UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE POLITICHE, ECONOMICHE E SOCIALI

Ms.C. Degree in Data Science and Economics

Cats and Dogs Image Classification

Luca Bertoletti

luca.bertoletti1@studenti.unimi.it

Abstract

This project aimed to develop an accurate classification model for distinguishing between images of cats and dogs using a Convolutional Neural Network (CNN). Several different CNN architectures were tested to determine which one performed the best on the input data. The selected model was then evaluated using cross-validation (CV) to compute the risk estimate. The best model achieved an accuracy close to 90%, indicating that it is effective at accurately classifying images of cats and dogs.

Contents

1 Introduction **4**

2 Input Data **4**

3 Foundations of CNNs **5**

3.1 Convolutional Layer 5

3.2 Stride 5

3.3 Pooling Layer 5

3.4 Dropout Layer 6

3.5 Fully Connected Layer 6

3.6 Activation Function 6

3.7 Optimizer 7

3.8 Loss Function 7

4 Models **8**

4.1 Model: 3 CL, different filters 8

4.2 Model 1: 3 CL, 32 Filters 9

4.3 Model 2: 3 CL, 64 Filters 10

4.4 Model 3: 4 CL, 32 Filters 11

5 Results **12**

6 Appendix **13**

1 Introduction

The goal of this project is to implement a Convolutional Neural Network (CNN) to correctly classify images of cats and dogs. The input data for this project consists of 25000 images that have been transformed to grayscale and equally distributed between the two classes. The models were designed using Keras, and five different architectures were tested to determine the best-performing model. Once the best model was identified, a risk estimate was applied through cross-validation to validate the performance of the model.

In the following sections of this report, I will discuss the processing of input data, the model architectures, the results and the final considerations of this project.

2 Input Data

The Cats vs Dogs dataset is a well-known dataset used for image classification tasks. It consists of 25000 images of cats and dogs, with each class having an equal number of samples. The dataset is widely used to evaluate the performance of machine learning algorithms for image classification. To prepare the data for classification, the images were transformed into grayscale, which reduces the dimensionality of the input data and can improve the performance of the model by reducing the computational complexity.

During the process of transforming the images to grayscale, some images were removed because they could not be properly converted. Despite this, the resulting dataset still contains a balanced number of samples for each class, with 12476 images of cats and 12470 images of dogs. This ensures that the model is trained and tested on an equal number of samples from each class, which is essential to avoid bias in the classification results. While losing some data is never ideal, the resulting dataset still contains a substantial amount of samples for each class, making it suitable for training and evaluating machine learning models.

The last step in preparing the input data was to create a pandas DataFrame in which each image is labeled according to whether it depicts a cat or a dog. By creating a DataFrame, we can easily keep track of the labels and the corresponding file names for each image in the dataset. This also allows to easily split the data into training, validation, and testing sets.

3 Foundations of CNNs

Convolutional Neural Networks (CNNs) are a type of deep learning algorithm based on the concept of convolution, which is a mathematical operation used to extract features from an image. CNNs are used for a variety of computer vision tasks, such as image classification or object detection. They are widely used in industry applications, such as self-driving cars and facial recognition systems. In the next pages I will discuss the key features of CNNs.

3.1 Convolutional Layer

Convolutional layers are a key component of Convolutional Neural Networks (CNNs) that perform the convolution operation on the input image to extract relevant features. The convolution operation involves sliding a small window called a filter or kernel over the input image, computing the dot product between the filter and the corresponding input region at each position, and producing a feature map that summarizes the activations of each filter across the input image. This process allows the CNN to detect different types of patterns in the input image, such as edges, corners, and textures, and create higher-level representations of the input image as it moves through the network. Convolutional layers typically have multiple filters that are learned during training, allowing the CNN to extract a diverse set of features from the input image.

3.2 Stride

In convolutional neural networks (CNNs), strides refer to the number of steps the filter (also known as a kernel) moves horizontally or vertically across the input image. When the stride is set to 1, the filter moves one pixel at a time. However, if the stride is set to a higher number, the filter will skip some pixels, moving faster across the image. The choice of stride size affects the output size of the convolutional layer. A smaller stride leads to more output pixels and a larger receptive field, but at the cost of computational efficiency. A larger stride reduces the output size and receptive field, but speeds up computation. Strides can be used to downsample an image or to adjust the spatial resolution of the feature maps in a CNN.
In this project I used the default stride value of 1.

