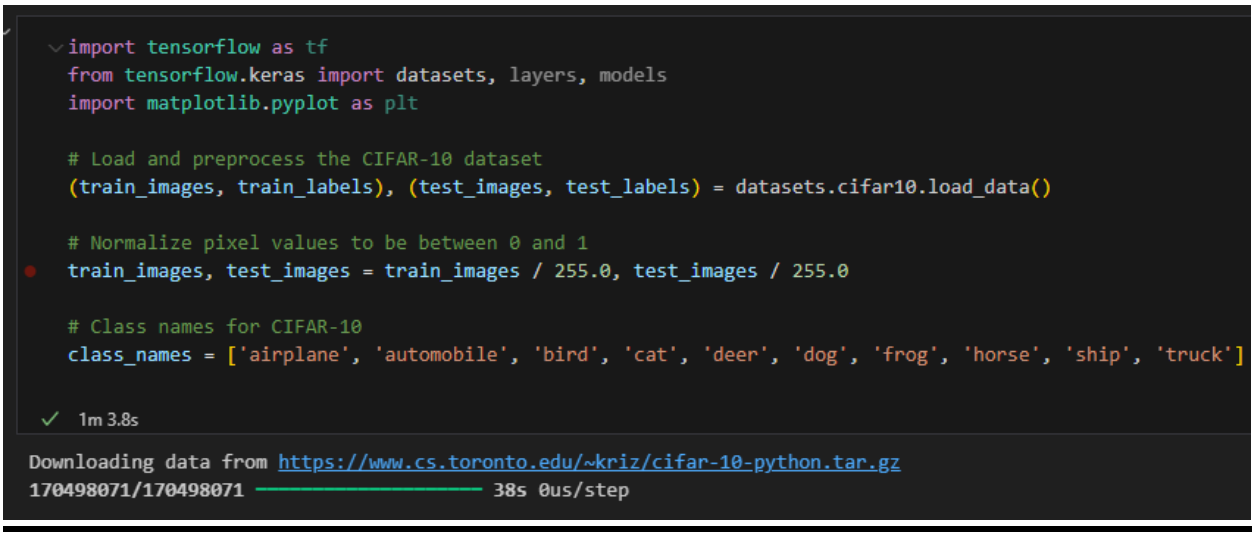
Using a deep learning framework of your choice (TensorFlow, PyTorch, etc.), implement a CNN to classify images from the CIFAR-10 dataset. Ensure your network includes convolutional layers, pooling layers, and fully connected layers. Evaluate the performance of your model and discuss any improvements you could make.

