IoT and Edge Computing (CS3100)
Project Report

Mobile Price Classification

Submitted by

Student Name: **PRANAV P KULKARNI**

USN:**1RVU22CSE119**

School of Computer Science and Engineering

RV University

Submitted to

Professor Name: **Prof.Chandramouleeswaran Sankaran**

Designation**: Professor**

School of Computer Science and Engineering

RV University

Submission date: 14-09-2024

IoT and Edge Computing (CS3100)
Project Report

1. Abstract:

The project aimed to build a mobile price classification model that could categorize devices based on attributes such as battery power, RAM, and display features. The dataset, sourced from Kaggle, included various technical specifications as inputs for predicting price range categories. Using StandardScaler for feature scaling, a classification model was trained and optimized for accuracy. The model was then converted to TensorFlow Lite (TFLite) and deployed on an ESP32 microcontroller, demonstrating efficient, low-power on-device predictions. Key outcomes include successful classification accuracy and effective deployment on ESP32, showcasing real-time, cost-effective mobile price predictions for embedded systems**.**

2. Introduction

Project Background and Relevance:

IoT (Internet of Things) and Edge Computing are transformative technologies that enable devices to collect, process, and analyze data at the edge of networks, closer to where data is generated. This reduces latency, improves response times, and enhances data privacy by minimizing data transfer to central cloud servers. In IoT, interconnected devices communicate and share data, providing real-time insights and automation across sectors like healthcare, agriculture, and smart cities. Edge Computing complements IoT by processing data locally on devices, making it ideal for applications requiring low latency and efficient resource usage, especially in remote or low-bandwidth environments. Together, IoT and Edge Computing empower innovations such as predictive maintenance, remote monitoring, and real-time analytics, paving the way for intelligent, autonomous systems across various industries.

Objectives:

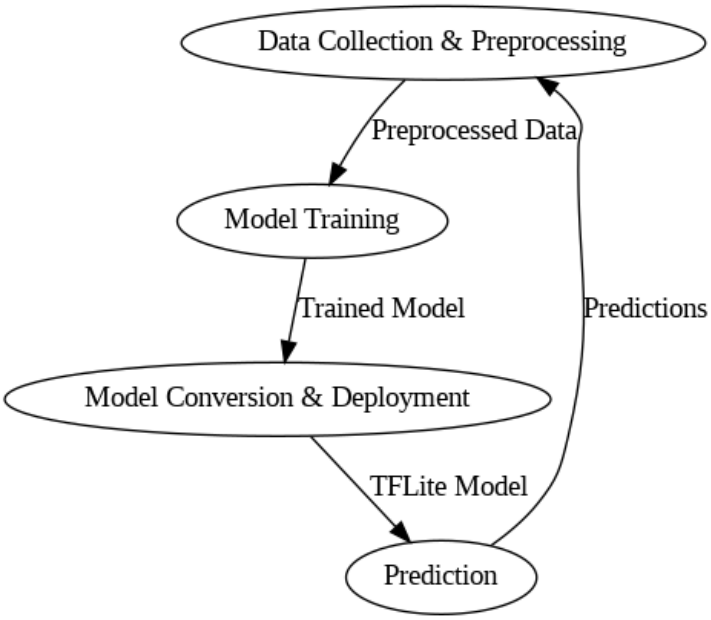
The project aims to develop a mobile price classification model that accurately categorizes mobile devices based on technical specifications, such as battery power, RAM, and display features. By deploying this model on an ESP32 microcontroller using TensorFlow Lite, the project seeks to demonstrate real-time, on-device price prediction capability within the constraints of an edge device. This deployment on an IoT-based platform exemplifies the use of Edge Computing for efficient, low-power data processing, making it applicable to resource-limited settings and enhancing the accessibility of machine learning predictions in embedded systems.

