Convolutional Neural Networks and Image Classification: cats and dogs

Marta Campagnoli
Data Science and Economics
Machine learning, statistical learning, deep learning and artificial intelligence
Machine Learning Module
ID 928635
marta.campagnoli@studenti.unimi.it

1 Introduction

This report analyzes the work conducted in order to develop a Deep Learning solution to classify images using a Convolutional Neural Network (CNN). In the following paragraphs, after a brief description of the theoretical background behind Neural Networks, I will describe the preprocessing techniques applied on the data, the developed network architectures and the training experiments conducted in this work along with their relevant results.

The specific aim of this project was to develop a binary classifier for dog and cats images; the choices made in constructing the different network architectures and for hyperparameters values, training loss and activation functions are explained throughout of this report. Of 9 initial models, 3 have been chosen and trained with 5-fold cross-validation; the best performing model was used for three additional experiments running 5-fold cross-validation changing one of the hyperparameters in the network.

2 Artificial Neural Networks: a brief introduction

The following paragraphs want to provide a synthesis of the theory behind Artificial Neural Networks, with a specific focus on the fundamental elements of Feedforward Neural Networks and Convolutional Neural Networks later referred to in the description of the experiments.

2.1 ANNs: Feedforward Neural Networks

Artificial Neural Networks are a class of learning algorithms used for both supervised and unsupervised tasks, such as prediction and classification, whose functioning is based on the structure of the human central nervous system, where the neurons that compose it are seen as computational devices that receive an input, process it, and emit an output. ANNs can be seen as a collection of artificial neurons, or nodes, connected in a graph characterized by a layered structure; specifically, feedforward neural networks are networks were the outputs of one layer are the input of the next layer, with no loops.

Formally, we can say neural networks are a combination of simple predictors of the form $\mathrm{g}(\mathbf{x}) = \sigma(\mathbf{w}\top\mathbf{x})$, where σ is an activation function, with x representing the input. Moreover, a feedforward Neural Network can be represented as a directed acyclic graph G = (V, E). Nodes V can either be input nodes V_i belonging to the input layer, hidden nodes V_h belonging to one or more hidden layers, or output nodes V_o belonging to the output layer. Each edge E connecting any two nodes (i, j) is associated to a parameter $w_{ij} \in \mathbb{R}$ also known as a weight.

In a multilayered feedforward neural network, multiple hidden layers are present; each node in a layer has incoming edges only from the previous layer and outgoing edges connecting it only to the next layer.

In practice, such networks compute functions of the form $\boldsymbol{f}$: $\mathbb{R}^d \to \mathbb{R}^n$; specifically, each node j that doesn't belong to the input layer computes a function $g(\boldsymbol{v})$ where v is the value of the nodes i such that i is connected to j.

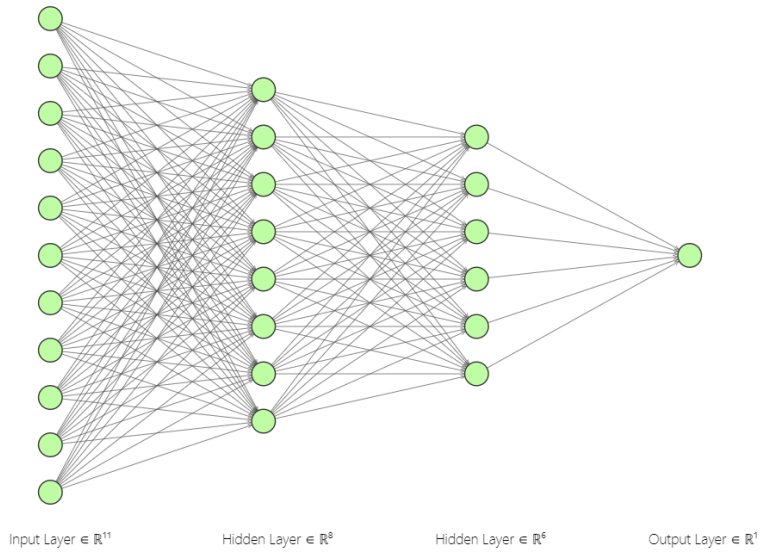

Figure 1: A simple example of a multilayered feedforward NN with fully connected layers

Moreover, calling W the $|V| \times |V|$ weight matrix and considering the graph G and the activation function σ, we can define the function $f = f_{G,W,\sigma}$ computed by the network. Going back to the initial definition, where we said networks can be seen as a combination of activation functions $g(\mathbf{x}) = \sigma(\mathbf{w}\top\mathbf{x})$, or more specifically, $g(\mathbf{x}) = \sigma(\mathbf{w}\top\mathbf{x} + \mathbf{b})$, we can now say $\mathbf{w}\top\mathbf{x}$ is the inner matrix multiplication between the inputs $\mathbf{x}$ and the weights $\mathbf{w}$, with $\mathbf{b}$ being a bias (or threshold) value subtracted to the product to generate what in known as the *net input*.

Since the actual goal of a neural network is to approximate a function *f*, and in order to do so we need non linearity, activation functions, which are generally non linear, apply a non-linear transformation to the computed net input. We can then say activation functions decide, according to their output, whether a neuron should be activated or not; the output of the neuron corresponds to the output of the activation function. The most common activation functions are later described.

In other words, neural networks are able to learn and approximate any function by learning from the data and updating connections - the weights - in the network during the training phase in order to minimize a defined loss (or cost) function measuring the error of the obtained function, with the weights updated through gradient descent and back-propagation; a complete description of these processes is out of the scope of this paper.

We talk about deep neural networks when referring to neural networks with multiple layers that aim to extract progressively higher level features from the data.

2.2 Activation functions

As previously introduced, most activation functions are non linear. Presented here is a list of the ones currently most used.

- **Sigmoid function:** the sigmoid takes form $\sigma(z) = 1/(1+e^{-z})$; taken the input, the output values of the function will fall in the (0,1) range. For this reason it's generally used when the output can or needs to be interpreted as a probability; in practice, other activation functions, like the ReLU, are preferred because of what is known as the *vanishing gradient problem* suffered by the sigmoid - since the weights in the network are updated through gradient descent, the sigmoid, which has derivative close to 0, will update weights very slowly causing the network to have a very long learning time.

- **ReLU function**: the ReLU, or Rectified Linear Unit, is one of the currently most common activation functions, as it solves the vanishing gradient problem. The ReLU computes the function $\sigma(z) = max(0, z)$.

