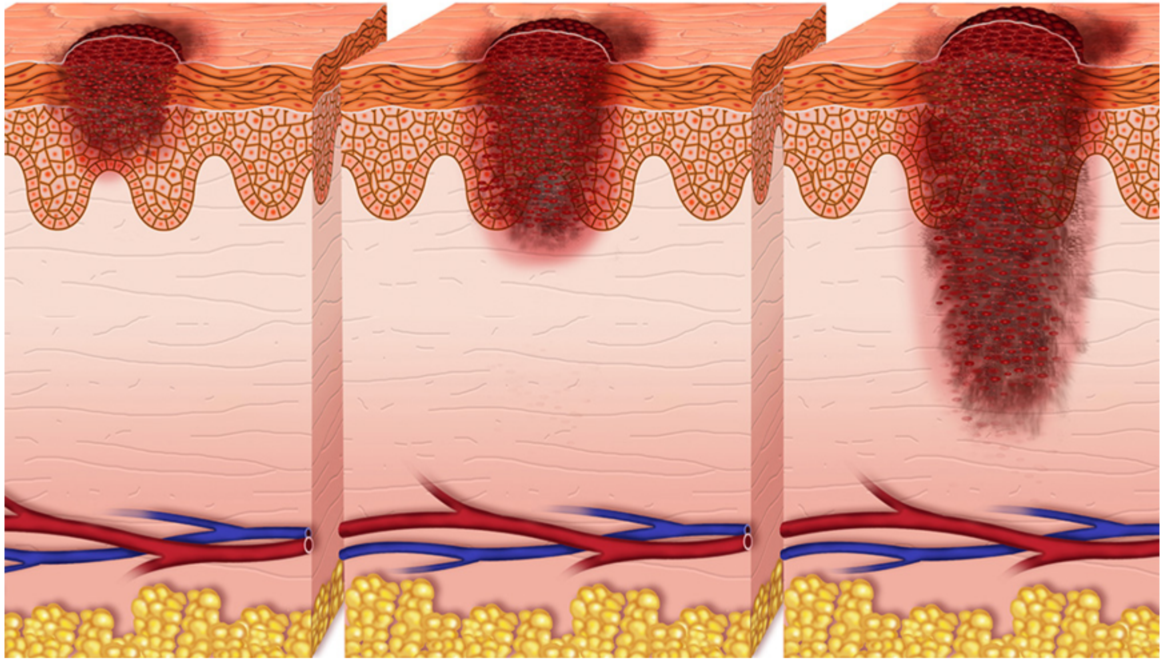
MULTI-CLASS IMAGE CLASSIFICATION OF SKIN DISEASES USING CNN



1.0 BUSINESS UNDERSTANDING

Skin diseases present in many different forms impacting individuals'overall health and well-being. Some of these skin diseases can be challenging to categorize and detect, which introduces complexity to the field of dermatology. The importance of accurate diagnosis cannot be overstated, as certain skin disorders, including various types of skin cancer, have the potential to be life-threatening. Early and accurate identification of the types of skin diseases is of great importance. The diagnostic process typically involves a range of methods, including visual image inspections, biopsies, and histopathological analyses. Distinguishing between benign and malignant lesions is particularly important as even minor or inconspicuous abnormalities can be difficult to detect. In response to these diagnostic challenges, cutting-edge technologies like deep learning algorithms offer the potential to revolutionize dermatological diagnostics, enhancing efficiency, reducing errors, and ultimately improving patient outcomes across various skin disordes types. Read more on below links:

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5817488/

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6403009/

1.1 Problem Statement

Dermatologists at Flatter Dermatological Clinic are facing difficulties in accurately determining or categorizing various skin conditions types when examining medical skin images. Currently, this task heavily relies on manual visual inspection and personal judgment which is time-consuming, prone to human error and can result in delayed or inaccurate diagnoses. This inefficiency increases the chances of missing important skin conditions patterns and making mistakes which could have life-threatening consequences.

1.2 Objectives

Main Objective: To build a convolutional neural network model capable of classifying the 9 different types of skin diseases with over 70% precision.

Other objectives are;

- To explore the distribution of the different types or class of skin images in the dataset.
- To assess the quality and consistency of images in the dataset per class.

Importing the relevant libraries

```
import cv2
import PIL
import glob
import time
import scipy
import random
import pathlib
import warnings
import os,shutil
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf
```

```
import matplotlib.pyplot as plt
from tensorflow import keras
from scipy import ndimage
from sklearn.manifold import TSNE
from tensorflow.keras import layers
from skimage import io, color, feature
from keras.utils import to_categorical
from sklearn.metrics import roc_curve, auc
from keras.metrics import Precision, Recall
from tensorflow.keras.regularizers import l2
from sklearn.metrics import confusion_matrix
from tensorflow.keras.optimizers import Adam
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
from skimage.feature import graycomatrix, graycoprops
from keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import VGG16, ResNet50
from sklearn.utils.class_weight import compute_class_weight
from tensorflow.keras.callbacks import LearningRateScheduler
from tensorflow.keras.optimizers.schedules import ExponentialDecay
from tensorflow.keras.utils import array_to_img, img_to_array, load_img
from tensorflow.keras.layers import GlobalAveragePooling2D, Dense, Dropout, LeakyReLU, BatchNormalization, Activation
warnings.filterwarnings("ignore")
```

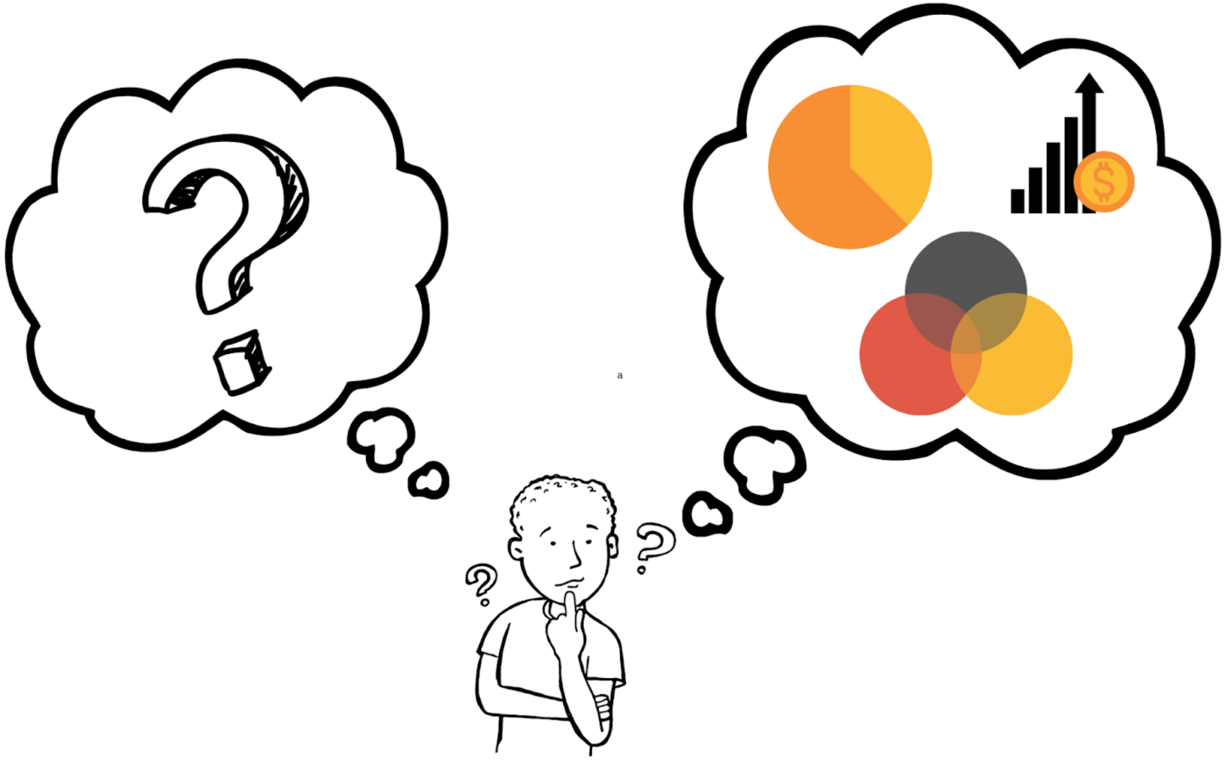
```
#Creating a directory within Colab environment
!mkdir -p /content/mydrive
```

```
# mounting Google Drive into the directory
from google.colab import drive
drive.mount('/content/mydrive', force_remount=True)
```

```
    Mounted at /content/mydrive
```

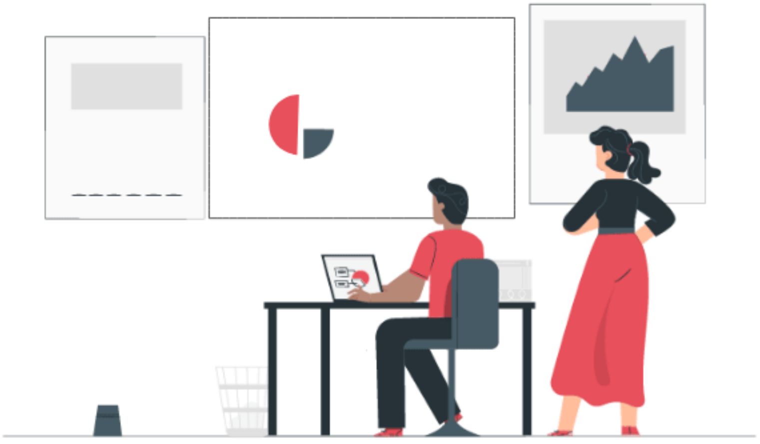
```
# Defining the path for train and test images
train_data_dir = pathlib.Path("/content/mydrive/MyDrive/CNN_PROJECT/data_cnn/Train")
test_data_dir = pathlib.Path("/content/mydrive/MyDrive/CNN_PROJECT/data_cnn/Test")
```