IoT and Edge Computing (CS3100)
Project Report

3. System Overview

System Architecture:

a) Outline of data flow.



b) **Key components:**

- **Data Collection & Preprocessing:** The dataset includes features such as battery power, clock speed, RAM, and other mobile specifications, along with the target variable price_range. Categorical features like blue, dual_sim, four_g, etc., are one-hot encoded, while numerical features are scaled using StandardScaler to standardize the data for the neural network model.
- **Model Training:** The preprocessed data is then used to train a neural network model built with TensorFlow. The model consists of an input layer, two hidden layers with ReLU activation functions, and an output layer with a softmax activation to predict mobile price ranges. The training process uses categorical cross-entropy loss and the Adam optimizer, running for 50 epochs to optimize the model's performance**.**

- **Model Conversion & Deployment:**
  Once the model is trained, it is saved and converted into TensorFlow Lite (TFLite) format, which optimizes the model for edge deployment. The converted TFLite model is then saved to a file named MobilePriceClassifyModel.tflite, ready for deployment on target devices for real-time inference.
- **Real-time Prediction:**
  For real-time prediction, the TFLite model is loaded using the TensorFlow Lite Interpreter, and the necessary tensors are allocated for efficient inference on the edge device. Real-time data, such as mobile specifications collected from IoT devices, is passed through the model to classify the mobile's price range, providing instant results on the device.

4. Details of the Dataset:

Dataset Details:

The dataset aims to solve the problem of categorizing mobile phones into distinct price ranges based on their features, which helps to understand how various hardware specifications relate to pricing. Instead of predicting the exact price, the model classifies each phone into one of four price ranges (0 to 3), representing increasing price levels. This approach simplifies price prediction and provides useful insights into the relative cost.

Input Features

The dataset includes 20 features that impact the price range:

1. **battery_power**: Battery capacity in mAh, influencing usage time and user appeal.
2. **blue**: Bluetooth availability (1 if available, 0 otherwise), indicating connectivity options.
3. **clock_speed**: Processor speed (in GHz), affecting device performance.
4. **dual_sim**: Whether the phone supports dual SIM cards (1 if yes, 0 otherwise).
5. **fc**: Front camera resolution (in megapixels), which affects the appeal for photography.
6. **four_g**: 4G connectivity support (1 if available, 0 otherwise), relevant for data speed.
7. **int_memory**: Internal memory storage (in GB), which impacts storage capacity.
8. **m_dep:** Mobile depth or thickness (in cm), contributing to design preference.
9. **mobile_wt**: Weight of the mobile (in grams), affecting portability and user experience.
10. **n_cores**: Number of cores in the processor, influencing multitasking capabilities.
11. **pc**: Primary camera resolution (in megapixels), an important factor for image quality.
12. **px_height and px_width**: Screen resolution in pixels (height and width), impacting display clarity.

13. **ram**: Random Access Memory (RAM in MB), critical for performance and app handling.
14. **sc_h and sc_w**: Screen height and width (in cm), related to screen size and usability.
15. **talk_time**: Maximum talk time on a single battery charge (in hours).
16. **three_g**: 3G connectivity support (1 if available, 0 otherwise), an alternative to 4G.
17. **touch_screen**: Touchscreen capability (1 if available, 0 otherwise).
18. **wifi**: WiFi connectivity availability (1 if available, 0 otherwise).

Output:

price_range: The target variable with values ranging from 0 to 3, where each value represents a specific price range:

0: Low

1: Medium-Low

2: Medium-High

3: High

5. Model Design:

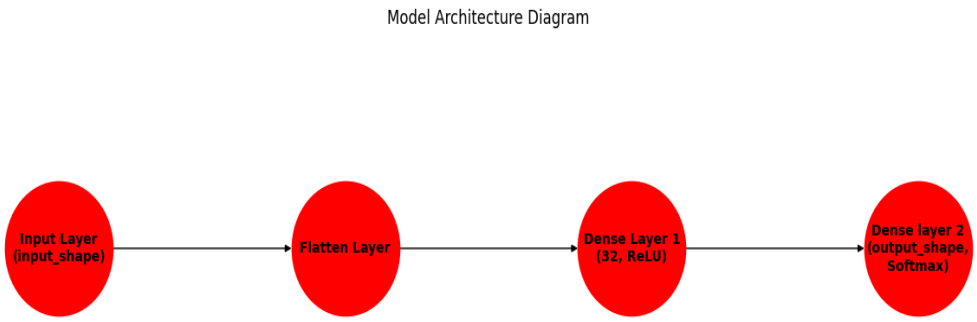


Figure 1: Model Architecture

The model architecture diagram you provided represents a simple neural network with the following layers:

IoT and Edge Computing (CS3100)
Project Report

Input Layer (input_shape): This is the entry point of the model, where the input data (represented by the variable X) is fed into the network. The input shape is determined by the shape of the input data, which is X.shape[1] in this case.

