UNIVERSITÀ
DEGLI STUDI
DI MILANO

Neural Networks

Cats vs Dogs

Project for the course of

Statistical Methods for Machine Learning

Student: Edoardo D'Angelo

Student code: 947729

Course: LM-48 Informatica Magistrale

Accademic Year 2021/2022

Report

Abstract

This paper will explain the procedures and results of developing a neural network for classifying images of cats and dogs. The basic network architectures are built from the examples shown during the course and each result obtained from their improvement through methods to avoid overfitting and tuning of hyperparameters are described and discussed. All models are evaluated through 5-fold cross validation, whose average results are calculated and displayed via graphs. After the experimentation and improvement of the architecture and obtaining a satisfactory result, a reflection will finally be made on the entire development, evaluating the performance from the best-built model and comparing them with those obtained from residual neural network architecture.

Contents

1. Introduction

Cats and Dogs recognition is a computer vision problem that seems simple when a human is trying to distinguish between the two pets, but computationally, it has been solved with satisfying accuracy only a few decades back with the introduction of Convolutional Neural Networks. Since then, it has mostly been used as a challenging problem for the design of neural network architectures aimed at dog or cat image classification. These images are contained in a dataset that can be used for learning how to develop, evaluate and use convolutional deep-learning neural networks for the classification of images. This includes how to develop a robust test harness for estimating the performance of the model and exploring improvements for the model by changing the parameters.
This project shows the different stages of developing an optimal and well-performing solution. Specifically, it will be shown how, starting from an extremely simple architecture, the solution has been improved through the analysis of previous results, thus adding layers and/or introducing techniques to improve subsequent performance.

2. Project Design and Setup

Since Cats vs Dogs is a well-known challenge in the field of machine learning, it was easy to find a multitude of examples on platforms such as Kaggle by searching the net for documentation on the subject. Many of these implementations were very different from each other, making it difficult to choose what could be a valid solution to be a starting point or comparison for this project.
Given the basic knowledge of both the python programming language and the implementation of machine learning with Tensorflow, it was decided to start the project by avoiding pre-existing exhaustive solutions, thus building very simple architectures and improving them as experience and knowledge of the subject accumulate. This way it was also possible to test new algorithms to be introduced on more computationally complex models on very simple neural networks, reducing the time and resources required to visualize their effects.
As a reference for the construction of the models, the examples of dense neural networks and convolutional neural networks shown in class were used, referring to their performance as basic thresholds.
The entire project was carried out on Google Colab, which provides a maximum of 12GB of RAM and a limited time of GPU usage in the free version. Initially, the limitations of this setup did not affect the project, as the simplest networks did not require the use of the GPU and a large portion of RAM, while for the most complex networks it was required to pay attention to consumption, like avoiding to fill the RAM, causing the reset of the work session, and having too long training, tuning, and testing periods, subsequently denying GPU usage for the following days.
Before starting the data elaboration and implementation of neural networks, it was necessary to define some constraint points following the project requests.

- K-Fold Cross validation: For risk estimation, delivery requires the use of 5-fold cross validation. The k-fold cross-validation procedure divides the given dataset into k non-overlapping folds. Each of the k folds is allowed to be used as a held-back test set, while all other folds collectively are used as a training dataset. A total of k models are fit and evaluated on the k hold-out test sets and the mean performance is reported.

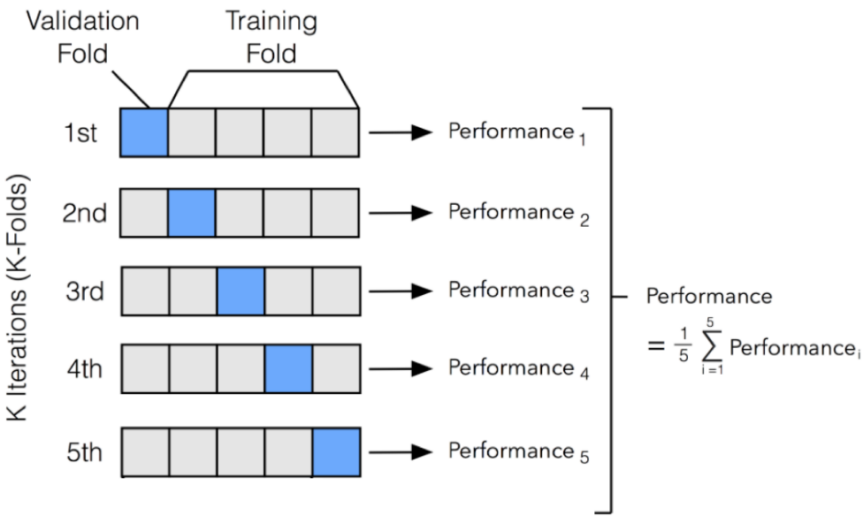


Figure 1: 5-fold cross validation procedure

- Zero-One loss: project delivery also explicitly requires that the evaluation of cross-validation estimates must be computed according to zero-one loss. Since the function is not differentiable, it cannot be inserted directly as a loss function during model training and validation. It was therefore necessary to build a custom evaluation metric, defining the zero-one loss as the complementary of the accuracy value. Just for double checking, within the iteration of k-fold cross validation, a pre-built function of the sklearn library is used to display the zero-one loss validation value of each fold and confirm that it matches that produced by the metric.
- Output Graphs: model performance is displayed through graphs containing the history of train loss and validation zero-one loss, training and validation accuracy, and the confusion matrix. Each iteration of the k-fold cross validation produces the previous graphs related to the single fold, and after training and evaluation, the graphs related to the average performance are generated.
- Activation function (Input/Hidden): as the activation function for all layers included in the input and hidden layer of each architecture, it is used the Rectified Linear function. This function has two major advantages over sigmoidal functions:

  1. ReLU is very simple to calculate, as it involves only a comparison between its input and the value 0.
  2. It has a derivative of either 0 or 1, depending on whether its input is respectively negative or not.

  As a consequence, the usage of ReLU helps to prevent the exponential growth in the computation required to operate the neural network. If the CNN scales in size, the computational cost of adding extra ReLUs increases linearly.

- Activation function (Output): Cats vs Dogs can be considered a classification problem, so the activation function chosen for the output layer was softmax. The softmax function converts a vector of N real numbers into a probability distribution of N possible outcomes and is often used as the last activation function of a neural network to normalize the output of a network to a probability distribution over predicted output classes.
- Training Loss: since softmax is the output activation function, Categorical Cross-Entropy is used as training loss, as it can be used as a loss function for a multi-class classification model where there are two or more output labels.

3. Data

The dataset consists of a collection of 25000 images, divided into 12500 photos of cats and 12500 photos of dogs. The images differ in both proportion and size.

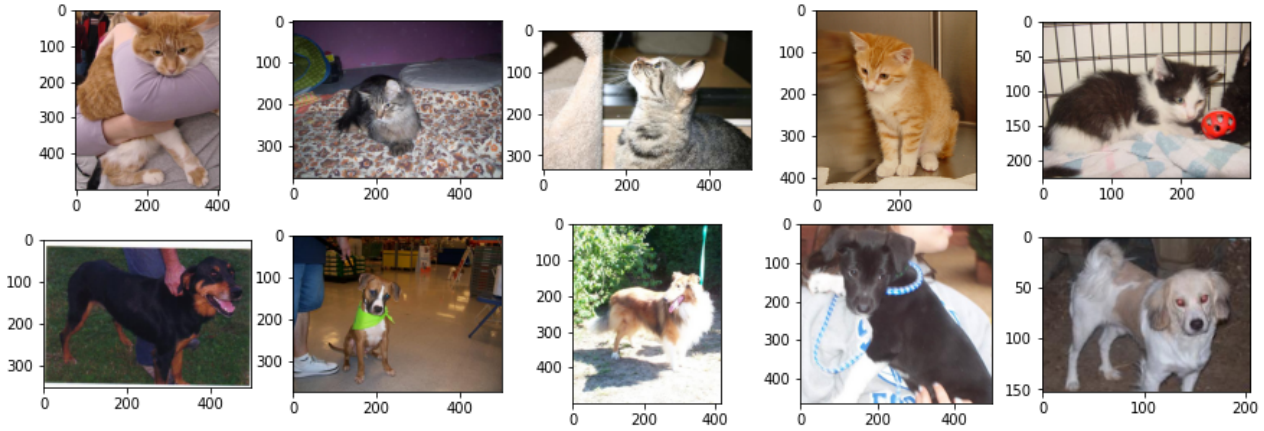


Figure 2: An example of cats and dogs images in the dataset

Looking at the folder examples, it can be seen that some images are not consistent with the dataset they belong to. Notably, some of these images contain drawn dogs or cats, some are heavily edited, while others contain only completely off-topic writings or subjects. It is not known whether they were intentionally inserted to simulate noise within the dataset. Here are some of the images in question:

