Machine Learning Project

Animal Images

Team Members

Mohanad Mohamed Ahmed

Youssef Mohamed Salah

Mohamed Yossri mahmoud

Mohamed Hamdy hessin

Sohile Adel Abd-Alghafoor

INTRODUCTION

Machine learning has revolutionized the way we solve complex problems in various fields, and image classification is one of the most popular applications of machine learning.

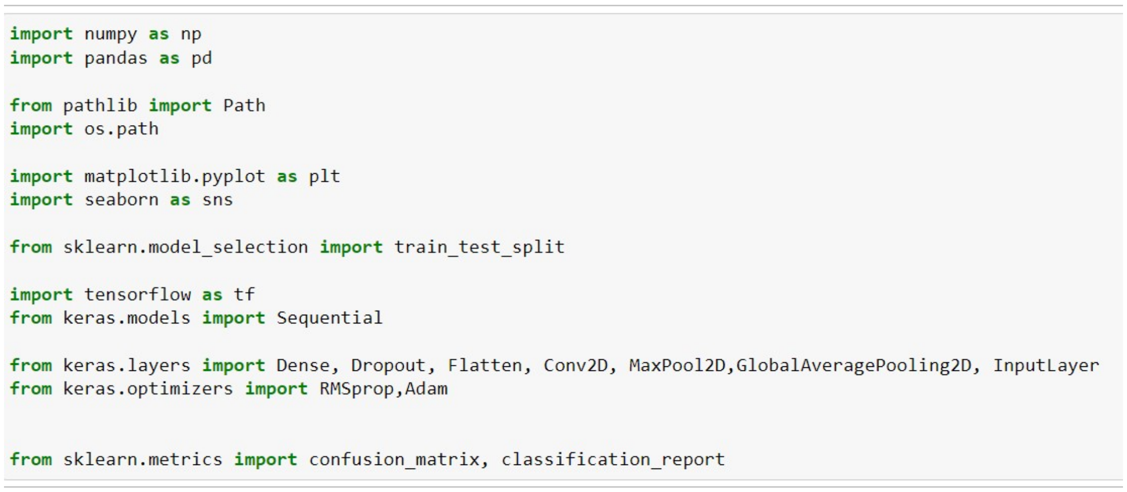
In this project, we aim to use machine learning algorithms to classify images of dogs and cats.

The data set used in this project will consist of a large number of images of cats and dogs. We will use various machine learning algorithms, such as Convolution Neural Networks (CNNs), and Artificial Neural Networks (ANNs) to train the model to recognize the features that distinguish cats from dogs

The final goal of this project is to create a model that can accurately classify images of dogs and cats with high accuracy.

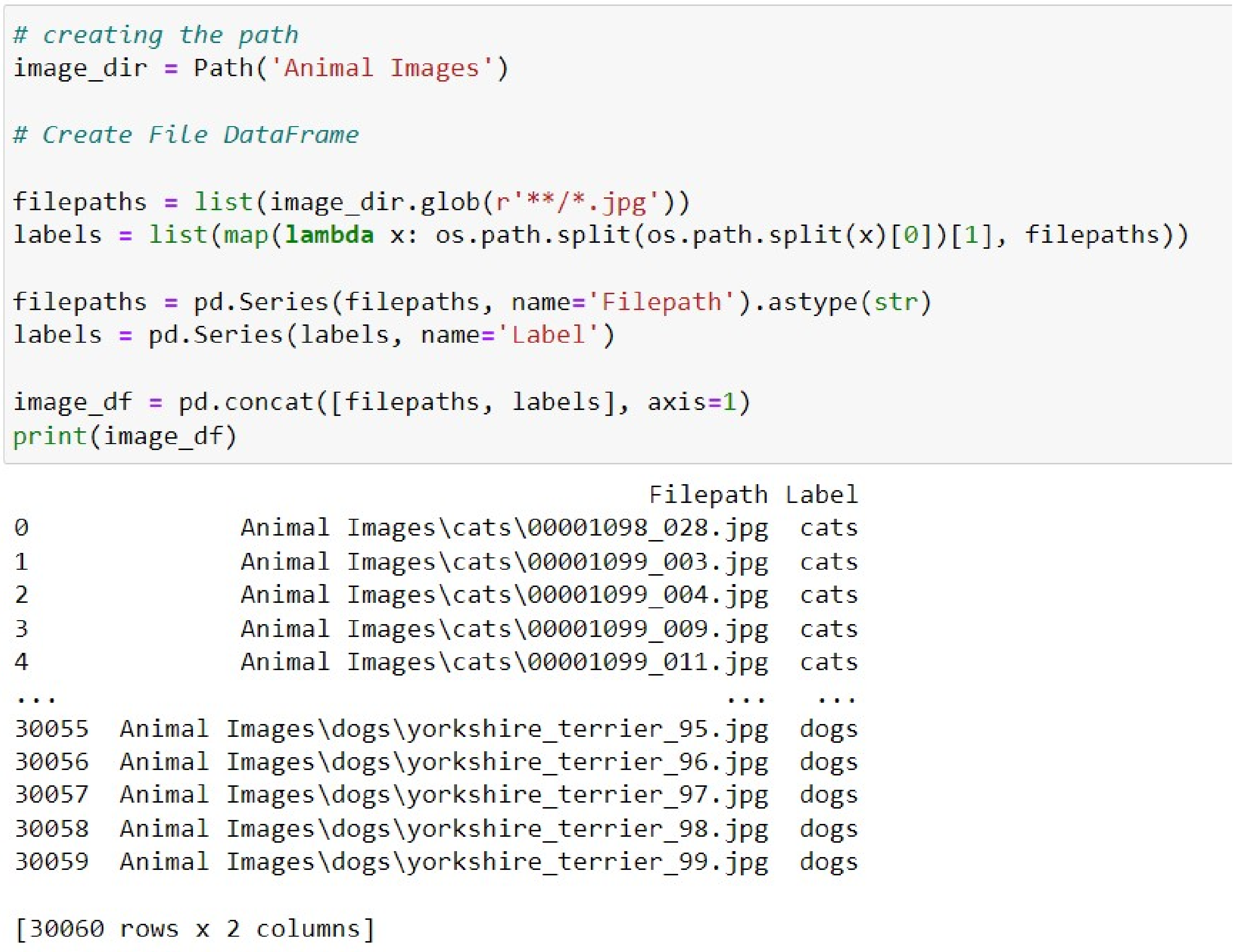
The first phase:

