Université Mohamed Premier
École Nationale des Sciences Appliquées
-Oujda-

Machine Learning Project Report

Field : Sécurité Informatique et Cybersécurité

Implementing a simple CNN model for "cat versus dog" classification task

Presented on : 10/12/2024

Prepared by :
Mr. Achraf ESSATAB
Ms. Meryem SABBAR

Supervised by :
Prof. Jawad OUBAHA

Academic year
2024/2025

Contents

1 Acknowledgements

I want to thank my colleague Meryem for her collaboration, our professor Mr. Jawad Oubaha for suggesting the subject and the source of inspiration of this project Mr. Harrison Kinsley, the owner of pythonprogramming.net. Harrison works impacted strongly the flow of this project and some parts of code implementation were directly copied from his blog.

2 Introduction

This project implements a simple CNN model using Keras framework. Using only three epochs on a dataset[1] of 25 000 pictures, we achieved 73 percent validation accuracy rate [2]. The model was able to distinguish correctly between dogs and cats in the "cat versus dog" classification task when used in a testing phase, which may indicate a generalized learning.

[1] www.kaggle.com/c/dogs-vs-cats

[2] the validation subset is 10 percent of the whole dataset

3 Loading and preprocessing data

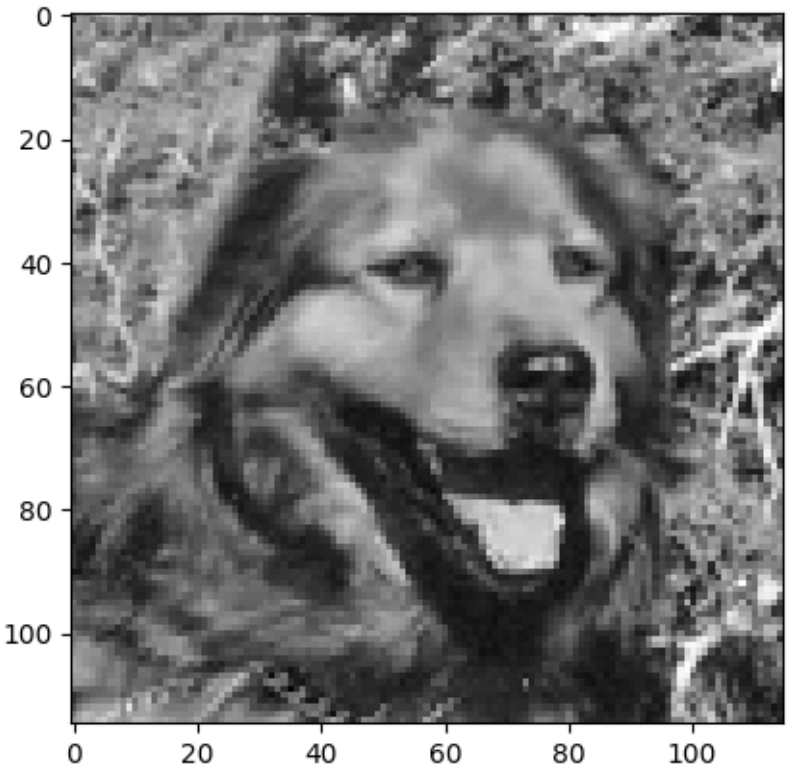
3.1 Balancing the dataset

In a classification task of two outputs we train the CNN model on two kinds of objects. The number of samples from each class must be the same. Hence, we used 12501 pictures of dogs and 12501 pictures of cats. Otherwise, the balancing step means: we configure the machine learning framework we are working on to deal with the difference of the size of each class. This is an important step to avoid that the model being biased to one class more than the other.

3.2 Converting the images to one color channel

While color may be a differentiating factor to decide between a picture of, for example, a dog and a reptile. It isn't a differentiating factor to distinguish between dogs and cats. Hence, our data were preprocessed to be in a grayscale, using only one color channel. Going with one color channel will also reduce the size of each sample and put less overhead on the computer during the training phase, so we may avoid potential crashes. This is done using the imread method of the cv2 module. The output of the code below shows how this preprocessing step affects one sample of the data.