- **Leaky ReLU**: the Leaky ReLU tries to solve what is known as the *dying ReLU problem*. Since the derivative of the ReLU is 0 for negative input values, the gradient doesn't propagate and

weights corresponding to these values are never updated during back-propagation; on the other hand, instead of taking value = 0 when the input is negative, the leaky ReLU has a small positive slope. The function computes $\sigma(z) = (\alpha\mathbb{I}\{z < 0\} + \mathbb{I}\{z \geq 0\}z$, where α is typically set at a value of the order of 10^{-2}.

- **Softmax function**: the softmax function is commonly used in the output layer of networks implemented for multi-class classification tasks. The function normalizes the output values into a probability distribution whose values sum to one by dividing the exponential of each output by the sum of the exponential of all outputs:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

2.3 Convolutional Neural Networks

Convolutional Neural Networks are feedforward networks that employ a new kind of layer which is able to improve the accuracy of ANNs in the field of image classification; because of this reason, the Neural Networks used in this work are CNNs. In the basic architecture of a CNN, Convolutionals Layers, Pooling Layers, and a Fully-Connected Layer are stacked to form the network. Before starting to describe the experiments conducted, this last theoretical paragraph describes the structure and elements of CNNs.

2.3.1 Convolutional Layers

The use of convolutional layers is what differentiates CNNs from standard feedforward neural networks, which are generally composed of what are know as fully-connected or dense layers; in such layers, each node i is connected to every node in the next layer through an edge associated to a weight w_i. Convolutional layers, which work under the assumption that the input is an image, contain instead a set of features or kernels, which are matrices of learnable parameters. Each filter has size smaller than the image, with three dimensions corresponding to its width, height, and depth, whose value corresponds to the number of channels in the image. During the forward pass, each filter slides (convolves) along the height and width of the image; at each position of the filter, the dot product between the weights of the filter and the input is calculated, and each of the results will constitute an element of a 2-dimensional activation map which represents the responses of the kernel at every spatial position. At each step each kernel slides of a number of pixels known as **stride**. The output volume is generated by stacking the activation maps produced by each filter; moreover, since the aim of a convolutional layer is to learn features by sliding through the image, the elements of each activation map share their weights.

Since each filter is smaller than the input, each element in the activation map, which can be seen as a neuron, is only connected to a small local region of the input volume, which means the receptive field size of each neuron corresponds to the filter size. This local connectivity allows the network to learn filters which have maximal response to a local region of the input; initial convolutional layers capture the low-level features, such as edges or blocks of color, while the later layers extract high-level features such as shapes.

Convolutional layers often use **zero-padding**, which consists in adding a border of pixels with value 0 to the input image to help ensure input and output have the same size by adding space for the kernel to convolute; like in fully connected layers, an activation function is employed to increase the non-linearity of the output.

2.3.2 Pooling Layers

Pooling layers are inserted between successive convolutional layers in order to reduce the spacial dimension of the feature maps, and consequently the amount of computation to be performed in the network thus potentially improving the speed of the training, and the number of weights in the network which can also be useful in controlling overfitting. This is achieved by downsampling each activation map indipendently through a criterion that chooses the most relevant freatures. The term pooling refers to the fact that such layers work by partitioning each map in pools, or filters, generally of size 2x2 with stride set to 2. Among the most common pooling layers we can find **Max Pooling**, which selects the maximum value for each pool, thus retaining the most relevant features, and **Average Pooling**, which computes the average value of each pool.

2.3.3 Fully Connected Layer

At the end of the structure of the CNN, the feature maps are flattened into a one-dimentional vector by a Flatten layer; its output will finally serve as input for the Fully Connected layer, otherwise known as Dense Layer, whose structure is useful in order to understand relationships among features. The Dense Layer provides the final output of the network.

2.3.4 Dropout Layer and Batch Normalization Layer

Dropout and Batch Normalization layers are often used in order to improve the accuracy of neural networks and controlling overfitting in the training phase. Their use is thus considered a regularization technique, along with early stopping and data augmentation. Described here are their main characteristics.

- Dropout Layer: this kind of layers are used to control the overfitting problem often encountered in training neural networks; they randomly "drop" a portion of the neurons in a layer with a set probability during the training of the network, thus leaving out some of the information learned by the network. They're used after the input, convolutional and dense layers.
- Batch Normalization Layer: Batch Normalization contributes to controlling overfitting and speeding the learning process. The layer takes the output of the Convolutional Layer as input and standardizes it for each mini-batch.

3 Methodology

This section of the report describes the data preprocessing and experiments conducted in this project. In practice, the work has been divided in three notebooks: the "PreCrossValidationTraining" notebook, which reports 19 training experiments on 9 different network architectures, the "5fold" notebook, containing 9 experiments ran using 5-fold cross-validation on 3 different architectures, and "Visualization", an additional notebook where images and network structures are visualized. The "PreCrossValidationTraining" notebook contains the experiments conducted to select a subset of models to use for training with 5-fold cross validation. In fact, the "5fold" notebook, which responds to the necessity of the work having a more limited running time, can be run indipendently from the others, because each of the notebooks contains the data preprocessing portion of the project. The notebooks suffer of some avoidable code repetitions which can be fixed with the implementation of auxiliary functions; however resource constraints limited the possibility of repeating the experiments, and the project was presented as is to mantain training outputs.

The experimental part of this work, while based on current literature on the best practices to improve the accuracy of an ANN, has not been adapted from other solutions; public sources have been used to learn and adapt techniques in order to build the dataset, display images and plots, and implement k-fold cross validation. Additional images and graphs not reported in this paper can be found in the "Visualization" notebook. The project has been run through Google Colab; some of the choices of settings and hyperparameters used during training, such as the rescaling size of the images and the use of regularization through early stopping to detect overfitting in the first batch of experiments, were made taking into considerations the limited resources at hand.

3.1 Dataset and data preprocessing

The dataset originally consisted of 25.000 JPEG colored pictures of variable size, evenly split between 12.500 pictures of cats and 12.500 pictures of dogs. The preprocessing phase of this work was conducted through the OpenCV Python library for computer vision and image elaboration and the OS module to efficiently associate each image to its label. Specifically, cat images have been associated to label 0 and dog images to label 1. Images were set to a 100x100 size and converted to grayscale, so that they have one channel, and kernels will have depth equal to 1; both choices were made because of the limited computational resources at disposal in the Colab environment. OpenCV automatically detects faulty images so that the final dataset contains 24 946 elements, which were cast into two arrays, one of images and one of labels. Both arrays were cast into tensors so that they could be used in TensorFlow; pixel values for the images were rescaled from the 0-255 to the 0-1 range. Lastly, for the first batch of

experiments, which don't use 5-fold cross validation, the dataset was divided between a training set and test set with a 80-20 split.

