Cats and Dogs Image Classification Using Deep Learning with Python

The Ultimate Guide for Building Convolutional Neural Networks (CNN) to Classify Images of Dogs and Cats using Keras and TensorFlow.

Blog By: Chaithanya Vamshi Sai

Student Id: 21152797

Introduction

In todays' data industry, there's a strong belief that when it comes to working with unstructured data, especially image data, deep learning models are the best. Deep learning algorithms like Convolutional Neural Networks (CNN) undoubtedly perform extremely well on image data.

Deep Learning algorithms can handle large amounts of data but it requires a large amount of data to train and also requires a lot of computing resources.

If we provide the right data and features, Deep learning models can perform adequately well and can even be used as an automated solution.

In this blog, I will demonstrate and show how we can harness the power of Deep Learning and perform image classification using Convolutional Neural Networks (CNN) to Classify Images of Dogs and Cats using Keras and TensorFlow.

Steps to Build an Optimised Convolutional Neural Network Image Classifier Model using Keras, TensorFlow & Hyperparameter Tuning with Keras Tuner.

1) Problem Statement
2) Importing Libraries
3) Unzipping and loading the dataset
4) Data Exploration
5) Data Augmentation
6) Model Building
7) Model Evaluation: CNN Model Loss and Accuracy Results
8) Hyperparameter Tuning of the CNN Model using Keras Tuner
9) Model Building: CNN Model with the Best Hyperparameters
10) Model Evaluation: Optimised CNN Model Loss and Accuracy Results

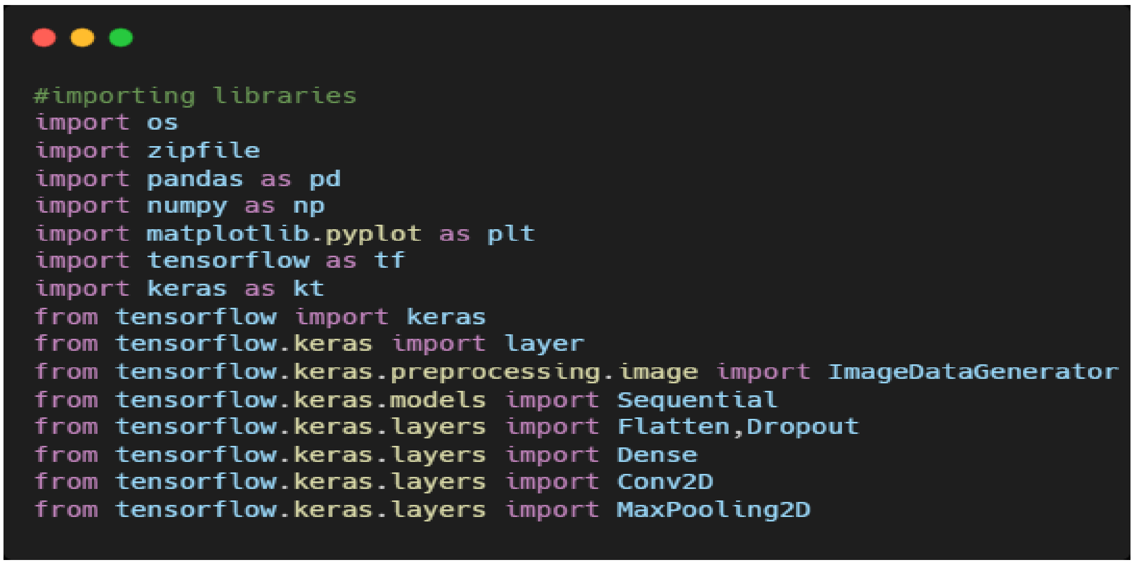
1. Problem Statement

The main objective of this task is we are given a dataset in a zip folder of a few thousand images of cats and dogs which contains separate training and validation directories.

Using this data, we will implement a Convolutional Neural Networks (CNN) to build a Binary Image Classifier by applying Image Augmentation and Hyperparameter tuning techniques using Keras and TensorFlow to achieve the best accuracy and performance.

2. Importing Libraries

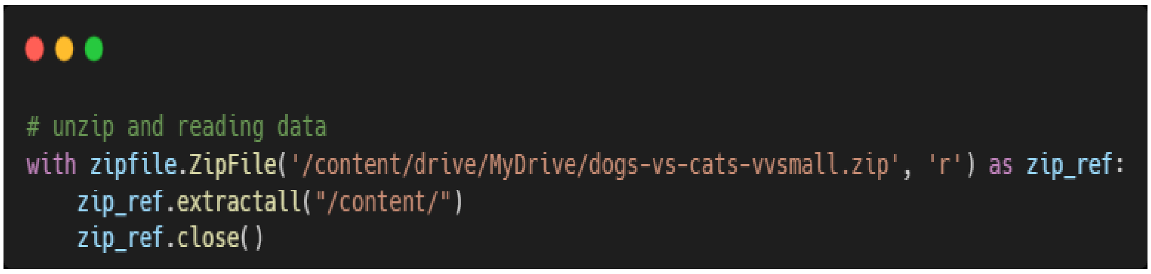
- **NumPy:** For working with arrays, linear algebra.
- **Pandas:** For reading/writing data.
- **Matplotlib:** For display images and plotting model results.
- **Zip file:** For Unzipping and reading the dataset.
- **TensorFlow Keras:** We use Keras, a high-level API to build and train deep learning models in TensorFlow. For more information, refer to the official Keras API documentation.



```
#importing libraries
import os
import zipfile
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import keras as kt
from tensorflow import keras
from tensorflow.keras import layer
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten,Dropout
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
```