Flatten Layer: This layer takes the input data and flattens it into a one-dimensional vector, preparing it for the subsequent dense (fully connected) layers.

Dense Layer 1 (32, ReLU): This is the first dense layer with 32 units. The activation function used is ReLU (Rectified Linear Unit), which introduces non-linearity to the model and helps it learn complex patterns in the data.

Output Layer (output_shape, Softmax): This is the final layer of the model, which produces the output. The number of units in this layer corresponds to the output shape, which is y.shape[1]. The Softmax activation function is used, which is common for classification problems where the model needs to output a probability distribution over the possible classes.

The code you provided sets the following hyperparameters:

DENSE1_SIZE = 32: The number of units in the first dense layer.
DENSE2_SIZE = 16: This line is not present in the diagram, so it's likely that the model does not have a second dense layer.
NUM_OF_EPOCHS = 50: The number of training epochs for the model.
BATCH_SIZE = 8: The batch size used during training.

The model is created using the **tf.keras.Sequential()** API, which allows you to easily stack layers to build the network. The model.add() method is used to add each layer to the model.

The model is then compiled with the following settings:

Loss function: categorical_crossentropy, which is commonly used for multi-class classification problems.

Optimizer: 'adam', which is a popular gradient-based optimization algorithm.

Metric: 'acc', which tracks the classification accuracy during training and evaluation.

Finally, the model.summary() method is called to print a summary of the model architecture, including the number of trainable and non-trainable parameters.
Overall, this model architecture represents a basic neural network for a classification task, with an input layer, a flatten layer, a single hidden dense layer, and an output layer with a Softmax activation.

IoT and Edge Computing (CS3100)
Project Report

6. Requirements

Hardware:

ESP32: A cost-effective, energy-efficient microcontroller featuring built-in Wi-Fi and Bluetooth, making it well-suited for real-time monitoring and control applications, particularly in IoT-based milk quality detection systems**.**

Software:

1) **Python Libraries:** Core libraries including TensorFlow (for model development and deployment), NumPy (for data manipulation), and Pandas (for data analysis).
2) **Python Version:** 3.7.16
3) **IDE:** Any IDE compatible with ESP32 and Python, such as Jupyter Notebook, Arduino IDE, and the EloquentTinyML library for model deployment**.**

7. Methodology

Setup:

Hardware Configuration:

This low-cost, low-power microcontroller with built-in Wi-Fi and Bluetooth is ideal for IoT applications. It serves as the edge device in this setup, running the machine learning model for real-time predictions on mobile price classification. It is designed to handle real-time data processing while ensuring energy efficiency.

Software:

1)Python Libraries:

1. **TensorFlow:** Used for building and deploying the neural network model.
2. **NumPy**: Used for numerical operations and data manipulation.
3. **Pandas:** Utilized for data analysis and cleaning prior to model training.

2)Python Version: 3.7.16, compatible with the necessary libraries and frameworks.

3)**IDE**: The development environment used includes Jupyter Notebook for model development and Arduino IDE or EloquentTinyML library for deploying the model onto the ESP32 microcontroller. The EloquentTinyML library facilitates the conversion of TensorFlow models to TensorFlow Lite models that can be run efficiently on the ESP32.

IoT and Edge Computing (CS3100)
Project Report

Code Summary:

1)**Data Preprocessing:**

a) **Feature Scaling:** The features used for prediction include both numerical and binary (categorical) features. The numerical features are scaled using StandardScaler to standardize the input data. This ensures that all numerical features are on the same scale, which helps the model converge faster during training and improves overall performance.