3.3 Pooling Layer

The purpose of a pooling layer is to downsample the feature maps generated by the convolutional layer. This downsampling helps to reduce the dimensionality of the data and make the network more computationally efficient. Pooling layers operate on small local regions of the input data, typically using either max pooling or average pooling to calculate the output value for each region.

Max pooling selects the maximum value from each region, while average pooling takes the average value. By reducing the spatial resolution of the feature maps, pooling layers help to make the network more robust to small variations in the input image while preserving the most important features.
In this project I used Max pooling.

3.4 Dropout Layer

Dropout layers are a regularization technique commonly used in neural networks to prevent overfitting. The basic idea behind dropout is to randomly ”drop out” (set to zero) some of the neurons in a layer during training, with a specified probability. This forces the network to learn more robust features and reduces the likelihood of overfitting to the training data.
In this project I used a Dropout value of 0.2.

3.5 Fully Connected Layer

A Flatten layer is often used in conjunction with fully connected layers in convolutional neural networks. It takes the output from the last convolutional or pooling layer, which is a 3D tensor of height, width, and depth, and flattens it into a 1D vector. This allows the output to be fed into a fully connected layer for final classification or prediction. The Flatten layer essentially reshapes the tensor into a form that can be processed by the fully connected layer. Fully connected layers, also known as dense layers, are a type of neural network layer where every neuron in the layer is connected to every neuron in the previous layer. These layers are typically used at the end of a neural network to transform the output of the preceding layers into a final output that can be used for prediction or classification. The number of neurons in a fully connected layer is typically chosen based on the complexity of the task and the size of the input data. In deep neural networks, fully connected layers can have millions or even billions of parameters, making them computationally expensive to train.

3.6 Activation Function

Activation functions are a crucial component of neural networks that introduce non-linearity into the model. They are applied to the output of each neuron in a layer to produce the output of that neuron, which is then fed to the next layer. The activation function introduces a threshold for the neuron’s output, allowing the network to learn complex, non-linear relationships between the input and output data. Some commonly used activation functions include the sigmoid, ReLU (Rectified Linear Unit), and tanh (hyperbolic tangent) functions. The sigmoid function is a smooth, S-shaped curve that maps any input to a value between 0 and 1, making it useful for binary classification tasks. ReLU is a simple and widely used activation function that sets any negative input to zero, allowing the network to learn faster and avoid the vanishing gradient problem.

3.7 Optimizer

The optimizer is a key component of training neural networks that determines how the model's parameters (weights and biases) are updated during the training process. The optimizer uses the gradient of the loss function with respect to the model's parameters to update them in the direction that minimizes the loss. Some commonly used optimizers include Stochastic Gradient Descent (SGD), Adam, and Adagrad.
In this project I used the Adam (Adaptive Moment Estimation), that is an adaptive optimization algorithm that is widely used for training deep neural networks. It is a combination of two other optimization algorithms, AdaGrad and RMSprop, and adapts the learning rate for each parameter based on the first and second moments of the gradients. Adam maintains a moving average of the gradient and the squared gradient for each parameter, which helps to adjust the learning rate for each parameter based on the variance and mean of the gradients. This adaptive learning rate ensures that the parameters are updated more effectively, leading to faster convergence and better performance. Adam is also robust to noisy or sparse gradients, making it an effective optimizer for a wide range of deep learning tasks. However, it may require more memory than other optimizers and can sometimes result in overfitting if the learning rate is not set appropriately. Despite its limitations, Adam is considered to be one of the most effective optimization algorithms for deep neural networks.

3.8 Loss Function

The loss function is a critical component of training a convolutional neural network (CNN). The goal of the loss function is to measure the difference between the predicted output of the network and the true output, which is typically represented as a scalar value. In CNNs, the most commonly used loss functions include cross-entropy loss, mean squared error (MSE) loss, and binary cross-entropy loss.
Binary cross-entropy loss is a widely used loss function for binary classification tasks in convolutional neural networks. It is a variation of cross-entropy loss and is designed to measure the difference between the predicted output and the true binary label, which is typically represented as either 0 or 1. Binary cross-entropy loss penalizes the network for predicting probabilities that are far from the true probabilities of the target class.

