MUFFIN VS CHIHUAHUA

Statistical Method for Machine Learning

submitted by

Michele Ghirardelli
(16441A)

Department of Computer Science ”Giovanni Degli Antoni”
University of Milan
May 2024

DECLARATION

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I/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 me/us or any other person for assessment on this or any other course of study.

Place : .... Signature of student : ....

Date : May 29, 2024 Name of student : Michele Ghirardelli

Chapter 1

INTRODUCTION TO THE PROJECT

The project aim to classify a dataset composed of photos of muffins and chihuahuas using Convolutional Neural Network.

1.1 OBJECTIVE OF THE PROJECT

- Experiment with different network architectures (at least 3) and training hyperparameters.
- Use 5-fold cross validation to compute your risk estimates.
- Documenting the influence of the choice of the network architecture and the tuning of the hyperparameters on the final cross-validated risk estimate.

1.2 PROJECT CONSTRAINTS

Images must be transformed from JPG to RGB (or grayscale) pixel values and scaled down. While the training loss can be chosen freely, the reported cross-validated estimates must be computed according to the zero-one loss.

1.3 WHAT ARE CONVOLUTIONAL NEURAL NETWORKS?

A particular type of feed-forward Neural Networks that are used for structured grid data, like images. Unlike the normal Neural Networks, they contain a

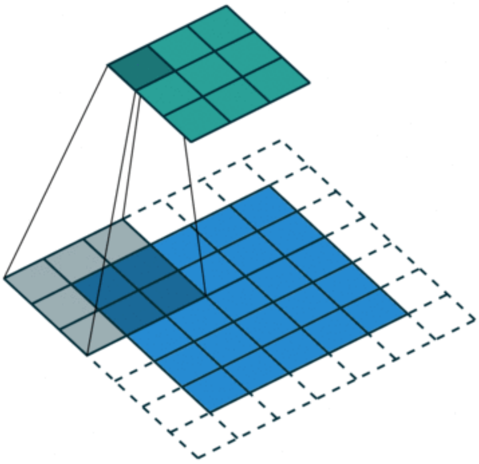


Figure 1.1: Convolution filter computation

particular type of layer called convolution layer. These layers are made by learnable filters (also called kernels), each filter scans the images by width and by height. The output is:

$$(S*K)(i,j) = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} S(i+m, j+n)\cdot K(m,n) \tag{1.1}$$

- S is the input matrix
- K is the kernel matrix
- (i,j) represents the position in the output feature map.
- M and N are the dimensions of the kernel matrix.

In other words we sum for each element (i,j) of the output feature map, the product of the kernel matrix and the corresponding sub-matrix of the image, Figure 1.1. The convolution layer has also an activation function applied element-wise to the output feature map.

There is also another layer used called Max Pooling, that is the most common pooling used. It selects the maximum value from a zone of usually 2x2 pixels. At the end of the CNN we are going to have fully connected layers usually with a sigmoid or soft-max function (for classification tasks) to produce the final prediction probabilities.

1.4 CONFIGURATION AND CODE

All the experiments will be run on Google Colab and the code can be found on this repo

Chapter 2

DATA PREPARATION

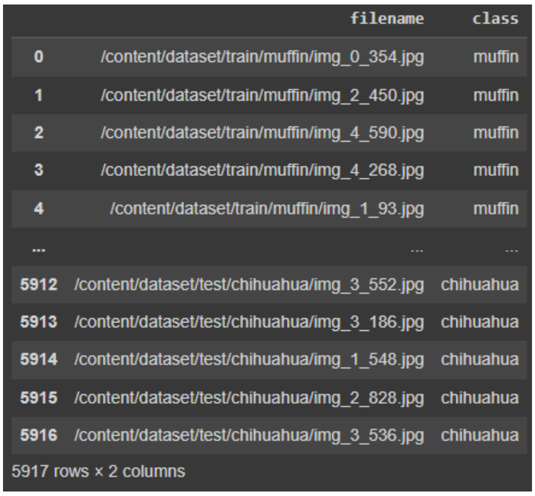
The dataset is composed of 5917 images divided in:

- Muffins 46% of the dataset, Figure 2.2
- Chihuahuas 54% of the dataset, Figure 2.3

A single folder has been created containing all the test and training files to create randomize folds for the 5-Fold estimation. A data frame with two columns was created, as shown in Figure 2.1. One column contains the filename path and the other column contains the class of the file. This data frame is used to create the folds for 5-fold cross-validation. During each fold, only the current random fold is loaded into memory, which helps increase efficiency in memory usage.

