UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE POLITICHE,
ECONOMICHE E SOCIALI

CNN: BINARY CLASSIFICATION

Machine Learning Module

Simona Caruso

A.A. 2023 - 2024

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 offenses 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.

Contents

1 Introduction

Binary classification problems are prevalent in the field of Machine Learning, and there are multiple algorithms available to solve these problems. Convolutional Neural Networks (CNNs) are commonly employed for image recognition and classification, representing a form of deep learning. In essence, convolution is the process of extracting features from an image by altering it. Through multiple convolution processes, the most significant features can be extracted to ultimately classify the images.
This is just a brief description of the process: the paper will delve into the details of the CNNs functioning through a common practical exercise: distinguishing between images of chihuahuas and muffins. Given that this project is for academic purposes, I opted to create simple models instead of using existing architectures in order to understand how manipulating inputs and modifying the structure of CNN impact the final results.

2 Understanding of Convolutional Neural Network

Images are three-dimensional arrays of pixels defined by height (h), width (w), and depth (d). Height and width vary depending on the image size, while depth refers to whether the image is in grayscale (d=1) or RGB color (d=3). The pixel values, regardless of the image format, typically range from 0 to 255. However, due to the wide range of these values, to simplify the optimization processes within the CNN and handle color scale variations more efficiently, pixel values are normalized. This normalization ensures that their values are in a range between 0 and 1.
Convolutional neural network architecture consists of:

- Convolution Layers.
- Pooling Layers.
- Dense Layers.

Let's examine the functioning of each of them below.

2.1 Convolution Layers

As previously mentioned in the introduction, the convolution operation is employed to extract features from an image. This extraction is achieved using a filter matrix (kernel), typically with dimensions 3×3. The image matrix is divided into submatrices of the same size as the kernel. For each submatrix, the dot product with the kernel is calculated; after performing the scalar product, a bias term is added, allowing us to handle potential translations or shifts in the data. Once this series of operations is completed, we obtain a feature matrix. The values within the filter and the biases are initially set randomly and then optimized through the backpropagation process, which we will discuss later.

As we've just explained, the matrix representing the image is divided into submatrices. The number of these submatrices depends not only on the kernel size but also on two other factors: the stride and the padding. The stride indicates how many pixels the submatrices should be spaced apart, while padding refers to the potential addition of pixels to the input

image. Typically, these pixels are symmetrically added on all sides of the image and their value is equal to 0. However, in practice, various types of padding exist. It's worth noting that padding can also be applied in subsequent layers with the goal of avoiding excessively small feature maps and, consequently, preventing information loss.

After obtaining the feature maps, an activation function is applied to introduce non-linearity into the model. The activation function most commonly used in this context is the Rectified Linear Unit (ReLU), defined by $f(x) = \max(0, x)$. This implies that negative values are set to 0, while positive values remain unchanged.

2.2 Pooling Layers

Pooling layers are used to reduce the dimensions of feature maps obtained from convolution layers while preserving essential information. Pooling can be thought as an empty $n \times n$ matrix overlaid on fixed-size subsections of the feature map. There are three types of pooling:

- Max Pooling: the pooling matrix is filled with the maximum values present in the subsections of the feature map;
- Average Pooling: the pooling matrix is filled with the average values of the subsections of the feature map;
- Sum Pooling: the pooling matrix is filled with the sum of the values in the subsections of the feature map.

2.3 Dense Layers

The pooled feature map or pooled feature maps are flattened into a single column through the operation known as flattening. The individual nodes then become the inputs of a classical neural network characterized by one or more fully connected layers, which include weights, biases, and activation functions. In this case as well, the weights and biases are optimized through backpropagation. The last layer is then connected to a single output node associated with the sigmoid activation function, determining the probability that the image belongs to one class rather than the other.

3 Backpropagation

As described in the previous section, in a Convolutional Neural Network, there are parameters that play a crucial role in the model's quality. In the initial phase of training, these parameters are randomly initialized. In this initial configuration, the model performs image classification and computes the loss function, representing how much the model's predictions deviate from reality. This loss function needs to be minimized, and this is where backpropagation comes into play.