4 Models

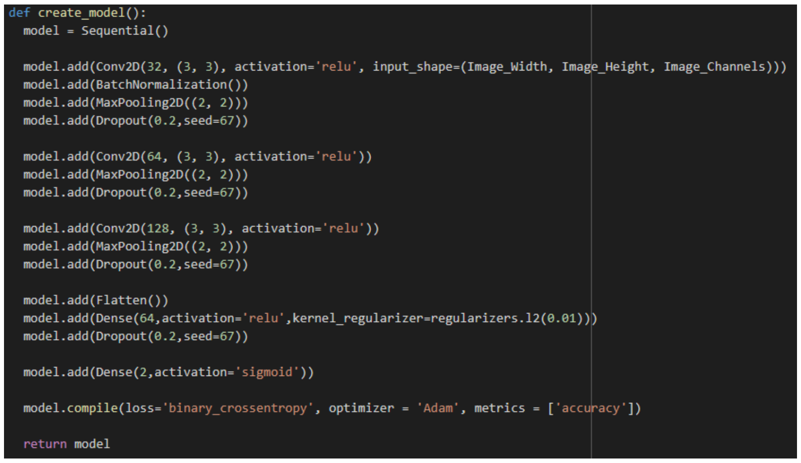
In this project, the performance of four different convolutional neural network (CNN) models were evaluated. By testing multiple models, I was able to identify which architecture performed best for this specific dataset and problem. To keep computational costs under control, the number of training epochs is limited to just 30.

I tested a variety of CNN architectures to identify the most effective configuration given our computational constraints. Specifically, I experimented with different numbers of layers and filters. Each of these factors played an important role in the final performance of the models.

I standardized the architecture of each CNN model such that the activation function of every layer except for the final one was set to rectified linear unit (ReLU). This choice was made based on the widespread use and success of ReLU in CNNs. The final dense layer, which outputs the prediction, was configured with a sigmoid activation function, that is a common choice for binary classification problems. For all CNN models, I used the Adam optimizer.

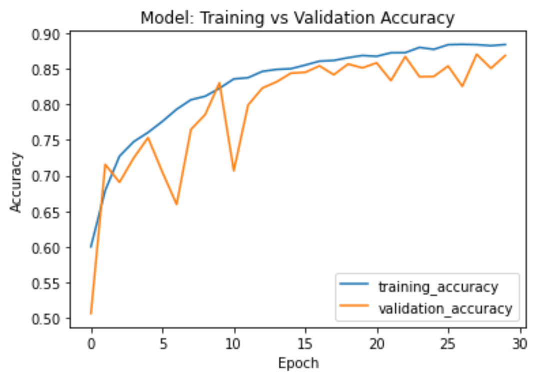
4.1 Model: 3 CL, different filters

The first model is composed by 3 different Convolutional Layers and its architecture is represented in Figure 1:

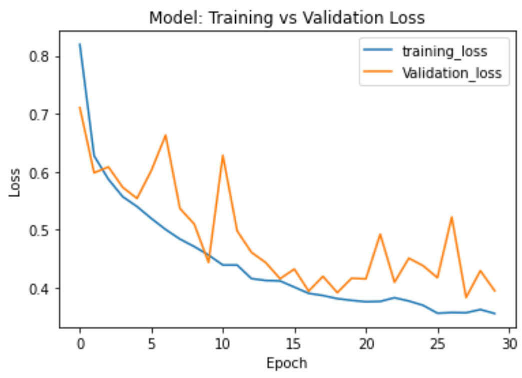


```
def create_model():
  model = Sequential()

  model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(Image_Width, Image_Height, Image_Channels)))
  model.add(BatchNormalization())
  model.add(MaxPooling2D((2, 2)))
  model.add(Dropout(0.2,seed=67))

  model.add(Conv2D(64, (3, 3), activation='relu'))
  model.add(MaxPooling2D((2, 2)))
  model.add(Dropout(0.2,seed=67))

  model.add(Conv2D(128, (3, 3), activation='relu'))
  model.add(MaxPooling2D((2, 2)))
  model.add(Dropout(0.2,seed=67))

  model.add(Flatten())
  model.add(Dense(64,activation='relu',kernel_regularizer=regularizers.l2(0.01)))
  model.add(Dropout(0.2,seed=67))

  model.add(Dense(2,activation='sigmoid'))

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

  return model
```