2.0 DATA UNDERSTANDING



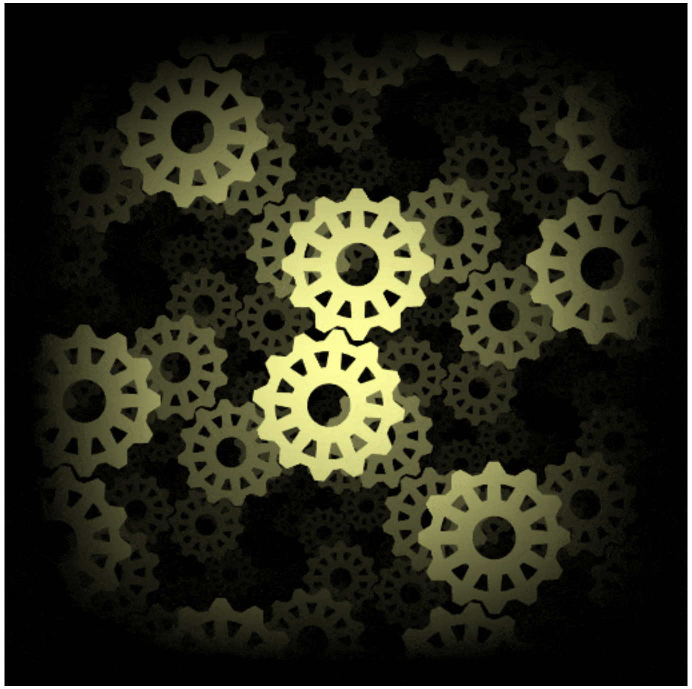
[ ] ↳ *5 cells hidden*

3.0 EXPLORATIVE DATA ANALYSIS (EDA)



[ ] ↳ 33 cells hidden

4.0 DATA PREPROCESSING



In the upcoming code cells, we will perform the following preprocessing steps on our image dataset:

- Create a new directory named 'split_cnn' to organize images that have been split from the original Train set into Training and Validation sets, while keeping the original Train set intact.
- Verify the counts of the split datasets and ensure the integrity of the original test set.
- Normalize our images by rescaling pixel values to ensure uniformity, and resize them to meet specific size requirements for optimal model performance.
- Address class imbalance issues, where some classes may have significantly more instances than others, to avoid bias towards classes with a higher number of images.

4.1 Splitting Data

```
# Defining the destination directories for the training and validation sets
training_data_dir = pathlib.Path("/content/mydrive/MyDrive/CNN_PROJECT/split_cnn/Training_set")
validation_data_dir = pathlib.Path("/content/mydrive/MyDrive/CNN_PROJECT/split_cnn/Validation_set")

# Creating the destination directories
training_data_dir.mkdir(parents=True, exist_ok=True)
validation_data_dir.mkdir(parents=True, exist_ok=True)

class_dirs = [dir.name for dir in test_data_dir.glob('*')]
```

```
# Splitting the data into training and validation sets while maintaining the original directory structure
for class_dir in class_dirs:
    class_images = list((train_data_dir / class_dir).glob('*.jpg'))
    train_images, val_images = train_test_split(class_images, test_size=0.4, random_state=123)
    (training_data_dir / class_dir).mkdir(parents=True, exist_ok=True)
    (validation_data_dir / class_dir).mkdir(parents=True, exist_ok=True)

    # Copying the images to their respective directories
    for train_image in train_images:
        shutil.copy(train_image, training_data_dir / class_dir / train_image.name)

    for val_image in val_images:
        shutil.copy(val_image, validation_data_dir / class_dir / val_image.name)

print("Data splitting complete.")
```

```
Data splitting complete.
```

We have opted to use a 60:40 split to ensure that we strike a balance between having enough data for training and having a sufficiently large validation set to assess model performance effectively. Given that we already have a separate test set, further dividing the training data into training and validation subsets allows us to evaluate model performance effectively without overly diminishing the size of the training dataset.

4.2 Count of image after split

```
#Total Count for created Training_set, Validation_set and Original Test
image_count_training = len(list(training_data_dir.glob('*/*.jpg')))
print("Training images:", image_count_training)
image_count_validation = len(list(validation_data_dir.glob('*/*.jpg')))
print("Validation images:", image_count_validation)
image_count_test = len(list(test_data_dir.glob('*/*.jpg')))
print("Test images:", image_count_test)
```

```
Training images: 1340
Validation images: 899
Test images: 118
```

4.3 Rescaling and Resizing

```
# transforming data in the directory split/train (1574 images)
batch_size =32
class_mode = 'categorical'
train_generator = ImageDataGenerator(rescale=1./255).flow_from_directory(
        training_data_dir,
        target_size=(64, 64), batch_size = batch_size, class_mode = class_mode)

# transforming  data in the directory split/validation (677 images)
val_generator = ImageDataGenerator(rescale=1./255).flow_from_directory(
        validation_data_dir,
        target_size=(64, 64), batch_size = batch_size, class_mode = class_mode)
```

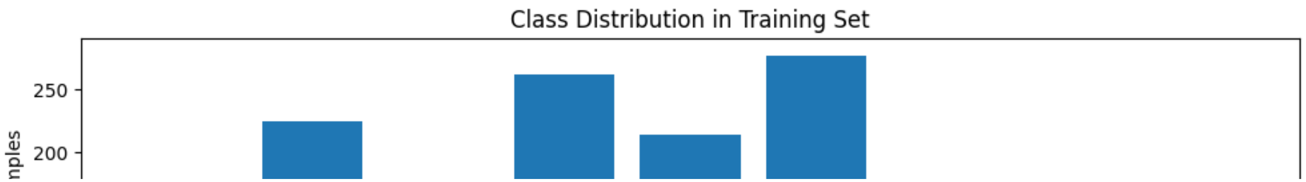
```
# tranforming the data in the directory test (118 images)
test_generator = ImageDataGenerator(rescale=1./255).flow_from_directory(
        test_data_dir,
        target_size=(64, 64), batch_size=batch_size, class_mode = class_mode)
```

```
Found 1340 images belonging to 9 classes.
Found 899 images belonging to 9 classes.
Found 118 images belonging to 9 classes.
```

The above code loads and preprocess the image datasets in a memory-efficient manner, especially when dealing with large datasets. The generators yield batches of data and labels that can be fed directly into the model during training, validation, or testing.

4.4 Checking for class imbalance

```
# class labels and counts for the training set
class_labels = train_generator.class_indices
class_counts = train_generator.classes

# Calculating the number of samples in each class
unique_classes, class_counts = np.unique(class_counts, return_counts=True)

# Mapping class labels to class names
class_names = {v: k for k, v in class_labels.items()}

plt.figure(figsize=(10, 5))
plt.bar(class_names.values(), class_counts)
plt.title('Class Distribution in Training Set')
plt.xlabel('Class')
plt.ylabel('Number of Samples')
plt.xticks(rotation=45, ha="right")  # Rotate x-axis labels for readability
plt.tight_layout()
plt.show()
```



The visualization above illustrates the class distribution, highlighting significant imbalances among the classes. Specifically, certain classes, such as sebarrheic keratosis, dermatofibroma, and actinic keratosis, exhibit considerably smaller sample sizes. In contrast, classes like melanoma, pigmented benign keratosis, and basal cell carcinoma are characterized by larger sample sizes. We shall use the augmenter to deal with imbalance in the second model.

5.0 MODELING

5.1 Baseline Model

```
# creating a function for model 1 and 2
def create_and_train_model(model_number, train_generator, val_generator):
```