The following numerical features are standardized:

b)Numerical Features:

```
‘battery_power', 'clock_speed', 'int_memory', 'm_dep', 'fc', 'mobile_wt',
'n_cores', 'pc', 'px_height', 'px_width', 'ram', 'sc_h', 'sc_w', 'talk_time'
```

For the binary features, which represent categorical data with values 0 and 1, dummy variables are created using one-hot encoding. These binary features include:

c)Binary Features:

```
'blue', 'dual_sim', 'four_g', 'three_g', 'touch_screen', 'wifi'
```

By using one-hot encoding for these binary features, they are transformed into dummy variables (0s and 1s), increasing the total number of features to 26. This consists of the 20 original numerical features plus 6 additional dummy features for the binary categories.

The **price range is encoded into four categories (0, 1, 2, and 3),** representing the classification target. This ensures the target is prepared for multi-class classification, which the model can predict based on the input features.

Thus, the final dataset includes **26 features**, with the numerical features standardized and the binary features encoded as dummy variables. This preprocessing step allows the model to learn effectively from the data and make accurate predictions for the mobile price range.

IoT and Edge Computing (CS3100)
Project Report

2) Model Building:

A simple neural network model is built using TensorFlow, consisting of the following layers:

a) **Input Layer:** The input layer flattens the data, transforming the 2D input into a 1D array to pass it to the next layer. The input shape is determined by the number of features in the dataset.
b) **Hidden Layer 1:** The first hidden layer consists of 32 nodes (as defined by DENSE1_SIZE), using the ReLU activation function. ReLU helps introduce non-linearity, enabling the model to learn complex patterns in the data.
c) **Output Layer:** The output layer has the same number of nodes as the number of classes in the target variable (i.e., price range). It uses the softmax activation function to predict the probability distribution over the four classes (price range 0, 1, 2, and 3).

Compilation:

The model is compiled using the **Adam optimizer,** which is efficient and adaptive for training deep learning models. The loss function is set to **categorical cross-entropy** since this is a multi-class classification problem. The model also tracks accuracy ('acc') during training to monitor its performance.

3) Model Training:

The model is trained for **50 epochs** with a **batch size of 8**. During training, the model's weights are adjusted based on the gradients computed using the backpropagation algorithm, optimizing the loss function. The Adam optimizer is used to improve convergence, and the model's performance is evaluated using accuracy.

4)Model Deployment:

Conversion to TensorFlow Lite: Once the model is trained, it is converted into a TensorFlow Lite format, which is optimized for edge devices such as the ESP32. This process reduces the model size, enabling it to run efficiently on limited-resource devices.

Model Deployment to ESP32: The converted model is loaded onto the ESP32 microcontroller using the EloquentTinyML library. This library allows the model to make predictions based on real-time data collected from the mobile device's features.

The ESP32 device is programmed to collect input data, preprocess it, and pass it to the trained model.

5) Performance Monitoring:

The ESP32's performance in terms of prediction time and accuracy is continuously monitored, and necessary optimizations are made for real-time deployment.

8. Implementation and Testing

Core Code Excerpts:

a.One-hot encoding:

```
df = pd.get_dummies(df, columns=['price_range','blue', 'dual_sim', 'four_g', 'three_g', 'touch_screen', 'wifi'])
df.head()
```

Python

wt	n_cores	pc	px_height	px_width	...	dual_sim_0	dual_sim_1	four_g_0	four_g_1	three_g_0	three_g_1	touch_screen_0	touch_screen_1	wifi_0	wifi_1
188	2	2	20	756	...	1	0	1	0	1	0	1	0	0	1
136	3	6	905	1988	...	0	1	0	1	0	1	0	1	1	0
145	5	6	1263	1716	...	0	1	0	1	0	1	0	1	1	0
131	6	9	1216	1786	...	1	0	1	0	0	1	1	0	1	0
141	2	14	1208	1212	...	1	0	0	1	0	1	0	1	1	0

Figure 2: One-hot encoding is applied to categorical features

In this code, the **`pd.get_dummies()`** function is used to apply one-hot encoding to categorical features in the dataset. Specifically, columns such as **`'price_range'`, `'blue'`, `'dual_sim'`, `'four_g'`, `'three_g'`, `'touch_screen'`,** and **`'wifi'`** are transformed into binary (0 or 1) columns. This process creates new columns for each category in these features, helping the model interpret them as distinct attributes rather than numerical values. The updated dataset is displayed using `df.head()` to show the first few rows with the new dummy-encoded columns.

