Keras

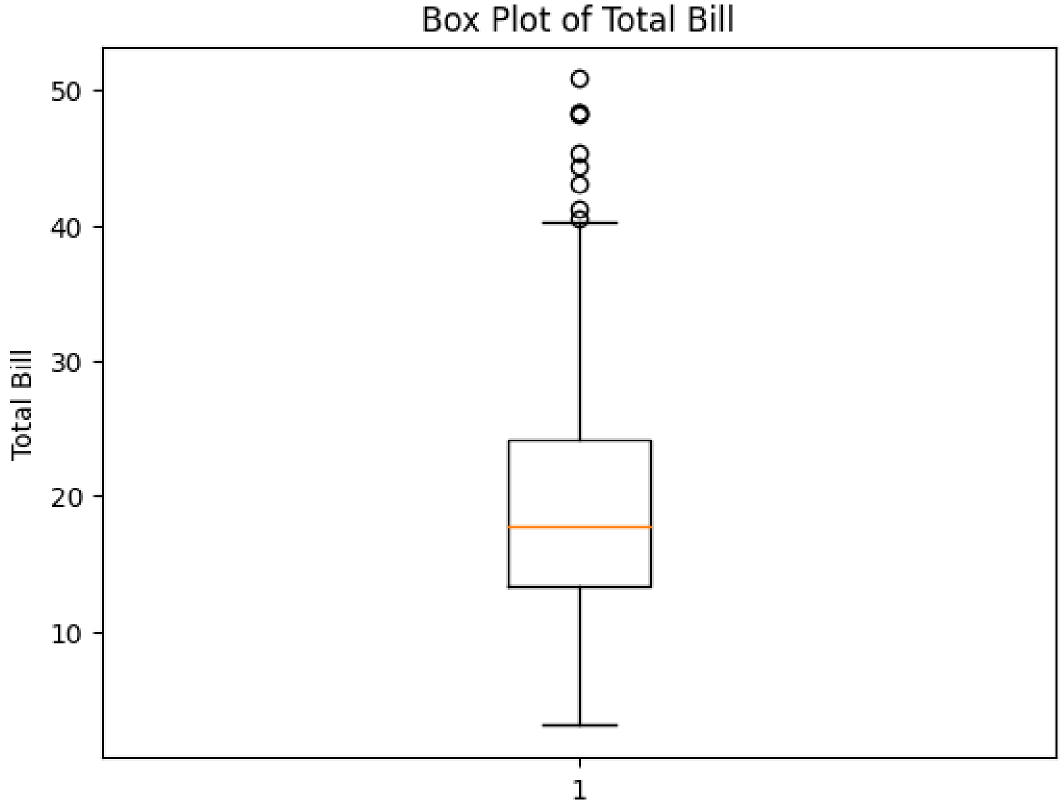
BOXPLOTIQRZSCORE

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
data = pd.read_excel('tips.xlsx')
def detect_outliers(data_column, column_name):
    data_values = data[data_column].values
    mean = np.mean(data_values)
    stddev = np.std(data_values)
    z_scores = (data_values - mean) / stddev
    # Set Z-score threshold for outliers
    z_threshold = 2
    # Calculate IQR and bounds for the IQR method
    q1 = np.percentile(data_values, 25)
    q3 = np.percentile(data_values, 75)
    iqr = q3 - q1
    iqr_lower_bound = q1 - 1.5 * iqr
    iqr_upper_bound = q3 + 1.5 * iqr
    # Identify outliers using both methods
    z_outliers = np.where(np.abs(z_scores) > z_threshold)[0]
    iqr_outliers = np.where((data_values < iqr_lower_bound) | (data_values > iqr_upper_bound))[0]
    # Print the outliers
    print(f"\n=== {column_name} ===")
    print("Outliers identified using Z-scores:", data_values[z_outliers])
    print("Outliers identified using IQR method:", data_values[iqr_outliers])
    print("Mean:", mean)
    print("Standard Deviation:", stddev)
    print("Z-scores:", z_scores
    plt.boxplot(data_values)
    plt.title(f"Box Plot of {column_name}")
```

