UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE POLITICHE,
ECONOMICHE E SOCIALI

CORSO DI LAUREA MAGISTRALE IN

Data Science and Economics

Image recognition with CNN - Binary classification of chiuaua vs Muffin

Student: Alessia Leo Folliero

Student number: 08399A

E-mail: alessia.leofolliero@studenti.unimi.it

1 Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

Contents

2 Introduction

This project was part of the machine learning examination. The aim of this project was to conduct a binary image classification of chihuahua vs muffins thanks to the use of Convolutional neural networks.
The requirements for this project were:

- Image transformation into RGB or Greyscale
- Data normalization
- use at least three model architecture
- Hyperparameter tuning
- K-fold cross validation

2.1 Dataset

The dataset proposed for this project has been taken from `https://www.kaggle.com/datasets/samuelcortinhas/muffin-vs-chihuahua-image-classification`. This dataset contains over 6000 images taken from google images, divided in test (approximately 80% of the whole dataset) and test (approximately 20 % of the whole dataset). Here some examples of the images contained in the dataset.

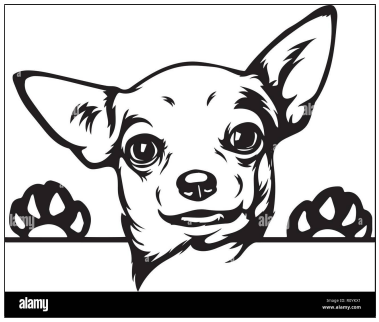

Figure 1: Chihuahua

Figure 2: Muffin

2.1.1 Data loading and pre-processing

Data were imported in Colab thanks to the kaggle Api. The images were already imported into train and test folder and within each folder there were two subfolders named chihuahua and muffin. The images were all in jpg format

2.1.2 Greyscale images

In on of the Colab files you are going to find on my Github (Neural Network muffins-chiuaua- greyscale.ipynb), I encoded the images as greyscale and resized them into

100,100. In order to normalize and split the training data into train and validation set I used the command ImageDataGenerator from keras.

2.1.3 RGB

In the Colab file named: Neural Network muffins-chiuaua- RGB.ipynb I instead encoded the images as RGB (Red, Green, Blue) and I resized them as 224,224. Also in this file I used ImageDataGenerator to normalize the images.

3 CCN theory

Neural Networks are algorithms modeled after our brains that try to simulate human biological neural network.

Convolutional Neural Network are a special kind of Neural Network that has at least one convolution layer. We usually use CNN because they are helpful if we want to obtain local information (features), to reduce the overall complexity of the model and because it needs just few preprocessing of the images.

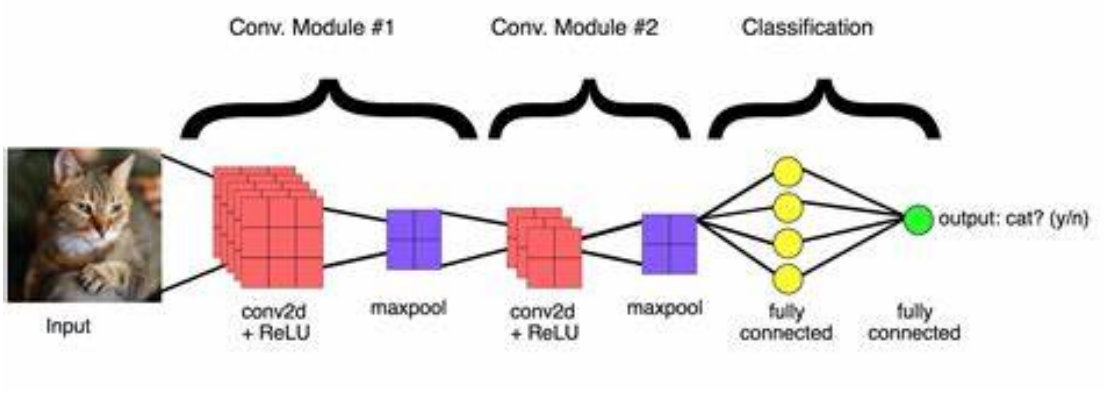

Figure 3: Example of CCN.

A CCN is composed by three main layers: Convolution, pooling and fully connected layer.

- **Convolutional Layer**: The filter is applied to our input to extract its features. A filter is applied to the image multiple times and create a feature map that helps to classify the image.
- **Pooling layer**: The aim of this layer is to reduce the dimensions of the feature map which helps in preserving the important information of the input images and reduce the computation time. It works similarly to the convolution layer but the function that is applied to the kernel and image is not linear. The most common pooling functions are: Max pooling or average pooling.
- **fully connected layer**: This layers connects the information extracted from previous layers and classifies the input with the corresponding label.

In between the second step (pooling) and the third one there is the **Flattening** step.

