6CS012 - Artificial Intelligence and Machine Learning. Implementation of Convolutional Neural Network using Keras.

Prepared By: Siman Giri {Module Leader - 6CS012}

March 23, 2025

——————— Worksheet - 5. ———————-

1 Instructions

This exercise sheet will help you understand and implement Convolutional Neural Networks from scratch using keras.

- This worksheet must be completed individually.
- All the solutions must be written in Jupyter Notebook {Preferred}.

Learning Objectives:

By the end of this tutorial, learners should be able to effectively train, evaluate, save, and reuse CNN models, providing a solid foundation for real-world machine learning projects.

- **Understand Model Compilation and Training:**
- **Evaluate and Test Model Performance:**
- **Make Predictions and Interpret Results:**

2 Building an End-to-End Image Classifier with CNNs.

Convolutional Neural Network has two important section:

1. **Convolutional Layer:**
2. **Fully Connected Layer:**

3 Understanding Convolutional Layer.

A convolutional layer is a core building block of Convolutional Neural Networks (CNNs). It applies convolution operations to extract spatial features from input images or tensors. In Keras, convolutional layers are implemented using the Conv2D class.

1. Conv2D Class in Keras:

The Conv2D class in Keras is used to create a **convolutional layer** in a neural network. Each neuron in a Conv2D layer applies a set of filters to local patches of the input data, capturing spatial hierarchies of features.

Syntax of Conv2D Layer.

```python
from tensorflow.keras.layers import Conv2D
layer = Conv2D(filters, kernel_size, strides=(1,1), padding="valid", activation=None, use_bias=True,
    kernel_initializer="glorot_uniform")
```

Main Arguments:

Argument	Description
`filters`	Number of filters (feature detectors) in the layer.
`kernel_size`	Size of the convolutional kernel (e.g., (3,3) or (5,5)).
`strides`	Step size for traversing input data. Default: (1,1).
`padding`	Defines padding type: `"valid"` (no padding) or `"same"` (zero-padding).
`activation`	Activation function applied to the output. Default: `None` (linear activation).
`use_bias`	Whether to include a bias term. Default: `True`.
`kernel_initializer`	Method to initialize the filter weights. Default: `"glorot_uniform"`.
`bias_initializer`	Method to initialize the bias. Default: `"zeros"`.
`kernel_regularizer`	Regularization applied to the filter weights (e.g., L1, L2).
`bias_regularizer`	Regularization applied to the bias.

Table 1: Arguments of the `Conv2D` Layer in Keras

2. How Does Conv2D Work?

Each neuron in a Conv2D layer computes:

$$Y = \text{activation}(W * X + b) \tag{1}$$

where:

- = Input data (image or feature map).
- = Set of learnable filters.
- = Convolution operation.
- = Bias term (optional).
- activation = Activation function applied to the output.

3. Pooling Layers in Keras

Pooling layers reduce the spatial dimensions of feature maps while retaining essential information. The most common types are max pooling and average pooling, implemented using the MaxPooling2D and AveragePooling2D classes in Keras.

Syntax of Pooling Layers.

```
from tensorflow.keras.layers import MaxPooling2D, AveragePooling2D
max_pool = MaxPooling2D(pool_size=(2,2), strides=None, padding="valid")
avg_pool = AveragePooling2D(pool_size=(2,2), strides=None, padding="valid")
```

Main Arguments:

Argument	**Description**
`pool_size`	Size of the pooling window (e.g., (2,2) for 2x2 pooling).
`strides`	Step size for sliding the pooling window. Default: Same as `pool_size`.
`padding`	Defines padding type: `"valid"` (no padding) or `"same"` (zero-padding).

Table 2: Arguments of the Pooling Layers in Keras

4. How Does Pooling Work?

Pooling layers downsample feature maps to reduce computational complexity and control overfitting.

- **Max Pooling:** Selects the maximum value within each pooling window.
- **Average Pooling:** Computes the average of all values in each pooling window.

The pooling operation can be expressed as:

$$Y_{i,j} = \max(X_{m,n}) \quad \text{or} \quad Y_{i,j} = \frac{1}{N} \sum X_{m,n} \tag{2}$$

where are the elements in the pooling window and is the number of elements in the window.

4 Training a Convolutional Neural Network.

Convolutional Neural Network are trained using forward and backward propagation.

1. Forward Propagation:

Forward propagation refers to the process in which the input data is passed through the network layer by layer until the final output layer. In this phase:

1. **Input Data Processing:**
   - The input data is an image's matrix representation (aka image tensor), which is passed through the network.
   - For each layer, the input tensor is processed by applying convolution, pooling and dense operations.
   - Each layer performs mathematical operations:
     - Convolutional + activation (ReLU) at Convolutional layer.
     - Summation $\sum \mathbf{W}\mathbf{X} + \mathbf{b}$ + activation (ReLU) at Dense layer.
     - Summation $\sum \mathbf{W}\mathbf{h}^{\mathbf{l}} + \mathbf{b}$ + softmax activation at Output layer.
2. **Final Output:**
   - The final output layer generates the predictions, which is a probability distribution in classification task.
   - The Output is compared to the true labels using a loss function (Categorical Cross Entropy). This comparison quantifies how well the model's prediction matches the true values.
   - The final error or cost is computed as average of loss function for all the data.

