DEEP LEARNING

MINI-PROJECT -2 TEAM

MEMBERS:

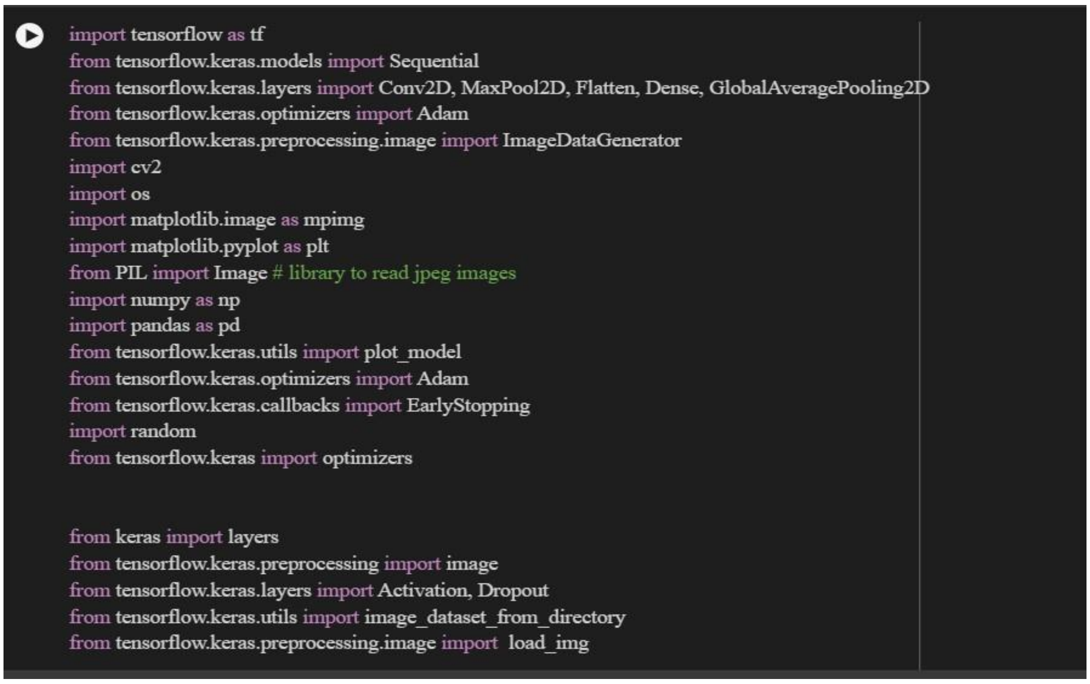
1. **Shafeena Farheen**

Question No:1 (10 marks)

Build a Convolution Neural Network to classify 6 classes of chess game images. Dataset Folder Name: Chess Conditions to consider:

- **Parameters should not cross 300000**
- **Should not use more than 4 layers (except input and output, including convolution and dense layers)**
- **Use Adam Optimizer Solution:**

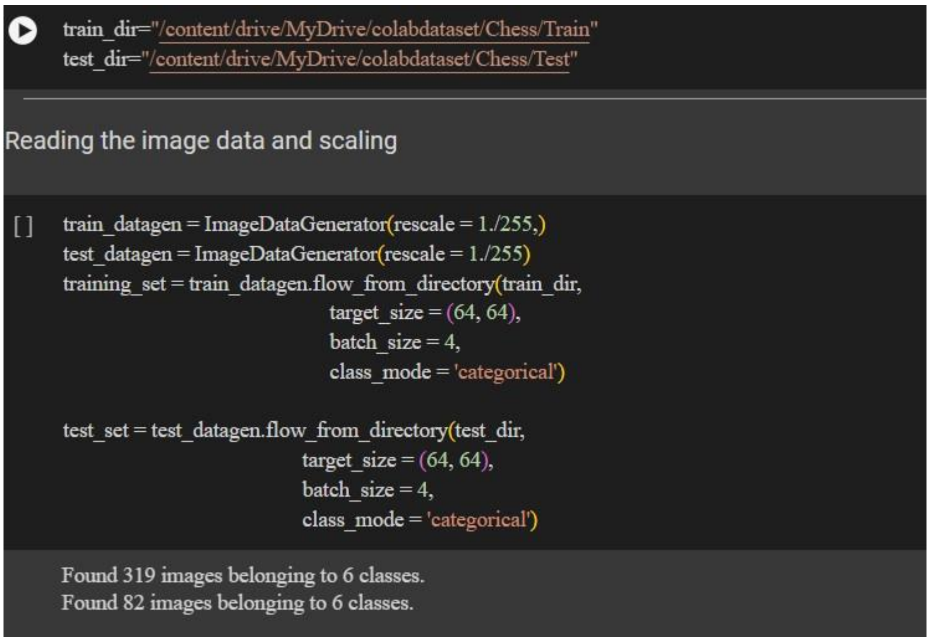
- **Import the required Libraries:**



```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPool2D, Flatten, Dense, GlobalAveragePooling2D
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import cv2
import os
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
from PIL import Image # library to read jpeg images
import numpy as np
import pandas as pd
from tensorflow.keras.utils import plot_model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
import random
from tensorflow.keras import optimizers


from keras import layers
from tensorflow.keras.preprocessing import image
from tensorflow.keras.layers import Activation, Dropout
from tensorflow.keras.utils import image_dataset_from_directory
from tensorflow.keras.preprocessing.image import  load_img
```

Inference:

Imported several libraries and modules related to TensorFlow, Keras, OpenCV, and other image processing libraries. This is commonly done for building and training deep learning models, especially for image classification tasks. **Reading the Chess Image data and Rescaling:**



```
train_dir="/content/drive/MyDrive/colabdataset/Chess/Train"
test_dir="/content/drive/MyDrive/colabdataset/Chess/Test"
```

Reading the image data and scaling

```
train_datagen = ImageDataGenerator(rescale = 1./255,)
test_datagen = ImageDataGenerator(rescale = 1./255)
training_set = train_datagen.flow_from_directory(train_dir,
                                                 target_size = (64, 64),
                                                 batch_size = 4,
                                                 class_mode = 'categorical')

test_set = test_datagen.flow_from_directory(test_dir,
                                            target_size = (64, 64),
                                            batch_size = 4,
                                            class_mode = 'categorical')
```

```
Found 319 images belonging to 6 classes.
Found 82 images belonging to 6 classes.
```

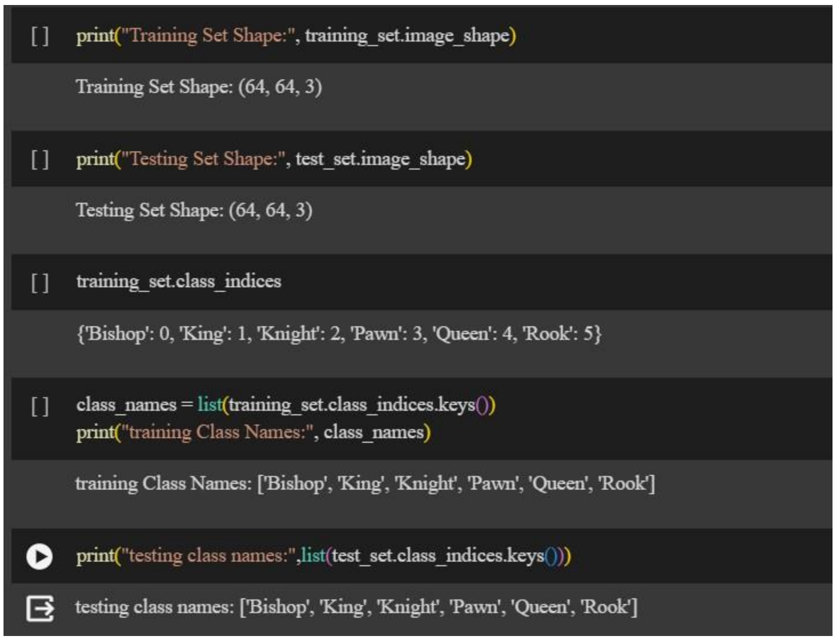
Inference:

- **ImageDataGenerator** from Keras to preprocess and augment your training and testing data. This is a common practice in deep learning to efficiently load and process large datasets for training neural networks.

Here's a breakdown of your code:

- **ImageDataGenerator Configuration:**
  - **rescale:** Rescaling factor applied to the pixel values. It's a common practice to scale pixel values to the range [0, 1].
- **flow_from_directory:** This method generates batches of augmented/normalized data from image files in a directory.
  - **train_dir and test_dir:** Paths to the directories containing the training and testing datasets, respectively.
  - **target_size:** Size to which all images will be resized during preprocessing.
  - **batch_size:** Number of images in each batch.
  - **class_mode:** Type of label arrays that are returned. In this case, 'categorical' indicates that the labels are one-hot encoded.
  - **training_set:** A generator for training data. It will yield batches of images and their corresponding labels during model training.
  - **test_set:** A generator for testing/validation data. It will yield batches of images and their corresponding labels for evaluating the model.

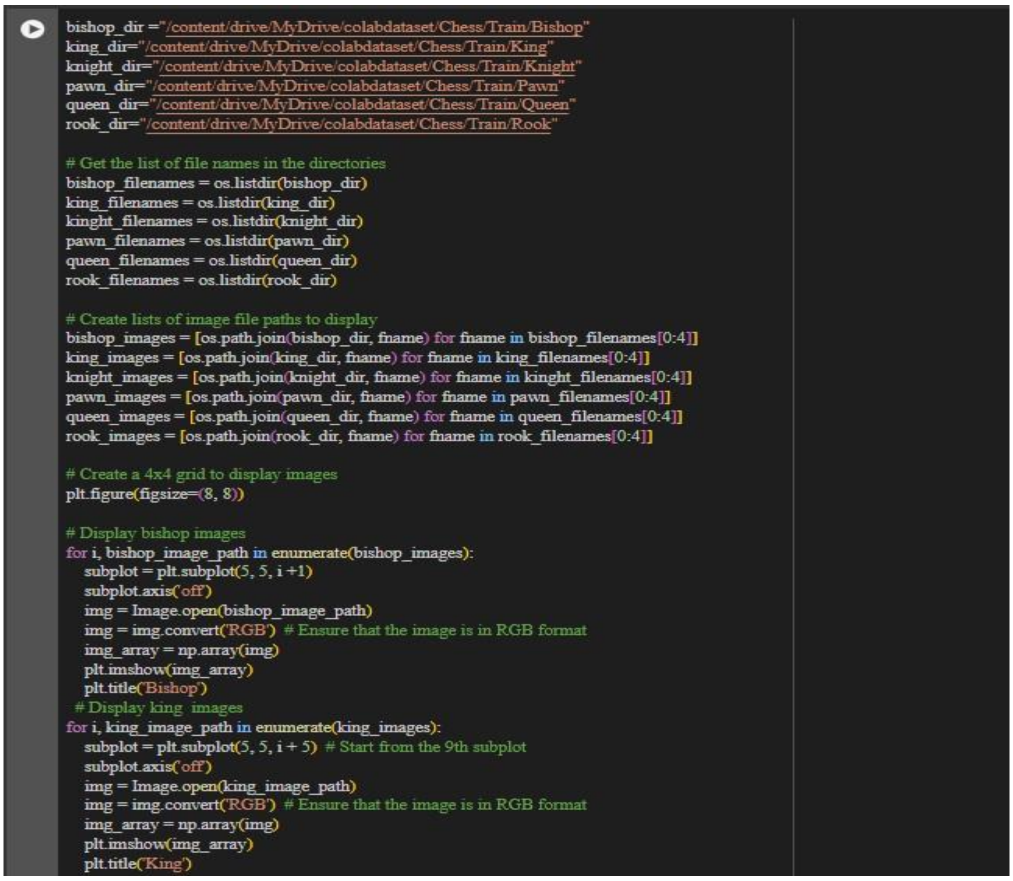
The chess image dataset has 6 class and 319 training samples and 89 testing samples.