Figure 1: Model Architecture



(a) Model Accuracy

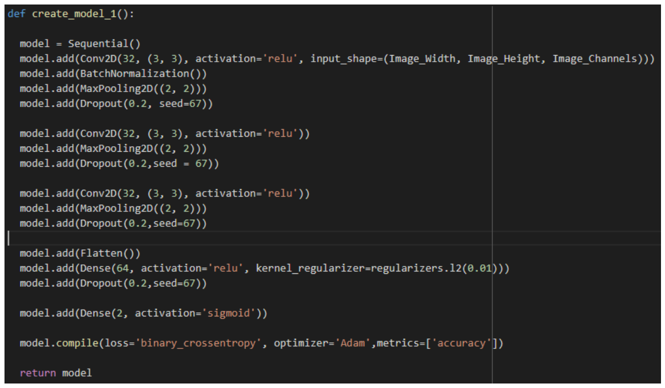


(b) Model Loss

As it can be seen from the Figure 1, the model is composed by 3 different CL, with 3x3 kernels size and increasing number of filters.
This model can't reach good values of loss, so we need to change the architecture to see an improvement. The model reached an accuracy of 87%.

4.2 Model 1: 3 CL, 32 Filters

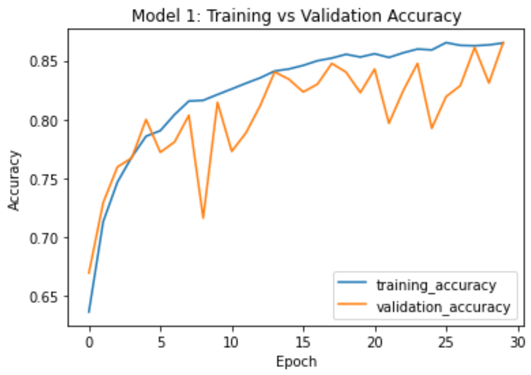
To increase the performance of the model I chose to try with a simpler model composed again by 3 CL but of the same number of filters. The structure of the model is shown below:



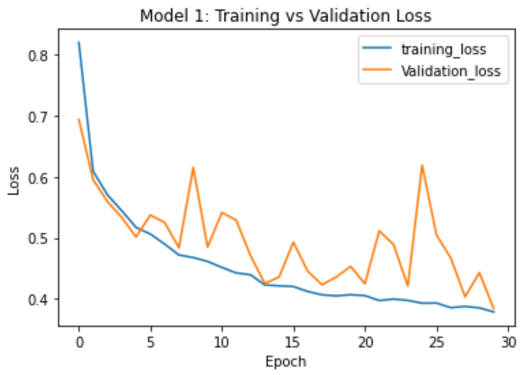
```
def create_model_1():

  model = Sequential()
  model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(Image_Width, Image_Height, Image_Channels)))
  model.add(BatchNormalization())
  model.add(MaxPooling2D((2, 2)))
  model.add(Dropout(0.2, seed=67))

  model.add(Conv2D(32, (3, 3), activation='relu'))
  model.add(MaxPooling2D((2, 2)))
  model.add(Dropout(0.2,seed = 67))

  model.add(Conv2D(32, (3, 3), activation='relu'))
  model.add(MaxPooling2D((2, 2)))
  model.add(Dropout(0.2,seed=67))

  model.add(Flatten())
  model.add(Dense(64, activation='relu', kernel_regularizer=regularizers.l2(0.01)))
  model.add(Dropout(0.2,seed=67))

  model.add(Dense(2, activation='sigmoid'))

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

  return model
```

Figure 3: Model 1 Architecture

As it can be seen from the Figure 3, the model is composed by 3 different CL, with 3x3 kernels size and 32 filters for each CL. Actually this model produced almost the same results in terms of accuracy than the previous one, but there is a slight reduction in terms of loss. This model reached an accuracy of 87%.



