6CS012 - Artificial Intelligence and Machine Learning. Practical Aspects of Training CNN for Image Classification Task.

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

March 30, 2025

Worksheet - 6.

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.

In this demonstration, we will implement Image Classification from scratch, without using pre-trained weights or pre-built keras application models. We will work with the `FruitsInAnazon` dataset and explore key deep learning techniques such as data augmentation, batch normalization, and dropout to enhance model performance. This example will provide a step -by- step workflow for training a custom neural network from the ground up.

2.1 Data Understanding and Visualizations.

In this section, we will perform following things:

- **Verify Dataset:** Before using images for training, we must load them from their respective folders. Each subdirectory in our dataset represents a class.

  1. Read the Dataset Directory.
  2. Extract the list of Class names (sub-folder names).

Sample Code for verifying Directory Structure .

```
import os
# Define dataset path
train_dir = "path/train dir"
# Get class names (subdirectories)
class_names = sorted(os.listdir(train_dir))
if not class_names:
    print("No class directories found in the train folder!")
else:
print(f"Found {len(class_names)} classes: {class_names}")
```

Expected Output:

```
Found 6 classes: ['acai', 'cupuacu', 'graviola', 'guarana', 'pupunha', 'tucuma']
```

- **Check for Corrupted Images:** When working with real world images, occurrences of corrupted images are common cause of error. Thu we must detect and remove them:
  - Open each image and verify if it's corrupted.
  - Store corrupted image paths for review and delete the image from the directory.

Sample Code for Checking Corrupted Image .

```
from PIL import Image, UnidentifiedImageError
corrupted_images = [] # List to store corrupted images path
# Loop through each class folder and check for corrupted images
for class_name in class_names:
    class_path = os.path.join(train_dir, class_name)
    if os.path.isdir(class_path): # Ensure it's a valid directory
        images = os.listdir(class_path)
        for img_name in images:
            img_path = os.path.join(class_path, img_name)
            try:
```

```
                with Image.open(img_path) as img:
                    img.verify() # Verify image integrity
            except (IOError, UnidentifiedImageError):
                corrupted_images.append(img_path)
# Print results
if corrupted_images:
    print("\nCorrupted Images Found:")
    for img in corrupted_images:
        print(img)
else:
    print("\nNo corrupted images found.")
```

Expected Output:

```
No corrupted images found.
```

- **Count class Balance:** A balanced dataset ensures that no class dominates the model. If some classes have significantly more images, the model may become biased.
  - Count the number of valid images in each class.
  - Print and evaluate the class distribution.

Sample Code for Checking Class Imbalance.

```
# Dictionary to store class counts
class_counts = {}
for class_name in class_names:
    class_path = os.path.join(train_dir, class_name)
    if os.path.isdir(class_path):
        images = [img for img in os.listdir(class_path) if img.lower().endswith(('.png', '.jpg'
            , '.jpeg'))]
        class_counts[class_name] = len(images) # Count images in each class
# Print Class Balance
print("\nClass Distribution:")
print("=" * 45)
print(f"{'Class Name':<25}{'Valid Image Count':>15}")
print("=" * 45)
for class_name, count in class_counts.items():
    print(f"{class_name:<25}{count:>15}")
print("=" * 45)
```

Figure 1: Expected Output

```
Class Distribution:
=============================================
Class Name                Valid Image Count
=============================================
acai                                     15
cupuacu                                  15
graviola                                 15
guarana                                  15
pupunha                                  15
tucuma                                   15
=============================================
```

- **Select Random Images for Visualization:** Visualizing images help verify the dataset is loaded correctly and if the images are representative of their classes.
  - Select one random image from each class.
  - Store the selected images for visualization.
  - Display randomly selected images in a grid format.

Sample Code for Random Image Selection.