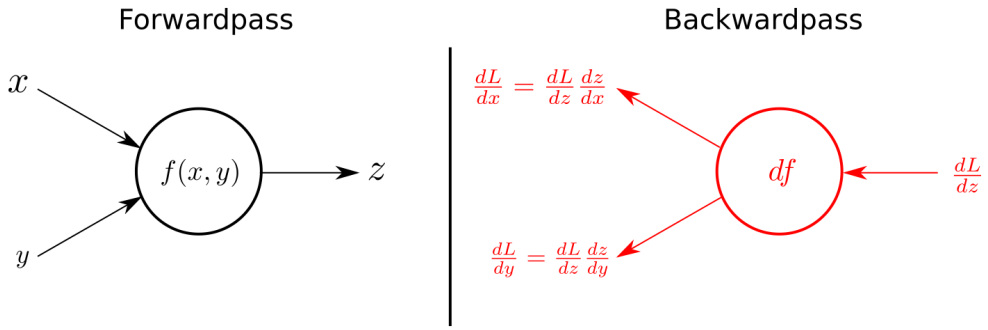

Figure 1: The forward pass on the left calculates z as a function $f(x, y)$ using the input variables x and y. The right side of the figures shows the backward pass. Receiving $\delta L/\delta z$, the gradient of the loss function with respect to z from above, the gradients of x and y on the loss function can be calculate by applying the chain rule, as shown in the figure. Source: *Understanding the backward pass through Batch Normalization Layer*.

The backpropagation is named as such because the parameter values are computed through a procedure that starts from the end of the neural network: for each parameter, the gradient descent is calculated, which is the derivative of the loss function with respect to the parameter. This derivative is computed using the chain rule as shown in Figure 1.

The new value of the parameter is determined by the difference between the value taken in the previous step and the product of the derivative value and the learning rate. This allows us to move along the loss function. In the initial phase, the learning rate will be relatively high, but after several steps, as we approach the minimum of the function, it will assume relatively low values.

In this project, however, the Adam optimizer was employed due to its notable effectiveness. Without delving too deeply into the technical details, we can describe Adam as an algorithm capable of dynamically adapting learning rates for various weights in the neural network through the computation of a moving average of the gradient and its corresponding variance.

4 Convolutional Neural Network in practice: Muffin or Chihuahua

In this section, we will put into practice the concepts discussed so far, providing additional insights specifically for binary classification. We will delve deeper into the previously mentioned topics, focusing on the initial data manipulation and the metrics used to manage the bias-variance trade-off. Subsequently, we will identify the optimal architecture for a CNN, perform hyperparameter tuning to select the best hyperparameters, and assess the model's effectiveness using the 5-fold cross-validation technique.

4.1 Data pre-processing

After importing the dataset containing images, we checked for class imbalance by counting the number of Chihuahua and Muffin images. There are 3199 Chihuahua images and 2718 muffin images, indicating a prevalence of the Chihuahua class. However, this imbalance doesn't seem excessive and shouldn't pose a significant issue.
We organized the data into a dataframe, specifying the path and classification of each image. This dataframe was then split into three sets:

- Training set: 75% of the images used for model training.
- Validation set: 10% of the images used to identify underfitting or overfitting situations.
- Test set: 20% of the observations used to measure the predictive ability of the selected model.

From these sets, we created image generators using the Keras `ImageDataGenerator` class with normalized pixels. The `flow_from_directory` method allows us to specify various details of the generator:

- `dataframe`: the reference pandas dataframe.
- `directory`: the path containing the images.
- `seed`: for generator reproducibility.
- `x_col`: the dataframe column containing file names.
- `y_col`: the dataframe column containing image classes.
- `target_size`: tuple specifying the height and width of the images.
- `batch_size`: the number of splits performed by the image generator. This split is crucial during model training for optimizing parameters based on the loss function calculated for each batch. The process repeats until a stopping criterion is met. The split is also done for the test set for efficiency, calculating the loss function for each batch and averaging for performance evaluation.
- `class_mode`: the type of classes.
- `drop_duplicates`: set to True by default, to eliminate duplicates based on file names.
- `color_mode`: specifies image dimensions (rgb or greyscale).
- `shuffle`: shuffles images after all batches have been examined and equal to True by default.

In our project, for all 3 generators, the target size is set to (224, 224), the batch size is 16, class mode is set to "binary", and color mode is set to "rgb". For the training and validation generators, shuffle is enabled (set to True), while for the test generator, it's disabled (set to False).

