**Objective**

To Use Convolution Neural Network Machine Learning Model and identify 30 different stop signs (example - speed limit signs, stop sign, direction signs) that has been provided in the form of images.

**Libraries used**

import pathlib

from pathlib import Path

import tensorflow as tf

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.image as mpimg

import warnings

warnings.filterwarnings('ignore')

import os

import PIL

import glob

from tensorflow import keras

from tensorflow.keras import layers

from tensorflow.keras.regularizers import l2

from tensorflow.keras.models import Sequential

from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPool2D, BatchNormalization

**Data Preparation**

root\_path = 'C:/Users/username/Road Signs classification dataset/'

The dataset has been uploaded in the provided path directory so that the model can find it with ease, In regards to the excel which shows the stop sign’s meanings, the file names were changed.

For example – The folder 0 was renamed as Speed Limit(5Km/h), folder 1 as Speed Limit(15Km/h) as per the excel sheet.

The dataset contains 743 images in the Training dataset and 280 images in the Testing dataset, the formats in which the images were available are PNG and JPG.

**Parameter Settings for Training Data**

batch\_size = 32

img\_height = 180

img\_width = 180

num\_classes = 30

30 denotes the number of classes into which the images are classified, Batch Size represents the number of images which are loaded and processed at the same time. Image width and Image height were given to set the dimension of the image that is to be imported to the model.

**Preprocessing**

train\_ds = tf.keras.preprocessing.image\_dataset\_from\_directory(

data\_dir\_train,

seed=123,

validation\_split= 0.2,

subset= 'training',

image\_size=(img\_height,img\_width),

batch\_size = batch\_size)

Validation splits as 0.2 keeps the validation as 20% of the training data as validation set and 80% remining as training data. The provided dataset is set to preprocess as per the parameters provided above in Parameter Settings. Seed as 123 represents we get the same split every time we run the code with this seed. Image height and width define the input image dimensions width in pixels that all images will be resized to before being imported into the model along with the batch size set for loading and preprocessing.

The same is done for Validation set

val\_ds = tf.keras.preprocessing.image\_dataset\_from\_directory(

data\_dir\_train,

seed=123,

validation\_split= 0.2,

subset= 'validation',

image\_size=(img\_height,img\_width),

batch\_size = batch\_size)

12595 is 80% of the dataset and 3148 is 20% of the dataset totalling the images taken from the Train folder in the drive is 15743.

class\_names = train\_ds.class\_names

print(class\_names)

The above code helps us to identify the number of classes available in the training dataset.

class\_names = val\_ds.class\_names

print(class\_names)

The same code helps us to get the class name from the validation set which should be equal to the number of classes available in the training set.

plt.figure(figsize=(10, 10))

num\_classes = 30

for i in range(num\_classes):

class\_images = list(data\_dir\_train.glob(class\_names[i] + '/\*.[jp]\*g')) + list(data\_dir\_train.glob(class\_names[i] + '/\*.png'))

if len(class\_images) > 0: # Ensure there are images in the directory

image = mpimg.imread(str(class\_images[0])) # Use the first image if available

plt.subplot(6, 5, i + 1) # 6 rows and 5 columns grid (for 30 images)

plt.title(class\_names[i]) # Title of the subplot is the class name

plt.imshow(image)

plt.axis('off') # Hide axis for cleaner visuals

else:

print(f"No images found for class: {class\_names[i]}")

plt.show()

After the preprocessing in order to check the images available in the dataset, to check the image from each class we are using the plot code to get images from both format jpg and png in the order of 6 rows and 5 columns 6\*5 = 30 images in the plot along with the name of the respective class’s name.

AUTOTUNE = tf.data.experimental.AUTOTUNE

train\_ds = train\_ds.cache().shuffle(1000).prefetch(buffer\_size=AUTOTUNE)

val\_ds = val\_ds.cache().prefetch(buffer\_size=AUTOTUNE)

To increase the speed of the process of optimization by keeping the next batch ready for training. The above code is used to help the module decide how many images to prepare while the model is being trained.

The shuffle is set to 1000 for the training data alone so that it can be randomized whereas the shuffle is not set for validation set since it should not be randomized to provide consistency. Cache is where the data is stored while being trained and prefetch helps us in getting the data for next batch.