```
    plt.ylabel(column_name)
    plt.show()
 detect_outliers('total_bill', 'Total Bill')
=== Total Bill ===
Outliers identified using Z-scores: [39.42 38.01 48.27 40.17 44.3  38.07 41.19 48.17 50.81 45.35
40.55 43.11 38.73 48.33]
Outliers identified using IQR method: [48.27 44.3  41.19 48.17 50.81 45.35 40.55 43.11 48.33]
Mean: 19.78594262295082
Standard Deviation: 8.884150577771132
 Z-scores: [-3.14711305e-01 -1.06323531e+00  1.37779900e-01  4.38315103e-01
  5.40744704e-01  6.19536705e-01 -1.23995452e+00  7.98507107e-01
 -5.34203307e-01 -5.63468908e-01 -1.07111451e+00  1.74175992e+00
 -4.91430507e-01 -1.52624903e-01 -5.57840908e-01  2.01939101e-01
 -1.06436091e+00 -3.93503306e-01 -3.16962505e-01  9.72582994e-02
 -2.10030504e-01  5.67366990e-02 -4.52034507e-01  2.21000952e+00
  3.83349840e-03 -2.22412104e-01 -7.22178510e-01 -7.98719310e-01
  2.15446301e-01 -1.53017018e-02 -1.15215771e+00 -1.61629703e-01
 -5.31952107e-01  1.01760699e-01 -2.25788904e-01  4.81087904e-01
 -3.91252106e-01 -3.21464905e-01 -1.23359303e-01  1.29264551e+00
 -4.21643306e-01 -2.61808105e-01 -6.58019309e-01 -1.13752491e+00
  1.19471831e+00 -1.68383303e-01  2.75103101e-01  1.41983831e+00
  9.86482309e-01 -1.96523304e-01 -8.15603311e-01 -1.06886331e+00
  1.69110792e+00 -1.10825931e+00  6.49927905e-01 -3.33113020e-02
  2.05129992e+00  7.45603907e-01 -9.61931312e-01  3.20616553e+00
  5.67366990e-02 -6.72652109e-01 -9.86694512e-01 -1.68383303e-01
 -2.47175304e-01  3.30990987e-02 -3.75493706e-01 -1.88154652e+00
  4.99830989e-02 -5.37580108e-01 -8.74134511e-01 -3.05706505e-01
  7.96255907e-01  6.18411105e-01 -5.69096908e-01 -1.04410011e+00
 -2.10030504e-01  8.34526308e-01  3.34759902e-01 -2.80943305e-01
 -3.89393021e-02 -3.51856105e-01 -1.09362651e+00  1.45135511e+00
 -4.28396906e-01  1.69335912e+00 -7.60448910e-01 -1.69508903e-01
  5.54251904e-01  1.54663900e-01  1.03375751e+00  3.04368702e-01
 -1.57988572e+00 -3.90126506e-01  3.33634302e-01  2.29442952e+00
  8.43531108e-01 -8.73008911e-01  1.37779900e-01 -8.24608111e-01
```

| | | | |
|---|---|---|---|
| -9.49549712e-01 | -4.95932907e-01 | 2.75930233e+00 | 2.96489502e-01 |
| 1.27649500e-01 | -4.98184107e-01 | 7.92486992e-02 | 6.10531905e-01 |
| -1.74011304e-01 | -6.16372108e-01 | -6.51265709e-01 | -1.41104572e+00 |
| 2.05805352e+00 | 4.68706304e-01 | 6.66811906e-01 | -2.78692105e-01 |
| 1.14181511e+00 | -1.02834171e+00 | -8.27984911e-01 | 4.83339104e-01 |
| -9.11279312e-01 | -7.16550509e-01 | -6.22000108e-01 | -4.31773706e-01 |
| -8.22356911e-01 | 1.12718231e+00 | -1.26809452e+00 | -5.92734508e-01 |
| -9.46172912e-01 | 3.41513502e-01 | -7.94609025e-02 | 5.44854990e-02 |
| -9.69810512e-01 | -8.47120111e-01 | -1.71760104e-01 | -1.26922012e+00 |
| -1.06436091e+00 | -6.34381709e-01 | -4.26145706e-01 | -7.45816110e-01 |
| -2.60682504e-01 | 1.63370232e+00 | 2.40924072e+00 | 8.17642307e-01 |
| -3.77744906e-01 | -1.28722972e+00 | -1.28987303e-01 | -8.91018511e-01 |
| -1.12626891e+00 | -1.38178012e+00 | -6.43386509e-01 | -7.49192910e-01 |
| -2.84320105e-01 | 5.36242304e-01 | -1.79450166e-03 | 1.13281031e+00 |
| 3.19490953e+00 | 5.86894305e-01 | -7.19927310e-01 | -3.70991306e-01 |
| 1.92934300e-01 | -8.02096110e-01 | -4.02508106e-01 | -6.72652109e-01 |
| -2.56180104e-01 | 5.32865504e-01 | 1.09639900e-01 | 1.34217191e+00 |
| -1.03509531e+00 | -1.03059291e+00 | 3.49206794e+00 | -4.47532107e-01 |
| -1.41104572e+00 | 1.35793031e+00 | -3.33846505e-01 | 1.47611831e+00 |
| -2.13407304e-01 | -5.97236908e-01 | -1.14652971e+00 | 1.67084712e+00 |
| 1.67309832e+00 | 3.98919103e-01 | 2.87749033e+00 | 3.80909503e-01 |
| 2.33720232e+00 | 1.01760699e-01 | 1.25398300e-01 | 1.20147191e+00 |
| -1.84141704e-01 | 3.73030302e-01 | -4.61039307e-01 | 2.70789839e-03 |
| 9.74100709e-01 | -4.84676907e-01 | -3.60860906e-01 | -1.37615212e+00 |
| -1.06323531e+00 | 2.62535593e+00 | -7.63825710e-01 | -7.06420109e-01 |
| -1.21108103e-01 | -7.93091310e-01 | -7.63825710e-01 | -3.81121706e-01 |
| 8.37510993e-02 | -3.73242506e-01 | 7.65864707e-01 | 2.13234312e+00 |
| 5.04725504e-01 | -7.90840110e-01 | 1.15644791e+00 | 6.87072706e-01 |
| 3.21291913e+00 | -7.33434510e-01 | 9.43709509e-01 | -7.75081710e-01 |
| 9.41458309e-01 | -9.22535312e-01 | -1.35589132e+00 | 1.16545271e+00 |
| -8.58376111e-01 | -7.16550509e-01 | -1.26134092e+00 | -4.28396906e-01 |
| -7.16550509e-01 | -3.95754506e-01 | -1.09137531e+00 | 7.47462992e-02 |
| -7.32308910e-01 | 2.62721501e-01 | 4.75459904e-01 | -4.61039307e-01 |
| -9.20284112e-01 | -1.01483451e+00 | -4.79048907e-01 | -1.09362651e+00 |

-8.08849711e-01  1.46823911e+00  1.80591912e+00  1.04051111e+00
 8.32275107e-01  3.24629502e-01 -2.21286504e-01 -1.13228903e-01]



CIFAR-10 Classification

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.activations import relu
import numpy as np
import matplotlib.pyplot as plt
# Load and preprocess the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0  # Normalize the images
y_train, y_test = to_categorical(y_train), to_categorical(y_test)  # Convert labels to categorical
# Define the model creation function
def create_model(hidden_units=None):
    model = models.Sequential([
        layers.Flatten(input_shape=(32, 32, 3)),
        layers.Dense(hidden_units[0], activation=relu),
        layers.Dense(hidden_units[1], activation=relu),
        layers.Dense(hidden_units[2], activation=relu),
        layers.Dense(10, activation='softmax')  # Output layer
    ])
    return model
# Initialize results dictionary and counter
results_dict = {}
counter = 1
# Create, compile, and train the model
model = create_model(hidden_units=[512, 256, 128])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
history = model.fit(x_train, y_train, epochs=5, batch_size=64, validation_split=0.2)
# Evaluate the model
```

```
test_loss, test_acc = model.evaluate(x_test, y_test)
model_info = f"Test accuracy: {round(test_acc * 100, 4)}%"
results_dict[counter] = model_info
counter += 1
# Print results
for key, value in results_dict.items():
    print(f"Run {key}: {value}")
# Function to plot the probability meter
def plot_probability_meter(predictions, image):
    class_labels = ["airplane", "automobile", "bird", "cat", "deer", "dog", "frog", "horse", "ship",
"truck"]
    fig, axs = plt.subplots(1, 2, figsize=(10, 2))
    # Plot the image
    axs[0].imshow(image)
    axs[0].axis('off')
    # Plot the prediction probabilities
    axs[1].barh(class_labels, predictions, color='skyblue')
    axs[1].set_xlim(0, 1)
    plt.tight_layout()
    plt.show()
# Select a few sample images and make predictions
num_images = 3
sample_images = x_train[:num_images]
predictions = model.predict(sample_images)
# Plot the predictions for the sample images
for i in range(num_images):
    plot_probability_meter(predictions[i], sample_images[i])
```

