Statistical Methods for Machine Learning
Experimental Project
Convolutional Neural Network for Image Classification

by Enrico Darra, 21571A

Università degli Studi di Milano

Data Science for Economics

May 26rd, 2024

ABSTRACT

This report presents a practical application of Convolutional Neural Networks (CNNs) for binary image classification tasks, specifically distinguishing between images of chihuahuas and muffins. Six different CNN architectures were developed and compared, with a primary focus on layer selection over data manipulation. The architectures varied in complexity, from baseline models to an all-convolution network. Various regularization techniques, specifically Dropout layers and Batch Normalization, were employed to try preventing overfitting and improve model performance. The most promising architecture, in terms of small overfitting, underwent hyperparameter fine-tuning and a 5-fold cross-validation. The final all-convolution architecture achieved values of 0.19 for the test zero-one loss, 0.80 for test accuracy.

We declare that this material, which We now submit for assessment, is entirely our 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 our work. We 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 us nor any other person for assessment on this or any other course of study.

CONTENTS

1. INTRODUCTION

Neural Networks (NN) have a design inspired to the structure of the human brain: several interconnected neurons receive, process and send information. Convolutional Neural Networks (CNN) are a class of Neural Networks that are often used for image classifications. CNN deploys Convolutional layers that are used to extract spatial features and patterns.

In this project we aimed at developing a few Convolutional Neural Network architectures and used them for solving binary image classification of pictures of chihuahuas and muffins. We compared six different architectures, focusing on different aspects of model-building: several Dense layers vs. only one, all-Convolutions network, regularizations technique such Batch Normalization. We are fully conscious of the benefits that data augmentation brings when training CNN, but that is not the main focus of this project; in fact, we focused primarily on the choice of layers rather than data manipulation. Moreover, we performed hyperparameter fine-tuning for the architecture showing less overfitting and obtained its risk estimate through 5-fold cross-validation.

2. DATASET

For training and testing our CNN, we have used images from the *Muffin vs Chihuahua* available on Kaggle[1]. The images are quite diverse, going from actual photos of dogs or muffins to more conceptual representations of the subjects. Some pictures are displayed in Figure 1. The dataset contains 5917 images already divided in train set, 4733 images, and test set, 1184 images. The dataset is fairly balanced having approximately 46% images of muffins and 54% of chihuahuas. The train set has been subsequently divided into validation set (20%) and train set (80%). Validation set has been used for validating the models throughout the exercise while the test set has been deployed only for prediction.

Figure 1

[1] https://www.kaggle.com/datasets/samuelcortinhas/muffin-vs-chihuahua-image-classification

2.1. DATA PREPROCESSING

A fundamental step is preprocessing the images, for several reasons, such as having an input compatible with the CNNs and introducing randomness to help mitigating overfitting.

Images are initially in JPEG format, we have transformed the images into 2-D tensors, like matrices, where each value corresponds to a pixel intensity with values ranging from 0 to 255. We have decided to use grayscale to represent the images, allowing us to condense the input and have an input that is one third in dimension of the input when using the RGB scale. Moreover, in this classification exercise, we consider that the most pertinent features for classifying dogs and muffins are related to physical attributes, traits and shapes rather than colors.

In addition, the original images are of varying dimensions; to address this we resized have images to a standardized dimension: (64×64). This allows a homogenization of the input data, ensuring uniformity across all images and input of the CNNs.

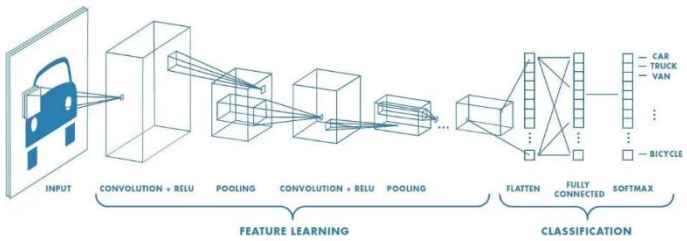
Throughout the following architectures' trainings, we consistently kept the batch size of the images at 32 images per batch.

3. CONVOLUTIONAL NEURAL NETWORKS

Convolutional Neural Networks find a great use in the sphere of image classification tasks, notably due to their efficiency at feature extraction. In practice, this is achieved but a hierarchy of features learning: early layers capture low-level features, like edges in images; deeper layers combine these to form high-level features, such as the shapes of a dog snout or a cupcake.

CNNs are fundamentally made of layers, layers themselves made of neurons; in particular, we can recognize three common elements in the architectures: the input layer, the hidden layers and the output layer (Figure 2[2]).

Figure 2



[2] Image from https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/

3.1.INPUT LAYER

The input layer serves as the entry point for data into the neural network. In the context of image classification, each neuron in the input layer represents a pixel in the image. For instance, in a grayscale image of size (64×64), there would be 4096 neurons in the input layer, with each neuron taking as input one pixel intensity value. The input layer does not perform any computation; rather, it merely passes the input data to the subsequent layers for processing.

3.2. OUTPUT LAYER

The Output layer, on the other hand, is responsible for producing the final output of the neural network. In the context of image classification, each neuron in the Output layer represents a class label; since for our classification task these labels are 2, we require only one neuron: activated if the image is a Muffin and deactivated if it is a Chihuahua. In fact, the activation of a neuron in the Output layer indicates the likelihood that the input image belongs to the corresponding class. The output layer typically employs a suitable activation function, sigmoid in our case, to ensure that the output values can be interpreted as probabilities, enabling the CNN to make predictions by selecting the class with the highest probability score.

3.3. HIDDEN LAYERS

The hidden layers constitute the core computational units of a neural network, where the intricate patterns and relationships within the input data are learned and encoded.

In the following paragraphs we describe the types of hidden layers present in our architectures and their main components; specifically, convolutional layers, pooling layers and dense layers. We follow by presenting the Dropout and Batch Normalization layers, techniques we tried for preventing overfitting.

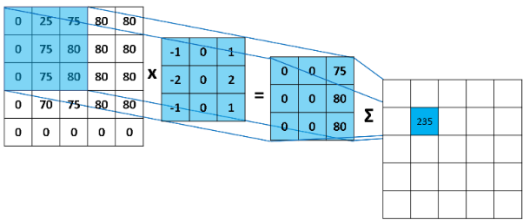
3.4. CONVOLUTIONAL LAYERS

The Convolutional layer is extremely suitable for processing grid-like data such as images. The Convolutional layer receives as input a two-dimensional tensor representing the pixel values of an image, when working with RGB images the input would be of 3 tensors. The main objective of the Convolutional Layer is to extract relevant features from the input data through a process known as convolution.