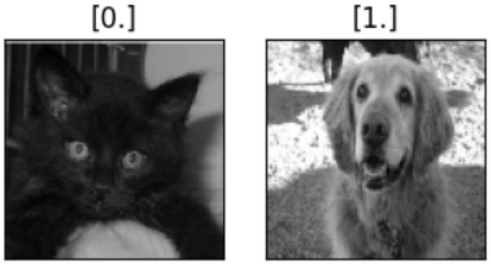

Figure 2: A sample of two resized images with their respective labels after being cast into tensors

3.2 Settings, Hyperparameters and Functions

While the next paragraph describes the network architectures used in this work, provided below is a list of elements and settings common to all experiments. Correctly choosing and tuning hyperparameters in ANNs is fundamental to obtain a well permorming classifier.

- **Optimizers:** alongside the loss, optimizers are one of the two arguments needed when compiling a neural network model. They are a function or algorithm which adjusts the weights and learning rate of the network in order to reduce the loss. A number of different optimizers exist, and three were used in this work:

  1. **Stochastic Gradient Descent, or SGD,** is a variant of Gradient Descent which updates the model parameters for each training example after the computation of the loss. In simple words, Gradient Descent moves the weights into the direction of the minimum of the loss function through backpropagation with the size of its step determined by the learning rate; since GD updates the weights after all the training data has been observed, SGD tries to overcome the slow rate of convergence of GD and the large memory required to compute GD by calculating at each step the gradient for one observation picked at random instead of calculating the gradient for the entire training set. However, both algorithms may get stuck at a local minima for the function, and other optimizers are currently favored over SGD. In this project, SGD has been used in one experiment in order to compare its performance to other optimizers, using its SGD with momentum variation, which brings to faster convergence, with learning rate set to 0.01 and momentum at 0.9.
  2. **Root Mean Square Propagation, or RMSprop,** still adapts the weights through gradient descent but uses a different adaptive learning rate for each parameter, scaling each rate by the moving average of the squared gradients; this is known to stabilize the learning process and make it faster while making the algorithm less prone to getting stuck in local minima and controlling the issue of exploding or vanishing gradients. All models were trained using both RMSprop and ADAM. The initial learning rate was set at 0.001.
  3. **Adaptive Moment Optimization, or ADAM,** performs gradient descent using both scaling and adaptive learning rates and is known to combine the adavantages of SGD with momentum and RMSprop.

- **Training hyperparameters:** the training of a neural network is executed by feeding the network a batch of training data points at each step, and the process of feeding the entire training dataset is repeated for a number of times known as epochs. For this work the number of **epochs** for each experiment was set to 50; the size of each **batch** of training examples fed to the network is 64. This choice was made based on the criterion for which small batches are recommended to start training, and after initial training experiments, not reported in this work for brevity, showed that the standard value of 32 for batches did not lead to satisfactory results for simple models on this dataset.

- **Earlystopping:** we call Earlystopping a callback, used during training as a regularization technique, which stops the training if the performance of the model on a monitored measure on the validation set doesn't improve for a given number of epochs; this can help in detecting overfitting. Two versions of earlystopping were used in the first part of the project, both monitoring the validation loss and stopping respectively after 10 and 15 epochs if the loss didn't improve, where the improvement when setting the function to *minimum mode* is meant as reaching a new minimum for the monitored metric. The choice of using earlystopping was ased on previous knowledge that models that do not use Batch Normalization and Dropout layers tend to overfit, and earlystopping sped up the testing process on the simpler of the 9 models that were trained in the part of the project conducted without 5-fold cross-validation.

- **Convolutional Layers:** the ReLU function has been used for all convolutional layers, with exception for one experiment conducted using the Leaky ReLU. All layers use zero-padding and don't introduce a bias in the computation of the input-weights dot product. This latest choice was based on the fact that Batch Normalization obviates to the need for the bias and the experiments were able to show the improvement in performance when introducing such layers. Since small kernels are known to better generalize features, they were set at size 3x3 while stride was kept at its (1,1) default value.

- **Pooling Layers:** activation maps downsampling was achieved through **MaxPooling** layers. Pool size was set to (2,2).

- **Dense Layers and Output:** two dense (or fully connected) layers are present in all models. The first is a layer of 512 units using the ReLU activation function, while the second, which represents the output layer, uses the Sigmoid activation function with 1 node. We can interpret the output in such a way that a predicted value ≥ 0.50 corresponds to the data point having a predicted label of value 1, and a predicted value < 0.50 corresponds to the data point having a predicted label of value 0.

- **Loss Function:** the first batch of experiments, conducted without cross-validation, as well as the training loss for 5-fold cross-validation, use **Binary Cross-Entropy**, coherent with the output of the Sigmoid activation function with one node; the BCE function implemented by Keras used in this work automatically assumes the predicted values correspond to a probability in the (0,1) range. The BCE, also known as *logloss*, penalizes wrong predictions through the logarithm function and has formula

$$BCE = -\frac{1}{N}\sum_{y=1}^{N} y_i \log(p(y_i)) + (1 - y_i)\log(1 - p(y_i))$$

  where y_i is the true label for each data point an $p(y_i)$ is the predicted probability.

  Reported cross-validated risk estimates in the experiments using 5-fold cross validation were computed with the **zero-one loss**. In order to correctly compute the value of the loss, in this case the outputs of the sigmoid function were rounded to 0-1 values.

3.3 Network Architectures

All network architectures are now described, specifying whether they were trained with 5-fold cross-validation or discarded after the first part of the work.

1. ***Stack128*** is a CNN presenting Convolutional, Maxpooling, and Dense Layers. Convolutional layers are stacked in 3 blocks of 2 layers each; a Maxpooling layer is placed after each block. In the first block both layers present 32 filters, in the second both have 64 filters, and in the third both have 128 filters. After the first phase, the network was not selected for training with 5-fold cross-validation.

2. ***Stack256*** presents a similar structure, with 4 blocks with 2 Convolutional layers each, with every block followed by a Maxpooling layer. The first three blocks are identical to the previous model, while the fourth block has 2 layers using 256 filters. The model was not used after the first phase.

3. ***Stack512*** has the same structure of the two previous models, but adds a final fifth block of two Convolutional layers with 512 filters and a Maxpooling layer. The model was not used after the first phase.

4. ***Drop256*** takes the structure of Stack256 and adds a Dropout layer with a 0.3 dropout rate after the Maxpooling layer of the first three blocks, and 0.4 rate Dropout layers after the pooling layer of the fourth block and after the first Dense layer. The model was selected after the first batch of experiments and trained with 5-fold cross-validation.