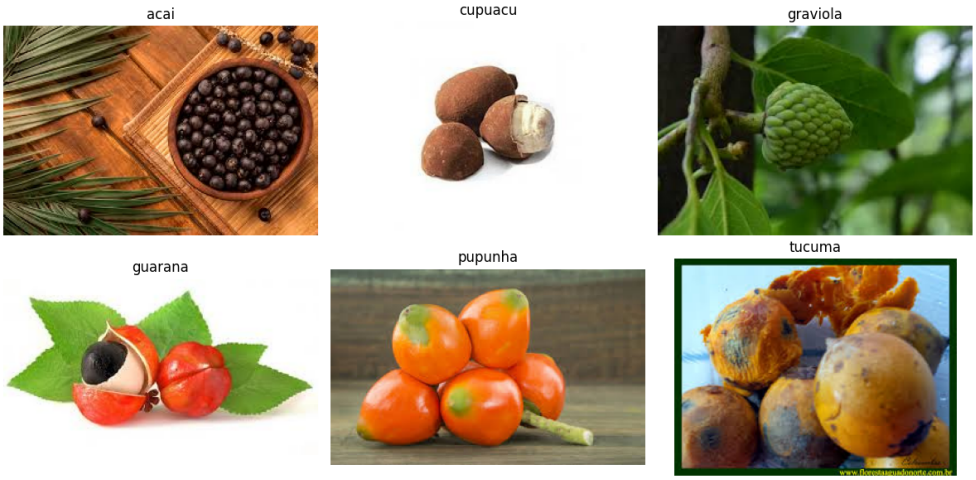
```
import random
selected_images = [] # Store image paths
selected_labels = [] # Store corresponding class names
for class_name in class_names:
    class_path = os.path.join(train_dir, class_name)
    if os.path.isdir(class_path):
        images = [img for img in os.listdir(class_path) if img.lower().endswith(('.png', '.jpg'
            , '.jpeg'))]
        if images: # Ensure the class folder is not empty
            selected_img = os.path.join(class_path, random.choice(images))
            selected_images.append(selected_img)
            selected_labels.append(class_name)
```

Sample Code for Random Image Plot.

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
# Determine grid size
num_classes = len(selected_images)
cols = (num_classes + 1) // 2 # Determine columns
rows = 2 # Fixed rows for layout
fig, axes = plt.subplots(rows, cols, figsize=(12, 6))
for i, ax in enumerate(axes.flat):
    if i < num_classes:
        img = mpimg.imread(selected_images[i])
        ax.imshow(img)
```

```
        ax.set_title(selected_labels[i])
        ax.axis("off")
    else:
        ax.axis("off") # Hide empty subplots
plt.tight_layout()
plt.show()
```

Figure 2: Expected Output



These are the preliminary task we do before starting any image classification project and are of great importance. Proper dataset preparation ensures that:

- **Data Integrity:** Detecting and removing corrupted images prevents unexpected errors training.
- **Balanced Dataset:** Checking class distribution helps avoid bais toward dominant classes.
- **Correct Labeling:** Visualizing random images ensures that the dataset is structured correctly.

These steps form the foundation of a good image classification model. Without them, the model might learn from incomplete, imbalanced, or corrupted data, leading to poor generalization.

2.2 Data Generation and Pre - processing:

We use the `image_dataset_from_directory` {We discussed this last week with detailed description about its various parameters} utility to generate the datasets, and we use Keras image preprocessing layers for image standardization and data augmentation. As dataset do not have seperate validation data, will be using 20% data from train dataset as validation data. For the demonstration further down I will be using `Devnagari Handwritten Digits` dataset from week 4.

Generating the Train and Val Dataset.

```
image_size = (28, 28)
batch_size = 32
train_ds, val_ds = keras.utils.image_dataset_from_directory(
    train_dir,
    validation_split=0.2,
    subset="both",
    seed=1337,
    image_size=image_size,
    batch_size=batch_size,
)
```

Understanding train_ds and val_ds:

- `train_ds or val_ds` are `tf.data.Dataset` objects.
- They do not have a fixed shape immediately, but yield batches of images and labels when iterated.
- To check the shape of a batch:

```
for images, labels in train_ds.take(1): # Take one batch
print("Images shape:", images.shape)
print("Labels shape:", labels.shape)
```

- Expected Output (Assuming `batch_size=32` and `image_size=28, 28`

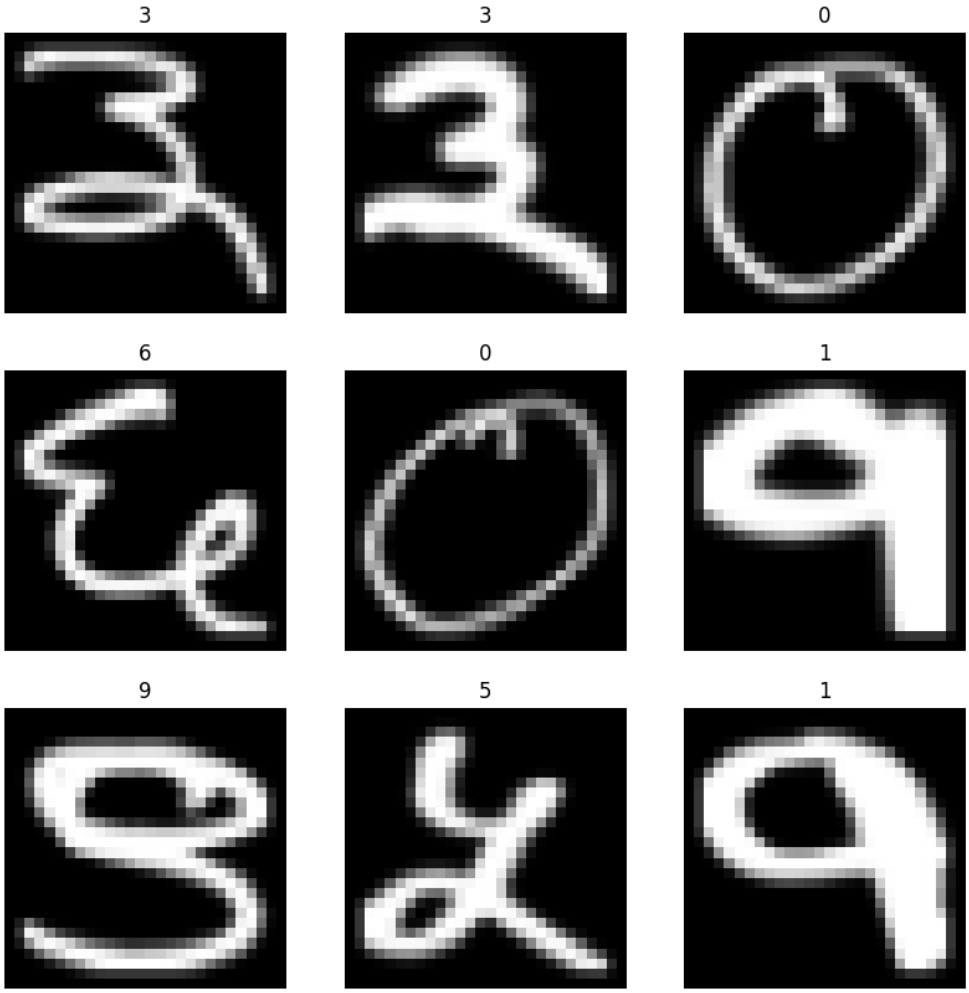
```
Images shape: (32, 28, 28, 1) # 32 images, each:height = 28, width = 28 and 1 color channels (
    RGB)
Labels shape: (32,) # 32 labels (one per image)
```

The above training dataset {`train_ds`} can be visualized using:

Visualizing training dataset