**CNN**

# Using layers.experimental.preprocessing.Rescaling, normalize pixel values between (0,1)

model = Sequential([layers.Rescaling(1./255, input\_shape=(img\_height, img\_width, 3))])

This creates a linear stack of layers in order in regards to the module Sequential, Rescaling helps us in normalization so that the pixel values are in the range of 0 to 1 by dividing each pixel by 255 which is the highest range of the pixel. The input shape specifies the shape of the input images. Here, each image will have a shape of 180x180 pixels based on the parameter settings with 3 colour channels (RGB).

# 1st convolution layer

model.add(Conv2D(filters = 16, kernel\_size = (3,3),padding = 'Same',

activation ='relu'))

model.add(MaxPool2D(pool\_size=(2,2)))

This is a 2D convolutional layer that applies filters (kernels) to the input image to extract features. Filters = 16, The number of filters used, kernel\_size = (3, 3): The size of the filters (3x3) used to scan the image, padding = 'Same': Ensures the output size remains the same as the input by padding the image with zeros around the borders.

Activation function used is relu(Rectified Linear Unit) applied to the output of the convolution used for faster learning in hidden layers.

MaxPool2D(pool\_size=(2, 2)): A max pooling layer that reduces the spatial dimensions of the image while retaining important information. The pool size of 2x2 means the input image is downsampled by a factor of 2. This helps in reducing computation and making the model more robust to small translations.

# 2nd convolution layer

model.add(Conv2D(filters = 32, kernel\_size = (3,3),padding = 'Same',

activation ='relu'))

model.add(MaxPool2D(pool\_size=(2,2)))

# 3rd convolution layer

model.add(Conv2D(filters = 64, kernel\_size = (3,3),padding = 'Same',

activation ='relu'))

model.add(MaxPool2D(pool\_size=(2,2)))

# 4th convolution layer

model.add(Conv2D(filters = 128, kernel\_size = (3,3),padding = 'Same',

activation ='relu'))

model.add(MaxPool2D(pool\_size=(2,2)))

The second, third and fourth convolutional layers are similar to the first but with increased filters (32,64 and 128 respectively). As we move deeper into the network, we use more filters to capture more complex and in-depth features of the image. MaxPooling is applied after each convolutional layer to reduce the spatial dimensions by only keeping the important feature of the images.

model.add(Flatten())

This layer flattens the output from the 2D convolutional and pooling layers into a 1D vector. This is necessary because fully connected layers expect 1D input, and we want to connect this flattened vector to the further layers.

# Fully connected layer 1 - model.add(Dense(128, activation='relu'))

# Fully connected layer 2 - model.add(Dense(128, activation='relu'))

This is a fully connected layer. 128 showcases the number of neurons in the layer. These layers are used to learn more abstract representations based on the features extracted by the convolutional layers. Activation function used again to allow the model to learn more complex patterns.

# Output layer - model.add(Dense(num\_classes, activation = "softmax"))

Dense in this layer corresponds to the output of the model, with num\_classes neurons (one per class in the classification task). Since we have 30 classes, the output layer will have 30 neurons. Activation function used is softmax as this is a multi-class classification. It converts the outputs into probabilities, where each output value represents the probability of the input image belonging to a specific class totaling to 1.

**Optimizer and Loss Function**

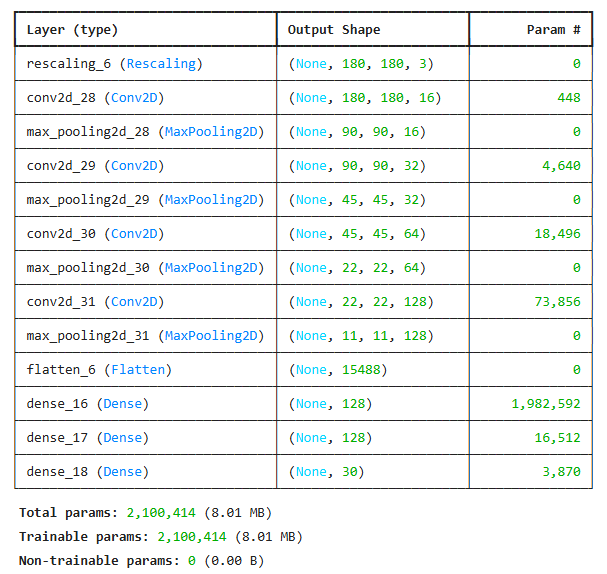
model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True),

metrics=['accuracy'])

