Name: Vibek Rana Magar

Uni ID: 2358119

Simple CNN Implemented using Keras.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
# Load a sample dataset (MNIST for simplicity)
(x_train, y_train), (x_test, y_test) =
keras.datasets.mnist.load_data()
# Normalize and reshape data
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0
x_train = np.expand_dims(x_train, axis=-1) # Add channel dimension
x_test = np.expand_dims(x_test, axis=-1)
# Define a simple CNN model
model = keras.Sequential([
layers.Conv2D(32, (3, 3), activation="relu", input_shape=(28, 28, 1)),
layers.MaxPooling2D((2, 2)),
layers.Conv2D(64, (3, 3), activation="relu"),
layers.MaxPooling2D((2, 2)),
layers.Flatten(),
layers.Dense(128, activation="relu"),
layers.Dense(10, activation="softmax") # 10 classes for MNIST digits
])
# Compile the model
model.compile(optimizer="adam",

loss="sparse_categorical_crossentropy",
metrics=["accuracy"])

# Train the model
model.fit(x_train, y_train, epochs=5, batch_size=32,
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:.4f}")
# Make predictions
predictions = model.predict(x_test[:5])
predicted_labels = np.argmax(predictions, axis=1)
print("Predicted labels:", predicted_labels)
print("Actual labels: ", y_test[:5])
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step

/usr/local/lib/python3.11/dist-packages/keras/src/layers/
convolutional/base_conv.py:107: 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 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 63s 32ms/step - accuracy: 0.9067 -
loss: 0.3004 - val_accuracy: 0.9859 - val_loss: 0.0426
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 58s 31ms/step - accuracy: 0.9861 -
loss: 0.0452 - val_accuracy: 0.9891 - val_loss: 0.0318
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 83s 31ms/step - accuracy: 0.9907 -
loss: 0.0292 - val_accuracy: 0.9903 - val_loss: 0.0323
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 79s 30ms/step - accuracy: 0.9931 -
loss: 0.0212 - val_accuracy: 0.9903 - val_loss: 0.0298
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 83s 31ms/step - accuracy: 0.9951 -
loss: 0.0150 - val_accuracy: 0.9918 - val_loss: 0.0261
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9894 - loss:
0.0316
Test accuracy: 0.9918
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 91ms/step
Predicted labels: [7 2 1 0 4]
Actual labels:  [7 2 1 0 4]
```

Exercise.

Task 1: Data Understanding and Visualization:

• Get the list of class directories from the train folder.

• Select one image randomly from each class.

• Display the images in a grid format with two rows using matplotlib.