SOFTWARE/HARDWARE REQUIREMENTS

- Windows PC
- VS Code/Jupyter notebook

PROCEDURE

Step 1



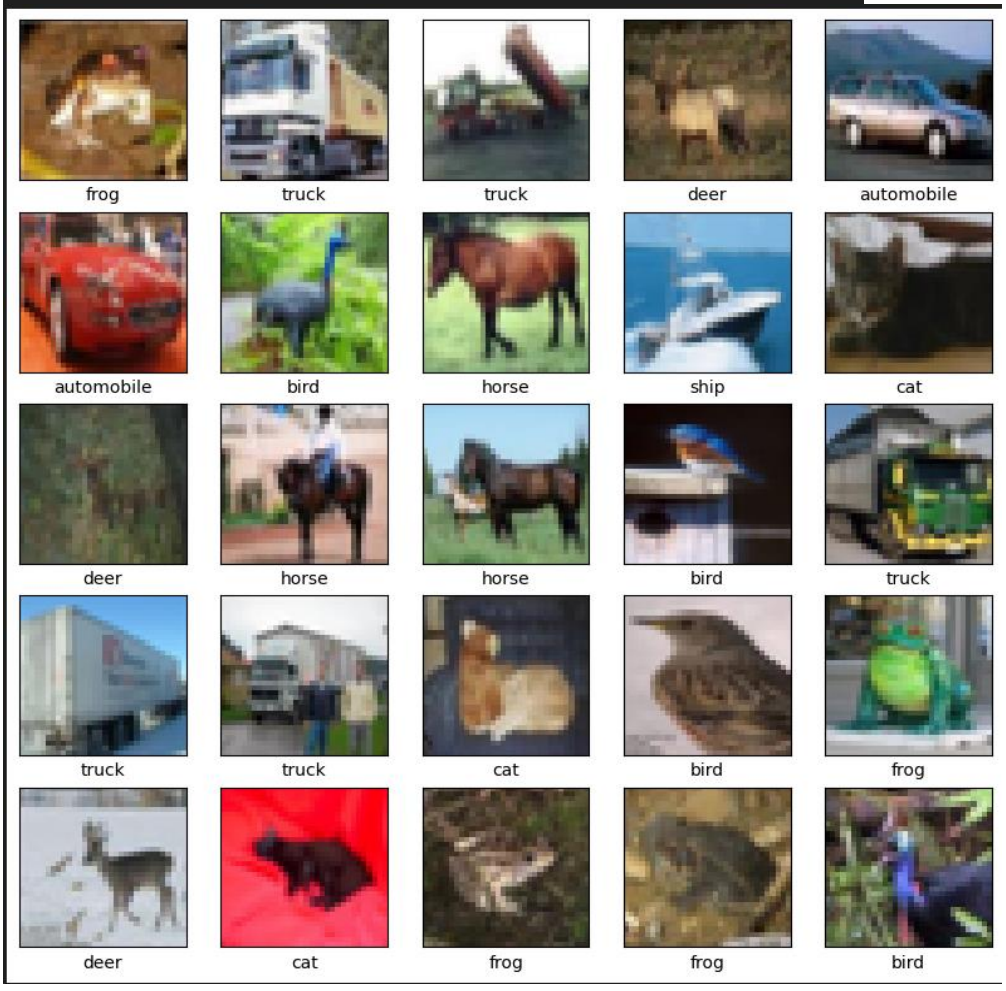
```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt

# Load and preprocess the CIFAR-10 dataset
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0

# Class names for CIFAR-10
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

✓ 1m 3.8s

Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 38s 0us/step
```

- tensorflow: A popular deep learning framework.
- datasets, layers, models: Submodules of Keras (which is integrated into TensorFlow), used for loading data, defining layers, and creating models.
- matplotlib.pyplot: A plotting library used for visualizing data.

Step 2

```
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i])
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```

✓ 1.2s



- datasets.cifar10.load_data(): Loads the CIFAR-10 dataset, which contains 60,000 32x32 color images in 10 classes, with 6,000 images per class.
- train_images, train_labels: Training data and labels.
- test_images, test_labels: Testing data and labels.
- train_images / 255.0, test_images / 255.0: Normalizes the image pixel values to be between 0 and 1 for faster convergence during training.
- class_names: List of class names corresponding to the CIFAR-10 dataset.

Step 3

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))

✓ 0.2s
```

- models.Sequential(): Initializes a sequential model.
- layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)): Adds a 2D convolutional layer with 32 filters, a 3x3 kernel size, ReLU activation, and an input shape of 32x32x3 (height, width, channels).
- layers.MaxPooling2D((2, 2)): Adds a max pooling layer with a 2x2 pool size to reduce the spatial dimensions of the feature maps.
- layers.Conv2D(64, (3, 3), activation='relu'): Adds another convolutional layer with 64 filters.
- layers.Flatten(): Flattens the 3D feature maps to 1D feature vectors.
- layers.Dense(64, activation='relu'): Adds a fully connected (dense) layer with 64 units and ReLU activation.
- layers.Dense(10): Adds the output layer with 10 units (one for each class).

Step 4

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

✓ 0.0s

- model.compile(): Configures the model for training.
- optimizer='adam': Uses the Adam optimization algorithm.
- loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True): Uses sparse categorical crossentropy loss (from_logits=True indicates that the output values are not normalized).
- metrics=['accuracy']: Monitors accuracy during training and evaluation.

