UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE

DIPARTIMENTO DI INFORMATICA
GIOVANNI DEGLI ANTONI

CORSO DI LAUREA MAGISTRALE IN
INFORMATICA

STATISTICAL METHODS FOR MACHINE LEARNING
NEURAL NETWORK PROJECT REPORT

Matteo Farè
Matr. Nr. 989345

ACADEMIC YEAR 2024-2025

Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I 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.

Contents

Chapter 1

Introduction

Inspired by the complex structure of the human brain, **Artificial Neural Networks (ANNs)** stand as a foundational paradigm in artificial intelligence. This study deal with the problem of image classification, specifically addressing the challenge of **binary classification**—discerning between images of muffins and chihuahuas within a given dataset.

The project primarily makes use of the **Keras** and **Tensorflow** frameworks to set up, construct, and train neural network models. It starts with a preprocessing step to assess the quality of the dataset, identifying corrupted files and checking for the possible existence of duplicate images—both factors impacting the performance of the training models. The data then undergoes a preparation phase, which involves transformations such as adjusting the size and color format of the images. Additionally, processes like data augmentation are applied to enhance results and overall performance.

This preprocessing step sets the stage for the subsequent exploration of various network architectures. To address the task at hand, three distinct network architectures were used: The **Multilayer Perceptron** (MLP), **Convolutional Neural Network** (CNN) and **MobileNet** model.

Beyond architectural variations, this study incorporates **hyperparameter tuning** to enhance overall performance, with the resulting optimal parameters and weights saved and adopted as the basis input model for **k-fold cross-validation** to calculate risk estimates, employing the **zero-one loss** function. At the end, the models are evaluated on the test set to have a final understanding of their generalization capabilities.

1.1 Project Overview

The project has been built working with a dataset recovered from the Kaggle website [1]. This dataset includes two categories of images, namely Muffin and Chihuahua. Briefly, the primary objective is to establish neural network models proficient in classifying these two categories and subsequently evaluate their performance.

As illustrated in **Figure 1** below, the architecture of this study is fundamentally organized into four blocks. The initial two blocks are dedicated to preprocessing and data preparation tasks, whereas the latter two blocks are focused on models construction, specifically, the classification process and subsequent evaluation.

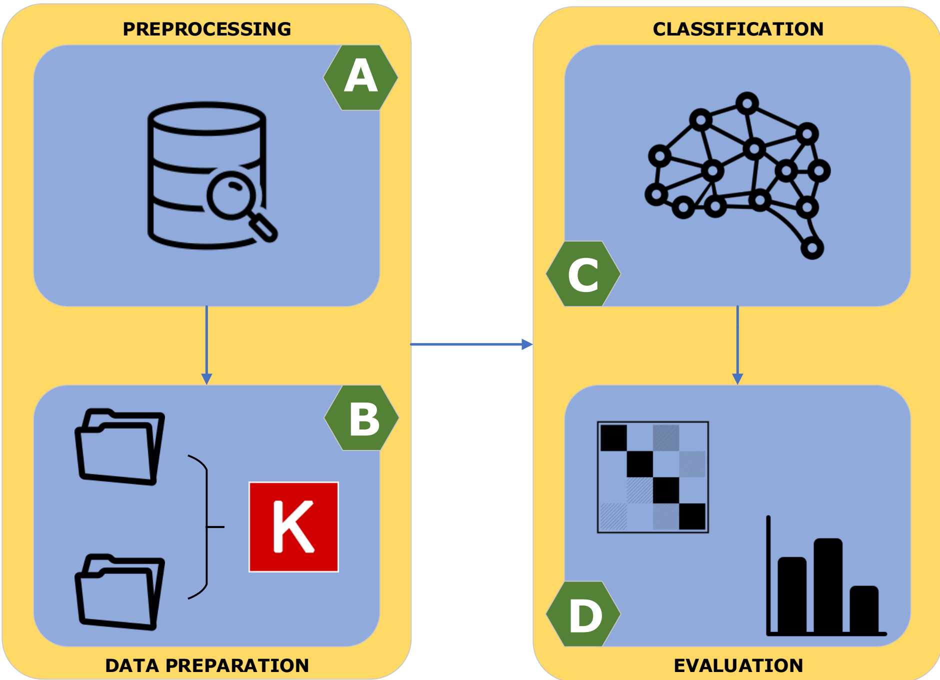

Figure 1: High level architecture

A. Preprocessing

In the preprocessing phase, the emphasis is on refining the dataset. The process involves systematically addressing corrupted files, detecting and managing duplicates through image hashing, and conducting a thorough dataset check.

B. Data Preparation

In the data preparation phase, the primary focus is on loading and enhancing the dataset. This involves using Keras and TensorFlow to load training, validation, and test datasets, applying data augmentation techniques such as flip, rotation, and zoom, and normalizing pixel values. The goal is to ensure the dataset is well-prepared and suitable for subsequent steps.