```
from google.colab import drive
drive.mount('/content/drive')
```

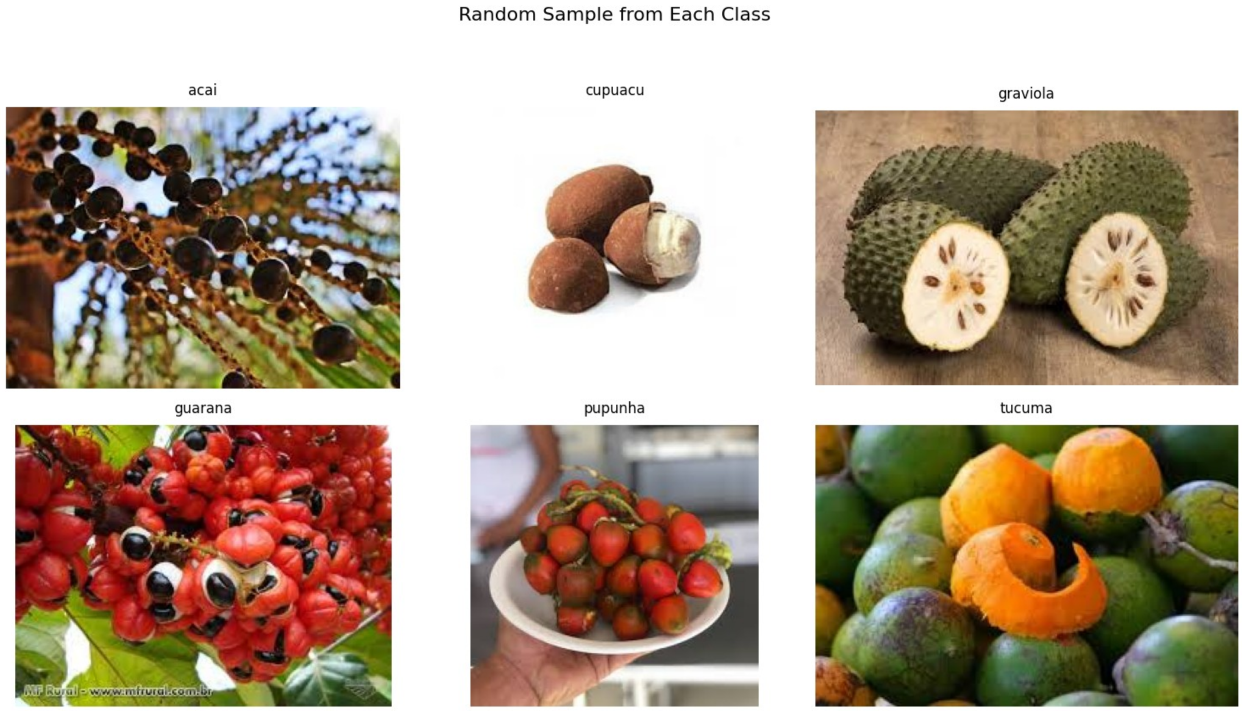
```
Drive already mounted at /content/drive; to attempt to forcibly
remount, call drive.mount("/content/drive", force_remount=True).
```

```python
import os
import random
import matplotlib.pyplot as plt
from PIL import Image

train_path = '/content/drive/MyDrive/AI &
ML/Week_5/FruitinAmazon/train'
test_path = '/content/drive/MyDrive/AI & ML/Week_5/FruitinAmazon/test'

class_dirs = [d for d in os.listdir(train_path)
              if os.path.isdir(os.path.join(train_path, d))]
class_dirs.sort()  # Sort alphabetically for consistent ordering

print("Found classes:", class_dirs)
```

```
Found classes: ['acai', 'cupuacu', 'graviola', 'guarana', 'pupunha',
'tucuma']
```

```python
num_classes = len(class_dirs)
cols = (num_classes + 1) // 2

plt.figure(figsize=(15, 8))
plt.suptitle("Random Sample from Each Class", fontsize=16, y=1.05)

for i, class_name in enumerate(class_dirs):
    # Get all images in the class directory
    class_path = os.path.join(train_path, class_name)
    images = [f for f in os.listdir(class_path)
             if os.path.isfile(os.path.join(class_path, f))]

    # Select a random image
    if images:  # Only proceed if there are images in the directory
        random_image = random.choice(images)
        img_path = os.path.join(class_path, random_image)

        # Open and display the image
        try:
            img = Image.open(img_path)

            # Create subplot
            plt.subplot(2, cols, i+1)
            plt.imshow(img)
            plt.title(class_name, pad=10)
            plt.axis('off')

        except Exception as e:
            print(f"Error loading image {img_path}: {e}")
    else:
        print(f"No images found in class: {class_name}")
```

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



• What did you Observe?

Answer: The output lists six Amazonian fruit classes: acal, cupuacu, graviola, guarana, pupunha, and tucuma. The list provides a clear overview of the different fruit types contained in your dataset's training folder.

2. Check for Corrupted Image:

```
from PIL import ImageFile


def check_and_remove_corrupted_images(directory):
    corrupted_images = []

    for root, _, files in os.walk(directory):
        for file in files:
            file_path = os.path.join(root, file)

            if not file.lower().endswith(('.png', '.jpg', '.jpeg',
'.gif', '.bmp')):
                continue

            try:
                with Image.open(file_path) as img:
                    img.verify()
```