Step 5

```
history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
```

✓ 11m 12.6s

```
Epoch 1/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 71s 44ms/step - accuracy: 0.3372 - loss: 1.7939 - val_accuracy: 0.5596 - val_loss: 1.2483
Epoch 2/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 69s 44ms/step - accuracy: 0.5606 - loss: 1.2349 - val_accuracy: 0.5790 - val_loss: 1.1918
Epoch 3/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 75s 39ms/step - accuracy: 0.6274 - loss: 1.0592 - val_accuracy: 0.6130 - val_loss: 1.0959
Epoch 4/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 60s 38ms/step - accuracy: 0.6658 - loss: 0.9509 - val_accuracy: 0.6583 - val_loss: 0.9637
Epoch 5/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 83s 39ms/step - accuracy: 0.6908 - loss: 0.8703 - val_accuracy: 0.6790 - val_loss: 0.9337
Epoch 6/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 61s 39ms/step - accuracy: 0.7165 - loss: 0.8116 - val_accuracy: 0.6887 - val_loss: 0.9086
Epoch 7/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 60s 38ms/step - accuracy: 0.7366 - loss: 0.7553 - val_accuracy: 0.7019 - val_loss: 0.8555
Epoch 8/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 72s 46ms/step - accuracy: 0.7506 - loss: 0.7142 - val_accuracy: 0.7091 - val_loss: 0.8395
Epoch 9/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 63s 41ms/step - accuracy: 0.7642 - loss: 0.6735 - val_accuracy: 0.7037 - val_loss: 0.8533
Epoch 10/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 57s 37ms/step - accuracy: 0.7758 - loss: 0.6393 - val_accuracy: 0.7067 - val_loss: 0.8627
```

- model.fit(): Trains the model on the training data.

- epochs=10: Specifies the number of epochs (full passes through the training dataset).
- validation_data=(test_images, test_labels): Uses the test data for validation during training.

Step 6

```
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print(f'Test accuracy: {test_acc}')
```

✓ 4.1s

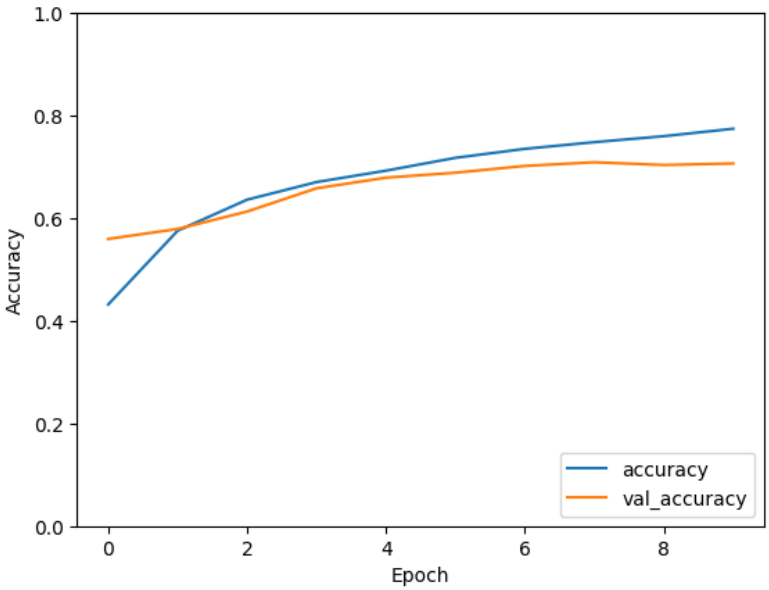
```
313/313 - 4s - 12ms/step - accuracy: 0.7067 - loss: 0.8627
Test accuracy: 0.7067000269889832
```

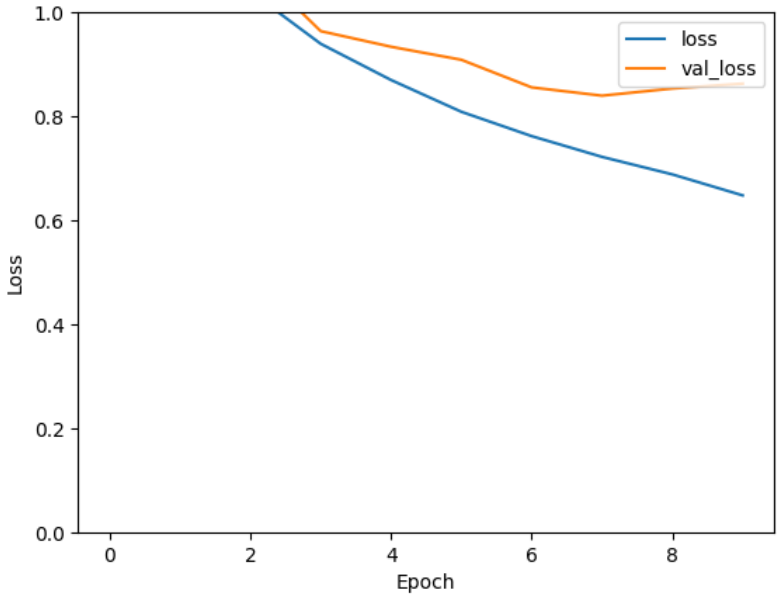
- model.evaluate(): Evaluates the model on the test data.
- verbose=2: Prints the evaluation progress.
- test_acc: Stores the test accuracy.