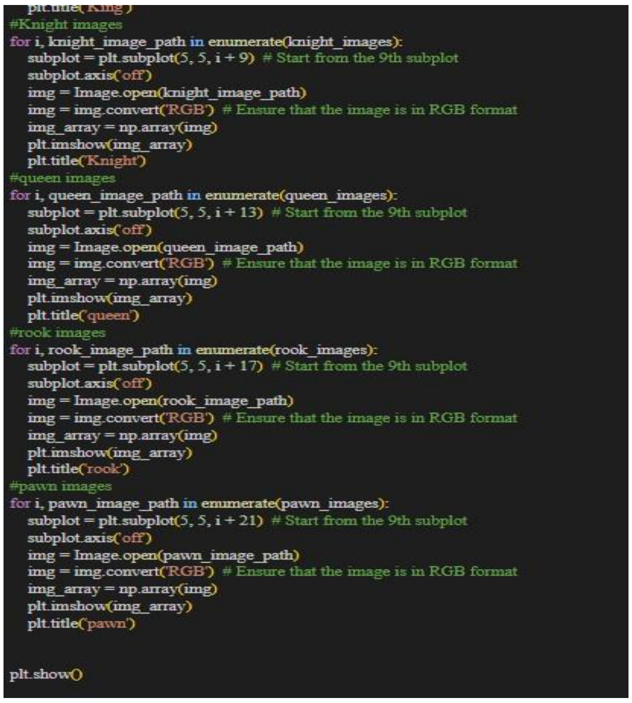
```
[ ] print("Training Set Shape:", training_set.image_shape)

    Training Set Shape: (64, 64, 3)

[ ] print("Testing Set Shape:", test_set.image_shape)

    Testing Set Shape: (64, 64, 3)

[ ] training_set.class_indices

    {'Bishop': 0, 'King': 1, 'Knight': 2, 'Pawn': 3, 'Queen': 4, 'Rook': 5}

[ ] class_names = list(training_set.class_indices.keys())
    print("training Class Names:", class_names)

    training Class Names: ['Bishop', 'King', 'Knight', 'Pawn', 'Queen', 'Rook']

    print("testing class names:",list(test_set.class_indices.keys()))

    testing class names: ['Bishop', 'King', 'Knight', 'Pawn', 'Queen', 'Rook']
```

The above code shows the shape of training and testing images and it also listed the class labels of chess dataset

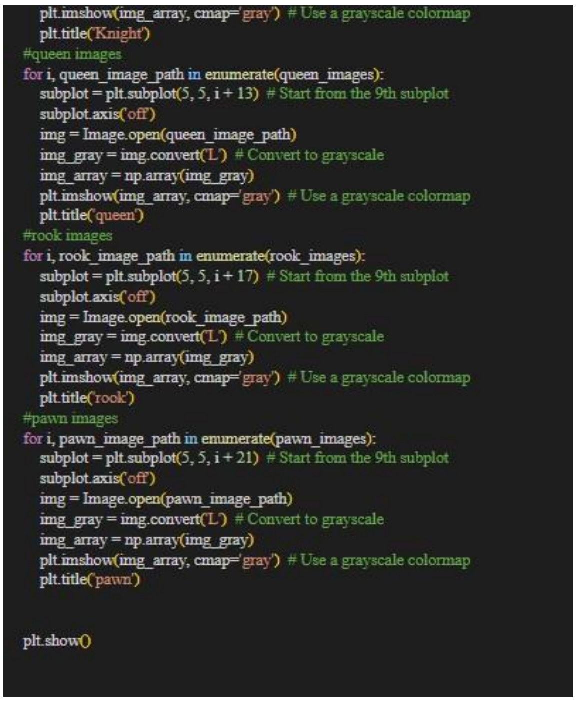
- **To display the images of different class in chess dataset**



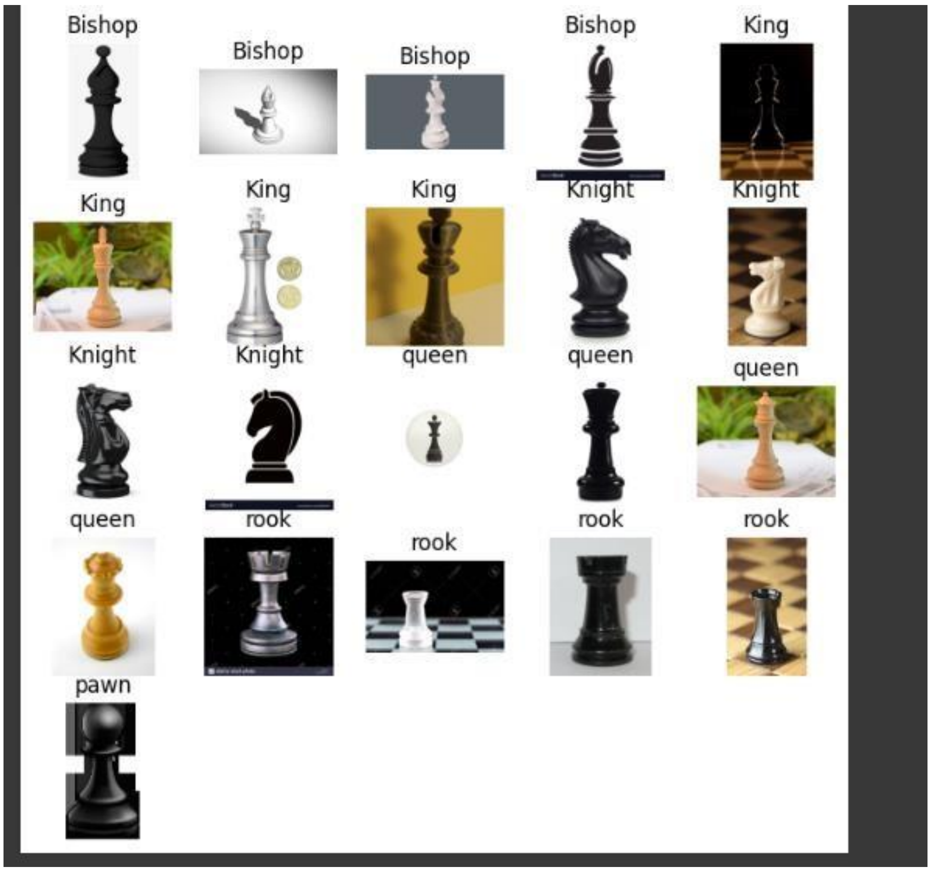
```python
bishop_dir ="/content/drive/MyDrive/colabdataset/Chess/Train/Bishop"
king_dir="/content/drive/MyDrive/colabdataset/Chess/Train/King"
knight_dir="/content/drive/MyDrive/colabdataset/Chess/Train/Knight"
pawn_dir="/content/drive/MyDrive/colabdataset/Chess/Train/Pawn"
queen_dir="/content/drive/MyDrive/colabdataset/Chess/Train/Queen"
rook_dir="/content/drive/MyDrive/colabdataset/Chess/Train/Rook"

# Get the list of file names in the directories
bishop_filenames = os.listdir(bishop_dir)
king_filenames = os.listdir(king_dir)
kinght_filenames = os.listdir(knight_dir)
pawn_filenames = os.listdir(pawn_dir)
queen_filenames = os.listdir(queen_dir)
rook_filenames = os.listdir(rook_dir)

# Create lists of image file paths to display
bishop_images = [os.path.join(bishop_dir, fname) for fname in bishop_filenames[0:4]]
king_images = [os.path.join(king_dir, fname) for fname in king_filenames[0:4]]
knight_images = [os.path.join(knight_dir, fname) for fname in kinght_filenames[0:4]]
pawn_images = [os.path.join(pawn_dir, fname) for fname in pawn_filenames[0:4]]
queen_images = [os.path.join(queen_dir, fname) for fname in queen_filenames[0:4]]
rook_images = [os.path.join(rook_dir, fname) for fname in rook_filenames[0:4]]

# Create a 4x4 grid to display images
plt.figure(figsize=(8, 8))

# Display bishop images
for i, bishop_image_path in enumerate(bishop_images):
    subplot = plt.subplot(5, 5, i +1)
    subplot.axis('off')
    img = Image.open(bishop_image_path)
    img = img.convert('RGB')  # Ensure that the image is in RGB format
    img_array = np.array(img)
    plt.imshow(img_array)
    plt.title('Bishop')
 # Display king  images
for i, king_image_path in enumerate(king_images):
    subplot = plt.subplot(5, 5, i + 5)  # Start from the 9th subplot
    subplot.axis('off')
    img = Image.open(king_image_path)
    img = img.convert('RGB')  # Ensure that the image is in RGB format
    img_array = np.array(img)
    plt.imshow(img_array)
    plt.title('King')
```

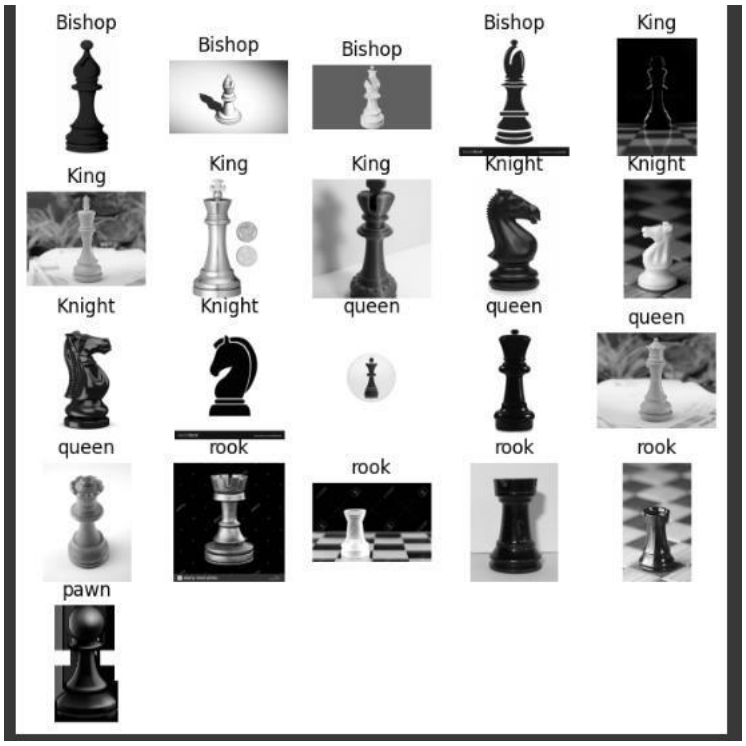


```python
#Knight images
for i, knight_image_path in enumerate(knight_images):
    subplot = plt.subplot(5, 5, i + 9)  # Start from the 9th subplot
    subplot.axis('off')
    img = Image.open(knight_image_path)
    img = img.convert('RGB')  # Ensure that the image is in RGB format
    img_array = np.array(img)
    plt.imshow(img_array)
    plt.title('Knight')
#queen images
for i, queen_image_path in enumerate(queen_images):
    subplot = plt.subplot(5, 5, i + 13)  # Start from the 9th subplot
    subplot.axis('off')
    img = Image.open(queen_image_path)
    img = img.convert('RGB')  # Ensure that the image is in RGB format
    img_array = np.array(img)
    plt.imshow(img_array)
    plt.title('queen')
#rook images
for i, rook_image_path in enumerate(rook_images):
    subplot = plt.subplot(5, 5, i + 17)  # Start from the 9th subplot
    subplot.axis('off')
    img = Image.open(rook_image_path)
    img = img.convert('RGB')  # Ensure that the image is in RGB format
    img_array = np.array(img)
    plt.imshow(img_array)
    plt.title('rook')
#pawn images
for i, pawn_image_path in enumerate(pawn_images):
    subplot = plt.subplot(5, 5, i + 21)  # Start from the 9th subplot
    subplot.axis('off')
    img = Image.open(pawn_image_path)
    img = img.convert('RGB')  # Ensure that the image is in RGB format
    img_array = np.array(img)
    plt.imshow(img_array)
    plt.title('pawn')


plt.show()
```



```python
    plt.imshow(img_array, cmap='gray')  # Use a grayscale colormap
    plt.title('Knight')
#queen images
for i, queen_image_path in enumerate(queen_images):
    subplot = plt.subplot(5, 5, i + 13)  # Start from the 9th subplot
    subplot.axis('off')
    img = Image.open(queen_image_path)
    img_gray = img.convert('L')  # Convert to grayscale
    img_array = np.array(img_gray)
    plt.imshow(img_array, cmap='gray')  # Use a grayscale colormap
    plt.title('queen')
#rook images
for i, rook_image_path in enumerate(rook_images):
    subplot = plt.subplot(5, 5, i + 17)  # Start from the 9th subplot
    subplot.axis('off')
    img = Image.open(rook_image_path)
    img_gray = img.convert('L')  # Convert to grayscale
    img_array = np.array(img_gray)
    plt.imshow(img_array, cmap='gray')  # Use a grayscale colormap
    plt.title('rook')
#pawn images
for i, pawn_image_path in enumerate(pawn_images):
    subplot = plt.subplot(5, 5, i + 21)  # Start from the 9th subplot
    subplot.axis('off')
    img = Image.open(pawn_image_path)
    img_gray = img.convert('L')  # Convert to grayscale
    img_array = np.array(img_gray)
    plt.imshow(img_array, cmap='gray')  # Use a grayscale colormap
    plt.title('pawn')


plt.show()
```