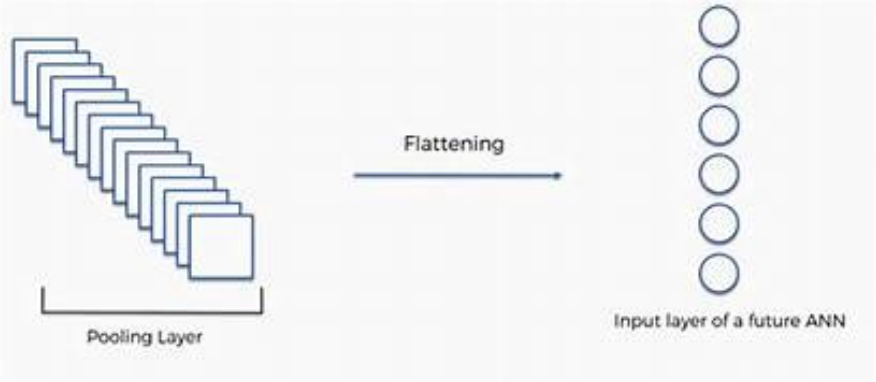

Figure 4: Example of the Flattening step.

In this step we convert data into a one dimensional array.The reason why we do that

is because in this way we are able to give it as an input to the Network in the following step.

In order to get the output, after we calculated the weighted sum of all the inputs and weights of the connection we need to add bias to this sum, since image recognition in not a linear operation. In order to introduce this non-linearity we use activation functions. There are serveral type of activation function i.e Sigmoid (Binary classification),ReLu and LReLu. Apart from those layer some other could be added to the CCN such as:

- **Dropout Layer**: In this layer a percentage p of nodes of the neural network is being dropped. This allow to have a new network architecture thanks to which the risk of over fitting is reduced.
- **BatchNormalization**: Is a normalization technique that normalize the output of the previous layer. Thanks to this technique learning becomes more efficient and it can be used as a regularization to avoid overfitting.

4 Project implementation

In all the architectures that you are going to find in my project I followed (more or less) the same scheme:

- Base model
- Improvements (Batchnormalization, Convolution Layers, Dropout ...)
- Data augmentation
- Hyperparameter tuning
- Kfold CV

4.1 Baseline model

This first model has as input images encoded as greyscale and resized as 100,100. I started this project by introducing a very simple architecture. The first architecture is the following:

```
model_1=Sequential(
    [Conv2D(32,(3,3),activation='relu',input_shape=(100,100,1)),
     MaxPool2D((2,2)),
     Conv2D(64,(3,3),activation='relu'),
     MaxPool2D((2,2)),
     Flatten(),
     Dense(128,activation='relu'),
     Dense(1,activation='sigmoid')]
)
```

Figure 5: First Model

After I compiled the model with binary crossentropy as loss, Adam as optimizer and accuracy as metrics. After training the model for 40 epochs with 32 as batch size this is the plot of the loss and accuracy for training and validation.

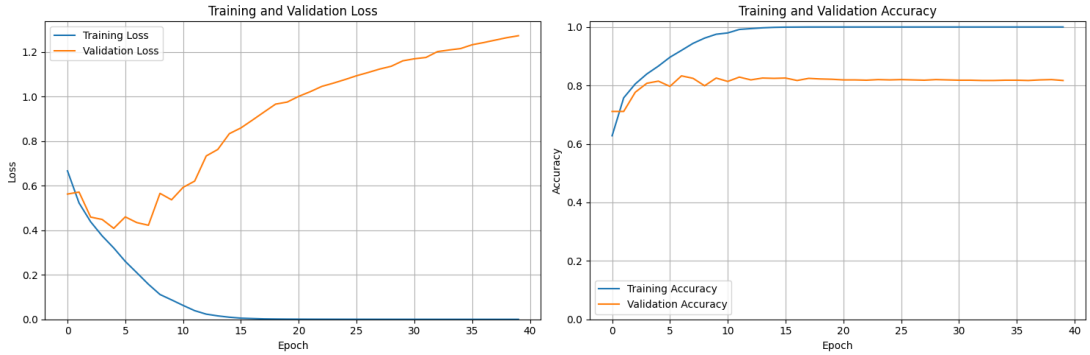

Figure 6: First Model

From the plot it can be seen that this first architecture performs poorly with our task. In fact from the model evaluation on the test set we get as result: 0.82 as accuracy and 1.29 as loss.

4.1.1 Improvements

- The first improvement that I made to this model was adding a convolution and pooling layer. This improvement did not workout, in fact, in the following plot you can observe that the validation loss is still high and the training loss goes down quickly which means that we have overfitting.

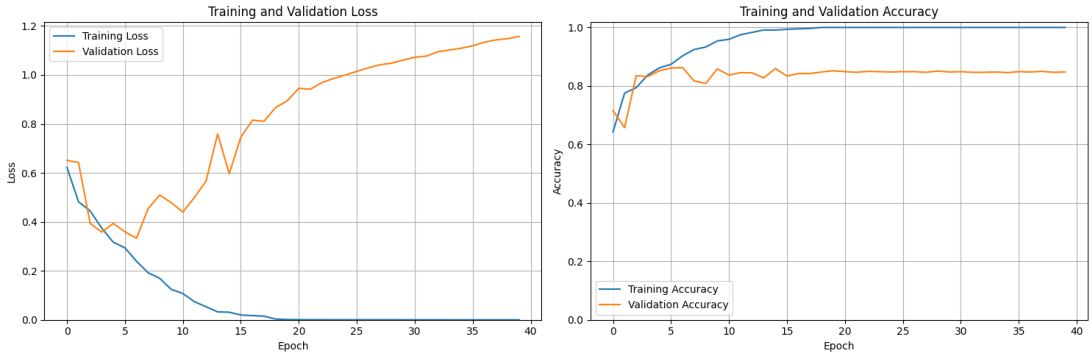

Figure 7: First Model improvement 1

- The further feature that I did add to this model was adding the strides equal to (2,2) to the first Maxpool2D layer and padding=’same’ in all the Convolution and pooling layers. The results, also in this case, did not improve.