```
                with Image.open(file_path) as img:
                    img.load()
            except (IOError, SyntaxError,
Image.DecompressionBombError) as e:
                print(f"Removed corrupted image: {file_path} - Error:
{str(e)}")
                corrupted_images.append(file_path)
                os.remove(file_path)

    return corrupted_images


corrupted = check_and_remove_corrupted_images(train_path)

if not corrupted:
    print("No corrupted images found.")
else:
    print(f"\nTotal corrupted images removed: {len(corrupted)}")

No corrupted images found.
```

Task 2: Loading and Preprocessing Image Data in keras:

```
import tensorflow as tf

train_dir = '/content/drive/MyDrive/AI &
ML/Week_5/FruitinAmazon/train'
img_height, img_width = 128, 128
batch_size = 32
validation_split = 0.2
seed = 123

train_ds_unmapped =
tf.keras.preprocessing.image_dataset_from_directory(
    train_dir,
    labels='inferred',
    label_mode='int',
    image_size=(img_height, img_width),
    interpolation='nearest',
    batch_size=batch_size,
    shuffle=True,
    validation_split=validation_split,
    subset='training',
    seed=seed
)

val_ds_unmapped = tf.keras.preprocessing.image_dataset_from_directory(
    train_dir,
    labels='inferred',
```

```
    label_mode='int',
    image_size=(img_height, img_width),
    interpolation='nearest',
    batch_size=batch_size,
    shuffle=False,
    validation_split=validation_split,
    subset='validation',
    seed=seed
)

class_names = train_ds_unmapped.class_names
print("Class names:", class_names)

normalization = tf.keras.layers.Rescaling(1./255)
train_ds = train_ds_unmapped.map(lambda x, y: (normalization(x), y))
val_ds = val_ds_unmapped.map(lambda x, y: (normalization(x), y))

for images, labels in train_ds.take(1):
    print("\nFirst training batch:")
    print("Images shape:", images.shape)
    print("Labels shape:", labels.shape)
    print("Pixel value range: ({:.2f}, {:.2f})".format(
        tf.reduce_min(images).numpy(),
        tf.reduce_max(images).numpy()
    ))
```

```
Found 90 files belonging to 6 classes.
Using 72 files for training.
Found 90 files belonging to 6 classes.
Using 18 files for validation.
Class names: ['acai', 'cupuacu', 'graviola', 'guarana', 'pupunha',
'tucuma']

First training batch:
Images shape: (32, 128, 128, 3)
Labels shape: (32,)
Pixel value range: (0.00, 1.00)
```

Task 3 - Implement a CNN with

```
import tensorflow as tf
from tensorflow.keras import layers, models


def create_cnn_model(input_shape=(128, 128, 3), num_classes=6):
    model = models.Sequential([
        layers.Conv2D(32, (3, 3), strides=1, padding='same',
activation='relu', input_shape=input_shape),
        layers.MaxPooling2D((2, 2), strides=2),
```

```
        layers.Conv2D(32, (3, 3), strides=1, padding='same',
activation='relu'),
        layers.MaxPooling2D((2, 2), strides=2),

        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        layers.Dense(128, activation='relu'),
        layers.Dense(num_classes, activation='softmax')
    ])
    return model

model = create_cnn_model(input_shape=(img_height, img_width, 3),
num_classes=len(class_names))

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

model.summary()
```

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 128, 128, 32)	896
max_pooling2d_2 (MaxPooling2D)	(None, 64, 64, 32)	0
conv2d_3 (Conv2D)	(None, 64, 64, 32)	9,248
max_pooling2d_3 (MaxPooling2D)	(None, 32, 32, 32)	0
flatten_1 (Flatten)	(None, 32768)	0
dense_2 (Dense)	(None, 64)	2,097,216

```
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ dense_3 (Dense)                      │ (None, 128)                 │
8,320 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ dense_4 (Dense)                      │ (None, 6)                   │
774 │
└──────────────────────────────────────┴─────────────────────────────┴
─────────────────┘

 Total params: 2,116,454 (8.07 MB)