The final error has to be propagated back to the first weight via all the in between layers.

2. Backward Propagation:

Backward propagation is the method used to update the model's weights to minimize the loss during training. This process begins after the forward pass and occurs during the training phase:

1. **Loss Calculation:** The loss function computes the difference between the model's predictions and the actual target values (ground truth).{we also discuss this in above section.}

2. **Gradient Calculation:** The gradients of the loss with respect to the model's weights are computed by applying the chain rule of calculus. This means that for each weight in the network, we calculate how much it contributed to the loss.Gradients are calculated layer by layer starting from the output layer fo fully connected layer:

   - **Gradients at Fully Connected Layer:**

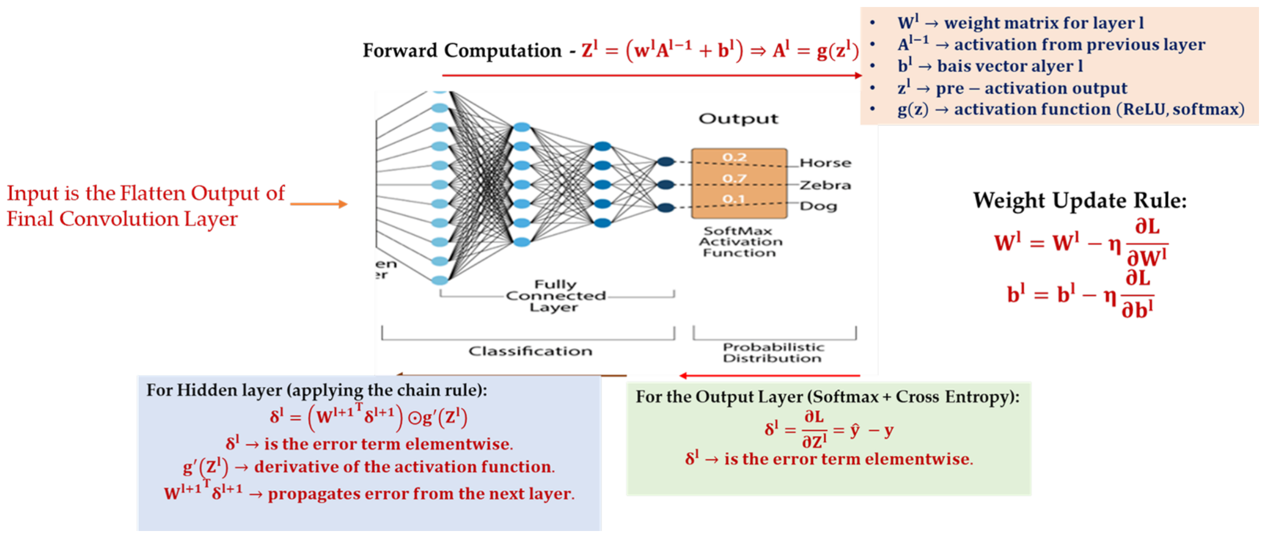

Forward Computation - $Z^l = (w^l A^{l-1} + b^l) \Rightarrow A^l = g(z^l)$
$W^l \rightarrow$ weight matrix for layer l
$A^{l-1} \rightarrow$ activation from previous layer
$b^l \rightarrow$ bais vector alyer l
$z^l \rightarrow$ pre – activation output
$g(z) \rightarrow$ activation function (ReLU, softmax)
Output
0.2
0.7
0.1
Horse
Zebra
Dog
SoftMax Activation Function
Input is the Flatten Output of Final Convolution Layer
Fully Connected Layer
Classification
Probabilistic Distribution
Weight Update Rule:
$W^l = W^l - \eta \frac{\partial L}{\partial W^l}$
$b^l = b^l - \eta \frac{\partial L}{\partial b^l}$
For Hidden layer (applying the chain rule):
$\delta^l = (W^{l+1^T} \delta^{l+1}) \odot g'(Z^l)$
$\delta^l \rightarrow$ is the error term elementwise.
$g'(Z^l) \rightarrow$ derivative of the activation function.
$W^{l+1^T} \delta^{l+1} \rightarrow$ propagates error from the next layer.
For the Output Layer (Softmax + Cross Entropy):
$\delta^l = \frac{\partial L}{\partial Z^l} = \hat{y} - y$
$\delta^l \rightarrow$ is the error term elementwise.

Figure 1: Forward and Backward Computation at Fully Connected Network.