5. ***Drop512*** similarly takes the structure of Stack512 and adds a Dropout layer with a 0.2 dropout rate after the Maxpooling layer of the first two blocks, a 0.3 rate Dropout layer after the pooling layer of the third and fourth blocks, and a 0.4 rate Dropout layer after the pooling layer of the fifth block and after the first Dense layer. The model was not selected after the first batch of experiments.

6. ***DropBatch256*** modifies the structure of Drop256, using the same dropout rates, but adding a third Convolutional layer for each block and a BatchNormalization layer directly after each Convolutional layer and after the first Dense layer. The model was selected to be trained with 5-fold cross validation.

7. ***DropBatch512*** modifies the structure of Drop512; similarly to the previous model, it adds a third Convolutional layer for each block and a BatchNormalization layer directly after each Convolutional layer and after the first Dense layer. Additionally, the dropout rate in the after the first two blocks was modified to 0.15, to 0.25 after the third and fourth block, and to 0.3 after the fifth block and the first dense layer. The model was selected for 5-fold cross-validation.

8. ***DropBatch1024*** implements a structure identical to DropBatch512, but adds a sixth block of three Convolutional layers with 1024 filters, followed by a BatchNormalization, MaxPooling and Dropout layer. The dropout rate is set to 0.2 after the first two blocks, to 0.25 from the third to fifth blocks, and to 0.3 after the sixth block and first dense layer. The network was not trained with 5-fold cross-validation.

9. ***DropBatchMod1024*** modifies DropBatch1024 by using Dropout layers only up to the third block; the dropout rate of the layers kept in the network was not modified. The network was not trained with 5-fold cross-validation.

3.4 Network training and evaluation

Before describing in depth the experiments and their results, I briefly lay out the structure of the project. After data preprocessing the project was carried out in three main phases:

- in the first phase, presented in the "PreCrossValidationTraining" notebook, a series of 19 experiments is conducted. The 9 different network architectures are trained without using 5-fold cross-validation, with the first model, *Stack128*, trained with SGD with momentum, RMSprop and ADAM as optimizers, and the other eight models trained using only RMSprop and ADAM. At the end of this phase, the three best performing architectures in terms of validation loss and binary accuracy are chosen to be used in the following phase;

- in the second phase, presented in the "5fold" notebook, the *Drop256, DropBatch256* and *DropBatch512* models are trained using 5-fold cross validation, each with both RMSprop and ADAM as optimizers;

- in the third and last phase, also presented in the "5fold" notebook, the *DropBatch256* model, evaluated as the best one in terms of loss and accuracy when trained with RMSprop, is trained again with 5-fold cross- validation in three different instances, changing respectively the activation function of all layers but the output one, the batch size and the starting learning rate for RMSprop.

4 Experiments and Results

This paragraph describes the experiments conducted to train the neural networks and their results. The first three architectures were used as a baseline to show that Dropout and Batch Normalization layers help

controlling overfitting and improve the performance of the network in terms of loss and accuracy through regularization, and to show that deepening the structure of a neural network improves its performance, which is the main intuition behind Deep Learning and the idea of stacking multiple layers in order for the network to progressively detect higher features from the data. The use of multiple stacked layers is at the base of the current best performing CNN classifiers, such as the VGG-19 architecture.

4.1 Training without 5-fold cross validation

In all training experiments without cross-validation, Binary Cross-Entropy was used for both training and validation loss. All tables for this portion of the work report the validation loss and accuracy obtained by the algorithm on the test set at the last epoch and, when using earlystopping callbacks, validation loss and accuracy on the test set when using the best optimized weights obtained during training, along with the number of epochs after which the algorithm stopped.

4.1.1 Stack128, Stack256, Stack512

- Stack128 was trained using SGD, RMSprop and ADAM. However, using the earlystopping callbacks with patience set to 10 showed that the algorithm started overfitting very fast, stopping as early as after 16 epochs for the RMSprop optimizer. For this reason, this model was not used for further experiments. Moreover, the SGD optimizer, which had the higher loss and lower accuracy, was discarded and only RMSprop and ADAM were used in further experiments. Details of the result are shown in the table below.

Stack128					
Optimizer	Loss	Accuracy	Best Weight Loss	Best Weights Accuracy	Epochs
SGD	1.4138	0.7609	0.4908	0.7631	19
RMSprop	0.5660	0.8433	0.2864	0.8758	16
ADAM	1.2558	0.8018	0.4463	0.8098	17

Table 1: Training results for Stack128.

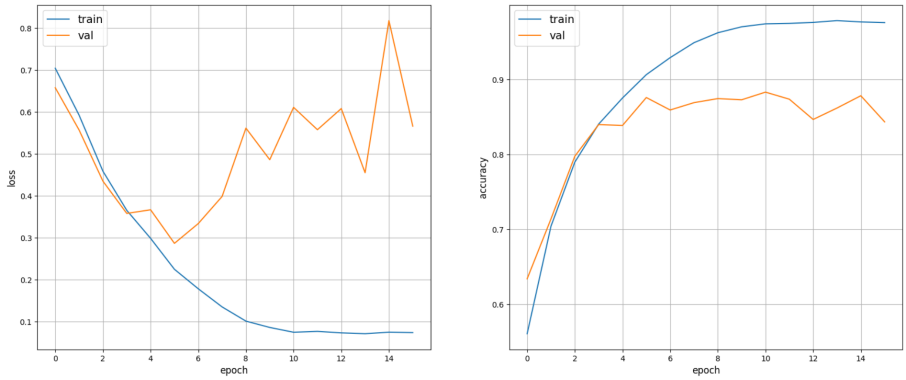

Figure 3: Plots of training and validation loss and accuracy for Stack128 trained with RMSprop, showing the algorithm overfits very quickly

- Stack256 and Stack512 were each trained on both RMSprop and ADAM, and while both loss and accuracy improved in both cases, none of the models reached satisfactory results, with similar final values for the measured metrics. When training Stack256, the earlystopping callback was used with patience set to 10, while it was set to 15 for the Stack512 model; only when training the latter architecture using RMSprop 36 epochs where reached, showing the networks presenting this structure still overfit very fast. As expected, deepening the structure of the network improves the results, but is not sufficient on its own, especially with a dataset of approximately 25.000 images.

Stack256					
Optimizer	Loss	Accuracy	Best Weight Loss	Best Weights Accuracy	Epochs
RMSprop	0.5134	0.8774	0.2051	0.9134	20
ADAM	0.6801	0.8577	0.3488	0.8507	18

Table 2: Training results for Stack256.