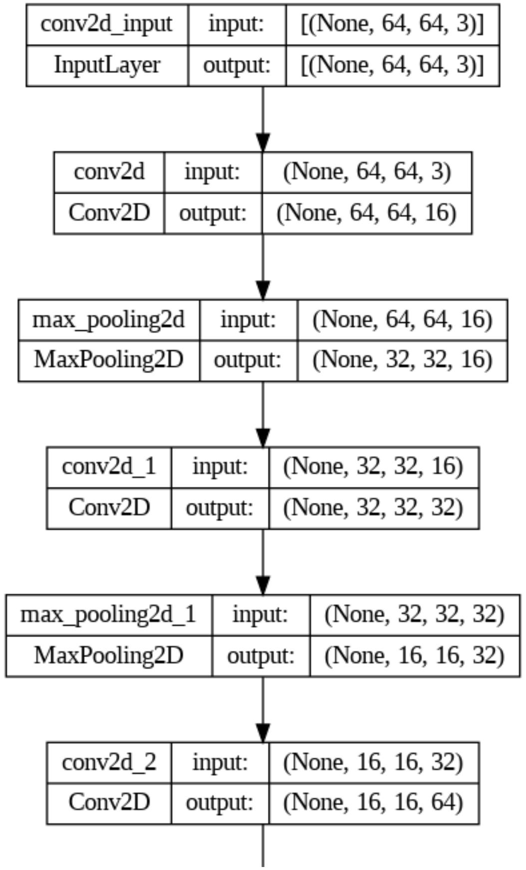
```
    # Creating the model layers
    early_stopping = EarlyStopping(monitor='val_loss',
                                   patience=10,
                                   restore_best_weights=True)

    np.random.seed(123)
    model = Sequential()
    model.add(layers.Conv2D(16, (3, 3), activation='relu', padding='same',
                            input_shape=(64, 64, 3)))
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Conv2D(32, (4, 4), activation='relu', padding='same'))
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Flatten())
    model.add(layers.Dense(128, activation='relu'))
    model.add(layers.Dense(9, activation='softmax'))

    model.compile(loss='categorical_crossentropy',
                  optimizer="adam",  # You can adjust the learning rate if needed
                  metrics=['accuracy', Precision(), Recall()])

    # Training the model and storing the history
    history = model.fit(train_generator,
                        epochs=100,
                        validation_data=val_generator,
                        callbacks=[early_stopping])

    return model, history


# To create and train model1 and get its history, you can call the function like this:
model1, history1 = create_and_train_model(1, train_generator, val_generator)
```

```
Epoch 1/100
42/42 [==============================] - 42s 717ms/step - loss: 2.0210 - accuracy: 0.2373 - precision: 0.0000e+00 - recall: 0.0000e+00 - val_loss: 1.9243 - val_accuracy: 0.2747 - val_precision: 0
Epoch 2/100
42/42 [==============================] - 24s 587ms/step - loss: 1.7752 - accuracy: 0.3619 - precision: 0.6410 - recall: 0.0746 - val_loss: 1.6679 - val_accuracy: 0.4260 - val_precision: 0.6199 - \
Epoch 3/100
42/42 [==============================] - 24s 560ms/step - loss: 1.6217 - accuracy: 0.4299 - precision: 0.6483 - recall: 0.1582 - val_loss: 1.5777 - val_accuracy: 0.4416 - val_precision: 0.6250 - \
Epoch 4/100
42/42 [==============================] - 24s 586ms/step - loss: 1.5156 - accuracy: 0.4694 - precision: 0.6964 - recall: 0.1866 - val_loss: 1.5078 - val_accuracy: 0.4861 - val_precision: 0.6336 - \
Epoch 5/100
42/42 [==============================] - 24s 584ms/step - loss: 1.4947 - accuracy: 0.4649 - precision: 0.6598 - recall: 0.2142 - val_loss: 1.5625 - val_accuracy: 0.4416 - val_precision: 0.6397 - \
Epoch 6/100
42/42 [==============================] - 26s 631ms/step - loss: 1.4233 - accuracy: 0.4955 - precision: 0.6647 - recall: 0.2575 - val_loss: 1.5365 - val_accuracy: 0.4750 - val_precision: 0.6474 - \
Epoch 7/100
42/42 [==============================] - 28s 655ms/step - loss: 1.3187 - accuracy: 0.5343 - precision: 0.6985 - recall: 0.3164 - val_loss: 1.3839 - val_accuracy: 0.5328 - val_precision: 0.6764 - \
Epoch 8/100
42/42 [==============================] - 23s 543ms/step - loss: 1.2947 - accuracy: 0.5425 - precision: 0.7152 - recall: 0.3373 - val_loss: 1.4079 - val_accuracy: 0.5017 - val_precision: 0.6296 - \
Epoch 9/100
42/42 [==============================] - 24s 575ms/step - loss: 1.2837 - accuracy: 0.5418 - precision: 0.6937 - recall: 0.3634 - val_loss: 1.4277 - val_accuracy: 0.5172 - val_precision: 0.6493 - \
Epoch 10/100
42/42 [==============================] - 24s 571ms/step - loss: 1.2538 - accuracy: 0.5619 - precision: 0.7209 - recall: 0.3701 - val_loss: 1.3475 - val_accuracy: 0.5317 - val_precision: 0.6839 - \
Epoch 11/100
42/42 [==============================] - 23s 550ms/step - loss: 1.1852 - accuracy: 0.5746 - precision: 0.7241 - recall: 0.4015 - val_loss: 1.4179 - val_accuracy: 0.5217 - val_precision: 0.6512 - \
```

```
Epoch 12/100
42/42 [==============================] - 23s 541ms/step - loss: 1.2073 - accuracy: 0.5769 - precision: 0.7097 - recall: 0.3978 - val_loss: 1.4377 - val_accuracy: 0.5184 - val_precision: 0.6501 - v
Epoch 13/100
42/42 [==============================] - 23s 561ms/step - loss: 1.1598 - accuracy: 0.5881 - precision: 0.7260 - recall: 0.4351 - val_loss: 1.3789 - val_accuracy: 0.5317 - val_precision: 0.6653 - v
Epoch 14/100
42/42 [==============================] - 22s 540ms/step - loss: 1.0717 - accuracy: 0.6216 - precision: 0.7623 - recall: 0.4642 - val_loss: 1.3596 - val_accuracy: 0.5562 - val_precision: 0.6580 - v
Epoch 15/100
42/42 [==============================] - 24s 585ms/step - loss: 1.0703 - accuracy: 0.6007 - precision: 0.7589 - recall: 0.4746 - val_loss: 1.4174 - val_accuracy: 0.5095 - val_precision: 0.6200 - v
Epoch 16/100
42/42 [==============================] - 22s 539ms/step - loss: 1.0498 - accuracy: 0.6157 - precision: 0.7430 - recall: 0.4769 - val_loss: 1.3590 - val_accuracy: 0.5528 - val_precision: 0.6760 - v
Epoch 17/100
42/42 [==============================] - 32s 761ms/step - loss: 1.0405 - accuracy: 0.6291 - precision: 0.7875 - recall: 0.4896 - val_loss: 1.4136 - val_accuracy: 0.5451 - val_precision: 0.6450 - v
Epoch 18/100
42/42 [==============================] - 35s 839ms/step - loss: 0.9489 - accuracy: 0.6560 - precision: 0.7688 - recall: 0.5112 - val_loss: 1.4789 - val_accuracy: 0.5217 - val_precision: 0.6200 - v
Epoch 19/100
42/42 [==============================] - 23s 553ms/step - loss: 0.9258 - accuracy: 0.6664 - precision: 0.7919 - recall: 0.5425 - val_loss: 1.4009 - val_accuracy: 0.5406 - val_precision: 0.6160 - v
Epoch 20/100
42/42 [==============================] - 22s 538ms/step - loss: 0.8890 - accuracy: 0.6896 - precision: 0.7981 - recall: 0.5754 - val_loss: 1.3765 - val_accuracy: 0.5539 - val_precision: 0.6667 - v
```

```
# Saving the entire baseline model including architecture and weights
model1.save("model1.h5")
```

```
from tensorflow.keras.models import load_model

# Load the saved baseline model
baseline_model = load_model("model1.h5")

# Visualize the loaded baseline model
from tensorflow.keras.utils import plot_model

# Plot the baseline model architecture
plot_model(baseline_model, to_file='baseline_model.png', show_shapes=True)
```

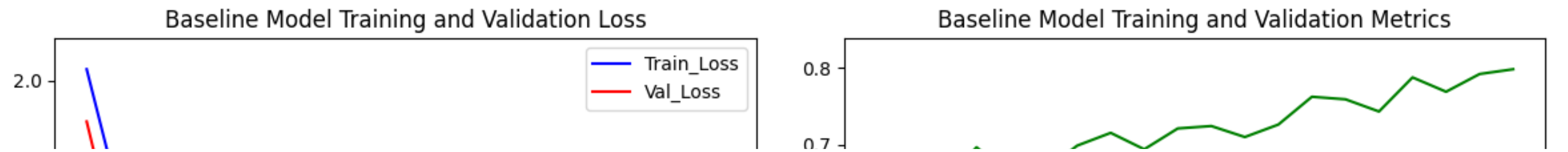



```
# Let us have a close look at precision of the above model
precision = model1.evaluate(train_generator, verbose=1)[2]
precision_v = model1.evaluate(val_generator, verbose=1)[2]
print("Precision: ", precision)
print("Validation Precision: ", precision_v)
```

```
42/42 [==============================] - 14s 328ms/step - loss: 1.1437 - accuracy: 0.5910 - precision: 0.7527 - recall: 0.4224
29/29 [==============================] - 9s 293ms/step - loss: 1.3475 - accuracy: 0.5317 - precision: 0.6839 - recall: 0.3393
Precision:  0.7526595592498779
Validation Precision:  0.6838564872741699
```

```
# Accessing training history from the 'history' object
training_loss = history1.history['loss']
validation_loss = history1.history['val_loss']
training_accuracy = history1.history['accuracy']
validation_accuracy = history1.history['val_accuracy']
training_precision = history1.history['precision']
validation_precision = history1.history['val_precision']
training_recall = history1.history['recall']
validation_recall = history1.history['val_recall']
```

```
# Number of epochs
epochs = range(1, len(training_loss) + 1)

# creating a plot
plt.figure(figsize=(12, 6))
plt.subplot(121)
plt.plot(epochs, training_loss, 'b-', label='Train_Loss')
plt.plot(epochs, validation_loss, 'r-', label='Val_Loss')
plt.title('Baseline Model Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Plotting Metrics
plt.subplot(122)
plt.plot(epochs, training_accuracy, 'b-', label='Train_Accuracy')
plt.plot(epochs, validation_accuracy, 'r-', label='Val_Accuracy')
plt.plot(epochs, training_precision, 'g-', label='Train_Precision')
plt.plot(epochs, validation_precision, 'c-', label='Val_Precision')
plt.plot(epochs, training_recall, 'm-', label='Train_Recall')
plt.plot(epochs, validation_recall, 'y-', label='Val_Recall')
plt.title('Baseline Model Training and Validation Metrics')
plt.xlabel('Epochs')
plt.ylabel('Metrics')
plt.legend()


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



We can see that our Baseline model seems to be overfitting on Accuracy, Precision and Recall Metrics.

- For instance the value of 0.7527 on training Precision means that our model correctly predicted the positive class with a precision of approximately 75.27% while the value of 0.6839 on the validation Precision means that our model correctly predicted the positive class with a precision of approximately 68.39%.
- Our training loss is 1.1437 and validation loss is 1.3475, meaning that the training loss outperforms the validation loss, a sign of overfitting. The lower the losses the better it is for our model to generalize well to new, unseen data.

We shall go ahead and augment our dataset to deal with overfitting and class imbalance

5.2 Models with Data Augmentation

5.2.1 Model 2: Using only Augmented data

```
pip install Augmentor
```

```
    Collecting Augmentor
      Downloading Augmentor-0.2.12-py2.py3-none-any.whl (38 kB)
    Requirement already satisfied: Pillow>=5.2.0 in /usr/local/lib/python3.10/dist-packages (from Augmentor) (9.4.0)
    Requirement already satisfied: tqdm>=4.9.0 in /usr/local/lib/python3.10/dist-packages (from Augmentor) (4.66.1)
    Requirement already satisfied: numpy>=1.11.0 in /usr/local/lib/python3.10/dist-packages (from Augmentor) (1.23.5)
    Installing collected packages: Augmentor
    Successfully installed Augmentor-0.2.12
```

```
import Augmentor
np.random.seed(123)