3.4.1. KERNEL

The kernel, or filter, is the key parameter in the convolution operation. It determines the type of features that the Convolutional layer learns to detect. In practice, kernels are small 2-D matrices of weights that slide spatially across the input data, performing element-wise multiplications with the local regions of the input. By tweaking the values of these weights, the convolutional layer is able to detect specific patterns. During the training process, the CNN learns to adjust the values of the kernels weights to extract relevant features (Figure 3[3]). The mathematical operation that occurs with the kernel involves element-wise multiplication of the kernel weights with the corresponding values in the input matrix plus a bias term, followed by summation of the results: $\sum_i^N w_i x_i + b$. The resulting sum forms a single entry in the feature map, such that each element in the feature map represents the outcome of a particular filter on a localized region of the input data.

Figure 3



3.4.2. STRIDE

When the kernel is moving across the input data, the parameter deciding the size of the steps made by the kernel is called stride. By adjusting the stride, we can control how much the kernel shifts between each computation, effectively influencing the size of the resulting feature map. Regarding our architectures, we employed strides of either value 1 or 2.

3.4.3. ACTIVATION FUNCTIONS

An essential element of the CNN Layers is the activation function, enabling the introduction of non-linearities into the network and allowing the CNN to learn and represent non-linear relationships between features in the data.

From a mathematical perspective, the activation function acts as a transformation applied to the output of each neuron in a neural network. The operation conducted ca n be expressed as $y = f(\sum_i^N w_i x_i + b)$, where x is the input vector, w is the weights associated with each input and b represents the bias term. After the linear transformation of the weighted sum of inputs, the activation function $f()$ introduce the non-linear transformation.

[3] Image from https://mlnotebook.github.io/post/CNN1/

In the context of the current project, we have employed the Rectified Linear Unit (ReLU) function, one of the most used activation functions for Convolutional Neural Networks. The ReLU function is as follows: $f(z) = max(0, z)$. In general, ReLU function presents numerous advantages in terms of computational efficiency due to its simplicity and it is popular because addresses the problem of the vanishing gradient[4].

Regarding the Output layer, we have employed the Sigmoid activation function. In fact, this function maps a continuous input into a probability $f(z) = \frac{1}{1+e^{-z}}$ that we interpret as the likelihood of the image having the label 1 if muffin or 0 if chihuahua.

3.5. POOLING LAYERS

Pooling layers are used for dimensionality reduction, they are useful to control overfitting and address the issue of the sensitivity of feature maps to the specific position of features in the image. In practice, a kernel slides over the input matrix and summarizes the selected values by, for example, taking maximum value or computing the average.

Moreover, even though Pooling layers are commonly used in building CNNs, some researchers believe it is possible to achieve performing models by using just Convolutional layers with a greater stride avoiding the Pooling layers[5]. For this reason, in the current project, we also tried building an architecture that make usage only of Convolutional layers with stride equal to 2. Regarding the other architectures, we have employed the Max Pooling, since it emphasizes the most significant features of the image.

3.6. DENSE LAYERS

The role of Dense layers is "combining" this information in order to generate the final prediction, through a web of connections between neurons of subsequent layers. From a mathematical perspective, neurons of the Dense layer receive one-dimension vectors as inputs, the values of these vectors are linearly transformed, using bias and weights, and then the activation function is applied. Since the Danse layer takes as inputs one-dimension vectors, while Convolution and Pooling layers' outputs are 2-dimensions tensors, a flattening operation is required.

[4] Raschka S, Mirjalili V. *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow*. Second Edition. Packt Publishing, 2017. 449-450

[5] Springenberg J T, Dosovitskiy A, Brox T, Riedmiller M. "Striving For Simplicity: The All Convolutional Net" (2015): arXiv:1412.6806

3.7. DROPOUT LAYERS

An important regularization technique of Convolutional neural Network architectures is the Dropout layer. In essence, during the training process a predefined number of randomly selected neurons of a layer are set equal to zero, hence deactivated. The main reason for using Dropout layers is to prevent overfitting by allowing several neurons to activate with similar inputs, avoiding the case of a specific neuron to be responsible for a particular pattern.

3.8. BATCH NORMALIZATION

A valuable technique for enhancing and stabilizing the learning process of Neural Networks is Batch Normalization. In practice, the inputs distributions are standardized by controlling for each mini-batch mean and variance then scaled and shifted by the addition of two learnable parameters. During the testing phase, Batch Normalization employes values for mean and variance that are an aggregation, the running average, of the values previously estimated during the training period.

Batches Normalization has a reputation of being very efficient in increasing the training speed, specifically because it helps smooth the loss function, facilitating the gradient-descent optimization algorithm in its process[6].

For these reasons, we tried implementing Batch Normalizations layers in some of our architectures.

3.9. LOSS FUNCTION

In the context of binary classification, typically the chosen loss function is binary cross-entropy, defined as $-\frac{1}{N}\sum_i^N y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)$ where $\hat{y}_i$ is the predicted label and y_i is the true label.

That is in fact the function we selected for measuring the goodness of fit of our architectures during the training phase. On the other hand, during the final testing phase we employed also the zero-one loss function, defined as $\frac{1}{N}\sum_i^N \delta_{\hat{y}_i \neq y_i}$ with $\delta_{\hat{y}_i = y_i} = \begin{cases} 0, & if\ \delta_{\hat{y}_i \neq y_i} \\ 1, & otherwise \end{cases}$

3.10. OPTIMIZER

During the training phase, the model parameters are iteratively adjusted in order to minimize the loss function: the method by which this is achieved is decided by the Optimizer.

[6]Madry A, Santurkar S, Tsipras D, Ilyas A. "How does batch normalization help optimization?" (2018): https://hdl.handle.net/1721.1/137779

Regarding this project architectures, we have employed the Adam optimizer, often a popular choice due to its computational efficiency, the low memory requirements and its adaptive learning rates, changing for each parameter.[7]

4. ARCHITECTURES

4.1. BASELINE MODEL

As our first architecture, we coded a very basic model that we used as baseline. This consists of 3 main layers: Convolution, Max Pooling and Output. The Convolution layer is configured with 32 filters of size (3×3), the ReLU activation function and no padding. Following, the Max Pooling window has size 2×2. Finally, after a Flattening operation, a Dense layer with a single neuron and Sigmoid activation function is tasked to classify the image and produce the output prediction. As seen in Table 1, we have a total of 31073 trainable parameters. This model, like all the following ones, was trained in 20 epochs using an early stopping of 5 epochs.