1-Importing Libraries and showing data



```
import numpy as np
import pandas as pd

from pathlib import Path
import os.path

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split

import tensorflow as tf
from keras.models import Sequential

from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPool2D,GlobalAveragePooling2D, InputLayer
from keras.optimizers import RMSprop,Adam


from sklearn.metrics import confusion_matrix, classification_report
```

Importing this libraries for using it in the code.



```
# creating the path
image_dir = Path('Animal Images')

# Create File DataFrame

filepaths = list(image_dir.glob(r'**/*.jpg'))
labels = list(map(lambda x: os.path.split(os.path.split(x)[0])[1], filepaths))

filepaths = pd.Series(filepaths, name='Filepath').astype(str)
labels = pd.Series(labels, name='Label')

image_df = pd.concat([filepaths, labels], axis=1)
print(image_df)
```

```
                                          Filepath Label
0                Animal Images\cats\00001098_028.jpg  cats
1                Animal Images\cats\00001099_003.jpg  cats
2                Animal Images\cats\00001099_004.jpg  cats
3                Animal Images\cats\00001099_009.jpg  cats
4                Animal Images\cats\00001099_011.jpg  cats
...                                              ...   ...
30055  Animal Images\dogs\yorkshire_terrier_95.jpg  dogs
30056  Animal Images\dogs\yorkshire_terrier_96.jpg  dogs
30057  Animal Images\dogs\yorkshire_terrier_97.jpg  dogs
30058  Animal Images\dogs\yorkshire_terrier_98.jpg  dogs
30059  Animal Images\dogs\yorkshire_terrier_99.jpg  dogs

[30060 rows x 2 columns]
```

In this code cell, a file DataFrame is created to store the file paths and corresponding labels of images in a directory.