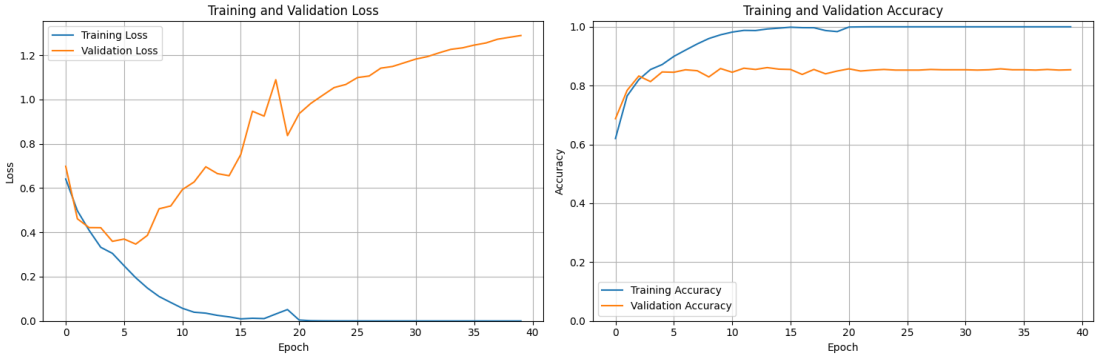

Figure 8: First Model improvement 2

This was my first attempt trying to tackle task and the overall conclusion is that this architecture is not working well on this kind of data, it is too simple and the results show high overfitting.

4.2 Second Architecture

Also for this second architecture the images that I gave as input were encoded as greyscale, resized as 100,100 and normalized (and splitted into train and validation) thanks to the use of ImageDataGenerator. This second Architecture is more complex than the first one, with three convolutional layers and dropout layers.

```
#defining model
def My_Second_Cnn():
  model_2=Sequential()

  #adding convolution layer
  model_2.add(Conv2D(32,kernel_size=(3,3),activation='relu',input_shape=(100,100,1)))
  #adding pooling layer
  model_2.add(MaxPool2D(pool_size=(2,2)))
  #Dropoutlayer
  model_2.add(Dropout(0.05))


  ###Second Convolutional Layer
  model_2.add(Conv2D(64,kernel_size=(3,3),activation='relu'))
  model_2.add(MaxPool2D(pool_size=(2,2)))
  model_2.add(Dropout(0.05))

  ###Third Convolutional Layer
  model_2.add(Conv2D(128,kernel_size=(3,3),activation='relu'))
  model_2.add(MaxPool2D(pool_size=(2,2)))
  model_2.add(Dropout(0.05))

  #adding fully connected layer
  model_2.add(Flatten())
  model_2.add(Dense(128,activation='relu'))
  model_2.add(Dropout(0.05))

  #adding output layer
  model_2.add(Dense(1,activation='sigmoid'))

  #compiling the model
  model_2.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])

  return model_2
```

Figure 9: Second Model

The model trainig was performed with batchsize=32 and 40 epochs. The following figure report the results.

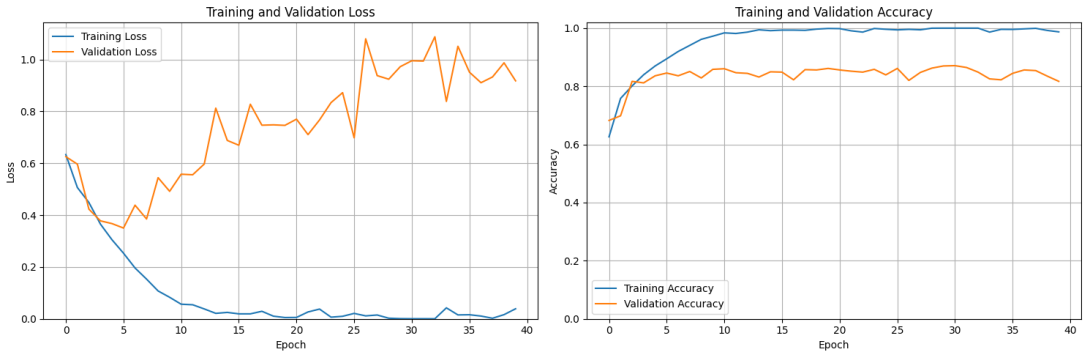

Figure 10: Accuracy and loss for training and validation

As we can already tell the model is not performing well, the plots show overfitting. In fact from the model evaluation on the test set we get as result: 0.83 as accuracy and 0.91 as loss.

4.2.1 Improvements

- The first modification of the second architecture was adding strides=(2,2) to the fisrt pooling layer. This model showed slightly improvement from the previous implementation.
- A further improvement was done by adding Batchnormalization with the aim of tackling overfitting.