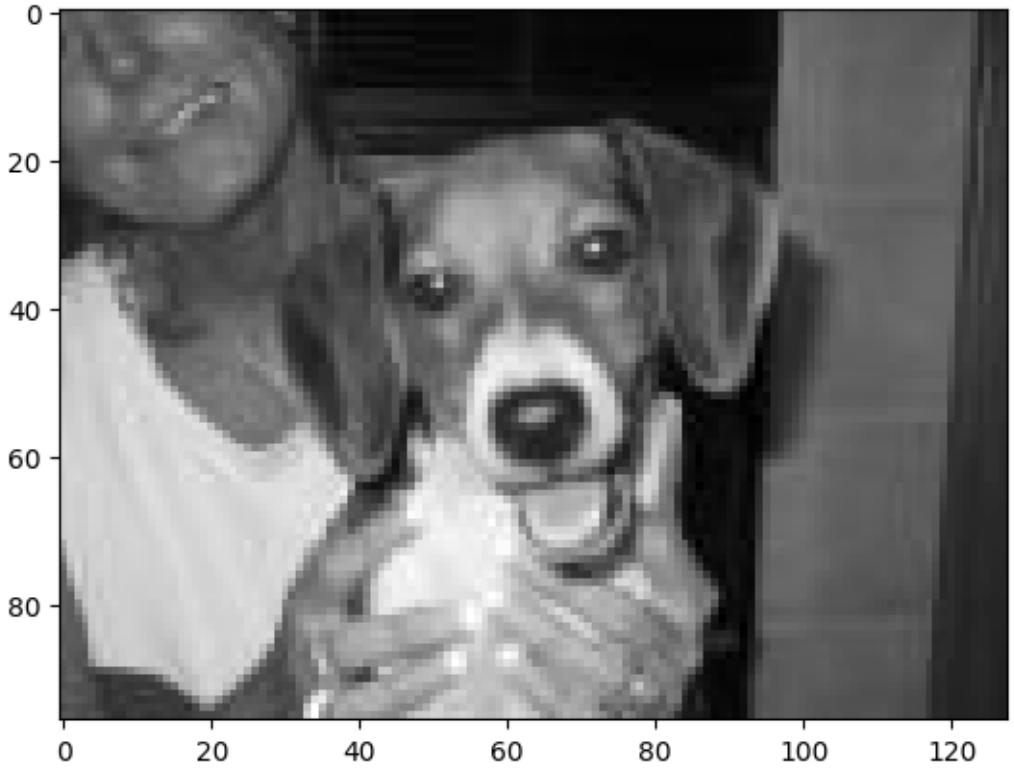
```
import numpy as np
import matplotlib.pyplot as plt
import os
import cv2
from tqdm import tqdm
DIR="./train"
classes=["dog","cat"]
#take all the data then shuffle
for animal in classes:
    path=os.path.join(DIR, animal)
    for img in os.listdir(path):
        img_array=cv2.imread(os.path.join(path,img), cv2.IMREAD_GRAYSCALE)
        plt.imshow(img_array,cmap='gray')
        plt.show()
        break
    break
```