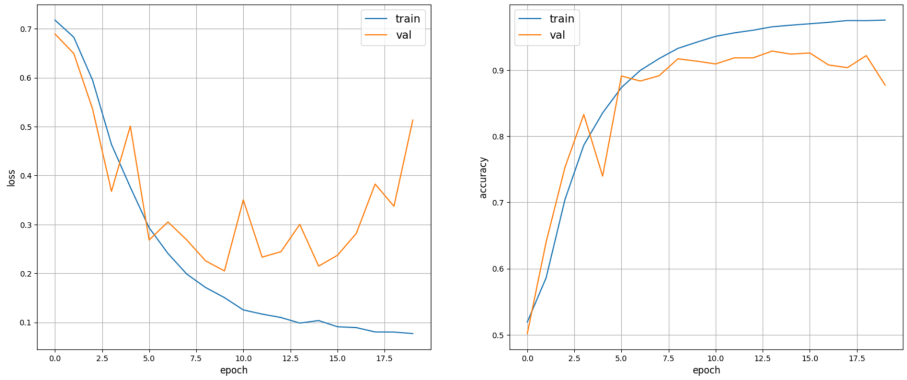

Figure 4: Plots of training and validation loss and accuracy for Stack256 trained with RMSprop, showing the algorithm still overfits quickly.

Stack512					
Optimizer	Loss	Accuracy	Best Weight Loss	Best Weights Accuracy	Epochs
RMSprop	0.4872	0.8733	0.2094	0.9166	36
ADAM	0.5399	0.8743	0.2996	0.8697	27

Table 3: Training results for Stack512.

4.1.2 Drop256, Drop512

- Drop256 and Drop512, both trained with both optimizers using earlystopping with patience set to 15, show as expected that introducing Dropout layers in CNNs helps controlling overfitting and leads to better results for Loss and Accuracy, showing the advantage brought by using regularization; in any case, since BatchNormalization is known to improve the performance of CNNs, ulterior architectures were implemented. However, Drop256 trained with RMSprop amd ADAM obtained the best results for the Loss, respectively 0.1886 and 0.1959; while keeping in mind that earlystopping was used, the model was selected to be trained with 5-fold cross validation. The two architectures were trained for different dropout rate values, tuned according to previous experiments that were not reported here to avoid redundancy. Moreover, since none of the models were trained for less than 40 epochs (with Drop512 with ADAM reaching the maximum of 50) further experiments were executed without earlystopping.

Drop256					
Optimizer	Loss	Accuracy	Best Weight Loss	Best Weights Accuracy	Epochs
RMSprop	0.1886	0.9287	0.1712	0.9357	44
ADAM	0.1959	0.9357	0.1721	0.9353	40

Table 4: Training results for Drop256.

Drop512					
Optimizer	Loss	Accuracy	Best Weight Loss	Best Weights Accuracy	Epochs
RMSprop	0.3020	0.9202	0.1815	0.9232	41
ADAM	0.2407	0.9212	0.1908	0.9283	50

Table 5: Training results for Drop512.

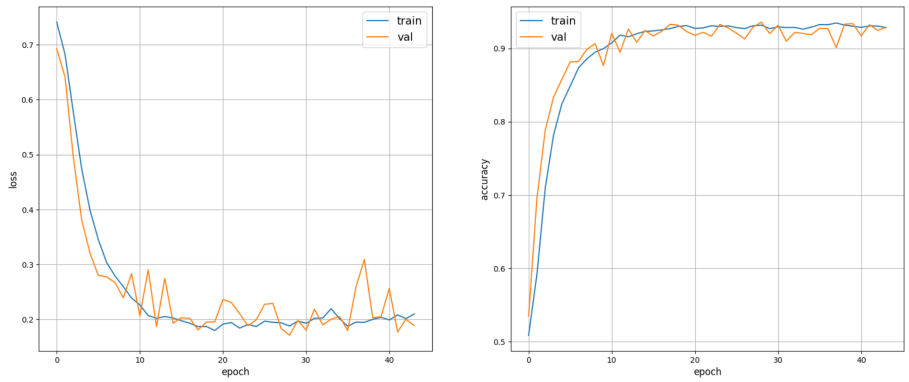

Figure 5: Plots of training and validation loss and accuracy for Drop256 trained with RMSprop, showing an improvement in overfitting.

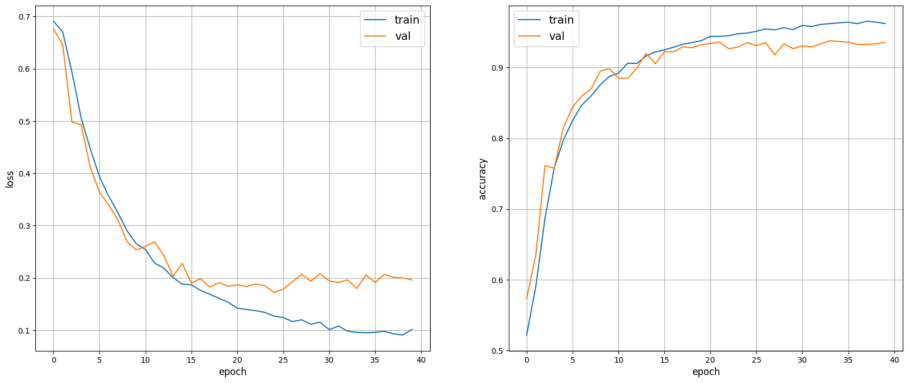

Figure 6: Plots of training and validation loss and accuracy for Drop256 trained with ADAM, showing an improvement in overfitting

4.1.3 DropBatch256, DropBatch512, DropBatch1024, DropBatchMod1024

As peviously introduced, BatchNormalization regularization layers were introduced to show the improvement brought by both normalizing data batches and introducing a bias in the computation of the input-weight inner product. In order to see if further deepening the network once Dropout and BatchNormalization were introduced could lead to a better performance, each block presents an additional convolutional layer, and furthermore the DropBatch1024 and DropBatchMod1024 networks were implemented and trained to observe if layers with an higher number of kernels lead to better performances.

- DropBatch256 and DropBatch512 were trained with both RMSprop and ADAM without earlystopping. Dropout rates were not modified with respect with the previous architecture using only Dropout layers. For DropBatch256, the loss increased for both optimizers but the accuracy incresed, obtaining the two best overall results with an accuracy of 0.9429 for RMSprop and of 0.9515 for ADAM. For DropBatch512, both the values for the loss and for accuracy improved with respect to previous training experiments. For this reason, both models were trained with 5-fold cross validation.