C. Classification

In the classification phase, an image classification procedure is established using Keras and TensorFlow. The workflow seamlessly integrates hyperparameter tuning and k-fold cross-validation for comprehensive model optimization.

D. Evaluation

In the evaluation phase, the performance of the models are assessed through the presentation of insightful metrics, such as loss and accuracy. The module further generates detailed classification reports and plots.

Chapter 2

Preprocessing and Data preparation

2.1 Dataset

As anticipated in the introduction, the dataset used for this project is the set of images called *"Muffin vs Chihuahua"*, retrieved from the Kaggle web platform. Comprising a total of 5917 JPG format images, the dataset is organized into two distinct folders, namely *"train"* and *"test"*, with the latter containing precisely 20% of the entire set. To get brief insight on the data, a concise depiction of the dataset's content is presented in **Figure 3** within the **Appendix A**.

Inside the train folder are stored 2559 images portraying chihuahuas and an additional 2174 images depicting muffins. The test folder mirrors this arrangement with 640 images dedicated to chihuahuas and 544 images of muffins.

2.2 Preprocessing

Ensuring the integrity and quality of training datasets is crucial for achieving good model performance.

The project utilizes several Python libraries, including *os* for file and directory operations, *ImageHash* for image hashing to detect duplicates, and *PIL* (Python Image Library) for image processing. Additionally, some custom utility functions are defined from separate modules for counting files, defining dataframes and plots.

The preprocessing phase begins with a corruption filter that identifies and handles corrupted image files within the dataset. This function traverses the dataset's subfolders and verifies the integrity of each image. Corrupted files are then presented to the user for deletion. In the context of this study, no instances of corrupt files were identified.

Subsequently, the project addresses the issue of duplicate images. Despite the information present on the web platform, several duplicates (i.e., same images, but of different sizes) were found during a manual inspection of the dataset, leading to the creation of a dedicated function to deal with this issue. In fact, identical instances may artificially inflate the importance of patterns, impacting the model's ability to generalize. To this purpose, traditional pixel-wise comparisons becomes impractical, so it was adopted a technique known as **average image hashing** from the ImageHash library. This algorithm is chosen for its simplicity and effectiveness in identifying identical images and to do so it exploits the following steps:

1. **Simplify the Image**: the algorithm simplifies the image by transforming its size and colors.

2. **Compute Average Pixel Value**: the average pixel value, representing the percent gray of the simplified image, is computed. This step contributes to the algorithm's perceptual nature, capturing the overall visual characteristics of the image.

3. **Generate the Hash**: the hash is generated by comparing each pixel's value to the previously calculated average. This comparison process results in a hash that encapsulates the essential features of the image in a way that is insensitive to minor variations and localized changes.

After implementing these two procedures, the size of the training set experienced a slight reduction. Specifically, the number of images of chihuahuas decreased from 2559 to 2377, and there was a drop in the number of images of muffins too, from the original 2174 to 2143. To enhance the analysis, **Figure 4** provides a visual summary of the class distribution within the dataset.

It's worth noting that while there is a slight imbalance in the dataset, it is not significant enough to negatively impact the performance of the models. Handling imbalances becomes more critical when dealing with severe disparities between class frequencies. In this case, the dataset's composition is such that the impact on model training and evaluation is expected to be minimal, and the provided preprocessing functions contribute to maintaining a high-quality dataset.

2.3 Data Preparation

The goal here is to create a complete solution that includes data loading, data augmentation, normalization, and conversion of the dataset into Numpy arrays suitable for model training.

The process begins with the loading of image datasets into Keras datasets using the **image_dataset_from_directory**[1] function, a powerful utility provided by TensorFlow and Keras to simplify the process of loading image datasets from directories. The data is organized into training, validation, and test sets, with specific configurations such as color mode, batch size, image size, and shuffle parameters tailored to the project's requirements.

- **Batch Size:** the dataset is divided into batches, each containing a specified number of samples. Choosing a batch size of 32 is a good way to find the right balance between computational efficiency and model stability for neural networks.
- **Color mode:** this argument is set to "rgb", indicating that the images are represented in the standard Red-Green-Blue color format. This choice is essential for capturing the full spectrum of colors present in the images, providing the model with rich information for classification.
- **Image Size:** the dimensions of the input images are crucial for the model's ability to discern features. In our code, the image size is set to 192x192, providing a compromise between computational efficiency and preserving relevant details for accurate classification.
- **Shuffle:** randomly shuffles the data (*if shuffle=True*), preventing the model from learning patterns based on the order of the input samples. A crucial step for improving model generalization. This operation is deliberately avoided for the test set. In fact, by maintaining its original order, one ensures an impartial evaluation of the model's capacity to generalize to new and unseen data.
- **Seed:** the seed parameter allows setting a seed for random operations, maintaining consistency across multiple runs.
- **Validation Split and Subset:** these two parameters work in tandem to automatically split the training dataset into training and validation subsets. In our code, validation_split is set to 0.2, indicating that 20% of the training data will be reserved for validation. The subset parameter is set to "both", ensuring that training and validation datasets are returned by this function. The validation set serves as a benchmark during the training process.