Table 1: Baseline model[8]

LAYER	OUTPUT SHAPE	PARAMETERS
Conv2D	(62, 62, 32)	320
MaxPooling	(31, 31, 32)	0
Flatten	(30752)	0
Dense	(1)	30 753
TOT parameters: 31 073 Trainable parameters: 31 073 Non-trainable parameters: 0		

4.1.1. RESULTS

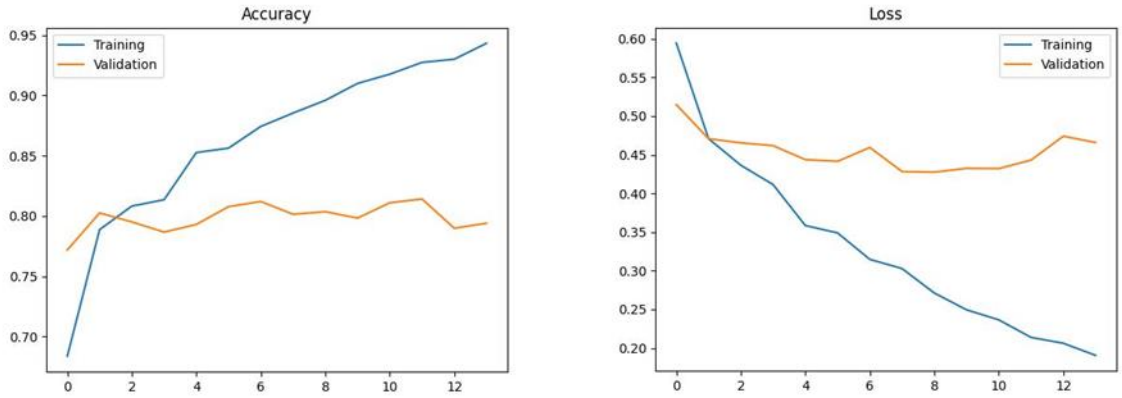
As seen from Figure 4, training accuracy and loss respectively increase and decrease quite smoothly. On the other hand, validation curves show very little improvement over the epochs, almost exhibiting no clear upward and downward trends. Final validation loss and accuracy are respectively 0.46 and 0.79 at epoch 14, since the early stopping blocked any further training, seeing there were not any improvements in regards of validation loss.

Specifically, the great divergence between training loss (accuracy) and validation loss (accuracy) indicates the presence of overfitting. In the following architectures we tried addressing the issue of overfitting and improving validation accuracy and loss.

[7] Kingma D, Ba J L. "Adam: A Method For Stochastic Optimization" (2015): arXiv:1412.6980

[8] Note: when not explicitly present in the table, the Activation function is assumed to be part of the Convolutional layer.

Figure 4: Baseline model Accuracy and Loss



4.2. BASELINE MODEL 2.0

As our first attempt to tackle the problem of overfitting, we reduced the initial learning rate of the ADAM optimizer from 0.001 to 0.0001. Fundamentally, a smaller learning rate can help the network prevent overshooting the minimum and getting "stuck", such as we have seen in the first Baseline model. Besides this change in the learning rate, we added a Dense layer with 64 neurons and ReLU function (Table 2). The rest of the architecture matches the previous Baseline model.

Table 2: Baseline model 2.0

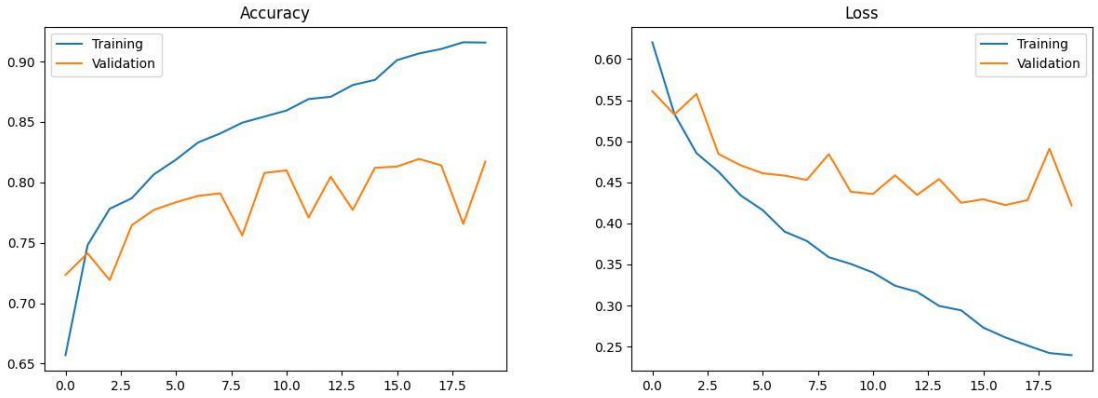
LAYER	OUTPUT SHAPE	PARAMETERS
Conv2D	(62, 62, 32)	320
MaxPooling	(31, 31, 32)	0
Flatten	(30 752)	0
Dense	(64)	1 968 192
Dense	(1)	65
TOT parameters: 1 968 577 Trainable parameters: 1 968 577 Non-trainable parameters: 0		

4.2.1. RESULTS

As observed from Figure 5, there is an improvement in terms of validation accuracy and loss. We can see that validation curves are slightly closer to the training curves; specifically, validation accuracy shows a modest upward trend terminating at a value around 0.81 for the 20^{th} epoch while validation loss displays a minor downward trend terminating at 0.42.

In the following architectures, we aimed at addressing overfitting, still strongly present, of the validation curves and upgrading the accuracy and loss final values.

Figure 5: Baseline model 2.0 Accuracy and Loss



4.3. FIRST ARCHITECTURE

In the following architecture (Table 3), we added a second Convolution layer and Max Pooling layer, preceded by a Dropout layer that deactivates 25% of the neurons. For the second Convolutional layer we increased the kernel size to 5x5. Regarding the second Convolution, we increased the number of filters to 96 such that this layer would capture higher-level features. Furthermore, the Dense layer preceding the Output Dense layer is formed by 128 neurons and the ReLU activation function. Finally, we included a second Dropout layer, deactivating 50% of neurons, right before the Output layer. Overall, this architecture results in having 756 705 trainable parameters.