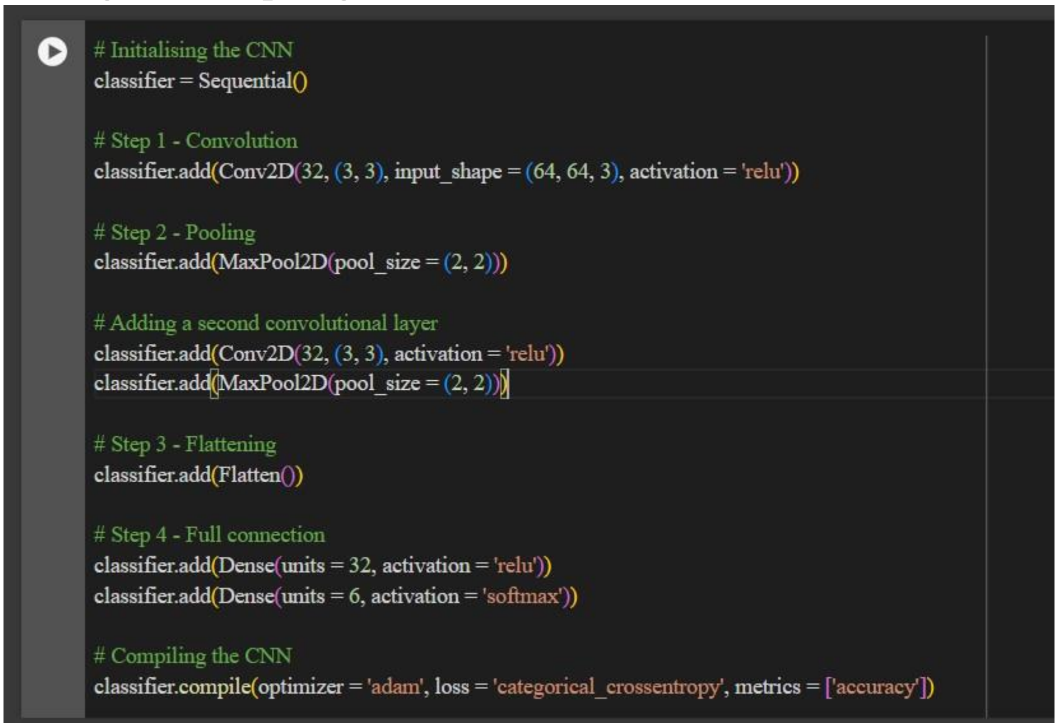


Inference:

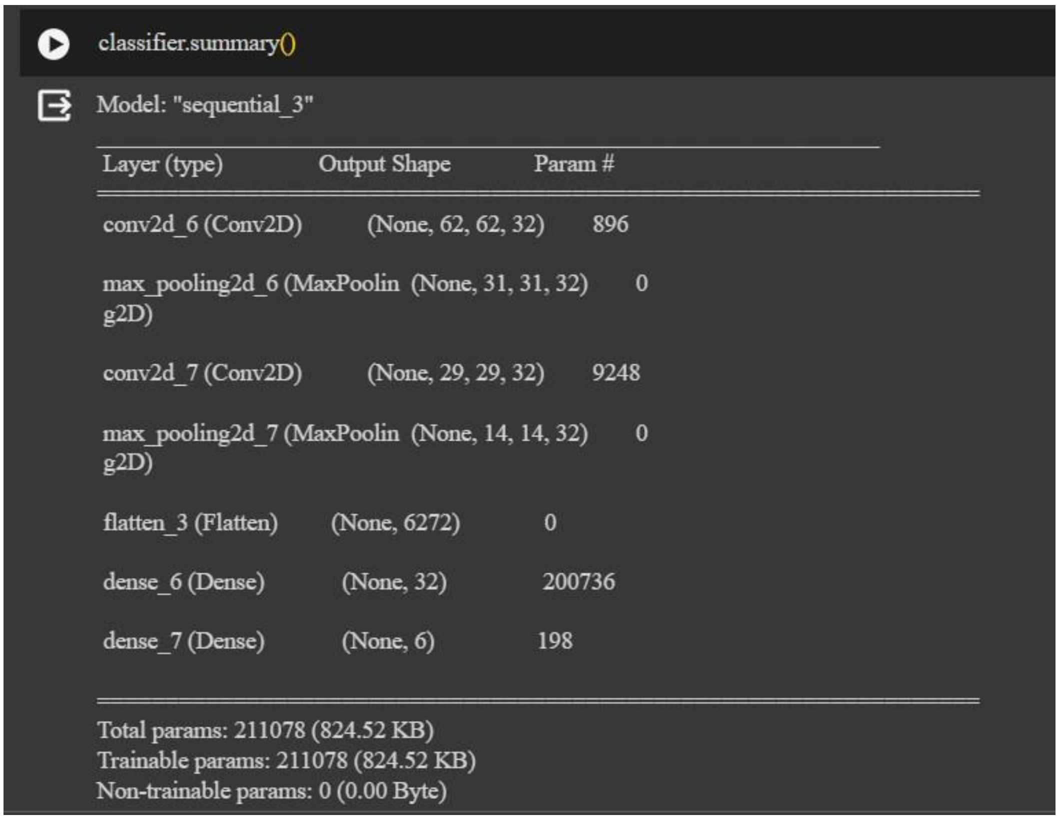
- we are loading and displaying sample images from different chess piece directories (Bishop, King, Knight, Pawn, Queen, Rook). we are using Matplotlib and subplots to create a grid and display the images.

- We displayed the chess images in both RGB and Gray scale image format.

- **Building and compiling the CNN model**



```
# Initialising the CNN
classifier = Sequential()

# Step 1 - Convolution
classifier.add(Conv2D(32, (3, 3), input_shape = (64, 64, 3), activation = 'relu'))

# Step 2 - Pooling
classifier.add(MaxPool2D(pool_size = (2, 2)))

# Adding a second convolutional layer
classifier.add(Conv2D(32, (3, 3), activation = 'relu'))
classifier.add(MaxPool2D(pool_size = (2, 2)))

# Step 3 - Flattening
classifier.add(Flatten())

# Step 4 - Full connection
classifier.add(Dense(units = 32, activation = 'relu'))
classifier.add(Dense(units = 6, activation = 'softmax'))

# Compiling the CNN
classifier.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])
```



```
classifier.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 62, 62, 32)        896

 max_pooling2d_6 (MaxPoolin  (None, 31, 31, 32)        0
 g2D)

 conv2d_7 (Conv2D)           (None, 29, 29, 32)        9248

 max_pooling2d_7 (MaxPoolin  (None, 14, 14, 32)        0
 g2D)

 flatten_3 (Flatten)         (None, 6272)              0

 dense_6 (Dense)             (None, 32)                200736

 dense_7 (Dense)             (None, 6)                 198

=================================================================
Total params: 211078 (824.52 KB)
Trainable params: 211078 (824.52 KB)
Non-trainable params: 0 (0.00 Byte)
```

Inference:

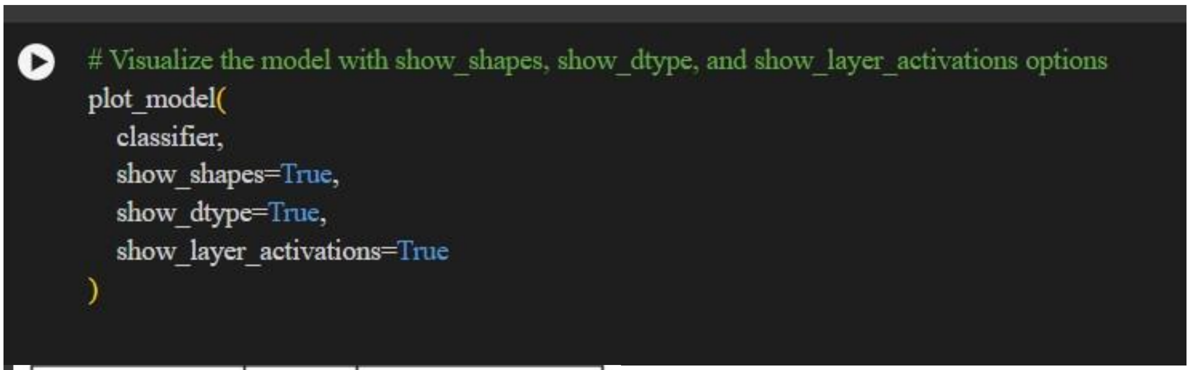
- We are building a Convolutional Neural Network (CNN) for a chess piece classification task.

Here's a breakdown of your code:

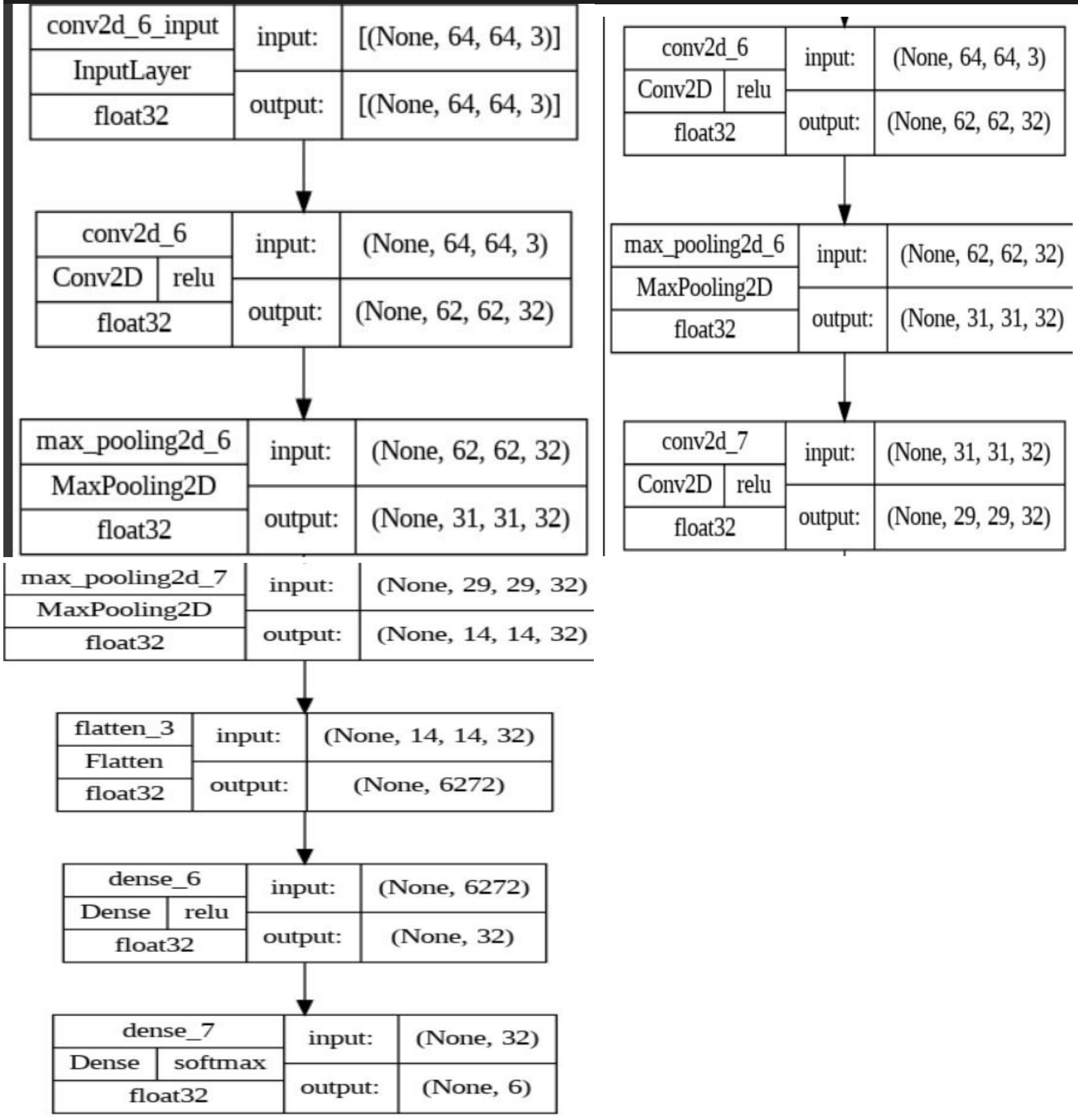
- **Sequential Model Initialization:**
  - classifier = Sequential(): Initializes a sequential model.
- **Convolutional Layers:**
  - **Conv2D(32, (3, 3), input_shape=(64, 64, 3), activation='relu'):** First convolutional layer with 32 filters, a kernel size of (3, 3), input shape (64, 64, 3) representing a 64x64 image with 3 color channels (RGB), and ReLU activation.
  - **MaxPool2D(pool_size=(2, 2)):** Max pooling layer with a pool size of (2, 2) to down-sample the spatial dimensions.
  - **Conv2D(32, (3, 3), activation='relu'):** Second convolutional layer with 32 filters and a kernel size of (3, 3).
  - **MaxPool2D(pool_size=(2, 2)):** Max pooling layer following the second convolutional layer.
- **Flattening:**
  - **Flatten():** Flattens the output from the previous layer into a 1D array.
- **Fully Connected (Dense) Layers:**
  - **Dense(units=32, activation='relu'**): Fully connected layer with 32 units and ReLU activation.
  - **Dense(units=6, activation='softmax'):** Output layer with 6 units (6 classes for chess pieces) and softmax activation for multi-class classification.
- **Compiling the Model:**
  - **compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'**]): Configures the model for training with the Adam optimizer, categorical crossentropy loss (suitable for multi-class classification), and accuracy as the evaluation metric.

- The total parameters represent the number of weights and biases in your model. The trainable parameters are those that will be updated during training, while non-trainable parameters are fixed.
- We used 4 layers (2 convolution layer,2 dense layer) and total parameters less than 300000 . Here the model learns 211078 learnable parameters.
- The Adam optimizer is an optimization algorithm commonly used in training deep neural networks. It combines ideas from two other popular optimizers: RMSprop and Momentum. The Adam optimizer adapts the learning rates of each parameter individually by considering both the first-order momentum and the second-order scaling of the gradients.

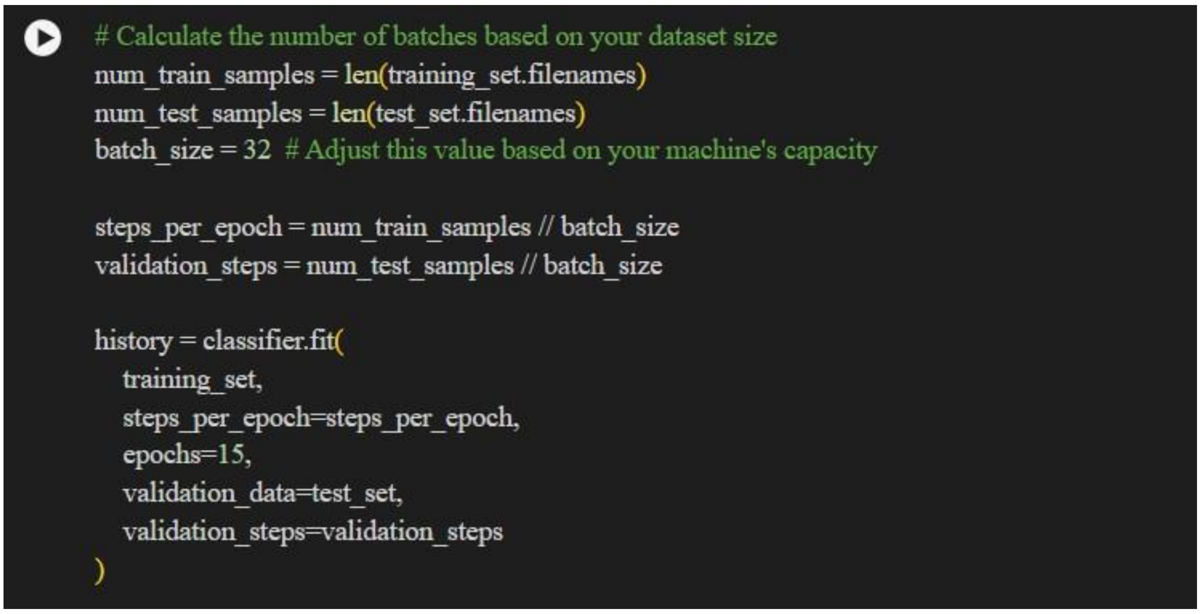
- **Visualize of the CNN model:**



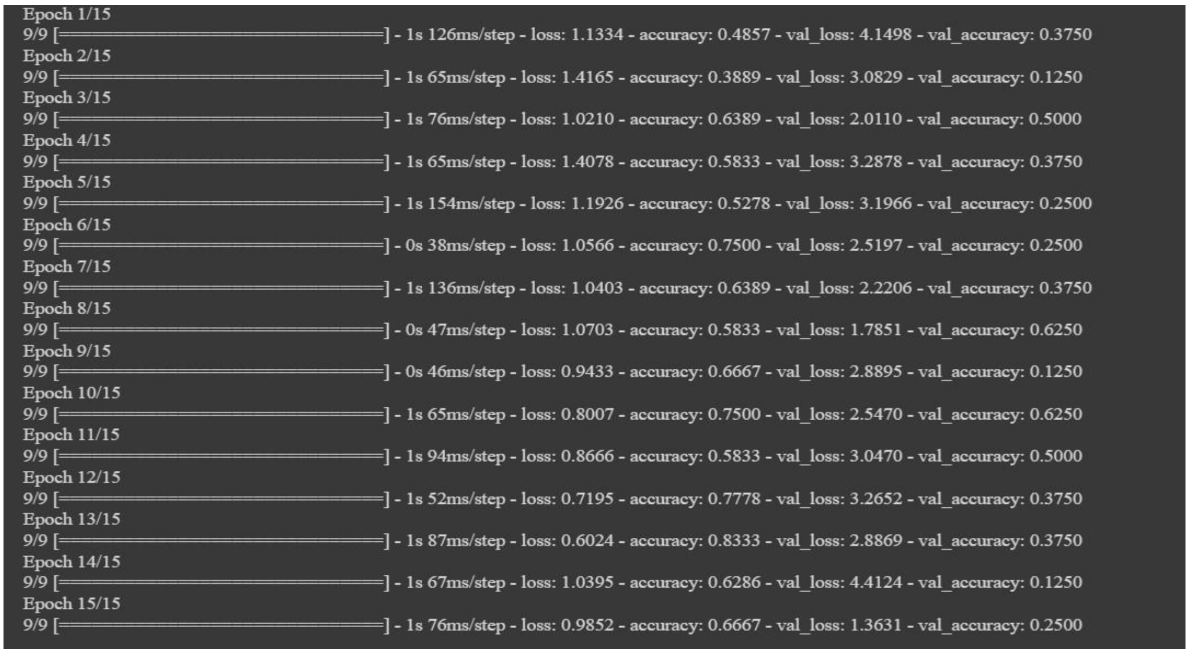
```python
# Visualize the model with show_shapes, show_dtype, and show_layer_activations options
plot_model(
    classifier,
    show_shapes=True,
    show_dtype=True,
    show_layer_activations=True
)
```


conv2d_6_input | InputLayer | float32 | input: [(None, 64, 64, 3)] | output: [(None, 64, 64, 3)]
conv2d_6 | Conv2D | relu | float32 | input: (None, 64, 64, 3) | output: (None, 62, 62, 32)
max_pooling2d_6 | MaxPooling2D | float32 | input: (None, 62, 62, 32) | output: (None, 31, 31, 32)
conv2d_6 | Conv2D | relu | float32 | input: (None, 64, 64, 3) | output: (None, 62, 62, 32)
max_pooling2d_6 | MaxPooling2D | float32 | input: (None, 62, 62, 32) | output: (None, 31, 31, 32)
conv2d_7 | Conv2D | relu | float32 | input: (None, 31, 31, 32) | output: (None, 29, 29, 32)
max_pooling2d_7 | MaxPooling2D | float32 | input: (None, 29, 29, 32) | output: (None, 14, 14, 32)
flatten_3 | Flatten | float32 | input: (None, 14, 14, 32) | output: (None, 6272)
dense_6 | Dense | relu | float32 | input: (None, 6272) | output: (None, 32)
dense_7 | Dense | softmax | float32 | input: (None, 32) | output: (None, 6)

- **Fitting the CNN Model with Chess dataset:**



```
# Calculate the number of batches based on your dataset size
num_train_samples = len(training_set.filenames)
num_test_samples = len(test_set.filenames)
batch_size = 32  # Adjust this value based on your machine's capacity

steps_per_epoch = num_train_samples // batch_size
validation_steps = num_test_samples // batch_size

history = classifier.fit(
    training_set,
    steps_per_epoch=steps_per_epoch,
    epochs=15,
    validation_data=test_set,
    validation_steps=validation_steps
)
```



```
Epoch 1/15
9/9 [==============================] - 1s 126ms/step - loss: 1.1334 - accuracy: 0.4857 - val_loss: 4.1498 - val_accuracy: 0.3750
Epoch 2/15
9/9 [==============================] - 1s 65ms/step - loss: 1.4165 - accuracy: 0.3889 - val_loss: 3.0829 - val_accuracy: 0.1250
Epoch 3/15
9/9 [==============================] - 1s 76ms/step - loss: 1.0210 - accuracy: 0.6389 - val_loss: 2.0110 - val_accuracy: 0.5000
Epoch 4/15
9/9 [==============================] - 1s 65ms/step - loss: 1.4078 - accuracy: 0.5833 - val_loss: 3.2878 - val_accuracy: 0.3750
Epoch 5/15
9/9 [==============================] - 1s 154ms/step - loss: 1.1926 - accuracy: 0.5278 - val_loss: 3.1966 - val_accuracy: 0.2500
Epoch 6/15
9/9 [==============================] - 0s 38ms/step - loss: 1.0566 - accuracy: 0.7500 - val_loss: 2.5197 - val_accuracy: 0.2500
Epoch 7/15
9/9 [==============================] - 1s 136ms/step - loss: 1.0403 - accuracy: 0.6389 - val_loss: 2.2206 - val_accuracy: 0.3750
Epoch 8/15
9/9 [==============================] - 0s 47ms/step - loss: 1.0703 - accuracy: 0.5833 - val_loss: 1.7851 - val_accuracy: 0.6250
Epoch 9/15
9/9 [==============================] - 0s 46ms/step - loss: 0.9433 - accuracy: 0.6667 - val_loss: 2.8895 - val_accuracy: 0.1250
Epoch 10/15
9/9 [==============================] - 1s 65ms/step - loss: 0.8007 - accuracy: 0.7500 - val_loss: 2.5470 - val_accuracy: 0.6250
Epoch 11/15
9/9 [==============================] - 1s 94ms/step - loss: 0.8666 - accuracy: 0.5833 - val_loss: 3.0470 - val_accuracy: 0.5000
Epoch 12/15
9/9 [==============================] - 1s 52ms/step - loss: 0.7195 - accuracy: 0.7778 - val_loss: 3.2652 - val_accuracy: 0.3750
Epoch 13/15
9/9 [==============================] - 1s 87ms/step - loss: 0.6024 - accuracy: 0.8333 - val_loss: 2.8869 - val_accuracy: 0.3750
Epoch 14/15
9/9 [==============================] - 1s 67ms/step - loss: 1.0395 - accuracy: 0.6286 - val_loss: 4.4124 - val_accuracy: 0.1250
Epoch 15/15
9/9 [==============================] - 1s 76ms/step - loss: 0.9852 - accuracy: 0.6667 - val_loss: 1.3631 - val_accuracy: 0.2500
```