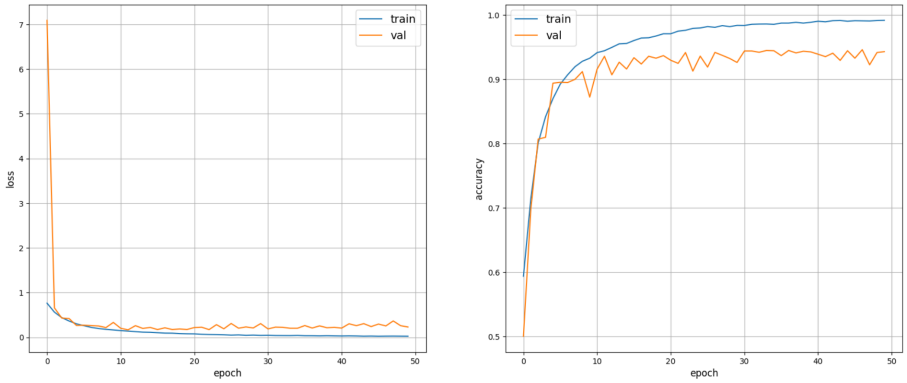

Figure 7: Plots of training and validation loss and accuracy for DropBatch256 trained with RMSprop

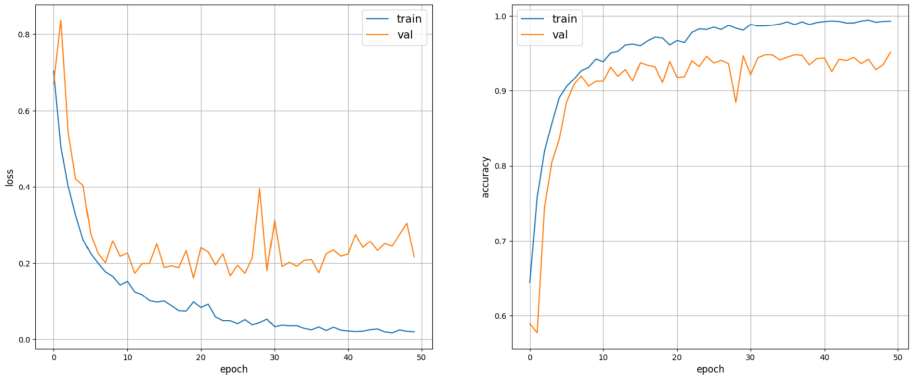

Figure 8: Plots of training and validation loss and accuracy for DropBatch256 trained with ADAM

DropBatch256		
Optimizer	Loss	Accuracy
RMSprop	0.2300	0.9429
ADAM	0.2157	0.9515

Table 6: Training results for DropBatch256.

DropBatch512		
Optimizer	Loss	Accuracy
RMSprop	0.2460	0.9425
ADAM	0.2107	0.9423

Table 7: Training results for DropBatch512

- DropBatch1024, DropBatchMod1024

  As for previous models, dropout rates for these architectures were chosen according to previous experiments that were not reported here. When looking at the results for DropBatch1024, while the values for the validation loss for the last epoch are lower than for the DropBatch256 model, the accuracy is lower or comparable. The values of the loss are also higher than for the Drop256 model, which was however trained on less epochs because of the use of the earlystopping callback. The choice of not training this model with 5-fold cross-validation was based on the overall observation of the results during training; as stated, the reported values refer to the results obtained by the algorithm on the test set at the last epoch, but when looking at values for the validation loss and accuracy for each epoch during training, the model shows a performance comparable to the previous architectures. For this reason, smaller models needing less computational resources were preferred.

As for the modified version of the architecture, neither experiments obtained significantly improved performance with respect to other architectures. These results are coherent with empirical and theoretical evidence that adding layers of existing size is more effective than modifying the size of existing layers, specifically in order to decrease the bias error.

DropBatch1024		
Optimizer	Loss	Accuracy
RMSprop	0.2039	0.9395
ADAM	0.2115	0.9429

Table 8: Training results for DropBatch1024.

DropBatchMod1024		
Optimizer	Loss	Accuracy
RMSprop	0.4022	0.9331
ADAM	0.2136	0.9483

Table 9: Training results for DropBatchMod1024.

4.2 Training with 5-fold cross validation

The next part of the work was conducted by training the three chosen models, Drop256, DropBatch256 and DropBatch512, with both optimizers and 5-fold cross validation. Ideally, all experiments should have been conducted this way, and on a larger grid of hyperparameters for each of the architectures; as previosuly stated, the choice of selecting a subset of models to train with 5-fold cross-validation was due to the limited computational and time resources provided by the Google Colab environament but otherwise needed in order to train a CNN.

For this part, the training loss remains the Binary Cross-Entropy while the validation loss is computed through the zero-one loss. The zero-one loss, which corresponds to the fraction of misclassified data points, so that the loss and Binary Accuracy sum up to 1, doesn't apply any penalization and it will be consequently lower than BCE as showed in previous results; in practice, BCE uses probability values and applies a penalization that considers the distance of the result from the true label of the data points, while the zero-loss works through a count of the classification errors. Tables for this paragraph show the average scores on the 5 folds for the zero-one loss and binary accuracy.

Drop256 5-Fold Cross Validation		
Optimizer	Zero-One Loss	Accuracy
RMSprop	0.0853	0.9147
ADAM	0.1130	0.8870

Table 10: Training results for Drop256, 5-fold cross-validation.

We see that the use of 5-fold cross validation shows more clearly the advantage of adding BatchNormalization layers, which also obviates to the choice of not using a bias in the Convolutional layers; the results for the loss for Drop256, using only Convolutional and Dropout layers, are noticeably higher than for DropBatch256. Particularly, the average score for the loss when training the model with ADAM is 0.1130, and binary accuracy is 0.8870; while the result for each epoch inside each fold were not shown, it can be hypothesized the algorithm started overfitting and is sensitive to changes in the training set, with the fourth fold scoring a binary accuracy as low as 0.7915 and a zero-one loss of 0.2028. On the other hand, DropBatch256 obtains the overall best results, with RMSprop having a slightly better performance with zero-one loss of 0.0567 and accuracy of 0.9433.

On the other hand, we see that training DropBatch512 with both optimizers yields slightly higher zero-one loss; taking into account that ulterior tuning of the hyperparameters could adjust the results, it can also be hypothesized that the small size of the images, the use of grayscale and the limited size of

DropBatch256 5-Fold Cross Validation		
Optimizer	Zero-One Loss	Accuracy
RMSprop	0.0567	0.9433
ADAM	0.0580	0.9425

Table 11: Training results for DropBatch256, 5-fold cross validation.

the dataset could limit the improvement in performance brought by increasing the number of kernels in the Convolutional layers.