```
plt.figure(figsize=(10, 10))
for images, labels in train_ds.take(1):Takes one batch of images from the dataset (train_ds).
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(np.array(images[i]).astype("uint8"))
        plt.title(int(labels[i]))
        plt.axis("off")
```

Figure 3: Sample Output



1. Idea behind Data Augmentation:

Data Augmentation is a technique used in deep learning, especially in computer vision, to artificially expand a dataset by creating modified versions of images.

- Why use Data Augmentation?
  - Prevents Overfitting: Increases dataset diversity without collecting new data.

  - Improves Generalization: Helps the model perform better on unseen data.
  - Reduces Data Dependency: Useful when collecting large datasets is difficult.
  - Makes model more Robust: Introduces real – world variations like lighting changes, rotations, and distortions.

- Common techniques for Data Augmentations:

  1. Geometric Transformations: Some Examples are:
     - Flipping: Horizontal or vertical reflection.
     - Rotation: Rotating images by small degree.
     - Cropping: Randomly selecting a part of the image.
     - Scaling: Resizing images while maintaining proportions.
     - Translation: Shifting images left, right, up or down.
  2. Color Transformations: Some Examples are:
     - Brightness Adjustment: Making the image darker or brighter.
     - Contrast Change: Increasing or decreasing contrast.
     - Noise Addition: Some Examples are:
     - Adding random variations in pixel intensity.
  3. Occlusion and Information Loss: Some Examples are:
     - Cutout: Hiding part of the image to force the model to focus on other details.

Figure 4: Data Augmentation Visualizations

2. Data Augmentation with Keras:

Data augmentation in Keras is a technique used to artificially expand the training dataset by applying random transformations to the existing images. This helps improve model generalization and prevents overfitting.There are two methods on performing Data Augmentation in Keras:

Using Keras `ImageDataGenerator` (Old API):

Keras previously used the `ImageDataGenerator` class to perform real-time augmentation while training. We can still use this:

ImageDataGenerator

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rotation_range=30, # Rotate images by up to 30 degrees
    width_shift_range=0.2, # Shift width by 20%
    height_shift_range=0.2, # Shift height by 20%
    shear_range=0.2, # Shear transformation
    zoom_range=0.2, # Zoom in/out by 20%
    horizontal_flip=True, # Flip images horizontally
    fill_mode='nearest' # Fill in missing pixels
)

# Load an example image
from tensorflow.keras.preprocessing import image
import matplotlib.pyplot as plt
import numpy as np
img = image.load_img('sample.jpg', target_size=(128, 128))
x = image.img_to_array(img) # Convert to NumPy array
x = np.expand_dims(x, axis=0)
# Generate augmented images
aug_iter = datagen.flow(x, batch_size=1)
# Visualize 7 augmented images
fig, ax = plt.subplots(1, 7, figsize=(15, 5))
for i in range(7):
    batch = next(aug_iter)
    ax[i].imshow(batch[0].astype('uint8'))
    ax[i].axis('off')
plt.show()
```

The above code is expected to do following:

- Loads an Sample Image. ✓
- It applies data augmentation:
  - rotation ✓
  - shifts ✓
  - shear ✓
  - zoom ✓
  - flips ✓

- It visualizes 5 different augmented versions of the image. ✓

The code is expected to display following output:

Figure 5: Expected Output

Using Keras `tf.keras.layers.Random*` (New API):

The newer `tf.keras.layers` API provides augmentation layers that can be directly added to a model. Thus we expect you to use this for this workshop and your portfolio project.

Sample Implementation of tf.keras.layers.Random

```
# This is a sample implementation, Thus I only use two augmentation, you can use more than two
    augmentation type for your project. Please check keras documentations for other available
    augmentation techniques.
data_augmentation_layers = [
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(0.1),
]
def data_augmentation(images):
    for layer in data_augmentation_layers:
        images = layer(images)
    return images
```

Key difference from `ImageDataGenerator`:

- Unlike ImageDataGenerator, which operates offline (creating augmented images before training), these layers are part of the computational graph and applied during training i.e. Can be used inside a model or independently before training.
- Can be used as part of a TensorFlow model for real-time augmentation on the GPU.