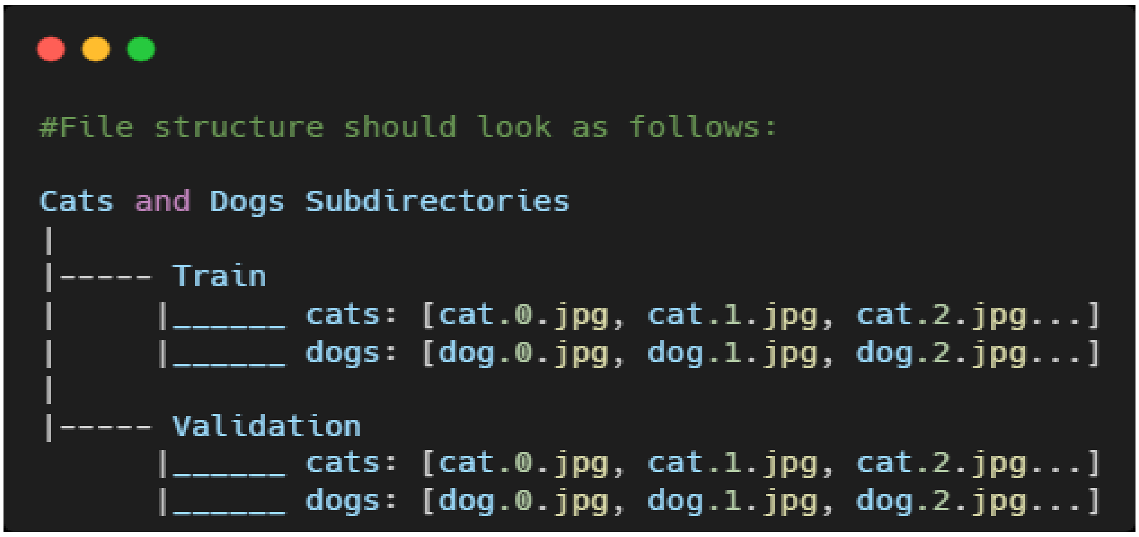
3. Unzipping and Loading the Dataset

We are given a dataset in a zip folder of a few thousand images of cats and dogs which contains separate training and validation subdirectories, which in turn each contains cats and dog subdirectories. By importing the ZipFile library, we will unzip and read the dataset directories.



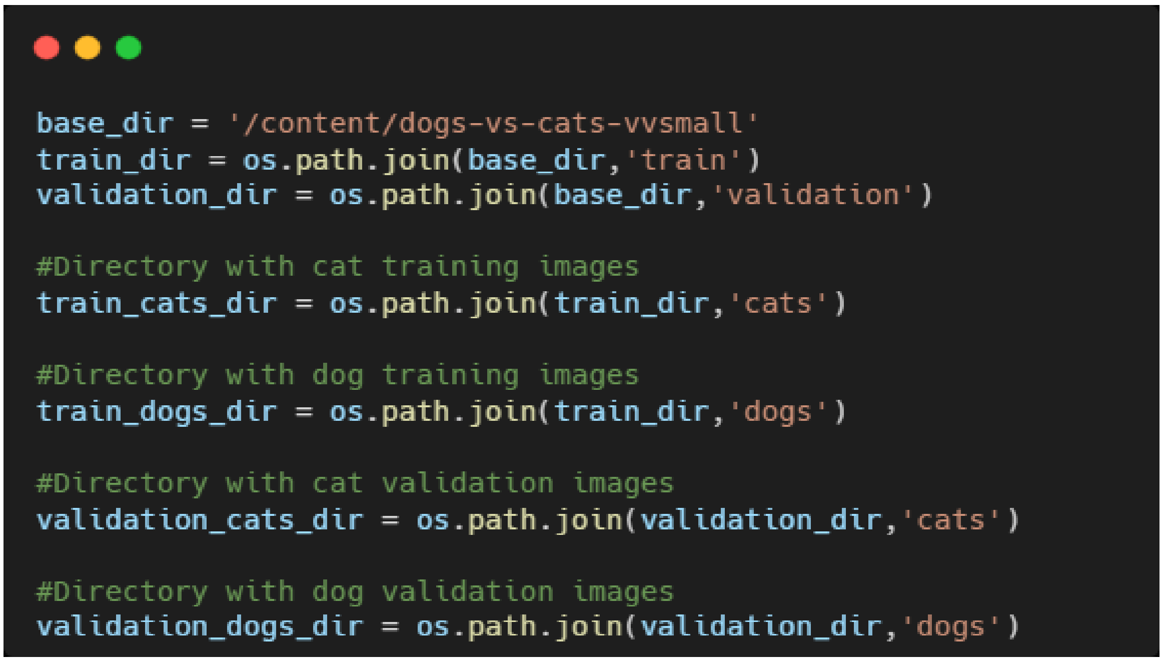
```
# unzip and reading data
with zipfile.ZipFile('/content/drive/MyDrive/dogs-vs-cats-vvsmall.zip', 'r') as zip_ref:
    zip_ref.extractall("/content/")
    zip_ref.close()
```

After unzipping the dataset, the file structure of cats and dogs subdirectories should look as below.



```
#File structure should look as follows:

Cats and Dogs Subdirectories
 |
 |----- Train
 |      |______ cats: [cat.0.jpg, cat.1.jpg, cat.2.jpg...]
 |      |______ dogs: [dog.0.jpg, dog.1.jpg, dog.2.jpg...]
 |
 |----- Validation
        |______ cats: [cat.0.jpg, cat.1.jpg, cat.2.jpg...]
        |______ dogs: [dog.0.jpg, dog.1.jpg, dog.2.jpg...]
```

Therefore, we will divide the dataset into training and validation data frames to perform data exploration in the next step.



```
base_dir = '/content/dogs-vs-cats-vvsmall'
train_dir = os.path.join(base_dir,'train')
validation_dir = os.path.join(base_dir,'validation')

#Directory with cat training images
train_cats_dir = os.path.join(train_dir,'cats')

#Directory with dog training images
train_dogs_dir = os.path.join(train_dir,'dogs')

#Directory with cat validation images
validation_cats_dir = os.path.join(validation_dir,'cats')

#Directory with dog validation images
validation_dogs_dir = os.path.join(validation_dir,'dogs')
```

4. Data Exploration

Summary of the number of images in the cat and dog training and validation dataset.

	Training Data Images	Validation Data Images
Cat	**3000**	**900**
Dog	**3000**	**900**
Total	**6000**	**1800**

5. Data Augmentation

Data Augmentation is a technique that can be used to artificially expand the size of a training dataset by creating modified versions of images in the dataset.

In simple terms, it is a method of applying different kinds of transformation to original images resulting in multiple transformed copies of the same image resulting in more data.

By applying Data Augmentation techniques, we can also avoid **Overfitting**. Overfitting occurs when we have a small number of training samples.

One solution to this problem is to augment our dataset so that it has a sufficient number and variety of training examples. With Data augmentation we generate more data from existing training samples, by applying transformations on images.