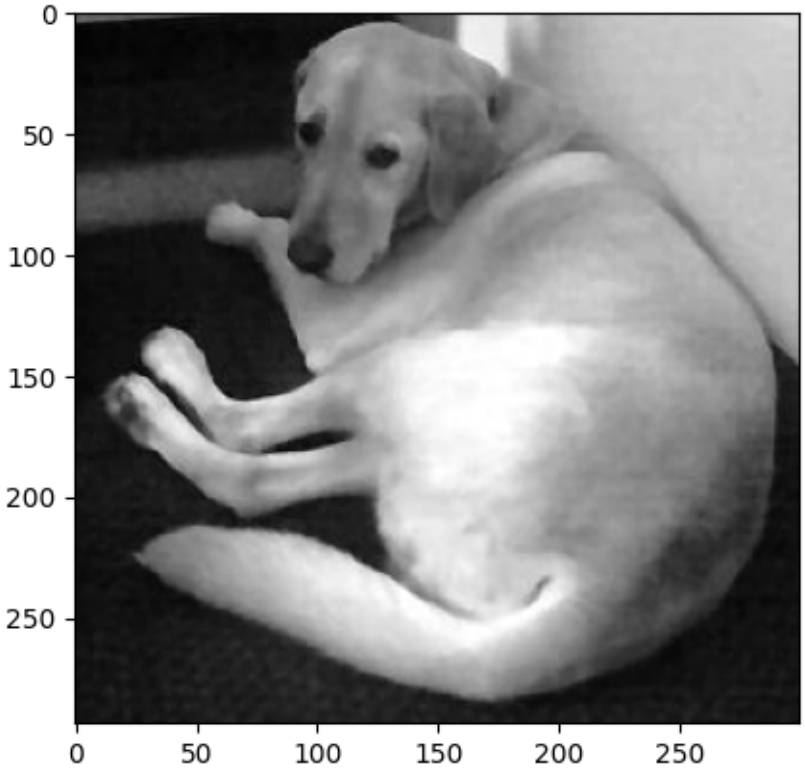


3.3 Using the same picture size

The kaggle dataset comes with pictures of different shapes, while having an efficient training process requires that each picture is of the same size. The size in this context means nxm number of pixels, while n for the height and m for the image width. The following code is to discover the size of two different samples in the dataset.

```
test_images=['69.jpg','70.jpg']
for img in test_images:
    img_array=cv2.imread(os.path.join(path,img), cv2.IMREAD_GRAYSCALE)
    plt.imshow(img_array,cmap='gray')
    plt.show()
```

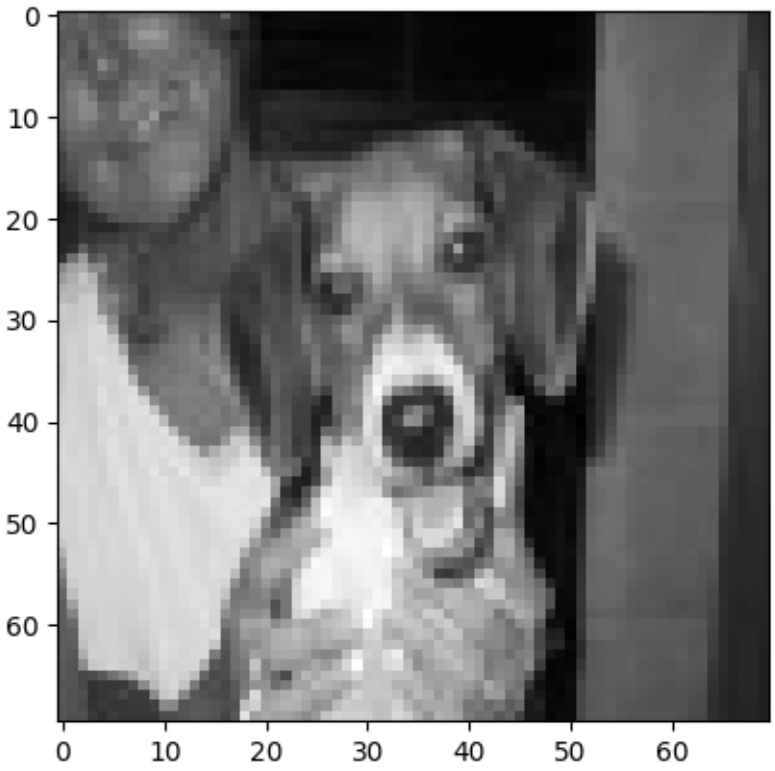


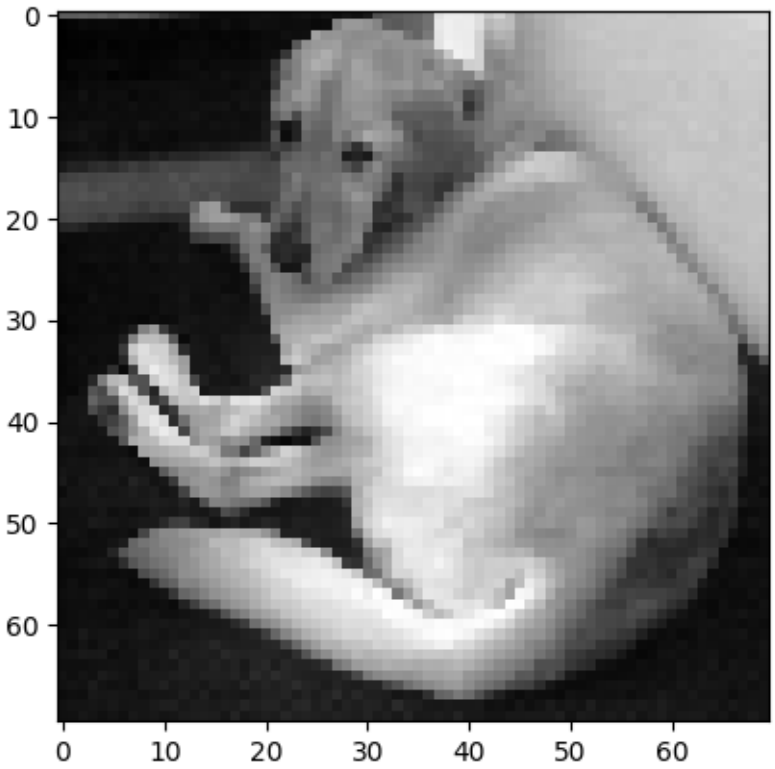


To give each picture the same shape, we used the resize method of the class cv2 to convert