Epoch 1/5

**625/625** ━━━━━━━━━━━━━━━━━━━━ **7s** 5ms/step - accuracy: 0.2573 - loss: 2.0378 - val_accuracy: 0.3357 - val_loss: 1.8555

Epoch 2/5

**625/625** ━━━━━━━━━━━━━━━━━━━━ **2s** 3ms/step - accuracy: 0.3735 - loss: 1.7315 - val_accuracy: 0.3871 - val_loss: 1.6978

Epoch 3/5

**625/625** ━━━━━━━━━━━━━━━━━━━━ **3s** 4ms/step - accuracy: 0.4152 - loss: 1.6313 - val_accuracy: 0.4142 - val_loss: 1.6401

Epoch 4/5

**625/625** ━━━━━━━━━━━━━━━━━━━━ **2s** 3ms/step - accuracy: 0.4375 - loss: 1.5709 - val_accuracy: 0.4320 - val_loss: 1.6006

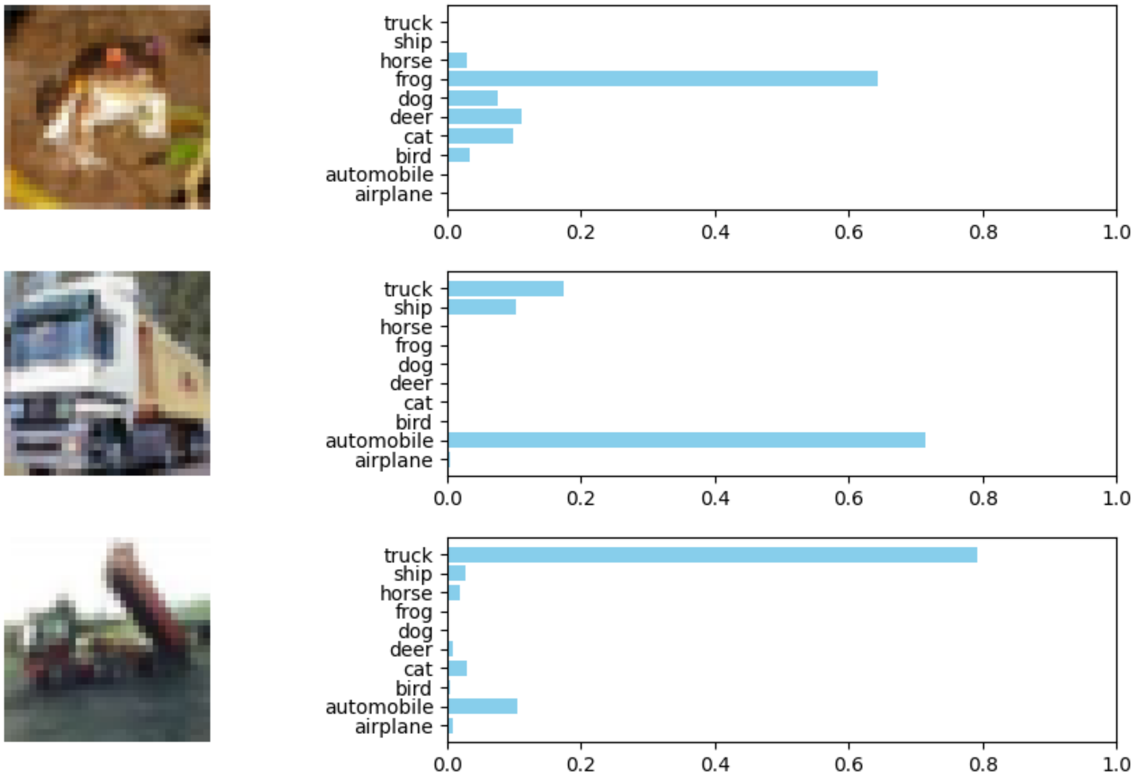
Epoch 5/5

**625/625** ━━━━━━━━━━━━━━━━━━━━ **2s** 3ms/step - accuracy: 0.4593 - loss: 1.5111 - val_accuracy: 0.4449 - val_loss: 1.5712