DataAugemntation Layer directly in Model.

```
model = keras.Sequential([
    layers.Input(shape=(128, 128, 3)),
    *data_augmentation_layers, # Include augmentation before the model layers
    layers.Conv2D(32, (3,3), activation="relu"),
    layers.MaxPooling2D(),
    layers.Flatten(),
    layers.Dense(10, activation="softmax")
])
```

Visualizing augemented Image.

```
plt.figure(figsize=(10, 10))
for images, _ in train_ds.take(1):
    for i in range(9):
        augmented_images = data_augmentation(images)
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(np.array(augmented_images[0]).astype("uint8"))
        plt.axis("off")
```

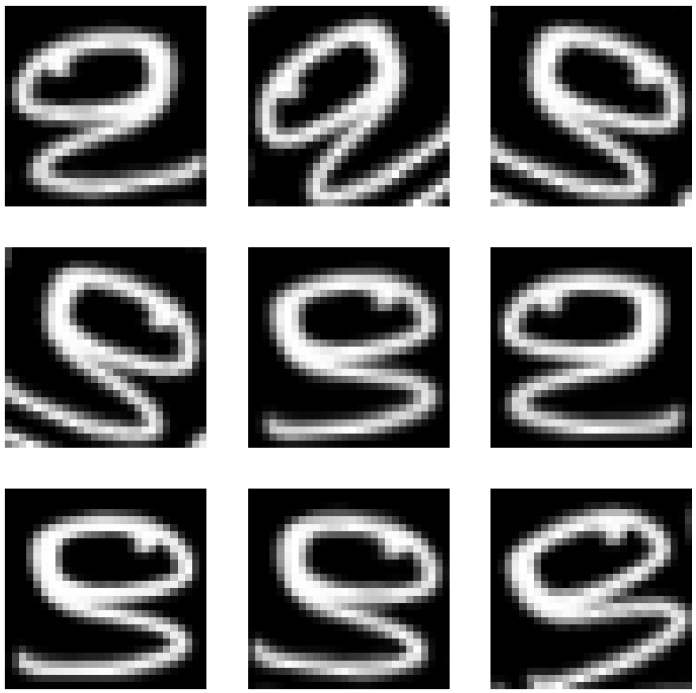


Figure 6: Visualizing Different Augmentations.

3. Data Pre - processing: Scaling the data with Keras:

The RGB channel values are in the [0, 255] range. This is not ideal for a neural network; in general you should seek to make your input values small. Here, we will standardize values to be in the [0, 1] by using a Rescaling layer at the start of our model. Keras provides two alternated to do this:

1. **Applying it to the dataset:** Last week, we followed a similar approach. The key change now is that we will apply data augmentation first, followed by rescaling, ensuring that the augmented data is also properly rescaled.

   ```
   augmented_train_ds = train_ds.map( lambda x, y: (data_augmentation(x, training=True), y))
   ```

2. **Make it part of the Model {Recommended Practice}:**

   Rescaling as the part of Model.

   ```
   inputs = keras.Input(shape=input_shape)
   x = data_augmentation(inputs)
   x = layers.Rescaling(1./255)(x)
   ... # Rest of the model
   ```

3. Why?

   - In the earlier version (last week style), data augmentation had to be applied before rescaling, which meant it ran on your device without GPU acceleration. However, by incorporating rescaling into the model, data augmentation can occur during model.fit(training), taking full advantage of GPU acceleration. **This approach is therefore recommended.**
   - Augmentation is inactive at test time, meaning it applies only during fit() method, not during evaluate(), and predict().

2.3 Model Building:

The sample model from last week:

End - to - End CNN Model.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
# 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])
```

This week we will add a BatchNormalization and Dropout layer to our Model.

Batch Normalization Layer

Batch Normalization (BN) is a technique used to improve the training of deep neural networks by reducing internal covariate shift. It normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. This helps in speeding up training and improving convergence.

1. BatchNormalization Class in Keras:

The BatchNormalization class in Keras normalizes the activations of the previous layer. It can be applied to both convolutional and dense layers.

Syntax of BatchNormalization Layer.