– **Output Layer of Fully Connected Layer:**

$$\delta^l = \frac{\partial L}{\partial z^l} \\ = \hat{y} - y$$

Where: δ^l is the error term element-wise.

– **For Hidden Layer applying the chain rule:**

$$\delta^{\mathbf{l}} = (\mathbf{W}_{\mathbf{l+1}}^{\mathbf{T}} \cdot \delta_{\mathbf{l+1}}) \odot \mathbf{a}'(\mathbf{z})^{\mathbf{l}}$$

Where:

* $*^{\mathbf{l}} \rightarrow$ `Computation at the layer` $\mathbf{l}$
* $\delta^{\mathbf{l}} \rightarrow$ `is the gradient for the layer` $\mathbf{l}$
* $\mathbf{a}'(\mathbf{z}^{\mathbf{l}}) \rightarrow$ `derivative of the activation function at layer` $\mathbf{l}$
* $(\mathbf{W}_{\mathbf{l+1}}^{\mathbf{T}} \cdot \delta_{\mathbf{l+1}}) \rightarrow$ `propagates the error from previous layer` $\mathbf{l+1}$

- **Gradients at Convolutional Layer:**
  - Before propagating gradients from the Fully Connected Network (FCN) back to the Convolutional layer, the gradient at the input layer of the FCN (which is a flattened vector) must be reshaped back into the matrix (or tensor) form that corresponds to the dimensions of the final feature map outputted by the last convolutional layer.

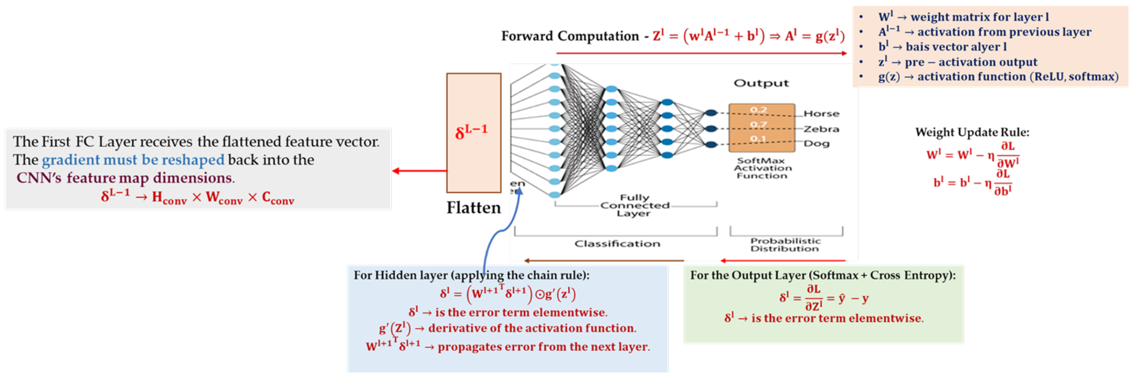

Forward Computation - $Z^l = (w^l A^{l-1} + b^l) \Rightarrow A^l = g(z^l)$
$W^l \rightarrow$ weight matrix for layer l
$A^{l-1} \rightarrow$ activation from previous layer
$b^l \rightarrow$ bais vector alyer l
$z^l \rightarrow$ pre – activation output
$g(z) \rightarrow$ activation function (ReLU, softmax)
The First FC Layer receives the flattened feature vector. The gradient must be reshaped back into the CNN's feature map dimensions.
$\delta^{L-1} \rightarrow H_{conv} \times W_{conv} \times C_{conv}$
δ^{L-1}
Flatten
Output
0.2
0.7
0.1
Horse
Zebra
Dog
SoftMax Activation Function
Fully Connected Layer
Classification
Probabilistic Distribution
Weight Update Rule:
$W^l = W^l - \eta \frac{\partial L}{\partial W^l}$
$b^l = b^l - \eta \frac{\partial L}{\partial b^l}$
For Hidden layer (applying the chain rule):
$\delta^l = \left(W^{l+1^T} \delta^{l+1}\right) \odot g'(z^l)$
$\delta^l \rightarrow$ is the error term elementwise.
$g'(Z^l) \rightarrow$ derivative of the activation function.
$W^{l+1^T} \delta^{l+1} \rightarrow$ propagates error from the next layer.
For the Output Layer (Softmax + Cross Entropy):
$\delta^l = \frac{\partial L}{\partial Z^l} = \hat{y} - y$
$\delta^l \rightarrow$ is the error term elementwise.

Figure 2: Start of Backpropagation in Convolutional Layer.

  - In the backward pass of a convolutional layer: we perform two main steps:
    * **Computing the Gradient w.r.t the filter F:**
      · This tells us how much each filter contributes to the loss.
      · Calculated as convolution between input X and the gradient of the error $\delta = \frac{\partial E}{\partial O}$.