b.Preparing X_train and X_test:

```
# Identify numerical and binary features
numerical_features = ['battery_power', 'clock_speed', 'int_memory', 'm_dep', 'fc', 'mobile_wt',
                      'n_cores', 'pc', 'px_height', 'px_width', 'ram', 'sc_h', 'sc_w', 'talk_time']

# Initialize the scaler
scaler = StandardScaler()

# Scale the numerical features in X_train
X_train_num_scaled = scaler.fit_transform(X_train[numerical_features])
X_test_num_scaled = scaler.transform(X_test[numerical_features])

# Convert the scaled numerical features back to DataFrame to keep column names
X_train_num_scaled = pd.DataFrame(X_train_num_scaled, columns=numerical_features, index=X_train.index)
X_test_num_scaled = pd.DataFrame(X_test_num_scaled, columns=numerical_features, index=X_test.index)

# Drop original numerical columns from X_train and X_test
X_train = X_train.drop(columns=numerical_features)
X_test = X_test.drop(columns=numerical_features)

# Concatenate the scaled numerical features back with the rest of X_train and X_test
X_train = pd.concat([X_train, X_train_num_scaled], axis=1)
X_test = pd.concat([X_test, X_test_num_scaled], axis=1)

# Now X_train and X_test have the scaled numerical features and the one-hot encoded binary features
print('Shape of X_train and X_test:', X_train.shape, X_test.shape)
print('Shape of y_train and y_test:', y_train.shape, y_test.shape)
```

```
Shape of X_train and X_test: (1600, 26) (400, 26)
Shape of y_train and y_test: (1600, 4) (400, 4)
```

Figure 3: The code scales numerical features and combines them with one-hot encoded binary features, preparing X_train and X_test for model training.

IoT and Edge Computing (CS3100)
Project Report

This code scales the numerical features in **X_train** and **X_test** using **StandardScaler** to standardize them, improving model performance. The scaled values are then converted back into DataFrames to retain the original column names. After dropping the original, unscaled columns, the scaled features are concatenated back with the remaining binary-encoded features. The output confirms that **X_train** and **X_test** have 26 features each, with corresponding target shapes for **y_train** and **y_test.**

c.Model Definition:

```
DENSE1_SIZE = 32
DENSE2_SIZE = 16
NUM_OF_EPOCHS = 50
BATCH_SIZE = 8

model = tf.keras.Sequential()

input_shape = X.shape[1]
print(input_shape)
model.add(tf.keras.layers.Flatten(input_shape =(X.shape[1],)))

model.add(tf.keras.layers.Dense(DENSE1_SIZE, activation='relu'))


output_shape = y.shape[1]
print(output_shape)
model.add(tf.keras.layers.Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['acc'])
model.summary()
```

```
26
4
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 26)                0

 dense (Dense)               (None, 32)                864

 dense_1 (Dense)             (None, 4)                 132

=================================================================
Total params: 996
Trainable params: 996
Non-trainable params: 0
_________________________________________________________________
```

Figure 4: Model defining

The model is a neural network built using TensorFlow, comprising three main layers. It starts with a Flatten layer to convert the input of **26 features into a single dimension**, followed by a Dense layer with **32 nodes using the ReLU activation function**. The output layer has 4 nodes with a **softmax activation function to predict one of four categories**. The model has

a total of **996 trainable parameters** and is compiled with **categorical cross-entropy loss** and the **Adam optimizer.**

