Neural Networks

Machine Learning Project

Tatiana Getling
t.getling@studenti.unimi.it
Matricola 963368

Table of Contents

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.

Introduction

The aim of this project is to develop a neural network classifier capable of distinguishing between images of muffins and Chihuahuas. The main objectives also include assessing and comparing the effectiveness of different neural network architectures and optimization of the training hyperparameters for optimal performance.

The first chapter summarizes the nature of the data and explains the preprocessing steps of the data preparation.

In the second chapter we introduce the Convolutional Neural Networks and provide a brief overview of their architecture and main components.

The third chapter focuses on the the model selection and discusses the approach used for selecting different models, including variations in layer depth, kernel size, activation functions and more.

The fourth chapter focuses on the training methodology, including the optimization algorithm, learning rate schedule, and regularization techniques employed. We discuss the impact of these choices on training convergence, model generalization, and computational efficiency.

In the fifth chapter we present the experimental results of our models and evaluate their performance on both the training and validation datasets. We analyze various metrics in order to provide insights into model's performance.

The final chapter of the report will summarize the results, discussing the strengths and weaknesses of the selected approach, as well as identify potential areas for improvement.

Data Preparation and Approach

The dataset used for training is provided by Kaggle (**Cortinhas, 2020**) and consists of images of Chihuahuas and muffins, separated into training (4733 images) and test (1184 images) folders. Each folder is further divided into 2 subfolders, consisting solely of images for one of the two classes (muffin/ Chihuahua), where each of the images is stored in JPG format.

Normalization and standardization

The first step of the data preparation consisted in rescaling the images to a standard size (128*128) in order to guarantee a uniform input. Furthermore, the images were transformed into RGB pixel values and normalized to a common scale (ranging from 0 to 1) in order to facilitate convergence during training.

To control the loading process and ensure that it has completed successfully and as expected, we have previewed the resulting set of the preprocessed images (combined from both subfolders in the training folder):

Fig.1 - Randomly selected images from the original (concatenated) set

Data augmentation

In order to introduce more variation to the dataset, it was decided to further perform data augmentation, which could potentially help avoid

overfitting and enhance the model's ability to generalize on unseen data, thus improving the performance of the model.

The data augmentation, performed using ImageDataGenerator capabilities, included introducing rotation, flipping, translation and shuffling. We have further previewed a subset of the resulting augmented dataset to visually compare the images with the original set.

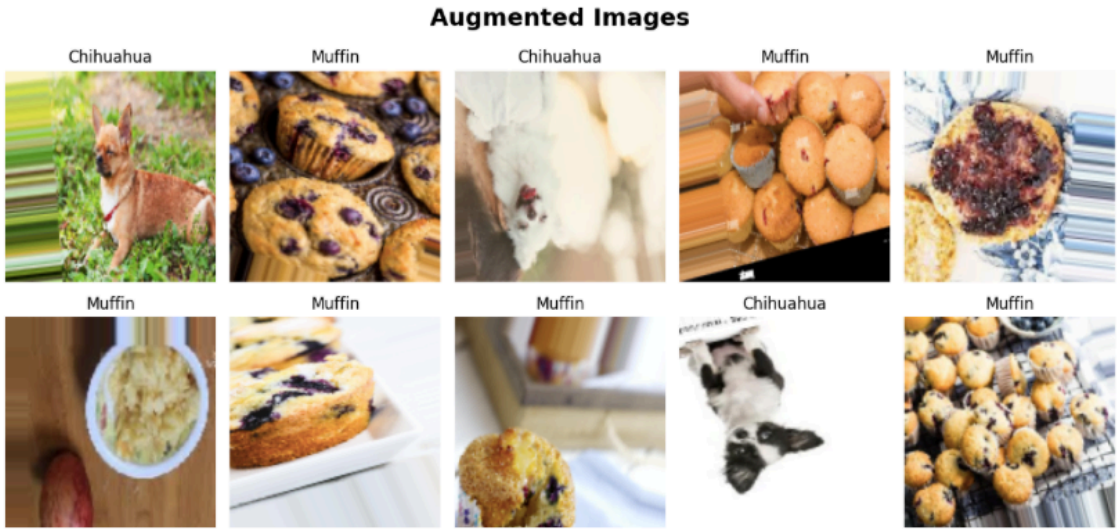

Fig. 2 - Randomly selected images from the augmented set

The resulting training dataset was further split into training and validation sets to be able to evaluate the models' performance during the training. The dataset was split with the 80/20 ratio, with 20% of the set being used for validation. To do so, we first determined the total number of available augmented samples. Additionally, the labels, which were initially one-hot encoded, were converted to binary format to maintain consistency in the data shapes and facilitate the evaluation process.