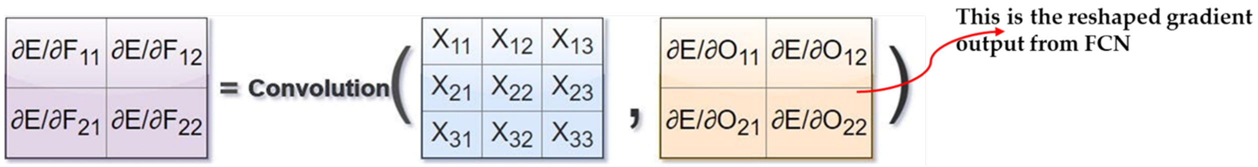

$\partial E/\partial F_{11}$
$\partial E/\partial F_{12}$
$\partial E/\partial F_{21}$
$\partial E/\partial F_{22}$
= Convolution
X_{11} X_{12} X_{13}
X_{21} X_{22} X_{23}
X_{31} X_{32} X_{33}
$\partial E/\partial O_{11}$
$\partial E/\partial O_{12}$
$\partial E/\partial O_{21}$
$\partial E/\partial O_{22}$
This is the reshaped gradient output from FCN

Figure 3: Gradient with respect to Filter:

    * **Computing the gradient w.r.t the input X:**
      · This tells us how much pixel in the input contributes to the loss.
      · Calculated as a convolution between flipped filter F and the gradient of the error $\delta = \frac{\partial E}{\partial O}$.

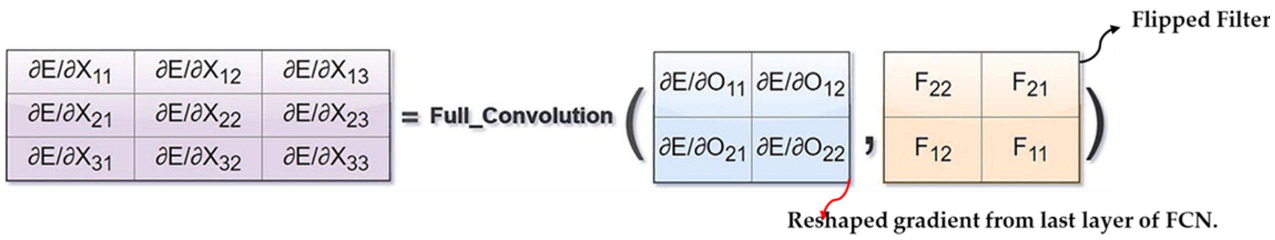

$\partial E/\partial X_{11}$ $\partial E/\partial X_{12}$ $\partial E/\partial X_{13}$
$\partial E/\partial X_{21}$ $\partial E/\partial X_{22}$ $\partial E/\partial X_{23}$
$\partial E/\partial X_{31}$ $\partial E/\partial X_{32}$ $\partial E/\partial X_{33}$
= Full_Convolution
$\partial E/\partial O_{11}$ $\partial E/\partial O_{12}$
$\partial E/\partial O_{21}$ $\partial E/\partial O_{22}$
F_{22} F_{21}
F_{12} F_{11}
Flipped Filter
Reshaped gradient from last layer of FCN.

Figure 4: Gradient with respect to Input.

3. **Gradient Descent:** The computed gradients are used to update weights of the network, This is done by:

$$\mathbf{w^l} = \mathbf{w^l} - \eta\mathbf{\Delta w}$$

$$\mathbf{b^l} = \mathbf{b^l} - \eta\mathbf{\Delta b}$$

   - The fraction by which the weights are updated is controlled by a hyper - parameter called learning rate denoted by η.

5 How Keras Handles Forward and Backward Propagation?

In Keras, forward and backward propagation are automatically managed when you call the `fit()` method. The model's architecture (composed of layers like `Conv2D`, `Dense`, etc.) is defined first, and then the `compile()` and `fit()` methods are called to start the training process.

Here is how Keras handles this internally:

1. Model Compilation:

- The `compile()` method configures the model for training. It specifies the optimizer (e.g., Adam or SGD), the loss function (e.g., sparse categorical cross-entropy), and the metrics to track (e.g., accuracy).
- Keras will then prepare the model for the forward and backward propagation steps.

Model Compilation.

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

2. Model Training:

- The `fit()` method trains the model by iterating through the training data. For each batch of data, Keras performs forward propagation to compute the predictions and loss, followed by backward propagation to update the model's weights.

Training the Model.

```
model.fit(train_ds, epochs=10, validation_data=val_ds)
```

During each epoch, Keras automatically:

- Performs the forward pass by processing the input data through the network.
- Computes the loss by comparing the output to the ground truth.
- Back-propagates the error to compute gradients.
- Updates the model's weights using the optimizer.

This iterative process continues for multiple epochs, with the model gradually improving its predictions as it learns from the data.