**313/313** ━━━━━━━━━━━━━━━━━━━━ **1s** 1ms/step - accuracy: 0.4595 - loss: 1.5316

Run 1: Test accuracy: 45.84%

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 369ms/step
```

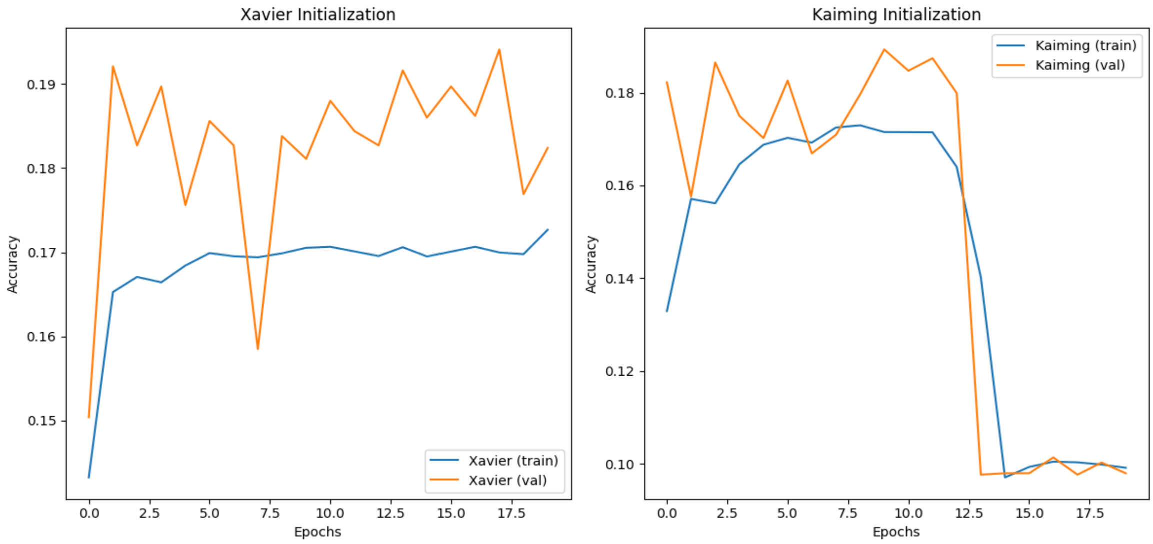



KAIMING AND XAVIERS INITIALIZATION

```
import tensorflow as tf

from tensorflow.keras import layers, models, initializers, regularizers

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.utils import to_categorical

import matplotlib.pyplot as plt

(train_images, train_labels), (test_images, test_labels) = cifar10.load_data()

train_images, test_images = train_images / 255.0, test_images / 255.0

train_labels, test_labels = to_categorical(train_labels), to_categorical(test_labels)


def create_model(initializer, dropout_rate=0.3, kernel_regularizer=None):

    model = models.Sequential()

    model.add(layers.Flatten(input_shape=(32, 32, 3)))

    model.add(layers.Dense(512, kernel_initializer=initializer, kernel_regularizer=kernel_regularizer, activation='relu'))

    model.add(layers.Dropout(dropout_rate))

    model.add(layers.Dense(256, kernel_initializer=initializer, kernel_regularizer=kernel_regularizer, activation='relu'))
```

```
    model.add(layers.Dropout(dropout_rate))

    model.add(layers.Dense(128, kernel_initializer=initializer, kernel_regularizer=kernel_regularizer,
activation='relu'))

    model.add(layers.Dropout(dropout_rate))

    model.add(layers.Dense(64, kernel_initializer=initializer, kernel_regularizer=kernel_regularizer,
activation='relu'))

    model.add(layers.Dropout(dropout_rate))

    model.add(layers.Dense(32, kernel_initializer=initializer, kernel_regularizer=kernel_regularizer,
activation='relu'))

    model.add(layers.Dropout(dropout_rate))

    model.add(layers.Dense(10, activation='softmax'))  # Output layer for 10 classes

    return model

xavier_initializer = initializers.glorot_normal()

kaiming_initializer = initializers.he_normal()

xavier_model = create_model(xavier_initializer, dropout_rate=0.3,
kernel_regularizer=regularizers.l2(0.01))

kaiming_model = create_model(kaiming_initializer, dropout_rate=0.3,
kernel_regularizer=regularizers.l2(0.01))

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

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

xavier_history = xavier_model.fit(train_images, train_labels, epochs=20, validation_split=0.2)

kaiming_history = kaiming_model.fit(train_images, train_labels, epochs=20, validation_split=0.2)

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)

plt.plot(xavier_history.history['accuracy'], label='Xavier (train)')

plt.plot(xavier_history.history['val_accuracy'], label='Xavier (val)')

plt.title('Xavier Initialization')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.subplot(1, 2, 2)

plt.plot(kaiming_history.history['accuracy'], label='Kaiming (train)')

plt.plot(kaiming_history.history['val_accuracy'], label='Kaiming (val)')

plt.title('Kaiming Initialization')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')
```