d.Model Training:

```
history = model.fit(X_train, y_train, batch_size=BATCH_SIZE,
                    epochs=NUM_OF_EPOCHS,
                    verbose=1, validation_split=0.2)
```

```
Epoch 1/50
160/160 [==============================] - 1s 3ms/step - loss: 1.3234 - acc: 0.3781 - val_loss: 1.1273 - val_acc: 0.5188
Epoch 2/50
160/160 [==============================] - 0s 1ms/step - loss: 0.9570 - acc: 0.6117 - val_loss: 0.8851 - val_acc: 0.6625
Epoch 3/50
160/160 [==============================] - 0s 1ms/step - loss: 0.7427 - acc: 0.7375 - val_loss: 0.7182 - val_acc: 0.7344
Epoch 4/50
160/160 [==============================] - 0s 1ms/step - loss: 0.6024 - acc: 0.8023 - val_loss: 0.6073 - val_acc: 0.7969
Epoch 5/50
160/160 [==============================] - 0s 2ms/step - loss: 0.5050 - acc: 0.8578 - val_loss: 0.5169 - val_acc: 0.8594
Epoch 6/50
160/160 [==============================] - 0s 1ms/step - loss: 0.4327 - acc: 0.8984 - val_loss: 0.4460 - val_acc: 0.8750
Epoch 7/50
160/160 [==============================] - 0s 2ms/step - loss: 0.3752 - acc: 0.9102 - val_loss: 0.3920 - val_acc: 0.9094
Epoch 8/50
160/160 [==============================] - 0s 1ms/step - loss: 0.3306 - acc: 0.9219 - val_loss: 0.3504 - val_acc: 0.9187
Epoch 9/50
160/160 [==============================] - 0s 1ms/step - loss: 0.2948 - acc: 0.9328 - val_loss: 0.3127 - val_acc: 0.9250
Epoch 10/50
160/160 [==============================] - 0s 1ms/step - loss: 0.2645 - acc: 0.9414 - val_loss: 0.2861 - val_acc: 0.9250
Epoch 11/50
160/160 [==============================] - 0s 1ms/step - loss: 0.2417 - acc: 0.9531 - val_loss: 0.2616 - val_acc: 0.9250
Epoch 12/50
160/160 [==============================] - 0s 1ms/step - loss: 0.2203 - acc: 0.9563 - val_loss: 0.2435 - val_acc: 0.9375
Epoch 13/50
...
Epoch 49/50
160/160 [==============================] - 0s 1ms/step - loss: 0.0460 - acc: 0.9961 - val_loss: 0.1107 - val_acc: 0.9469
Epoch 50/50
160/160 [==============================] - 0s 1ms/step - loss: 0.0448 - acc: 0.9961 - val_loss: 0.1154 - val_acc: 0.9438
```

Figure 5: Training the model with 50 epochs

e. Tflite conversion:

```
tf.saved_model.save(model, "saved_mobile_seq_model_keras_dir")
converter = tf.lite.TFLiteConverter.from_saved_model("saved_mobile_seq_model_keras_dir")
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset

tflite_model = converter.convert()
```

```
INFO:tensorflow:Assets written to: saved_mobile_seq_model_keras_dir\assets
INFO:tensorflow:Assets written to: saved_mobile_seq_model_keras_dir\assets
```

Figure 6: Converting a trained TensorFlow model to TensorFlow Lite format

f. ESP32 Integration:

```python
with open('MobilePriceClassifyModel.tflite', 'wb') as f:
  f.write(tflite_model)


interpreter = tf.lite.Interpreter(model_path="MobilePriceClassifyModel.tflite")
interpreter.allocate_tensors()
```

Figure 7: Saving and loading the Mobile Price Classification model in TensorFlow Lite format for on-device inference.

This code saves the converted TensorFlow Lite model as **MobilePriceClassifyModel.tflite** and loads it for inference. First, the model is saved as a .tflite file, then loaded into an Interpreter instance, which is responsible for managing the model's execution on-device. The **interpreter.allocate_tensors()** function allocates memory for the model's tensors, preparing it for further processing or inference.

g.Inference Function:

```
initfResult(fResult);
fRes = ml.predict(input1, fResult);
Serial.print("\nThe output value returned for input1 is:\n");
Serial.println(fRes);
displayOutput(fResult);

initfResult(fResult);
fRes = ml.predict(input2, fResult);
Serial.print("\nThe output value returned for input2 is:\n");
Serial.println(fRes);
displayOutput(fResult);
Serial.println();

initfResult(fResult);
fRes = ml.predict(input3, fResult);
Serial.print("\nThe output value returned for input3 is:\n");
Serial.println(fRes);
displayOutput(fResult);

initfResult(fResult);
fRes = ml.predict(input4, fResult);
Serial.print("\nThe output value returned for input4 is:\n");
Serial.println(fRes);
displayOutput(fResult);
```