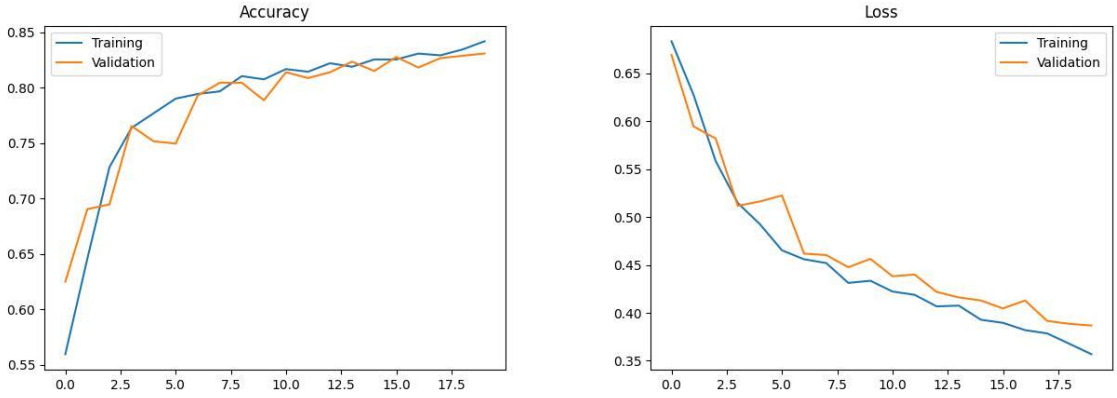
Table 3: First architecture

LAYER	**OUTPUT SHAPE**	**PARAMETERS**
Conv2D	(62, 62, 64)	640
MaxPooling	(31, 31, 64)	0
Dropout (0.25)	(31, 31, 64)	0
Conv2D	(14, 14, 96)	153 696
MaxPooling	(7, 7, 96)	0
Dropout (0.25)	(7, 7, 96)	0
Flatten	(4 704)	0
Dense	(128)	602 240
Dropout (0.5)	(128)	0
Dense	(1)	129
TOT parameters: 756 705 Trainable parameters: 756 705 Non-trainable parameters: 0		

4.3.1. RESULTS

Observing Figure 6, we can see the validation curves follow a trend much more adherent to the training curves. Nonetheless, minor fluctuations of the validation loss and accuracy are still present. Confronting the present model to our latest baseline model, we can appreciate a great improvement in regards of the diminished overfitting, meaning the smaller divergence between training and validation curves. Finally, at the latest epoch, validation loss and accuracy are respectively 0.38 and 0.83. Overall, a second Convolutional layer seems to help the network to learn better.

Figure 6: First architecture Accuracy and Loss



4.4. SECOND ARCHITECTURE

Regarding the following architecture, we have decided to use a different approach: adding more Dense layers to see if there is space of improvements when the CNN combines the features previously learnt.

We reduced the number of filters for the Convolutional layers, of which there are two, increasing the kernel size and increasing the stride for the second. Primarily, we increased the number of Dense layers with progressively diminishing number of neurons and separated them by Dropout layers. From Table 4, it is possible to see how drastically more trainable parameters there are, specifically 1 400 033, compared to the previous architecture.

Table 4: Second architecture

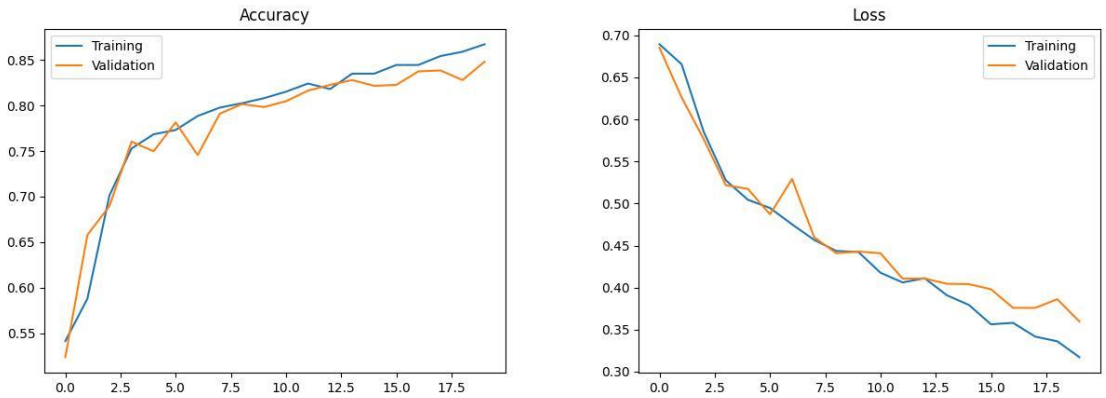
LAYER	OUTPUT SHAPE	PARAMETERS
Conv2D	(62, 62, 64)	640
MaxPooling	(31, 31, 64)	0
Dropout (0.2)	(31, 31, 64)	0
Conv2D	(14, 14, 96)	153 696

MaxPooling	(7, 7, 96)	0
Dropout (0.2)	(7, 7, 96)	0
Flatten	(4 704)	0
Dense	(256)	1 204 480
Dropout (0.5)	(256)	0
Dense	(128)	32 896
Dropout (0.3)	(128)	0
Dense	(64)	8 256
Dropout (0.1)	(64)	0
Dense	(1)	65
TOT parameters: 1 400 033 Trainable parameters: 1 400 033 Non-trainable parameters: 0		

4.4.1. RESULTS

Based on the accuracy and loss graph in Figure 7, it appears that the model's performance is quite good compared to the baseline architectures, as validation accuracy and loss are quite close to the training curves. Primarily, we can observe how validation accuracy and loss are fairly near to the training curves up to epoch 12, after they start diverging, indicating a possible overfitting scenario. Final validation accuracy and loss are respectively 0.84 and 0.35.

Figure 7: Second architecture Accuracy and Loss



4.5. THIRD ARCHITECTURE

In regards of the third architecture, we took inspiration from the article of Springenberg, J.T. et al. (2014)[9]: essentially, we built the CNN without the use of Max Pooling layers. Instead, we employed a stride equal to 2 in order to reduce the size of the image representation.

[9] Springenberg J T, Dosovitskiy A, Brox T, Riedmiller M. "Striving For Simplicity: The All Convolutional Net" (2015): arXiv:1412.6806

From Table 5 we can observe the structure of this architecture. We deployed 3 Convolutional layers with increasing number of filters, same kernel size of (3×3), with Dropout layers in between the Convolutional layers. We employed only one Dense layer with 128 neurons, preceding one last Dropout layer.

With this architecture, we aimed at testing whether, for this project purposes, a model without Pooling layers could match, if not outperform, the architectures using Max Pooling layers.