In addition to these three generators, two additional generators were created for the training and validation images, incorporating not only normalization but also various modifications. These include random rotation within a range of -10 to +10 degrees, width shift of up to 20% both to the right and left, height shift of up to 30% both upwards and downwards, a stretch transformation of 30%, a zoom range of 30%, channel color shift (expressed in pixel intensity) of up to 10, and random horizontal flipping. This image transformation technique is known as data augmentation and is employed to mitigate potential overfitting situations.

In Figure 2 and Figure 3, we visualize the images from the two training generators along with their respective binary classification (0 = chihuahua; 1 = muffin).

Figure 2: Normalized training pictures

Figure 3: Normalized and augmented training pictures

4.2 Metrics to evaluate models and stop criteria

Before proceeding with the introduction of the models, we will focus on the metrics used to evaluate the model fit and the set stopping criterion.

The choice of a model is generally made through the examination of learning curve plots, specifically the loss vs. epoch and accuracy vs. epoch. The loss function, in our case of binary classification, is given by binary cross-entropy (1), allowing us to measure the deviation between the predicted probability distribution and the true labels of the dataset. This function emphasizes errors, especially when the class probability associated with the observation significantly deviates from the true label. High values of binary cross-entropy indicate poor model performance, while low values indicate good model adaptability.

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(p_i) + (1-y_i)\log(1-p_i)\right) \tag{1}$$

On the other hand, accuracy is simply the proportion of correctly classified observations.

A CNN is fed by two datasets: the training set and the validation set. By examining the same metrics calculated on the validation data, we can understand the model fit:

- Underfitting: the training loss curve shows an increasing trend, while the validation loss curve is higher and shows no decreasing trend.
- Overfitting: The training loss curve is low while the validation loss is much higher.
- Good Fit: Both curves are decreasing and are close to each other.

This approach helps us interpret how well the model is adapting to the data during the training process.

The behavior of the curves is observed over multiple epochs: if there are too few epochs, the model may not learn enough and suffer from underfitting, while too many epochs may cause the model to memorize the training data too well, leading to overfitting. To prevent overfitting, we implement early stopping based on the error. Specifically, if the training loss stops decreasing for five consecutive epochs, we stop the training process, ensuring that the model doesn't become overly specialized to the training data. In our project, we set the maximum number of epochs to 50 to balance between training time and model performance.

4.3 Model 1: Base Model

This first model adopts a straightforward architecture with the following layers:

- **First convolutional layer**:
  - 32 filters of size 3×3 with padding to maintain the output matrix size.
  - Activation function: ReLU.
- **First pooling layer**:

  - Pooling size of 2×2 with a stride of 2.

- **Second convolutional layer**:
  - 64 filters of size 3×3, using the same padding settings as the first convolutional layer.
- **Second pooling layer**:
  - Same dimensions as the first pooling layer.
- **Dense layer**:
  - Layer with 64 neurons and ReLU activation.
- **Final layer**:
  - 1 neuron with sigmoid activation for output.

The input data utilized consists of normalized images. By fitting the model, results were obtained after 21 epochs, as shown in Figures 4 and 5, respectively.

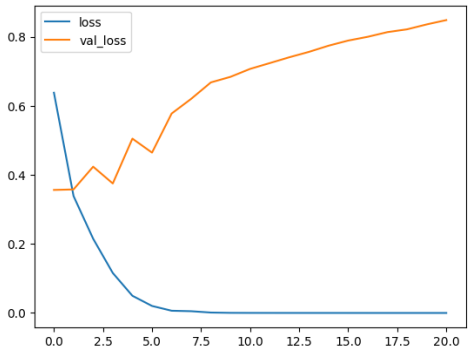

Figure 4: Loss and Validation Loss for Model 1

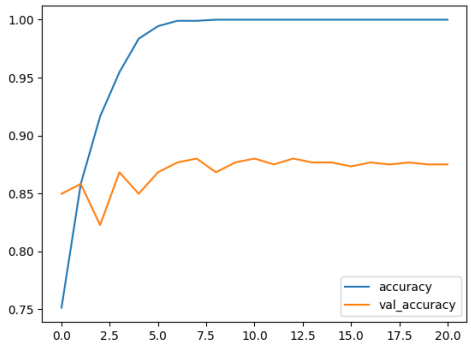

Figure 5: Accuracy and Validation Accuracy for Model 1

The issue of overfitting is evident: this problem could be related either to the excessive simplicity of the CNN architecture or to the model's excessive adaptation to the details of the training images.