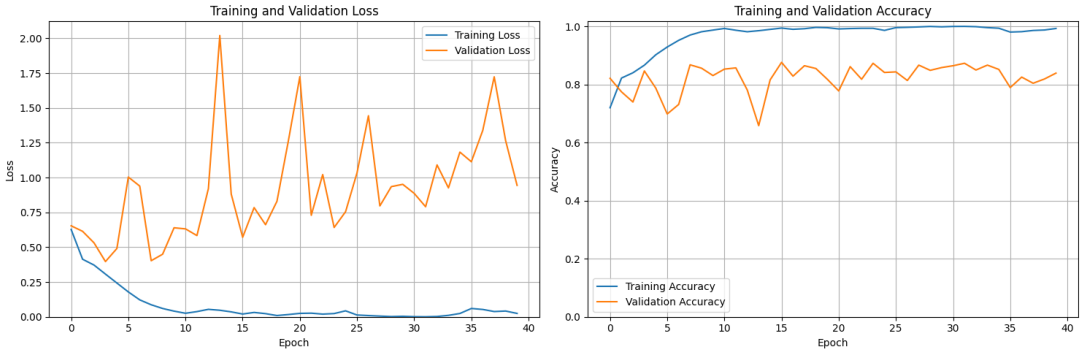

Figure 11: Secon Model improvement 1

As we can deduce from the plot this did not help improve the architecture and after the model evaluation we obtained worse results for both accuracy and loss.

- In the third architecture I introduced thanks to ImageDataGenerator some data augmentation (zoom) to the train and validation set. I did train the model with and without batchnormalization and I obtained better results without BatchNormalization and those are the results I am going to show you. If you are interested in checking also the other result go to the jupyter folder named Neural Network muffins-chiuaua- greyscale.ipynb and check for Second_model_3.

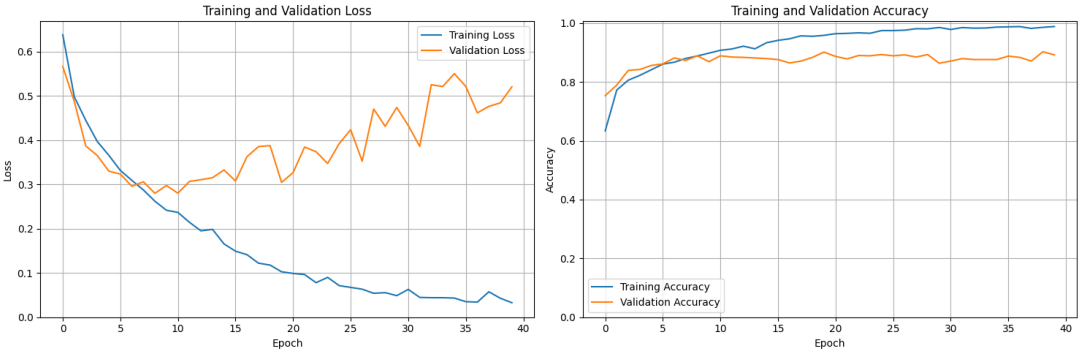

Figure 12: Second Model improvement 2

From the figure 12 we can see that we had a slight improvment but we still have high overfitting and high error.

- This third improvement involves the use of hyperparameter tuning. In order to perform the hyperparameter tuning I used keras library with the RandomSearch. Since this process took many trials and steps I am going to summarize the process and show you just the final result.
  - Hyperparameter tuning of convolutional filters, dense units and dropout rate (just one for all the dropout layers)
  - Using the tuned convolutional filters and dense units I used this architecture to tune the dropout rate for all the dropout layers
  - Tuning of the learning rate
  - Tuning of the kernel size for the convolutional layers
  - I also decided to optimize the pool feature in the pooling layers, but this gave worse result, so I stopped with the result obtained in the previous step.

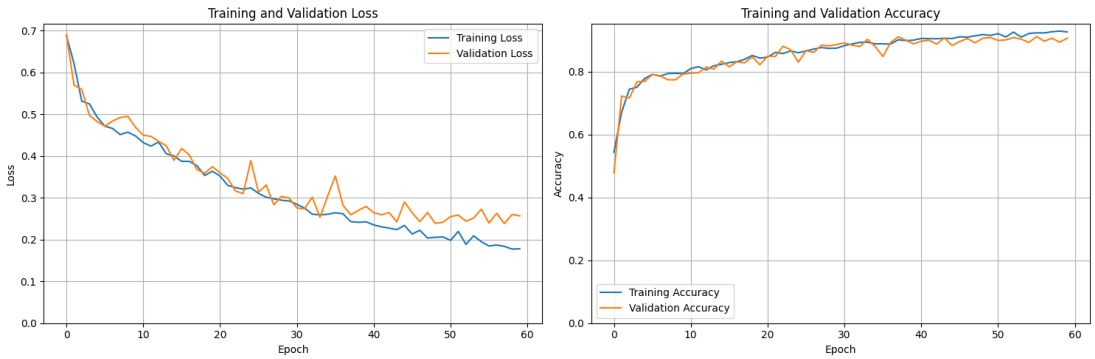

Figure 13: Second Model improvement 3 (Hyperparameter tuning)

This plot shows that this architecture is now performing well with our data and in

fact with the model evaluation we obtain 0.85 as accuracy and 0.33 as loss. The loss is still high and the accuracy could be improved.

From this second Architecture and it's improvements we finally obtain good results, but the loss is still high and there is still a little bit of overfitting starting from epoch 30.