# Defining a function to create an Augmentor pipeline
def create_augmentor_pipeline(input_dir, output_dir, num_samples):
    p = Augmentor.Pipeline(input_dir, output_dir)
    p.rotate(probability=0.7, max_left_rotation=25, max_right_rotation=25)
    p.zoom_random(probability=0.5, percentage_area=0.8)
    p.random_contrast(probability=0.5, min_factor=0.7, max_factor=1.3)
    p.random_brightness(probability=0.5, min_factor=0.7, max_factor=1.3)
    p.random_color(probability=0.5, min_factor=0.5, max_factor=2.0)
    p.random_distortion(probability=0.5, grid_width=4, grid_height=4, magnitude=8)
    p.flip_left_right(probability=0.5)
    p.random_erasing(probability=0.2, rectangle_area=0.2)
    p.sample(4000)

# Define the directories for your augmented data
input_dir = '/content/mydrive/MyDrive/CNN_PROJECT/split_cnn/Training_set'
output_dir = '/content/mydrive/MyDrive/CNN_PROJECT/split_cnn/Augmented_set1'
```

```
# Create an Augmentor pipeline and generate augmented data
create_augmentor_pipeline(input_dir, output_dir, num_samples=4000)
```

```
Initialised with 1340 image(s) found.
Output directory set to /content/mydrive/MyDrive/CNN_PROJECT/split_cnn/Augmented_set1.Processing <PIL.Image.Image image mode=RGB size=600x450 at 0x7ECD459D1E70>: 100%|██████████| 4000/4000 [10:52<
```

```
output_dir = '/content/mydrive/MyDrive/CNN_PROJECT/split_cnn/Augmented_set1'
# Now, create data generators for training and validation
train_datagen = tf.keras.preprocessing.image.ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

# Load augmented data from the output directory
train_generator2 = train_datagen.flow_from_directory(
    output_dir,
    target_size=(64, 64),
    batch_size=32,
    class_mode='categorical'
)
```

```
Found 4000 images belonging to 9 classes.
```

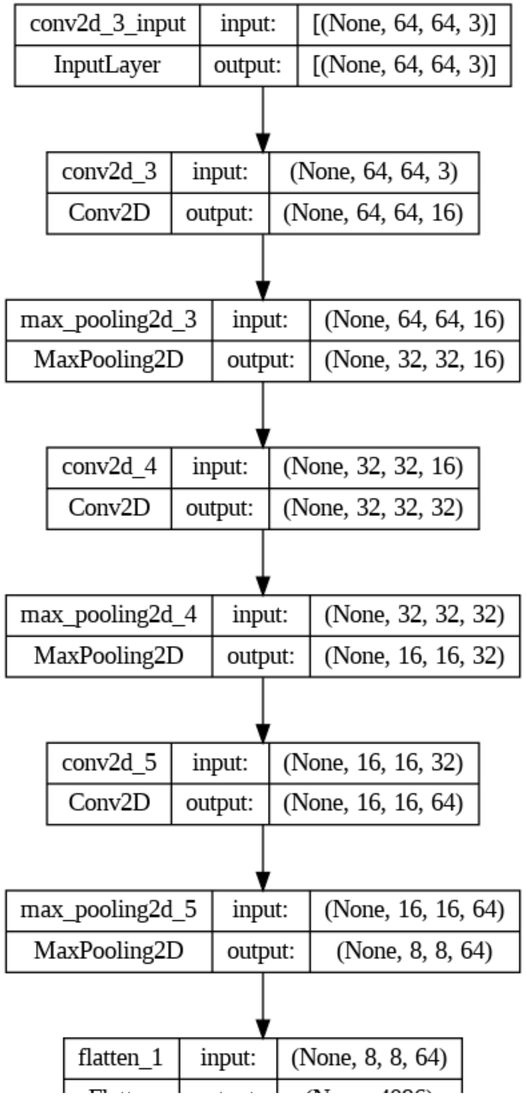
```
# To create and train model2 and get its history, you can call the function like this:
model2, history2 = create_and_train_model(2, train_generator2, val_generator)
```

```
Epoch 1/100
125/125 [==============================] - 1310s 10s/step - loss: 1.8375 - accuracy: 0.3290 - precision_1: 0.5325 - recall_1: 0.0553 - val_loss: 1.6535 - val_accuracy: 0.3960 - val_precision_1
Epoch 2/100
125/125 [==============================] - 35s 284ms/step - loss: 1.6158 - accuracy: 0.4033 - precision_1: 0.6175 - recall_1: 0.1392 - val_loss: 1.4977 - val_accuracy: 0.4850 - val_precision_1
Epoch 3/100
125/125 [==============================] - 37s 294ms/step - loss: 1.5307 - accuracy: 0.4478 - precision_1: 0.6365 - recall_1: 0.1900 - val_loss: 1.3972 - val_accuracy: 0.5083 - val_precision_1
Epoch 4/100
125/125 [==============================] - 34s 276ms/step - loss: 1.4657 - accuracy: 0.4708 - precision_1: 0.6451 - recall_1: 0.2368 - val_loss: 1.5470 - val_accuracy: 0.4538 - val_precision_1
Epoch 5/100
125/125 [==============================] - 36s 286ms/step - loss: 1.4442 - accuracy: 0.4827 - precision_1: 0.6440 - recall_1: 0.2533 - val_loss: 1.3602 - val_accuracy: 0.5284 - val_precision_1
Epoch 6/100
125/125 [==============================] - 36s 291ms/step - loss: 1.4186 - accuracy: 0.4897 - precision_1: 0.6480 - recall_1: 0.2697 - val_loss: 1.3435 - val_accuracy: 0.5228 - val_precision_1
Epoch 7/100
125/125 [==============================] - 37s 295ms/step - loss: 1.3845 - accuracy: 0.5055 - precision_1: 0.6811 - recall_1: 0.2878 - val_loss: 1.4435 - val_accuracy: 0.4739 - val_precision_1
Epoch 8/100
125/125 [==============================] - 36s 291ms/step - loss: 1.3553 - accuracy: 0.5110 - precision_1: 0.6722 - recall_1: 0.3045 - val_loss: 1.3798 - val_accuracy: 0.5206 - val_precision_1
Epoch 9/100
125/125 [==============================] - 36s 291ms/step - loss: 1.3569 - accuracy: 0.5077 - precision_1: 0.6648 - recall_1: 0.3025 - val_loss: 1.2992 - val_accuracy: 0.5395 - val_precision_1
Epoch 10/100
125/125 [==============================] - 36s 285ms/step - loss: 1.3241 - accuracy: 0.5268 - precision_1: 0.6837 - recall_1: 0.3205 - val_loss: 1.2990 - val_accuracy: 0.5250 - val_precision_1
Epoch 11/100
125/125 [==============================] - 36s 291ms/step - loss: 1.3088 - accuracy: 0.5180 - precision_1: 0.6808 - recall_1: 0.3220 - val_loss: 1.3070 - val_accuracy: 0.5428 - val_precision_1
Epoch 12/100
```

```
125/125 [==============================] - 35s 280ms/step - loss: 1.3006 - accuracy: 0.5225 - precision_1: 0.6794 - recall_1: 0.3433 - val_loss: 1.2620 - val_accuracy: 0.5617 - val_precision_1
Epoch 13/100
125/125 [==============================] - 36s 292ms/step - loss: 1.2927 - accuracy: 0.5205 - precision_1: 0.7002 - recall_1: 0.3380 - val_loss: 1.2809 - val_accuracy: 0.5228 - val_precision_1
Epoch 14/100
125/125 [==============================] - 35s 283ms/step - loss: 1.2838 - accuracy: 0.5332 - precision_1: 0.6917 - recall_1: 0.3550 - val_loss: 1.3058 - val_accuracy: 0.5284 - val_precision_1
Epoch 15/100
125/125 [==============================] - 38s 305ms/step - loss: 1.2446 - accuracy: 0.5527 - precision_1: 0.7102 - recall_1: 0.3688 - val_loss: 1.3166 - val_accuracy: 0.5295 - val_precision_1
Epoch 16/100
125/125 [==============================] - 35s 279ms/step - loss: 1.2563 - accuracy: 0.5455 - precision_1: 0.7034 - recall_1: 0.3670 - val_loss: 1.2493 - val_accuracy: 0.5451 - val_precision_1
Epoch 17/100
125/125 [==============================] - 37s 299ms/step - loss: 1.2345 - accuracy: 0.5508 - precision_1: 0.7032 - recall_1: 0.3820 - val_loss: 1.2749 - val_accuracy: 0.5484 - val_precision_1
Epoch 18/100
125/125 [==============================] - 35s 279ms/step - loss: 1.2226 - accuracy: 0.5418 - precision_1: 0.7000 - recall_1: 0.3792 - val_loss: 1.2557 - val_accuracy: 0.5395 - val_precision_1
Epoch 19/100
125/125 [==============================] - 36s 288ms/step - loss: 1.2158 - accuracy: 0.5573 - precision_1: 0.7099 - recall_1: 0.3835 - val_loss: 1.2464 - val_accuracy: 0.5651 - val_precision_1
Epoch 20/100
125/125 [==============================] - 37s 293ms/step - loss: 1.2058 - accuracy: 0.5625 - precision_1: 0.7136 - recall_1: 0.3930 - val_loss: 1.2334 - val_accuracy: 0.5640 - val_precision_1
Epoch 21/100
125/125 [==============================] - 35s 279ms/step - loss: 1.1961 - accuracy: 0.5590 - precision_1: 0.7227 - recall_1: 0.3988 - val_loss: 1.2760 - val_accuracy: 0.5662 - val_precision_1
Epoch 22/100
125/125 [==============================] - 37s 294ms/step - loss: 1.1698 - accuracy: 0.5795 - precision_1: 0.7164 - recall_1: 0.4205 - val_loss: 1.2922 - val_accuracy: 0.5417 - val_precision_1
Epoch 23/100
125/125 [==============================] - 35s 280ms/step - loss: 1.1761 - accuracy: 0.5702 - precision_1: 0.7175 - recall_1: 0.4058 - val_loss: 1.2586 - val_accuracy: 0.5506 - val_precision_1
Epoch 24/100
125/125 [==============================] - 36s 285ms/step - loss: 1.1462 - accuracy: 0.5792 - precision_1: 0.7129 - recall_1: 0.4185 - val_loss: 1.2455 - val_accuracy: 0.5573 - val_precision_1
Epoch 25/100
125/125 [==============================] - 36s 293ms/step - loss: 1.1625 - accuracy: 0.5690 - precision_1: 0.7188 - recall_1: 0.4148 - val_loss: 1.2844 - val_accuracy: 0.5628 - val_precision_1
Epoch 26/100
125/125 [==============================] - 35s 278ms/step - loss: 1.1394 - accuracy: 0.5810 - precision_1: 0.7244 - recall_1: 0.4257 - val_loss: 1.2581 - val_accuracy: 0.5584 - val_precision_1
Epoch 27/100
125/125 [==============================] - 35s 281ms/step - loss: 1.1164 - accuracy: 0.5978 - precision_1: 0.7308 - recall_1: 0.4465 - val_loss: 1.3257 - val_accuracy: 0.5428 - val_precision_1
Epoch 28/100
125/125 [==============================] - 37s 293ms/step - loss: 1.1342 - accuracy: 0.5803 - precision_1: 0.7218 - recall_1: 0.4333 - val_loss: 1.2558 - val_accuracy: 0.5428 - val_precision_1
```