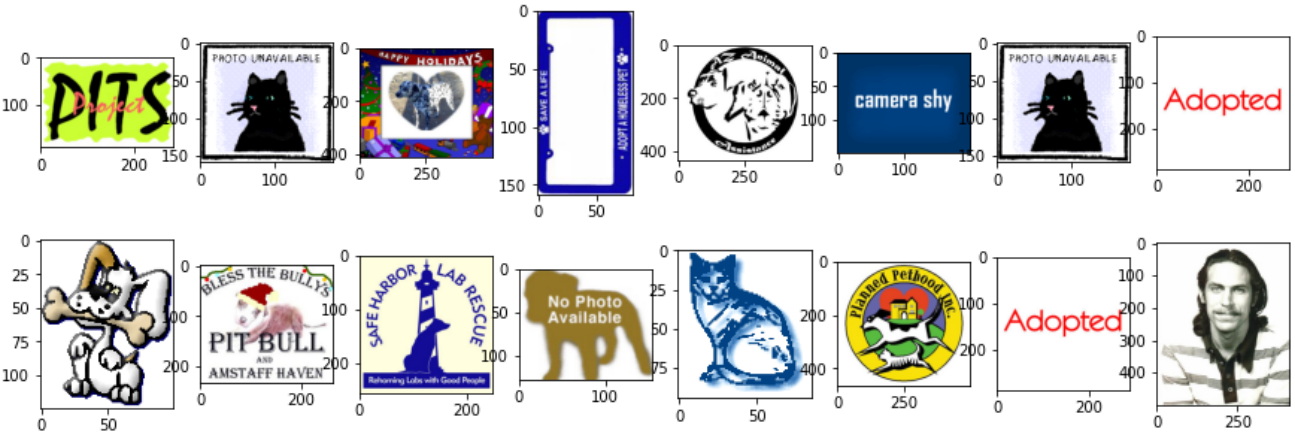


Figure 3: The inconsistent images in the dataset

In the early stages of the project, the entire dataset is used, thus including these images. However, they are excluded during data preparation of the best network implementations. In this document, it will be specified in the various chapters dedicated to implementations if the cleaned dataset has been used.

3.1 Preparation

To speed up upload, the dataset was copied to GitHub, allowing the repository to be cloned directly into Colab.
After specifying the functions for creating charts and the output paths for saving them, the size for resizing images is determined. This value is defined according to the complexity of the model, as too large a size of the image applied to particularly complex model results in overloading the Colab RAM and therefore the loss of session data. The standard image size used for the simplest models is 75x75 pixels, for more complex models the size will be specified in the subsection dedicated to the implementation.
Being able to directly access the folders, the images are added in an array together with a label associated with the folders to which they belonged, as dogs are associated with the value 0, while cats have the value 1.
As the images iterate, they are rescaled and converted to grayscale. Corrupted images, which were found to be 54 within this dataset, are also checked and removed. Coincidentally, included in this issue is a portion of images previously described as inconsistent with dogs and cats.

3.2 Shuffle

The generated array is then shuffled randomly. This is necessary because if this operation is not carried out, it will cause most iterations of the 5-fold cross validation to produce incorrect behaviors, leading to having 4 times a validation fold composed only of dogs or cats.

3.3 Classification

Images and labels are then divided into an input and an output array, the first containing image information (height, width, color levels) and the second structured to represent the percentage of membership in each class. At the end of this preparation, the reworked images are shown and classified according to their label.

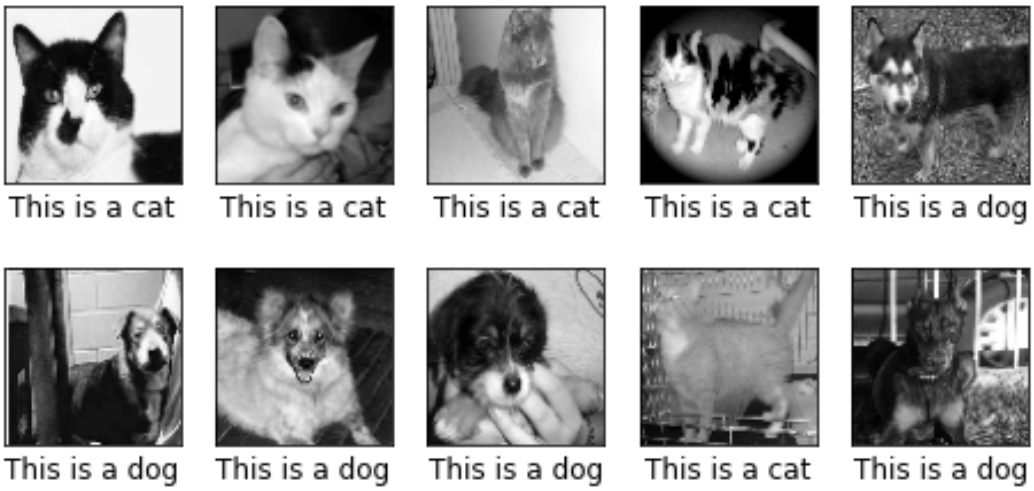


Figure 4: Some of the grayscale and resized images

Given the need for multiple tests in CNN testing, pickle files containing a grayscale shuffled input and output were also generated, to speed up the restoration of the working environment after the session reset.

3.4 Normalization

When data is not zero-centered, the values of the pixels can range from 0 and 255. During forward propagation, dot products of these pixel values are performed with the weight matrix for that particular layer. Multiplying these large pixel values takes a lot of computation resources and time, hence the model converges very slowly. On the other hand, if the data is normalized or zero-centered, the pixel values are small and the computation required and time to converge the model reduces significantly. It was then decided to normalize the data and it was done by dividing the data array by 255.0, bringing pixel values into the range [0,1].

4. Approach

All implementations and their results are listed below, while the related model codes and average graphs of the results can be viewed in the appendices A.

4.1 Neural Network with 2/N Dense Layers

For the first implementation attempts, neural networks composed solely of dense layers were chosen. This choice was taken to acquire the minimum accuracy threshold that a neural network can obtain from this specific dataset and have an extremely simple and very fast model to tow and evaluate for testing new algorithms to be applied to more complex solutions.
The first model coincides with the one presented in the example tutorial, which is a dense implementation based on two 64-unit layers followed by the final layer computing the softmax probabilities for each of the 2 categories corresponding to the 2 classes. For training a network of this complexity, it was decided to use a batch size of 32 and epochs equal to 20.

```
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.6641467213630676 - Accuracy: 0.5913827419281006 - Zero-One N Loss: 2039 - Zero-One Loss: 0.40861723446893783
------------------------------------------------------------------------
> Fold 2 - Loss: 0.687013566493988 - Accuracy: 0.5888955593109131 - Zero-One N Loss: 2051 - Zero-One Loss: 0.4111044297454399
------------------------------------------------------------------------
> Fold 3 - Loss: 0.6553612351417542 - Accuracy: 0.6077370047569275 - Zero-One N Loss: 1957 - Zero-One Loss: 0.3922629785528162
------------------------------------------------------------------------
> Fold 4 - Loss: 0.6567394137382507 - Accuracy: 0.6169573068618774 - Zero-One N Loss: 1911 - Zero-One Loss: 0.3830426939266386
------------------------------------------------------------------------
> Fold 5 - Loss: 0.6736499667167664 - Accuracy: 0.5860893726348877 - Zero-One N Loss: 2065 - Zero-One Loss: 0.4139106033273201
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.6673821806907654
> Accuracy: 0.5982123970985412 (+- 0.012020858299940582)
> Zero-One N Loss: 2004.6 (+- 60.01866376386598)
> Zero-One Loss: 0.4017875880042305 (+- 0.012020851472084725)
```

Figure 5: 5-Cross Validation Results

The results show an accuracy of about 60%, only 10% better than the case of binary random guessing. Moreover, from the graphs illustrated in A.1, it is possible to see how the validation accuracy is extremely unstable from the earliest epochs and diverges from training accuracy, a clear symptom of overfitting. Although they are pretty bad, these results are very useful to understand the key points for improving the network.