Figure 8: Inference results for four different input feature sets, displaying predicted outputs for each input using a pre-trained model on the ESP32.

h.Testing:

```
# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print("Test Accuracy:", accuracy)
print("Test Loss:", loss)
```

```
13/13 [==============================] - 0s 1ms/step - loss: 0.0954 - acc: 0.9575
Test Accuracy: 0.9574999809265137
Test Loss: 0.09543399512767792
```

Figure 8: Testing the accuracy of the model

9. Results

Data Output:

a. Python Output:

```
Random input0_data: [[0.99472827 0.12711795 0.30644953 0.9852901  0.03399667 0.4070768
  0.87210685 0.8088839  0.9857608  0.793143   0.26953802 0.11912376
  0.9702564  0.7488302  0.24202262 0.09072242 0.69194186 0.13692406
  0.26782095 0.9166444  0.3228377  0.08546323 0.3780372  0.42000934
  0.39746937 0.9088032 ]]
TFLite Model output for random input0_data: [[0.         0.         0.98046875 0.01953125]]
1/1 [==============================] - 0s 19ms/step
Keras Model output for random input0_data: [[9.3765097e-16 4.8534167e-07 9.8460478e-01 1.5394819e-02]]

Custom input1_data: [[ 1.         0.         0.         1.         1.         0.
   0.         1.         0.         1.         0.         1.
   1.5483596  -1.2362298   0.754832   -0.00893505  0.6201112   1.4257103
  -1.5474459   0.0202687   0.27571163  1.1819974  -0.5804763  -0.5280613
   0.75950885 -1.4451226 ]]
TFLite Model output for custom input1_data: [[0.    0.875 0.125 0.   ]]
1/1 [==============================] - 0s 20ms/step
Keras Model output for custom input1_data: [[7.4067628e-07 8.8681602e-01 1.1318328e-01 4.9287208e-10]]

Custom input2_data: [[ 0.         1.         1.         0.         0.         1.
   0.         1.         0.         1.         0.         1.
  -1.3795348   0.83711153  0.91976255 -1.4036738  -0.9935611  -0.04624633
   1.5103159  -1.4764966  -0.2587623  -0.5565473   1.3364532  -0.29019582
  -1.0848686   0.91592914]]
TFLite Model output for custom input2_data: [[0.         0.         0.59765625 0.40234375]]
1/1 [==============================] - 0s 16ms/step
Keras Model output for custom input2_data: [[2.65441690e-19 1.25709265e-08 5.71884274e-01 4.28115696e-01]]
```

Figure 10: Predicted outputs for input 1 and input 2

b. ESP32 Serial Monitor Output:

```
The output value returned for input1 is:
0.00
0.00 0.88 0.13 0.00
The output value returned for input2 is:
0.00
0.00 0.00 0.60 0.40

The output value returned for input3 is:
0.82
0.82 0.18 0.00 0.00
The output value returned for input4 is:
0.00
0.00 0.00 0.06 0.94
The output value returned for input1 is:
0.00
0.00 0.88 0.13 0.00
The output value returned for input2 is:
0.00
0.00 0.00 0.60 0.40

The output value returned for input3 is:
0.82
0.82 0.18 0.00 0.00
```

Figure 11: Output in Serial Monitor

Performance Notes:

Memory and Resources: The ESP32 performs well within its memory constraints, as the model is optimized to fit within the device's available memory. With a **3.63KB tensor arena size**, the ESP32 is able to perform inferences with minimal resource usage

10. Conclusion

In this project, **a mobile price classification model** was built using TensorFlow, employing a neural network architecture with two dense layers. The model was trained using standardized numerical features and one-hot encoded categorical data, achieving an efficient classification of mobile price ranges. After training, the model was saved and converted to TensorFlow Lite format for optimized deployment on edge devices. This conversion ensures reduced model size and faster inference, making it ideal for deployment on resource-constrained hardware like ESP32. Overall, this approach provides a practical solution for real-time mobile price classification in IoT-based applications.

GitHub Link:
https://github.com/pranavvk18/mobile_price_classification_model_on_ESP32

****