```
from tensorflow.keras.layers import BatchNormalization
layer = BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001, center=True, scale=True,
    beta_initializer="zeros", gamma_initializer="ones")
```

Main Arguments:

Argument	Description
`axis`	Axis along which to normalize (e.g., `-1` for channels last).
`momentum`	The momentum for the moving average of the mean and variance. Default: 0.99.
`epsilon`	A small constant to prevent division by zero. Default: 0.001.
`center`	Whether to add a learnable bias to the normalized output. Default: `True`.
`scale`	Whether to scale the normalized output by a learnable factor. Default: `True`.
`beta_initializer`	Initializer for the bias term. Default: `"zeros"`.
`gamma_initializer`	Initializer for the scaling factor. Default: `"ones"`.

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

2. How Does Batch Normalization Work?

Batch Normalization works by normalizing the input to each neuron in a given layer:

$$Z' = \frac{Z - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{1}$$

where:

- X = Input data (activations from the previous layer).
- μ_B = Mean of the batch.
- σ_B^2 = Variance of the batch.

- ϵ = Small constant added for numerical stability.
- $\hat{X}$ = Normalized output.

The normalized output is then scaled and shifted by learnable parameters, γ and β, respectively:

$$a' = \gamma Z' + \beta \tag{2}$$

Dropout Layer:

Dropout is a regularization technique used to prevent overfitting in neural networks by randomly setting a fraction of input units to zero during training. This helps in making the model more generalizable by reducing its reliance on specific neurons. During inference, all neurons are used, but their outputs are scaled by the dropout rate.

1. Dropout Class in Keras:

The Dropout class in Keras randomly sets a fraction of the input units to zero during training. It can be applied to any layer where overfitting might be a concern.

Syntax of Dropout Layer.

```
from tensorflow.keras.layers import Dropout
layer = Dropout(rate, noise_shape=None, seed=None)
```

Main Arguments:

Argument	Description
`rate`	Fraction of input units to drop. Should be a value between 0 and 1. E.g., 0.2 means dropping 20% of units.
`noise_shape`	1D tensor of the same length as the input shape, indicating which axes should be dropped together.
`seed`	Integer value used to seed the random number generator for reproducibility. Default: `None`.

Table 2: Arguments of the `Dropout` Layer in Keras

2. How Does Dropout Work?

Dropout works by randomly setting a fraction r of the input units to zero during training. This is done for each forward pass:

$$\hat{X} = \text{Dropout}(X, r) \tag{3}$$

where:

- X = Input data (activations from the previous layer).
- r = Dropout rate (fraction of neurons to drop, typically between 0 and 1).
- $\hat{X}$ = Modified input with some neurons randomly set to zero.

The dropout operation is performed only during training, and during inference, the output is scaled by $\frac{1}{1-r}$ to account for the fact that fewer neurons are active during training.

$$Y = \frac{\hat{X}}{1-r} \tag{4}$$

This scaling ensures that the expected value of the activations remains the same during training and testing.

2.3.1 Following is the Sample with the Model Implemented with BatchNormalization and Dropout Layer.

```
data_augmentation_layers = [
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(0.1),
]


def data_augmentation(images):
    for layer in data_augmentation_layers:
        images = layer(images)
    return images

# Define the model using the Sequential API with a list of layers
model = Sequential([
    # Data augmentation applied using Lambda layer
    layers.Lambda(data_augmentation, input_shape=(224, 224, 3)),

    # Rescaling layer to normalize pixel values
    layers.Rescaling(1./255),

    # First Convolutional Block
    Conv2D(32, (3, 3), padding='same', activation=None),
    BatchNormalization(),
    Activation('relu'),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    # Second Convolutional Block
    Conv2D(64, (3, 3), padding='same', activation=None),
    BatchNormalization(),
    Activation('relu'),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    # Third Convolutional Block
    Conv2D(128, (3, 3), padding='same', activation=None),
    BatchNormalization(),
    Activation('relu'),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    # Fourth Convolutional Block
    Conv2D(256, (3, 3), padding='same', activation=None),
    BatchNormalization(),
```

```
    Activation('relu'),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    # Flatten the output of the last Conv2D layer
    Flatten(),

    # First Fully Connected Layer
    Dense(512, activation=None),
    BatchNormalization(),
    Activation('relu'),
    Dropout(0.5),

    # Second Fully Connected Layer
    Dense(256, activation=None),
    BatchNormalization(),
    Activation('relu'),
    Dropout(0.5),

    # Third Fully Connected Layer
    Dense(128, activation=None),
    BatchNormalization(),
    Activation('relu'),
    Dropout(0.5),