The first experiment that has been carried out on neural networks is that of tuning hyperparameters. In particular, looking at the various solutions present on online platforms, the number of neurons placed inside the dense layers for an optimal solution varied from implementation to implementation, making everything very confusing. Hence it was decided to use keras turner for the search for the optimal hyperparameters in the implementation of the neural networks of the project. KerasTuner is a scalable hyperparameter optimization framework that automates the process of manually searching for optimal hyperparameters. The keras tuner library provides an implementation of algorithms like random search, hyperband, and bayesian optimization (the one chosen for this project) for hyperparameter tuning. These algorithms find good hyperparameters settings in less number of trials without trying all possible combinations and they search for hyperparameters in the direction that is giving good results.
Implementing this feature in such a simple architecture allowed the testing of this functionality using the minimum of Colab's resources, as well as an easy understanding of the tuner capabilities and its implementation technique. As parameters required by the tuner for the search for the best HP, it was decided to use epochs equal to 20 and extract a random sample of images equal to 20% from the dataset for the split into the training set and test set, as applying the k-fold cross validation to each trial of the tuner would have considerably increased the execution time, especially in the most complex models.

```
Search space summary
Default search space size: 3
num_of_neurons_l1 (Int)
{'default': None, 'conditions': [], 'min_value': 16, 'max_value': 512, 'step': 16, 'sampling': 'linear'}
num_of_neurons_l2 (Int)
{'default': None, 'conditions': [], 'min_value': 16, 'max_value': 512, 'step': 16, 'sampling': 'linear'}
learning_rate (Choice)
{'default': 0.001, 'conditions': [], 'values': [0.001, 0.0001], 'ordered': True}
Trial 10 Complete [00h 04m 56s]
val_categorical_accuracy: 0.64368736743927

Best val_categorical_accuracy So Far: 0.6494989991188049
Total elapsed time: 00h 54m 57s
[INFO] optimal number of neurons in first layer: 240
[INFO] optimal number of neurons in second layer: 368
[INFO] optimal learning rate: 0.0001
```

Figure 6: Tuning Summary and Results

Given the overfitting demonstrated by the previous model, early stopping was also introduced. Early Stopping is a regularization technique for deep neural networks that stops training when parameter updates no longer improve on a validation set. The current best parameters are stored and updated during training, and when parameter updates no longer yield an improvement (after a set number of iterations) training is stopped and the last best parameters are used. For all implementations, the early stopping patience was set to 5.

```
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.6413754224777222 - Accuracy: 0.6388777494430542 - Zero-One N Loss: 1802 - Zero-One Loss: 0.361122244488978
------------------------------------------------------------------------
> Fold 2 - Loss: 0.6541816592216492 - Accuracy: 0.633794367313385 - Zero-One N Loss: 1827 - Zero-One Loss: 0.36620565243535774
------------------------------------------------------------------------
> Fold 3 - Loss: 0.6469125151634216 - Accuracy: 0.6249749660491943 - Zero-One N Loss: 1871 - Zero-One Loss: 0.3750250551212668
------------------------------------------------------------------------
> Fold 4 - Loss: 0.6371978521347046 - Accuracy: 0.6390058398246765 - Zero-One N Loss: 1801 - Zero-One Loss: 0.36099418721186616
------------------------------------------------------------------------
> Fold 5 - Loss: 0.6660495400428772 - Accuracy: 0.6323912739753723 - Zero-One N Loss: 1834 - Zero-One Loss: 0.3676087392262979
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.649143397808075
> Accuracy: 0.6338088393211365 (+- 0.0051526630394608365)
> Zero-One N Loss: 1827.0 (+- 25.635912310662945)
> Zero-One Loss: 0.36619117569675336 (+- 0.005152666776035625)
```

Figure 7: 5-Cross Validation Results

The effect of tuning on the number of neurons is an increase of 3% on the accuracy value and the halving of the standard deviation relative to zero-one loss between folds, implying better consistency. As can be seen from the graphs in A.2, the early stopping technique stopped training prematurely to avoid overfitting.

In this model, hypertuning of the number of dense layers within the hidden layer was also tested, looking for what is the optimal number of layers and neurons per layer. The tuning was carried out using the same previous setup, with the only difference being the choice in the number of dense layers between 2 and 5, and the step of possible neurons per layer, which goes from 16 to 32 to reduce the search space. However, the optimal number of dense layers identified by the tuner turned out to be 2, thus changing only the number of neurons per layer.

```
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.6454205513000488 - Accuracy: 0.6362725496292114 - Zero-One N Loss: 1815 - Zero-One Loss: 0.3637274549098196
------------------------------------------------------------------------
> Fold 2 - Loss: 0.6552247405052185 - Accuracy: 0.6355983018875122 - Zero-One N Loss: 1818 - Zero-One Loss: 0.36440168370414916
------------------------------------------------------------------------
> Fold 3 - Loss: 0.6492553353309631 - Accuracy: 0.6239727139472961 - Zero-One N Loss: 1876 - Zero-One Loss: 0.3760272599719383
------------------------------------------------------------------------
> Fold 4 - Loss: 0.6487869024276733 - Accuracy: 0.628983736038208 - Zero-One N Loss: 1851 - Zero-One Loss: 0.3710162357185809
------------------------------------------------------------------------
> Fold 5 - Loss: 0.6411421298980713 - Accuracy: 0.6396071314811707 - Zero-One N Loss: 1798 - Zero-One Loss: 0.3603928643014632
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.647965931892395
> Accuracy: 0.6328868865966797 (+- 0.00563335134110868)
> Zero-One N Loss: 1831.6 (+- 28.06136133547337)
> Zero-One Loss: 0.3671130997211902 (+- 0.005633341020973698)
```

Figure 8: 5-Cross Validation Results

The results show accuracy almost identical to that of the previous architecture, as expected from the tuning results of the model. Given the mediocre performance obtained, it was decided to switch to architectures more suitable for image processing.

4.2 CNN with 1/3/M Conv+MaxPool Layer

Convolutional neural networks are a type of neural network that is typically used for image recognition tasks. The way that a convolutional neural network works is by taking an image and breaking it down into a series of smaller images, or filter maps. Each filter map is then processed by the neural network in order to identify features within the image.
The first CNN architecture consists solely of a convolutional layer followed by a MaxPooling2D layer, used to reduce image dimensionality through pixel downsampling, and one dense layer.

Training is carried out with epochs equal to 10 and a batch size of 64. This architecture is proposed only to verify the difference in performance between the best dense implementation model previously described and a simple CNN model. For that reason, tuning techniques and early stopping are also not used.

```
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.5837339758872986 - Accuracy: 0.7450901865959167 - Zero-One N Loss: 1272 - Zero-One Loss: 0.2549098196392786
------------------------------------------------------------------------
> Fold 2 - Loss: 0.5926339030265808 - Accuracy: 0.7430346608161926 - Zero-One N Loss: 1282 - Zero-One Loss: 0.25696532371216674
------------------------------------------------------------------------
> Fold 3 - Loss: 0.601264476776123 - Accuracy: 0.7360192537307739 - Zero-One N Loss: 1317 - Zero-One Loss: 0.2639807576668671
------------------------------------------------------------------------
> Fold 4 - Loss: 0.5916826128959656 - Accuracy: 0.7316095232963562 - Zero-One N Loss: 1339 - Zero-One Loss: 0.2683904590098216
------------------------------------------------------------------------
> Fold 5 - Loss: 0.5801448225975037 - Accuracy: 0.7352174520492554 - Zero-One N Loss: 1321 - Zero-One Loss: 0.26478252154740434
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.5898919582366944
> Accuracy: 0.738194215297699 (+- 0.005058341209740238)
> Zero-One N Loss: 1306.2 (+- 25.166644591601795)
> Zero-One Loss: 0.2618057763151077 (+- 0.0050583357366785495)
```

Figure 9: 5-Cross Validation Results

Making a comparison with previous results, it is possible to notice an increase of 10per in accuracy values, which goes from 63% to 73%. However, the percentage deviation of the loss between the individual folds remains unchanged. Setting the number of epochs for training equal to 10 shows the presence of overfitting, as shown in A.4 graphs, and it will therefore be solved in subsequent architectures through early stopping and other techniques.

The second proposed architecture is the one shown in class using the MNIST dataset for demonstration purposes, and it is a variation of the three-block VGG model. It was therefore decided to modify it for use on the CatsVsDogs dataset. To prevent overfitting, the early stopping technique previously illustrated has been reintroduced. In addition, this model introduces a different type of layer to the architecture, namely the dropout layer, which is a mask that nullifies the contribution of some neurons towards the next layer and leaves unmodified all others. Dropout layers are important in training CNNs because they prevent overfitting on the training data.

```
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.3791038990020752 - Accuracy: 0.8591182231903076 - Zero-One N Loss: 703 - Zero-One Loss: 0.14088176352705406
------------------------------------------------------------------------
> Fold 2 - Loss: 0.33730819821357727 - Accuracy: 0.8677089810371399 - Zero-One N Loss: 660 - Zero-One Loss: 0.13229104028863503
------------------------------------------------------------------------
> Fold 3 - Loss: 0.370612233877182 - Accuracy: 0.8707155585289001 - Zero-One N Loss: 645 - Zero-One Loss: 0.12928442573662058
------------------------------------------------------------------------
> Fold 4 - Loss: 0.3935149908065796 - Accuracy: 0.8612948656082153 - Zero-One N Loss: 692 - Zero-One Loss: 0.13870515133293249
------------------------------------------------------------------------
> Fold 5 - Loss: 0.3817075490951538 - Accuracy: 0.8604930639266968 - Zero-One N Loss: 696 - Zero-One Loss: 0.13950691521346958
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.37244937419891355
> Accuracy: 0.863866138458252 (+- 0.004521366378577453)
> Zero-One N Loss: 679.2 (+- 22.5867217630182)
> Zero-One Loss: 0.13613385921974236 (+- 0.004521363545581102)
```

Figure 10: 5-Cross Validation Results

The results show that the introduction of more convolutional layers has a high increase in performance, with the accuracy value increasing by 13%, going from 73% to 86%. The standard deviation of the zero-one loss also decreases slightly.