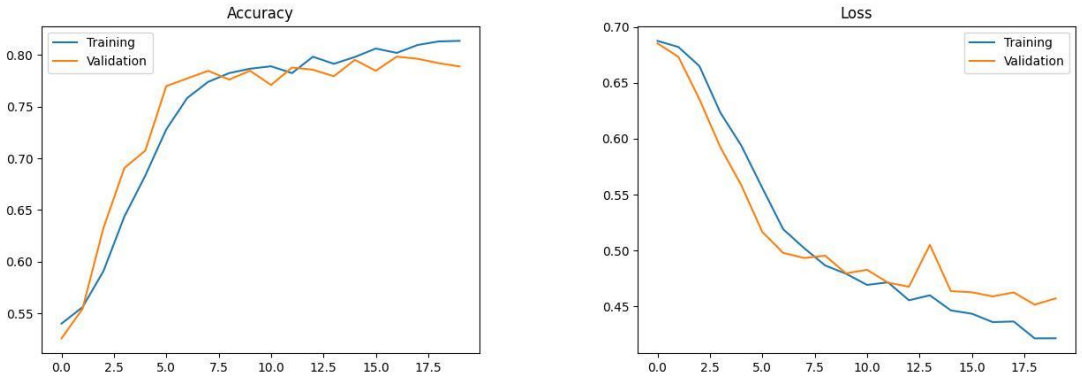
Table 5: Third architecture

LAYER	**OUTPUT SHAPE**	**PARAMETERS**
Conv2D	(31, 31, 32)	320
Dropout (0.2)	(31, 31, 32)	0
Conv2D	(15, 15, 48)	13 872
Dropout (0.2)	(15, 15, 48)	0
Conv2D	(7, 7, 64)	27 712
Dropout (0.2)	(7, 7, 64)	0
Flatten	(3 136)	0
Dense	(128)	401 536
Dropout (0.5)	(128)	0
Dense	(1)	129
TOT parameters: 443 569 Trainable parameters: 443 569 Non-trainable parameters: 0		

4.5.1. RESULTS

Looking at Figure 8, we can observe how the validation curves slightly diverge from the training curves across the whole 20 epochs, but still following a close trend. Moreover, validation accuracy and loss present some fluctuations, particularly pronounced at epoch 13th. Furthermore, from epoch 12th we can appreciate how the divergency between validation and training curves accentuates, particularly visible in the loss plot. At epoch 20th, validation loss is 0.45 while validation accuracy is 0.78.

Figure 8: Third architecture Accuracy and Loss



4.6. FOURTH ARCHITECTURE

For our fourth model, we decided to try integrating Batch Normalization hoping for better performance of the CNN; in fact, as discussed previously, the benefits of this technique should be multiple.

From Table 6, we can see the model architecture. We implemented 3 Convolutional layers with the kernel size of (3×3) of increasing number of filters, before the activation functions are activated, we apply the aforementioned normalization. We employed Max Pooling layers as well. Continuing, only one Dense layer with 256 neurons is utilized and, for this layer too, we deployed a Batch Normalization layer. Finally, a Dropout layer precedes the Output layer.

Table 6: Fourth architecture

LAYER	OUTPUT SHAPE	PARAMETERS
Conv2D	(62, 62, 32)	320
Batch Normalization	(62, 62, 32)	128
Activation	(62, 62, 32)	0
MaxPooling	(31, 31, 32)	0
Conv2D	(29, 29, 64)	18 496
Batch Normalization	(29, 29, 64)	256
Activation	(29, 29, 64)	0
MaxPooling	(14, 14, 64)	0
Conv2D	(12, 12, 96)	55 392
Batch Normalization	(12, 12, 96)	384
Activation	(12, 12, 96)	0
MaxPooling	(6, 6, 96)	0
Flatten	(3 456)	0
Dense	(256)	884 992
Batch Normalization	(256)	1024
Activation	(256)	0

Dropout (0.5)	(256)	0
Dense	(1)	257
TOT parameters: 961 249 Trainable parameters: 960 353 Non-trainable parameters: 896		

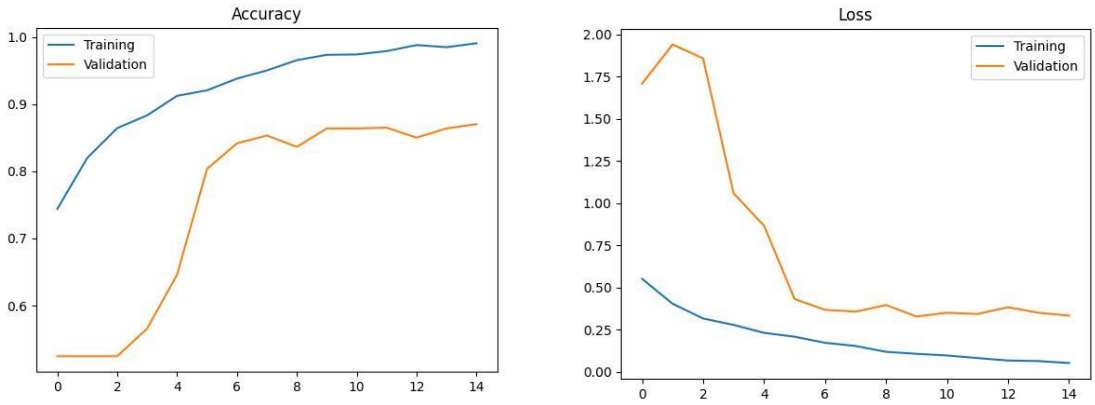
4.6.1. RESULTS

Observing Figure 9, we can clearly see this is the worst architecture we have tried so far in terms of divergency between the validation and training curves. Specifically, after a notable peak in the validation loss (a trough for the validation accuracy) at the 1st epoch, the validation loss follows a smooth downward trend up until epoch 6th, stabilizing itself for the rest of the epochs. The early stop gets activated; hence, the final epoch is only the 15th. Here, the final values for validation loss and accuracy are respectively 0.33 and 0.87.

Overall, in terms of numeric values, the current architecture is doing better than the models previously tested: the great discrepancy between validation and training curves is due to the great improvement of the training accuracy and loss. Regardless, the fact that validation curves seem to stabilize at such an early epoch may be seen as an issue. Moreover, Batch Normalization's different behavior for the training and testing stages can present problems. Specifically, while for the training period the mean and variance by which the batch is standardized are, in fact, based on this same batch, during the inference stage the statistics used are their running average. Some argues that this running average may lag and result in a discrepancy.[10] Furthermore, we tested that increasing the batch size, doubling it or increasing it to three times the value we have consistently used (32), would yield results worse than the baselines architectures.

[10] "*The Danger of Batch Normalization in Deep Learning*": https://www.mindee.com/blog/batch-normalization#:~:text=However%2C%20it%20is%20too%20costly,on%20a%20batch%20of%20data.&text=This%20works%20well%20in%20practice,do%20not%20make%20sense%20anymore.
"*Curse of Batch Normalization*": https://towardsdatascience.com/curse-of-batch-normalization-8e6dd20bc304

Figure 9: Fourth architecture Accuracy and Loss



4.7. FIFTH ARCHITECTURE

In the following architecture, we decided to combine some aspects of the previous model we have tried. In practice, we built a model that uses only the Convolutional layers for downsampling, we implemented Batch Normalization and employed more than one Dense layer. Our goal was observing how different aspects of the architectures, that allegedly should improve performance, would behave together. Regardless of the results about Batch Normalization we got from the Fourth model, we decided to give it another chance.