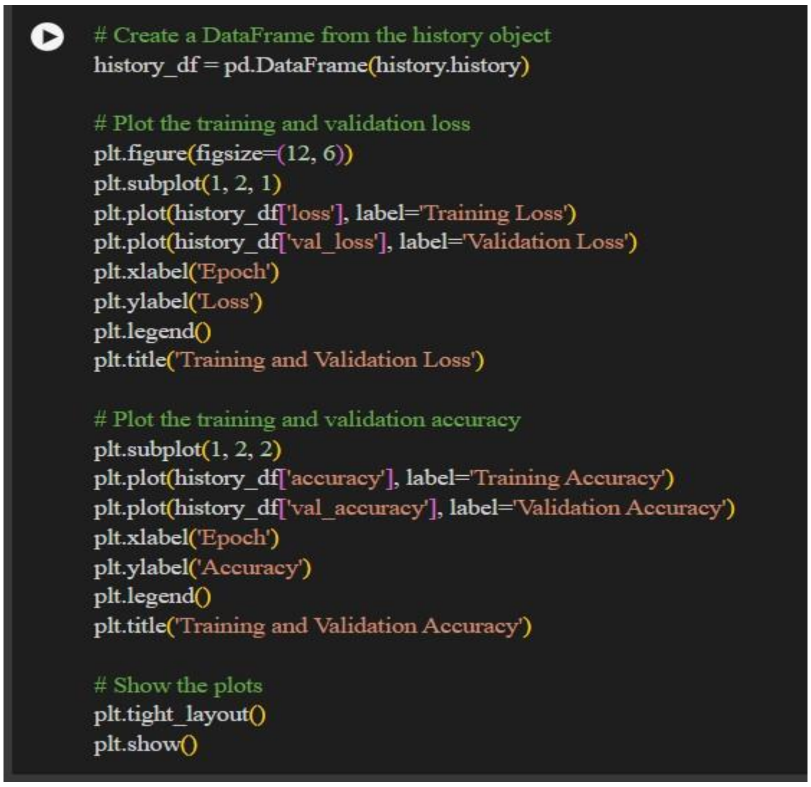
Inference:

- The fit method to train your model (classifier) on the training dataset (training_set) and validate it on the test dataset (test_set). The steps_per_epoch and validation_steps parameters are used to determine the number of batches to process in each epoch during training and validation, respectively. Here's a brief breakdown of your code:

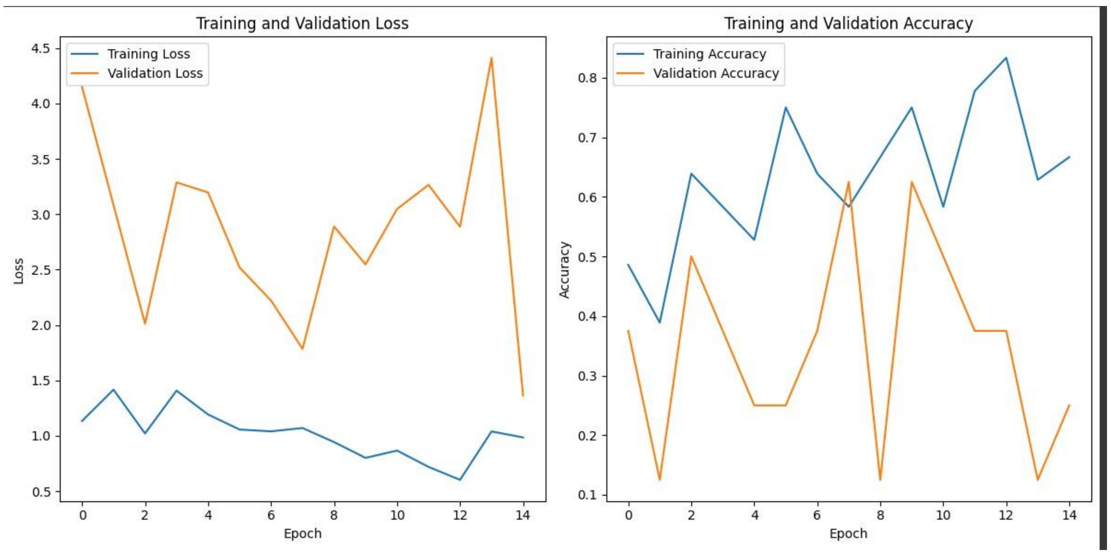
  - **num_train_samples and num_test_samples:** These variables store the number of samples in your training and test datasets, respectively.

- **batch_size:** This variable specifies the number of samples in each batch. Adjust this value based on your machine's capacity and memory constraints.
- **steps_per_epoch:** It determines the number of batches to process in each epoch during training. It is calculated as the total number of training samples divided by the batch size.
- **validation_steps:** Similar to steps_per_epoch, it determines the number of batches to process during validation.
- **history:** It stores the training and validation metrics over epochs.
- **fit:** This method trains the model for a fixed number of epochs (epochs). It uses the training dataset (training_set) for training and the validation dataset (test_set) for validation.

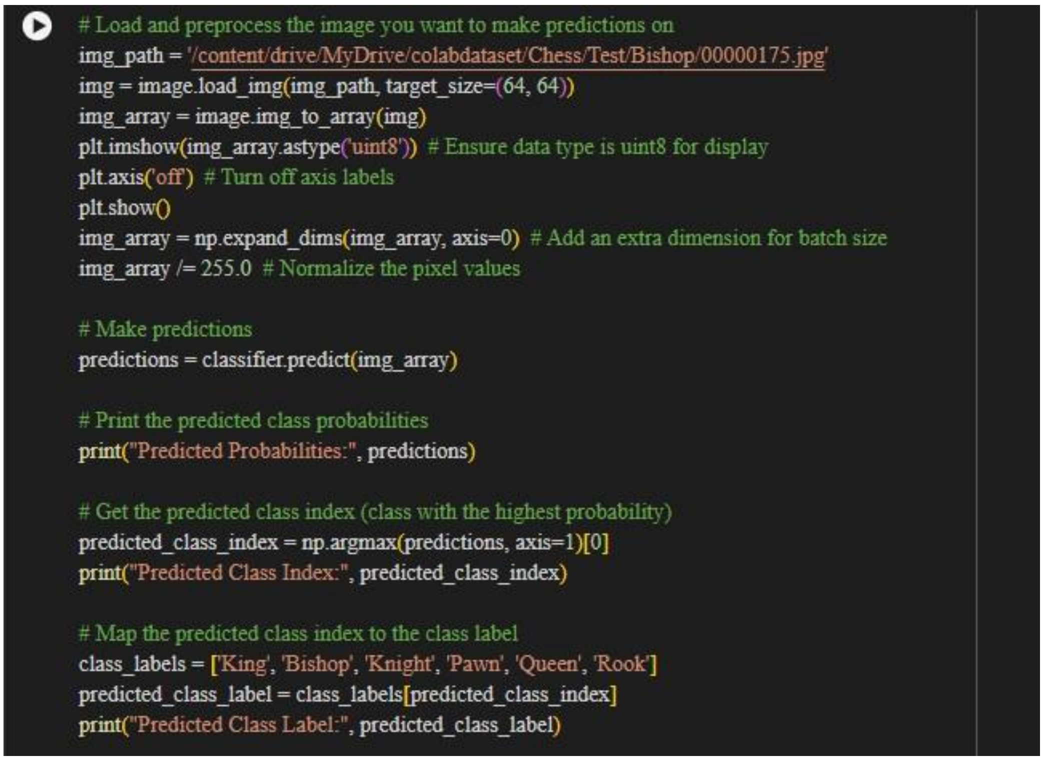
- **Evaluating the Model**



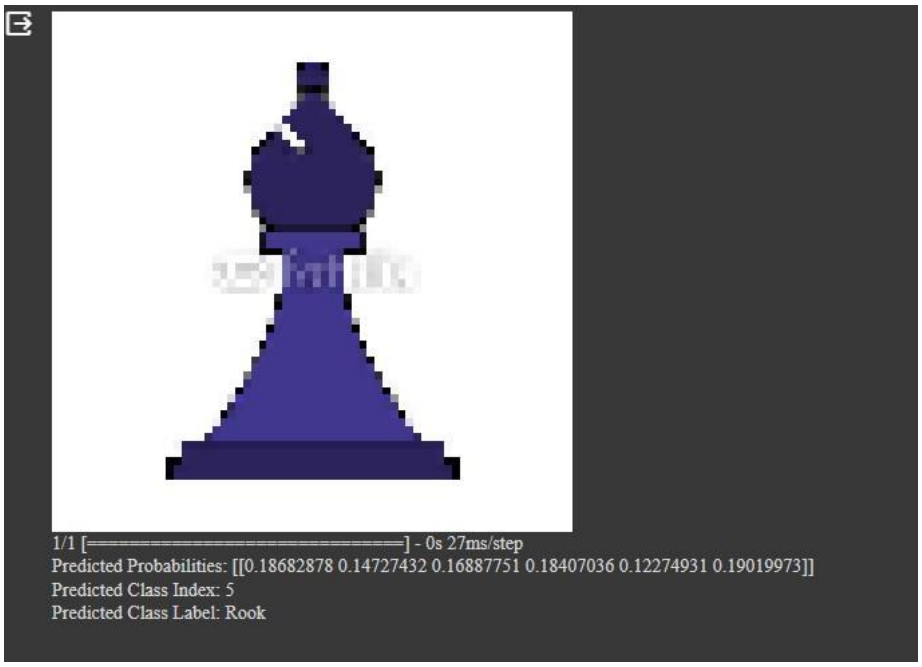
```
# Create a DataFrame from the history object
history_df = pd.DataFrame(history.history)

# Plot the training and validation loss
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(history_df['loss'], label='Training Loss')
plt.plot(history_df['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss')

# Plot the training and validation accuracy
plt.subplot(1, 2, 2)
plt.plot(history_df['accuracy'], label='Training Accuracy')
plt.plot(history_df['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy')

# Show the plots
plt.tight_layout()
plt.show()
```



- Predicting the chess class using CNN base model



```
# Load and preprocess the image you want to make predictions on
img_path = '/content/drive/MyDrive/colabdataset/Chess/Test/Bishop/00000175.jpg'
img = image.load_img(img_path, target_size=(64, 64))
img_array = image.img_to_array(img)
plt.imshow(img_array.astype('uint8'))  # Ensure data type is uint8 for display
plt.axis('off')  # Turn off axis labels
plt.show()
img_array = np.expand_dims(img_array, axis=0)  # Add an extra dimension for batch size
img_array /= 255.0  # Normalize the pixel values

# Make predictions
predictions = classifier.predict(img_array)

# Print the predicted class probabilities
print("Predicted Probabilities:", predictions)

# Get the predicted class index (class with the highest probability)
predicted_class_index = np.argmax(predictions, axis=1)[0]
print("Predicted Class Index:", predicted_class_index)

# Map the predicted class index to the class label
class_labels = ['King', 'Bishop', 'Knight', 'Pawn', 'Queen', 'Rook']
predicted_class_label = class_labels[predicted_class_index]
print("Predicted Class Label:", predicted_class_label)
```



```
1/1 [==============================] - 0s 27ms/step
Predicted Probabilities: [[0.18682878 0.14727432 0.16887751 0.18407036 0.12274931 0.19019973]]
Predicted Class Index: 5
Predicted Class Label: Rook
```