4.3 Third Architecture

The input images of this architecture were the same as the other architecture but I decided to train the model with the augmented data. This model was deeper than the previous one, with one added convolution layer.

```
#defining model
def My_Third_Cnn():
  model_3=Sequential()

  #adding convolution layer
  model_3.add(Conv2D(32,kernel_size=(3,3),activation='relu',input_shape=(100,100,1)))
  #adding pooling layer
  model_3.add(MaxPool2D(pool_size=(2,2)))
  #Dropoutlayer
  model_3.add(Dropout(0.05))


  ###Second Convolutional Layer
  model_3.add(Conv2D(64,kernel_size=(3,3),activation='relu'))
  model_3.add(MaxPool2D(pool_size=(2,2)))
  model_3.add(Dropout(0.05))

  ###Third Convolutional Layer
  model_3.add(Conv2D(128,kernel_size=(3,3),activation='relu'))
  model_3.add(MaxPool2D(pool_size=(2,2)))
  model_3.add(Dropout(0.05))

  ###Fourth Convolutional Layer
  model_3.add(Conv2D(256,kernel_size=(3,3),activation='relu'))
  model_3.add(MaxPool2D(pool_size=(2,2)))
  model_3.add(Dropout(0.05))

  #adding fully connected layer
  model_3.add(Flatten())
  model_3.add(Dense(128,activation='relu'))
  model_3.add(Dropout(0.05))

  #adding output layer
  model_3.add(Dense(1,activation='sigmoid'))

  #compiling the model
  model_3.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])

  return model_3
```

Figure 14: Third Model

Also this model was trained on 40 epochs and with batch size equal to 32.

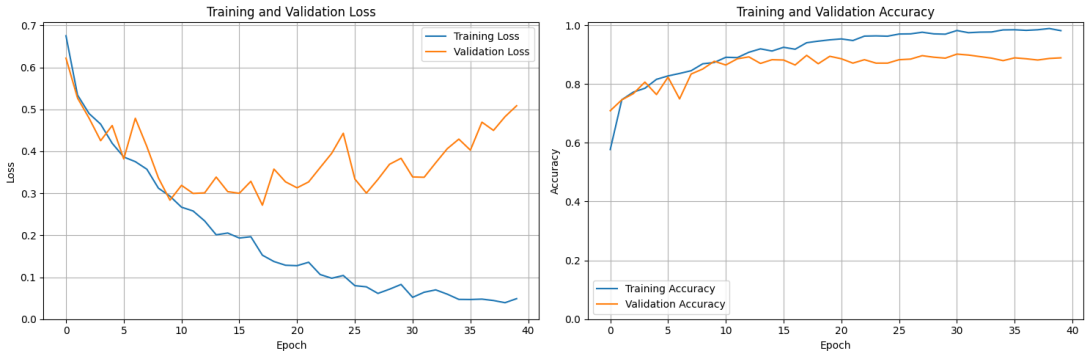

Figure 15: Accuracy and loss for training and validation

From the plot we can see that there is overfitting, probably for this type of data dropout of 0.05 is too low.

4.3.1 Improvements

Also in this case the following steps will be summarized:

- First improvement: Convolution filters and dropout rate tuning
- Second improvement: Kernel and learning rate tuning

After the tuning of the previously listed hyperparameters the plot of the loss and accuracy for the training and validation loss is the following:

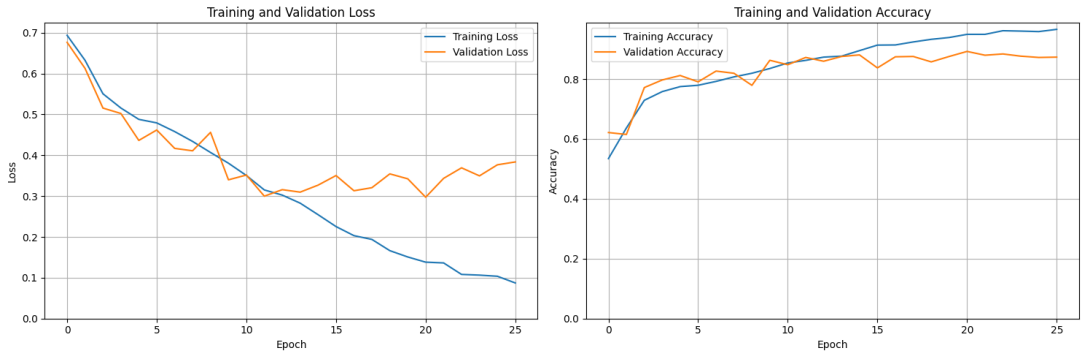

Figure 16: Third model improvements after tuning

The plot shows that the model training is stopped at the 25th epoch and the reason why the training process stops there is because I added to the training of the algorithms the callbacks feature with an early stop, in order to prevent overfitting. This results showed in figure 16 depict that this model is overfitting and it is too powerful for the input data.