The images are going to be scaled down to 256x256x3 pixels to speedup the training of the models. The pixel values are read as RGB. It was created a generator for data augmentation (using ImageDataGenerator of Keras) with different parameters to make the dataset more varied and train the model on different type of images:

- **Rescale:** Rescale the pixel values to the range [0,1] dividing by 255. This normalization helps the convergence of the model.
- **Rotation Range:** Randomly rotate the image between [-40,+40] degrees.
- **Zoom Range:** Randomly zoom-in on the image. The value set is 0.2 so the



	filename	class
0	/content/dataset/train/muffin/img_0_354.jpg	muffin
1	/content/dataset/train/muffin/img_2_450.jpg	muffin
2	/content/dataset/train/muffin/img_4_590.jpg	muffin
3	/content/dataset/train/muffin/img_4_268.jpg	muffin
4	/content/dataset/train/muffin/img_1_93.jpg	muffin
...	...	...
5912	/content/dataset/test/chihuahua/img_3_552.jpg	chihuahua
5913	/content/dataset/test/chihuahua/img_3_186.jpg	chihuahua
5914	/content/dataset/test/chihuahua/img_1_548.jpg	chihuahua
5915	/content/dataset/test/chihuahua/img_2_828.jpg	chihuahua
5916	/content/dataset/test/chihuahua/img_3_536.jpg	chihuahua

5917 rows × 2 columns

Figure 2.1: Dataframe with all the data

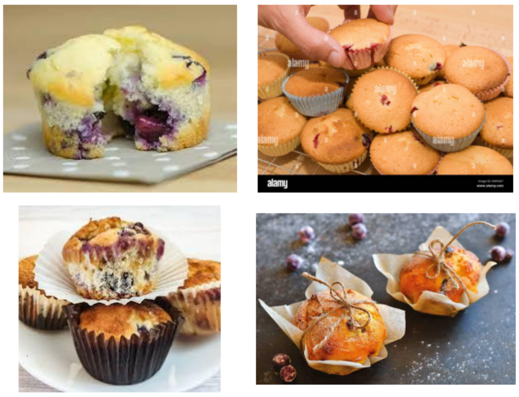


Figure 2.2: Muffins

image will be zoomed out to 80% or zoomed in to 120% in comparison to the original size.

- **Horizontal Flip:** Randomly flip the image horizontally.

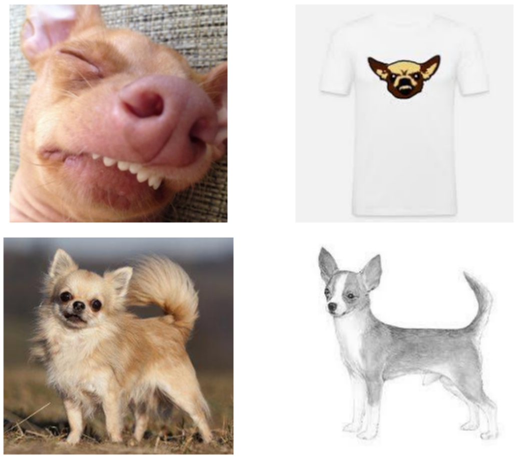


Figure 2.3: Chihuahuas

Chapter 3

MODEL ARCHITECTURES AND HYPERPARAMETERS

To achieve significant results 3 architectures with different numbers of layers and number of filters were used:

3.1 FIRST ARCHITECTURE

This architecture is the simplest one. As we can see on Figure 3.1, it is constructed by 1 convolution layer, 1 Max Pooling2D, 1 flatten layer and 2 dense layers for the output. All the layers are using a ReLU activation function.