From the graphs in A.5, it is also possible to see how the divergence between training and validation accuracy has significantly decreased compared to the previous implementation. However, from the analysis of the training epochs, it can be assumed that there is an underfitting effect, as at the 20th epoch the validation accuracy is still possibly growing.

```
Epoch 15/20
312/312 [==============================] - 6s 18ms/step - loss: 0.2442 - categorical_accuracy: 0.8962 - custom_zero_loss: 0.1037 - val_loss: 0.3388 - val_categorical_accuracy: 0.8645
Epoch 16/20
312/312 [==============================] - 5s 16ms/step - loss: 0.2296 - categorical_accuracy: 0.9029 - custom_zero_loss: 0.0971 - val_loss: 0.4149 - val_categorical_accuracy: 0.8487
Epoch 17/20
312/312 [==============================] - 5s 16ms/step - loss: 0.2238 - categorical_accuracy: 0.9041 - custom_zero_loss: 0.0959 - val_loss: 0.3443 - val_categorical_accuracy: 0.8659
Epoch 18/20
312/312 [==============================] - 5s 15ms/step - loss: 0.2130 - categorical_accuracy: 0.9087 - custom_zero_loss: 0.0913 - val_loss: 0.3763 - val_categorical_accuracy: 0.8559
Epoch 19/20
312/312 [==============================] - 4s 14ms/step - loss: 0.1988 - categorical_accuracy: 0.9162 - custom_zero_loss: 0.0838 - val_loss: 0.3484 - val_categorical_accuracy: 0.8677
Epoch 20/20
312/312 [==============================] - 5s 16ms/step - loss: 0.1940 - categorical_accuracy: 0.9192 - custom_zero_loss: 0.0807 - val_loss: 0.3373 - val_categorical_accuracy: 0.8677
```

Figure 11: Training Underfitting

Following the guideline of the work done on dense implementations, tuning is carried out to look for optimal values for network hyperparameters. For this particular configuration, two hypertuning sessions were performed. The first tuning was carried out on the dropout value and the size of the filters of the convolutional layers while maintaining the original proportion between them.

```
Search space summary
Default search space size: 4
num_of_filters (Choice)
{'default': 32, 'conditions': [], 'values': [32, 64, 128], 'ordered': True}
dropout_value (Choice)
{'default': 0.3, 'conditions': [], 'values': [0.3, 0.4, 0.5], 'ordered': True}
num_of_neurons (Int)
{'default': None, 'conditions': [], 'min_value': 32, 'max_value': 512, 'step': 32, 'sampling': 'linear'}
learning_rate (Choice)
{'default': 0.001, 'conditions': [], 'values': [0.001, 0.0001], 'ordered': True}

Trial 20 Complete [00h 05m 03s]
val_categorical_accuracy: 0.8575150370597839

Best val_categorical_accuracy So Far: 0.8721442818641663
Total elapsed time: 01h 07m 52s
[INFO] optimal number of filters: 128
[INFO] optimal number of neurons: 256
[INFO] optimal dropout value: 0.5
[INFO] optimal learning rate: 0.001
```

Figure 12: Tuning Summary and Results

Given the results of the previous architecture, the number of epochs has been increased from 20 to 40. This impacted computation times, but they remained acceptable through the use of early stopping.

```
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.48097488284111023 - Accuracy: 0.8683366775512695 - Zero-One N Loss: 657 - Zero-One Loss: 0.1316633266533066
------------------------------------------------------------------------
> Fold 2 - Loss: 0.36144962906837463 - Accuracy: 0.8727200031280518 - Zero-One N Loss: 635 - Zero-One Loss: 0.1272800160352776
------------------------------------------------------------------------
> Fold 3 - Loss: 0.4125235676765442 - Accuracy: 0.8721186518669128 - Zero-One N Loss: 638 - Zero-One Loss: 0.12788133894568054
------------------------------------------------------------------------
> Fold 4 - Loss: 0.44083479046821594 - Accuracy: 0.8500701785087585 - Zero-One N Loss: 748 - Zero-One Loss: 0.14992984566045298
------------------------------------------------------------------------
> Fold 5 - Loss: 0.39629024267196655 - Accuracy: 0.8586891293525696 - Zero-One N Loss: 705 - Zero-One Loss: 0.14131088394467828
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.4184146225452423
> Accuracy: 0.8643869280815124 (+- 0.008743619868033155)
> Zero-One N Loss: 676.6 (+- 43.61009057546201)
> Zero-One Loss: 0.1356130822478792 (+- 0.008743627108233872)
```

Figure 13: 5-Cross Validation Results

For a very small increase in the accuracy value, a doubling in the standard deviation of the loss is obtained, resulting in less consistency between folds. The overall performance is, therefore, worse than the previous ones.

The second tuning was carried out by looking for the optimal number of convolutional and maxpooling layer pairs and dense layers plus dropout layers, as well as the size of the filters and the dropout value. For better tuning of the learning rate, it was also decided to introduce an additional technique for its optimization, namely ReduceLROnPlateau, which is a callback to reduce the learning rate when a metric has stopped improving. The callback monitors a quantity and if no improvement is seen for a selected number of epochs, it reduces the learning rate of a given factor at each iteration, until a minimum value is reached. This allows the model to take smaller learning steps to the optimal solution, without fluctuating around the global minimum. To allow the correct application of keras tuner on this architecture, the clean dataset was used and the image size was increased to 100x100.

```
Search space summary
Default search space size: 5
num_of_filters (Choice)
{'default': 32, 'conditions': [], 'values': [32, 64], 'ordered': True}
num_of_conv_layers (Int)
{'default': None, 'conditions': [], 'min_value': 2, 'max_value': 5, 'step': 1, 'sampling': 'linear'}
dropout_value (Choice)
{'default': 0.4, 'conditions': [], 'values': [0.4, 0.5], 'ordered': True}
num_of_neurons (Choice)
{'default': 64, 'conditions': [], 'values': [64, 128], 'ordered': True}
num_of_layers (Int)
{'default': None, 'conditions': [], 'min_value': 1, 'max_value': 3, 'step': 1, 'sampling': 'linear'}
Trial 20 Complete [00h 02m 15s]
val_categorical_accuracy: 0.9043321013450623

Best val_categorical_accuracy So Far: 0.9043321013450623
Total elapsed time: 00h 34m 19s
[INFO] optimal number of starting filters: 32
[INFO] optimal number of conv layers: 3
[INFO] optimal dropout value: 0.4
[INFO] optimal number of last dense layer neurons: 128
[INFO] optimal number of dense layers: 2
```

Figure 14: Tuning Summary and Results

Note that, unlike the previous tuning, the optimal value for the filter size is that of the starting architecture, which equals 32, and also the optimal number of neurons in the last dense layer coincides.

```
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.5499472618103027 - Accuracy: 0.9045326709747314 - Zero-One N Loss: 476 - Zero-One Loss: 0.09546730846369833
------------------------------------------------------------------------
> Fold 2 - Loss: 0.6475592255592346 - Accuracy: 0.9047332406044006 - Zero-One N Loss: 475 - Zero-One Loss: 0.09526674689129566
------------------------------------------------------------------------
> Fold 3 - Loss: 0.5026096105575562 - Accuracy: 0.9081428050994873 - Zero-One N Loss: 458 - Zero-One Loss: 0.09185720016044929
------------------------------------------------------------------------
> Fold 4 - Loss: 0.6159674525260925 - Accuracy: 0.8856799006462097 - Zero-One N Loss: 570 - Zero-One Loss: 0.11432009626955475
------------------------------------------------------------------------
> Fold 5 - Loss: 0.5241544246673584 - Accuracy: 0.9013237357139587 - Zero-One N Loss: 492 - Zero-One Loss: 0.09867629362214203
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.5680475950241088
> Accuracy: 0.9008824706077576 (+- 0.007901616259318229)
> Zero-One N Loss: 494.2 (+- 39.397461847179954)
> Zero-One Loss: 0.09911752908142801 (+- 0.007901616896746873)
```

Figure 15: 5-Cross Validation Results

Compared to previous results, accuracy shows an increase of 4%, finally reaching 90% on average. The standard deviation also improves, but remains worse than the initial one, as does the divergence between training and validation accuracy shown in A.7. Note that the architectural change and the introduction of ReduceLROnPlateau have also increased the number of epochs required for accuracy convergence.