Optimizer is a function used to provide correct weight and bias in order to avoid errors and increase accuracy, the optimizer used here is Adam. Loss Function used is SparseCategoricalCrossentropy in order to measure how well the network is performing It calculates the difference between predicted and actual labels, and the model tries to minimize this loss during training. This specific loss function is appropriate when we have more than two classes (in this case, 30)

**Summary of Training Data throughout the model layers**

****

**Model Trainning**

epochs = 20

history = model.fit(

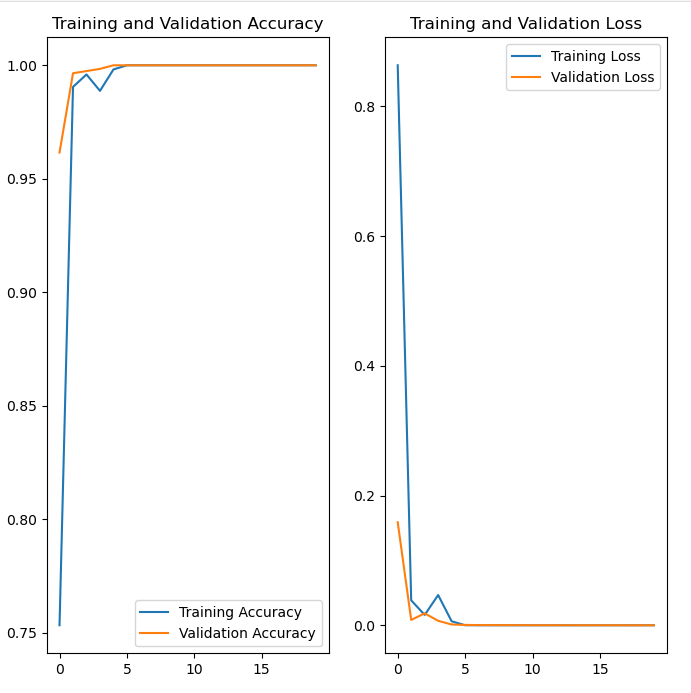
train\_ds,

validation\_data=val\_ds,

epochs=epochs)

1. Epochs refer to how many times the entire dataset is passed through the model during training and in our code, we have set the epochs as 20. model.fit() is the function used to train the model. It takes several arguments and starts the process of fitting (training) the model on our data. This is our training dataset. It contains the training data (images and labels) and is used to teach the model how to make predictions.
2. Validation\_data contains data that the model does not see during training but is used to evaluate the model's performance on data that it has not been trained on. By using validation data, we see how well the model generalizes and prevent overfitting. Every epoch, the model will make predictions on the validation set and compute the loss and accuracy based on that data.
3. This history includes details of training and validation loss and validation accuracy for each epoch which we to visualize the model's learning over time (for example, plotting loss/accuracy charts).

**Visualization of Training and Validation – Accuracy/Loss**



**Insights**

* The training accuracy score started from 75% and reached 100% after epoch 6 and remained constant throughout the remaining epochs execution.
* In terms of validation accuracy, it started well above 95% and reached 100% in the next few epochs execution and remained constant.
* The Training Loss started from 80% and declined to around 1 in the remaining epochs execution.
* Validation loss started from 20% and remained constant less than 2% throughout the remaining process.

**Data Augmentation**

# Since the model overfits, data augumentation strategy is used

# Data augmentation is artificially increasing the amount of data needed to train robust AI models

data\_aug = keras.Sequential([

layers.RandomFlip(mode="horizontal\_and\_vertical",

input\_shape=(img\_height,img\_width,3)),

layers.RandomRotation(0.2, fill\_mode='reflect'),

layers.RandomZoom(height\_factor=(0.2, 0.3),

width\_factor=(0.2, 0.3), fill\_mode='reflect')])

Data Augmentation is used to artificially increase the size of our dataset by applying random transformations to the input images. The goal is to make the model more robust and generalizable, helping it perform better on unseen data, especially when there is a risk of overfitting.