```
# Saving the entire model2 including architecture and weights
model2.save("model2.h5")
```

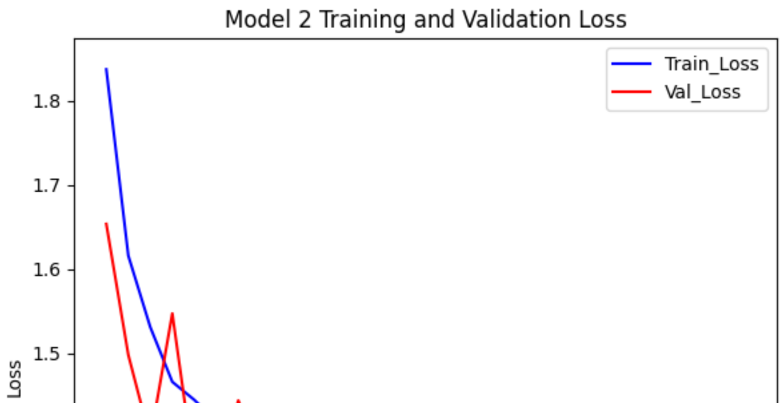
```
from tensorflow.keras.models import load_model

# Load the saved second model
second_model = load_model("model2.h5")

# Visualize the loaded second model
from tensorflow.keras.utils import plot_model

# Plot the second model architecture
plot_model(second_model, to_file='second_model.png', show_shapes=True)
```

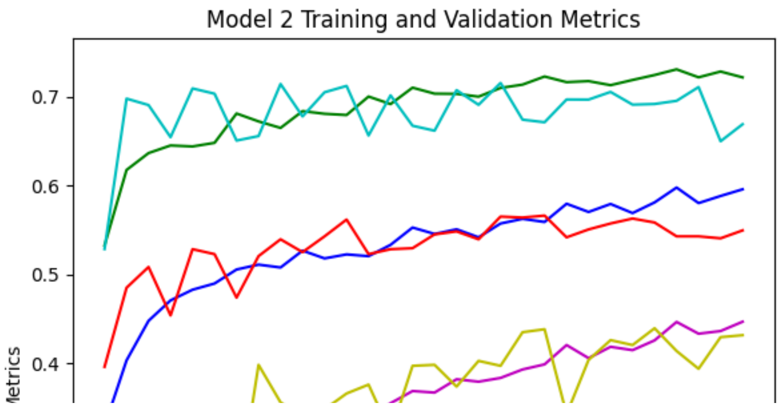



```
# Let us have a close look at precision of the above model
precision = model2.evaluate(train_generator2, verbose=1)[2]
precision_v = model2.evaluate(val_generator, verbose=1)[2]
print("Precision: ", precision)
print("Validation Precision: ", precision_v)
```

```
125/125 [==============================] - 26s 206ms/step - loss: 1.1367 - accuracy: 0.5753 - precision_1: 0.7067 - recall_1: 0.4530
29/29 [==============================] - 10s 333ms/step - loss: 1.2334 - accuracy: 0.5640 - precision_1: 0.6741 - recall_1: 0.4349
Precision:  0.7067082524299622
Validation Precision:  0.6741379499435425
```

```
# Accessing training history from the 'history' object
training_loss = history2.history['loss']
validation_loss = history2.history['val_loss']
training_accuracy = history2.history['accuracy']
validation_accuracy = history2.history['val_accuracy']
training_precision = history2.history['precision_1']
validation_precision = history2.history['val_precision_1']
training_recall = history2.history['recall_1']
validation_recall = history2.history['val_recall_1']

# Number of epochs
epochs = range(1, len(training_loss) + 1)

# creating a plot
plt.figure(figsize=(12, 6))
plt.subplot(121)
plt.plot(epochs, training_loss, 'b-', label='Train_Loss')
plt.plot(epochs, validation_loss, 'r-', label='Val_Loss')
plt.title('Model 2 Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Plotting Metrics
plt.subplot(122)
plt.plot(epochs, training_accuracy, 'b-', label='Train_Accuracy')
plt.plot(epochs, validation_accuracy, 'r-', label='Val_Accuracy')
plt.plot(epochs, training_precision, 'g-', label='Train_Precision')
plt.plot(epochs, validation_precision, 'c-', label='Val_Precision')
plt.plot(epochs, training_recall, 'm-', label='Train_Recall')
plt.plot(epochs, validation_recall, 'y-', label='Val_Recall')
plt.title('Model 2 Training and Validation Metrics')
plt.xlabel('Epochs')
plt.ylabel('Metrics')
plt.legend()


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





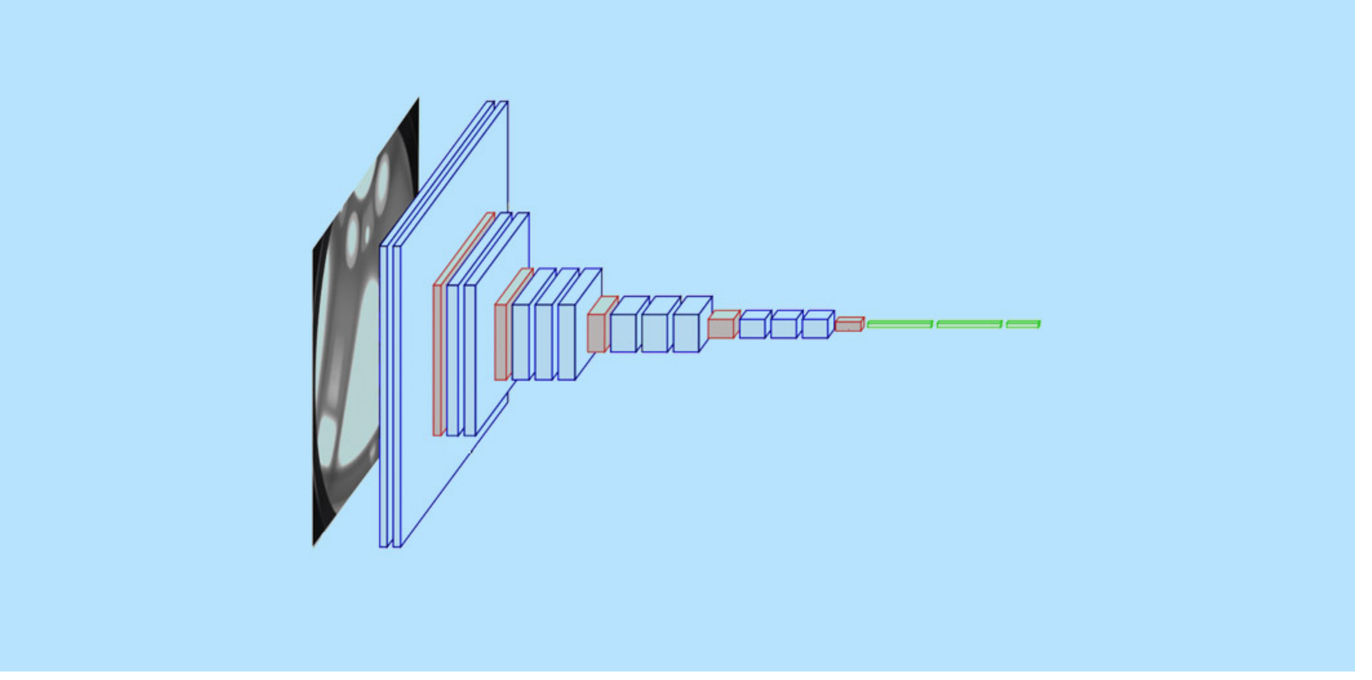
After dealing with class imbalance and augmentation using the augmenter it appears from the metrics that overfitting has reduced to some extent:

- While data augmentation has likely helped in reducing overfitting, it may not completely eliminate it. The training loss (1.1367) is lower than the validation loss (1.2334), which aligns with the idea that the model fits the training data better.
- For instance the training precision (0.7067) 70.67% is higher than the validation precision (0.6741) 67.41%. This suggests that there might still be some overfitting, as the model is performing better on the training data than on unseen validation data.

Due to the limited size of our dataset and the observed issue of overfitting in our models we have decided to use a pre-trained model known as VGG16 to enhance the overall performance of our models and address the challenges posed by the small dataset.