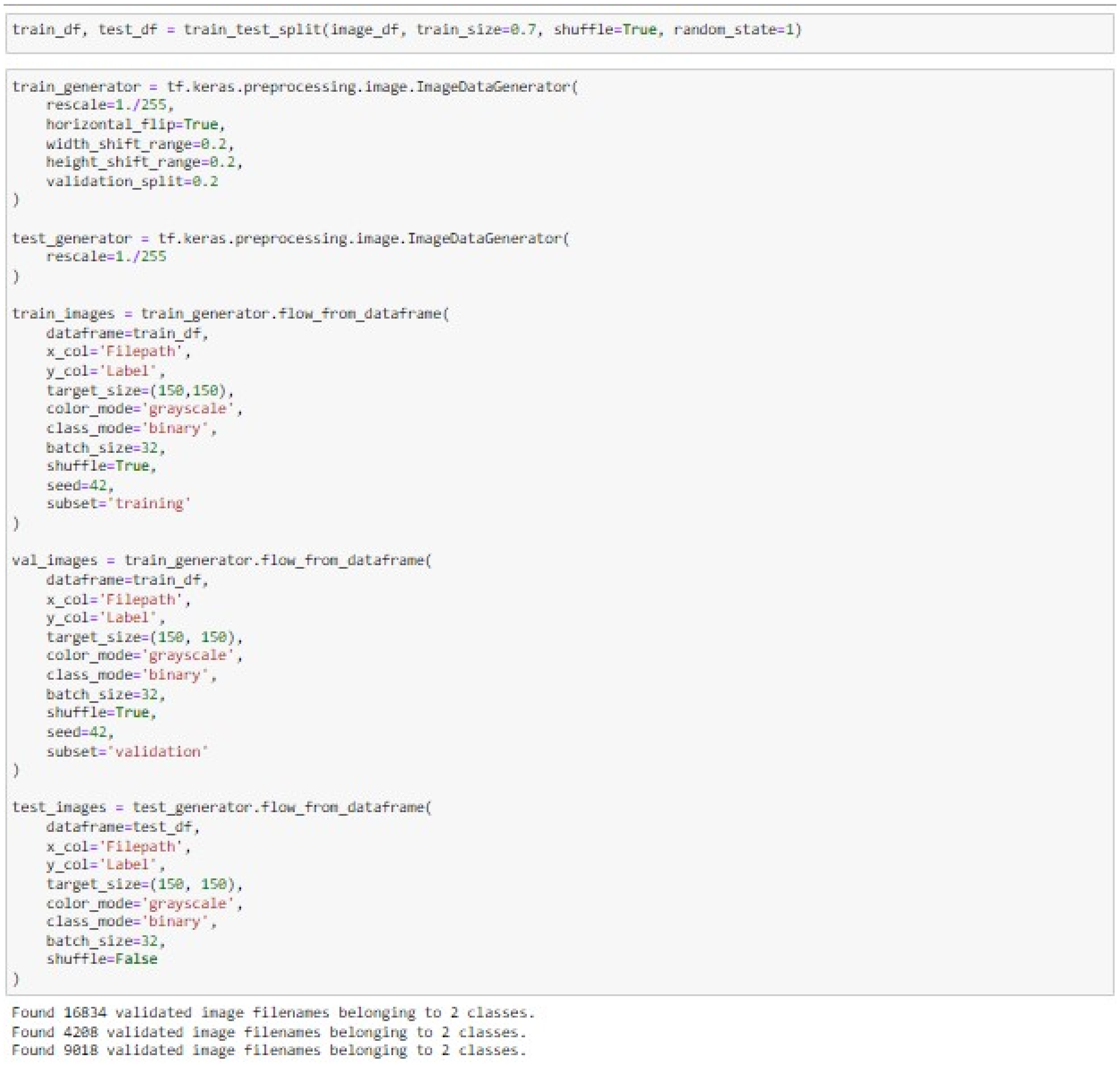
Then creating a Path object in a variable named '**image_dir**' representing the directory path 'Animal Images'. The Path class is part of the pathlib module and provides convenient methods for working with file paths.

Then creating file data frame by creating a variable '**filepath**' which stores the glob method of the Path object to recursively search for all files with the extension '.jpg' in the specified directory and its subdirectories. The ** pattern matches any number of subdirectories.

Then extracting the labels from the file paths and store it in variable named **'labels'**. The os.path.split function splits a path into the directory part and the filename part. In this case, it is called twice to get the parent directory name, which represents the label. The map function applies this operation to each file path in the filepaths list.

Finally, the filepaths and labels lists are converted into pandas series and then combined into a DataFrame called **image_df** using the pd.concat() function.

2- Splitting data for training ,validation ,and testing.



```
train_df, test_df = train_test_split(image_df, train_size=0.7, shuffle=True, random_state=1)
```

```
train_generator = tf.keras.preprocessing.image.ImageDataGenerator(
    rescale=1./255,
    horizontal_flip=True,
    width_shift_range=0.2,
    height_shift_range=0.2,
    validation_split=0.2
)

test_generator = tf.keras.preprocessing.image.ImageDataGenerator(
    rescale=1./255
)

train_images = train_generator.flow_from_dataframe(
    dataframe=train_df,
    x_col='Filepath',
    y_col='Label',
    target_size=(150,150),
    color_mode='grayscale',
    class_mode='binary',
    batch_size=32,
    shuffle=True,
    seed=42,
    subset='training'
)

val_images = train_generator.flow_from_dataframe(
    dataframe=train_df,
    x_col='Filepath',
    y_col='Label',
    target_size=(150, 150),
    color_mode='grayscale',
    class_mode='binary',
    batch_size=32,
    shuffle=True,
    seed=42,
    subset='validation'
)

test_images = test_generator.flow_from_dataframe(
    dataframe=test_df,
    x_col='Filepath',
    y_col='Label',
    target_size=(150, 150),
    color_mode='grayscale',
    class_mode='binary',
    batch_size=32,
    shuffle=False
)
```

```
Found 16834 validated image filenames belonging to 2 classes.
Found 4208 validated image filenames belonging to 2 classes.
Found 9018 validated image filenames belonging to 2 classes.
```

First, we splatted the data into 70% training and 30% testing using **train_test_split** function.