3.2 SECOND ARCHITECTURE

In this architecture two convolution layers and two MaxPooling layers were added. Also the number of filters was increased, Figure 3.2 for the details. All the layers are again using a ReLU activation function.

3.3 THIRD ARCHITECTURE

The last architecture is based on the second one, with the difference of a dropout layer being inserted after each couple of convolution layer + MaxPooling layer, Figure 3.3 for the details. During training each dropout layer sets randomly a portion of neurons, of the previous layer, to zero. By not relying too heavily on

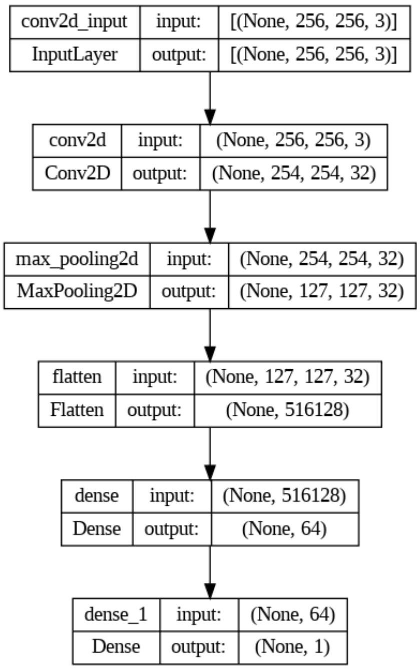

conv2d_input | input: | [(None, 256, 256, 3)]
InputLayer | output: | [(None, 256, 256, 3)]
conv2d | input: | (None, 256, 256, 3)
Conv2D | output: | (None, 254, 254, 32)
max_pooling2d | input: | (None, 254, 254, 32)
MaxPooling2D | output: | (None, 127, 127, 32)
flatten | input: | (None, 127, 127, 32)
Flatten | output: | (None, 516128)
dense | input: | (None, 516128)
Dense | output: | (None, 64)
dense_1 | input: | (None, 64)
Dense | output: | (None, 1)

Figure 3.1: First Architecture

a particular neuron the risk of overfitting is reduced. All the layers are still using a ReLU activation function.

3.4 HYPERPARAMETERS

For every architecture 3 experiments were run, each with a different set of HyperParameters, divided in:

- **Default Parameters:** BatchSize = 32, epochs = 3, learning rate of default of Adam Optimizer so 0.001
- **Lr Parameters:** BatchSize = 32, epochs = 3, learning rate = 0.0001
- **Batch Parameters:** BatchSize = 16, epochs = 6, learning rate of default of Adam Optimizer so 0.001

The aim of these experiments is to see how the different architectures perform on the same set of hyperparameters and how those sets of hyperparameters