0 5 10 15 20 25 30 0 5 10 15 20 25 30

5.2.2: Model 3 - Using Pre-trained Model(VGG16)



VGG16 is a machine learning model that has been trained on a large dataset before being applied to a specific task or problem. It is known for its simplicity and effectiveness in image classification tasks.

```
# Set the hyperparameters
dense_units = 128
epochs = 100
filters = 64
kernel_size = (3, 3)

# Set random seed for reproducibility
np.random.seed(123)

# Create the model
model3 = Sequential()

# Add a base model (VGG16 in this example)
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
model3.add(base_model)
model3.add(GlobalAveragePooling2D())

# Add Dense layers
model3.add(Dense(256, activation='relu'))
model3.add(Dense(128, activation='relu'))
model3.add(Dense(64, activation='relu'))

# Output layer
model3.add(Dense(9, activation='softmax'))
```

```
# Compile the model with the specified learning rate
optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001)
model3.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy', Precision(), Recall()])

# Print a summary of the model
model3.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58889256/58889256 [==============================] - 0s 0us/step
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 7, 7, 512)         14714688

 global_average_pooling2d (  (None, 512)               0
 GlobalAveragePooling2D)

 dense_4 (Dense)             (None, 256)               131328

 dense_5 (Dense)             (None, 128)               32896

 dense_6 (Dense)             (None, 64)                8256

 dense_7 (Dense)             (None, 9)                 585

=================================================================
Total params: 14887753 (56.79 MB)
Trainable params: 14887753 (56.79 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

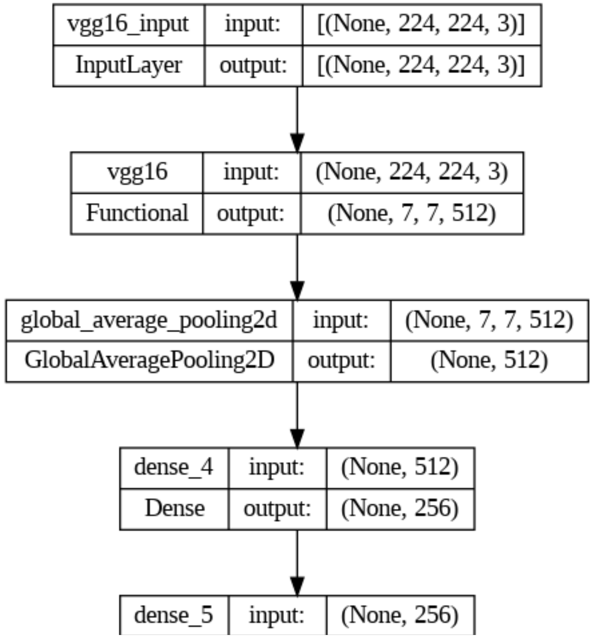
```
# Define EarlyStopping callback
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=10,
    restore_best_weights=True
)

# Train the model
history3 = model3.fit(
  train_generator2,
  validation_data=val_generator,
  epochs=100,
  callbacks=[early_stopping]
)
```

```
Epoch 1/100
125/125 [==============================] - 51s 329ms/step - loss: 1.7811 - accuracy: 0.3350 - precision_2: 0.6280 - recall_2: 0.0920 - val_loss: 1.3783 - val_accuracy: 0.4816 - val_precision_2: 0
Epoch 2/100
125/125 [==============================] - 40s 319ms/step - loss: 1.4023 - accuracy: 0.5123 - precision_2: 0.6565 - recall_2: 0.2795 - val_loss: 1.3690 - val_accuracy: 0.5217 - val_precision_2: 0
Epoch 3/100
125/125 [==============================] - 37s 298ms/step - loss: 1.2913 - accuracy: 0.5437 - precision_2: 0.6998 - recall_2: 0.3770 - val_loss: 1.2793 - val_accuracy: 0.5684 - val_precision_2: 0
Epoch 4/100
125/125 [==============================] - 39s 308ms/step - loss: 1.1822 - accuracy: 0.5875 - precision_2: 0.7264 - recall_2: 0.4380 - val_loss: 1.2243 - val_accuracy: 0.5862 - val_precision_2: 0
Epoch 5/100
125/125 [==============================] - 38s 308ms/step - loss: 1.1223 - accuracy: 0.6072 - precision_2: 0.7404 - recall_2: 0.4678 - val_loss: 1.1301 - val_accuracy: 0.5818 - val_precision_2: 0
Epoch 6/100
```

```
125/125 [==============================] - 36s 291ms/step - loss: 1.0207 - accuracy: 0.6457 - precision_2: 0.7494 - recall_2: 0.5300 - val_loss: 1.1713 - val_accuracy: 0.5951 - val_precision_2: 0
Epoch 7/100
125/125 [==============================] - 37s 300ms/step - loss: 0.9811 - accuracy: 0.6505 - precision_2: 0.7702 - recall_2: 0.5512 - val_loss: 1.0744 - val_accuracy: 0.6240 - val_precision_2: 0
Epoch 8/100
125/125 [==============================] - 38s 307ms/step - loss: 0.9144 - accuracy: 0.6780 - precision_2: 0.7778 - recall_2: 0.5740 - val_loss: 1.0390 - val_accuracy: 0.6251 - val_precision_2: 0
Epoch 9/100
125/125 [==============================] - 38s 303ms/step - loss: 0.8509 - accuracy: 0.7038 - precision_2: 0.7869 - recall_2: 0.6148 - val_loss: 1.0313 - val_accuracy: 0.6485 - val_precision_2: 0
Epoch 10/100
125/125 [==============================] - 36s 291ms/step - loss: 0.8073 - accuracy: 0.7165 - precision_2: 0.7982 - recall_2: 0.6348 - val_loss: 1.1900 - val_accuracy: 0.6307 - val_precision_2: 0
Epoch 11/100
125/125 [==============================] - 39s 313ms/step - loss: 0.7780 - accuracy: 0.7283 - precision_2: 0.7991 - recall_2: 0.6535 - val_loss: 1.0525 - val_accuracy: 0.6418 - val_precision_2: 0
Epoch 12/100
125/125 [==============================] - 50s 402ms/step - loss: 0.7312 - accuracy: 0.7435 - precision_2: 0.8167 - recall_2: 0.6705 - val_loss: 1.1109 - val_accuracy: 0.6440 - val_precision_2: 0
Epoch 13/100
125/125 [==============================] - 40s 321ms/step - loss: 0.7178 - accuracy: 0.7452 - precision_2: 0.8083 - recall_2: 0.6840 - val_loss: 1.3177 - val_accuracy: 0.5984 - val_precision_2: 0
Epoch 14/100
125/125 [==============================] - 37s 296ms/step - loss: 0.6394 - accuracy: 0.7665 - precision_2: 0.8278 - recall_2: 0.7128 - val_loss: 1.1892 - val_accuracy: 0.6385 - val_precision_2: 0
Epoch 15/100
125/125 [==============================] - 41s 326ms/step - loss: 0.5991 - accuracy: 0.7865 - precision_2: 0.8342 - recall_2: 0.7385 - val_loss: 1.2100 - val_accuracy: 0.6518 - val_precision_2: 0
Epoch 16/100
125/125 [==============================] - 36s 291ms/step - loss: 0.6186 - accuracy: 0.7835 - precision_2: 0.8383 - recall_2: 0.7347 - val_loss: 1.1566 - val_accuracy: 0.6418 - val_precision_2: 0
Epoch 17/100
125/125 [==============================] - 39s 310ms/step - loss: 0.5669 - accuracy: 0.7947 - precision_2: 0.8432 - recall_2: 0.7515 - val_loss: 1.3026 - val_accuracy: 0.6129 - val_precision_2: 0
Epoch 18/100
125/125 [==============================] - 39s 312ms/step - loss: 0.5406 - accuracy: 0.8052 - precision_2: 0.8511 - recall_2: 0.7657 - val_loss: 1.3316 - val_accuracy: 0.6174 - val_precision_2: 0
Epoch 19/100
125/125 [==============================] - 38s 302ms/step - loss: 0.4896 - accuracy: 0.8200 - precision_2: 0.8558 - recall_2: 0.7880 - val_loss: 1.3478 - val_accuracy: 0.6229 - val_precision_2: 0
```

```
# # Saving the entire model3 including architecture and weights
# model3.save("model3.h5")
```

```
from tensorflow.keras.models import load_model

# Load the saved third model
third_model = load_model("model3.h5")

# Visualize the loaded the third model
from tensorflow.keras.utils import plot_model

# Plot the third model architecture
plot_model(third_model, to_file='third_model.png', show_shapes=True)
```



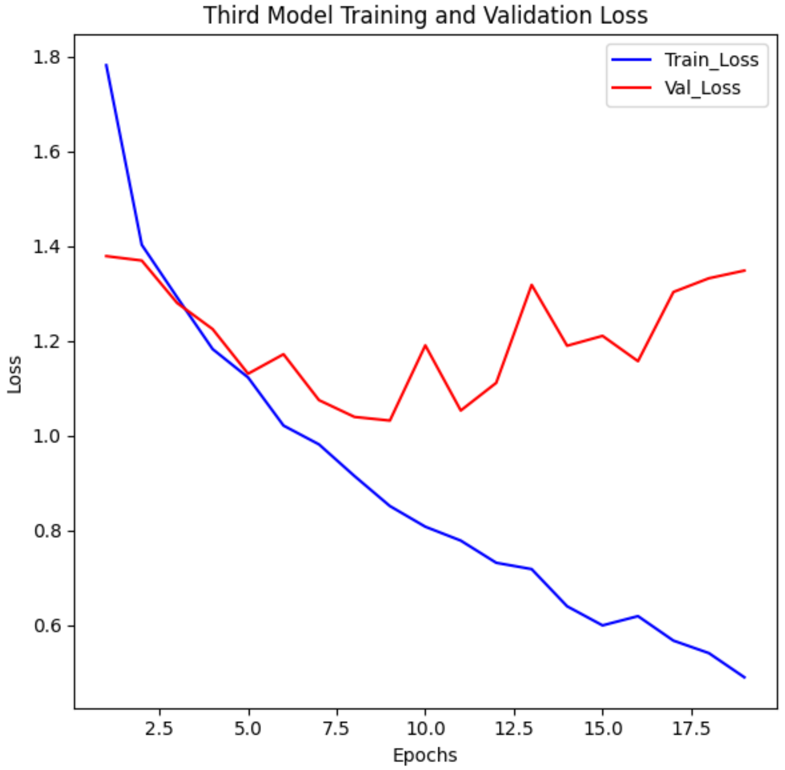