In this task, we will apply different image data augmentation techniques like

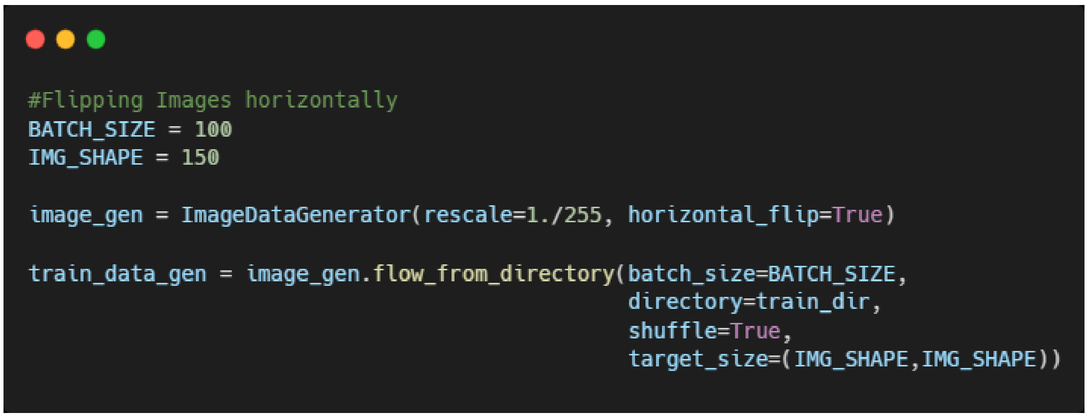
- Flipping
- Rotating
- Zooming

5.1 Image Augmentation with ImageDataGenerator

The Keras deep learning neural network library provides the capability to fit models using image data augmentation using the **ImageDataGenerator** class.

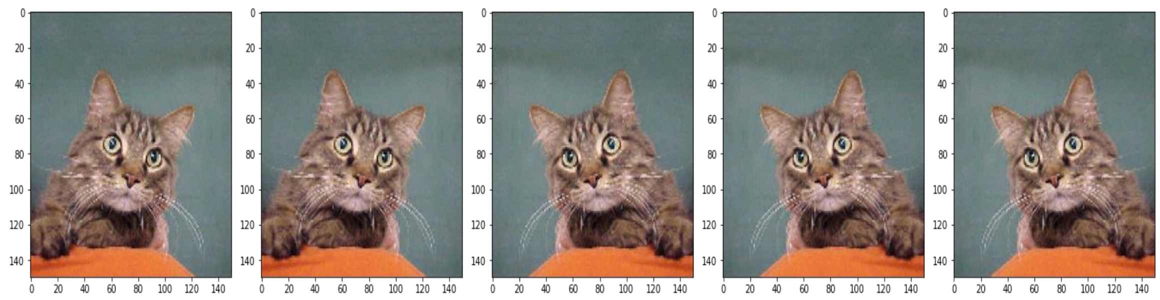
a) Flipping Images Horizontally

Let’s apply horizontal flip augmentation to our dataset and check how individual images will look after the transformation. We can do this by making “**horizontal_flip=True**” as an argument to the **ImageDataGenerator** class.



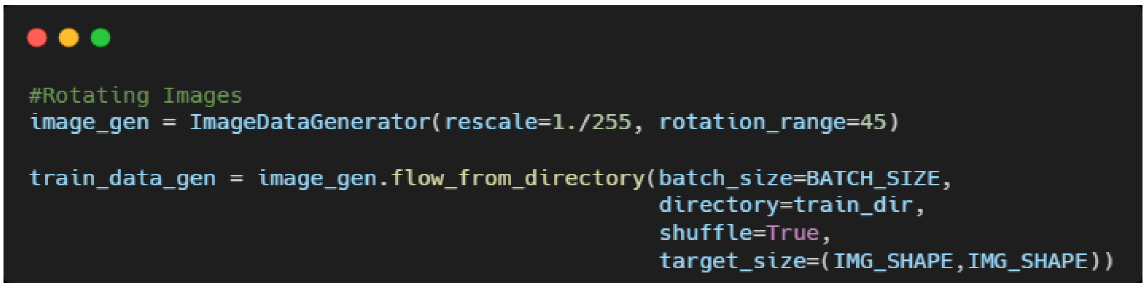
```
#Flipping Images horizontally
BATCH_SIZE = 100
IMG_SHAPE = 150

image_gen = ImageDataGenerator(rescale=1./255, horizontal_flip=True)

train_data_gen = image_gen.flow_from_directory(batch_size=BATCH_SIZE,
                                               directory=train_dir,
                                               shuffle=True,
                                               target_size=(IMG_SHAPE,IMG_SHAPE))
```

Displaying Images after Horizontal Flipping Augmentation



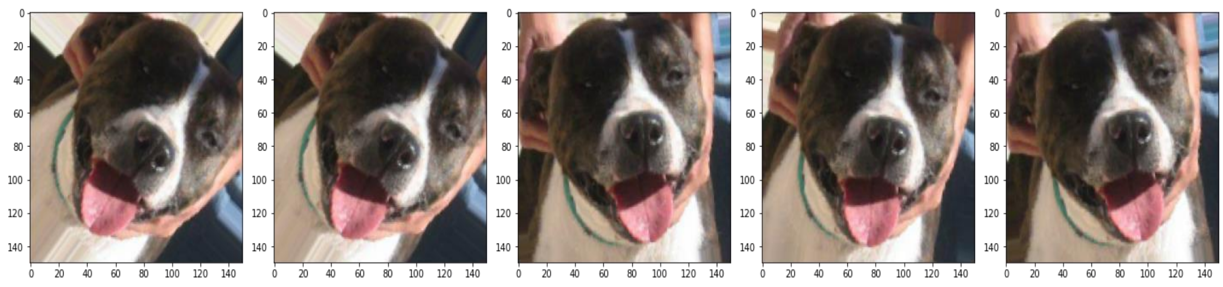
b) Rotating Images

Let’s apply rotating image augmentation which will randomly rotate the image up to a specified number of degrees. Here, we’ll set it to 45.