Step 7

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.legend(loc='lower right')
plt.show()

plt.plot(history.history['loss'], label='loss')
plt.plot(history.history['val_loss'], label = 'val_loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.ylim([0, 1])
plt.legend(loc='upper right')
plt.show()
```

✓ 0.3s





- history.history['accuracy']: Training accuracy for each epoch.
- history.history['val_accuracy']: Validation accuracy for each epoch.
- plt.plot(): Plots the training and validation accuracy.
- plt.xlabel('Epoch'), plt.ylabel('Accuracy'): Labels the x and y axes.
- plt.ylim([0, 1]): Sets the y-axis limits.
- plt.legend(loc='lower right'): Adds a legend in the lower right corner.
- plt.show(): Displays the plot.
- Similar plotting is done for the training and validation loss.

Possible Improvements

- Deeper Network: Adding more layers to capture more complex features.
- Data Augmentation: Increases the diversity of the training data to improve generalization.
- Dropout Layers: Prevents overfitting by randomly dropping units during training.

AIM

Construct a feedforward neural network to predict housing prices based on the provided
dataset. Include input normalization, hidden layers with appropriate activation
functions, and an output layer. Train the network using backpropagation and evaluate its
performance using Mean Squared Error (MSE).

```
Bedrooms,Bathrooms,SquareFootage,Location,Age,Price
3,2,1500,Urban,10,300000
4,3,2000,Suburban,5,400000
2,1,800,Rural,20,150000
3,2,1600,Urban,12,310000
4,3,2200,Suburban,8,420000
2,1,900,Rural,25,160000
5,4,3000,Urban,3,600000
3,2,1400,Suburban,15,290000
3,2,1300,Rural,30,180000
4,3,2500,Urban,7,500000
```

You can copy this data into a CSV file named housing_prices.csv.

SOFTWARE/HARDWARE REQUIREMENTS

- Windows PC
- VS Code/Jupyter notebook

PROCEDURE

Step 1

```python
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

# Load the dataset
data = pd.read_csv('housing_prices.csv')

# Check the data
print(data.head())

# Preprocess the data
X = data.drop('Price', axis=1)
y = data['Price']

# One-hot encode the categorical 'Location' column and standardize numerical columns
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), ['Bedrooms', 'Bathrooms', 'SquareFootage', 'Age']),
        ('cat', OneHotEncoder(), ['Location'])
    ])

X_preprocessed = preprocessor.fit_transform(X)

# Normalize the target variable
y = y / 1e5  # Scaling down the price for better model performance

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_preprocessed, y, test_size=0.2, random_state=42)
```

✓ 0.0s

	Bedrooms	Bathrooms	SquareFootage	Location	Age	Price
0	3	2	1500	Urban	10	300000
1	4	3	2000	Suburban	5	400000
2	2	1	800	Rural	20	150000
3	3	2	1600	Urban	12	310000
4	4	3	2200	Suburban	8	420000

- Library Imports: We import the necessary libraries for data manipulation, model building, and preprocessing.
- Load the Dataset: We load the dataset from the CSV file using pd.read_csv('housing_prices.csv').
- Check the Data: Print the first few rows of the dataset to ensure it is loaded correctly.

Preprocess the Data:

- We separate the features (X) from the target variable (y).
- We use ColumnTransformer to preprocess the data:
- StandardScaler standardizes the numerical columns (Bedrooms, Bathrooms, SquareFootage, Age).
- OneHotEncoder encodes the categorical 'Location' column.
- We normalize the target variable (price) by dividing it by 100,000 to bring it to a similar scale as the input features.

- Split the Data: We split the data into training and testing sets using train_test_split.

Step 2