images to 70x70 pixels as in the code below:

```
IMG_SIZE=70
test_images=['69.jpg','70.jpg']
for img in test_images:
    img_array=cv2.imread(os.path.join(path,img), cv2.IMREAD_GRAYSCALE)
    resized_img=cv2.resize(img_array, (IMG_SIZE, IMG_SIZE))
    plt.imshow(resized_img,cmap='gray')
    plt.show()
```





3.4 Preparing Keras-friendly data structures

Keras framework, in this classification task, expects to work on two numpy arrays. X array that holds images in grayscale, it is a tensor of 70x70 matrices with each element is the value of the corresponding pixel in grayscale mode. Also, **we need the samples to be randomly picked** before passing them to the training phase. This is important to avoid a **biased training**. To comply with these specifications, the workflow of the data preprocessing phase is as the following:

- Saving images and lables as part **one list of data size elements**. Each element of the list is a list of two elements, image and lable of a sample.
- **Shuffling** the list on the sample level. This is done using the random module of python.
- Finally, **saving this intermediate new data structure** to avoid the data-preprocessing phase in future implementations of the model.

We used pickel module to save and load a dump of the data. After loading the data it must be converted to numpy arrays to comply with Keras requirements. The code below is the entire preprocessing phase.

```
import numpy as np
import matplotlib.pyplot as plt
import os
import cv2
from tqdm import tqdm
import random
import pickle
```

```
DIR="./Kaggle_PetImages"
classes=["Dog","Cat"]
#take all the data then shuffle
training_data=[]
X = [] #image
Y = [] #label
IMG_SIZE=70
def create_training_data():
    for category in classes:  # do dogs and cats

        path = os.path.join(DIR,category)  # create path to dogs and cats
        class_num = classes.index(category)  # get the classification  (0 or␣
↪a 1). 0=dog 1=cat
        for img in tqdm(os.listdir(path)):  # iterate over each image per␣
↪dogs and cats
            try:
                img_array = cv2.imread(os.path.join(path,img) ,cv2.
↪IMREAD_GRAYSCALE)  # convert to gray
                new_array = cv2.resize(img_array, (IMG_SIZE, IMG_SIZE))  #␣
↪resize to normalize data size
                training_data.append([new_array, class_num])
            except Exception as e:  # in the interest in keeping the output␣
↪clean...
                pass


create_training_data()

#shuffling the data
random.shuffle(training_data)
#Unpacking labels and images in separate arrays
for features,label in training_data:
    X.append(features)
    Y.append(label)

#Saving data
pickle_out = open("X.pickle","wb")
pickle.dump(X, pickle_out)
pickle_out.close()

pickle_out = open("Y.pickle","wb")
pickle.dump(Y, pickle_out)
pickle_out.close()
```

```
Corrupt JPEG data: 399 extraneous bytes before marker 0xd90:11<02:37, 74.17it/s]
Corrupt JPEG data: 254 extraneous bytes before marker 0xd9
Corrupt JPEG data: 65 extraneous bytes before marker 0xd9
Warning: unknown JFIF revision number 0.00
Corrupt JPEG data: 162 extraneous bytes before marker 0xd9
Corrupt JPEG data: 2230 extraneous bytes before marker 0xd9:11<01:28, 82.11it/s]
```

```
Corrupt JPEG data: 226 extraneous bytes before marker 0xd9
Corrupt JPEG data: 1403 extraneous bytes before marker 0xd9

Corrupt JPEG data: 99 extraneous bytes before marker 0xd9
Corrupt JPEG data: 239 extraneous bytes before marker 0xd9
Corrupt JPEG data: 214 extraneous bytes before marker 0xd90:14<02:54, 66.95it/s]
Corrupt JPEG data: 1153 extraneous bytes before marker 0xd9:20<02:17, 57.22it/s]
Corrupt JPEG data: 128 extraneous bytes before marker 0xd9

100%|-----------------------------------| 12501/12501 [03:39<00:00, 57.01it/s]
```

At this point, we completed the data preprocessing phase on **25000 samples** and we are ready to implement the convolutional neural network before training it using Keras framework.