DropBatch512 5-Fold Cross Validation		
Optimizer	Zero-One Loss	Accuracy
RMSprop	0.0667	0.9332
ADAM	0.0585	0.9415

Table 12: Training results for DropBatch512, 5-fold cross validation.

4.3 Best performing model: further training experiments

In the last part of the work the best performing model, DropBatch256, whose structure can be seen in Figure 9, was trained another 3 times with 5-fold cross validation using RMSprop; in each of the experiments one of the hyperparameters was changed.

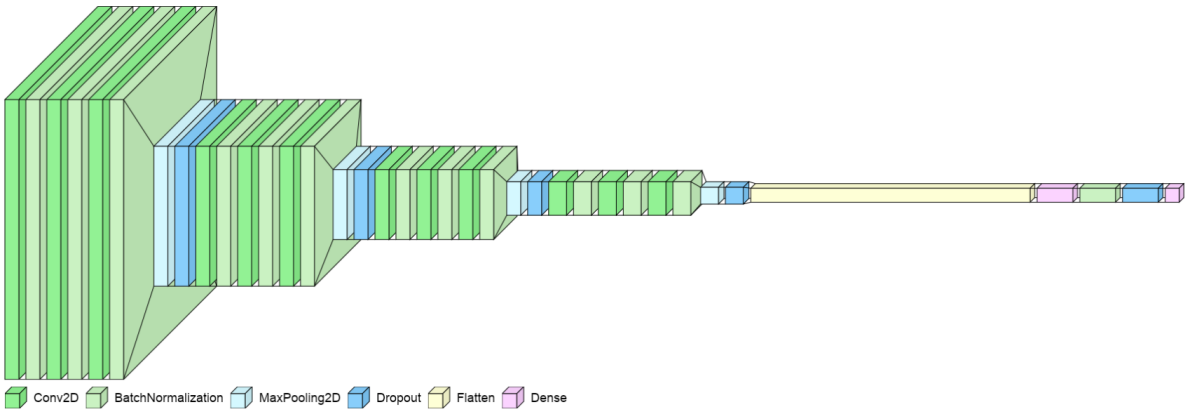

Figure 9: 3D model of DropBatch256, the best performing model

- In the first case, the activation function for all Convolutional layers and the first Dense layer was chenged to the Leaky ReLU, with $\alpha = 0.01$, in order to see if accuracy is improved by addressing the possible problem of the *dying ReLU*. We see the value for the average zero-one loss has decreased slightly to 0.0559, and the average binary accuracy has slightly increased to 0.9441. However the improvement is small, and when looking at the confusion matrix for predictions on the test set at fold 5 for the DropBatch256 model for both activation functions, the model using ReLU misclassified 271 examples on a total of 4989, while the model using Leaky ReLU misclassified 340 data points. Both confusion matrices can be found in Appendix A, along with the results of the training experiment for each fold.
- For the second case, the initial learning rate was changed to 0.002, to see if the the network responded positively to a faster starting learning rate. This was disproven, as the training lead to an higher zero-one loss and lower accuracy: speeding up the training has not improved the results.
- For the third and last training experiment, batches of size 32 were used, to see if a more complex architecture obtained better results with smaller batches; the measures metrics did not improve, but are comparable to the ones obtained on the DropBatch512 model trained with ADAM.

Modified DropBatch256, 5-Fold Cross Validation		
Hyperparameter	Zero-One Loss	Accuracy
Leaky ReLU	0.0559	0.9441
Learning Rate 0.002	0.0690	0.9309
Batch Size 32	0.0587	0.9412

Table 13: Results of further training experiments on DropBatch256. All experiments conducted using the RMSprop optimizer

We can conclude that the DropBatch256 model yielded the overall best results, specifically when trained with RMSprop; in Appendix A additional tables can be found showing the loss and accuracy values by fold for both the RMSprop and ADAM training experiments on the architecture, along with a sample of images with the prediction results for the algorithm trained with RMSprop. DropBatch256 enjoys the improvement brought by regularization techniques, while using hyperparameters suitable to the size of the dataset and characteristics of the data.

5 Conclusions

The aim of this work was to develop a network architecture in order to perform binary classification on images; while the best classifier reached an accuracy of 94.33% with the ReLU activation function and of 94.41% with Leaky ReLU, proving that models using both Dropout and Batch Normalization layers for regularization obtain the best performance, the best CNN architectures currently developed for image classification can reach a 99% accuracy rate, showing the results obtained in this work could be improved. It can be considered that instead of adding blocks of layers with an increasing number of kernels, the architectures could have been deepened by ulteriorly adding layers with up to 256 kernels to the best performing models. On the other hand, it is to be noticed that the dataset is comprised of 25 000 pictures, scaled down to 100x100 size, and in greyscale. A larger size for the images, and the use of RGB pictures with kernels of depth 3 could have improved the results significantly, and even though it can be argued that color doesn't particularly distinguish cats from dogs, certain fur patterns do.
Data augmentation, a regularization technique which augments the number of pictures in a dataset by rotating, flipping, cropping and adjusting the brightness of existing images to increase the number of available datapoints could be applied; data augmentation is known to control overfitting. Furthermore, other regularization techniques, such as weight decay and L1 and L2 regularization, have not been used in this work. Moreover, this project focused on building different architectures for the networks; the work could have been expanded by further tuning the hyperparamenters on a limited number of models, such as using a grid of values for the starting learning rate for the optimizers and for the batch sizes, and by further exploring possible results when using the Leaky Relu.
Lastly, it can be noticed that, while RMSprop and ADAM where chosen because they are currently considered two of the best performing optimizers, ADAM can be considered a modification of RMSprop, and neither of the two showed a clear advantage in terms of loss, accuracy, overfitting and stability of the learning process; clearer results could have been reached by choosing a different optimizer for comparison or continuing training with SGD instead of discarding its use after training the first model. As a last addition, it is to be remembered that more complex structures of the networks lead to an increase of the number of weights to be learned in a network; while the matter of the number of parameters in the network hasn't been discussed in this paper, networks need a number of learnable parameters coherent with that data available for training, making the choice of structure and hyperparameters fundamental for the performance of ANNs.

Acknowledgments

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

Bibliography

Alzubaidi et *al.*, Review of deep learning: concepts, CNN architectures, challenges, applications, future directions, in the *Journal of Big Data*, 2021.

Karen Simonyan, Andrew Zisserman, Very Deep Convolutional Networks for Large-Scale Image Recognition, *International Conference on Learning Representations (ICLR)*, 2015.

Michael Nielsen, Chapter 1 - Using neural nets to recognize handwritten digits, in *Neural Networks and Deep Learning* , 2019.