 Trainable params: 2,116,454 (8.07 MB)

 Non-trainable params: 0 (0.00 B)
```

Task 4: Compile the Model

```
# Compile the model with recommended settings
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

extra_metrics = [
    tf.keras.metrics.SparseTopKCategoricalAccuracy(k=2,
name='top2_accuracy'),
    tf.keras.metrics.SparseCategoricalCrossentropy(name='xentropy')
]

# Verify compilation
print("Model successfully compiled!")
print("Optimizer:", model.optimizer.get_config()['name'])
print("Loss function:", model.loss)
print("Metrics:", [m.name for m in model.metrics])

Model successfully compiled!
Optimizer: adam
Loss function: sparse_categorical_crossentropy
Metrics: ['loss', 'compile_metrics']
```

Task 4: Train the Model

```
import numpy as np
from sklearn.metrics import classification_report

callbacks = [
    tf.keras.callbacks.ModelCheckpoint(
```

```
        filepath='best_model.h5',
        monitor='val_accuracy',
        save_best_only=True,
        mode='max',
        verbose=1
    ),
    tf.keras.callbacks.EarlyStopping(
        monitor='val_loss',
        patience=15,
        restore_best_weights=True,
        verbose=1
    )
]

# Train the model
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=250,
    batch_size=16,
    callbacks=callbacks,
    verbose=1
)

# Plot training history
plt.figure(figsize=(12, 5))

# Accuracy plot
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Val Accuracy')
plt.title('Accuracy over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

# Loss plot
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.title('Loss over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

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

```
Epoch 1/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 388ms/step - accuracy: 0.1447 - loss:
```