```
plt.legend()
plt.tight_layout()
plt.show()
```



```
Epoch 1/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.1232 - loss: 5.0562 - val_accuracy: 0.1504 - val_loss: 2.2591
Epoch 2/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1622 - loss: 2.2187 - val_accuracy: 0.1921 - val_loss: 2.1624
Epoch 3/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1658 - loss: 2.2000 - val_accuracy: 0.1827 - val_loss: 2.1613
Epoch 4/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.1702 - loss: 2.2019 - val_accuracy: 0.1897 - val_loss: 2.1397
Epoch 5/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1677 - loss: 2.2048 - val_accuracy: 0.1756 - val_loss: 2.1554
Epoch 6/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.1676 - loss: 2.2013 - val_accuracy: 0.1856 - val_loss: 2.1384
Epoch 7/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.1698 - loss: 2.1956 - val_accuracy: 0.1827 - val_loss: 2.1372
Epoch 8/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1688 - loss: 2.2004 - val_accuracy: 0.1585 - val_loss: 2.1903
Epoch 9/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1654 - loss: 2.1985 - val_accuracy: 0.1838 - val_loss: 2.1523
Epoch 10/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1706 - loss: 2.1924 - val_accuracy: 0.1811 - val_loss: 2.1275
Epoch 11/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.1676 - loss: 2.2000 - val_accuracy: 0.1880 - val_loss: 2.1418
Epoch 12/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.1681 - loss: 2.1936 - val_accuracy: 0.1844 - val_loss: 2.1356
Epoch 13/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.1668 - loss: 2.1917 - val_accuracy: 0.1827 - val_loss: 2.1353
Epoch 14/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1741 - loss: 2.1916 - val_accuracy: 0.1916 - val_loss: 2.1630
Epoch 15/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - accuracy: 0.1687 - loss: 2.2097 - val_accuracy: 0.1860 - val_loss: 2.1342
Epoch 16/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1677 - loss: 2.1901 - val_accuracy: 0.1897 - val_loss: 2.1413
Epoch 17/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.1696 - loss: 2.1980 - val_accuracy: 0.1862 - val_loss: 2.1354
Epoch 18/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.1706 - loss: 2.1899 - val_accuracy: 0.1941 - val_loss: 2.1435
Epoch 19/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.1690 - loss: 2.1944 - val_accuracy: 0.1769 - val_loss: 2.1492
Epoch 20/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.1730 - loss: 2.1829 - val_accuracy: 0.1824 - val_loss: 2.1570
Epoch 1/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 11s 5ms/step - accuracy: 0.1145 - loss: 7.5723 - val_accuracy: 0.1822 - val_loss: 2.2577
Epoch 2/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1551 - loss: 2.2616 - val_accuracy: 0.1576 - val_loss: 2.2500
Epoch 3/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1558 - loss: 2.2357 - val_accuracy: 0.1865 - val_loss: 2.1546
Epoch 4/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1650 - loss: 2.2068 - val_accuracy: 0.1750 - val_loss: 2.1529
Epoch 5/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1729 - loss: 2.1978 - val_accuracy: 0.1702 - val_loss: 2.1871
Epoch 6/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1685 - loss: 2.2004 - val_accuracy: 0.1826 - val_loss: 2.1557
Epoch 7/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1703 - loss: 2.1910 - val_accuracy: 0.1669 - val_loss: 2.1979
Epoch 8/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1734 - loss: 2.1938 - val_accuracy: 0.1709 - val_loss: 2.1864
Epoch 9/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1687 - loss: 2.1936 - val_accuracy: 0.1796 - val_loss: 2.1790
Epoch 10/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.1720 - loss: 2.1913 - val_accuracy: 0.1893 - val_loss: 2.1464
Epoch 11/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1698 - loss: 2.1941 - val_accuracy: 0.1847 - val_loss: 2.1343
Epoch 12/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.1741 - loss: 2.1879 - val_accuracy: 0.1874 - val_loss: 2.1299
Epoch 13/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1705 - loss: 2.1973 - val_accuracy: 0.1799 - val_loss: 2.1678
Epoch 14/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1523 - loss: 2.2353 - val_accuracy: 0.0977 - val_loss: 2.3129
Epoch 15/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.0961 - loss: 2.3117 - val_accuracy: 0.0980 - val_loss: 2.3051
Epoch 16/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.0983 - loss: 2.3050 - val_accuracy: 0.0980 - val_loss: 2.3040
Epoch 17/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.0976 - loss: 2.3045 - val_accuracy: 0.1014 - val_loss: 2.3029
Epoch 18/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.0997 - loss: 2.3046 - val_accuracy: 0.0977 - val_loss: 2.3028
Epoch 19/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.0995 - loss: 2.3037 - val_accuracy: 0.1003 - val_loss: 2.3029
Epoch 20/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.0987 - loss: 2.3044 - val_accuracy: 0.0980 - val_loss: 2.3034
```

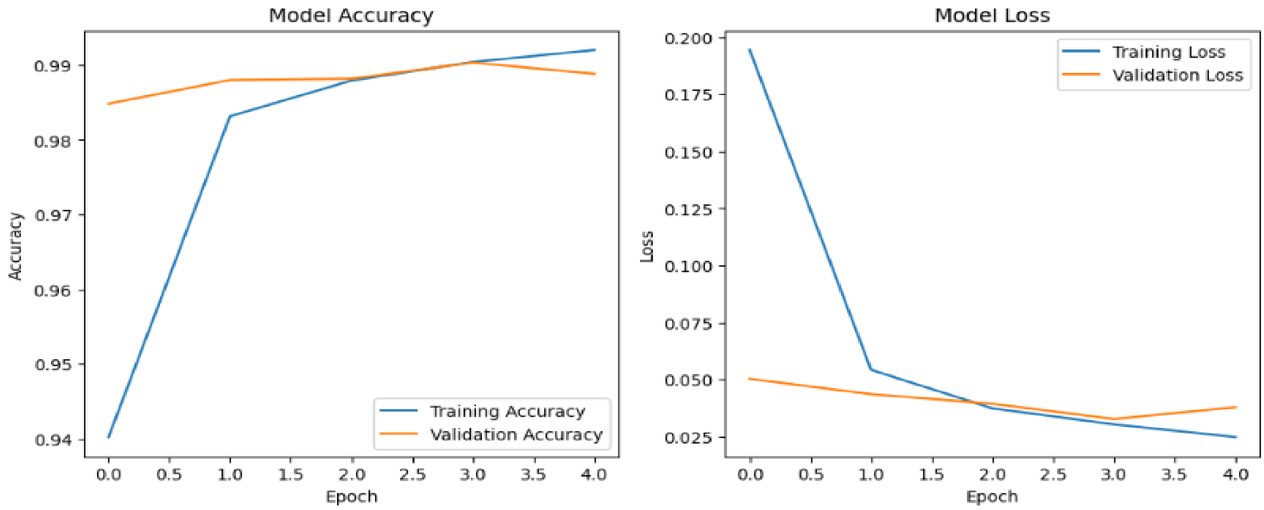
DIGIT CLASSIFICATION ON MNIST DATASET

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
# Load MNIST dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
# Normalize the pixel values to the range [0, 1]
train_images, test_images = train_images / 255.0, test_images / 255.0
# Reshape the images to add an extra dimension for the channels (grayscale = 1 channel)
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1)
test_images = test_images.reshape(test_images.shape[0], 28, 28, 1)
# One-hot encode the labels
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
# Define the CNN model architecture
model = tf.keras.Sequential([
   tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
   tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
   tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
   tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),

   tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
   tf.keras.layers.Flatten(),
   tf.keras.layers.Dense(64, activation='relu'),
   tf.keras.layers.Dense(10, activation='softmax')
])
# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
# Train the model
history = model.fit(train_images, train_labels, epochs=5, batch_size=64, validation_split=0.1)
# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(test_images, test_labels)
```