4.4 Model 2: Data Augmentation

The structure of the second model is identical to that of the first, but the input images for fitting are the augmented images. Through augmentation, the problem of overfitting could be addressed since the images undergo realistic transformations, and therefore, the details "memorized" by the model are more extensive and complex. Let's observe the model's behavior on these modified images in Figures 6 and 7.
We can observe a significant improvement obtained, and we could already be satisfied with these results. However, we have decided to present additional models that involve further complexities so that, in case users want to replicate the codes on different data and the results obtained with only data augmentation are not satisfactory, they will have the option to apply additional techniques for improvements.

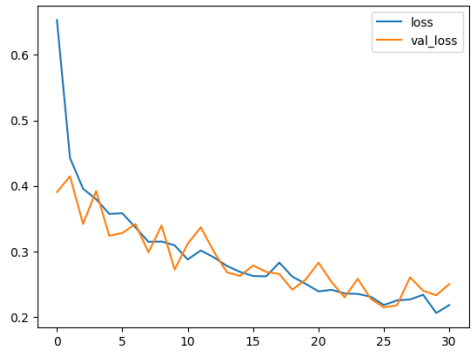

Figure 6: Loss and Validation Loss for Model 2

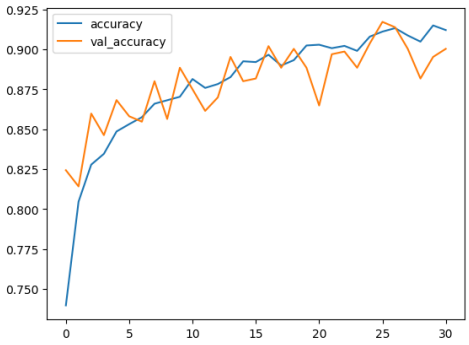

Figure 7: Accuracy and Validation Accuracy for Model 2

4.5 Model 3: Dropout Layers

This third model incorporates some small modifications to the CNN architecture. As discussed in the preceding models and the introduction of this paper, the model's states include convolutional, pooling, and fully connected layers. Moreover, we will delve into an additional layer known as dropout layer.

To mitigate the model's dependence on the training data and address potential overfitting issues, dropout layers can be introduced within the traditional CNN architecture. During training, a portion of neurons is randomly deactivated, preventing the model from overly focusing on image details. The percentage of neurons to be deactivated is determined during the architecture construction phase.

The architecture of the third model is the same as the first and second. However, after each max-pooling layer, a dropout layer has been added with a deactivation of 25% of neurons, and an additional dropout layer with a deactivation of 30% of neurons before the final output layer. The fitting results are visible in Figures 8 and 9.

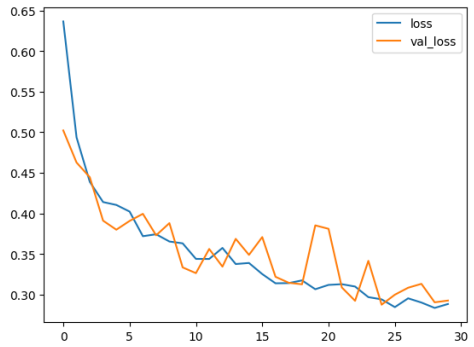

Figure 8: Loss and Validation Loss for Model 3

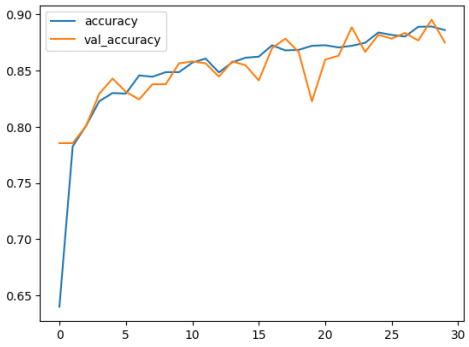

Figure 9: Accuracy and Validation Accuracy for Model 3

In this case as well, it seems there is no overfitting. However, the results obtained in this third model are worse compared to the second one in terms of loss and accuracy.