4.3 CNN with N Conv + BatchNorm + MaxPool + Dropout Layers

Following the analysis of the graphs produced by the previous model and observing a discrepancy of almost 10% between the training and validation accuracy, as a new attempt to improve the custom network it was decided to introduce a dropout layer following each convolutional layer. In addition, due to the introduction of the ReduceLROnPlateau, batch normalization layers have been introduced to speed up training and use higher learning rates.
Hypertuning was then carried out, which led to the following optimization:

```
Search space summary
Default search space size: 4
num_of_filters (Choice)
{'default': 32, 'conditions': [], 'values': [32, 64], 'ordered': True}
dropout_value (Choice)
{'default': 0.2, 'conditions': [], 'values': [0.2, 0.3, 0.4, 0.5], 'ordered': True}
num_of_conv_layers (Int)
{'default': None, 'conditions': [], 'min_value': 2, 'max_value': 5, 'step': 1, 'sampling': 'linear'}
num_of_layers (Int)
{'default': None, 'conditions': [], 'min_value': 1, 'max_value': 3, 'step': 1, 'sampling': 'linear'}

Trial 20 Complete [00h 05m 49s]
val_categorical_accuracy: 0.9025270938873291

Best val_categorical_accuracy So Far: 0.92158043384552
Total elapsed time: 01h 42m 33s
[INFO] optimal number of starting filters: 32
[INFO] optimal number of conv layers: 5
[INFO] optimal dropout value: 0.4
[INFO] optimal number of dense layers: 1
```

Figure 16: Tuning Summary and Results

Just as in the previous architecture, the optimal filter size of the first convolutional layer is 32, and the same dropout value is also maintained. The model built on the optimized hyperparameters produced the following validated cross-parameter results:

```
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.21689538657665253 - Accuracy: 0.9117528796195984 - Zero-One N Loss: 440 - Zero-One Loss: 0.08824709185720014
------------------------------------------------------------------------
> Fold 2 - Loss: 0.2116153985261917 - Accuracy: 0.916967511177063 - Zero-One N Loss: 414 - Zero-One Loss: 0.0830324909747292
------------------------------------------------------------------------
> Fold 3 - Loss: 0.17476655542850494 - Accuracy: 0.9330124258995056 - Zero-One N Loss: 334 - Zero-One Loss: 0.06698756518251103
------------------------------------------------------------------------
> Fold 4 - Loss: 0.20350639522075653 - Accuracy: 0.9265944361686707 - Zero-One N Loss: 366 - Zero-One Loss: 0.07340553549939832
------------------------------------------------------------------------
> Fold 5 - Loss: 0.20274952054023743 - Accuracy: 0.9243882894515991 - Zero-One N Loss: 377 - Zero-One Loss: 0.07561171279582835
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.20190665125846863
> Accuracy: 0.9225431084632874 (+- 0.007442323296305624)
> Zero-One N Loss: 386.2 (+- 37.10741165858918)
> Zero-One Loss: 0.07745687926193341 (+- 0.0074423208300419385)
```

Figure 17: 5-Cross Validation Results

The accuracy of the model has an increase of 2% compared to the previous one, keeping the zero-one loss deviation between the folds unchanged. However, performance graphs in A.8 show much less difference over epochs between training and validation accuracy values than previous results. The number of epochs required for convergence remains equivalent to previous models.

During the execution of the tuner, starting from the 10th trial, the values of the hyperparameters underwent minimal variations, which led to equally minimal variations in the final performance. From this information, it can be assumed that this configuration is possibly optimal for the fixed dataset, and further changes to this network architecture will not lead to a significant improvement in performance.

4.4 ResNet50

To assess whether the performance of the latest architecture is good enough, a comparison was made with the implementation of ResNet50, which is a variant of ResNet model that has 48 Convolution layers along with 1 MaxPool and 1 Average Pool layer. For this architecture, it was necessary to reconstruct the dataset while maintaining the 3 color levels of the images. This created problems on the Colab platform due to RAM consumption, requiring a reduction in image size, going from 75x75 to 60x60. Reconstructing the dataset also removed images that were not considered consistent. The model built using ResNet50 and using the early stopping and ReduceLROnPlateu techniques led to the following results:

```
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.5361605286598206 - Accuracy: 0.9217810034751892 - Zero-One N Loss: 390 - Zero-One Loss: 0.07821901323706382
------------------------------------------------------------------------
> Fold 2 - Loss: 0.581388533115387 - Accuracy: 0.9171680808067322 - Zero-One N Loss: 413 - Zero-One Loss: 0.08283192940232653
------------------------------------------------------------------------
> Fold 3 - Loss: 0.4802125096321106 - Accuracy: 0.9277978539466858 - Zero-One N Loss: 360 - Zero-One Loss: 0.07220216606498198
------------------------------------------------------------------------
> Fold 4 - Loss: 0.4445512890815735 - Accuracy: 0.9267950057983398 - Zero-One N Loss: 365 - Zero-One Loss: 0.07320497392699554
------------------------------------------------------------------------
> Fold 5 - Loss: 0.5570046901702881 - Accuracy: 0.9193742275238037 - Zero-One N Loss: 402 - Zero-One Loss: 0.08062575210589651
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.519863510131836
> Accuracy: 0.9225832343101501 (+- 0.0041278678117262985)
> Zero-One N Loss: 386.0 (+- 20.581545131500697)
> Zero-One Loss: 0.07741676694745288 (+- 0.004127867054051492)
```

Figure 18: 5-Cross Validation Results

The results obtained are comparable to that of the previous CNN model, except for the deviation, which is almost halved, and for the number of epochs necessary for the convergence of accuracy, showing overall better performance. It should also be reported that, during a further test on ResNet50, using standard image sizes led to a 2/3% increase in performance, right before freezing the Colab session.

5. Results Summary

Taking the first 20 images of the shuffled array as samples, these are the results of the prediction made on the last architectures previously discussed:

Figure 19: Model Prediction for CNN N Layers and ResNet50

Making a comparison between the various architectures, the starting accuracy was about 60% with dense implementations, passing through the introduction of CNN with a value of 74% and improving the model until reaching the peak of 92%. Overall, considering the dense implementations, an increase of 32% was obtained, while for CNN alone it is equal to 18%. As for the standard deviation, in the succession of implementations, it has undergone increases and decreases that have led it to remain stable around 0.7%, only improving by 0.3% in the ResNet50. As complexity, the number of layers, and overfitting reduction techniques have been introduced, the average number of epochs required for convergence has also increased significantly, starting from a maximum of 10 to a minimum of 30. This value is smaller in ResNet50, which requires fewer epochs to reach peak training and validation performances.

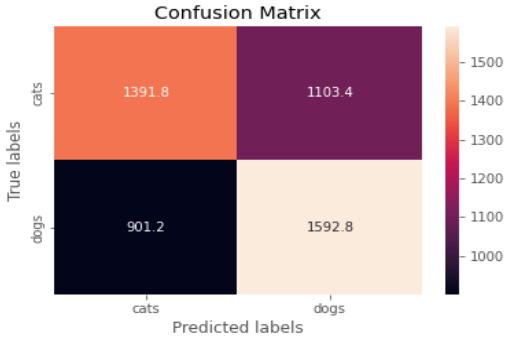
6. Conclusion

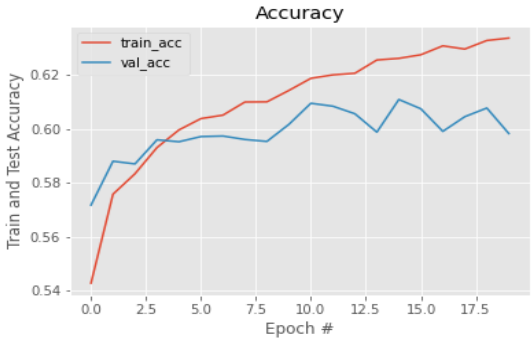
Starting from an extremely simple implementation of both dense architectures and convolutional neural networks, it was possible to analyze graphs and results, thus introducing the correct techniques to reduce errors and improve overall performance, reaching a final value of very positive accuracy and being able to confront the obtained architecture results with the one produced by a well-known model in the CNN environment.
A reflection on the project concerns the introduction of data augmentation, a technique widely used in most solutions of the Cats vs Dogs challenge, but not used in any of the implementations proposed here. The reason for this decision is closely linked to the interpretation of the delivery, and since the experimentation on the network architecture was explicitly requested, it was preferred to focus on it, as opposed to the dataset that was given. However, this does not exclude the introduction of data augmentation in future works, along with other possibilities such as transfer learning.