Inference:

- An image for prediction using the trained model (classifier).
  Here's a breakdown of the code:
- **Load and Display the Image:**
  - ❖ img_path: Path to the image you want to make predictions on.
  - ❖ img = image.load_img(img_path, target_size=(64, 64)): Load the image and resize it to the target size of (64, 64).
  - ❖ img_array = image.img_to_array(img): Convert the image to a NumPy array.
- **Display the Image**:
  - ❖ plt.imshow(img_array.astype('uint8')): Display the image using Matplotlib.
  - ❖ plt.axis('off'): Turn off axis labels.
  - ❖ plt.show(): Show the displayed image.
- **Preprocess the Image for Prediction:**
  - ❖ img_array = np.expand_dims(img_array, axis=0): Add an extra dimension for batch size, as the model expects batches.
  - ❖ img_array /= 255.0: Normalize the pixel values to be in the range [0, 1].
- **Make Predictions:**

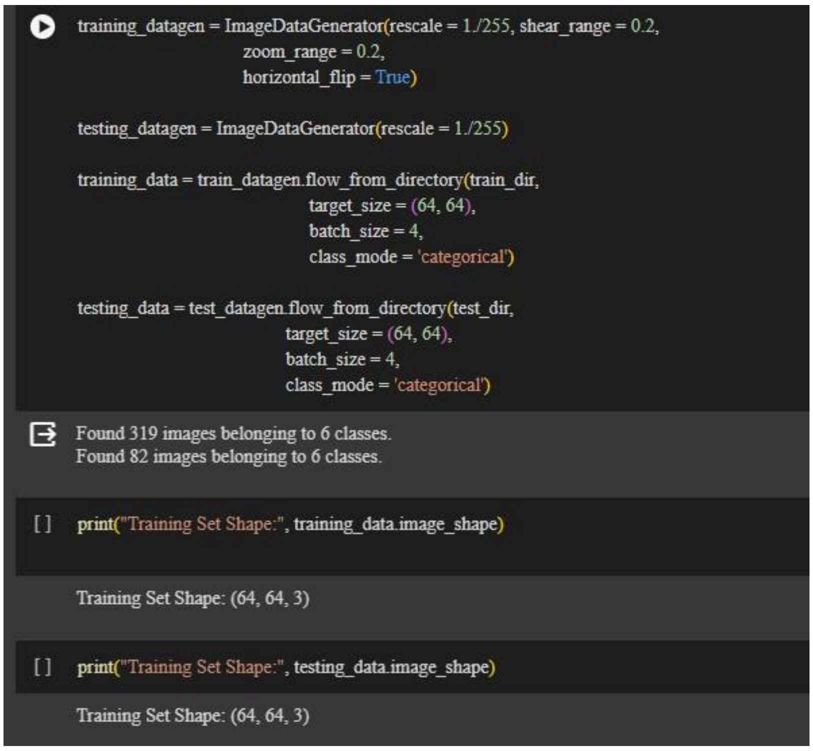
    - ❖ predictions = classifier.predict(img_array): Use the trained model to predict the class probabilities.
  - **Print Predicted Class Probabilities:**
    - ❖ print("Predicted Probabilities:", predictions): Print the predicted class probabilities.
  - **Get Predicted Class Index:**
    - ❖ predicted_class_index = np.argmax(predictions, axis=1)[0]: Get the index of the class with the highest probability.
  - **Map Index to Class Label:**
    - ❖ class_labels = ['King', 'Bishop', 'Knight', 'Pawn', 'Queen', 'Rook']: List of class labels.
    - ❖ predicted_class_label = class_labels[predicted_class_index]: Map the predicted class index to the corresponding class label.
  - **Print Predicted Class Label:**
    - ❖ print("Predicted Class Label:", predicted_class_label): Print the final predicted class label.
- The Base CNN model is built without any augmentation or optimization of the hyper parameter and it works by predicting most of chess labels correctly and fail to predict few classes like Bishop. Let us try to optimize the model by using augmentation, earlystopping, hyperparameter.

Question No:2 (15 marks)

Improve the baseline model (model build in question2) performance and save the weights of improved model Conditions to consider:

- **Apply Data Augmentation if required**
- **No parameter limit**
- **Can use any number of layers**
- **Use any optimizers of your choice**
- **Use early stopping and save best model callbacks Solution:**

- **Data Augmentation:**



```
training_datagen = ImageDataGenerator(rescale = 1./255, shear_range = 0.2,
                        zoom_range = 0.2,
                        horizontal_flip = True)

testing_datagen = ImageDataGenerator(rescale = 1./255)

training_data = train_datagen.flow_from_directory(train_dir,
                            target_size = (64, 64),
                            batch_size = 4,
                            class_mode = 'categorical')

testing_data = test_datagen.flow_from_directory(test_dir,
                            target_size = (64, 64),
                            batch_size = 4,
                            class_mode = 'categorical')
```

```
Found 319 images belonging to 6 classes.
Found 82 images belonging to 6 classes.
```

```
print("Training Set Shape:", training_data.image_shape)
```

```
Training Set Shape: (64, 64, 3)
```

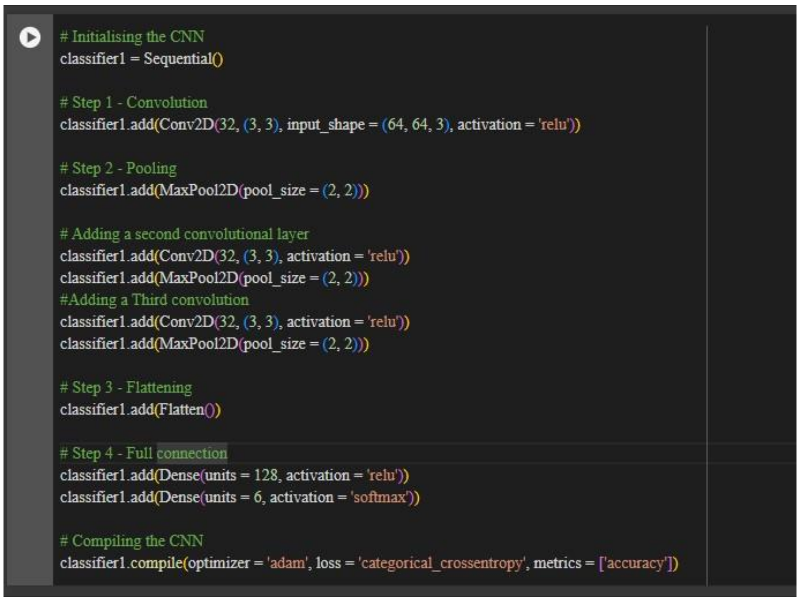
```
print("Training Set Shape:", testing_data.image_shape)
```

```
Training Set Shape: (64, 64, 3)
```

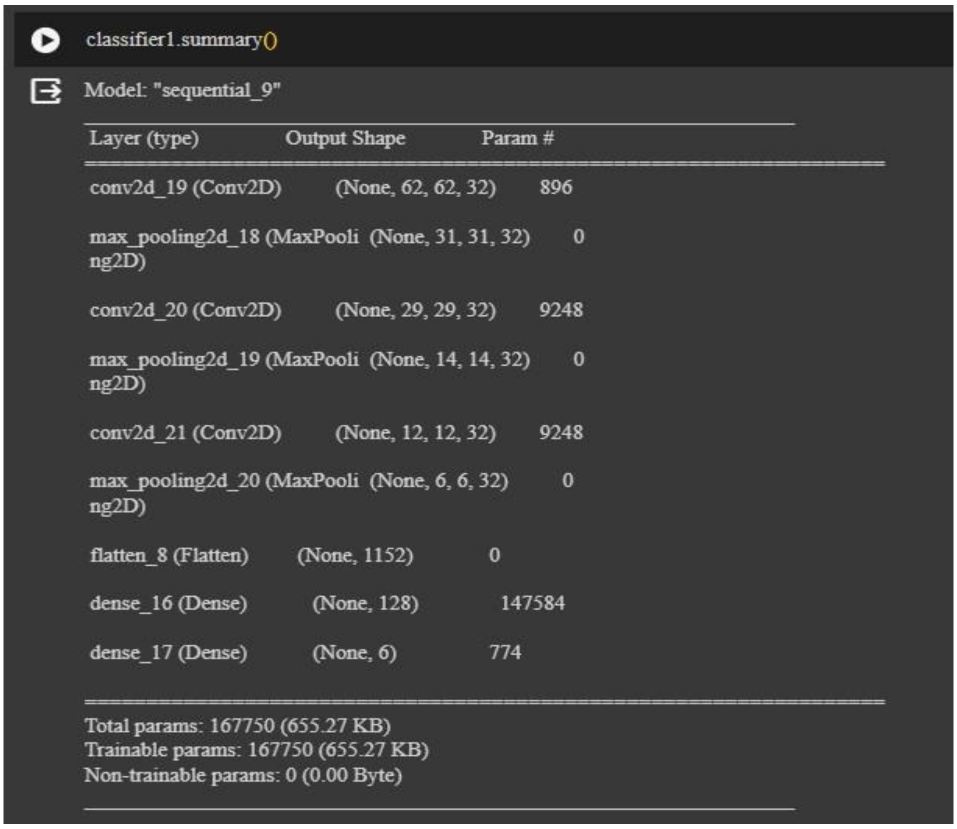
Inference:

- Data augmentation is a technique commonly used in machine learning, and particularly in computer vision tasks such as image classification, to artificially increase the diversity of a training dataset. It involves applying various transformations to the original images, creating new, slightly modified versions.
- Rotation: Rotating the image by a certain angle.
- Shearing: Applying a shear transformation that tilts the image.
- Zooming: Zooming into or out of the image.
- Flipping: Mirroring the image horizontally or vertically.
- Translation: Shifting the image horizontally or vertically.
- Brightness adjustments: Changing the brightness of the image.
- Here, In our CNN model, we used shearing,horizontal flip,zooming to classify the chess coins.
- Image generator is used for data augmentation and reading the images using flow_from_directory function.