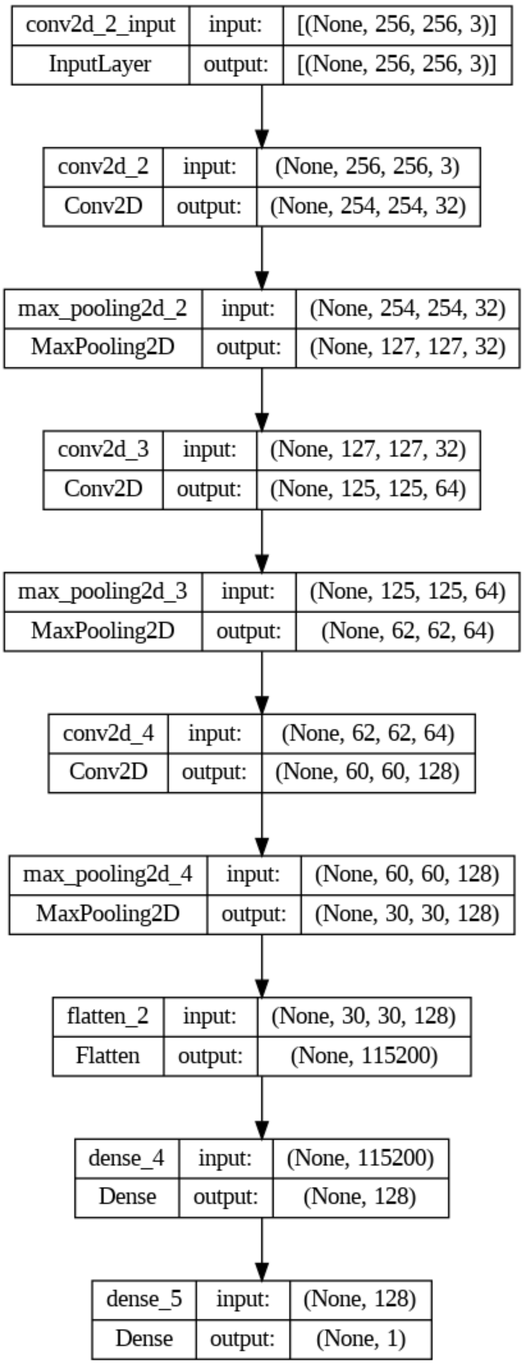


Figure 3.2: Second Architecture

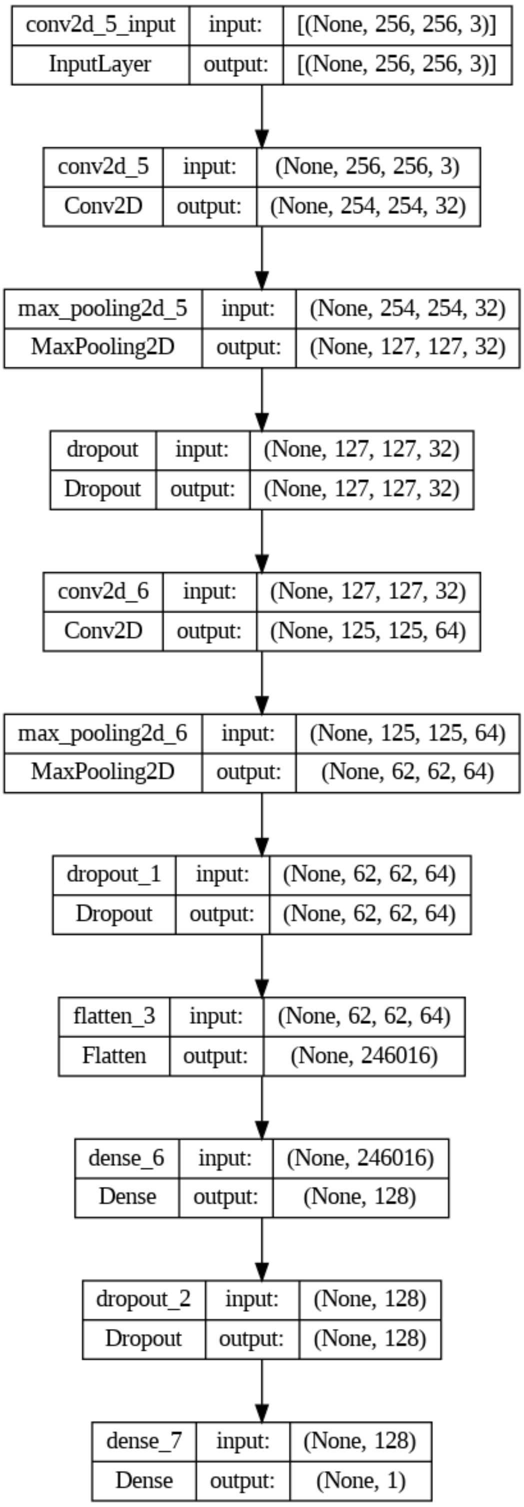

conv2d_5_input | input: | [(None, 256, 256, 3)]
InputLayer | output: | [(None, 256, 256, 3)]
conv2d_5 | input: | (None, 256, 256, 3)
Conv2D | output: | (None, 254, 254, 32)
max_pooling2d_5 | input: | (None, 254, 254, 32)
MaxPooling2D | output: | (None, 127, 127, 32)
dropout | input: | (None, 127, 127, 32)
Dropout | output: | (None, 127, 127, 32)
conv2d_6 | input: | (None, 127, 127, 32)
Conv2D | output: | (None, 125, 125, 64)
max_pooling2d_6 | input: | (None, 125, 125, 64)
MaxPooling2D | output: | (None, 62, 62, 64)
dropout_1 | input: | (None, 62, 62, 64)
Dropout | output: | (None, 62, 62, 64)
flatten_3 | input: | (None, 62, 62, 64)
Flatten | output: | (None, 246016)
dense_6 | input: | (None, 246016)
Dense | output: | (None, 128)
dropout_2 | input: | (None, 128)
Dropout | output: | (None, 128)
dense_7 | input: | (None, 128)
Dense | output: | (None, 1)

Figure 3.3: Third Architecture

change the convergence of the models.

Chapter 4

TRAINING AND EVALUATION

Every architecture was trained and evaluated with all 3 sets of Hyperparameters. As said in the objectives, Section 1.1, zero-one loss was used as the validation loss, being calculated as:

$$\text{Zero-One Loss} = \frac{1}{n}\sum_{i=1}^{n} \mathbb{I}(\hat{y}_i \neq y_i)$$

Then the Validation Accuracy is calculated with the formula:

$$\text{Accuracy} = 1 - \text{Zero-One Loss}$$

4.1 FIRST ARCHITECTURE

Here below the table with the resulting values of zero-one loss, used for validation of each fold and their Arithmetic mean. At Figure 4.1 a Train Loss Graph displaying the trend of the loss values following the epochs:

Experiment	**Fold 1**	**Fold 2**	**Fold 3**	**Fold 4**	**Fold 5**	**Mean**
Default Parameters	0.1571	0.1563	0.1589	0.4661	0.1632	0.2203
Lr Paramenters	0.1698	0.2035	0.1536	0.1849	0.1753	0.1774
Batch Paramenters	0.1622	0.1258	0.1738	0.1396	0.1584	0.1520

Experiment	Execution Time (min)	Accuracy
Default Parameters	31.86	77.9692%
Lr Paramenters	33.56	82.2560%
Batch Paramenters	61.77	84.8049%

4.2 SECOND ARCHITECTURE

Here below the table with the resulting values of zero-one loss, used for validation of each fold and their Arithmetic mean. At Figure 4.2 a Train Loss Graph displaying the trend of the loss values following the epochs:

Experiment	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Mean
Default Parameters	0.1706	0.1275	0.1450	0.1181	0.1554	0.1433
Lr Parameters	0.1875	0.1427	0.1432	0.1276	0.1875	0.1577
Batch Parameters	0.1174	0.1275	0.1233	0.0942	0.1182	0.1161

Experiment	Execution Time (min)	Accuracy
Default Parameters	34.65	85.6691%
Lr Parameters	33.66	84.2286%
Batch Parameters	64.59	88.3890%

4.3 THIRD ARCHITECTURE

Here below the table with the resulting values of zero-one loss, used for validation of each fold and their Arithmetic mean. At Figure 4.3 a Train Loss Graph displaying the trend of the loss values following the epochs:

Experiment	**Fold 1**	**Fold 2**	**Fold 3**	**Fold 4**	**Fold 5**	**Mean**
Default Parameters	0.1706	0.1883	0.2092	0.2292	0.1918	0.1978
Lr Parameters	0.2078	0.2568	0.2040	0.2622	0.1823	0.2226
Batch Parameters	0.1824	0.2255	0.1310	0.1618	0.1524	0.1706

Experiment	**Execution Time (min)**	**Accuracy**
Default Parameters	34.91	80.2168%
Lr Parameters	40.08	77.7407%
Batch Parameters	65.98	82.9371%