A. Appendices

A.1 NN 4.1: 2 Dense Layers Data Graphs

```
model = Sequential()

#input layer
model.add(Flatten(input_shape=(75,75)))

#hidden layers
model.add(Dense(64, activation="relu"))
model.add(Dense(64, activation="relu"))

#output layer
model.add(Dense(2, activation="softmax"))

model.compile (
loss = "categorical_crossentropy",
optimizer = "adam",
metrics = [tf.keras.metrics.CategoricalAccuracy(),custom_zero_loss]
)
```





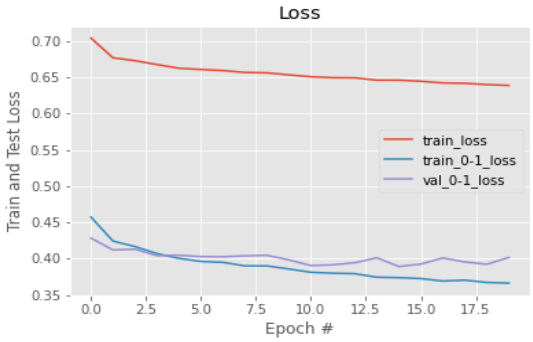
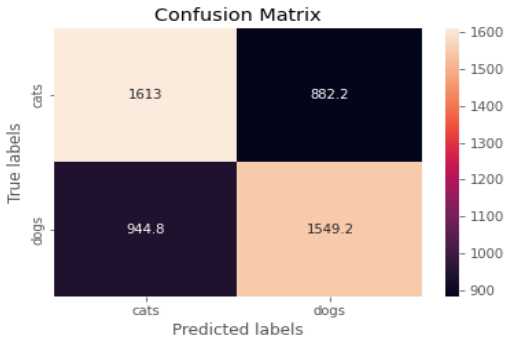
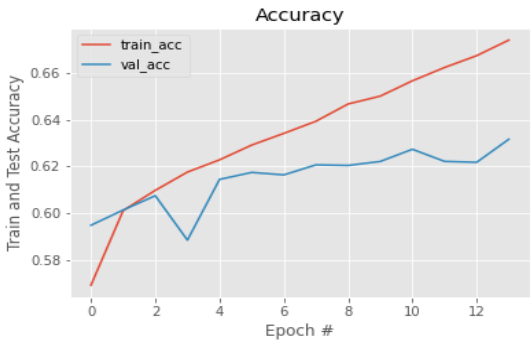


Fig. A.1: Model Code, Confusion Matrix, Mean Accuracy and Mean Loss

A.2 NN 4.1: Hypertuning 2 Dense Layers Data Graphs

```
model = Sequential()

#input layer
model.add(Flatten(input_shape=(75, 75)))

#hidden layers
hp_units=hp.Int('num_of_neurons_l1',min_value=16,max_value=512,step=16)
model.add(Dense(units=hp_units,activation='relu'))
hp_units=hp.Int('num_of_neurons_l2',min_value=16,max_value=512,step=16)
model.add(Dense(units=hp_units,activation='relu'))

#output layer
model.add(Dense(2,activation='softmax'))

model.compile (
loss = "categorical_crossentropy",
optimizer =keras.optimizers.Adam(hp.Choice('learning_rate',values=[1e-2,1e-3,1e-4])),
metrics = [tf.keras.metrics.CategoricalAccuracy(),custom_zero_loss]
)
```





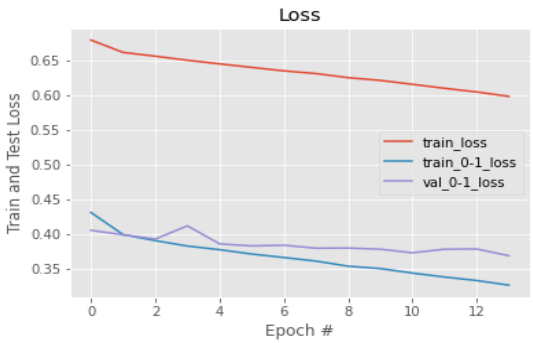


Fig. A.2: Model Code, Confusion Matrix, Mean Accuracy and Mean Loss

A.3 NN 4.1: Hypertuning M Dense Layers Data Graphs

```
model = Sequential()

#input layer
model.add(Flatten(input_shape=(75, 75)))

#hidden layer
for i in range(hp.Int('num_of_layers',2,6)):
  hp_units=hp.Int('num_of_neurons'+str(i),min_value=32,max_value=512,step=32)
  model.add(Dense(units=hp_units,activation='relu'))

#output layer
model.add(Dense(2,activation='softmax'))

model.compile (
loss = "categorical_crossentropy",
optimizer =keras.optimizers.Adam(hp.Choice('learning_rate',values=[1e-3,1e-4])),
metrics = [tf.keras.metrics.CategoricalAccuracy(),custom_zero_loss]
)
```

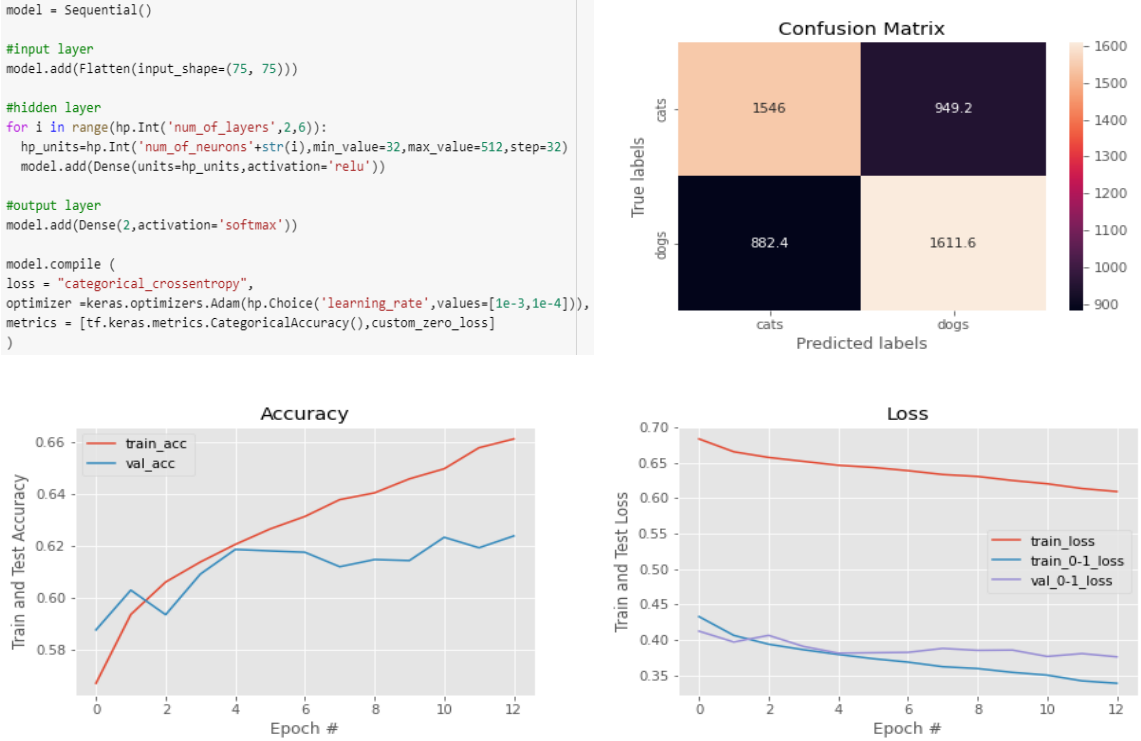


Fig. A.3: Model Code, Confusion Matrix, Mean Accuracy and Mean Loss

A.4 CNN 4.2: 1 Convolutional+MaxPooling Layer Data Graphs

```
model = Sequential()
#input layer & hidden layer
model.add (Conv2D (64,(3,3),activation='relu',input_shape = X.shape[1:]))
model.add (MaxPooling2D (pool_size=(2,2)))
model.add (Flatten ())

#output layer
model.add(Dense(2, activation="softmax"))

model.compile (
loss = "categorical_crossentropy",
optimizer = "adam",
metrics = [tf.keras.metrics.CategoricalAccuracy(),custom_zero_loss]
)
```