Qiuhong Ke, Jun Liu, Mohammed Bennamoun, Senjian An, Ferdous Sohel, Farid Boussaid, Chapter 5 - Computer Vision for Human–Machine Interaction, in *Computer Vision for Assistive Healthcare*, pages 127-145, Academic Press, 2018.

Appendix A

DropBatch256 RMSprop, 5-Fold Cross Validation		
Fold	Zero-One Loss	Accuracy
Fold 1	0.0525	0.9474
Fold 2	0.0587	0.9412
Fold 3	0.0569	0.9430
Fold 4	0.0611	0.9388
Fold 5	0.0543	0.9456

Table 14: DropBatch256 5-fold results by fold, RMSprop with no variations

DropBatch256 ADAM, 5-Fold Cross Validation		
Fold	Zero-One Loss	Accuracy
Fold 1	0.0609	0.9390
Fold 2	0.0579	0.9420
Fold 3	0.0521	0.9478
Fold 4	0.0631	0.9368
Fold 5	0.0535	0.9464

Table 15: DropBatch256 5-fold results by fold, ADAM

DropBatch256 RMSprop Leaky ReLU, 5-Fold Cross Validation		
Fold	Zero-One Loss	Accuracy
Fold 1	0.0494	0.9505
Fold 2	0.0529	0.9470
Fold 3	0.0573	0.9426
Fold 4	0.0653	0.9346
Fold 5	0.0543	0.9456

Table 16: DropBatch256 5-fold results by fold, RMSprop with Leaky ReLU

Figure 10: Sample of 9 images with corresponding Predicted Label and True Label in parenthesis, done with BatchNorm256 in its base form after 5-cross validation

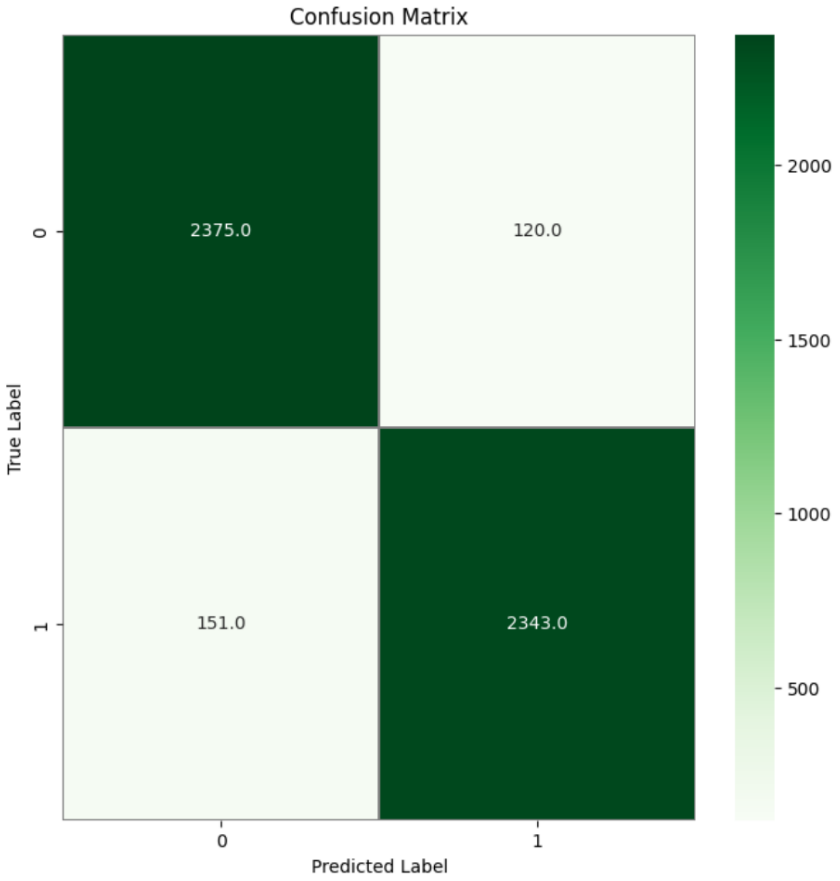

Figure 11: Confusion Matrix for BatchNorm256 after 5-fold cross validation

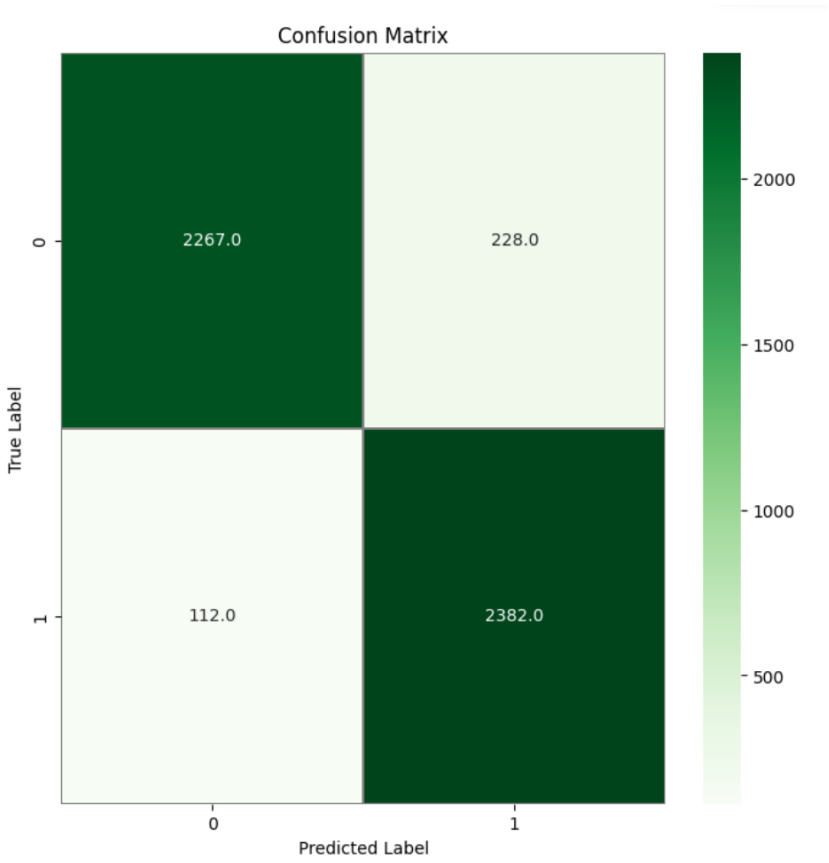

Figure 12: Confusion Matrix for BatchNorm256 after 5-fold cross validation, Leaky ReLU variation