* Sequential() is once again used to stack multiple layers, allowing us to define a sequence of transformations that will be applied to the images.
* Flip randomly flips images both horizontally and vertically.
* Rotation applies a random rotation to the image within a specific range. Reflect is used when parts of the image are rotated out of the frame, this option specifies how to fill in the empty areas. Reflect mode means the image is padded with a reflection of the original image’s border. This avoids creating blank spaces.
* Zooming randomly zooms into the image by a factor between 20% and 30% for both the height and width and just like reflect fills in the blank spaces using the reflection of original image’s edges.

**Visualize the augmentation strategy for one instance**

plt.figure(figsize=(12, 12))

# Get one batch of images and labels from the training dataset

for images, labels in train\_ds.take(1):

for i in range(min(30, len(images))):

ax = plt.subplot(6, 5, i + 1)

plt.imshow(data\_aug(images)[i].numpy().astype("uint8"))

plt.title(class\_names[labels[i]])

plt.axis("off")

**Augmented Image**



**Training Model with Augmented Data**

# Let us create a new model with Dropout layer and l2 regularizer as the previous model overfits

# Use the augmented data for creating the model

# Let us add one more layer to improve accuracy

model = Sequential([data\_aug,

layers.Rescaling(1./255, input\_shape=(img\_height, img\_width,3))])

# 1st convolution layer with Dropout layer

model.add(Conv2D(filters = 16, kernel\_size = (3,3),padding = 'Same',

activation ='relu'))

model.add(MaxPool2D(pool\_size=(2,2)))

model.add(Dropout(0.10))

# 2nd convolution layer with Dropout layer

model.add(Conv2D(filters = 32, kernel\_size = (3,3),padding = 'Same',

activation ='relu'))

model.add(MaxPool2D(pool\_size=(2,2)))

model.add(Dropout(0.20))

# 3rd convolution layer with Dropout layer

model.add(Conv2D(filters = 32, kernel\_size = (3,3),padding = 'Same',

activation ='relu'))

model.add(MaxPool2D(pool\_size=(2,2)))

model.add(Dropout(0.10))

# 4th convolution layer with Dropout layer

model.add(Conv2D(filters = 64, kernel\_size = (3,3),padding = 'Same',

activation ='relu'))

model.add(MaxPool2D(pool\_size=(2,2)))

model.add(Dropout(0.20))

# 5th convolution layer with Dropout layer

model.add(Conv2D(filters = 128, kernel\_size = (3,3),padding = 'Same',

activation ='relu'))

model.add(MaxPool2D(pool\_size=(2,2)))

model.add(Dropout(0.10))

model.add(Flatten())

# Fully connected layer 1 with l2 regularizer

model.add(Dense(128, activation='relu', kernel\_regularizer=l2(0.02)))

# Fully connected layer 2 with Dropout layer

model.add(Dense(128, activation='relu'))

model.add(Dropout(0.10))

# Output layer

model.add(Dense(num\_classes, activation = "softmax"))

Dropout is a regularization technique used in neural networks to prevent overfitting and improve generalization by randomly setting a fraction of the input units (neurons) to zero during training. For example – If dropout is set as .10, 10% of the neurons are randomly dropped or set to zero during.

Everything Except the dataset being Augmented and the dropout technique, it’s same as the Training model used with the original training dataset with multiple convolutional layers and 2 connected layers.

Once again the same Optimizer Adam and Loss function SparseCategoricalCrossentropy is used for the augmented data.

epochs=40

history = model.fit(

train\_ds,

validation\_data=val\_ds,

epochs=epochs)

The same model with augmented data is once again ran with the same 40 epochs in regards to the training and validation dataset taken initially. The visualization of the comparison between Training Accuracy/Validation Accuracy and Training Loss/Validation Loss is provided for reference.

**Visualization of Augmented model’s Training and Validation – Accuracy/Loss**



**Insights**

* The Training Accuracy starts from 0.2 and climbs up to more than 95% at the end of 40 Epochs however the computation took around 2 mins for each epoch.
* Validation accuracy started from 0.35 and the maximum achieved at the end of 40 epochs is few pointers less than 90%.
* Training loss started from 3 and ended less than .5 at the end of 40 epochs
* Validation loss started from 2.5 and climbed down to less than 1 after running 40 epochs.