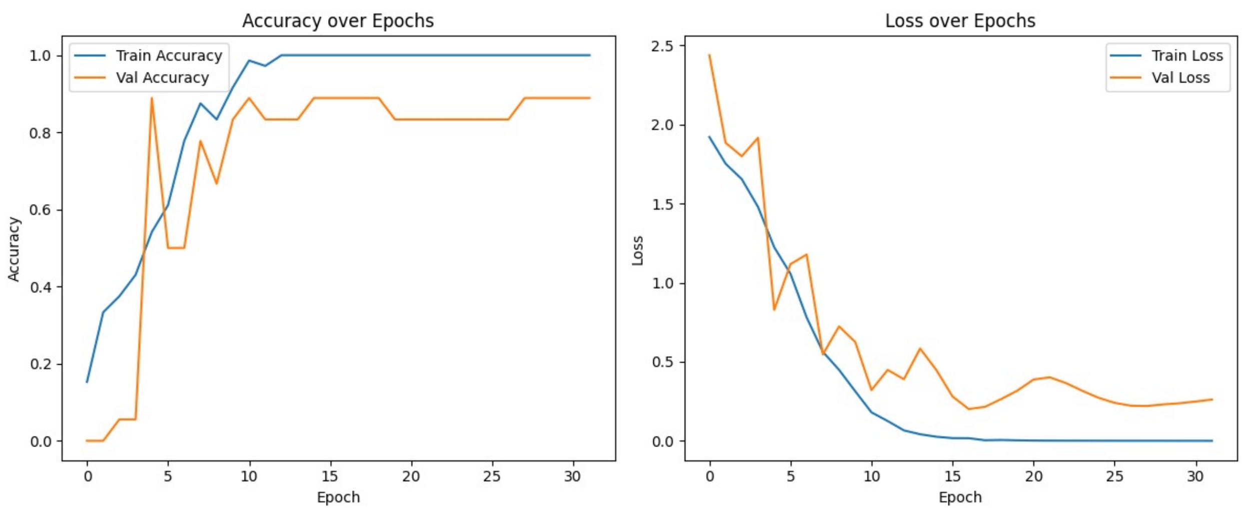
```
1.8700
Epoch 1: val_accuracy improved from -inf to 0.00000, saving model to
best_model.h5

WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.

 3/3 ━━━━━━━━━━━━━━━━━━━━ 4s 679ms/step - accuracy: 0.1467 - loss:
1.8827 - val_accuracy: 0.0000e+00 - val_loss: 2.4377
Epoch 2/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 597ms/step - accuracy: 0.3455 - loss:
1.7822
Epoch 2: val_accuracy did not improve from 0.00000
3/3 ━━━━━━━━━━━━━━━━━━━━ 3s 781ms/step - accuracy: 0.3424 - loss:
1.7746 - val_accuracy: 0.0000e+00 - val_loss: 1.8843
Epoch 3/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 581ms/step - accuracy: 0.3698 - loss:
1.6617
Epoch 3: val_accuracy improved from 0.00000 to 0.05556, saving model
to best_model.h5

WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.

 3/3 ━━━━━━━━━━━━━━━━━━━━ 3s 725ms/step - accuracy: 0.3711 - loss:
1.6598 - val_accuracy: 0.0556 - val_loss: 1.7988
Epoch 4/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 344ms/step - accuracy: 0.4300 - loss:
1.5145
Epoch 4: val_accuracy did not improve from 0.05556
3/3 ━━━━━━━━━━━━━━━━━━━━ 2s 512ms/step - accuracy: 0.4301 - loss:
1.5056 - val_accuracy: 0.0556 - val_loss: 1.9156
Epoch 5/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 352ms/step - accuracy: 0.4931 - loss:
1.2770
Epoch 5: val_accuracy improved from 0.05556 to 0.88889, saving model
to best_model.h5

WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.
```

```
 3/3 ━━━━━━━━━━━━━━━━━━━━ 2s 561ms/step - accuracy: 0.5052 - loss:
1.2637 - val_accuracy: 0.8889 - val_loss: 0.8277
Epoch 6/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 357ms/step - accuracy: 0.6204 - loss:
1.0499
Epoch 6: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 2s 444ms/step - accuracy: 0.6181 - loss:
1.0517 - val_accuracy: 0.5000 - val_loss: 1.1168
Epoch 7/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 355ms/step - accuracy: 0.8113 - loss:
0.7453
Epoch 7: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 2s 466ms/step - accuracy: 0.8030 - loss:
0.7540 - val_accuracy: 0.5000 - val_loss: 1.1779
Epoch 8/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 365ms/step - accuracy: 0.8542 - loss:
0.5906
Epoch 8: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 2s 467ms/step - accuracy: 0.8594 - loss:
0.5837 - val_accuracy: 0.7778 - val_loss: 0.5458
Epoch 9/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 640ms/step - accuracy: 0.8299 - loss:
0.4623
Epoch 9: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 4s 802ms/step - accuracy: 0.8307 - loss:
0.4590 - val_accuracy: 0.6667 - val_loss: 0.7235
Epoch 10/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 348ms/step - accuracy: 0.9306 - loss:
0.2943
Epoch 10: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 2s 516ms/step - accuracy: 0.9271 - loss:
0.2991 - val_accuracy: 0.8333 - val_loss: 0.6247