- **Building the optimized CNN model by adding one more Convolution layer:**



```
# Initialising the CNN
classifier1 = Sequential()

# Step 1 - Convolution
classifier1.add(Conv2D(32, (3, 3), input_shape = (64, 64, 3), activation = 'relu'))

# Step 2 - Pooling
classifier1.add(MaxPool2D(pool_size = (2, 2)))

# Adding a second convolutional layer
classifier1.add(Conv2D(32, (3, 3), activation = 'relu'))
classifier1.add(MaxPool2D(pool_size = (2, 2)))
#Adding a Third convolution
classifier1.add(Conv2D(32, (3, 3), activation = 'relu'))
classifier1.add(MaxPool2D(pool_size = (2, 2)))

# Step 3 - Flattening
classifier1.add(Flatten())

# Step 4 - Full connection
classifier1.add(Dense(units = 128, activation = 'relu'))
classifier1.add(Dense(units = 6, activation = 'softmax'))

# Compiling the CNN
classifier1.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])
```



```
classifier1.summary()

Model: "sequential_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_19 (Conv2D)          (None, 62, 62, 32)        896

 max_pooling2d_18 (MaxPooli  (None, 31, 31, 32)        0
 ng2D)

 conv2d_20 (Conv2D)          (None, 29, 29, 32)        9248

 max_pooling2d_19 (MaxPooli  (None, 14, 14, 32)        0
 ng2D)

 conv2d_21 (Conv2D)          (None, 12, 12, 32)        9248

 max_pooling2d_20 (MaxPooli  (None, 6, 6, 32)          0
 ng2D)

 flatten_8 (Flatten)         (None, 1152)              0

 dense_16 (Dense)            (None, 128)               147584

 dense_17 (Dense)            (None, 6)                 774

=================================================================
Total params: 167750 (655.27 KB)
Trainable params: 167750 (655.27 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Inference:

- We are building a Convolutional Neural Network (CNN) for a chess piece classification task.
  Here's a breakdown of your code:

- **Sequential Model Initialization:**

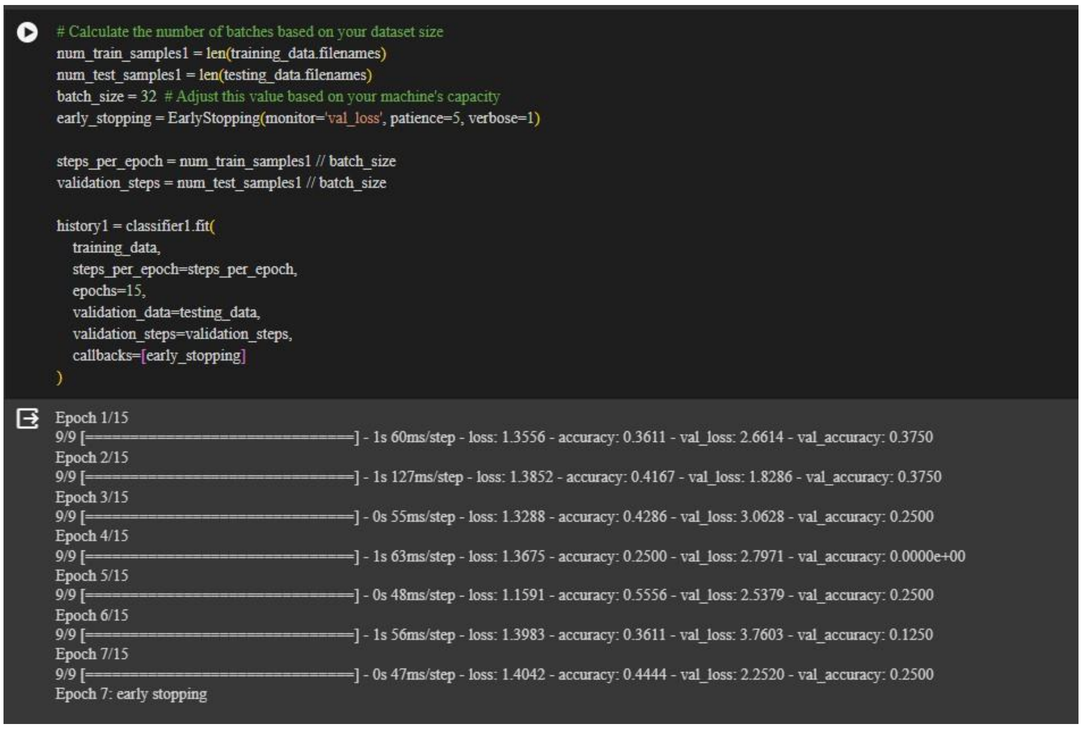
❖

classifier = Sequential(): Initializes a sequential model.

- **Convolutional Layers:**
  - ❖ **Conv2D(32, (3, 3), input_shape=(64, 64, 3), activation='relu'):** First convolutional layer with 32 filters, a kernel size of (3, 3), input shape (64, 64, 3) representing a 64x64 image with 3 color channels (RGB), and ReLU activation.
  - ❖ **MaxPool2D(pool_size=(2, 2)):** Max pooling layer with a pool size of (2, 2) to down-sample the spatial dimensions.
  - ❖ **Conv2D(32, (3, 3), activation='relu'):** Second convolutional layer with 32 filters and a kernel size of (3, 3).
  - ❖ **MaxPool2D(pool_size=(2, 2)):** Max pooling layer following the second convolutional layer.
- **Flattening:**
  - ❖ **Flatten():** Flattens the output from the previous layer into a 1D array.
- **Fully Connected (Dense) Layers:**
  - ❖ **Dense(units=32, activation='relu'**): Fully connected layer with 32 units and ReLU activation.
  - ❖ **Dense(units=6, activation='softmax'):** Output layer with 6 units (6 classes for chess pieces) and softmax activation for multi-class classification.
- **Compiling the Model:**
  - ❖ **compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'**]): Configures the model for training with the Adam optimizer, categorical crossentropy loss (suitable for multi-class classification), and accuracy as the evaluation metric.

- The total parameters represent the number of weights and biases in your model. The trainable parameters are those that will be updated during training, while non-trainable parameters are fixed.
- We used 5 layers (3 convolution layer,2 dense layer) .Here, the model learns 167750 learnable parameters.
- The Adam optimizer is an optimization algorithm commonly used in training deep neural networks. It combines ideas from two other popular optimizers: RMSprop and Momentum. The Adam optimizer adapts the learning rates of each parameter individually by considering both the first-order momentum and the second-order scaling of the gradients.

- **Fitting the chess data to new optimized CNN classifier1 Model:**



```
# Calculate the number of batches based on your dataset size
num_train_samples1 = len(training_data.filenames)
num_test_samples1 = len(testing_data.filenames)
batch_size = 32  # Adjust this value based on your machine's capacity
early_stopping = EarlyStopping(monitor='val_loss', patience=5, verbose=1)

steps_per_epoch = num_train_samples1 // batch_size
validation_steps = num_test_samples1 // batch_size

history1 = classifier1.fit(
    training_data,
    steps_per_epoch=steps_per_epoch,
    epochs=15,
    validation_data=testing_data,
    validation_steps=validation_steps,
    callbacks=[early_stopping]
)
```

```
Epoch 1/15
9/9 [==============================] - 1s 60ms/step - loss: 1.3556 - accuracy: 0.3611 - val_loss: 2.6614 - val_accuracy: 0.3750
Epoch 2/15
9/9 [==============================] - 1s 127ms/step - loss: 1.3852 - accuracy: 0.4167 - val_loss: 1.8286 - val_accuracy: 0.3750
Epoch 3/15
9/9 [==============================] - 0s 55ms/step - loss: 1.3288 - accuracy: 0.4286 - val_loss: 3.0628 - val_accuracy: 0.2500
Epoch 4/15
9/9 [==============================] - 1s 63ms/step - loss: 1.3675 - accuracy: 0.2500 - val_loss: 2.7971 - val_accuracy: 0.0000e+00
Epoch 5/15
9/9 [==============================] - 0s 48ms/step - loss: 1.1591 - accuracy: 0.5556 - val_loss: 2.5379 - val_accuracy: 0.2500
Epoch 6/15
9/9 [==============================] - 1s 56ms/step - loss: 1.3983 - accuracy: 0.3611 - val_loss: 3.7603 - val_accuracy: 0.1250
Epoch 7/15
9/9 [==============================] - 0s 47ms/step - loss: 1.4042 - accuracy: 0.4444 - val_loss: 2.2520 - val_accuracy: 0.2500
Epoch 7: early stopping
```

Inference:

- we are continuing the development of your model training code, and you've added early stopping as a callback to your training process. This is a good practice to prevent overfitting and save computational resources.

  Here's a breakdown of the new additions:

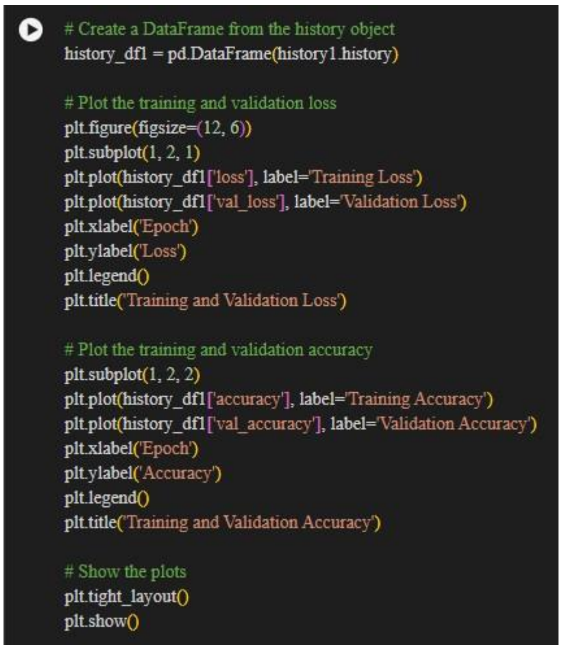
  - **Early Stopping Callback:**
    - **early_stopping:** This callback monitors the validation loss (val_loss) and stops training if there is no improvement after a certain number of epochs (specified by patience). In this case, training will stop after 5 epochs without improvement.
  - **Parameters and Callbacks in fit Method:**
    - **callbacks=[early_stopping]:** This parameter is used to pass a list of callbacks to the fit method. In this case, you're using early stopping as the callback.
  - **Training Process with Early Stopping:**
    - Training will run for a maximum of 15 epochs, but it may stop earlier if the validation loss does not improve for 5 consecutive epochs.

This setup ensures that your model will stop training if the validation loss does not decrease for 5 consecutive epochs, thus preventing overfitting and potentially saving time during training. The training of

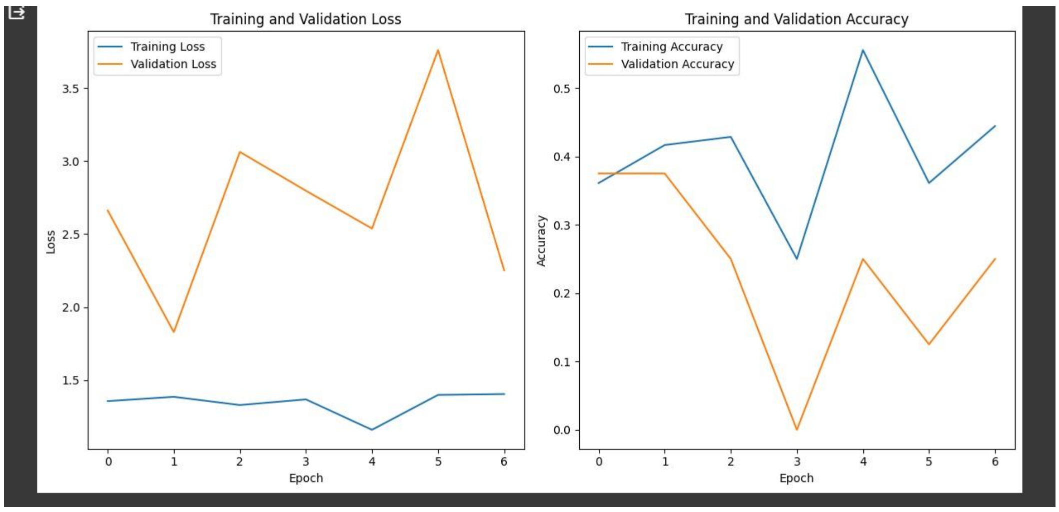
❖

Classifier1 CNN model stops at 7 epochs after waiting for 5 consecutive epoch and provides the accuracy of 40 percentage.

- **Evaluate the optimized classifer1 model:**



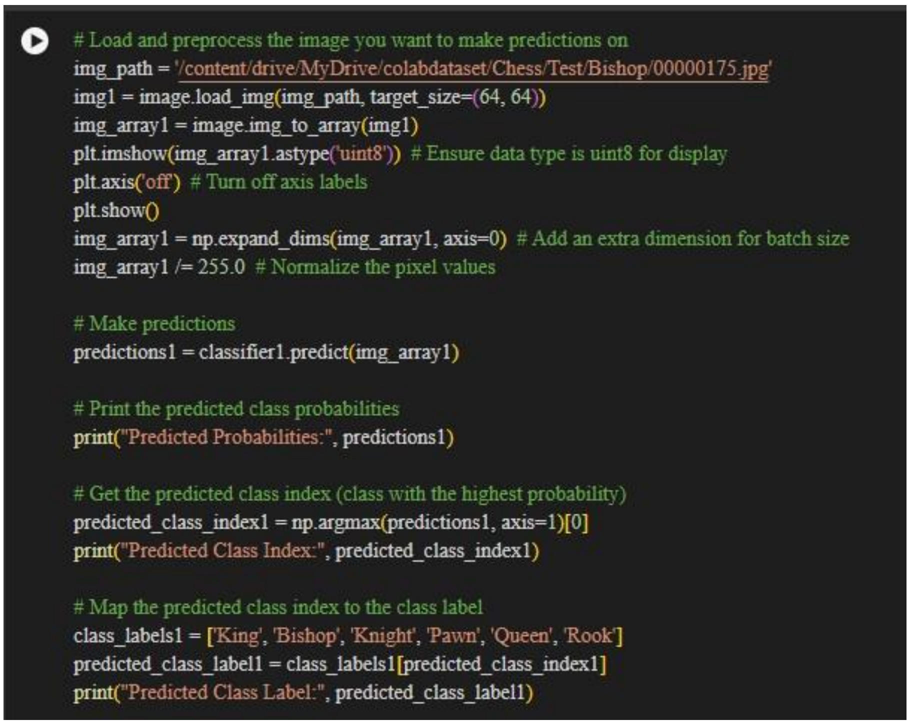
```
# Create a DataFrame from the history object
history_df1 = pd.DataFrame(history1.history)