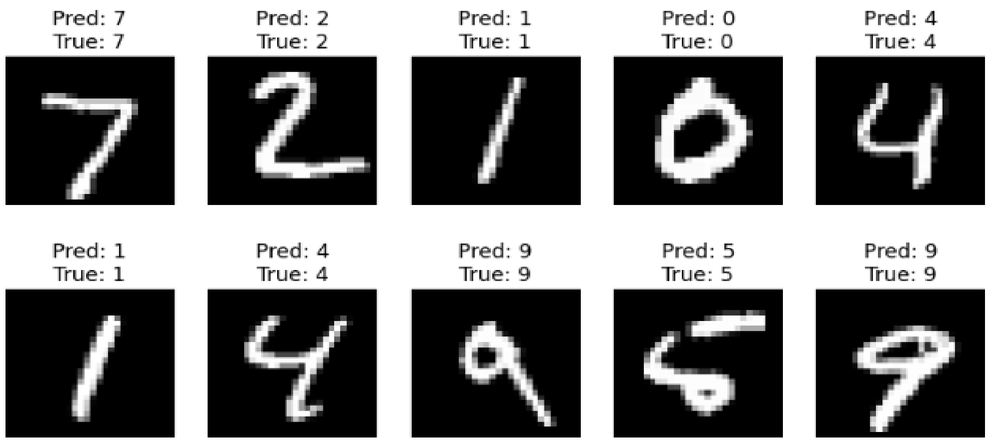
```python
print(f"\nTest Accuracy: {test_acc * 100:.4f}%")
# Visualize accuracy and loss during training
def plot_history(history):
    plt.figure(figsize=(12, 5))
    # Accuracy plot
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Training Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title('Model Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    # Loss plot
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Model Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
plot_history(history)
# Predict the first 10 test images
predictions = model.predict(test_images[:10])

# Display predicted and actual labels for the first 10 test images
def display_predictions(images, predictions, labels):
    plt.figure(figsize=(10, 5))
    # Loop through the first 10 images
    for i in range(10):
        plt.subplot(2, 5, i + 1)
        # Display the image (reshaping it to 28x28 as it was initially)
        plt.imshow(images[i].reshape(28, 28), cmap='gray')
        # Show predicted label and actual label for each image
```

```
        plt.title(f"Pred: {predictions[i].argmax()}\nTrue: {labels[i].argmax()}")
        plt.axis('off')  # Remove axis for clarity
    plt.show()
# Call function to display first 10 images and their predicted vs actual labels
display_predictions(test_images[:10], predictions, test_labels[:10])
```

```
Epoch 1/5
844/844 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.8619 - loss: 0.4472 - val_accuracy: 0.9848 - val_loss: 0.0503
Epoch 2/5
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9806 - loss: 0.0602 - val_accuracy: 0.9880 - val_loss: 0.0435
Epoch 3/5
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9877 - loss: 0.0375 - val_accuracy: 0.9882 - val_loss: 0.0394
Epoch 4/5
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9906 - loss: 0.0293 - val_accuracy: 0.9903 - val_loss: 0.0326
Epoch 5/5
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9925 - loss: 0.0227 - val_accuracy: 0.9888 - val_loss: 0.0378
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9886 - loss: 0.0361

Test Accuracy: 99.1900%
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 154ms/step
```

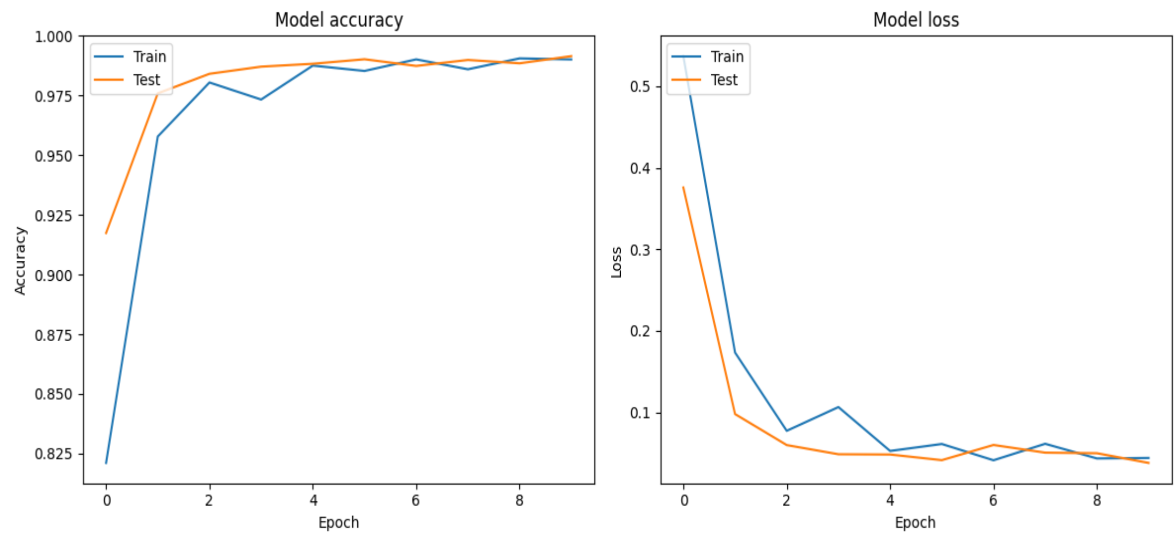


PRETRAINED VGGNET DIGIT CLASSIFICATION ON MNIST DATASET

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import models, layers
from tensorflow.keras.applications import VGG19
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
# Load MNIST dataset
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# Preprocess the data
y_train = to_categorical(y_train)  # One-hot encoding of labels
y_test = to_categorical(y_test)
# Pad the images to match the input size of VGG19 (48x48, you might want to increase this in practice)
x_train = np.pad(x_train, ((0, 0), (10, 10), (10, 10)), mode='constant', constant_values=255)
x_test = np.pad(x_test, ((0, 0), (10, 10), (10, 10)), mode='constant', constant_values=255)
# Convert grayscale images to RGB by stacking the image three times along the last axis
x_train = np.stack([x_train] * 3, axis=-1)
x_test = np.stack([x_test] * 3, axis=-1)
# Normalize pixel values to the range [0, 1]
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Load VGG19 without the top classification layers
vgg_model = VGG19(weights='imagenet', include_top=False, input_shape=(48, 48, 3))
# Create a new model with custom classification layers
model = models.Sequential()
# Add the VGG19 base model
model.add(vgg_model)
# Add flattening and dense layers for classification
model.add(layers.Flatten())
```