Epoch 11/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 356ms/step - accuracy: 0.9954 - loss:
0.1803
Epoch 11: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 2s 465ms/step - accuracy: 0.9931 - loss:
0.1802 - val_accuracy: 0.8889 - val_loss: 0.3210
Epoch 12/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 356ms/step - accuracy: 0.9699 - loss:
0.1328
Epoch 12: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 3s 524ms/step - accuracy: 0.9705 - loss:
0.1309 - val_accuracy: 0.8333 - val_loss: 0.4485
Epoch 13/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 362ms/step - accuracy: 1.0000 - loss:
0.0717
Epoch 13: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 2s 530ms/step - accuracy: 1.0000 - loss:
0.0704 - val_accuracy: 0.8333 - val_loss: 0.3896
```

```
Epoch 14/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 355ms/step - accuracy: 1.0000 - loss:
0.0400
Epoch 14: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 3s 528ms/step - accuracy: 1.0000 - loss:
0.0405 - val_accuracy: 0.8333 - val_loss: 0.5836
Epoch 15/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 573ms/step - accuracy: 1.0000 - loss:
0.0313
Epoch 15: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 3s 749ms/step - accuracy: 1.0000 - loss:
0.0300 - val_accuracy: 0.8889 - val_loss: 0.4481
Epoch 16/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 546ms/step - accuracy: 1.0000 - loss:
0.0174
Epoch 16: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 4s 688ms/step - accuracy: 1.0000 - loss:
0.0174 - val_accuracy: 0.8889 - val_loss: 0.2806
Epoch 17/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 633ms/step - accuracy: 1.0000 - loss:
0.0227
Epoch 17: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 3s 810ms/step - accuracy: 1.0000 - loss:
0.0213 - val_accuracy: 0.8889 - val_loss: 0.2010
Epoch 18/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 366ms/step - accuracy: 1.0000 - loss:
0.0039
Epoch 18: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 2s 464ms/step - accuracy: 1.0000 - loss:
0.0039 - val_accuracy: 0.8889 - val_loss: 0.2151
Epoch 19/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 356ms/step - accuracy: 1.0000 - loss:
0.0051
Epoch 19: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 2s 491ms/step - accuracy: 1.0000 - loss:
0.0053 - val_accuracy: 0.8889 - val_loss: 0.2641
Epoch 20/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 629ms/step - accuracy: 1.0000 - loss:
0.0033
Epoch 20: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 3s 799ms/step - accuracy: 1.0000 - loss:
0.0033 - val_accuracy: 0.8333 - val_loss: 0.3176
Epoch 21/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 356ms/step - accuracy: 1.0000 - loss:
0.0016
Epoch 21: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 4s 451ms/step - accuracy: 1.0000 - loss:
0.0017 - val_accuracy: 0.8333 - val_loss: 0.3877
Epoch 22/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 362ms/step - accuracy: 1.0000 - loss:
```