Overview of Convolutional Neural Networks

The selected architecture is a Convolutional Neural Network (CNN). CNNs are often used for similar tasks, as they have proven to be extremely effective for image, speech and audio inputs. A typical CNN architecture consists of multiple layers, including convolutional layers for feature extraction, pooling layers for dimensionality reduction, and fully connected (dense) layers for classification.

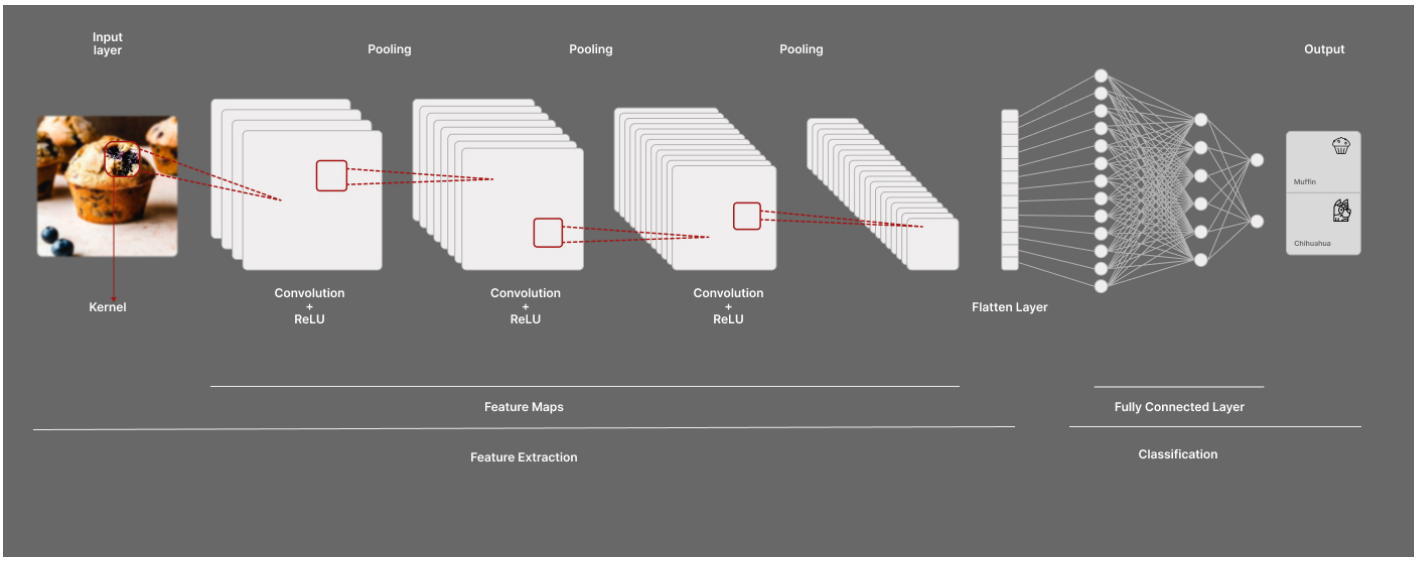

Fig. 3 - Schematic representation of a CNN structure

The first layer of a CNN is the convolutional layer, which may be followed by additional convolutional layer or pooling layer. The final layer of a CNN is the fully-connected layer. Each layer increases the complexity of the CNN: earlier layers focus on simple features, such as colors and edges. As the image data progresses through the layers of the CNN, it starts to recognize larger elements or shapes of the object until it finally identifies the intended object (**IBM, n.d.**).

The majority of the computation occurs within the **convolutional layer**, containing 3 main components: the input data, a filter, and a feature map. The input data is an RGB image represented as a three-dimensional (width, height, depth) matrix of pixel values. The filter, usually a small matrix (e.g., 3x3), «scans» the input image and computes a dot product between the filter and the input

pixels. The results form a feature map that highlights the presence of specific features.