4.6 Model 4: Final Model

The fourth model involves the addition of three more layers compared to the third one. Before flattening, we have:

- **First convolutional layer**:
  - 128 filters of size 3×3 with padding set to `'same'` and ReLU activation function.
- **First pooling layer**:
  - Pooling size of 2×2 and strides=2.
- **Dropout layer**:
  - Deactivation of 25% of neurons.

The fitting results are available in Figures 10 and 11.

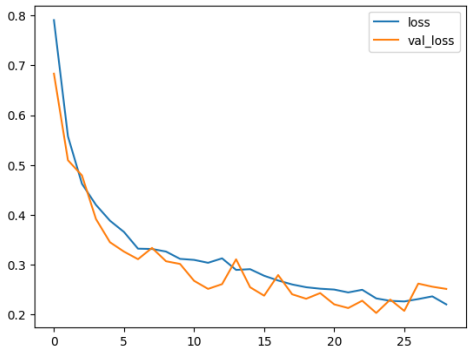

Figure 10: Loss and Validation Loss for Model 4

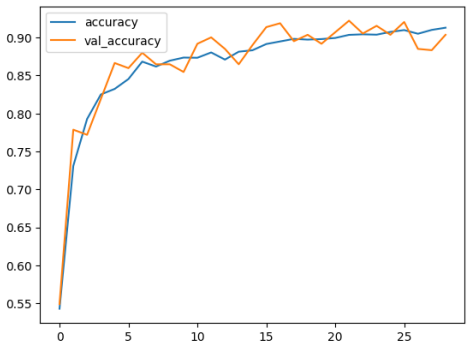

Figure 11: Accuracy and Validation Accuracy for Model 4

The hyperparameters for all models were randomly set by me, but this is certainly not the best choice. In the next chapter, we will present the results of hyperparameter tuning applied to the architecture of this last model, as it has produced the most satisfactory results.

5 Hyperparameter Tuning

Hyperparameter tuning is crucial to search for optimal values of hyperparameters. For this task, the KerasTuner library is employed.

The first step involves defining a function for building the model. In this function, the architecture follows the structure of the fourth model, but the hyperparameters are not predefined. Instead, they are set to specific ranges with defined step sizes.

The model configuration includes the following steps:

- **First Convolutional Layer:**

  - Number of filters: ranged from 16 to 64 with a step size of 16.
  - Fixed kernel size: 3×3.
  - Activation function: ReLu.
- **First Pooling Layer:**
  - Pooling size: 2×2.
- **First Dropout Layer:**
  - Proportion of dropped neurons: ranged from 0.05 to 0.3 with a step size of 0.05.
- **Second Convolutional Layer:**
  - Number of filters: ranged from 32 to 128 with a step size of 32.
  - Fixed kernel size: 3×3.
  - Activation function: ReLu.
- **Second Pooling Layer:**
  - Pooling size: 2×2.
- **Second Dropout Layer:**
  - Proportion of dropped neurons: ranged from 0.05 to 0.3 with a step size of 0.05.
- **Third Convolutional Layer:**
  - Number of filters: ranged from 64 to 256 with a step size of 64.
  - Fixed kernel size: 3×3.
  - Activation function: ReLu.
- **Third Pooling Layer:**
  - Pooling size: 2×2.
- **Third Dropout Layer:**
  - Proportion of dropped neurons: ranged from 0.05 to 0.3 with a step size of 0.05.
- **First Dense Layer:**
  - Number of neurons: ranged from 32 to 256 with a step size of 32.
  - Activation function: ReLu.
- **First Dense Dropout Layer:**
  - Fraction of dropped neurons: ranged from 0.2 to 0.5 with a step size of 0.1.
- **Output Layer:**
  - Single fixed neuron.
  - Activation function: Sigmoid.

This setup allows for defining a wide range of hyperparameter configurations to effectively explore the hyperparameter space. This process helps in finding the optimal combination that maximizes the model's performance.

The hyperparameter search could also be extended to explore whether to use dropout layers, determine optimal activation functions, and define the ideal learning rate. However, since hyperparameter tuning is costly both in terms of time and computational resources, we have chosen to restrict the search to a limited number of hyperparameters.

Selecting a tuner class for optimization, we seek the ideal values for hyperparameters. Various tuning algorithms exist, but we have chosen Bayesian optimization. With the goal of maximizing the validation accuracy function, the Bayesian optimization process works as follows:

1. Model execution and Validation Accuracy measurement: Initially, the model is trained using a random set of hyperparameters, and its performance is evaluated on the validation set through validation accuracy.