4.4 Fourth Architecture

This architecture has as input the same input as the third architecture. This architecture is even deeper than the previous one.

```
#defining model
def My_Fourth_Cnn():
  model_4=Sequential()

  #adding convolution layer
  model_4.add(Conv2D(128,kernel_size=(3,3),activation='relu',input_shape=(100,100,1)))
  #adding pooling layer
  model_4.add(MaxPool2D(pool_size=(2,2)))
  #Dropoutlayer
  model_4.add(Dropout(0.05))


  ###Second Convolutional Layer
  model_4.add(Conv2D(128,kernel_size=(3,3),activation='relu'))
  model_4.add(MaxPool2D(pool_size=(2,2)))
  model_4.add(Dropout(0.05))

  ###Third Convolutional Layer
  model_4.add(Conv2D(128,kernel_size=(3,3),activation='relu'))
  model_4.add(MaxPool2D(pool_size=(2,2)))
  model_4.add(Dropout(0.05))

  ###Fourth Convolutional Layer
  model_4.add(Conv2D(256,kernel_size=(3,3),activation='relu'))
  model_4.add(MaxPool2D(pool_size=(2,2)))
  model_4.add(Dropout(0.05))

  #adding fully connected layer
  model_4.add(Flatten())
  model_4.add(Dense(256,activation='relu'))
  model_4.add(Dropout(0.05))

  #adding output layer
  model_4.add(Dense(1,activation='sigmoid'))

  #compiling the model
  model_4.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])

  return model_4
```

Figure 17: Fourth Architecture

Also this model was trained with augmented data, for 40 epochs with batch size equal to 32.

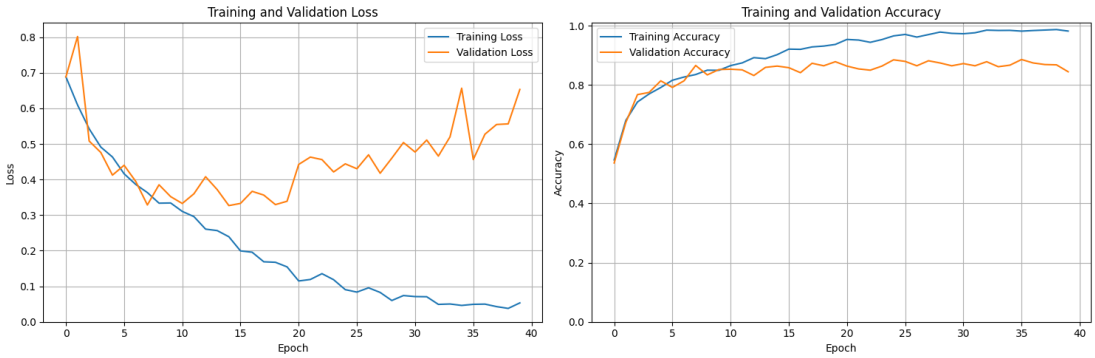

Figure 18: Accuracy and loss for training and validation

From the plot we can see that this model is overfitting and performing poorly. The evaluation of the model in fact gives back as result 0.88 as accuracy and 0.49 as loss.

4.4.1 Improvements

Summary of the modifications that have been made to this architecture:

- tuning of the Convolution filters and dropout rate
- Adding BatchNormalization to the model (worse results so dropped soon after)
- tuning of the learning rate and kernels

The results of the tuned model after training are showed in the following figure.

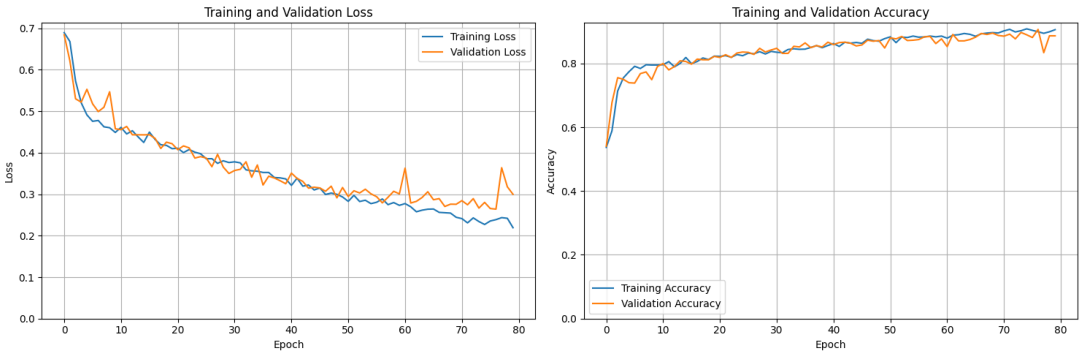

Figure 19: Accuracy and loss for training and validation

This results shows a major improvement with respect to all the previous results. The training and validation loss plot still shows a little overfitting from the fifthy epoch on. The evaluation of the model gives as result 0.83 of accuracy and 0.41 as loss. The result we obtained still need to be improved.