Then we used **'tf.keras.preprocessing.image.ImageDataGenerator**' which is used to generate batches of image data for training, validation, and testing. The data generators provide a way to preprocess and increase the image data during model training.

And the The **'flow_from_dataframe()'** method is used to generate the data flow for testing/validation.

3- Visualization.

```
for i in range(20):
  plt.subplot(4,5,i+1)
  batch = train_images[i]
  x, y = batch
  plt.imshow(x[i],cmap='gray_r')
  plt.title("is : {:.0f}".format(y[i]))
  plt.subplots_adjust(hspace=0.5)
  plt.axis('off')
```

for i in range(20): creates a loop that iterates 20 times. This loop is used to display 20 images with their labels.

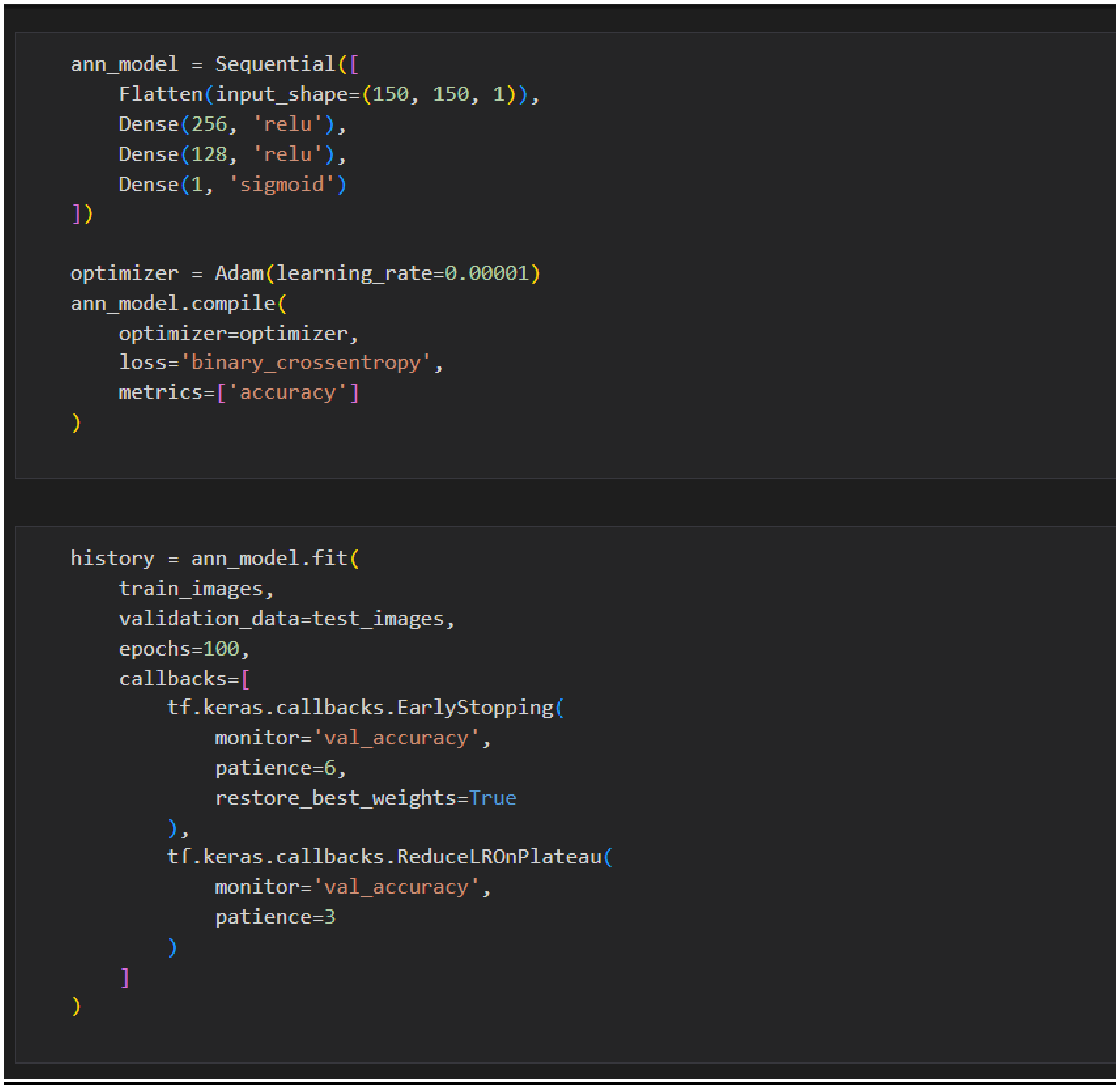
plt.subplot(4,5,i+1) defines a subplot within the grid. The arguments (4, 5, i+1) specify that the grid should have 4 rows, 5 columns, and the index of the current iteration plus 1. This creates a grid of subplots with 20 cells.