6 Simple CNN Implemented using Keras.

Here's is a simple Convolutional Neural Network implemented using keras. The code includes model creation, compilation training (fitting), evaluation, and prediction.

End - to - End CNN Model.

```python
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])
```

Breaking Down CNN's each layer:

1. **Input Layer:**

```python
input_shape=(28, 28, 1)
```

   - The input consists of grayscale images of $28 \times 28 \times 1$.
   - This is the expected input format for the first convolutional layer.

2. **First Convolutional Layer:**

```python
layers.Conv2D(32, (3, 3), activation="relu", input_shape=(28, 28, 1)),
```

- Applies **32** filters of size 3×3 to detect patterns in the input images.
- Uses **ReLU** activation, which helps non linearity and avoids vanishing gradients.
- The output feature map has **32** channels.
- Output shape: We applied "valid" padding the output size is:

$$\mathbf{\frac{28-3}{1} + 1 = 26}$$

Thus the final output shape is: $\mathbf{(26, 26, 32)}$.

3. **First Pooling Layer:**

```
layers.MaxPooling2D((2, 2)),
```

- Performs **max pooling** with $\mathbf{2} \times \mathbf{2}$ filter and Stride $= 2$, reducing spatial dimensions by half.
- Helps reduce computation and extract dominant features.
- Output Shape:

$$\mathbf{\frac{26-2}{2} + 1 = 13}$$

Thus the output shape is: $\mathbf{13, 13, 32}$.

4. **Second Convolutional Layer:**

```
layers.Conv2D(32, (3, 3), activation="relu", input_shape=(28, 28, 1)),
```

- Applies **64** filters of size 3×3 to detect patterns in the input images.
- Uses **ReLU** activation, which helps non linearity and avoids vanishing gradients.
- The output feature map has **64** channels, enabling the model to learn more complex features.
- Output shape: We applied "valid" padding the output size is:

$$\mathbf{\frac{13-3}{1} + 1 = 11}$$

Thus the final output shape is: $\mathbf{(11, 11, 64)}$.

5. **Second Pooling Layer:**

```
layers.MaxPooling2D((2, 2)),
```

- Another $\mathbf{2} \times \mathbf{2}$ max pooling, reducing the spatial size by half.
- Output Shape:

$$\mathbf{\frac{11}{2} = 5\{5.5 \rightarrow 5\}}$$

Thus the output shape is: $\mathbf{5, 5, 64}$.

6. **Flatten Layer:**

```
layers.Flatten(),
```

- Converts the **3D feature maps** ($\mathbf{5 \times 5 \times 64}$) into a vector 1D vector of 1600 elements for the fully connected dense layer.
- The output size is:
  $$5 \times 5 \times 64 = 1600$$
  So, the output shape is $(\mathbf{1600},)$

7. **Fully Connected Dense Layer:**

```
        layers.Dense(128, activation="relu"),
```

- Fully Connected layer with **128** neurons.
- Uses **ReLU** activation for non - linearity.
- Extracts high - level features from the previous layer.
- The output shape: $(\mathbf{128},)$.

8. **Output Layer:**

```
        layers.Dense(10, activation="softmax") # 10 classes for MNIST digits
```

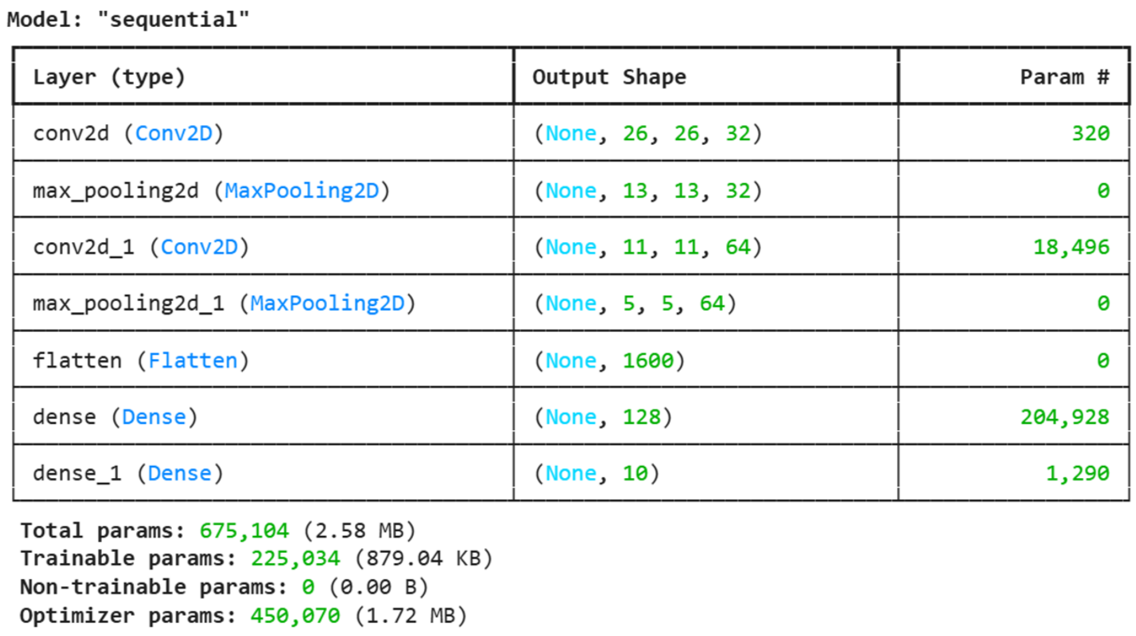
- A Dense layer with 10 neurons, one for each digit class (0 - 9).
- Uses Softmax activation, which converts the output into probabilities.
- The Output shape: $(\mathbf{10},)$