# Plot the training and validation loss
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(history_df1['loss'], label='Training Loss')
plt.plot(history_df1['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss')

# Plot the training and validation accuracy
plt.subplot(1, 2, 2)
plt.plot(history_df1['accuracy'], label='Training Accuracy')
plt.plot(history_df1['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy')

# Show the plots
plt.tight_layout()
plt.show()
```



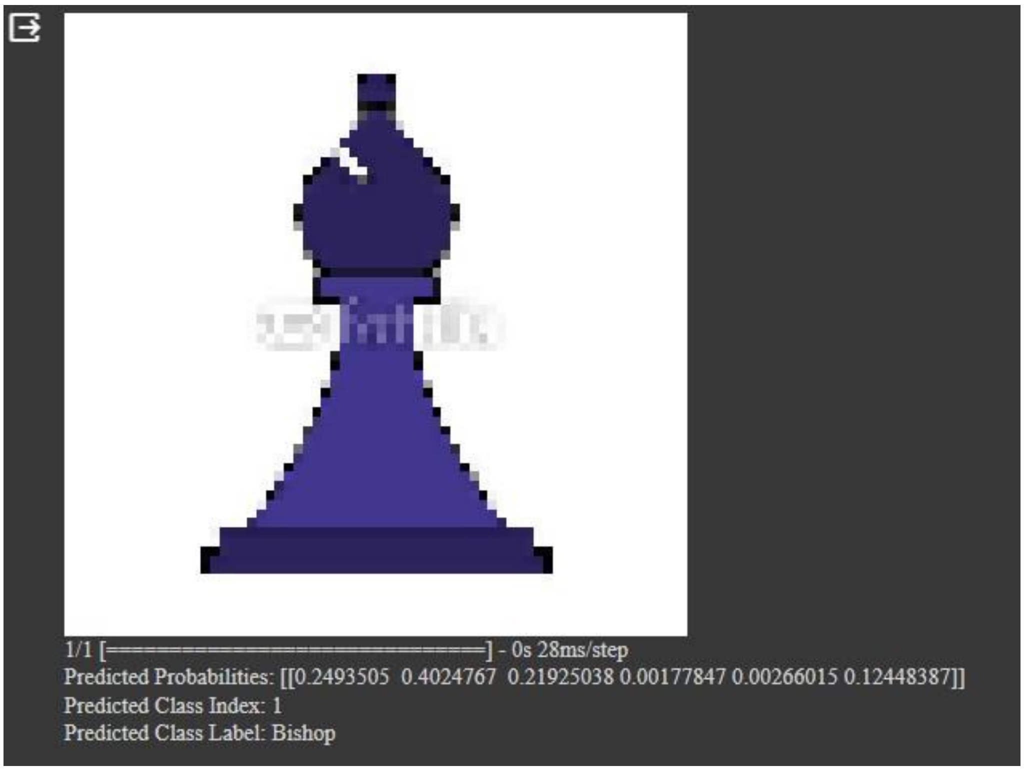
Inference:

- The model evaluated and its loss and accuracy are plotted using matplotlib subplots.
- The optimal model is produced by stopping the training at 7 epoch. It prevents the overfitting of the model.

- **Predicting the chess coin label using the classsifier1 model:**



```python
# Load and preprocess the image you want to make predictions on
img_path = '/content/drive/MyDrive/colabdataset/Chess/Test/Bishop/00000175.jpg'
img1 = image.load_img(img_path, target_size=(64, 64))
img_array1 = image.img_to_array(img1)
plt.imshow(img_array1.astype('uint8'))  # Ensure data type is uint8 for display
plt.axis('off')  # Turn off axis labels
plt.show()
img_array1 = np.expand_dims(img_array1, axis=0)  # Add an extra dimension for batch size
img_array1 /= 255.0  # Normalize the pixel values

# Make predictions
predictions1 = classifier1.predict(img_array1)

# Print the predicted class probabilities
print("Predicted Probabilities:", predictions1)

# Get the predicted class index (class with the highest probability)
predicted_class_index1 = np.argmax(predictions1, axis=1)[0]
print("Predicted Class Index:", predicted_class_index1)

# Map the predicted class index to the class label
class_labels1 = ['King', 'Bishop', 'Knight', 'Pawn', 'Queen', 'Rook']
predicted_class_label1 = class_labels1[predicted_class_index1]
print("Predicted Class Label:", predicted_class_label1)
```



```
1/1 [==============================] - 0s 28ms/step
Predicted Probabilities: [[0.2493505  0.4024767  0.21925038 0.00177847 0.00266015 0.12448387]]
Predicted Class Index: 1
Predicted Class Label: Bishop
```

Inference:

- An image for prediction using the trained model (classifier). Here's a breakdown of the code:
  - **Load and Display the Image:**
    - ❖ img_path: Path to the image you want to make predictions on.

❖

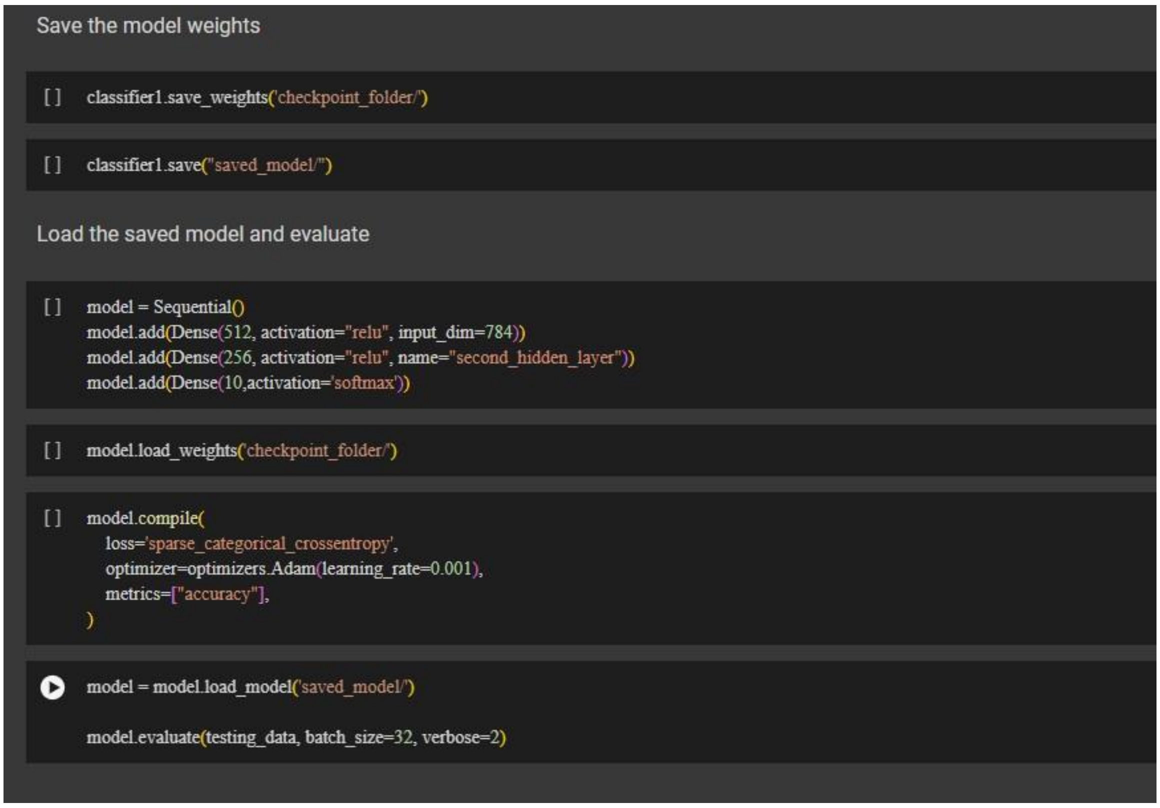
img = image.load_img(img_path, target_size=(64, 64)): Load the image and resize it to the target size of (64, 64).

- ❖ img_array = image.img_to_array(img): Convert the image to a NumPy array.

- **Display the Image**:
  - ❖ plt.imshow(img_array.astype('uint8')): Display the image using Matplotlib.
  - ❖ plt.axis('off'): Turn off axis labels.
  - ❖ plt.show(): Show the displayed image.
- **Preprocess the Image for Prediction:**
  - ❖ img_array = np.expand_dims(img_array, axis=0): Add an extra dimension for batch size, as the model expects batches.
  - ❖ img_array /= 255.0: Normalize the pixel values to be in the range [0, 1].
- **Make Predictions:**
  - ❖ predictions = classifier.predict(img_array): Use the trained model to predict the class probabilities.
- **Print Predicted Class Probabilities:**
  - ❖ print("Predicted Probabilities:", predictions): Print the predicted class probabilities.
- **Get Predicted Class Index:**
  - ❖ predicted_class_index = np.argmax(predictions, axis=1)[0]: Get the index of the class with the highest probability.
- **Map Index to Class Label:**
  - ❖ class_labels = ['King', 'Bishop', 'Knight', 'Pawn', 'Queen', 'Rook']: List of class labels.
  - ❖ predicted_class_label = class_labels[predicted_class_index]: Map the predicted class index to the corresponding class label.
- **Print Predicted Class Label:**
  - ❖ print("Predicted Class Label:", predicted_class_label): Print the final predicted class label.

- The optimized classifier1 CNN model is builtby applying augmentation ,optimization of the hyper parameter and it works good in predicting most of chess labels correctly compared to the Base Model . As you see above, the model correctly predicts the chess coin image as “Bishop”.

- **Save the Model weights and Load the saved Model**



Save the model weights

```
[ ]  classifier1.save_weights('checkpoint_folder/')
```

```
[ ]  classifier1.save("saved_model/")
```

Load the saved model and evaluate

```
[ ]  model = Sequential()
     model.add(Dense(512, activation="relu", input_dim=784))
     model.add(Dense(256, activation="relu", name="second_hidden_layer"))
     model.add(Dense(10,activation='softmax'))
```

```
[ ]  model.load_weights('checkpoint_folder/')
```

```
[ ]  model.compile(
       loss='sparse_categorical_crossentropy',
       optimizer=optimizers.Adam(learning_rate=0.001),
       metrics=["accuracy"],
     )
```

```
     model = model.load_model('saved_model/')

     model.evaluate(testing_data, batch_size=32, verbose=2)
```

Inference:

- The save_weights method in Keras allows you to save the weights of a model to a file. In your case, you're saving the weights of classifier1 to a folder named 'checkpoint_folder'. This is a good practice as it allows you to later load these weights and use the trained model for making predictions or further training.
- We are using the save method to save the entire model (classifier1) to a directory named "saved_model/". This is a good practice as it not only saves the model's weights but also its architecture and optimizer configuration.
- load_model function to load the entire model, including its architecture, weights, and optimizer state. After loading the model, you can then use the evaluate method to evaluate its performance on the testing data.