batch = train_images[i] retrieves the i-th batch of images from the train_images generator. Each batch typically contains multiple images and their labels.

x, y = batch unpacks the batch into x (images) and y (labels).

plt.imshow(x[i], cmap='gray_r') displays the i-th image from the batch using imshow() function from matplotlib. cmap='gray_r' specifies that the colormap

ANN



```python
ann_model = Sequential([
    Flatten(input_shape=(150, 150, 1)),
    Dense(256, 'relu'),
    Dense(128, 'relu'),
    Dense(1, 'sigmoid')
])

optimizer = Adam(learning_rate=0.00001)
ann_model.compile(
    optimizer=optimizer,
    loss='binary_crossentropy',
    metrics=['accuracy']
)
```

```python
history = ann_model.fit(
    train_images,
    validation_data=test_images,
    epochs=100,
    callbacks=[
        tf.keras.callbacks.EarlyStopping(
            monitor='val_accuracy',
            patience=6,
            restore_best_weights=True
        ),
        tf.keras.callbacks.ReduceLROnPlateau(
            monitor='val_accuracy',
            patience=3
        )
    ]
)
```

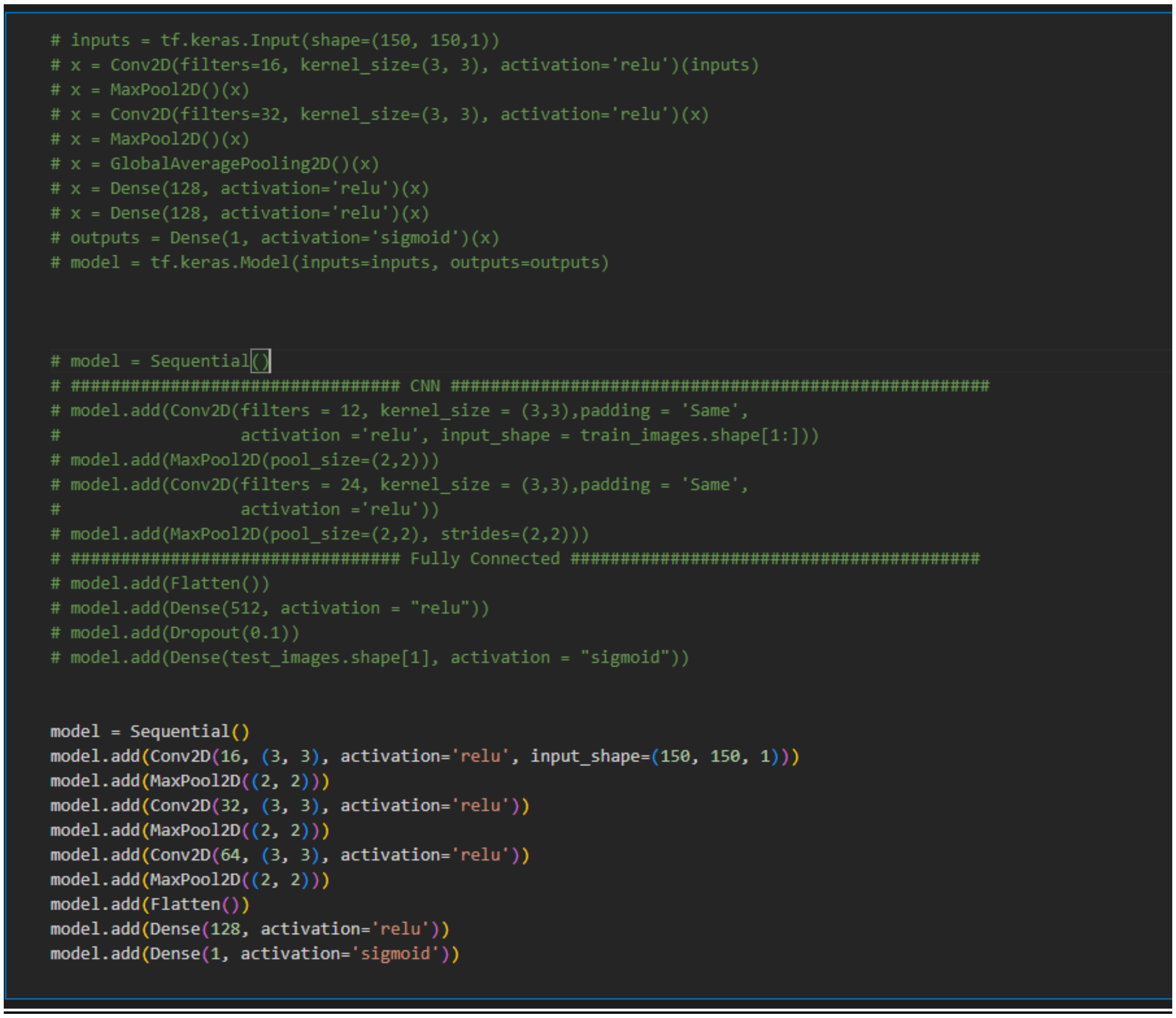
ANNs consist of an input layer, hidden layers, and an output layer. Each neuron in a layer is connected to every neuron in the previous layer.