9. **Computing Total Number of Parameters:**

Layer	**Output Shape**	**Parameters**
Conv2D (32, 3×3)	(26, 26, 32)	320
MaxPooling2D (2×2)	(13, 13, 32)	0
Conv2D (64, 3×3)	(11, 11, 64)	18,496
MaxPooling2D (2×2)	(5, 5, 64)	0
Flatten	(1600,)	0
Dense (128)	(128,)	204,928
Dense (10)	(10,)	1,290
Total Parameters	-	**225,034**

Table 3: Summary of CNN Model Layers and the Parameters at each Layer.

Final Model Summary:



```
Model: "sequential"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                        ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ conv2d (Conv2D)                     │ (None, 26, 26, 32)          │             320 │
├─────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ max_pooling2d (MaxPooling2D)        │ (None, 13, 13, 32)          │               0 │
├─────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ conv2d_1 (Conv2D)                   │ (None, 11, 11, 64)          │          18,496 │
├─────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ max_pooling2d_1 (MaxPooling2D)      │ (None, 5, 5, 64)            │               0 │
├─────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ flatten (Flatten)                   │ (None, 1600)                │               0 │
├─────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense (Dense)                       │ (None, 128)                 │         204,928 │
├─────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_1 (Dense)                     │ (None, 10)                  │           1,290 │
└─────────────────────────────────────┴─────────────────────────────┴─────────────────┘
 Total params: 675,104 (2.58 MB)
 Trainable params: 225,034 (879.04 KB)
 Non-trainable params: 0 (0.00 B)
 Optimizer params: 450,070 (1.72 MB)
```

Figure 5: Model Summary Using model.summary()

7 Exercise.

Implement an End to End CNN Model for Image Classification Task.

Objective

In this exercise, you will build and train a Convolutional Neural Network to classify fruits in Amazon using TensorFlow and Keras.

Task 1: Data Understanding and Visualization:

Download the Provided Data and Complete the following task.

1. **Load and visualize images from a dataset stored in directories, where each subdirectory represents a class.** You are expected to write a code:
   - 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`.
   - **Expected Output:**

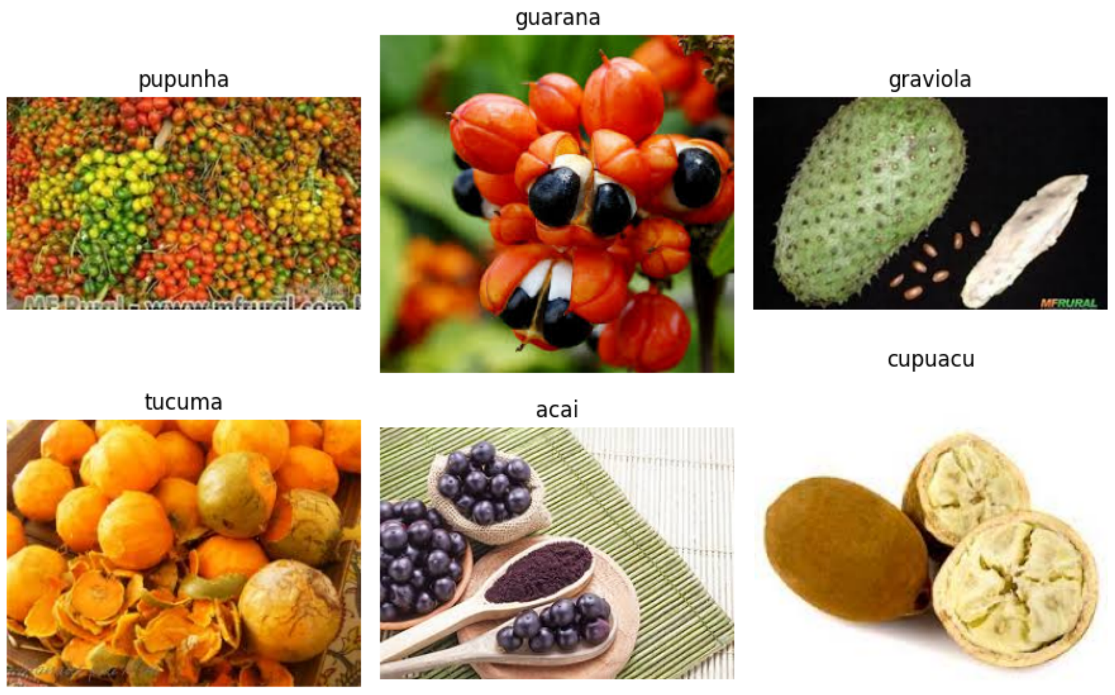


Figure 6: Expected Output.