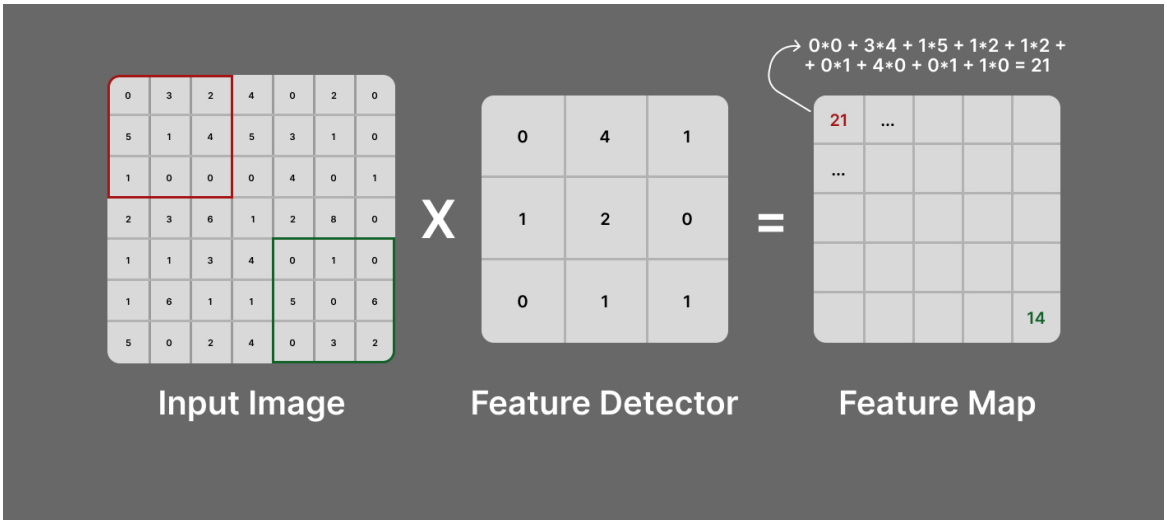

Fig. 4 - Schematic representation of the work of a filter

The hyperparameters in the convolutional layers that need to be considered before training include number of filters (smaller numbers might lead to underfitting, while larger values may increase the model complexity and training time), stride (the step size by which the filter moves across the input: a larger stride reduces the output size) and padding (valid/same/full). To introduce non-linearity, a Rectified Linear Unit (ReLU) activation function is applied after each convolution.

The hyperparameters in the convolutional layers that need to be considered before training include number of filters (smaller numbers might lead to underfitting, while larger values may increase the model complexity and training time), stride (the step size by which the filter moves across the input: a larger stride reduces the output size) and padding (valid/same/full). To introduce non-linearity, a Rectified Linear Unit (ReLU) activation function is applied after each convolution. Additional convolution layers create a hierarchical representation by building upon the feature maps from previous layers, thus enabling the network to recognize complex patterns.

The Pooling layers help reduce the spatial dimensions of the input volume, thus reducing the complexity and controlling overfitting of the model. **The Flatten Layer** is used to convert the 3D feature maps into a 1D feature vector, which can be further fed into the fully connected (dense) layers. In the **fully-connected** layer, every node in the output layer connects to every node in the previous layer. This layer integrates features detected by convolutional and pooling layers to perform classification.

Optimizers play a crucial role in training Convolutional Neural Networks. Based on assessing the particular task objectives, it was decided to choose the Adam (Adaptive Moment Estimation) optimizer, which is commonly used due to its adaptive learning rate capabilities. It computes adaptive learning rates for each parameter by estimating the first and second moments of the gradients. This results in faster convergence and greater efficiency, especially in problems with sparse gradients.

Another critical choice for a CNN is the choice of the **loss function**, as it directly influences the model's learning process by quantifying how well or poorly the model performs. For classification tasks, two primary loss functions are commonly used: categorical cross-entropy and binary cross-entropy. Within the project we focus on a binary classification task, which is why we consider binary cross-entropy as it directly compares the predicted probabilities with the true binary labels.

Model Selection

The model selection on the initial stage included selecting the number of layers, layer types, activation functions, adding dropout layers and batch normalization. The hyperparameter tuning included adjusting the learning rate, the batch size, the optimizer choice, the number of epochs and regularization.

It was decided to start with a simple model architecture and build on it, adjusting accordingly based on the training results.

The first model included an input layer, a single convolutional layer with 32 filters and ReLU activation function followed by a max pooling layer, a flatten layer, a fully connected layer with 128 neurons and ReLU as the activation function and the output layer with the sigmoid activation function. Starting with a moderate number of filters (32) in the convolutional layer allows the network to capture essential features without overwhelming the model with parameters. The 3x3 kernel size is a standard choice that effectively captures local patterns in the data. The Rectified Linear Unit (ReLU) activation function introduces non-linearity, allowing the network to learn complex patterns.