4.5 Fifth Architecture

Since the Architecture that I've tried so far did not work as I expected them to do I decided to change the way I encoded and resized the input images. In fact for this last architecture I decided to encode the images as RGB and resize them as 224,224.
What I did learn from the previous architecture is that all the previous architectures suffered from overfitting, so I decided to introduce other data augmentation features in the ImageDataGenerator function such as rotation_range and width_shift_range. This Fifth architecture is actually similar to the second architecture, the main difference is the input shape which in this case is (224,224,3), the three is encluded because in this case images are encoded as RGB. The architecture is the one you can find in the following figure.

```python
#defining model
def My_Second_Cnn():
  model_2=Sequential()

  #adding convolution layer
  model_2.add(Conv2D(32,kernel_size=(3,3),activation='relu',input_shape=(224,224,3)))
  #adding pooling layer
  model_2.add(MaxPool2D(pool_size=(2,2)))
  #Dropoutlayer
  model_2.add(Dropout(0.05))

  ###Second Convolutional Layer
  model_2.add(Conv2D(64,kernel_size=(3,3),activation='relu'))
  model_2.add(MaxPool2D(pool_size=(2,2)))
  model_2.add(Dropout(0.05))

  ###Third Convolutional Layer
  model_2.add(Conv2D(128,kernel_size=(3,3),activation='relu'))
  model_2.add(MaxPool2D(pool_size=(2,2)))
  model_2.add(Dropout(0.05))

  #adding fully connected layer
  model_2.add(Flatten())
  model_2.add(Dense(128,activation='relu'))
  model_2.add(Dropout(0.05))

  #adding output layer
  model_2.add(Dense(1,activation='sigmoid'))

  #compiling the model
  model_2.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])

  return model_2
```

Figure 20: Fifth Architecture

This model was trained for 40 epochs and with batch size of 32.

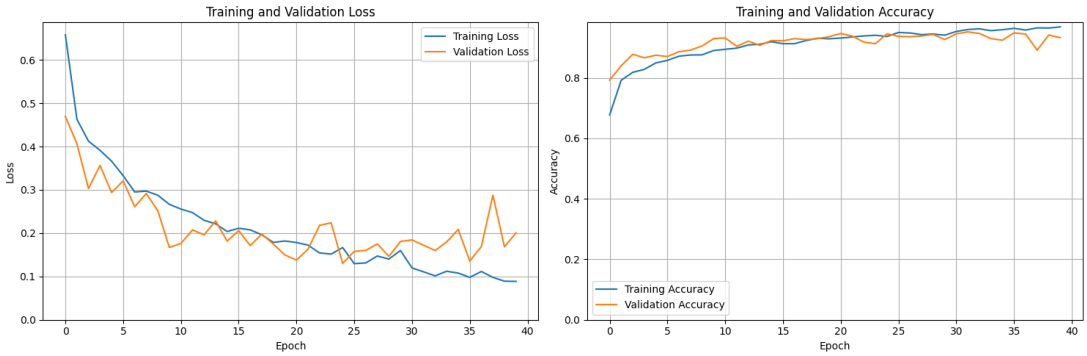

Figure 21: Accuracy and loss for training and validation

From the plot we can see that the model has a good performance and in fact the results that I obtained from the evaluation of the model are 0.93 as accuracy and 0.19 as loss.

4.5.1 Improvements

Also in this case I am going to summarize the improvements that I made to this model and show you just the final best architecture.

- Convolution filters and dropout rate tuning
- Tuning of the learning rate
- Kernel size tuning
- Adding BatchNormalization (Worsen the results)

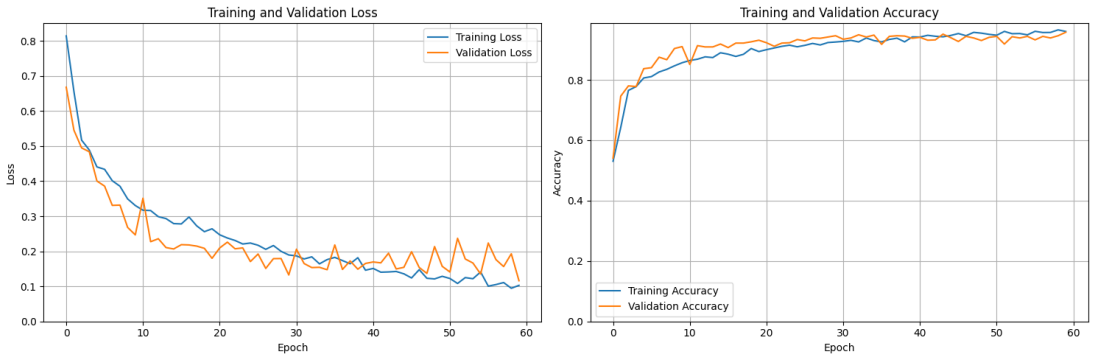

Figure 22: Accuracy and loss for training and validation

The evaluation of this architecture gives as accuracy 94% and 19% as loss. For this architecture you can find in the project the calculus for the Recall, Precision, F1 score and ROC curve(with 0.98 AUC). The following images depict the Confusion matrix of the architecture.