   - What did you Observe?

2. **Check for Corrupted Image:** Write a script that verifies whether the image in the `train` directory are valid. If any corrupted images are found, the script must remove the image from the directory and print the message which image have been removed, if none found print ”No Corrupted Images Found.”

   - **Hint:**
     - Iterate through each class subdirectory and check if each image is valid.
     - Use the `Image.open()` function to attempt to load each image.
     - If the image is corrupted i.e. raises an `IOError` or `SyntaxError`, remove the image from the directory and print `f"Removed corrupted image: {image_path}"`.
     - Maintain a list of all corrupted image paths for reporting.
   - Expected Output: `No corrupted images found.`

Task 2: Loading and Preprocessing Image Data in keras:

In this section, we will load and preprocess image data from a directory using the

`image_dataset_from_directory` function in Keras.

This function is used to load images from a directory structure where subdirectories represent different classes. We will also apply basic preprocessing, including resizing and batching and normalization of images. The same function can be used to load train and validation data.

Sample Implementation.

```
train_ds = 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=123 )
```

Main Arguments:

Argument	Description
`train_dir`	Path to the directory containing the image data, where each subdirectory represents a class.
`labels`	How labels are assigned: 'inferred' means the subdirectory names will be used as labels.
`label_mode`	Specifies how labels are encoded: 'int' means labels are encoded as integers.
`image_size`	Target size of the images after resizing (e.g., (128, 128) for 128x128 images).
`interpolation`	Interpolation method used for resizing the images (e.g., 'nearest').
`batch_size`	Number of samples per batch during training or validation.
`shuffle`	Whether to shuffle the dataset. It is set to `True` for the training data, `False` for validation.
`validation_split`	Fraction of data to reserve for validation.
`subset`	Specifies whether to use the training subset or validation subset.
`seed`	A seed for reproducibility of the dataset split.

Table 4: Arguments for `image_dataset_from_directory` in Keras

Sample Implementation.

```
import tensorflow as tf
# Define image size and batch size
img_height = 128 # Example image height
img_width = 128 # Example image width
batch_size = 32
validation_split = 0.2 # 80% training, 20% validation
# Create a preprocessing layer for normalization
rescale = tf.keras.layers.Rescaling(1./255) # Normalize pixel values to [0, 1]
# Create training dataset with normalization
train_ds = 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=123
)
# Apply the normalization (Rescaling) to the dataset
train_ds = train_ds.map(lambda x, y: (rescale(x), y))
# Create validation dataset with normalization
val_ds = 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=123
)
# Apply the normalization (Rescaling) to the validation dataset
val_ds = val_ds.map(lambda x, y: (rescale(x), y))
```

The code defines a dataset for training and validation of an image classification model in keras. In above code:

- **Normalization (Rescaling):**

  ```
  rescale = tf.keras.layers.Rescaling(1./255) # Normalize pixel values to [0, 1]
  ```

- This layer normalizes the pixel values of images. The pixel values in images typically range from 0 to 255, so dividing by 255 scales the values to the range [0, 1]. This is a common preprocessing step for neural networks to improve training stability and model performance.

- **Image and Batch Size Parameters:**

  ```
  img_height = 128 # Reshapes to image height
  img_width = 128 # Reshapes image width
  batch_size = 32
  validation_split = 0.2 # 80% training, 20% validation
  ```

Task 3 - Implement a CNN with

Follow the following Structure and Hyper - parameters:

Convolutional Architecture:

- **Convolutional Layer 1:**
  - Filter Size (F): $(3, 3)$
  - Number of Filters (k): 32
  - Padding (P): same
  - Stride (s): 1
- **Activation Layer:** ReLU activation
- **Pooling Layer 1:** Max pooling
  - Filter Size (F): $(2, 2)$
  - Stride (s): 2
- **Convolutional Layer 2:**
  - Filter Size (F): $(3, 3)$
  - Number of Filters (k): 32
  - Padding (P): same
  - Stride (s): 1
- **Activation Layer:** ReLU activation
- **Pooling Layer 2:** Max pooling
  - Filter Size (F): $(2, 2)$
  - Stride (s): 2

Fully Connected Network Architecture:

- **Flatten Layer:** Flatten the input coming from the convolutional layers
- **Input Layer:**
- **Hidden Layer - 2:**
  - Number of Neurons: 64
  - Number of Neurons: 128
- **Output Layer:**
  - Number of Neurons: num_classes (number of output classes)

Task 4: Compile the Model

Model Compilation

- Choose an appropriate optimizer (e.g., Adam), loss function (e.g., sparse categorical crossentropy), and evaluation metric (e.g., accuracy).