ANNs are trained using back propagation, where the error between the predicted and actual output is propagated back through the network to update the weights.

ANNs are designed to work with tabular data, where each column represents a feature

ANNs have a large number of parameters, which can lead to overfitting if the model is not properly regularized.

CNN Report



```
# inputs = tf.keras.Input(shape=(150, 150,1))
# x = Conv2D(filters=16, kernel_size=(3, 3), activation='relu')(inputs)
# x = MaxPool2D()(x)
# x = Conv2D(filters=32, kernel_size=(3, 3), activation='relu')(x)
# x = MaxPool2D()(x)
# x = GlobalAveragePooling2D()(x)
# x = Dense(128, activation='relu')(x)
# x = Dense(128, activation='relu')(x)
# outputs = Dense(1, activation='sigmoid')(x)
# model = tf.keras.Model(inputs=inputs, outputs=outputs)



# model = Sequential()
# ################################# CNN ######################################################
# model.add(Conv2D(filters = 12, kernel_size = (3,3),padding = 'Same',
#                  activation ='relu', input_shape = train_images.shape[1:]))
# model.add(MaxPool2D(pool_size=(2,2)))
# model.add(Conv2D(filters = 24, kernel_size = (3,3),padding = 'Same',
#                  activation ='relu'))
# model.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))
# ################################# Fully Connected ########################################
# model.add(Flatten())
# model.add(Dense(512, activation = "relu"))
# model.add(Dropout(0.1))
# model.add(Dense(test_images.shape[1], activation = "sigmoid"))


model = Sequential()
model.add(Conv2D(16, (3, 3), activation='relu', input_shape=(150, 150, 1)))
model.add(MaxPool2D((2, 2)))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPool2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPool2D((2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

We actually tried many cnn models before we settled on this one

The last one

This code defines a sequential model in Keras, a high-level API for building neural networks. The model consists of several layers that are added sequentially. Let's go through each line and understand what each layer does:

Creating a new instance of the Sequential class, which represents a linear stack of layers. This function is the best option if you have one data input and one data output it just flows from top to bottom.

We add a 2D Convolutional layer to the model. The layer has 16 filters, each with a size of 3x3. The activation function used is ReLU (Rectified Linear Unit). The input_shape parameter specifies the shape of the input data, which is a 3D tensor with dimensions (height=150, width=150, channels=1). This layer is typically used for extracting spatial features from images.

Adding a 2D max pooling layer to the model. The layer performs down-sampling by taking the maximum value within each 2x2 region of the input.

Max pooling is used to reduce the spatial dimensions of the data while preserving important features. It goes through images and condenses it down so it goes what is the max value of this region and it only returns that region.

Adding another Convolutional layer with 32 filters, followed by a max pooling layer, and then another Convolutional layer with 64 filters followed by a max pooling layer. These layers progressively extract more complex features from the input data.

Adding a flatten layer to the model. It reshapes the output from the previous layer into a flat 1D vector. This is required because the subsequent layers in the model are fully connected (dense) layers that expect input in vector form.

We add a fully connected (dense) layer to the model. The layer has 128 units, and the activation function used is ReLU. Dense layers are responsible for learning high-level representations of the data by connecting every neuron in the previous layer to every neuron in the current layer.

after that, add the final fully connected layer to the model. It has a single unit, which is used for binary classification. The activation function used is the sigmoid function, which maps the output to a value between 0 and 1, representing the probability of the input belonging to the positive class.

In summary, this code defines a sequential model with several convolutional layers for feature extraction, followed by fully connected layers for classification. The model takes in 2D input data, typically images, and produces a binary classification output.

In this code, the Adam optimizer is instantiated and configured with a learning rate of 0.001. Then, the model's compilation is performed, specifying the optimizer, loss function, and metrics to be used during training. Let's break it down:

First creating an instance of the Adam optimizer from the Keras library to update the weights and biases of the neural network during training. The learning_rate parameter determines the step size at which the optimizer adjusts the parameters. In this case, a learning rate of 0.001 is specified.

Compiling the model by configuring its learning process. The compile method of the model is called with several arguments:

The optimizer 'Adam': This specifies the optimizer to be used during training. In this case, the Adam optimizer is specified, which was instantiated earlier.

The loss 'binary_crossentropy': This sets the loss function to be used. The loss function is a measure of how well the model's predictions match the true labels during training. The 'binary_crossentropy' loss function is commonly used for binary classification problems.