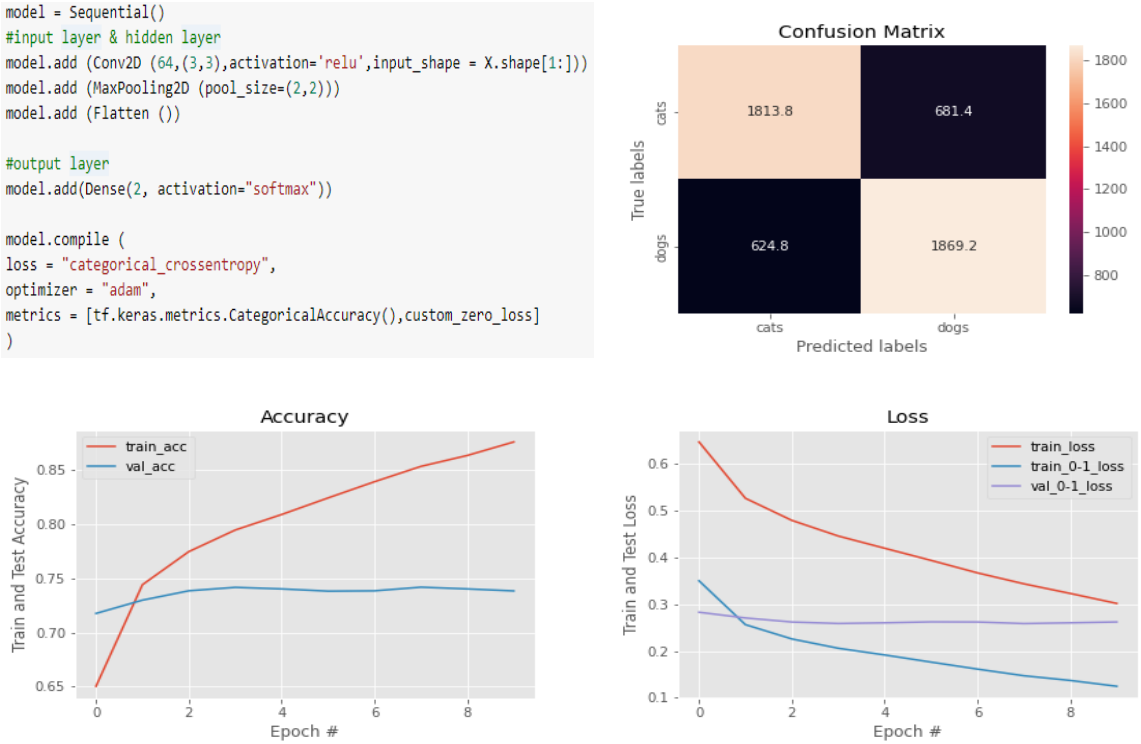
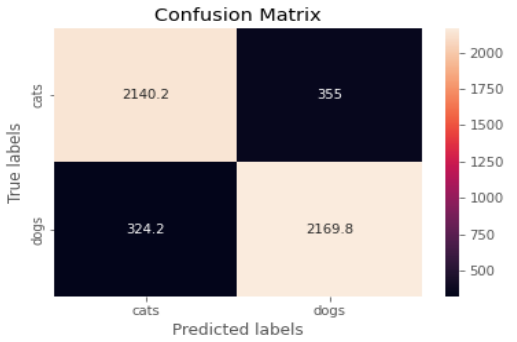
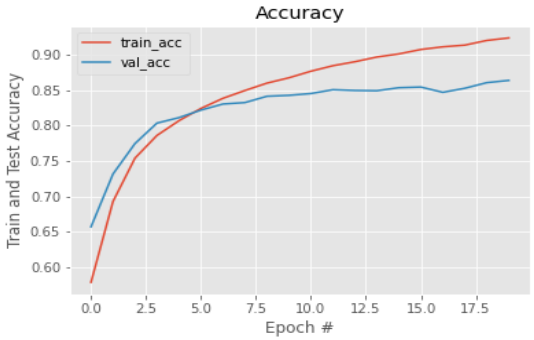


Fig. A.4: Model Code, Confusion Matrix, Mean Accuracy and Mean Loss

A.5 CNN 4.2: 3 Convolutional+MaxPooling Layer Data Graphs

```
model = Sequential()

#input layer
model.add (Conv2D (32,(3,3), activation='relu',input_shape = X.shape[1:]))

#hidden layer
model.add (MaxPooling2D (pool_size=(2,2)))
model.add (Conv2D (64,(3,3), activation='relu'))
model.add (MaxPooling2D (pool_size=(2,2)))
model.add (Conv2D (64,(3,3), activation='relu'))
model.add (Flatten ())
model.add (Dropout (0.5))
model.add (Dense(64, activation='relu'))
model.add (Dropout (0.5))

#output layer
model.add(Dense(2, activation="softmax"))

model.compile (
loss = "categorical_crossentropy",
optimizer = keras.optimizers.Adam(learning_rate=0.001),
metrics = [tf.keras.metrics.CategoricalAccuracy(),custom_zero_loss]
)
```





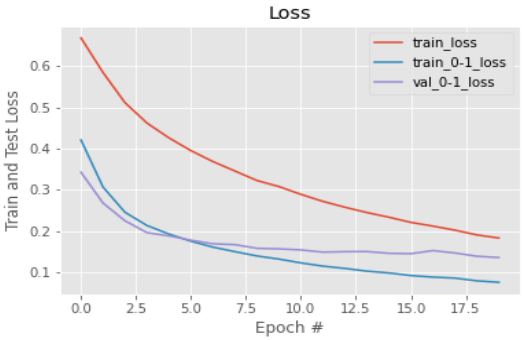
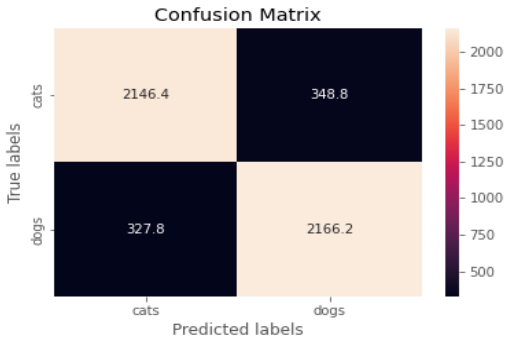
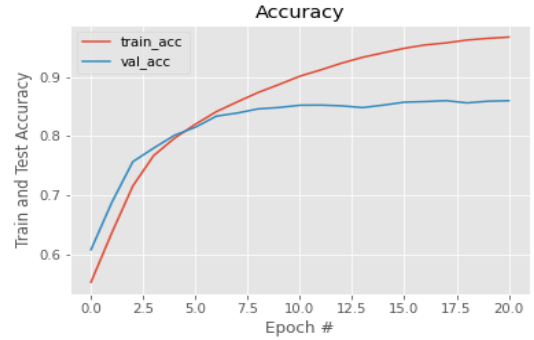


Fig. A.5: Model Code, Confusion Matrix, Mean Accuracy and Mean Loss

A.6 CNN 4.2: Hypertuning 3 Convolutional+MaxPooling Layer Data Graphs

```
model = Sequential()

hp_filters=hp.Choice('num_of_filters',values=[32,64,128])
hp_dropout=hp.Choice('dropout_value',values=[0.3,0.4,0.5])
hp_units=hp.Int('num_of_neurons',min_value=32,max_value=512,step=32)

#input layer
model.add (Conv2D (hp_filters,(3,3), activation='relu',input_shape = X.shape[1:]))

#hidden layer
model.add (MaxPooling2D (pool_size=(2,2)))
model.add (Conv2D (hp_filters*2,(3,3), activation='relu'))
model.add (MaxPooling2D (pool_size=(2,2)))
model.add (Conv2D (hp_filters*2,(3,3), activation='relu'))
model.add (Flatten ())
model.add (Dropout (hp_dropout))
model.add(Dense(units=hp_units,activation='relu'))
model.add (Dropout (hp_dropout))

#output layer
model.add(Dense(2, activation="softmax"))

model.compile (
loss = "categorical_crossentropy",
optimizer = keras.optimizers.Adam(hp.Choice('learning_rate',values=[1e-3,1e-4])),
metrics = [tf.keras.metrics.CategoricalAccuracy(),custom_zero_loss]
)
```





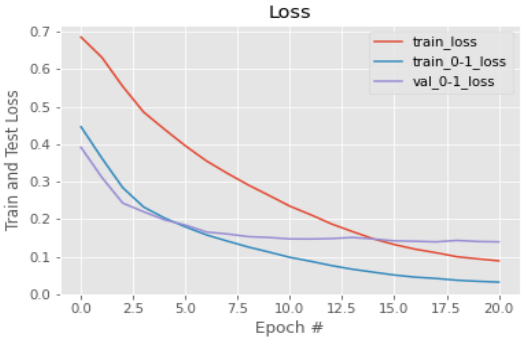


Fig. A.6: Model Code, Confusion Matrix, Mean Accuracy and Mean Loss

A.7 CNN 4.2: Hypertuning N Convolutional+MaxPooling Layer Data Graphs

```
model = Sequential()

hp_filters=hp.Choice('num_of_filters',values=[32,64])
hp_nConv=hp.Int('num_of_conv_layers',2,5)
hp_dropout=hp.Choice('dropout_value',values=[0.4,0.5])
hp_units=hp.Choice('num_of_neurons',values=[64,128])
hp_nlayers=hp.Int('num_of_layers',1,3)

for i in range(hp_nConv):
  if(i==0):
    #input layer
    model.add (Conv2D (min(hp_filters*pow(2,i),512),(3,3), activation='relu',input_shape = X.shape[1:]))
  else:
    #hidden layer
    model.add (Conv2D (min(hp_filters*pow(2,i),512),(3,3), activation='relu'))
  model.add (MaxPooling2D (pool_size=(2,2)))
model.add (Conv2D (min(hp_filters*pow(2,hp_nConv),512),(3,3), activation='relu'))
model.add (Flatten ())
model.add (Dropout (hp_dropout))
for i in range(hp_nlayers):
  model.add(Dense(units=hp_units*pow(2,hp_nlayers-1-i),activation='relu'))
  model.add (Dropout (hp_dropout))

#output layer
model.add(Dense(2, activation="softmax"))
```