```
0.0013
Epoch 22: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 2s 449ms/step - accuracy: 1.0000 - loss:
0.0013 - val_accuracy: 0.8333 - val_loss: 0.4016
Epoch 23/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 357ms/step - accuracy: 1.0000 - loss:
0.0011
Epoch 23: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 2s 526ms/step - accuracy: 1.0000 - loss:
0.0011 - val_accuracy: 0.8333 - val_loss: 0.3652
Epoch 24/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 347ms/step - accuracy: 1.0000 - loss:
0.0011
Epoch 24: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 2s 514ms/step - accuracy: 1.0000 - loss:
0.0010 - val_accuracy: 0.8333 - val_loss: 0.3168
Epoch 25/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 630ms/step - accuracy: 1.0000 - loss:
8.0754e-04
Epoch 25: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 3s 807ms/step - accuracy: 1.0000 - loss:
8.1007e-04 - val_accuracy: 0.8333 - val_loss: 0.2725
Epoch 26/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 599ms/step - accuracy: 1.0000 - loss:
8.1341e-04
Epoch 26: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 3s 706ms/step - accuracy: 1.0000 - loss:
7.9022e-04 - val_accuracy: 0.8333 - val_loss: 0.2403
Epoch 27/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 357ms/step - accuracy: 1.0000 - loss:
6.7833e-04
Epoch 27: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 2s 453ms/step - accuracy: 1.0000 - loss:
6.6379e-04 - val_accuracy: 0.8333 - val_loss: 0.2223
Epoch 28/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 355ms/step - accuracy: 1.0000 - loss:
5.3098e-04
Epoch 28: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 3s 441ms/step - accuracy: 1.0000 - loss:
5.3126e-04 - val_accuracy: 0.8889 - val_loss: 0.2204
Epoch 29/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 359ms/step - accuracy: 1.0000 - loss:
5.0068e-04
Epoch 29: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 3s 527ms/step - accuracy: 1.0000 - loss:
4.9511e-04 - val_accuracy: 0.8889 - val_loss: 0.2305
Epoch 30/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 355ms/step - accuracy: 1.0000 - loss:
4.1161e-04
Epoch 30: val_accuracy did not improve from 0.88889
```

```
3/3 ━━━━━━━━━━━━━━━━━━━━ 2s 523ms/step - accuracy: 1.0000 - loss:
4.1415e-04 - val_accuracy: 0.8889 - val_loss: 0.2378
Epoch 31/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 468ms/step - accuracy: 1.0000 - loss:
4.0001e-04
Epoch 31: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 2s 804ms/step - accuracy: 1.0000 - loss:
3.9522e-04 - val_accuracy: 0.8889 - val_loss: 0.2487
Epoch 32/250
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 1.0000 - loss:
3.3574e-04
Epoch 32: val_accuracy did not improve from 0.88889
3/3 ━━━━━━━━━━━━━━━━━━━━ 5s 2s/step - accuracy: 1.0000 - loss:
3.3944e-04 - val_accuracy: 0.8889 - val_loss: 0.2611
Epoch 32: early stopping
Restoring model weights from the end of the best epoch: 17.
```