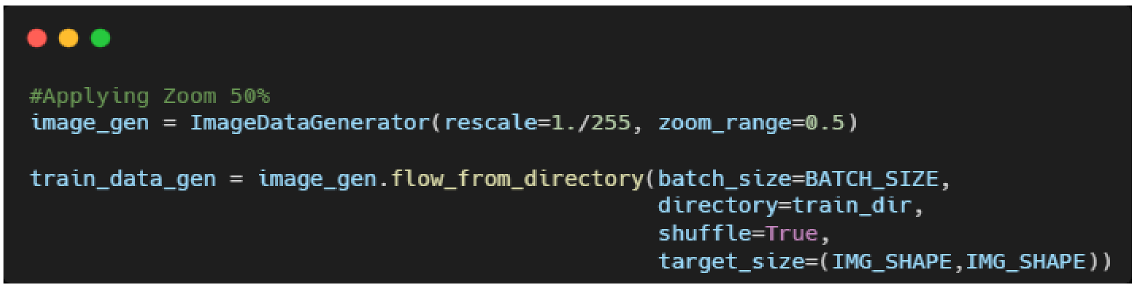
```
#Rotating Images
image_gen = ImageDataGenerator(rescale=1./255, rotation_range=45)

train_data_gen = image_gen.flow_from_directory(batch_size=BATCH_SIZE,
                                               directory=train_dir,
                                               shuffle=True,
                                               target_size=(IMG_SHAPE,IMG_SHAPE))
```

Displaying Images after Rotating Image Augmentation



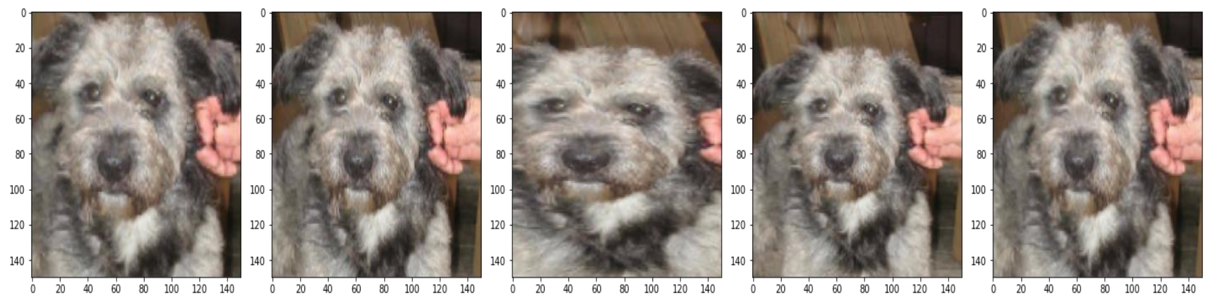
c) Zooming Images

Let’s apply zooming image augmentation which will zoom the image and set it to 50%.



```
#Applying Zoom 50%
image_gen = ImageDataGenerator(rescale=1./255, zoom_range=0.5)

train_data_gen = image_gen.flow_from_directory(batch_size=BATCH_SIZE,
                                               directory=train_dir,
                                               shuffle=True,
                                               target_size=(IMG_SHAPE,IMG_SHAPE))
```

Displaying Images after Zooming Augmentation



6. Model Building

6.1 Creating Training Data Generator

We will apply all data augmentation techniques together on our training dataset generator for model training and we will only add the rescaling transformation to the validation dataset generator.

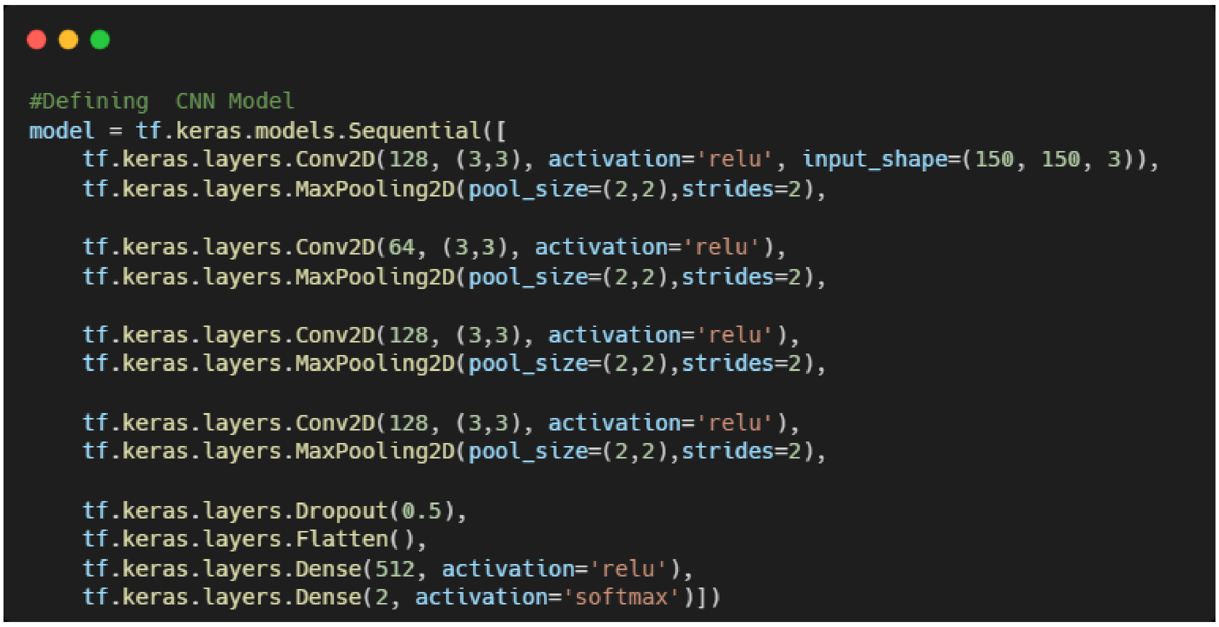
```
#Creating Training dataset Generator
image_gen_train = ImageDataGenerator(rescale=1./255,rotation_range=45,
      width_shift_range=0.2,height_shift_range=0.2,shear_range=0.2,
      zoom_range=0.5,horizontal_flip=True,fill_mode='nearest')

train_data_gen = image_gen_train.flow_from_directory(batch_size=BATCH_SIZE,
                                                     directory=train_dir,
                                                     shuffle=True,
                                                     target_size=(IMG_SHAPE,IMG_SHAPE),
                                                     class_mode='binary')
```

6.2 Creating Validation Data Generator

```
#Creating Validation dataset Generator

image_gen_val = ImageDataGenerator(rescale=1./255)

val_data_gen = image_gen_val.flow_from_directory(batch_size=BATCH_SIZE,
                                                 directory=validation_dir,
                                                 target_size=(IMG_SHAPE, IMG_SHAPE),
                                                 class_mode='binary')
```

6.3 Defining the CNN Model

- The model consists of **four Convolution (Conv2D)** layers with a **Max pool (MaxPool2D)** layer in each of them.
- **Dropout** is a regularization method, where a proportion of nodes in the layer are randomly ignored by setting their weights to zero for each training sample. This technique also **improves generalization** and **reduces overfitting**. Hence, to avoid that we will be using a Dropout layer with a probability of 50%.
- The **Flatten layer** is used to convert the final feature maps into a single 1D vector. This flattening step is needed to make use of fully connected layers after some convolutional/max_pool layers.
- Further, we have a fully connected **Dense layer** with 512 units and '**Relu**' is the rectifier activation function that is used to add nonlinearity to the network.
- In the end, our model will be giving probabilities for two classes (Dog or Cat) we will use the **'SoftMax'** function.