```
# Build the feedforward neural network model
model = models.Sequential()
model.add(layers.Dense(128, activation='relu', input_shape=(X_train.shape[1],)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(1))  # Output layer for regression (predicting prices)

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

✓ 0.0s
```

Define the Model:

- models.Sequential(): Initializes a sequential model.
- layers.Dense(128, activation='relu', input_shape=(X_train.shape[1],)): Adds a dense layer with 128 units and ReLU activation. The input_shape is set to the number of features in the training set.

- layers.Dense(64, activation='relu'): Adds a dense layer with 64 units and ReLU activation.
- layers.Dense(32, activation='relu'): Adds a dense layer with 32 units and ReLU activation.
- layers.Dense(1): Adds the output layer with a single unit to predict the price.

Compile the Model:

- model.compile(optimizer='adam', loss='mean_squared_error'): Compiles the model with the Adam optimizer and Mean Squared Error (MSE) loss function.

Step 3

```
# Train the model
history = model.fit(X_train, y_train, epochs=200, validation_split=0.2, batch_size=8)
```

✓ 27.6s

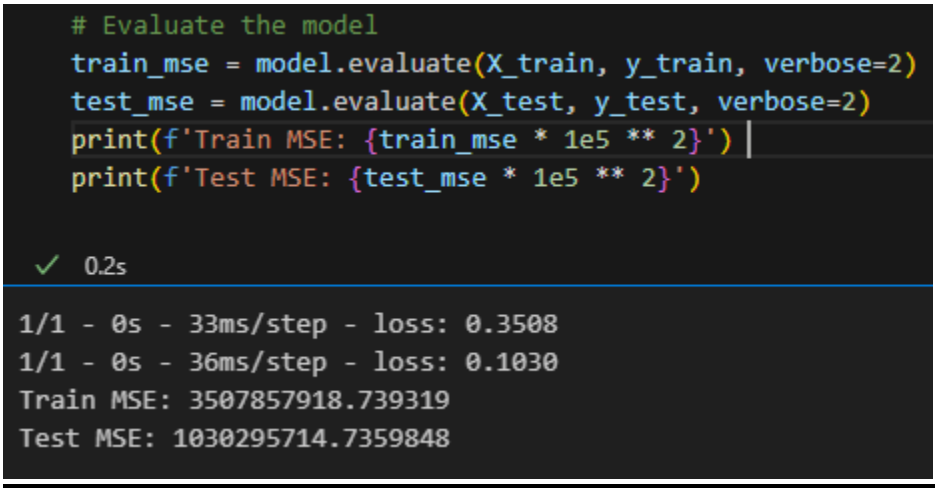
```
Epoch 1/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step - loss: 11.7793 - val_loss: 23.4476
Epoch 2/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 81ms/step - loss: 11.2251 - val_loss: 22.5869
Epoch 3/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 78ms/step - loss: 10.7179 - val_loss: 21.7326
Epoch 4/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 86ms/step - loss: 10.2978 - val_loss: 20.9692
Epoch 5/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 147ms/step - loss: 9.9498 - val_loss: 20.2661
Epoch 6/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 85ms/step - loss: 9.6451 - val_loss: 19.5916
Epoch 7/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 80ms/step - loss: 9.3703 - val_loss: 18.9222
Epoch 8/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 189ms/step - loss: 9.0976 - val_loss: 18.2944
Epoch 9/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 80ms/step - loss: 8.8299 - val_loss: 17.6748
Epoch 10/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 78ms/step - loss: 8.5722 - val_loss: 17.0435
Epoch 11/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 86ms/step - loss: 8.3168 - val_loss: 16.4184
Epoch 12/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 79ms/step - loss: 8.0526 - val_loss: 15.7909
Epoch 13/200
...
Epoch 199/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 83ms/step - loss: 1.9290e-04 - val_loss: 1.4028
Epoch 200/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 166ms/step - loss: 1.8868e-04 - val_loss: 1.4026
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

Train the Model:

- model.fit(X_train, y_train, epochs=200, validation_split=0.2, batch_size=8): Trains the model on the training data.
    - epochs=200: Specifies the number of epochs (full passes through the training dataset).
    - validation_split=0.2: Uses 20% of the training data for validation.
    - batch_size=8: Sets the number of samples per gradient update.