Specifically, as can be observed from Table 7, the architecture starts with 3 Convolution layers, all with kernel size (3×3). There are not any Pooling layers, Convolutional layers have their strides equal to 2; these Convolutions are each one followed by a Batch Normalization layer and a Dropout layer. Follows 3 Dense layers, each one is batch-normalized and there is a Dropout layer before the Output layer.

Table 7: Fifth architecture

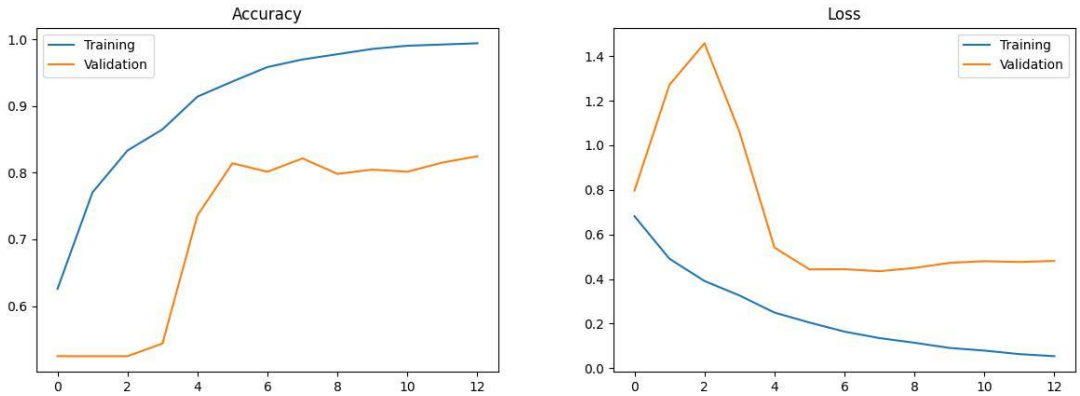
LAYER	OUTPUT SHAPE	PARAMETERS
Conv2D	(31, 31, 32)	320
Batch Normalization	(31, 31, 32)	128
Activation	(31, 31, 32)	0
Conv2D	(15, 15, 48)	13 872
Batch Normalization	(15, 15, 48)	192
Activation	(15, 15, 48)	0
Conv2D	(7, 7, 64)	27 712
Batch Normalization	(7, 7, 64)	256
Activation	(7, 7, 64)	0
Flatten	(3 136)	0
Dense	(256)	803 072

Batch Normalization	(256)	1024
Activation	(256)	0
Dense	(128)	32 896
Batch Normalization	(128)	512
Activation	(128)	0
Dense	(64)	8 256
Batch Normalization	(64)	256
Activation	(64)	0
Dropout (0.5)	(64)	0
Dense	(1)	65
TOT parameters: 888 561 Trainable parameters: 887 377 Non-trainable parameters: 1 184		

4.7.1. RESULTS

Looking at Figure 10, we observe plots somewhat resembling the ones from the Fourth architecture. We can see the training curves smoothly approaching the values of 0.05 and 0.99 respectively for loss and accuracy at the 13th epoch. On the other hand, the validation loss peaks at the 2nd epoch, sharply decreases then stabilizes after the 5th epoch; similarly, validation accuracy is quite low for the first 3 epochs, then it improves steeply up until stabilizing after the 5th epoch. Final validation loss and accuracy are respectively 0.48 and 0.82. Overall, we can infer that the great discrepancy between validation and training curves is due to the Batch Normalization; while, comparing this model to the Fourth architecture, we can see that the addition of a second and third Dense layer did not produce any improvement, hence making this model overcomplicate.

Figure 10: Fifth architecture Accuracy and Loss



4.8. FINAL ARCHITECTURE

At last, after our several architectures trials, we decided to build a simpler architecture as the final one. We observed that, for our case, the usage of Batch Normalization leads to a great divergence between training and validation results. Moreover, we could infer that the use of several Dense layers may not be helping the model better mapping the learnt features to the final output. Overall, we decided to build the Final architecture using the Third architecture structure. Even if this architecture was not the one with the lowest validation loss ad highest validation accuracy, the main reason that brought us to choose this as the final was that this presented the smallest divergence between the validation and training curves. In fact, due to the computational limitations of our project, we decided to focus more on reducing overfitting rather than getting smaller loss values, when it is fairly clear that the latter can be achieved by increasing the pixels of input data and using RGB format.

As seen from figure Table 5, this model is composed of 3 Convolutional layers, each one followed by Max Pooling and a Dropout layer. For the aforementioned reasons, we employed only Dropout layers as regularization technique. There is only one Dense layer with ReLU function before one last Dropout layer, preceding the Output layer.

4.8.1. HYPERPARAMETERS TUNING

For the Final architecture, we performed hyperparameters tuning. Specifically, we looked for the best values of number of filters used in the Convolutional layers; probability for the Dropout layers; number of units for the Dense layer. As optimizer, we employed the Bayesian Optimizer. From table 8 we can observe the model with the best hyperparameters found by the optimizer.

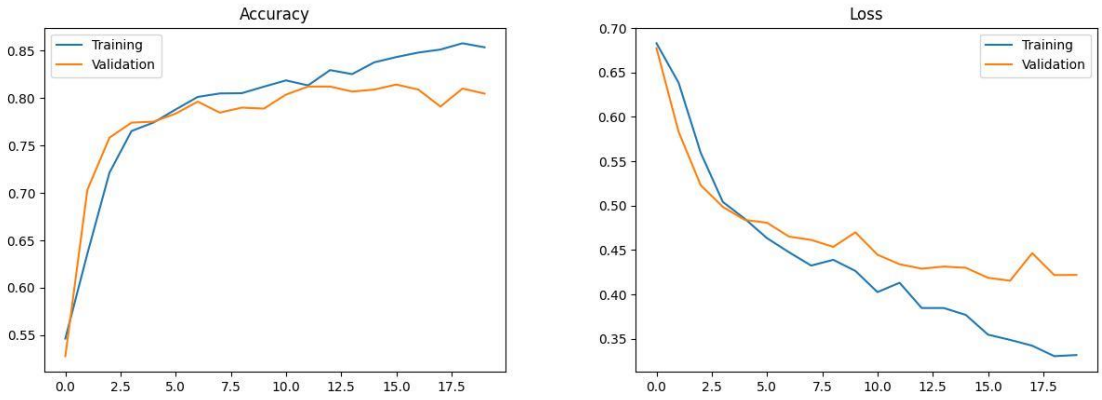
Table 8: Final architecture tuned

LAYER	OUTPUT SHAPE	PARAMETERS
Conv2D	(31, 31, 64)	640
Dropout (0.1)	(31, 31, 64)	0
Conv2D	(15, 15, 96)	55 392
Dropout (0.1)	(15, 15, 96)	0
Conv2D	(7, 7, 128)	110 720
Dropout (0.1)	(7, 7, 128)	0
Flatten	(6 272)	0
Dense	(256)	1 605 888
Dropout (0.5)	(256)	0
Dense	(1)	257
TOT parameters: 1 772 897 Trainable parameters: 1 772 897 Non-trainable parameters: 0		