**Augmentor – Library**

path\_to\_training\_dataset = root\_path+'Train/'

import Augmentor

for i in class\_names:

p = Augmentor.Pipeline(path\_to\_training\_dataset + i)

p.rotate(probability=0.7, max\_left\_rotation=10, max\_right\_rotation=10)

p.sample(50)

* Augmentor used here is a third-party Python library dedicated to image augmentation. It works by defining a pipeline of augmentation operations and applying them to the images in a directory.

We create a pipeline for each class (directory of images) inside our path\_to\_training\_dataset.

* Rotation adds random rotations to the images with a probability of 70%,
* We then generate 50 augmented images for each class. This will physically generate new augmented images and store them in a new directory.
* All the augmentations are done beforehand, and new images are saved to disk, which can be used later in training.

# Total number of added new samples

data\_dir\_train = Path(data\_dir\_train)

# Total number of added new samples

image\_count\_train = len(list(data\_dir\_train.glob('\*/output/\*.jpg'))) + len(list(data\_dir\_train.glob('\*/output/\*.png')))

print(image\_count\_train)

This code calculates the total number of newly added image samples (in .jpg and .png formats) that are stored under an output folder within each class subdirectory inside our training dataset.

# New path

path\_list\_new = [x for x in glob.glob(os.path.join(data\_dir\_train, '\*', 'output', '\*.jpg'))] + [x for x in glob.glob(os.path.join(data\_dir\_train, '\*', 'output', '\*.png'))]

# Print the result to see the paths

print(path\_list\_new)

This code collects the file paths of all newly augmented image samples (in .jpg and .png format) located in output folders within each class directory of our training dataset.

This uses the glob module to search the filesystem for files matching specific pattern and matches all .jpg/png images in the output folders across all class directories which are then concatenated using +, to get both .jpg and .png files. The final result is stored in path\_list\_new, which is a list of strings, each being the path to an image file.

# Respective class names

# Include both .jpg and .png files and extract the directory names

lesion\_list\_new = [

os.path.basename(os.path.dirname(os.path.dirname(y)))

for y in glob.glob(os.path.join(data\_dir\_train, '\*', 'output', '\*.jpg')) + glob.glob(os.path.join(data\_dir\_train, '\*', 'output', '\*.png'))]

print(lesion\_list\_new)

This code snippet is used to extract the class names stop sign types for each augmented image based on its directory structure.

dataframe\_dict\_new = dict(zip(path\_list\_new, lesion\_list\_new))

This code snippet is used to create a pandas DataFrame that maps image file paths to their respective class labels, and then it checks how many images exist in each class after augmentation. Each element of path\_list\_new is paired with the corresponding class name in lesion\_list\_new.

# Create a dataframe for storing paths and labels

df = pd.DataFrame(list(dataframe\_dict\_new.items()),columns = ['Path','Label'])

Retrieves the key-value pairs (image paths and their corresponding labels) from the dictionary and converts them into a List.

# Check for the number of images in each class

So, df['Label'].value\_counts() will give us a count of how many images belong to each class (after augmentation) The result of the value\_count is provided below for reference.

Label

Don't G o straight or left 1550

Go Left or Right 1550

Unknown 1550

Speed Limit(80Km per h) 1550

Speed Limit(70Km per h) 1550

Speed Limit(60Km per h) 1550

Speed Limit(5Km per h) 1550

Speed Limit(50Km per h) 1550

Speed Limit(40Km per h) 1550

Speed Limit(30Km per h) 1550

Speed Limit(15Km per h) 1550

Roundabout Mandatory 1550

No U-Turn 1550

No Stopping 1550

No Horn 1550

No Entry 1550

No Car 1550

Keep Right 1550

Keep Left 1550

Horn 1550

Go Right 1550

Go Left 1550

Don't Overtake from Left 1550

Don't Go Straight 1550

Don't Go Right 1550

Don't Go Left or Right 1550

Don't Go Left 1550

Go Straight or right 1550

Go Straight 1550

Watch out for Cars 1550

**Parameter Settings for Testing Data**

batch\_size = 32

img\_height = 180

img\_width = 180

num\_classes = 30

**Preprocessing**

# TRAIN DATASET created using augmented data and original training data

# Dataset is created using seed=123 and tf.keras.preprocessing.image\_dataset\_from\_directory

# Resizing images to the size img\_height\*img\_width

data\_dir\_train = root\_path+'TRAIN/'

train\_ds = tf.keras.preprocessing.image\_dataset\_from\_directory(

data\_dir\_train,

seed=123,

validation\_split = 0.2,

subset = 'training',

image\_size=(img\_height, img\_width),

batch\_size=batch\_size)