2. Surrogate Function creation: Using the results obtained from initial experiments, a surrogate function (often a Gaussian Process) is constructed to approximate the relationship between hyperparameters and validation accuracy.

3. Selection of new hyperparameters: Utilizing an acquisition function, new values for hyperparameters are chosen. This function guides the search to balance exploration (examining new regions) and exploitation (exploiting regions with promising objective function values).

4. Convergence and stop criteria: The process of selecting new hyperparameters through the surrogate function is iterated until convergence or a predetermined stop criterion is reached. Once the process is completed, the different sets of tested hyperparameters are evaluated, and the one that maximizes the objective function is chosen, depending on the optimization goal. In case convergence or the stop criterion is not reached, steps 2, 3, and 4 are iteratively repeated using the hyperparameters selected in step 3 to construct the surrogate function in step 2.

The maximum number of iterations we selected is 5, and at the end of the search process, we obtained the following values for the hyperparameters:

- **First Convolutional Layer:**
  - Number of filters: 16
- **First Dropout Layer:**
  - Proportion of dropped neurons: 0.1
- **Second Convolutional Layer:**
  - Number of filters: 64
- **Second Dropout Layer:**

  - Proportion of dropped neurons: 0.1

- **Third Convolutional Layer:**
  - Number of filters: 64
- **Third Dropout Layer:**
  - Proportion of dropped neurons: 0.1
- **First Dense Layer:**
  - Number of neurons: 192
- **First Dense Dropout Layer:**
  - Fraction of dropped neurons: 0.3

We then evaluate the model results by observing Figures 12 and 13.

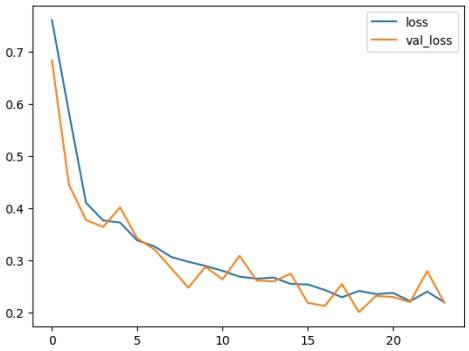

Figure 12: Loss and Validation Loss for the Tuned Model

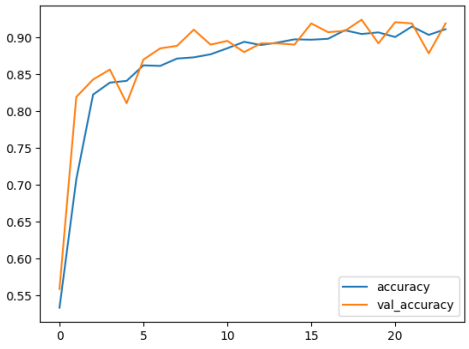

Figure 13: Accuracy and Validation Accuracy for the Tuned Model

The model exhibits a good fit, hence, we can proceed with evaluating the results using the test set.

6 Predictions on test set

The `predict()` function allows us to obtain labels for the test images using the trained model. The result is an array of the same length as the test set with values ranging from 0 to 1, representing the probability of belonging to the class labeled as 1 (muffin). A threshold value of 0.5 has been set, so if the probability associated with the sample is less than 0.5, the image will be labeled as 0 (chihuahua), while if it is equal to or greater than 0.5, the image will be labeled as 1 (muffin). Predictions are then compared with the actual classes to obtain accuracy metrics. Note that chihuahuas, labeled as 0, are considered the negative class, while muffins, labeled as 1, are defined as the positive class.

6.1 Confusion Matrix and Metrics for Model evaluation

The confusion matrix provides an immediate visualization to quantify correctly classified and misclassified images.