Task 5: Evaluate the Model

```
# Load test dataset
test_dir = '/content/drive/MyDrive/AI & ML/Week_5/FruitinAmazon/test'
test_ds = tf.keras.preprocessing.image_dataset_from_directory(
    test_dir,
    image_size=(img_height, img_width),
    batch_size=batch_size,
    label_mode='int'
).map(lambda x, y: (normalization(x), y))

# Evaluate on test set
test_loss, test_acc = model.evaluate(test_ds)
print(f'\nTest Accuracy: {test_acc:.4f}')
print(f'Test Loss: {test_loss:.4f}')
```

```
Found 30 files belonging to 6 classes.
1/1 ━━━━━━━━━━━━━━━━━━━━ 4s 4s/step - accuracy: 0.8000 - loss: 0.9398

Test Accuracy: 0.8000
Test Loss: 0.9398
```

Task 6: Save and Load the Model

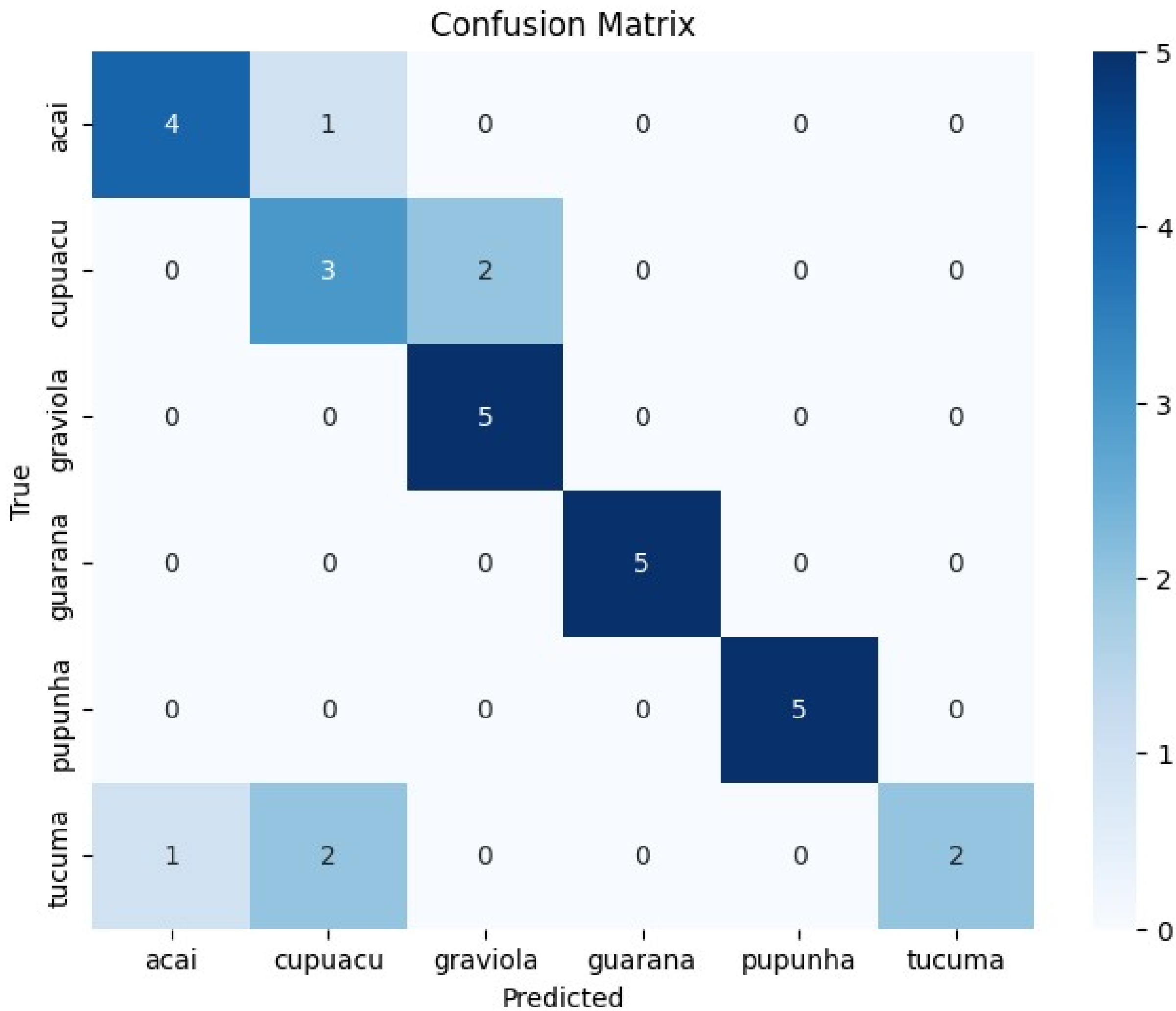
```
model.save('fruit_classifier.h5')

# Load the saved model
loaded_model = tf.keras.models.load_model('fruit_classifier.h5')

# Verify loaded model
loaded_loss, loaded_acc = loaded_model.evaluate(test_ds)
print(f'\nLoaded Model Test Accuracy: {loaded_acc:.4f}')
print(f'Loaded Model Test Loss: {loaded_loss:.4f}')

WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.
WARNING:absl:Compiled the loaded model, but the compiled metrics have
yet to be built. `model.compile_metrics` will be empty until you train
or evaluate the model.

1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 986ms/step - accuracy: 0.8000 - loss:
0.9398

Loaded Model Test Accuracy: 0.8000
Loaded Model Test Loss: 0.9398
```

Task 7: Predictions and Classification Report

```
import numpy as np
from sklearn.metrics import classification_report

y_true = []
y_pred = []

for images, labels in test_ds:
    y_true.extend(labels.numpy())
    y_pred.extend(np.argmax(loaded_model.predict(images), axis=1))

# Classification report
print('\nClassification Report:')
print(classification_report(
    y_true,
    y_pred,
```

```
    target_names=class_names
))

# Confusion matrix visualization
from sklearn.metrics import confusion_matrix
import seaborn as sns

cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_names,
            yticklabels=class_names)
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 523ms/step

Classification Report:
              precision    recall  f1-score   support

        acai       0.80      0.80      0.80         5
     cupuacu       0.50      0.60      0.55         5
    graviola       0.71      1.00      0.83         5
     guarana       1.00      1.00      1.00         5
     pupunha       1.00      1.00      1.00         5
      tucuma       1.00      0.40      0.57         5

    accuracy                           0.80        30
   macro avg       0.84      0.80      0.79        30
weighted avg       0.84      0.80      0.79        30
```


Confusion Matrix
True
acai
cupuacu
graviola
guarana
pupunha
tucuma
Predicted
acai
cupuacu
graviola
guarana
pupunha
tucuma
4 1 0 0 0 0
0 3 2 0 0 0
0 0 5 0 0 0
0 0 0 5 0 0
0 0 0 0 5 0
1 2 0 0 0 2
0
1
2
3
4
5