4.8.2. RESULTS

Looking at Figure 11, we can observe the training and validation results for the tuned Final architecture. Specifically, in regard to accuracy, both training and validation curves start with a steep increase up to epoch 5; after that, the training curve continues growing at a slower rate while validation accuracy begins to plateau not showing an upward trend, meaning the model is starting to converge and the additional epochs of training are not so helpful. Because of this, we see a discrepancy between validation accuracy and training accuracy; hence the presence of overfitting. Regarding the loss plot, we can draw similar conclusions: sharp declining of both training and validation curves in the first epochs; training curve continuing its descent at a slower rate while the training one, still showing a timid downward trend, diverges from the former showing the presence of overfitting. The values of validation accuracy and loss at the last epoch are respectively 0.80 and 0.42.

Figure 11: Final Architecture Accuracy and Loss



4.8.3. 5-FOLD CROSS VALIDATION

In order to produce a more robust overview of our Final architecture performance, we conducted a 5-fold cross-validation. Briefly, k-fold cross-validation is a resampling technique where the dataset is split into k folds; the model is trained k times on k-1 folds, then evaluated using the hold-out k-th fold as the validation set. By adopting the 5-fold cross-validation, we are hence able to evaluate our model 5 times on independent test sets, obtaining performance results less depending on variance.

In Figure 12 we can observe the validation and training curves of the 5 folds, while in Figure 13 we presented the average of the 5 training and validation curves. Primarily, we can see

that our model validation curves have a sharp improvement over the first 4 epochs, after that both validation accuracy and loss tend to almost stabilize with respectively small upward and downward trends. Moreover, we can observe an increasing discrepancy between training and validation curves starting after the 4th epoch, indicating the presence of overfitting. Average validation loss and accuracy across the 5 folds, at the last epoch, are respectively 0.40 and 0.81.

Figure 12: Final architecture 5-fold CV Accuracy and Loss

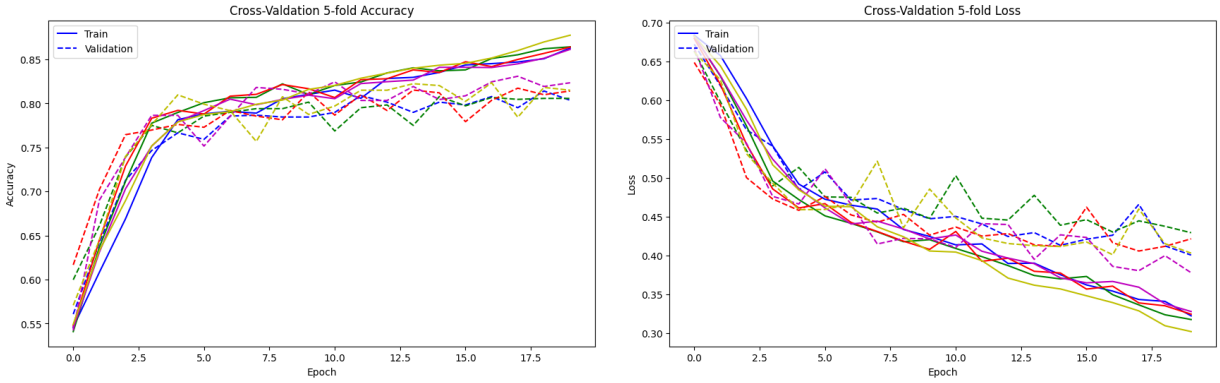
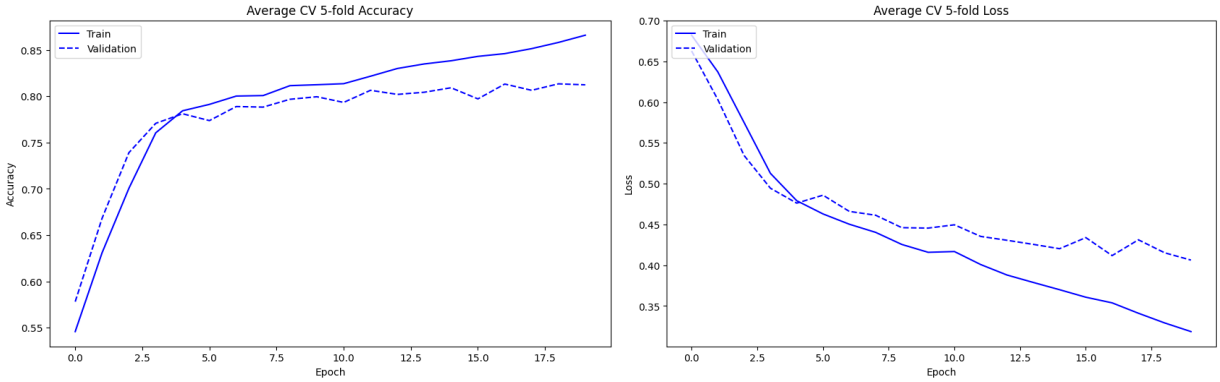


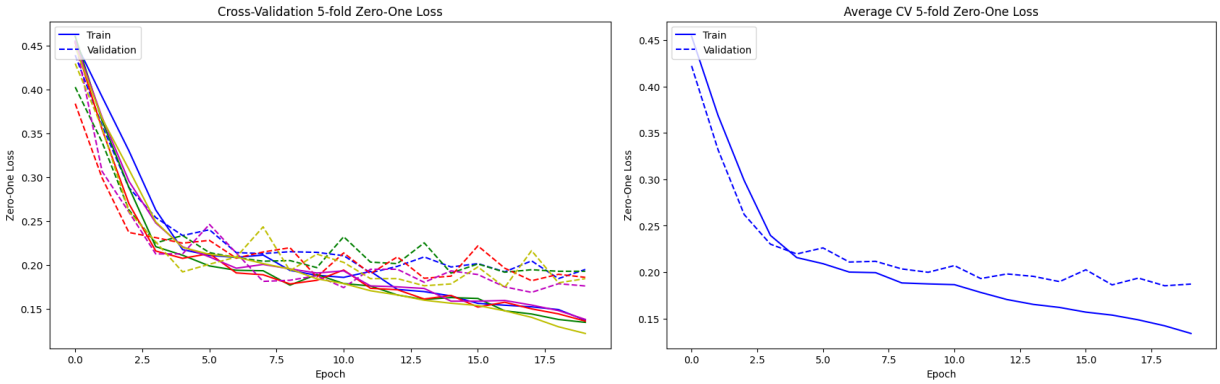
Figure 13: Final architecture 5-fold CV Average Accuracy and Loss



4.8.3.1. ZERO-ONE LOSS

Moreover, observing Figure 14, we can see the results of the tuned Final architecture in terms of the zero-one loss. Similarly to the previous plots, we observe a sharp declining of both validation and training curves up to the 4th epoch, after which the training loss decreases with a lower trend while the validation loss appears to almost be stable. These create a discrepancy between the two curves that is imputable to overfitting. Average training and validation zero-one loss, across the 5 folds, are respectively 0.13 and 0.18.