```
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))  # 10 classes for digits (0-9)
# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
# Train the model
history = model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))
# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc}")
# Visualize the model's performance (e.g., loss and accuracy)
plt.figure(figsize=(12, 5))
# Plot training & validation accuracy values
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train')
plt.plot(history.history['val_accuracy'], label='Test')
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(loc='upper left')
# Plot training & validation loss values
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train')
plt.plot(history.history['val_loss'], label='Test')
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(loc='upper left')

plt.tight_layout()
plt.show()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg19/vgg19_weights_tf_dim_ordering_tf_kern
80134624/80134624 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
Epoch 1/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 129s 62ms/step - accuracy: 0.6054 - loss: 1.0898 - val_accuracy: 0.9174 - val_loss: 0.3756
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 132s 61ms/step - accuracy: 0.9468 - loss: 0.2201 - val_accuracy: 0.9759 - val_loss: 0.0978
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 141s 61ms/step - accuracy: 0.9805 - loss: 0.0768 - val_accuracy: 0.9841 - val_loss: 0.0598
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 142s 61ms/step - accuracy: 0.9738 - loss: 0.1077 - val_accuracy: 0.9871 - val_loss: 0.0485
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 142s 61ms/step - accuracy: 0.9885 - loss: 0.0470 - val_accuracy: 0.9883 - val_loss: 0.0482
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 114s 61ms/step - accuracy: 0.9867 - loss: 0.0543 - val_accuracy: 0.9902 - val_loss: 0.0413
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 143s 62ms/step - accuracy: 0.9893 - loss: 0.0440 - val_accuracy: 0.9874 - val_loss: 0.0599
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 141s 61ms/step - accuracy: 0.9847 - loss: 0.0679 - val_accuracy: 0.9899 - val_loss: 0.0505
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 142s 61ms/step - accuracy: 0.9921 - loss: 0.0355 - val_accuracy: 0.9885 - val_loss: 0.0498
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 114s 61ms/step - accuracy: 0.9897 - loss: 0.0461 - val_accuracy: 0.9915 - val_loss: 0.0379
313/313 ━━━━━━━━━━━━━━━━━━━━ 4s 14ms/step - accuracy: 0.9897 - loss: 0.0453
Test accuracy: 0.9915000200271606
```

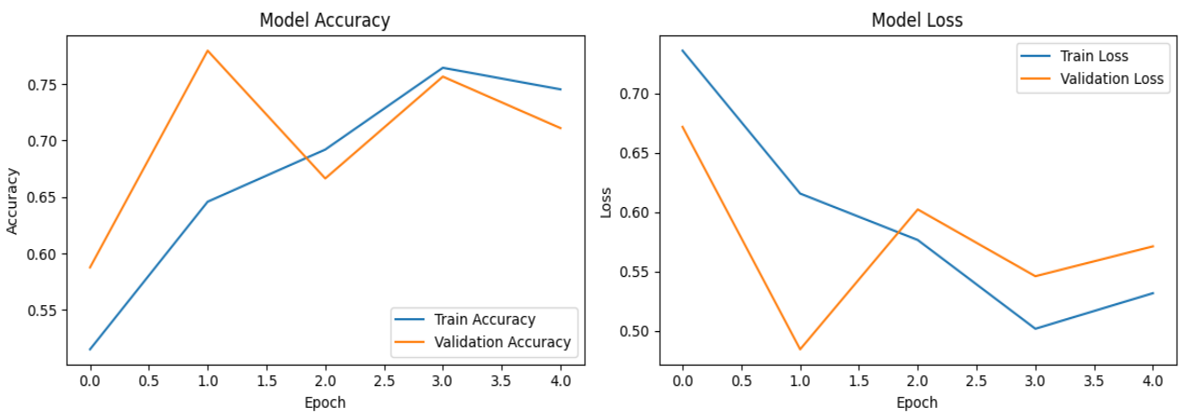


SimpleRNN

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
import matplotlib.pyplot as plt
max_features = 10000  # Only consider the top 10,000 words
maxlen = 200  # Cut texts after this number of words
batch_size = 32
# Load the dataset, keeping only the top `max_features` words
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
# Pad sequences to ensure uniform input size
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
# Build the SimpleRNN model
model = Sequential()
model.add(Embedding(max_features, 128))  # Embedding layer
model.add(SimpleRNN(128, activation='tanh'))  # RNN layer
model.add(Dropout(0.5))  # Add dropout to prevent overfitting
model.add(Dense(1, activation='sigmoid'))  # Output layer for binary classification
# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Early stopping callback to stop training when validation accuracy stops improving
early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
# Train the model and store the history
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=20, validation_split=0.2,
callbacks=[early_stopping])
# Evaluate the model
score, accuracy = model.evaluate(x_test, y_test, batch_size=batch_size)
```

```
print(f'Test score: {score:.4f}')
print(f'Test accuracy: {accuracy:.4f}')
# Plot accuracy and loss over epochs
plt.figure(figsize=(12, 4))
# Plot training & validation accuracy values
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
# Plot training & validation loss values
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.tight_layout()
plt.show()

# Function to decode reviews
def decode_review(review):
    word_index = imdb.get_word_index()
    reverse_word_index = {value: key for key, value in word_index.items()}
    decoded_review = ' '.join([reverse_word_index.get(i - 3, '?') for i in review])
    return decoded_review
# Sample reviews for classification
sample_reviews = [
    "This movie was fantastic! I loved it.",
```

```
    "I didn't like this film at all. It was boring and too long.",
    "An average film, nothing special.",
]
# Preprocess sample reviews (correct the offset for reserved tokens)
def preprocess_reviews(reviews):
    encoded_reviews = []
    word_index = imdb.get_word_index()
    for review in reviews:
        encoded_review = [word_index.get(word.lower(), 0) + 3 for word in review.split()]
        encoded_reviews.append(encoded_review)
    return sequence.pad_sequences(encoded_reviews, maxlen=maxlen)
# Prepare sample reviews for prediction
encoded_sample_reviews = preprocess_reviews(sample_reviews)
# Make predictions
predictions = model.predict(encoded_sample_reviews)
predicted_classes = (predictions > 0.5).astype("int32")  # 1 for positive, 0 for negative
# Display the results
for review, prediction in zip(sample_reviews, predicted_classes):
    sentiment = "Positive" if prediction[0] == 1 else "Negative"
    print(f"Review: {review}\nSentiment: {sentiment}\n")
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 233ms/step
Review: This movie was fantastic! I loved it.
Sentiment: Positive

Review: I didn't like this film at all. It was boring and too long.
Sentiment: Negative

Review: An average film, nothing special.
Sentiment: Positive
```

```
Epoch 1/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 18s 24ms/step - accuracy: 0.5029 - loss: 0.7474 - val_accuracy: 0.5876 - val_loss: 0.6716
Epoch 2/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 15s 23ms/step - accuracy: 0.5780 - loss: 0.6703 - val_accuracy: 0.7796 - val_loss: 0.4842
Epoch 3/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 21s 23ms/step - accuracy: 0.7211 - loss: 0.5516 - val_accuracy: 0.6664 - val_loss: 0.6021
Epoch 4/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 14s 23ms/step - accuracy: 0.7494 - loss: 0.5145 - val_accuracy: 0.7566 - val_loss: 0.5458
Epoch 5/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 14s 23ms/step - accuracy: 0.7666 - loss: 0.5047 - val_accuracy: 0.7110 - val_loss: 0.5710
782/782 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - accuracy: 0.7810 - loss: 0.4782
Test score: 0.4742
Test accuracy: 0.7846
```