For the second model we consider a deeper neural network, which includes an additional convolutional layer with a larger number of filters, an additional pooling layer to help in further reducing the spatial dimensions and maintaining computational efficiency. We have further increased the number of neurons in the fully connected layer to allow for more complex feature combinations and added a dropout layer to prevent overfitting.

For the third model we have chosen an even deeper architecture, adding more convolutional layers and choosing larger values for the number of filters and adjusting the number of neurons in the dense layers. A deeper network can learn more abstract and detailed features. Each layer builds on the previous one, capturing increasingly complex patterns. As layers deepen, increasing the

filter count allows the network to capture a richer set of features at each level. A larger fully connected layer at the end of the network can capture more intricate combinations of features, which is beneficial for complex tasks. Dropout continues to serve as an important regularization technique to prevent overfitting, especially in a deeper network.

Further tuning of the initial models is to be evaluated based on the primary training results and comparison of the models' performance.

Result analysis and interpretation

In order to access further steps for tuning the initial models we have completed the first round of training and compared the training results. The first model (the simplest one) was trained on 10 epochs with batch size equal to 32 and the optimizer's learning rate is set as default (0.001). No early stopping was applied for the inital run of the training.

The model performed with the following results:

Metric	Training	Validation
Loss	0.00455216	0.773717
Accuracy	1	0.79704
Precision	1	0.7655351
Recall	1	0.804147

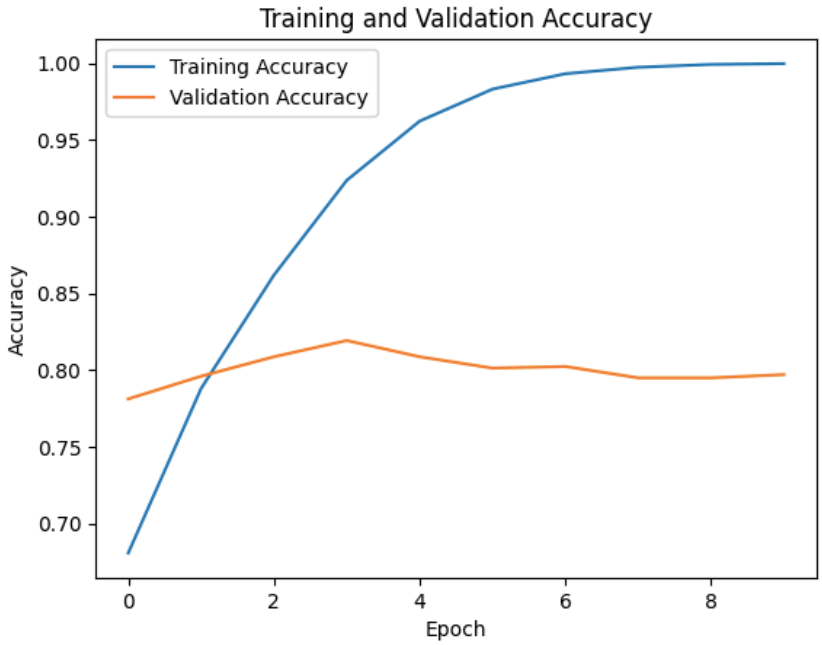

Fig. 5 - Graph representing the first model's training results

By assessing the different metrics of the model's performance as well as the learning curve over the epochs (Fig. 5), we can notice that while the model performs extremely well on the training set, the validation metrics show minor

fluctuations over the epochs and do not improve significantly. This data suggests that the model might be overfitting. In order to improve the performance we need to consider a more complex architecture, as well as introduce regularization techniques. The second model described in the previous chapter follows these requirements, so for the next round of model training we implement a more complex model containing more convolutional layers and a dropout layer, before evaluating the need for further adjustment of hyperparameters.

From the results of the training of the second model we can see that the validation accuracy, as well as other metrics, has improved significantly compared to the first model:

Metric	Training	Validation
Loss	0.0143081	0.581456
Accuracy	0.998944	0.831924
Precision	1	0.806236
Recall	0.997701	0.834101

These results show that a more complex model performs significantly better, so we further compare it with the third model, containing an additional convolutional layer with more filters followed by another pooling layer.

The third model showed improvements in loss, precision, and accuracy, but a decline in recall, indicating that while the model is better at avoiding false positives, it is missing more true positives.

Metric	Training	Validation
Loss	0.0861391	0.463102
Accuracy	0.979667	0.837209
Precision	0.989982	0.85533
Recall	0.965517	0.776498

This suggests the need for adjustments to balance precision and recall. To address this, we are going to adjust the learning rate, implement early stopping, and use model checkpoints to prevent overfitting. Additionally, we will increase the number of epochs to give the model more opportunity to learn, while monitoring performance to avoid overtraining.