4 Setting the convolutional neural network using Keras framework

4.1 Network architecture

The basic structure of a CNN follows a sequential pipeline: Convolution - Pooling - Convolution - Pooling - Fully Connected Layer - Output.

4.1.1 Convolution:

The convolutional operation is the cornerstone of CNNs. It involves applying a set of filters (or kernels) to the original input data, such as an image, to generate feature maps. These feature maps highlight key patterns or features (e.g., edges, textures) in the data, making the CNN adept at recognizing visual elements.

4.1.2 Pooling:

Pooling, also known as subsampling or downsampling, reduces the spatial dimensions of the feature maps, which helps to minimize computational complexity and prevent overfitting. The most common pooling method is max-pooling, where a small region (e.g., 2x2) is scanned, and only the maximum value within that region is retained. This operation preserves the most significant features while discarding less important details.

4.1.3 Fully connected layer:

These layers resemble traditional neural networks, where each neuron is connected to every neuron in the preceding layer. Fully connected layers aggregate and interpret the high-level features extracted by the convolutional and pooling layers, leading to the final classification or prediction.

4.1.4 Convolutional layers:

Unlike the fully connected layers, convolutional layers do not connect every neuron to every other neuron. Instead, they focus on local connections, which allows CNNs to process spatial hierarchies and detect patterns efficiently. Figure 1 depicts the used network architecture:

4.2 Configuring the network

Using Keras APIs we can eventually implement whatever CNN network architecture. This project implements a sequential network with the following characteristics:

- Two identical layers that relate directly with the input:
  - Each starts with a two-dimensional convolutional layer of 64 neurons, and each characterized by a 3 by 3 kernel.
  - Following the convolutional layer is an activation layer that uses the Relu function.
  - A MaxPooling layer that outputs a feature map that is reduced in size two the twin next layer.

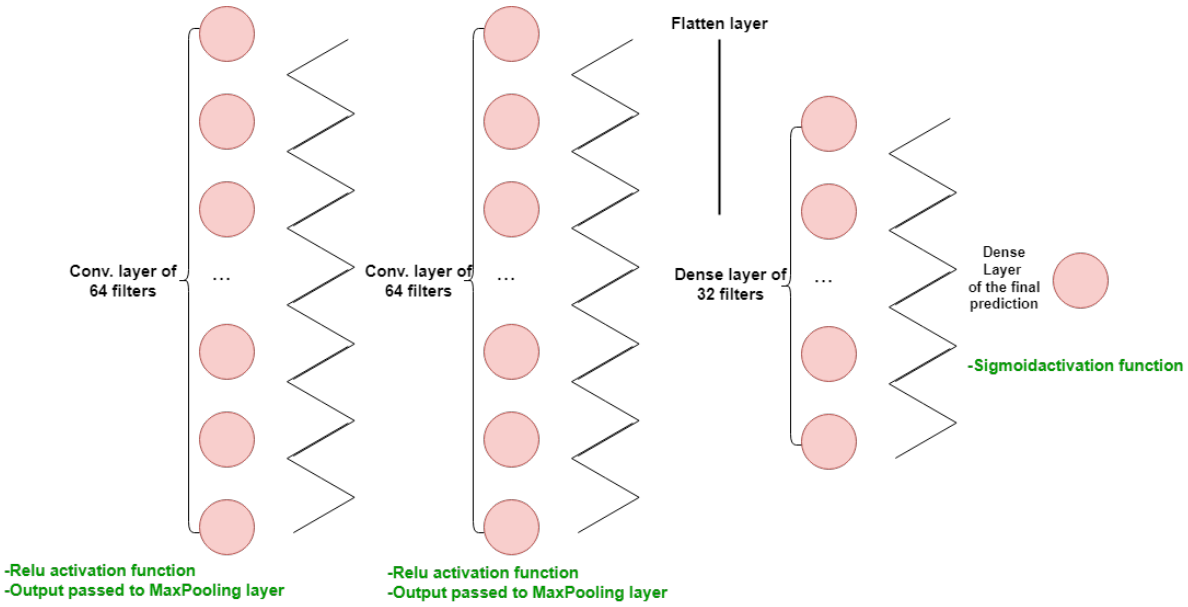


Figure 1: CNN architecture

- The output of the second layer of the two identical layers is fed to a dense layer of 32 nodes after it has been flattened by a flatten layer.
- A second dense layer with only one node uses sigmoid activation function ends up the network.