LSTM AND GRU

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Embedding, SimpleRNN, LSTM, GRU
# Load the IMDB dataset
max_features = 20000  # Number of words to consider as features
maxlen = 100  # Cut texts after this number of words (max. length)
batch_size = 32
# Load and preprocess the dataset
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
# Function to create model based on RNN type
def create_model(rnn_type, units=128):
    model = Sequential()
    model.add(Embedding(max_features, 128))
    if rnn_type == 'SimpleRNN':
        model.add(SimpleRNN(units))
    elif rnn_type == 'LSTM':
        model.add(LSTM(units))
    elif rnn_type == 'GRU':
        model.add(GRU(units))
```

```python
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
# Train models and store their histories
rnn_types = ['SimpleRNN', 'LSTM', 'GRU']
histories = {}
for rnn_type in rnn_types:
    print(f"Training {rnn_type} model...")
    model = create_model(rnn_type)
    history = model.fit(x_train, y_train, batch_size=batch_size, epochs=5, validation_data=(x_test,
y_test))
    histories[rnn_type] = history
    test_loss, test_acc = model.evaluate(x_test, y_test)
    print(f"Test accuracy: {test_acc}")
# Visualization of results
def plot_history(histories, metric='accuracy'):
    plt.figure(figsize=(12, 8))
    for rnn_type in histories:
        plt.plot(histories[rnn_type].history[metric], label=f'{rnn_type} training {metric}')
        plt.plot(histories[rnn_type].history[f'val_{metric}'], label=f'{rnn_type} validation {metric}')
    plt.title(f'Model {metric.capitalize()} Comparison')
    plt.ylabel(metric.capitalize())
    plt.xlabel('Epochs')
    plt.legend(loc='best')
    plt.show()
# Plot accuracy and loss
plot_history(histories, 'accuracy')
plot_history(histories, 'loss')
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz
17464789/17464789 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Training SimpleRNN model...
Epoch 1/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 70s 87ms/step - accuracy: 0.5750 - loss: 0.6679 - val_accuracy: 0.6670 - val_loss: 0.5988
Epoch 2/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 68s 87ms/step - accuracy: 0.7185 - loss: 0.5457 - val_accuracy: 0.7312 - val_loss: 0.5364
Epoch 3/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 80s 85ms/step - accuracy: 0.7990 - loss: 0.4309 - val_accuracy: 0.7660 - val_loss: 0.5132
Epoch 4/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 82s 85ms/step - accuracy: 0.7969 - loss: 0.4242 - val_accuracy: 0.6626 - val_loss: 0.6343
Epoch 5/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 82s 85ms/step - accuracy: 0.7930 - loss: 0.4361 - val_accuracy: 0.6449 - val_loss: 0.6684
782/782 ━━━━━━━━━━━━━━━━━━━━ 11s 14ms/step - accuracy: 0.6443 - loss: 0.6757
Test accuracy: 0.6448799967765808
Training LSTM model...
Epoch 1/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 258s 327ms/step - accuracy: 0.7082 - loss: 0.5310 - val_accuracy: 0.8428 - val_loss: 0.3609
Epoch 2/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 215s 267ms/step - accuracy: 0.9114 - loss: 0.2317 - val_accuracy: 0.8541 - val_loss: 0.3511
Epoch 3/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 302s 319ms/step - accuracy: 0.9534 - loss: 0.1342 - val_accuracy: 0.8416 - val_loss: 0.4105
Epoch 4/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 260s 316ms/step - accuracy: 0.9703 - loss: 0.0881 - val_accuracy: 0.8413 - val_loss: 0.4811
Epoch 5/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 225s 268ms/step - accuracy: 0.9810 - loss: 0.0568 - val_accuracy: 0.8416 - val_loss: 0.6157
782/782 ━━━━━━━━━━━━━━━━━━━━ 54s 70ms/step - accuracy: 0.8396 - loss: 0.6304
Test accuracy: 0.8416000008583069
Training GRU model...
Epoch 1/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 170s 213ms/step - accuracy: 0.7293 - loss: 0.5073 - val_accuracy: 0.8578 - val_loss: 0.3294
Epoch 2/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 222s 239ms/step - accuracy: 0.9150 - loss: 0.2191 - val_accuracy: 0.8524 - val_loss: 0.3483
Epoch 3/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 179s 210ms/step - accuracy: 0.9636 - loss: 0.1080 - val_accuracy: 0.8450 - val_loss: 0.4429
Epoch 4/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 205s 214ms/step - accuracy: 0.9820 - loss: 0.0541 - val_accuracy: 0.8364 - val_loss: 0.5845
Epoch 5/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 201s 213ms/step - accuracy: 0.9929 - loss: 0.0221 - val_accuracy: 0.8344 - val_loss: 0.7296
782/782 ━━━━━━━━━━━━━━━━━━━━ 27s 34ms/step - accuracy: 0.8322 - loss: 0.7474
Test accuracy: 0.8343600034713745
```