The implementation of these changes improved significantly the performance of the model:

Metric	Training	Validation
Loss	0.0435834	0.410037
Accuracy	0.991022	0.862579
Precision	0.997086	0.868932
Recall	0.983333	0.824885

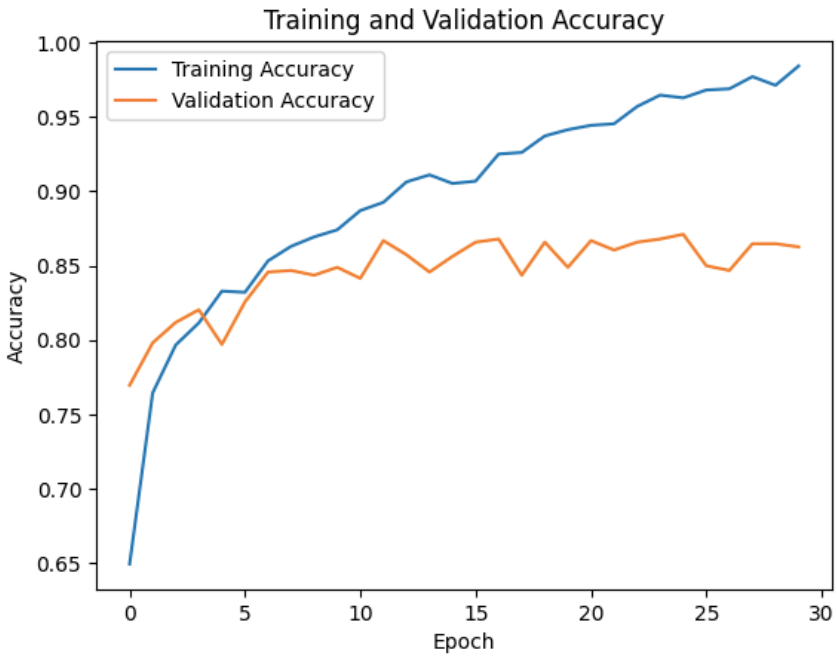

Fig. 6 - Graph representing the adjusted model's training results

As we can observe, the validation loss decreased, both accuracy and precision increased, while recall also showed improvement. The performance of this model seems promising, so we can proceed to perform the 5-fold cross-

validation to ensure that the model generalizes well across different data subsets.

5-fold Cross-Validation results	
Metric	**Validation**
Accuracy	0.8423561
Loss	0.3663232
Precision	0.8611047
Recall	0.7885057
Zero-One Loss	0.493741

Based on the cross-validation results, we can see that the model has good overall performance, with strong accuracy, precision, and recall metrics. The model is consistent across folds, suggesting it generalizes well. However, the value for the validation loss , even though not excessively high, could be improved. Considering the high values for accuracy and precision the model doesn't seem to be over- or under- fitting. The loss value indicates that there might be room for improvement in terms of reducing the prediction errors, so for the next step we are going to perform hyperparameter tuning to try and lower the validation loss, while also maintaining/improving the accuracy and precision.

For the hyperparameter tuning it was decided to use the hyperband algorithm. Hyperband is an adaptive method that combines random search and early stopping, by dynamically allocating resources to promising configurations based on their performance. It is implemented using the Keras Tuner library that contains some other hyperparameter tunes as well, such as Random Search, Sklearn and Bayesian optimization (**Tensorflow, n.d.**).

The best value for the validation accuracy found by the tuner gave us the value of 0.89006, which is a significantly better result compared to the previous models. We can see that the optimal model is achieved with 128 filters on the

first and third convolutional layers, 96 filters on the second convolutional layer, 64 filters on the dense layer, a dropout rate of 0.5 and a learning rate of 0.0001.

After training the model with the adjusted parameters, we can see that the accuracy has improved significantly, almost reaching 0.9 of the validation accuracy.