These requirements were achieved with the following python code:

```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D

import pickle

pickle_in = open("X.pickle","rb")
X = pickle.load(pickle_in)

pickle_in = open("Y.pickle","rb")
Y = pickle.load(pickle_in)

X = X/255.0

model = Sequential()

model.add(Conv2D(64, (3, 3), input_shape=X.shape[1:]))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3)))
```

```
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())  # this converts our 3D feature maps to 1D feature␣
↪vectors

model.add(Dense(32))

model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

This initial compilation of the model was based on three fundamental parameters:

- The loss parameter specifies the function used to evaluate how well the model's predictions match the actual target values. binary_crossentropy is ideal for binary classification problems, where the target output is either 0 or 1. This loss function calculates the cross-entropy loss between the predicted probability and the actual class label, making it effective for guiding the model during training to improve its predictions.
- The optimizer parameter determines the optimization algorithm used to adjust the model's weights and biases during training. The adam optimizer, short for Adaptive Moment Estimation, combines the benefits of two other popular optimizers—RMSprop and SGD with momentum. It adapts the learning rate for each parameter dynamically, making it efficient and widely used for deep learning models due to its robust performance across various tasks.
- The metrics parameter defines the additional evaluation metrics used to monitor the model's performance during training and testing. By specifying accuracy, the model will calculate the proportion of correct predictions out of the total predictions. This metric is intuitive and widely used, especially in classification tasks, to provide a straightforward measure of the model's success in predicting the correct labels.

4.3 Training the model

To train the model, all we need is to call a method called fit on the model we defined previously with the preffered configurations. The line below starts the training of the model while specifying 10 percent of the data to be validation data, deviding the dataset on batches of 32 sample. The training in this confiiguration will be in the 3 epochs. An epoch is a pass through all the dataset while adjusting the wieghts of the network:

```
model.fit(X, Y, batch_size=32, epochs=3, validation_split=0.1)
```

The result of this initial training is 75 percent validation accuracy rate. We saved the initially trained model using the method in the below code:

```
model.save('CNN.model')
```

Whenever we want to load the model we do as in the code below:

```
model = tf.keras.models.load_model("CNN.model")
```

4.4 Using the model

After loading a trained saved model, we fed the sample we want the model to predict on in an appropriate format[3] to the model using predict method. The following code make a prediction on picture never exposed to the model.

```
import cv2
import tensorflow as tf

CATEGORIES = ["Dog", "Cat"]  # will use this to convert prediction num to␣
↪string value


def prepare(filepath):
    IMG_SIZE = 70
    img_array = cv2.imread(filepath, cv2.IMREAD_GRAYSCALE)  # read in the␣
↪image, convert to grayscale
    new_array = cv2.resize(img_array, (IMG_SIZE, IMG_SIZE))  # resize image␣
↪to match model's expected sizing
    return new_array.reshape(-1, IMG_SIZE, IMG_SIZE, 1)  # return the image␣
↪with shaping that TF wants.

#Loading the model
model = tf.keras.models.load_model("CNN.model")
```

```
#Magic
prediction = model.predict([prepare('photo.jpg')])
print(CATEGORIES[int(prediction[0][0])])
```

```
1/1 [==============================] - 0s 42ms/step
Dog
```

[3] In this case, 70 by 70 pixels picture in grayscale mode, presented as a numpy array.

5 Conclusion

This project represents an initial implementation of a convolutional neural network (CNN) for the binary classification task of distinguishing between cats and dogs. While this serves as a foundational step, the real expertise in this field lies in the meticulous optimization of the model—fine-tuning hyperparameters, experimenting with advanced architectures, and leveraging techniques to improve performance and generalization. This optimization process is both an art and a science and could be the subject of a future project.

Beyond this specific task, CNNs have transformative applications across various domains. In self-driving cars[4], they are pivotal for tasks like object detection and lane recognition, ensuring safety and navigation. Similarly, in medicine, CNNs play a crucial role in diagnostics, such as identifying skin cancer[5] from medical images with high accuracy. These applications highlight the immense potential of CNNs in solving complex, real-world problems

[4] https://www.thinkautonomous.ai/blog/lane-detection/
[5] https://news.mit.edu/2021/artificial-intelligence-tool-can-help-detect-melanoma-0402