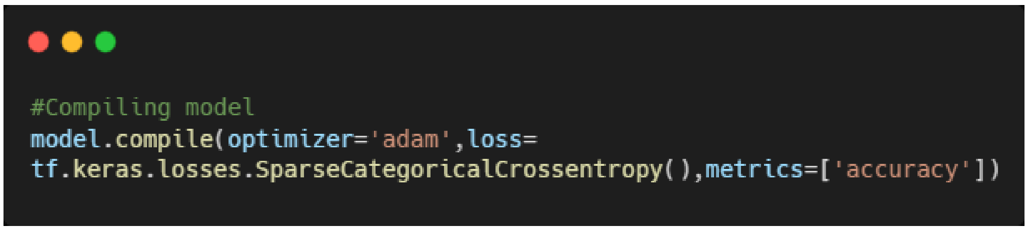


```
#Defining  CNN Model
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(128, (3,3), activation='relu', input_shape=(150, 150, 3)),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2),strides=2),

    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2),strides=2),

    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2),strides=2),

    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2),strides=2),

    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(2, activation='softmax')])
```

6.4 Compiling the Model

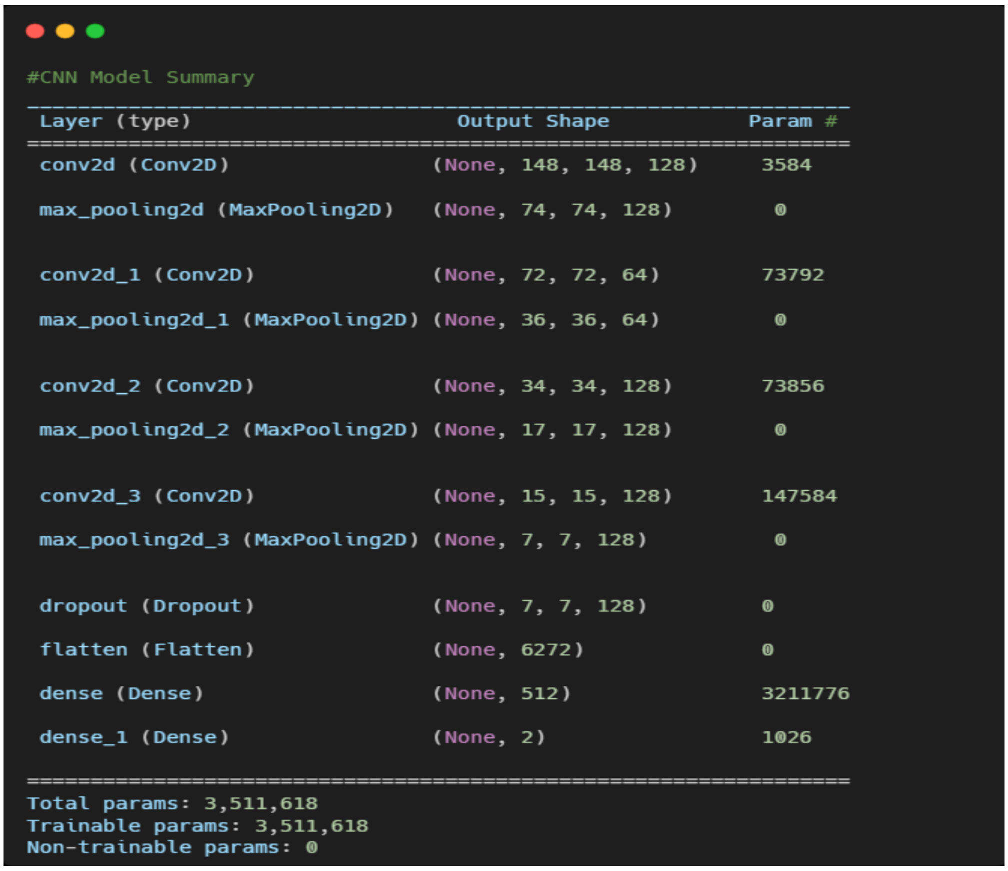
Once our layers are added to the model, we need to set up a score function, a loss function, and an optimization algorithm.

- We define the **Loss function** to measure how poorly our model performs on images with known labels and we will use "**sparse_categorical_crossentropy**".
- The most important function is the **Optimizer**. This function will iteratively improve parameters to minimize the loss. We will go with the "**Adam**" optimizer.
- The metric function "**Accuracy**" is used to evaluate our model's performance by checking the training and validation accuracy of the model.



```
#Compiling model
model.compile(optimizer='adam',loss=
tf.keras.losses.SparseCategoricalCrossentropy(),metrics=['accuracy'])
```