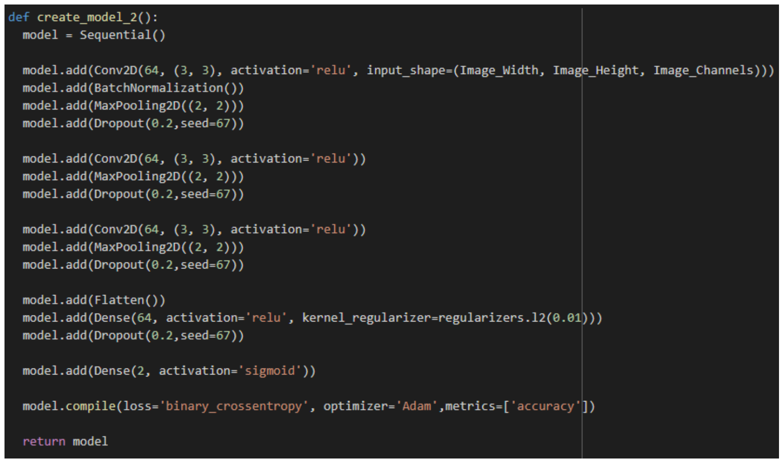
(a) Model 1 Accuracy



(b) Model 1 Loss

4.3 Model 2: 3 CL, 64 Filters

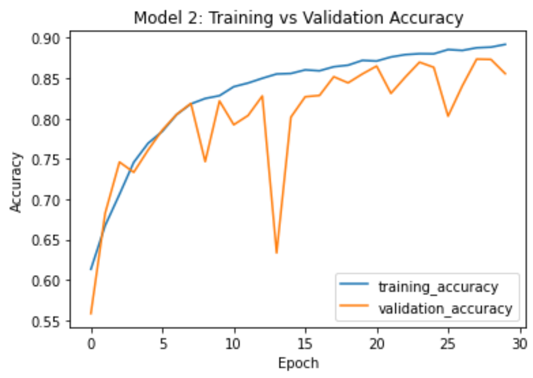
This model is always composed by 3 CL but I increased the number of filters to 64 and its structure can be seen in the Figure below:



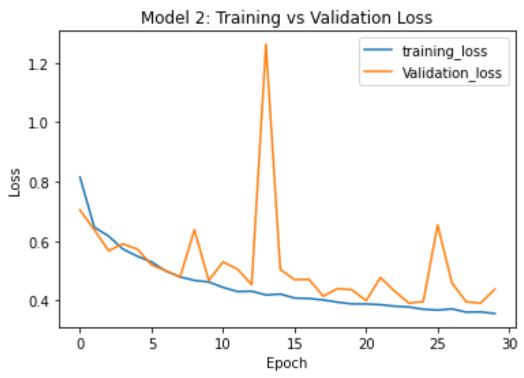
```
def create_model_2():
  model = Sequential()

  model.add(Conv2D(64, (3, 3), activation='relu', input_shape=(Image_Width, Image_Height, Image_Channels)))
  model.add(BatchNormalization())
  model.add(MaxPooling2D((2, 2)))
  model.add(Dropout(0.2,seed=67))

  model.add(Conv2D(64, (3, 3), activation='relu'))
  model.add(MaxPooling2D((2, 2)))
  model.add(Dropout(0.2,seed=67))

  model.add(Conv2D(64, (3, 3), activation='relu'))
  model.add(MaxPooling2D((2, 2)))
  model.add(Dropout(0.2,seed=67))

  model.add(Flatten())
  model.add(Dense(64, activation='relu', kernel_regularizer=regularizers.l2(0.01)))
  model.add(Dropout(0.2,seed=67))

  model.add(Dense(2, activation='sigmoid'))

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

  return model
```

Figure 5: Model 2 Architecture

As it can be seen from the Figure 5, the model is composed by 3 different CL, with 3x3 kernels size and 64 filters for each CL. The model behaved slightly worst than the previous one, reaching worst values of accuracy and loss. It reached an accuracy of 85%.