The metrics ['accuracy']: This specifies the evaluation metric(s) to be used during training and testing. In this case, the 'accuracy' metric is used, which calculates the proportion of correctly predicted samples.

By calling a compiling function, the model is prepared for training. The optimizer, loss function, and metrics are configured, allowing the model to update its parameters based on the training data and evaluate its performance.

train_images and val_images are the training and validation data, respectively. These are the input datasets used for training and evaluating the model's performance.

The model will be trained for 100 epochs.

Callbacks is a list of callbacks that are executed during training at various stages. Callbacks allow you to perform certain actions during training, such as early stopping or adjusting the learning rate.

The keras function "EarlyStopping" is a callback that monitors a specified metric (in this case, 'val_loss', which is the validation loss) and stops training early if the metric does not improve after a certain number of epochs. The patience=5 parameter indicates that training will be stopped if there is no improvement in the validation loss for 5 consecutive epochs. The restore_best_weights=True parameter ensures that the best model weights based on the validation loss are restored at the end of training.

The keras function "ReduceLROnPlateau" is a callback that reduces the learning rate when a specified metric (in this case, 'val_loss') has stopped improving. The patience=10 parameter indicates that if there is no improvement in the validation loss for 10 consecutive epochs, the learning rate will be reduced.

The method starts the training process with the specified configurations. During training, the model will iterate over the training data for the specified number of epochs, compute the loss and metrics, and update the model's weights and biases based on the optimizer's algorithm. The validation data is used to monitor the model's performance on unseen data and provide insights into its generalization ability.

The training progress and relevant information, such as the loss and metrics for each epoch, are stored in the history object. This object can be used later to analyze the training results, visualize the learning curves, or make predictions with the trained

Then the trained model is evaluated on the test data to assess its performance. The evaluation results, including the test loss and accuracy, are then printed. Let's break it down:

The evaluate method we use "model.evaluate" is called on the trained model to evaluate its performance on the test data (test_images). and show that the progress updates should be displayed during the evaluation The evaluate method computes the specified metrics (loss and accuracy) on the provided test data and returns the results.

Finally we print the evaluation results. The test loss is formatted as a floating-point number with 5 decimal places, and the test accuracy is formatted as a percentage with 2 decimal places.

The first result corresponds to the test loss, which is the value of the loss function computed on the test data.

The second result corresponds to the test accuracy, which is the value of the accuracy metric computed on the test data.

By printing these values, you can assess how well the trained model performs on the unseen test data. The test loss indicates how well the model's predictions match the true labels, with lower values indicating better performance. The test accuracy represents the proportion of correctly predicted samples in the test data, with higher values indicating better accuracy

CNN VS ANN

Convolution Neural Networks (CNNs) and Artificial Neural Networks (ANNs) are both types of neural networks used in machine learning, but they have some key differences.

Input Data: ANNs are designed to work with tabular data, where each column represents a feature, while CNNs are suited for handling image and audio data.

Architecture: ANNs consist of an input layer, hidden layers, and an output layer. Each neuron in a layer is connected to every neuron in the previous layer. CNNs, on the other hand, use convolution layers, pooling layers, and fully connected layers. Convolution layers perform feature extraction by applying filters to the input data, while pooling layers reduce the size of the output from the convolution layers.

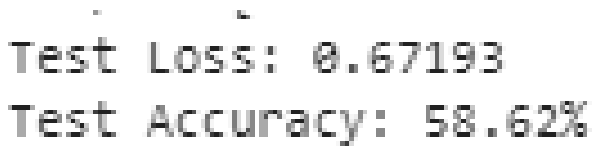
Parameters: ANNs have a large number of parameters, which can lead to overfitting if the model is not properly regularized. CNNs have fewer parameters due to the use of shared weights in convolution layers, which helps in reducing overfitting.

Training : ANNs are trained using back propagation, where the error between the predicted and actual output is propagated back through the network to update the weights. CNNs also use back propagation, but with an additional technique called gradient descent with momentum, which helps in achieving faster convergence.

Performance: CNNs have shown great performance in tasks such as image classification, object detection, and segmentation. ANNs are better suited for tabular data tasks such as regression and classification.

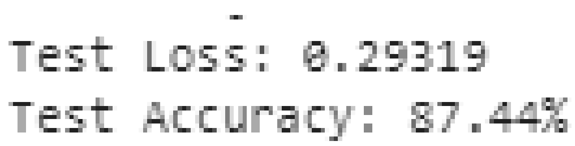
ACCURACY :

ANN Model accuracy is 58.62%