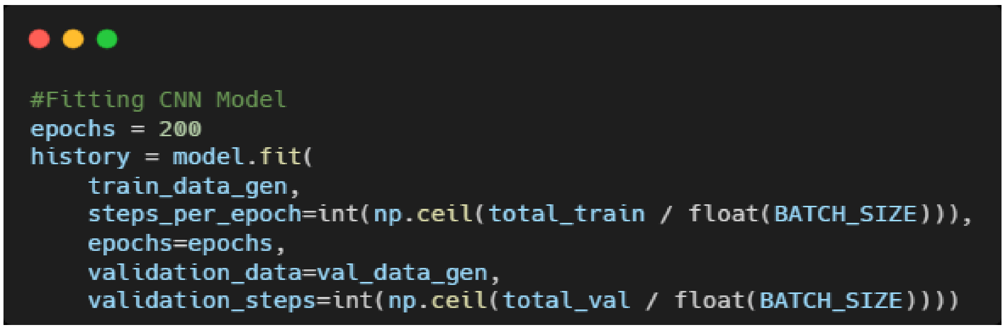
6.5 CNN Model Summary



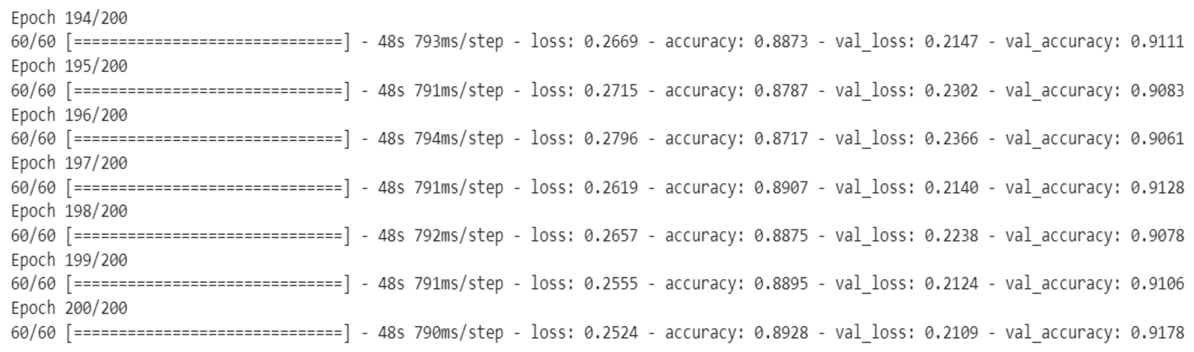
```
#CNN Model Summary
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 148, 148, 128)     3584

 max_pooling2d (MaxPooling2D)  (None, 74, 74, 128)       0


 conv2d_1 (Conv2D)           (None, 72, 72, 64)        73792

 max_pooling2d_1 (MaxPooling2D) (None, 36, 36, 64)      0


 conv2d_2 (Conv2D)           (None, 34, 34, 128)       73856

 max_pooling2d_2 (MaxPooling2D) (None, 17, 17, 128)     0


 conv2d_3 (Conv2D)           (None, 15, 15, 128)       147584

 max_pooling2d_3 (MaxPooling2D) (None, 7, 7, 128)       0


 dropout (Dropout)           (None, 7, 7, 128)         0

 flatten (Flatten)           (None, 6272)              0

 dense (Dense)               (None, 512)               3211776

 dense_1 (Dense)             (None, 2)                 1026

=================================================================
Total params: 3,511,618
Trainable params: 3,511,618
Non-trainable params: 0
```

6.5 Model Training

To start training, we will use the "**model.fit**" method to train the data and parameters with **epochs = 200**.



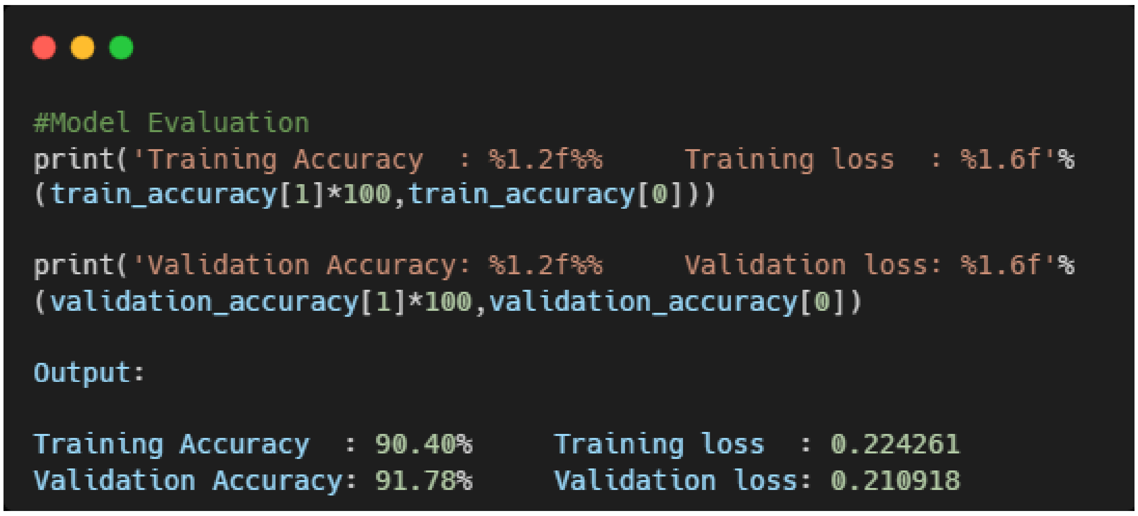
```
#Fitting CNN Model
epochs = 200
history = model.fit(
    train_data_gen,
    steps_per_epoch=int(np.ceil(total_train / float(BATCH_SIZE))),
    epochs=epochs,
    validation_data=val_data_gen,
    validation_steps=int(np.ceil(total_val / float(BATCH_SIZE))))
```



```
Epoch 194/200
60/60 [==============================] - 48s 793ms/step - loss: 0.2669 - accuracy: 0.8873 - val_loss: 0.2147 - val_accuracy: 0.9111
Epoch 195/200
60/60 [==============================] - 48s 791ms/step - loss: 0.2715 - accuracy: 0.8787 - val_loss: 0.2302 - val_accuracy: 0.9083
Epoch 196/200
60/60 [==============================] - 48s 794ms/step - loss: 0.2796 - accuracy: 0.8717 - val_loss: 0.2366 - val_accuracy: 0.9061
Epoch 197/200
60/60 [==============================] - 48s 791ms/step - loss: 0.2619 - accuracy: 0.8907 - val_loss: 0.2140 - val_accuracy: 0.9128
Epoch 198/200
60/60 [==============================] - 48s 792ms/step - loss: 0.2657 - accuracy: 0.8875 - val_loss: 0.2238 - val_accuracy: 0.9078
Epoch 199/200
60/60 [==============================] - 48s 791ms/step - loss: 0.2555 - accuracy: 0.8895 - val_loss: 0.2124 - val_accuracy: 0.9106
Epoch 200/200
60/60 [==============================] - 48s 790ms/step - loss: 0.2524 - accuracy: 0.8928 - val_loss: 0.2109 - val_accuracy: 0.9178
```

7. Model Evaluation

As the model training is completed, the loss and accuracy metrics are displayed.



```
#Model Evaluation
print('Training Accuracy  : %1.2f%%     Training loss  : %1.6f'%
(train_accuracy[1]*100,train_accuracy[0]))

print('Validation Accuracy: %1.2f%%     Validation loss: %1.6f'%
(validation_accuracy[1]*100,validation_accuracy[0]))

Output:

Training Accuracy  : 90.40%     Training loss  : 0.224261
Validation Accuracy: 91.78%     Validation loss: 0.210918
```

Results:

60/60 - 41s 681ms/step - loss: 0.2243 - accuracy: 0.9040

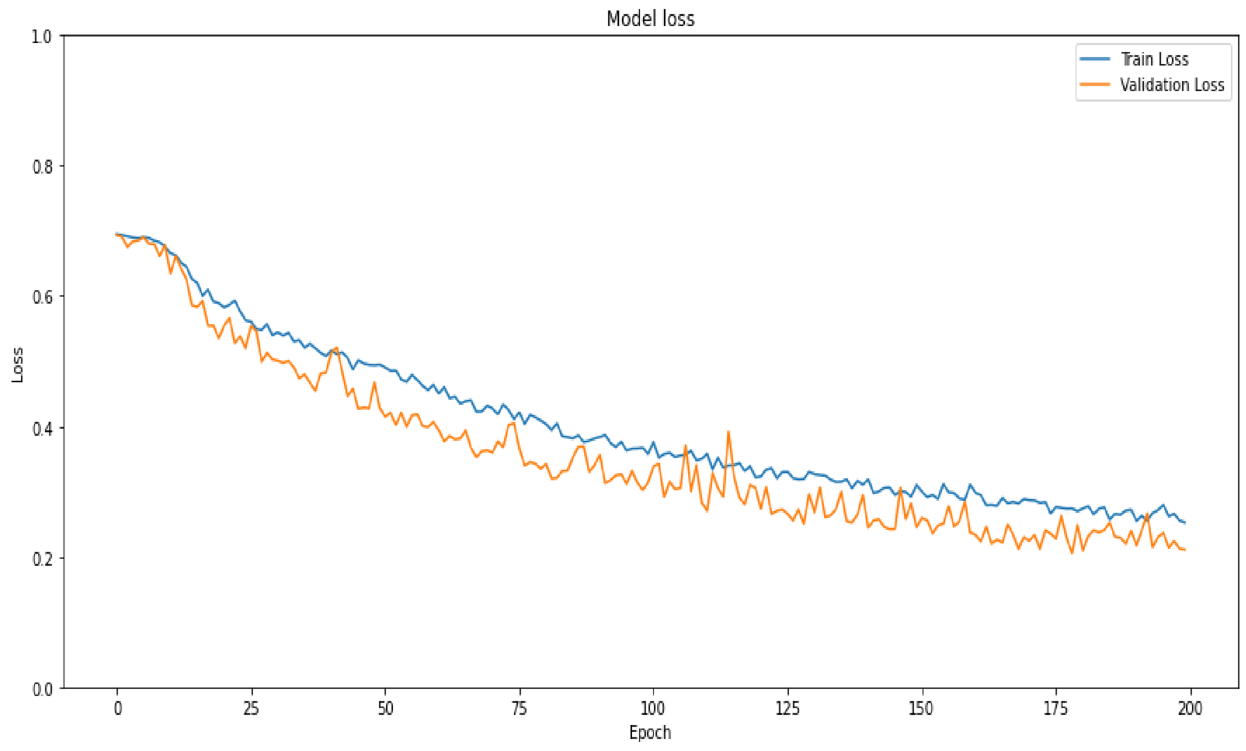
Training Accuracy: 90.40%

18/18 - 5s 273ms/step - loss: 0.2109 - accuracy: 0.9178

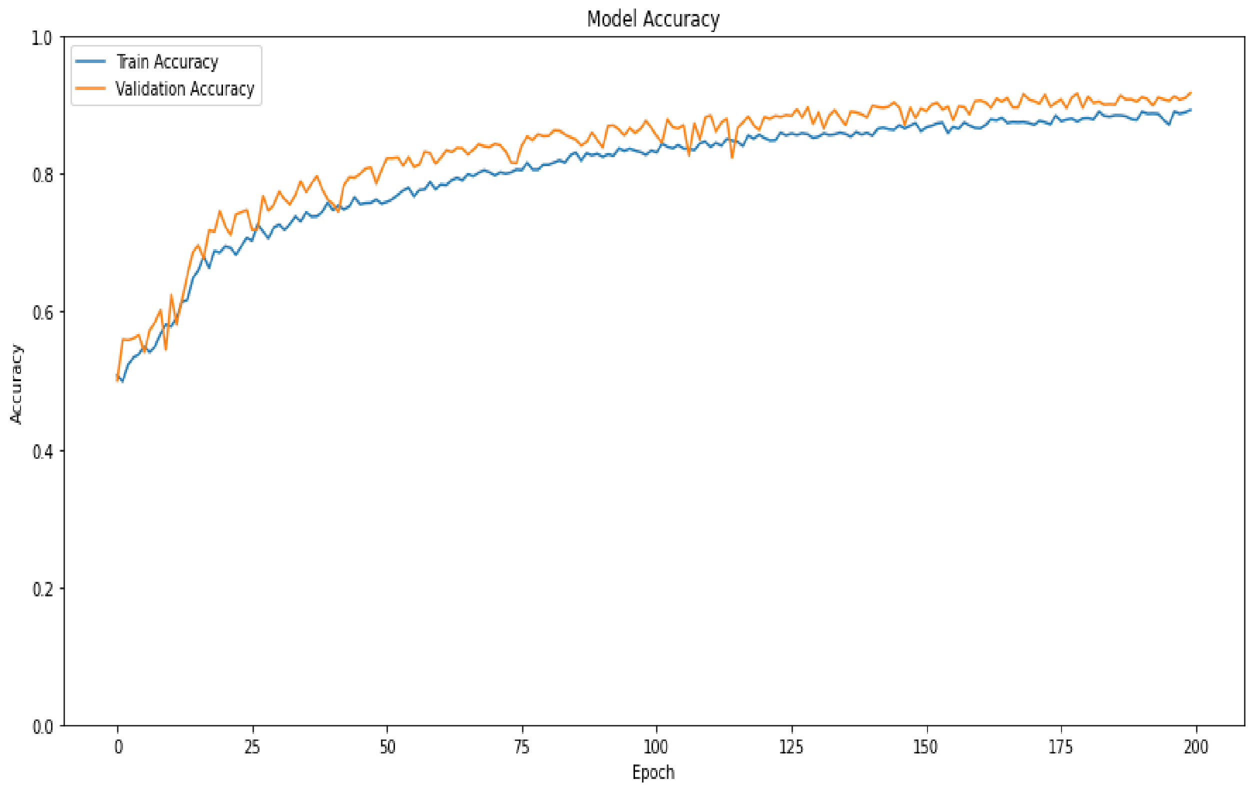
Validation Accuracy: 91.78%

- We can observe that the model accuracy of the **Training data** is 90.40% and the **validation data** is **91.78% (≈92%)** after 200 epochs.
- The validation accuracy is slightly greater than the training accuracy in almost every training. That means that our model doesn't overfit the training set.
- Hence, we can say that our Convolutional Neural Network (CNN) model is more **generalized** and prevented overfitting.