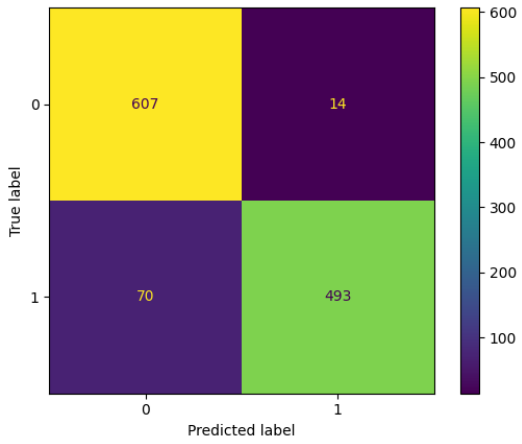

Figure 14: Confusion Matrix for test images' predictions

From Figure 14, we can observe the following:

- 607 images of Chihuahua were correctly classified as such. (True Negative = TN)
- 14 images of Chihuahua were misclassified as muffins. (False Positive = FP)
- 493 images of muffins were correctly classified as such. (True Positive = TP)
- 70 images of muffins were misclassified as Chihuahua. (False Negative = FN)

Furthermore, we can utilize some metrics to help us understand the quality of classification. Accuracy is a measure that allows us to understand how often the model predicts correctly and is represented by the proportion of correct classifications over the total predictions. Precision allows us to understand how often a single class is predicted correctly and is given by the fraction of the total number of observations correctly classified for a given class over the total number of observations classified as belonging to the class. Recall is a metric used to measure how good the model is at identifying all instances belonging to a single class and is given by the fraction of correct classifications for a single class over the total number of instances belonging to that class. Eventually we have the F1-score, which is a combination of precision and recall and computes how many times the model predicted instances correctly across the entire dataset.

We present the results of the metrics measured on our predictions in Table 1. Overall, we can say that the model's performance is good, although we observe that the precision for muffins is higher than that for chihuahuas, indicating that our model tends to predict the class of chihuahuas more easily. Similarly, for recall, we have a lower value for the muffin class, precisely because, as just mentioned, our model has a slight tendency to misclassify

images of muffins as chihuahuas. Nevertheless, the metric values are good, and we can conclude that the achieved model is a solid one.

	precision	recall	f1-score	support
chihuahua	0.90	0.98	0.94	621
muffin	0.97	0.88	0.92	563
accuracy			0.93	1184

Table 1: Classsification Metrics

6.2 Sensitivity, Specifity and ROC Curve

To assess whether the model effectively distinguishes between classes regardless of the probability threshold, the Receiver Operating Characteristic (ROC) curve, which relates the false positive rate to the true positive rate, and the Area Under the Curve (AUC) are employed.

The True Positive Rate (TPR), also known as recall, is defined as:

$$Sensitivity = Recall = \frac{TP}{TP + FN} \tag{2}$$

The False Positive Rate (FPR) corresponds to 1 - Specificity:

$$Specificity = \frac{TN}{FP + TN} \tag{3}$$

$$FPR = 1 - Specifity \tag{4}$$

If the classes are perfectly distinguished, meaning probabilities are all associated with 0 for chihuahua images and 1 for muffin images, then regardless of the threshold, classifications will be correct. Consequently, the ROC curve will exhibit an upper-left corner, indicating that when the FPR is 0, the TPR is 1. (Figure 15)

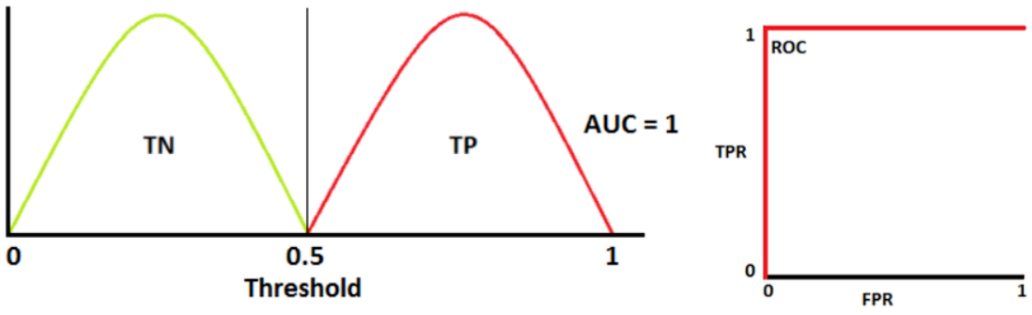

Figure 15: ROC Curve for a perfect classifier

In cases where some probabilities associated with images are not well-defined, meaning they are distant from 0 and 1, the classification will depend on the threshold chosen. Consequently, increasing the TPR might also result in an increase in the FPR. This is because an increase in TPR signifies the model correctly classifies more positive cases, but concurrently, it might start incorrectly classifying some negative cases as positive (false positives). (Figure 16)