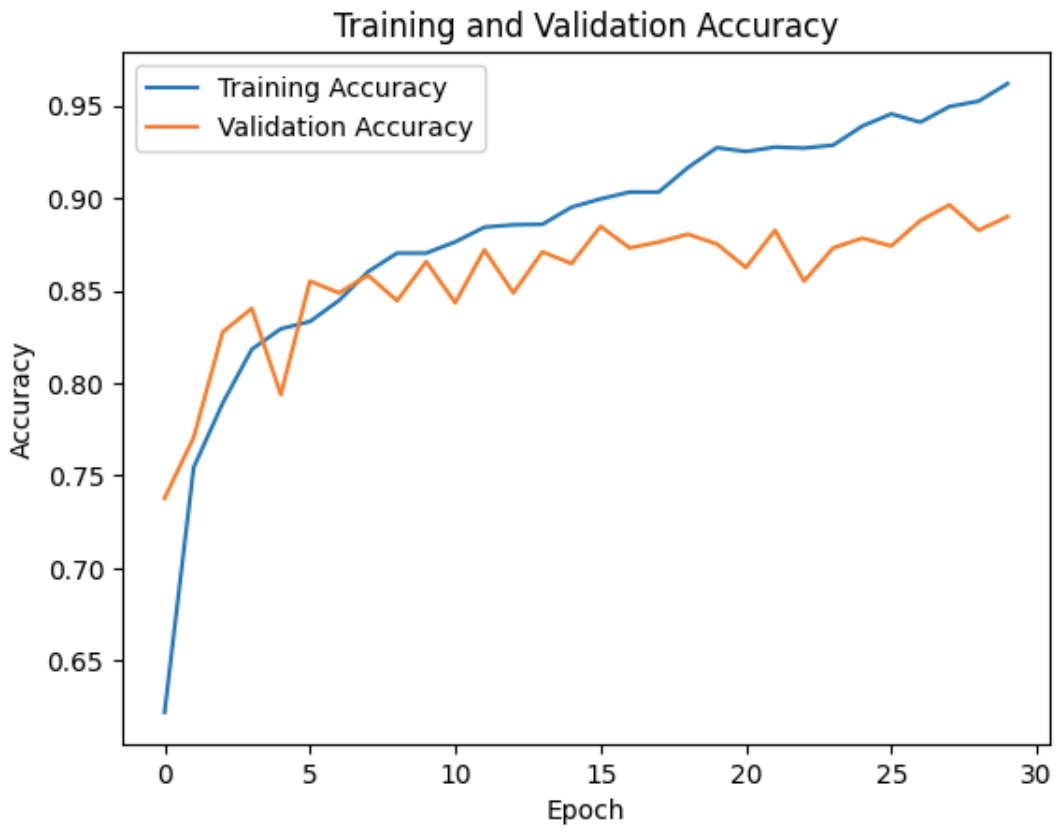

Fig.7 - Graph of the tuned model results

Since the results of this training look promising, we proceeded to perform the 5-fold cross-validation to make sure out model generalizes well.

Finally, we evaluated out model on the test data with the following results:

Metric	Result
Accuracy	0.8826013803482056
Precision	0.8558875322341919
Recall	0.8952205777168274
Zero-one loss	0.11739864945411682

Loss	0.3270859718322754

We can see that the model's accuracy remained quite high on the test data with the value of 0.88, while zero-one loss is consistent with the accuracy showing the result value of 0.11. This indicates to us that the tuned model outperformed our previous models and showed great capacity of generalizing on the unseen data, which means the objective if the project is met.

Conclusion

In this project we developed Convolutional Neural Network models for a binary classification task, experimenting with three different architectures of increasing complexity. We performed a preliminary evaluation of the models' performance on the training data, before performing 5-fold cross-validation on the most promising model with zero-one loss as one of the primary metrics along with accuracy and precision. We then used the Hyperband algorithm to fine-tune the model to try and improve the metrics observed during the cross-validation.

The final results for the selected model indicate that the model generalizes well to unseen data and has a promising performance with the accuracy of 0.88 and zero-one loss of 0.11.

Future work could involve exploring more advanced architectures and fine-tuning the model with a greater range of parameter values (which would require greater computational power), more layers, a larger number of epochs as well as additional hyperparameter optimization, for instance, using other hyparaparameter tuning algorithms, such as Grid Search or Bayesian Optimization, to further improve the accuracy of our model.

1. Cortinhas, S. (n.d.). *Muffin vs. Chihuahua Image Classification* [Data set]. Kaggle. Retrieved from https://www.kaggle.com/datasets/samuelcortinhas/muffin-vs-chihuahua-image-classification
2. IBM. (n.d.). *Convolutional neural networks*. Retrieved from https://www.ibm.com/topics/convolutional-neural-networks
3. Tensorflow. (n.d.). *Introduction to the Keras Tuner.* Retrieved from https://www.tensorflow.org/tutorials/keras/keras_tuner