    # Fourth Fully Connected Layer
    Dense(64, activation=None),
    BatchNormalization(),
    Activation('relu'),
    Dropout(0.5),

    # Output Layer with 10 neurons (for classification task with 10 classes)
    Dense(10, activation='softmax')
])

# Compile the model with Adam optimizer, sparse categorical crossentropy loss, and accuracy as the
    metric
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy', # Use this for integer labels
              metrics=['accuracy'])

# Summary of the model
model.summary()
```

3 Task - 1:

Repeat all the task from worksheet - 5 but, try to improve the model from last week with same dataset.

- Use Data Augmentation to increase the number of training image.
- Use deeper model with BN and DropOut layer as presented above.
- Understand the Model Summary and Training Behavior.

4 Image classification via fine-tuning with VGG16

Understanding Transfer Learning:

In above section we tried to implement Image classification model from scratch without using pre-trained weights, in this section we will try to use pre - trained weights to improve our model performance.

1. Load the Pre - trained Model:

In this step, you load a pre-trained model (e.g., VGG16) without the final classification layer. This allows you to use the learned features from a large dataset (e.g., ImageNet) and fine-tune the model for your own task.

```
from tensorflow.keras.applications import VGG16
# Load the VGG16 model pre-trained on ImageNet, without the top classification layer
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
```

In above code:

- `weights='imagenet'`: Specifies that the model is pre-trained on the ImageNet dataset.
- `include_top=False`: Excludes the final dense layers that are used for classification (this is where you will add your own layers).
- `input_shape=(28, 28, 1)`: Specifies the input shape of the images.

2. Freeze the Layers:

In this step, you freeze the layers of the pre-trained model, so their weights do not get updated during training. This helps in preserving the knowledge from the pre-training on ImageNet.

```
# Freeze the layers of the base model
for layer in base_model.layers:
    layer.trainable = False
```

This loop goes through each layer in the `base_model` and sets layer.trainable = False, which ensures that these layers will not be updated during backpropagation.

3. Add a Custom Layers:

Now, you add your custom layers to the model. This part consists of a global pooling layer (to reduce dimensionality), followed by dense layers to adjust the model for your 10-class classification problem.

```
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
# Add custom layers on top of the pre-trained model
x = base_model.output
x = GlobalAveragePooling2D()(x) # Reduces dimensions (height, width) to a single vector per image
x = Dense(1024, activation='relu')(x) # Fully connected layer with 1024 neurons
x = Dense(10, activation='softmax')(x) # Output layer for 10 classes (with softmax for multi-class
    classification)
```

The above code:

- `GlobalAveragePooling2D()`: This reduces the spatial dimensions (height, width) of the feature maps produced by the base model into a single vector for each image. This helps reduce the number of parameters and prevent overfitting.
- `Dense(1024, activation='relu')` A fully connected layer with 1024 neurons and the ReLU activation function.
- `Dense(10, activation='softmax')`: The final layer, with 10 neurons (one for each class in your dataset) and the softmax activation function to output class probabilities.

4. Create the Final Model:

This step combines the base model (with the custom layers added on top) to create the final model.

```
from tensorflow.keras.models import Model
# Create the final model
model = Model(inputs=base_model.input, outputs=x)
```

The above code:

- `inputs=base_model.input`: The input to the model is the input from the base model (the VGG16 model).
- `outputs=x`: The output of the model is the custom layers you added on top (the 10-class classification layer).

5. Compile and Fit the model.

```
from tensorflow.keras.optimizers import Adam
# Compile the model
model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
model.fit( #......)
```

5 Task - 2:

To - Do:

- Implement transfer learning using a pre-trained model trained on ImageNet weights, freeze the layers of the model, and fit it only on the output layer to classify the fruits dataset.
- Evaluate the model's performance and generate an inference output and classification report.
- Did the performance improved compared to training from scratch.

Expected Deliverables:

- Trained Model: A trained model with the frozen layers and fine-tuned output layer.
- Inference Output: Predicted class labels for the validation dataset.
- Classification Report: A detailed classification report showing the performance of the model for each class.

Good Luck.