# VALIDATION DATASET created using augmented data and original training data

# Dataset is created using seed=123 and tf.keras.preprocessing.image\_dataset\_from\_directory

# Resizing images to the size img\_height\*img\_width

train\_ds = tf.keras.preprocessing.image\_dataset\_from\_directory(

data\_dir\_train,

seed=123,

validation\_split = 0.2,

subset = 'validation',

image\_size=(img\_height, img\_width),

batch\_size=batch\_size)

Preprocessing is done with the augmented data again with the same parameter settings as before where the seed is 123, image\_height and image\_width is 180 and the batch size is 32.

**CNN Model with Augmented Data**

model = Sequential([layers.Rescaling(1./255, input\_shape=(img\_height, img\_width,3))])

# 1st convolution layer

model.add(Conv2D(filters = 16, kernel\_size = (3,3),padding = 'Same',

activation ='relu'))

model.add(MaxPool2D(pool\_size=(2,2)))

# 2nd convolution layer

model.add(Conv2D(filters = 32, kernel\_size = (3,3),padding = 'Same',

activation ='relu'))

model.add(MaxPool2D(pool\_size=(2,2)))

# 3rd convolution layer

model.add(Conv2D(filters = 32, kernel\_size = (3,3),padding = 'Same',

activation ='relu'))

model.add(MaxPool2D(pool\_size=(2,2)))

# 4th convolution layer

model.add(Conv2D(filters = 64, kernel\_size = (3,3),padding = 'Same',

activation ='relu'))

model.add(MaxPool2D(pool\_size=(2,2)))

# 5th convolution layer with Dropout layer

model.add(Conv2D(filters = 64, kernel\_size = (3,3),padding = 'Same',

activation ='relu'))

model.add(MaxPool2D(pool\_size=(2,2)))

model.add(Dropout(0.25))

model.add(Flatten())

# Fully connected layer 1 with Dropout layer and l2 regularizer

model.add(Dense(128, activation='relu', kernel\_regularizer=l2(0.0105)))

model.add(Dropout(0.20))

# Output layer

model.add(Dense(num\_classes, activation = "softmax"))

**Augmentor with Optimizer and Loss Function**

# Compile the model using 'adam' optimizer and 'SparseCategoricalCrossentropy' loss function

model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True),

metrics=['accuracy'])

# Train the model for 20 epochs

epochs=20

history = model.fit(

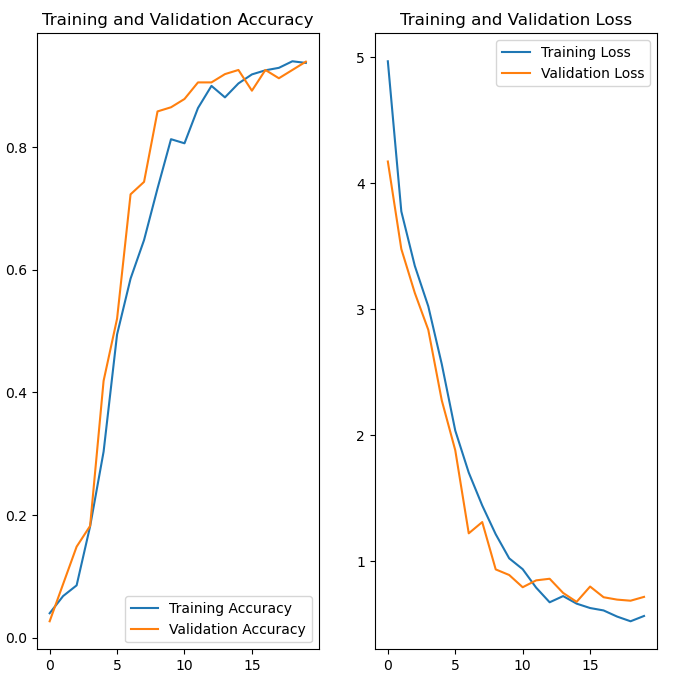
train\_ds,

validation\_data=val\_ds,

epochs=epochs)

The Model is executed with data that has been augmented using Augmentor module from the library, however the process remains the same as we did with the training data.