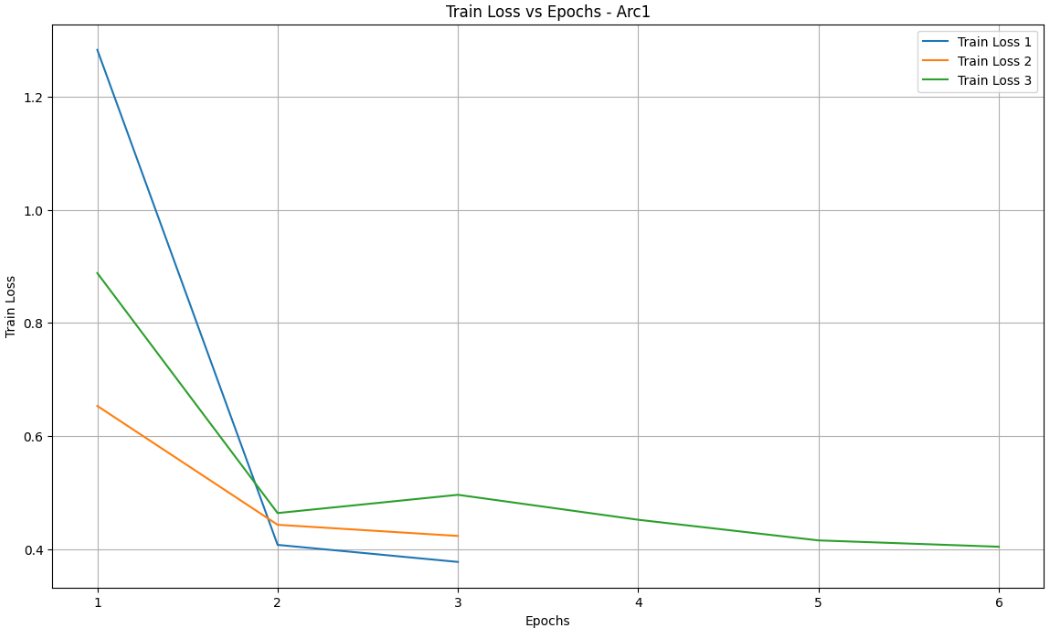


Figure 4.1: Training Loss vs Epochs, in the legend the number after train loss indicate the first, second and third experiment.