```
# Let us have a close look at precision of the above model
precision = model3.evaluate(train_generator2, verbose=1)[2]
precision_v = model3.evaluate(val_generator, verbose=1)[2]
print("Precision: ", precision)
print("Validation Precision: ", precision_v)
```

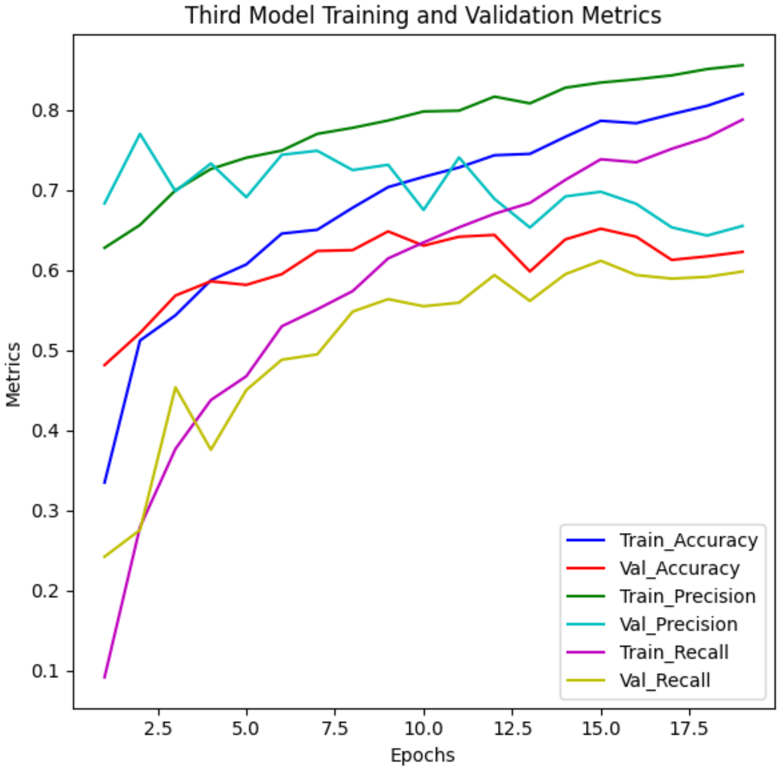
```
125/125 [==============================] - 28s 222ms/step - loss: 0.7620 - accuracy: 0.7368 - precision_2: 0.8148 - recall_2: 0.6545
29/29 [==============================] - 10s 327ms/step - loss: 1.0313 - accuracy: 0.6485 - precision_2: 0.7316 - recall_2: 0.5640
Precision:  0.8148148059844971
Validation Precision:  0.7316017150878906
```

| dense_7 | input: | (None, 64) |

```
# Accessing training history from the 'history' object
training_loss = history3.history['loss']
validation_loss = history3.history['val_loss']
training_accuracy = history3.history['accuracy']
validation_accuracy = history3.history['val_accuracy']
training_precision = history3.history['precision_2']
validation_precision = history3.history['val_precision_2']
training_recall = history3.history['recall_2']
validation_recall = history3.history['val_recall_2']

# Number of epochs
epochs = range(1, len(training_loss) + 1)

# creating a plot
plt.figure(figsize=(12, 6))
plt.subplot(121)
plt.plot(epochs, training_loss, 'b-', label='Train_Loss')
plt.plot(epochs, validation_loss, 'r-', label='Val_Loss')
plt.title('Third Model Training and Validation Loss')
```

```
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Plotting Metrics
plt.subplot(122)
plt.plot(epochs, training_accuracy, 'b-', label='Train_Accuracy')
plt.plot(epochs, validation_accuracy, 'r-', label='Val_Accuracy')
plt.plot(epochs, training_precision, 'g-', label='Train_Precision')
plt.plot(epochs, validation_precision, 'c-', label='Val_Precision')
plt.plot(epochs, training_recall, 'm-', label='Train_Recall')
plt.plot(epochs, validation_recall, 'y-', label='Val_Recall')
plt.title('Third Model Training and Validation Metrics')
plt.xlabel('Epochs')
plt.ylabel('Metrics')
plt.legend()


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





After introducing a pre-trained model it appears that our metrics and loss performance are improving:

- While introducing VVG16 has likely helped in reducing overfitting and increasing the metrics, it has not completely eliminated it. The training loss (0.7620) is lower than the validation loss (1.0313), which is an improvement from our second model
- The training precision (0.8148) 81.48% is higher than the validation precision (0.7316) 73.16%. This suggests that there might still be some overfitting, as the model is performing better on the training data than on unseen validation data. The metrics are however higher than the second model.

We shall implement dropout to prevent overfitting and to enhance the overall performance of our model.

5.2.3: Model 4 - Using Tuned(Dropout) Pre-trained Model(VGG16)

```
# Set the hyperparameters
dense_units = 128
epochs = 100
filters = 64
kernel_size = (3, 3)
learning_rate = 0.0001

# Set random seed for reproducibility
np.random.seed(123)

# Create the model
model4 = Sequential()

# Add a base model (VGG16 in this example)
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
model4.add(base_model)
model4.add(GlobalAveragePooling2D())

# Add Dropout layers
model4.add(Dropout(0.5))

# Add Dense layers
model4.add(Dense(256, activation='relu'))
model4.add(Dense(128, activation='relu'))
model4.add(Dense(64, activation='relu'))

# Output layer
model4.add(Dense(9, activation='softmax'))

# Compile the model with the specified learning rate
optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
model4.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy', Precision(), Recall()])

# Print a summary of the model
model4.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 7, 7, 512)         14714688

 global_average_pooling2d_1  (None, 512)               0
  (GlobalAveragePooling2D)
```

```
 dropout (Dropout)           (None, 512)               0

 dense_8 (Dense)             (None, 256)               131328

 dense_9 (Dense)             (None, 128)               32896

 dense_10 (Dense)            (None, 64)                8256

 dense_11 (Dense)            (None, 9)                 585

=================================================================
Total params: 14887753 (56.79 MB)
Trainable params: 14887753 (56.79 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

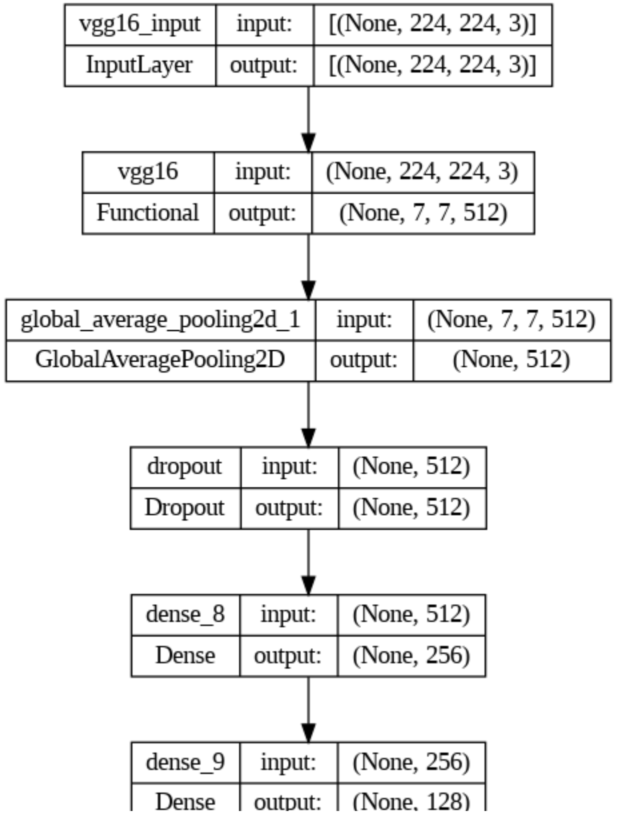
```
# Define EarlyStopping callback
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=10,
    restore_best_weights=True
)

# Train the model
history4 = model4.fit(
  train_generator2,
  validation_data=val_generator,
  epochs=100,
  callbacks=[early_stopping]
)
```

```
Epoch 1/100
125/125 [==============================] - 43s 300ms/step - loss: 2.0672 - accuracy: 0.2048 - precision_3: 0.6098 - recall_3: 0.0063 - val_loss: 1.8992 - val_accuracy: 0.3426 - val_precision_3
Epoch 2/100
125/125 [==============================] - 38s 302ms/step - loss: 1.8412 - accuracy: 0.3083 - precision_3: 0.5255 - recall_3: 0.0593 - val_loss: 1.7989 - val_accuracy: 0.3181 - val_precision_3
Epoch 3/100
125/125 [==============================] - 36s 291ms/step - loss: 1.7219 - accuracy: 0.3530 - precision_3: 0.6080 - recall_3: 0.0985 - val_loss: 1.6836 - val_accuracy: 0.3604 - val_precision_3
Epoch 4/100
125/125 [==============================] - 36s 290ms/step - loss: 1.6586 - accuracy: 0.3820 - precision_3: 0.6617 - recall_3: 0.1437 - val_loss: 1.5232 - val_accuracy: 0.4249 - val_precision_3
Epoch 5/100
125/125 [==============================] - 37s 299ms/step - loss: 1.5200 - accuracy: 0.4475 - precision_3: 0.6771 - recall_3: 0.2155 - val_loss: 1.3765 - val_accuracy: 0.5250 - val_precision_3
Epoch 6/100
125/125 [==============================] - 37s 294ms/step - loss: 1.4017 - accuracy: 0.5017 - precision_3: 0.6551 - recall_3: 0.2825 - val_loss: 1.2368 - val_accuracy: 0.5551 - val_precision_3
Epoch 7/100
125/125 [==============================] - 37s 298ms/step - loss: 1.3006 - accuracy: 0.5395 - precision_3: 0.7098 - recall_3: 0.3663 - val_loss: 1.2178 - val_accuracy: 0.5651 - val_precision_3
Epoch 8/100
125/125 [==============================] - 37s 299ms/step - loss: 1.2316 - accuracy: 0.5673 - precision_3: 0.7252 - recall_3: 0.4215 - val_loss: 1.2961 - val_accuracy: 0.5373 - val_precision_3
Epoch 9/100
125/125 [==============================] - 38s 302ms/step - loss: 1.1816 - accuracy: 0.5800 - precision_3: 0.7369 - recall_3: 0.4355 - val_loss: 1.1945 - val_accuracy: 0.5729 - val_precision_3
Epoch 10/100
125/125 [==============================] - 38s 305ms/step - loss: 1.1049 - accuracy: 0.6083 - precision_3: 0.7524 - recall_3: 0.4770 - val_loss: 1.1927 - val_accuracy: 0.5929 - val_precision_3
Epoch 11/100
125/125 [==============================] - 37s 293ms/step - loss: 1.0577 - accuracy: 0.6267 - precision_3: 0.7547 - recall_3: 0.4978 - val_loss: 1.1575 - val_accuracy: 0.6107 - val_precision_3
Epoch 12/100
125/125 [==============================] - 38s 304ms/step - loss: 0.9897 - accuracy: 0.6562 - precision_3: 0.7771 - recall_3: 0.5318 - val_loss: 1.2723 - val_accuracy: 0.6073 - val_precision_3
Epoch 13/100
125/125 [==============================] - 38s 301ms/step - loss: 0.9591 - accuracy: 0.6745 - precision_3: 0.7727 - recall_3: 0.5583 - val_loss: 1.1617 - val_accuracy: 0.6251 - val_precision_3
Epoch 14/100
```