* The model is created using a Sequential class with 5 convolutional layers, followed by dropout layers, a flatten layer, and fully connected layers.
* Each convolutional layer applies ReLU activation and uses padding='Same', ensuring the output feature map size remains consistent with the input dimensions.
* Max pooling layers are used after each convolutional layer to downsample the feature maps, reducing spatial dimensions and computational load.
* Dropout layers are applied after certain convolutional and fully connected layers with range of 0.2 to prevent overfitting during training.
* The fully connected layer has 128 neurons and uses ReLU as the activation function. It also applies L2 regularization to reduce overfitting by penalizing large weights. The output from this layer is then passed to the final output layer.
* The output layer has num\_classes units and uses the softmax activation function, which is mostly used for multi-class classification tasks.
* The model is compiled using the Adam optimizer and Sparse Categorical Crossentropy loss function.



**Insights**

* The training Accuracy of the Augmented data starts just above 0.0 and ends up around 0.9
* Validation Accuracy starts along with the training accuracy ands ends around the same
* Training Loss starts from 5 at the start of epoch 1 and ends up less than 0.5 at the end of 20 epochs
* Validation Loss starts just above 4 and ends a little higher than the training loss.

**Testing the model**

# Step 1: Extract the 152nd image from validation dataset (for example)

image\_index = 107

inference\_image = list(val\_ds.unbatch().map(lambda x, y: x).take(image\_index + 1))[-1]

# Step 2: Show max, min, shape

print(inference\_image.numpy().max(), inference\_image.numpy().min())

print(inference\_image.shape)

# Step 3: Predict using the model

prediction = model.predict(inference\_image.numpy().reshape(1, 180, 180, 3))

predicted\_class = np.argmax(prediction)

# Step 4: Get the predicted class name

predicted\_class\_name = class\_names[predicted\_class]

# Step 5: Display the image being tested with class name

plt.imshow(inference\_image.numpy().astype("uint8"))

plt.title(f"Predicted Class: {predicted\_class\_name}")

plt.axis("off")

plt.show()

1. We will use the validation dataset val\_ds to access individual images. It uses map(lambda x, y: x) to discard labels and extract only the image tensors. The take(image\_index + 1) method fetches the first 108 images, and the last one ([-1]) is selected for inference.
2. The image is converted to a NumPy array using .numpy() and its maximum, minimum pixel values, and shape are printed. This helps verify that the image is properly normalized and has the expected shape of (180, 180, 3). The pixel values should typically lie between 0 and 1 due to the Rescaling(1./255) layer.
3. The model is used to predict the class of the image by reshaping it into a batch of one image. The model.predict function returns a vector of probabilities across all classes. np.argmax is then used to determine the index of the highest probability, i.e., the predicted class.
4. The predicted class index is mapped to a human-readable class name using the class\_names list. This step translates the numeric prediction into a meaningful label like 'cat' or 'dog'. It’s essential for interpreting the model’s output in real-world terms.
5. Finally, the image is displayed using matplotlib.pyplot with its predicted class title shown above it. The image is converted back to 8-bit integers using .astype("uint8") to render it correctly. The axes are turned off for a cleaner display.

**Conclusion**

* High Accuracy (93.9%) -Our model correctly predicts nearly 94% of the validation samples, indicating it has learned the task well and is generalizing effectively.
* High Precision (95.4%) - A precision score over 95% means the model's positive predictions are very reliable — when it predicts a class, it's likely correct.
* Strong Recall (93.9%) - The recall being nearly equal to accuracy indicates the model is not missing many actual class instances — it's detecting them consistently.
* Balanced F1-Score (93.8%) - The F1-score reflects a strong balance between precision and recall. This suggests the model performs consistently across classes, even in potentially imbalanced data.