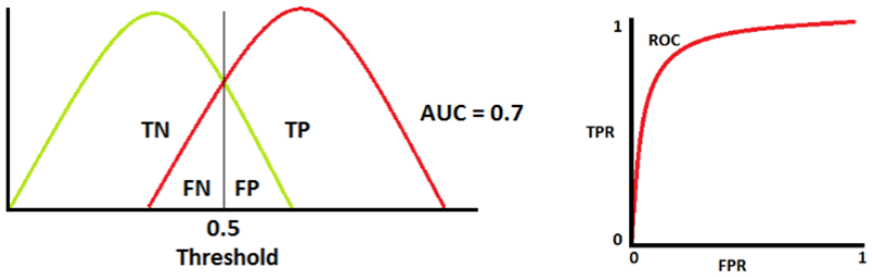

Figure 16: ROC Curve for overlapping predictions given different thresholds

The worst-case scenario is that of a random model, where all probabilities associated with the images are centered around 0.5. In this case, regardless of the probability threshold chosen, the TPR and the False Positive Rate FPR will be proportional. (Figure 17)

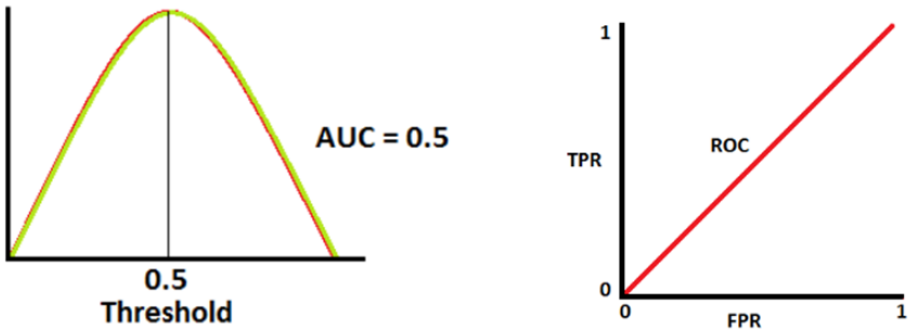

Figure 17: ROC Curve for a Random Model

In general, we observe that a larger area under the curve corresponds to a better classifier. This area is measured by the AUC, which ideally has a value of 1. In our case, the model appears to be robust: as depicted in Figure 18, the curve approaches the upper-left corner, and the AUC is determined to be 0.98.

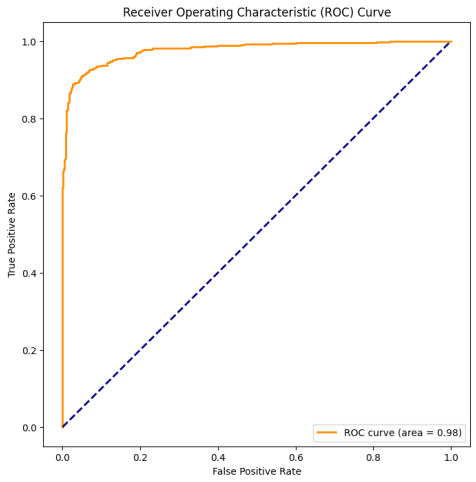

Figure 18: Roc Curve for the Tuned Model

7 Cross Validation

k-fold cross-validation implies the division of the training set into k subsets called folds. The selected model is then trained k times, each time using k-1 folds as the training set and the remaining fold as the validation set. In each iteration, a different fold serves as the validation set. Through cross-validation, we ensure that the model architecture performs well even when exposed to diverse input images. Below the results obtained from cross-validation:

k	Validation Accuracy	Validation Loss
1	0.8843	0.3101
2	0.9172	0.1959
3	0.9180	0.2000
4	0.9290	0.1808
5	0.9145	0.2167
Average	0.9126	0.2205

Table 2: *5-fold* Cross Validation results

The average validation loss in the first iteration of k-fold cross-validation is higher than in the others. This disparity may be due to chance, given that results across the other folds are more consistent. However, it is crucial to note that the overall average of validation loss and validation accuracy aligns with the results obtained from our tuned model. Consequently, we can confidently confirm the robustness of the model.