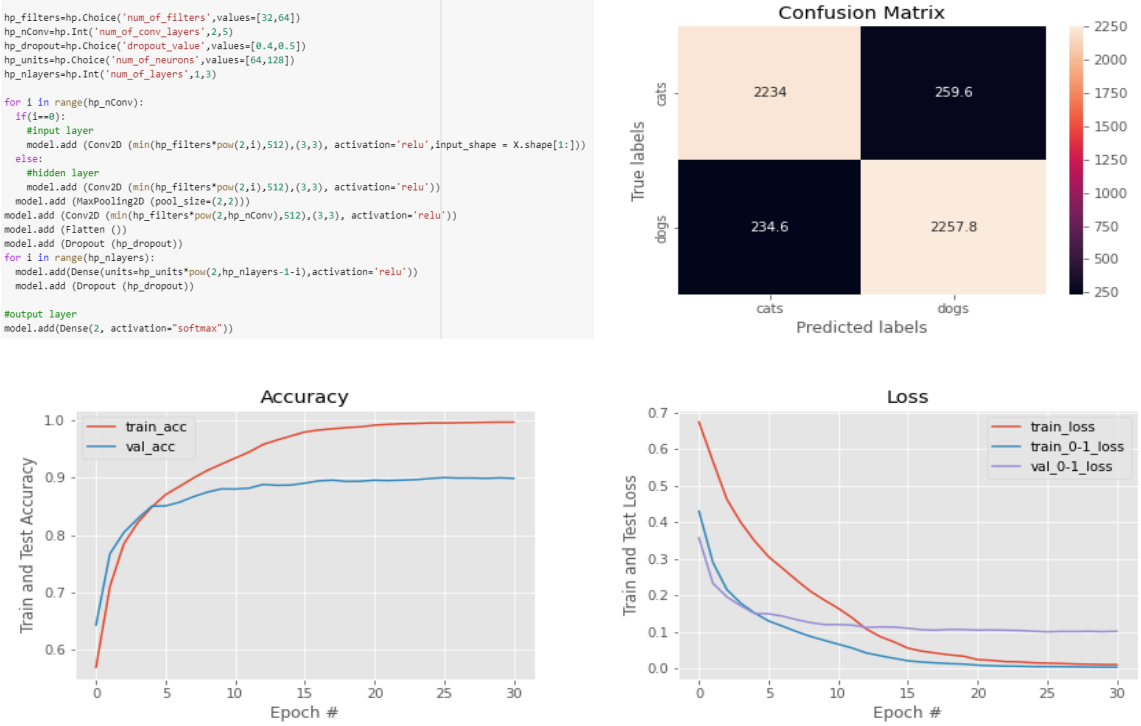


Fig. A.7: Model Code, Confusion Matrix, Mean Accuracy and Mean Loss

A.8 CNN 4.3: Hypertuning N Convolutional + MaxPooling + Batch-Normalization + Dropout Layers Data Graphs

```
model = Sequential()

hp_filters=hp.Choice('num_of_filters',values=[32,64])
hp_dropout=hp.Choice('dropout_value',values=[0.2,0.3,0.4,0.5])
hp_nlayers=hp.Int('num_of_layers',1,3)

for i in range(hp.Int('num_of_conv_layers',2,5)):
  if(i==0):
    #input layer
    model.add (Conv2D (min(hp_filters*pow(2,i),512),(3,3), activation='relu',input_shape = X.shape[1:]))
  else:
    #hidden layer
    model.add (Conv2D (min(hp_filters*pow(2,i),512),(3,3), activation='relu'))
  model.add(BatchNormalization())
  model.add(MaxPooling2D(pool_size=(2,2)))
  model.add(Dropout(hp_dropout))
model.add (Flatten ())
for i in range(hp_nlayers):
  model.add(Dense(units=512/pow(2,i),activation='relu'))
  model.add(BatchNormalization())
  model.add (Dropout (hp_dropout))

#output layer
model.add(Dense(2, activation="softmax"))

model.compile (
loss = "categorical_crossentropy",
optimizer = "adam",
metrics = [tf.keras.metrics.CategoricalAccuracy(),custom_zero_loss]
)
```

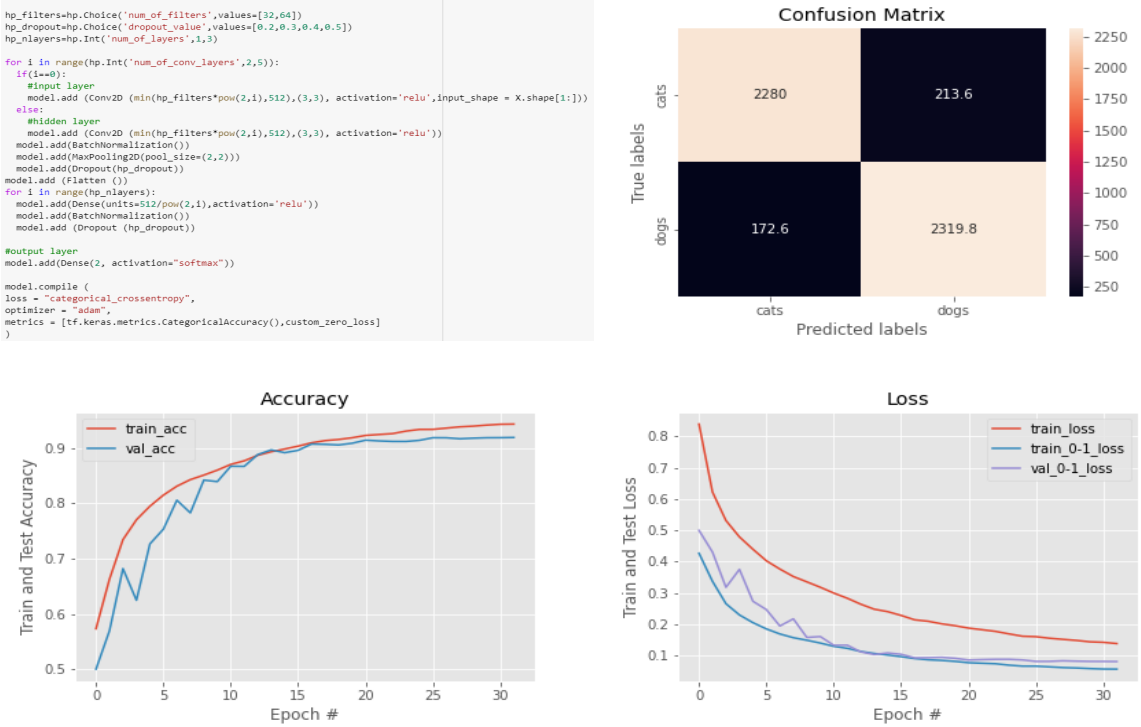
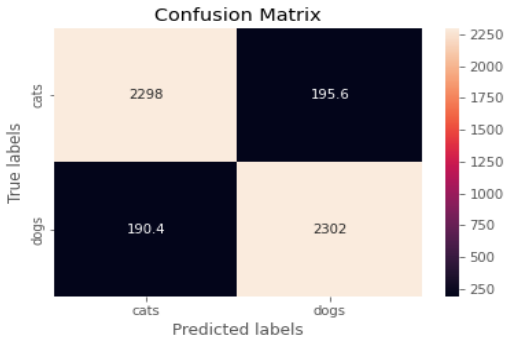
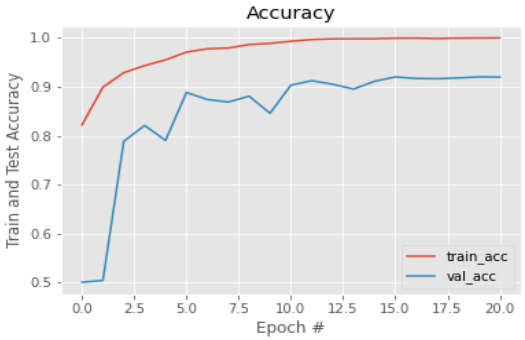


Fig. A.8: Model Code, Confusion Matrix, Mean Accuracy and Mean Loss

A.9 ResNet50 4.4 Data Graphs

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resnet50 (Functional)       (None, 2048)              23587712

 flatten (Flatten)           (None, 2048)              0

 dense (Dense)               (None, 512)               1049088

 dense_1 (Dense)             (None, 2)                 1026

=================================================================
Total params: 24,637,826
Trainable params: 24,584,706
Non-trainable params: 53,120
_________________________________________________________________
```





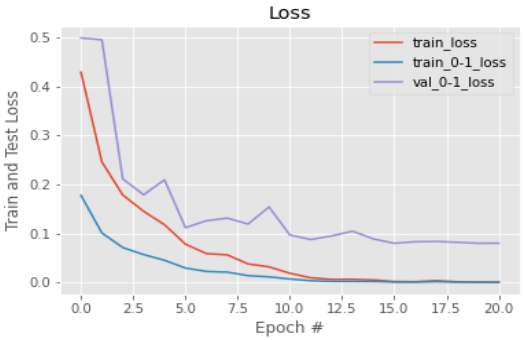


Fig. A.9: Model Code, Confusion Matrix, Mean Accuracy and Mean Loss