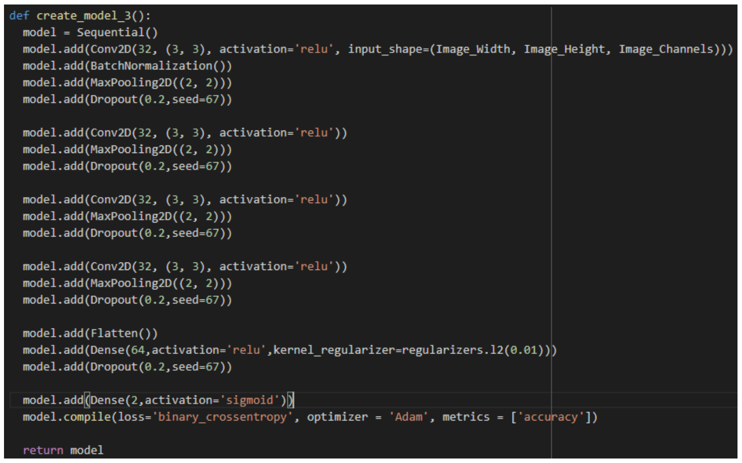
(a) Model 2 Accuracy



(b) Model 2 Loss

4.4 Model 3: 4 CL, 32 Filters

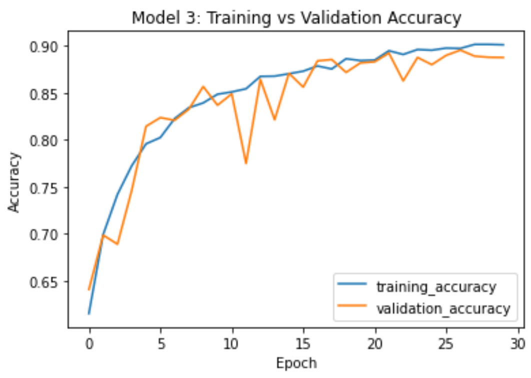
For the last model I added a CL and chose to use 32 filters for every CL since model 1 performed well. The model structure is shown below:



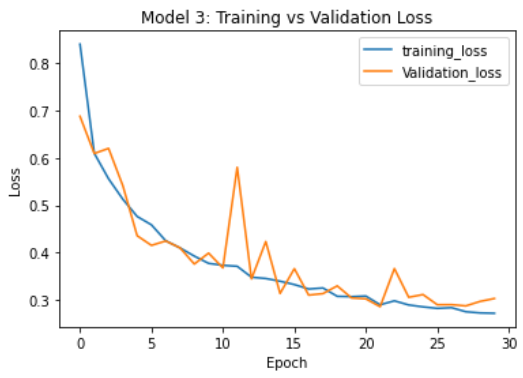
```python
def create_model_3():
  model = Sequential()
  model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(Image_Width, Image_Height, Image_Channels)))
  model.add(BatchNormalization())
  model.add(MaxPooling2D((2, 2)))
  model.add(Dropout(0.2,seed=67))

  model.add(Conv2D(32, (3, 3), activation='relu'))
  model.add(MaxPooling2D((2, 2)))
  model.add(Dropout(0.2,seed=67))

  model.add(Conv2D(32, (3, 3), activation='relu'))
  model.add(MaxPooling2D((2, 2)))
  model.add(Dropout(0.2,seed=67))

  model.add(Conv2D(32, (3, 3), activation='relu'))
  model.add(MaxPooling2D((2, 2)))
  model.add(Dropout(0.2,seed=67))

  model.add(Flatten())
  model.add(Dense(64,activation='relu',kernel_regularizer=regularizers.l2(0.01)))
  model.add(Dropout(0.2,seed=67))

  model.add(Dense(2,activation='sigmoid'))
  model.compile(loss='binary_crossentropy', optimizer = 'Adam', metrics = ['accuracy'])

  return model
```

Figure 7: Model 3 Architecture

As it can be seen from the Figure 7, the model is composed by 4 different CL, with 3x3 kernels size and 32 filters for each CL. In the plots below the 2 curves follow each other for each epoch and, in terms of accuracy and loss, its values are lower than previous models.
Since this is the model that performed better so far, I used it for the Cross-Validation process. The learning curves, for each split, can be found in the appendix.
The model performed well also in terms of accuracy because it reached a value of almost 90%.