Figure 14: Final architecture 5-fold CV Zero-One Loss and Average Zero-One Loss



4.8.4. PREDICTION

At last, we tested our tuned Final architecture on the test set. Specifically, we obtained a test loss of 0.42, test accuracy of 0.80 and test zero-one loss of 0.19.

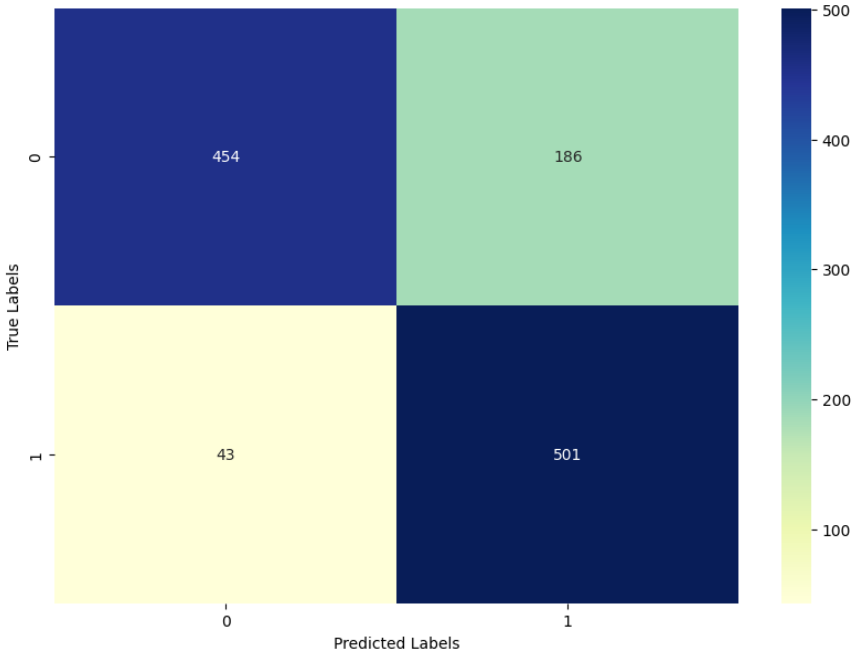
Moreover, in Table 9 we can observe the classification report. For Precision, we can observe values of 0.88 and 0.74 respectively for Chihuahua and Muffin; since Precision is the ratio is of correctly predicted labels over the total predicted labels, we can see that when our model is more likely to be correct when it identifies the Chihuahua category. Regarding Recall, since this value measures the correctly predicted labels over all labels, of that category obviously, we can observe that, with values respectively of 0.74 and 0.88 for Chihuahua and Muffin, our model does a better job when predicting the Muffin category. F1-Score, defined as the weighted average of the two former measures, is 0.81 both for Chihuahua and Muffins.

In Figure 15 we can see the confusion matrix, meaning the predicted labels against the true labels. Essentially, we find a confirmation of what stated earlier: our model is better at identifying the Muffin category (higher Recall), but when it predicts a label to be Muffin it is less likely to be correct to when it predicts a label to be Chihuahua.

Table 9: Classification Report

	Precision	**Recall**	**F1-Score**	**Support**
0 - Chihuahua	0.88	0.74	0.81	640
1 - Muffin	0.74	0.88	0.81	544

Figure 15: Confusion Matrix



5. CONCLUSION

The task of developing a Convolutional Neural Network has been a process of continuous learning, trials, improvements and several failures. It is indeed not a simple and linear task, some might say it is a reminder of the famous *no free lunch* theorem, meaning there is no one-fits-all solution that is generally better than the rest, it is in fact a process of trials and errors.

We started with a baseline model, experimented with different architectures, hyperparameters and regularization techniques trying to combat overfitting and improve the model performance.

We saw that employing Dropout layers helps counter overfitting. We observed that adopting several Dense layers does not yield significant results and might instead overcomplicate the model. We experimented with an all-convolution architecture, inspired by Springenberg J. T. *et al*. (2014), and decided to bring an architecture of this type as the final one. We tried adopting Batch Normalization but, in our case, this led to a great divergence between training and testing values. We combined several of the aforementioned aspects into one architecture, only to find results similar to the one mentioned above and an overcomplex model. Finally, we tuned the hyperparameters of our all-convolution final architecture, performed 5-fold

Cross-validation, obtaining final test values of 0.19 for the zero-one loss, while, using binary cross-entropy as the loss function, we got 0.80 for the test accuracy and 0.42 for the test loss.

5.1. LIMITATIONS AND FUTURE WORK

Due to computational limitations, we were able to process the image input only of size (64×64). This posed a serious constraint since several images have small details that could be lost; moreover, we could not stack too many Convolutional and Pooling layers together or the feature map would become too small and not meaningful.

Moreover, the Dataset used in this report contains some "noisy data", meaning unrelated images that are neither muffin nor chihuahua. By cleaning the dataset of these, we might expect a better model performance.

Furthermore, it is quite clear that the next step for improving our model performance would be, besides changing the actual architectures, implementing techniques of data augmentations to increase the training set. Finally, other regularization and normalization techniques, such as Layer Normalization, could be implemented to see if these are more suitable.

6. BIBLIOGRAPHY

- Raschka S, Mirjalili V. *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow*. Second Edition. Packt Publishing, 2017.
- Springenberg J T, Dosovitskiy A, Brox T, Riedmiller M. “Striving For Simplicity: The All Convolutional Net” (2015): arXiv:1412.6806
- Madry A, Santurkar S, Tsipras D, Ilyas A. "How does batch normalization help optimization?" (2018): https://hdl.handle.net/1721.1/137779
- Kingma D, Ba J L. “Adam: A Method For Stochastic Optimization” (2015): arXiv:1412.6980
- “*The Danger of Batch Normalization in Deep Learning*”: https://www.mindee.com/blog/batch-normalization#:~:text=However%2C%20it%20is%20too%20costly,on%20a%20batch%20of%20data.&text=This%20works%20well%20in%20practice,do%20not%20make%20sense%20anymore.
- “*Curse of Batch Normalization*”: https://towardsdatascience.com/curse-of-batch-normalization-8e6dd20bc304