```
Test Loss: 0.67193
Test Accuracy: 58.62%
```

CNN Model accuracy is 87.44%



```
Test Loss: 0.29319
Test Accuracy: 87.44%
```

CNN Model accuracy is far better than ANN model

F score , recall, precision

F1 score is the harmonic mean of precision and recall. It is a measure of the models accuracy that considers both precision and recall. The formula for calculating the F1 score is:

$$F1\,score = 2 \cdot \frac{Precision * Recall}{Precision + Recall}$$

The F1 score ranges from 0 to 1, with 1 being the best possible score.

Recall is the ratio of true positives to the total number of actual positives. It is a measure of the models ability to correctly identify positive cases. The formula for calculating recall is:

$$\text{Recall} = \frac{TP}{TP + FN}$$

Recall ranges from 0 to 1, with 1 being the best possible score.

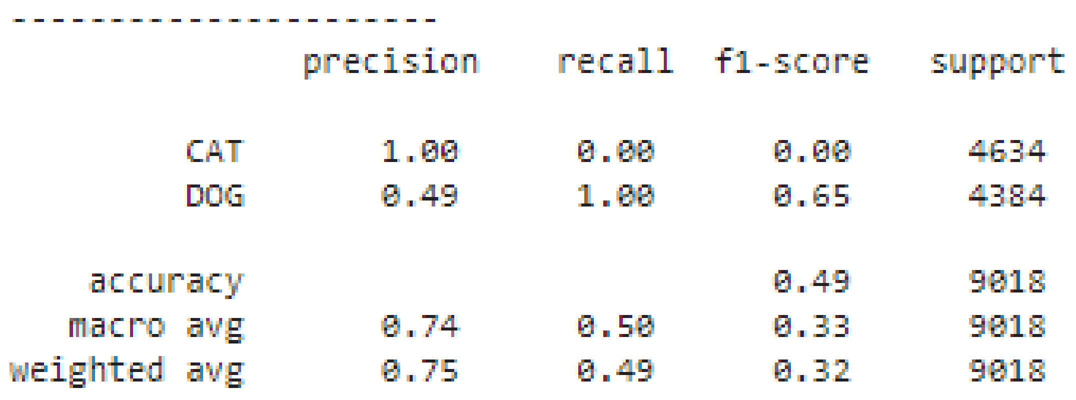
Precision: Precision is the ratio of true positives to the total number of predicted positives. It is a measure of the models ability to correctly predict. The formula for calculating precision is:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Precision ranges from 0 to 1, with 1 being the best possible score.

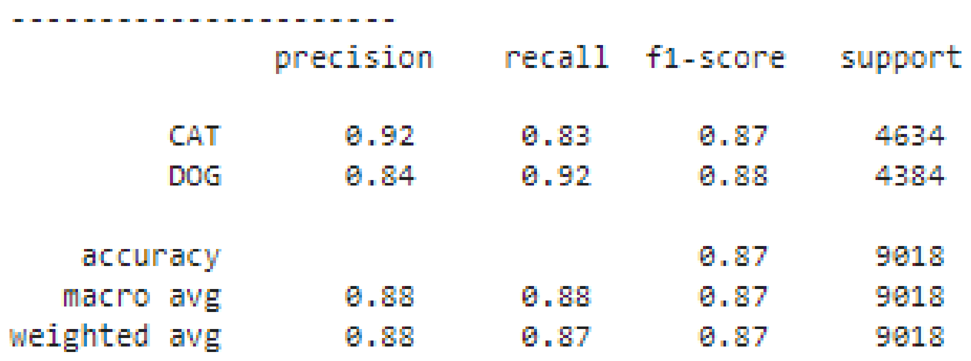
ANN Model F score , recall, precision values

ANN model F score , recall, precision



```
----------------------
              precision    recall  f1-score   support

         CAT       1.00      0.00      0.00      4634
         DOG       0.49      1.00      0.65      4384

    accuracy                           0.49      9018
   macro avg       0.74      0.50      0.33      9018
weighted avg       0.75      0.49      0.32      9018
```

CNN model F score , recall, precision



```
----------------------
              precision    recall  f1-score   support

         CAT       0.92      0.83      0.87      4634
         DOG       0.84      0.92      0.88      4384

    accuracy                           0.87      9018
   macro avg       0.88      0.88      0.87      9018
weighted avg       0.88      0.87      0.87      9018
```

The cnn model have a higher F score , accuracy value which makes it better , Because CNNs are specialized for working with image and audio data, while ANNs are better suited for tabular data. CNNs have a unique architecture that includes convolution and pooling layers, which help in reducing the number of parameters and achieving better performance on image-related tasks. ANNs, on the other hand, have a simpler architecture and are better suited for tasks such as regression and classification.

CONCLUSION

In conclusion, our machine learning project on classification of dogs and cats has been successful in building an accurate image classification model.

We evaluated the performance of our model using various metrics such as Accuracy, F1 score, recall, specificity, and precision. Our model achieved high accuracy in classifying images of cats and dogs, which can have numerous applications such as identifying animals in wildlife conservation and identifying lost pets in animal shelters.