(a) Model 3 Accuracy



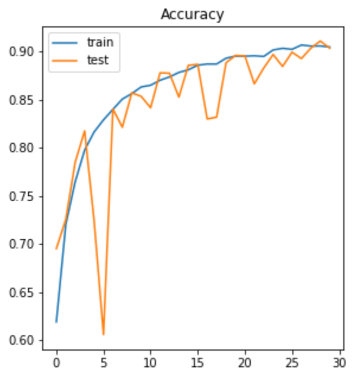
(b) Model 3 Loss

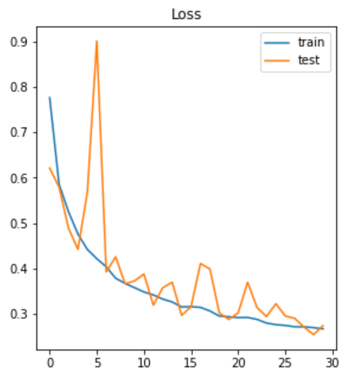
5 Results

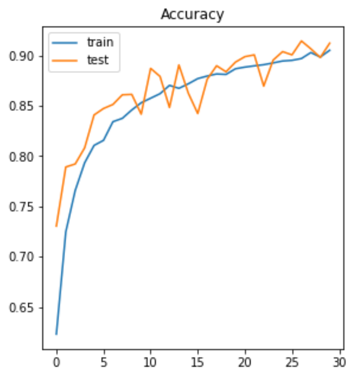
In this short project I experimented with different CNN architectures to find the model that can give a good performance in classifying cats and dogs images.
Starting from a model with an increasing number of filters, I then moved to models with same number of filters, reducing the computational costs, that performed well in the end.
The better results obtained by the last model can be explained by the fact a smaller number of filters results in fewer model parameters, which reduces the model's capacity to memorize the training data and forces it to learn more generalizable features.
The final model can correctly classify 9 images out of 10.

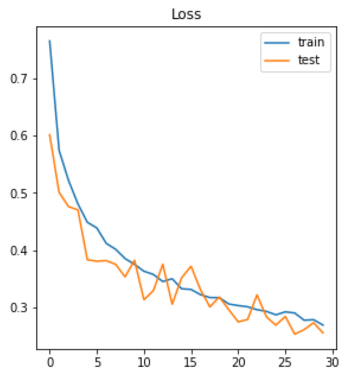
6 Appendix

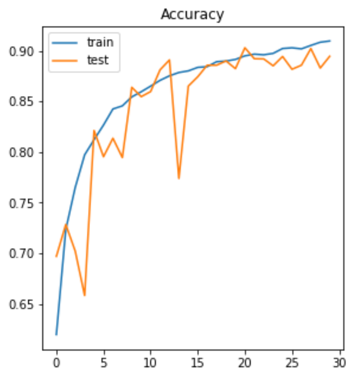
Learning curves for each split of the CV

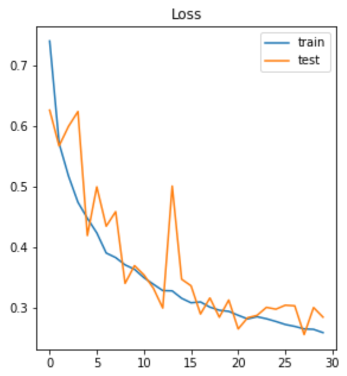


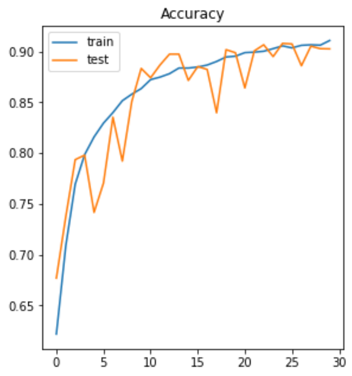


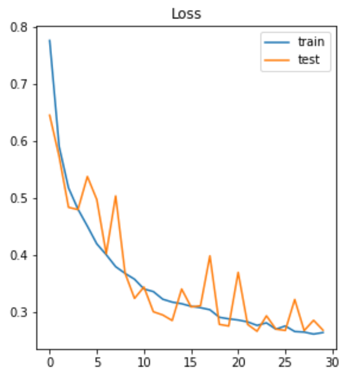


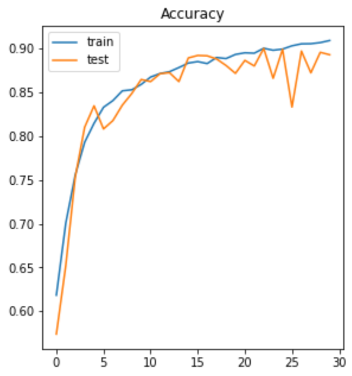


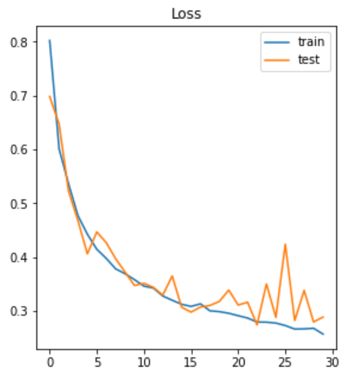

Accuracy
train
test


Loss
train
test


Accuracy
train
test


Loss
train
test


Accuracy
train
test


Loss
train
test