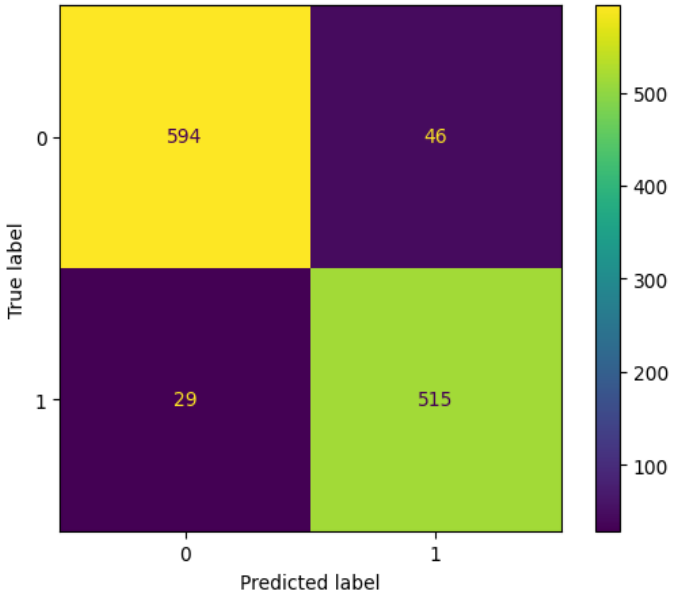

Figure 23: Confusion Matrix

4.6 K-Fold CV and stratified K-Fold

KFold Cross Validation is a resampling techniques used to evaluate deep learning models used mostly when the training set size is small. This kind of resampling technique is also used in order to have more robust results. This procedure has a k parameter that refers to the number of groups into which the dataset is going to be split into. In this case the k parameter is set to be 5. In this case the evaluation of the results is done thanks to the accuracy and zero one loss.
The difference between KFold and Stratified KFold:

- KFold divide the dataset into kfolds
- Stratified makes sure that each fold has given a label, the same proportion of observations.

I decided to do K-fold cross validation just with the architectures that performed better.

4.6.1 K-Fold CV with second architecture

k	Accuracy	0-1 loss
1	0.88	0.12
2	0.87	0.13
3	0.86	0.14
4	0.84	0.16
5	0.88	0.12

Table 1: Metrics in each fold

metrics	Average
accuracy	0.86
loss	0.34
0-1 loss	0.14

Table 2: Metrics average

4.6.2 K-fold CV with Fourth architecture

k	Accuracy	0-1 loss
1	0.88	0.12
2	0.87	0.13
3	0.86	0.14
4	0.84	0.16
5	0.88	0.12

Table 3: Metrics in each fold

metrics	Average
accuracy	0.86
loss	0.28
0-1 loss	0.14

Table 4: Metrics average

4.6.3 Stratified KFold CV with Fifth architecture and RGB input

k	Accuracy	0-1 loss
1	0.94	0.05
2	0.94	0.06
3	0.94	0.06
4	0.94	0.06
5	0.94	0.06

Table 5: Metrics in each fold

Given the so small loss range it can be deduced that this architecture is stable.

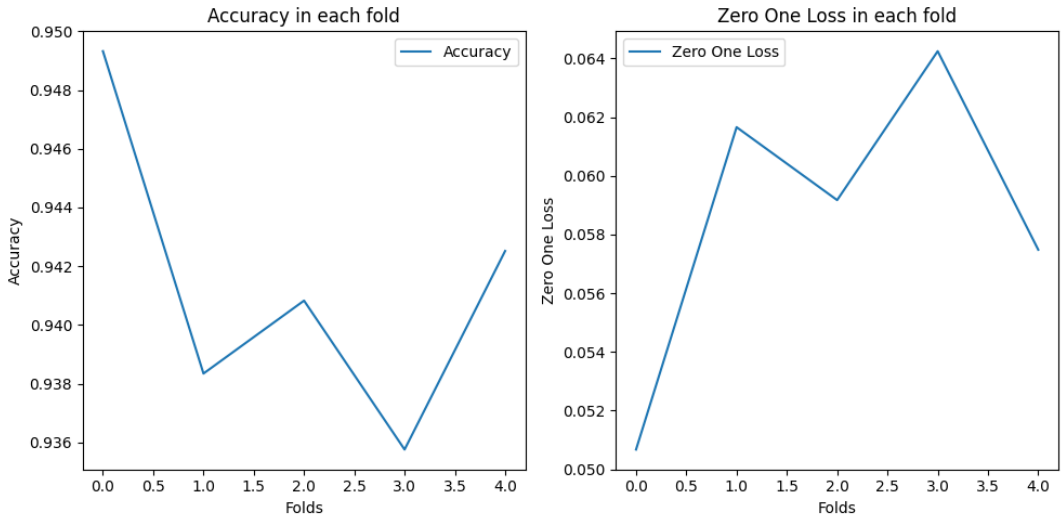

Figure 24: Accuracy and loss for training and validation

metrics	Average
accuracy	0.94
loss	0.21
0-1 loss	0.06

Table 6: Metrics average

5 Further Improvements to the project

Further improvements that could be made to improve this projects are:

- Deepens the best model
- tune the batch_size and number of epochs
- Try with different optimization and activation functions

6 Conclusion and comments

The accuracy and loss that was obtained with the best model (Fifth architecture with K-fold CV) is 94% and the 0-1 loss is 0.06. This is actually a quite good and robust (thanks to K-fold CV and data augmentation) result. Data augmentation, hyperparameter tuning, and dropout layers have been key for improving the architectures.
Thanks to this project I learned that greyscale images are not always the best choice with which to start an image recognition task and that all hyperparameters are important in order to improve the model performances of your model.

7 References

https://machinelearningmastery.com/how-to-calculate-precision-recall-f1-and-more-for-deep-learning-models/

https://stackoverflow.com/questions/31421413/how-to-compute-precision-recall-accuracy-and-f1-score-for-the-multiclass-case

https://www.baeldung.com/cs/training-validation-loss-deep-learning

https://stackoverflow.com/questions/50825936/confusion-matrix-on-images-in-cnn-keras

https://www.datacamp.com/tutorial/cnn-tensorflow-python