```
125/125 [==============================] - 38s 305ms/step - loss: 0.8896 - accuracy: 0.6850 - precision_3: 0.7921 - recall_3: 0.5878 - val_loss: 1.1395 - val_accuracy: 0.6274 - val_precision_3
Epoch 15/100
125/125 [==============================] - 53s 428ms/step - loss: 0.8892 - accuracy: 0.6980 - precision_3: 0.7941 - recall_3: 0.5947 - val_loss: 1.3492 - val_accuracy: 0.5873 - val_precision_3
Epoch 16/100
125/125 [==============================] - 66s 528ms/step - loss: 0.8492 - accuracy: 0.7180 - precision_3: 0.8000 - recall_3: 0.6280 - val_loss: 1.1658 - val_accuracy: 0.6274 - val_precision_3
Epoch 17/100
125/125 [==============================] - 50s 401ms/step - loss: 0.7749 - accuracy: 0.7297 - precision_3: 0.8164 - recall_3: 0.6505 - val_loss: 1.2177 - val_accuracy: 0.6274 - val_precision_3
Epoch 18/100
125/125 [==============================] - 50s 398ms/step - loss: 0.7501 - accuracy: 0.7430 - precision_3: 0.8138 - recall_3: 0.6590 - val_loss: 1.1275 - val_accuracy: 0.6374 - val_precision_3
Epoch 19/100
125/125 [==============================] - 52s 414ms/step - loss: 0.7188 - accuracy: 0.7567 - precision_3: 0.8284 - recall_3: 0.6842 - val_loss: 1.1790 - val_accuracy: 0.6251 - val_precision_3
Epoch 20/100
125/125 [==============================] - 53s 428ms/step - loss: 0.7041 - accuracy: 0.7560 - precision_3: 0.8266 - recall_3: 0.6877 - val_loss: 1.3207 - val_accuracy: 0.6385 - val_precision_3
Epoch 21/100
125/125 [==============================] - 40s 319ms/step - loss: 0.6428 - accuracy: 0.7707 - precision_3: 0.8421 - recall_3: 0.7080 - val_loss: 1.3331 - val_accuracy: 0.6307 - val_precision_3
Epoch 22/100
125/125 [==============================] - 49s 391ms/step - loss: 0.6168 - accuracy: 0.7895 - precision_3: 0.8476 - recall_3: 0.7343 - val_loss: 1.5173 - val_accuracy: 0.6162 - val_precision_3
Epoch 23/100
125/125 [==============================] - 40s 322ms/step - loss: 0.5763 - accuracy: 0.7980 - precision_3: 0.8586 - recall_3: 0.7405 - val_loss: 1.2757 - val_accuracy: 0.6296 - val_precision_3
Epoch 24/100
125/125 [==============================] - 38s 307ms/step - loss: 0.5767 - accuracy: 0.7972 - precision_3: 0.8530 - recall_3: 0.7473 - val_loss: 1.4916 - val_accuracy: 0.6029 - val_precision_3
Epoch 25/100
125/125 [==============================] - 37s 293ms/step - loss: 0.5520 - accuracy: 0.8077 - precision_3: 0.8654 - recall_3: 0.7567 - val_loss: 1.4141 - val_accuracy: 0.6218 - val_precision_3
Epoch 26/100
125/125 [==============================] - 39s 314ms/step - loss: 0.5112 - accuracy: 0.8267 - precision_3: 0.8728 - recall_3: 0.7790 - val_loss: 1.3713 - val_accuracy: 0.6207 - val_precision_3
Epoch 27/100
125/125 [==============================] - 37s 294ms/step - loss: 0.4848 - accuracy: 0.8298 - precision_3: 0.8719 - recall_3: 0.7897 - val_loss: 1.4654 - val_accuracy: 0.6140 - val_precision_3
Epoch 28/100
125/125 [==============================] - 37s 296ms/step - loss: 0.4605 - accuracy: 0.8375 - precision_3: 0.8724 - recall_3: 0.8050 - val_loss: 1.3300 - val_accuracy: 0.5907 - val_precision_3
```

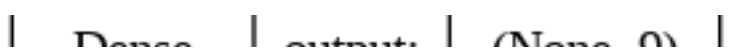
```
# Saving the entire model4 including architecture and weights
model4.save("model4.h5")
```

```
from tensorflow.keras.models import load_model

# Load the saved fourth model
fourth_model = load_model("model4.h5")

# Visualize the loaded the fourth model
from tensorflow.keras.utils import plot_model

# Plot the fourth model architecture
plot_model(fourth_model, to_file='fourth_model.png', show_shapes=True)
```




```
# Let us have a close look at precision of the above model
precision = model4.evaluate(train_generator2, verbose=1)[2]
precision_v = model4.evaluate(val_generator, verbose=1)[2]
print("Precision: ", precision)
print("Validation Precision: ", precision_v)
```

```
125/125 [==============================] - 26s 210ms/step - loss: 0.6525 - accuracy: 0.7715 - precision_3: 0.8343 - recall_3: 0.7135
29/29 [==============================] - 10s 325ms/step - loss: 1.1275 - accuracy: 0.6374 - precision_3: 0.7092 - recall_3: 0.5940
Precision:  0.8342589735984802
Validation Precision:  0.7091633677482605
```



```
# Accessing training history from the 'history' object
training_loss = history4.history['loss']
validation_loss = history4.history['val_loss']
training_accuracy = history4.history['accuracy']
validation_accuracy = history4.history['val_accuracy']
training_precision = history4.history['precision_3']
validation_precision = history4.history['val_precision_3']
training_recall = history4.history['recall_3']
validation_recall = history4.history['val_recall_3']

# Number of epochs
epochs = range(1, len(training_loss) + 1)
```

```
# creating a plot
plt.figure(figsize=(12, 6))
plt.subplot(121)
plt.plot(epochs, training_loss, 'b-', label='Train_Loss')
plt.plot(epochs, validation_loss, 'r-', label='Val_Loss')
plt.title('Fourth Model Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Plotting Metrics
plt.subplot(122)
plt.plot(epochs, training_accuracy, 'b-', label='Train_Accuracy')
plt.plot(epochs, validation_accuracy, 'r-', label='Val_Accuracy')
plt.plot(epochs, training_precision, 'g-', label='Train_Precision')
plt.plot(epochs, validation_precision, 'c-', label='Val_Precision')
plt.plot(epochs, training_recall, 'm-', label='Train_Recall')
plt.plot(epochs, validation_recall, 'y-', label='Val_Recall')
plt.title('Fourth Model Training and Validation Metrics')
plt.xlabel('Epochs')
plt.ylabel('Metrics')
plt.legend()


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

After introducing dropout regularization technique it appears that our metrics and loss performance are becoming worse than the previous model with increased overfitting:

- There is about 0.475 difference between training loss (0.6525) and the validation loss (1.1275), which is a decrease in performance from our third model which had a difference of about 0.269
- The training precision (0.8343) 83.43% is higher than the validation precision (0.7092) 70.92%. There is still some overfitting, as the model is performing better on the training data than on unseen validation data.

Best Model

We clearly see that the third model performs better than the other models. We shall proceeed with the third model and predict the images per class.

```
# Define the number of rows and columns for the grid
num_rows = 5
num_cols = 5

# Create an iterator from the test generator to access the test images and their labels
test_iterator = iter(test_generator)

# Create a figure with subplots for displaying the images in a grid
fig, axes = plt.subplots(5, 5, figsize=(15, 15))

# Define class names
class_names = [
    "actinic keratosis",
    "basal cell carcinoma",
    "dermatofibroma",
    "melanoma",
    "nevus",
    "pigmented benign keratosis",
    "seborrheic keratosis",
    "squamous cell carcinoma",
    "vascular lesion"
]

# Loop through the grid and display images with actual and predicted class labels
for row in range(num_rows):
    for col in range(num_cols):
        # Get a batch of images and labels from the test generator
        images, labels = next(test_iterator)

        # Select the first image from the batch
        sample_image = images[0]

        # Resize the image to the target shape (assuming your model expects 224x224 images)
        sample_image = tf.image.resize(sample_image, (224, 224))

        # Make a prediction using your loaded model
```

```
        predictions = third_model.predict(sample_image.numpy().reshape(1, 224, 224, 3))

        # Get the predicted class name
        predicted_class = class_names[np.argmax(predictions, axis=1)[0]]

        # Get the actual class name
        actual_class = class_names[np.argmax(labels[0])]

        # Display the image with the actual and predicted class names
        axes[row, col].imshow(sample_image.numpy())
        axes[row, col].set_title(f"Actual: {actual_class}\nPredicted: {predicted_class}")
        axes[row, col].axis('off')

plt.show()
```