Step 4



Evaluate the Model:

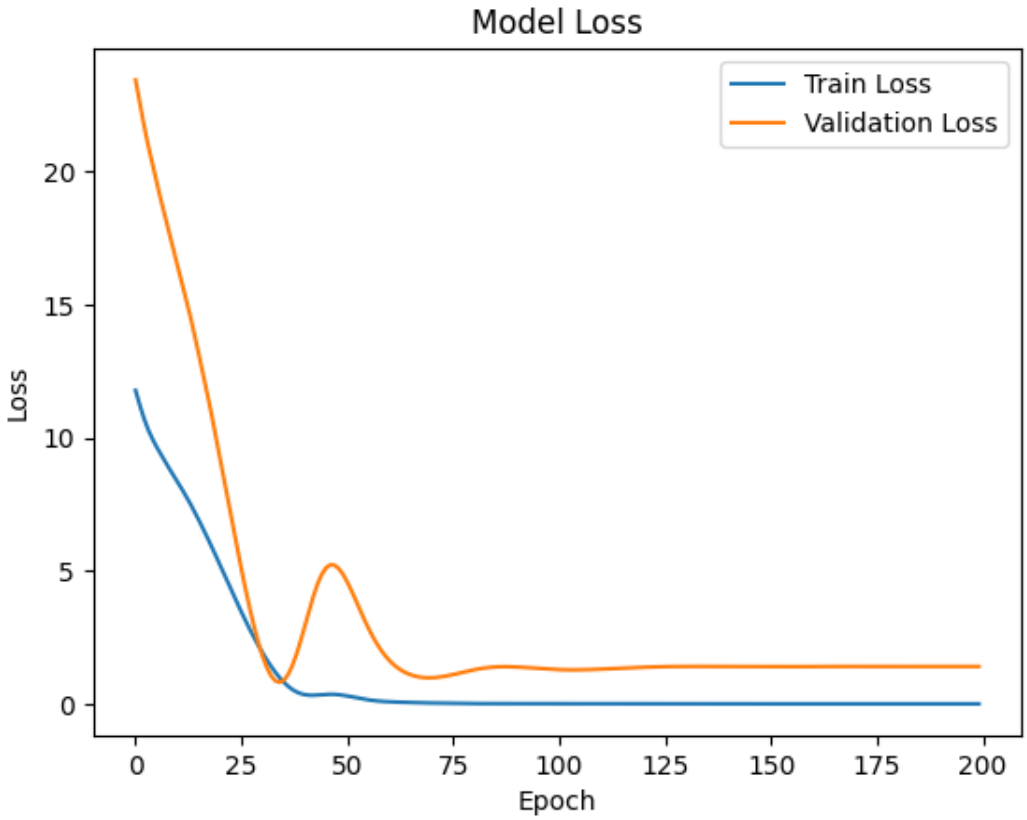
- model.evaluate(X_train, y_train, verbose=2): Evaluates the model on the training data.
- model.evaluate(X_test, y_test, verbose=2): Evaluates the model on the testing data.
- print(f'Train MSE: {train_mse * 1e5 ** 2}'): Prints the training MSE, converting it back to the original scale by multiplying by 100,000 squared.
- print(f'Test MSE: {test_mse * 1e5 ** 2}'): Prints the testing MSE, converting it back to the original scale.

Step 5

```
import matplotlib.pyplot as plt

# Plot training & validation loss values
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('
plt.ylabel(
plt.legend(loc='upper right')
plt.show()
```

(function) loc: Literal['upper right']

0.2s



Plot Training History:

- plt.plot(history.history['loss'], label='Train Loss'): Plots the training loss over epochs.
- plt.plot(history.history['val_loss'], label='Validation Loss'): Plots the validation loss over epochs.
- plt.title('Model Loss'): Sets the title of the plot.
- plt.xlabel('Epoch'), plt.ylabel('Loss'): Labels the x and y axes.

- plt.legend(loc='upper right'): Adds a legend in the upper right corner.
- plt.show(): Displays the plot.

***** The error here is high because the amount of data is very small