Task 4: Train the Model

Model Training

- Use the model.fit() function to train the model. Set the batch size to 16 and the number of epochs to 250.
- Use val_ds for validation.
- Use callbacks such as `ModelCheckpoint` and `EarlyStopping` for saving the best model and avoiding overfitting.

Task 5: Evaluate the Model

Model Evaluation

- After training, evaluate the model using `model.evaluate()` on the test set to check the test accuracy and loss.

Task 6: Save and Load the Model

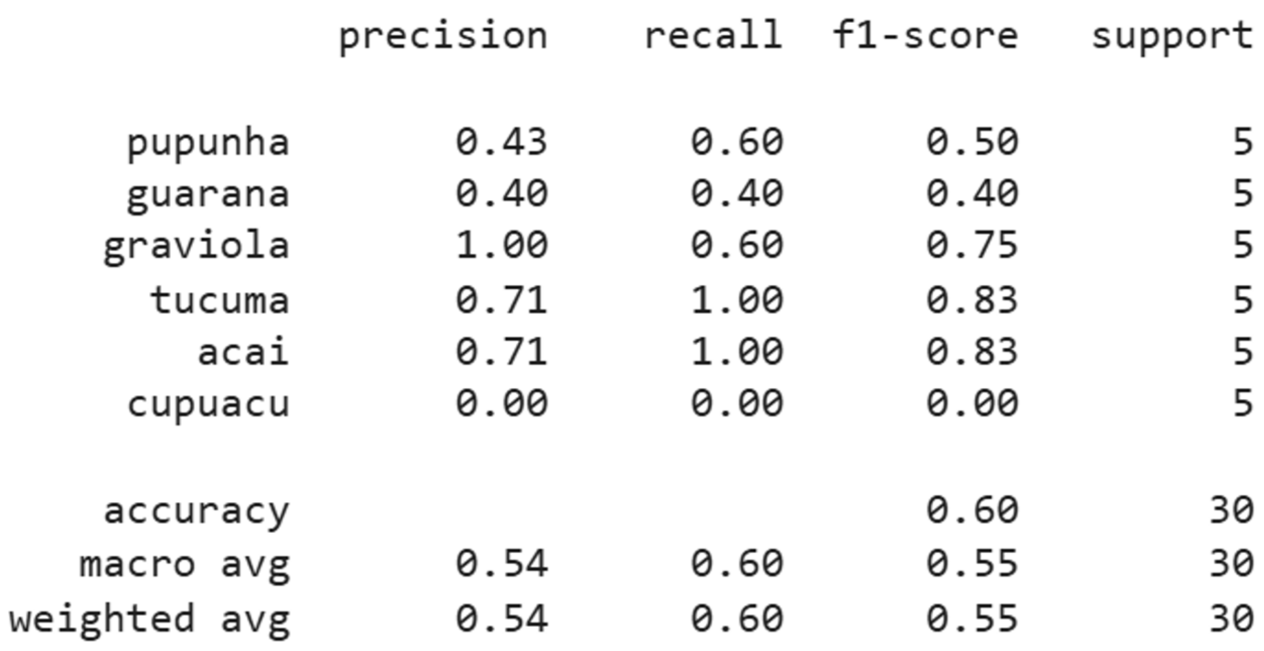
Model Saving and Loading

- Save the trained model to an `.h5` file using `model.save()`.
- Load the saved model and re-evaluate its performance on the test set.

Task 7: Predictions and Classification Report

Making Predictions

- Use `model.predict()` to make predictions on test images.
- Convert the model's predicted probabilities to digit labels using `np.argmax()`.
- Also use `from sklearn.metrics import classification_report` to report the Classification Report of your Model Performance.



```
              precision    recall  f1-score   support

     pupunha       0.43      0.60      0.50         5
     guarana       0.40      0.40      0.40         5
    graviola       1.00      0.60      0.75         5
      tucuma       0.71      1.00      0.83         5
        acai       0.71      1.00      0.83         5
     cupuacu       0.00      0.00      0.00         5

    accuracy                           0.60        30
   macro avg       0.54      0.60      0.55        30
weighted avg       0.54      0.60      0.55        30
```

Figure 7: A sample Classification Report.

Expected Deliverables

- **Code Implementation:** Complete code for building, training, evaluating, saving, and loading the model.
- **Visualization:** Graphs showing the training and validation loss and accuracy.
- **Classification Report:** Display the final Classification Report on test data.
- **Saved Model:** Submit the saved `.h5` model file.

——————— Good Luck. ———————-