7.1 CNN Model Loss on Train and Validation Data



7.2 CNN Model Accuracy on Train and Validation Data



8. Hyperparameter Tuning of the CNN Model using Keras Tuner

Hyperparameter Tuning

The process of selecting the right set of hyperparameters for the ML/DL model is called Hyperparameter tuning.

In this task, we will use the Bayesian Hyperparameter tuning technique using Keras Tuner API. In Bayesian optimization, the performance function is modelled as a sample from a Gaussian process over the hyperparameter value.

Bayesian optimization takes advantage of previously tested combinations to sample the next one more efficiently.

8.1 Implementation of Bayesian optimization Hyperparameter Tuning of CNN Model using Keras Tuner

1. Hyperparameter Tuning "Learning Rate"
2. Hyperparameter Tuning "Dropout Rate"
3. Hyperparameter Tuning "Convolutional (Conv2D) Layers"
4. Hyperparameter Tuning "Convolutional Kernel (Conv_Kernel)"
5. Hyperparameter Tuning "Dense Layers"

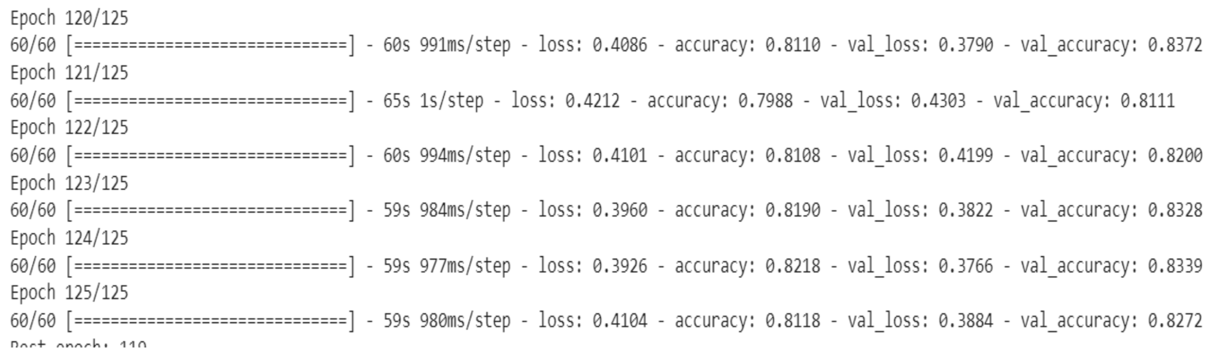
Python Implementation of Hyperparameter tuning of the CNN Model using Bayesian Optimisation is available on **GitHub**

The best hyperparameters obtained on the dataset after rigorous hours of training and computing resources are shown below using Bayesian Optimisation hyperparameter tuning.

Hyperparameter	Hyperparameter Values	Best Hyperparameter Values
Learning Rate	[0.0001, 0.001, 0.01]	0.0001
Dropout Rate 1	Range (0.0,0.5,0.05)	0.0
Dropout Rate 2	Range (0.0,0.5,0.05)	0.0
Conv2D Layer 1	Range (16,128,32)	112
Conv2D Layer 2	Range (32,128,32)	96
Conv_Kernel 1	Range (3,5)	5
Conv_Kernel 2	Range (3,5)	5
Dense Layer	Range (16,128,32)	112

8.2 Model Training Neural Network with the Best Hyperparameters

Building the model with the optimal hyperparameters obtained and training it on the data for **epochs = 125**.



```
Epoch 120/125
60/60 [==============================] - 60s 991ms/step - loss: 0.4086 - accuracy: 0.8110 - val_loss: 0.3790 - val_accuracy: 0.8372
Epoch 121/125
60/60 [==============================] - 65s 1s/step - loss: 0.4212 - accuracy: 0.7988 - val_loss: 0.4303 - val_accuracy: 0.8111
Epoch 122/125
60/60 [==============================] - 60s 994ms/step - loss: 0.4101 - accuracy: 0.8108 - val_loss: 0.4199 - val_accuracy: 0.8200
Epoch 123/125
60/60 [==============================] - 59s 984ms/step - loss: 0.3960 - accuracy: 0.8190 - val_loss: 0.3822 - val_accuracy: 0.8328
Epoch 124/125
60/60 [==============================] - 59s 977ms/step - loss: 0.3926 - accuracy: 0.8218 - val_loss: 0.3766 - val_accuracy: 0.8339
Epoch 125/125
60/60 [==============================] - 59s 980ms/step - loss: 0.4104 - accuracy: 0.8118 - val_loss: 0.3884 - val_accuracy: 0.8272
Best epoch: 110
```

8.3 Model Evaluation Neural Network with the Best Hyperparameters

As the model training is completed, the loss and accuracy metrics are displayed.

Results:

60/60 - 48s 790ms/step - loss: 0.3876 - accuracy: 0.8238

Training Accuracy: 82.38%

18/18 - 6s 322ms/step - loss: 0.3884 - accuracy: 0.8272

Validation Accuracy: 82.72%

8.4 Overall Model Accuracy Results Summary

CNN Model	Epochs	Hyperparameter Tuning	Training Accuracy	Testing Accuracy
Model 1	**200**	**No**	**90.40%**	**91.78%**
Model 2	**125**	**Yes**	**82.38%**	**82.72%**

Overall Model Accuracy Results Summary

- The first CNN model is trained for **200 epochs** and the Accuracy given by the Training set is **90.40%** and the Accuracy given by the Testing set is **91.78%. (≈92%)**
- The model with the best hyperparameters is only trained for **125 epochs** and the Accuracy given by the Training set is **82.38%** and the Accuracy given by the Testing set is **82.72%**.

Deep Learning algorithms can handle large amounts of data but it requires a large amount of training and computing resources.

Since the training process is more time consuming and I believe if we can increase the number of epochs from 125 to 200 to the hyperparameter model we can see an improvement in the accuracy.

Hence, we can summarize that both the neural network models are more generalized (learned well) and prevented overfitting.

9. Conclusion

In this blog, we discussed how to approach the image classification problem by implementing the Convolutional Neural networks (CNN) model using Keras, TensorFlow with **92% accuracy**.

We can explore this work further by trying to improve the accuracy by using advanced Deep Learning algorithms and hyperparameter tuning techniques.

10. References

[1] https://www.tensorflow.org/tutorials/keras/classification

[2] https://keras.io/guides/keras_tuner/getting_started/

[3] https://keras.io/api/keras_tuner/tuners/bayesian/#bayesianoptimization-class

[4] https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/