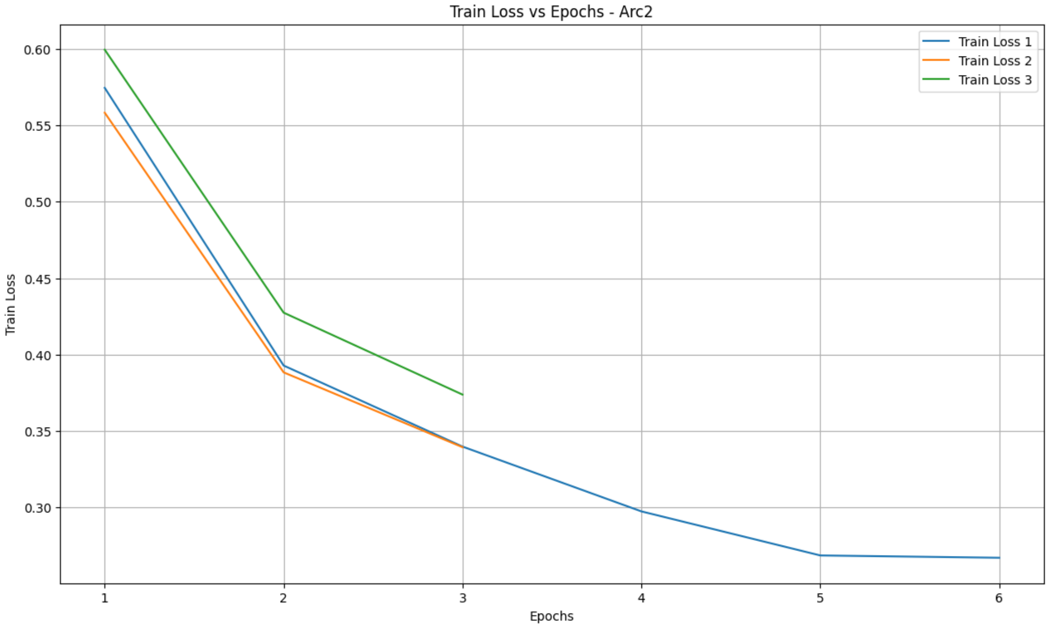


Figure 4.2: Training Loss vs Epochs, in the legend the number after train loss indicate the first, second and third experiment.

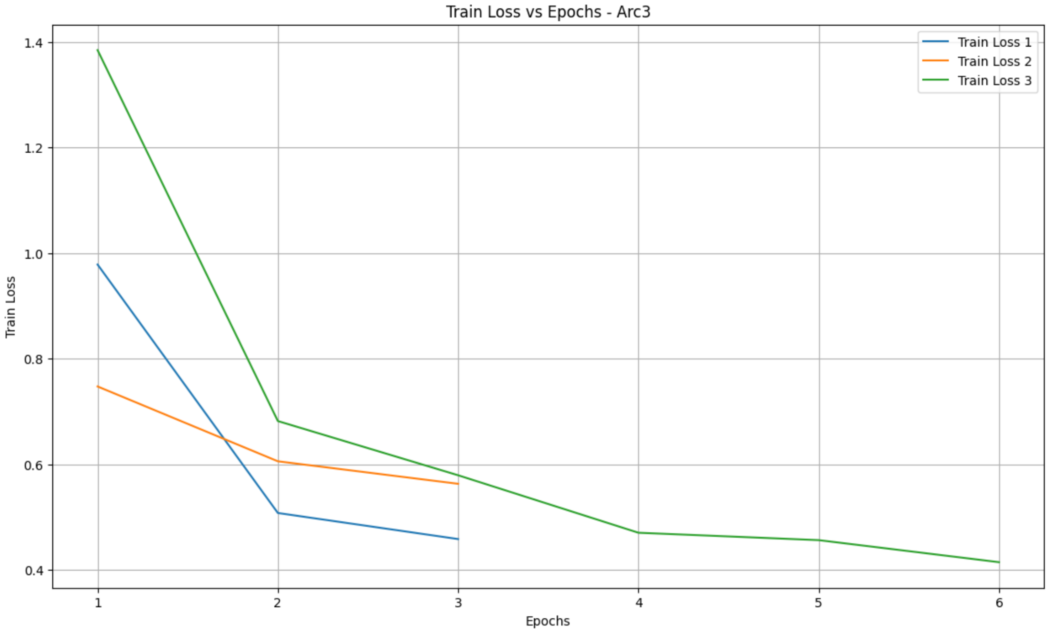


Figure 4.3: Training Loss vs Epochs, in the legend the number after train loss indicate the first, second and third experiment.

Chapter 5

CONCLUSION

As shown in the tables in Chapter 4, Training and Evaluation, increasing the number of epochs and decreasing the batch size leads to an increase in accuracy and a decrease in loss, in all three architectures. The worst performing architecture was predictably the first one, being the most simple. Instead, the tables of the folds show that the second architecture performed the best overall. Lastly, analyzing the third architecture, it can be inferred from the experiments that inserting a Dropout layer did not help to generalize the understanding of the images. It can be speculated that setting a set of random neurons to zero is not useful when detailed detection of very similar images is needed. It seems that in the MuffinVSChihuahua problem adding more convolution layers helps capturing more details in the images and so is the most effective approach.

Overall the experiment was limited by the computational power offered by Google Colab GPUs, so a stabilization of the loss function was not reached, but could be studied by increasing the number of epochs in a freer environment. For future work adding a learning rate scheduler to decrease the learning rate proportionally of the numbers of epochs is recommended.