[1] `https://keras.io/api/data_loading/image/#imagedatasetfromdirectory-function`

The successful execution of this process results the following outcome:

```
> Training and Validation:
    Found 4520 files belonging to 2 classes.
    Using 3616 files for training.
    Using 904 files for validation.

> Test:
    Found 1184 files belonging to 2 classes.

> Class Names:
   - Class 0 = chihuahua
   - Class 1 = muffin
```

Then, to ensure consistency and speed up training, the data is standardized by rescaling pixel values to the range [0, 1].

Data augmentation is an important step in enhancing model generalization, preventing overfitting and improving the model's ability to recognize different patterns within the data. Note that, it is selectively applied only to the training set to avoid distorting the validation and test dataset. The implemented operation include random horizontal flips, rotations, and zooming (an example of this process's outcome is presented in **Appendix A**: **Figure 5**).

Subsequently, the Keras datasets undergo a transformation into split arrays, distinguishing between input values (X) and target values (Y). This step is really important because it takes advantage of how arrays work in libraries like NumPy, widely employed with frameworks like Keras and TensorFlow.

The conversion into arrays offers several advantages. Firstly, it simplifies the management and manipulation of data, providing a structured and standardized format. Secondly, the array representation facilitates seamless integration with neural network layers, streamlining the data flow through the model. Furthermore, the use of arrays enables efficient vectorized operations, enhancing computational performance during training.

Chapter 3

Classification and Evaluation

3.1 Classification

The basic workflow adopted during this phase starts with defining the models, namely **Multilayer Perceptron** (MLP), **Convolutional Neural Network** (CNN) and **MobileNet**. Following this, it is implemented the tuning process of the hyperparameters to improve performances. The achieved best configurations are then saved, and finally used in the application of k-fold cross-validation technique.

3.1.1 Models Overview

Before delving into the specifics of the adopted implementation, let's first briefly present the characteristics of the chosen models.

- **Multilayer Perceptron:** The MLP [2] is a fundamental neural network architecture characterized by an input layer, multiple hidden layers, and an output layer. Neurons in each layer are densely connected to those in adjacent layers, allowing for the learning of complex relationships within the data. The model is versatile and applicable to various machine learning tasks.

- **Convolutional Neural Network:** The CNNs [3] are designed for effective feature extraction from structured grid-like data, making them particularly adept at processing visual information such as images. Their architecture, comprising convolutional, pooling, and fully connected layers, enables hierarchical feature extraction from input data. Through successive convolutional layers, CNNs progressively discover increasingly abstract and discriminative features, capturing complex details essential for robust pattern recognition.

- **MobileNet:** MobileNet [4] is a lightweight convolutional neural network architecture specifically crafted for efficient and resource-friendly image processing tasks. Its key innovation lies in the use of depthwise separable convolutions, a strategy aimed at achieving high performance with minimal computational overhead. This approach significantly reduces the number of parameters, making MobileNet suitable for deployment on devices with limited computational resources.

3.1.2 Models Architecture

In the following sections, are outlined the adopted implementation of the models presented before. Regarding the general process of model building, particularly the tuning of hyperparameters through **Keras HyperParameters class**[1], it was decided to focus on a limited set of parameters (dense layer units, dropout rate and learning rate) to reduce the search and training times of the models.

Multilayer Perceptron

The MLP model is defined using a sequential architecture, presented in **Appendix A**: **Figure 6**, consists of five fully connected dense layers, each utilizing the rectified linear unit (ReLU) activation function.

The input layer takes the form of a flatten layer, its role is to transform the two-dimensional array representing the image into a one-dimensional vector, optimizing the network's ability to handle the data.

The first four dense layers have 256, 128, 64, and 32 units respectively. The fifth dense layer's number of units is a hyperparameter, which can be tuned between 32 and 512 with a step size of 32. Furthermore, to counteract overfitting, a dropout layer is added with a tunable dropout rate between 0.2 and 0.5 with a step size of 0.1.

Then, the output layer features a sigmoid activation function, specifically adapted for binary classification tasks.

Additionally, the learning rate is also tuned for the Adam optimizer, with choices of 1e-2, 1e-3, and 1e-4.

Convolutional Neural Network

The employed CNN architecture, presented in **Appendix A**: **Figure 7**, starts with a **convolutional 2D layer**[2] featuring 32 filter size and a 3x3 kernel size, followed by batch normalization, max pooling, and dropout layer incorporated to improve the model's robustness and prevent overfitting. This pattern is replicated with five additional convolutional layers sized at 64 and 128.

[1] `https://keras.io/api/keras_tuner/hyperparameters/`

[2] `https://keras.io/api/layers/convolution_layers/convolution2d/`

The tuning process is applied to the "fully-connected-structure" of the CNN, specifically after the flatten layer. At this stage, a dense layer is defined with tunable units ranging from 32 to 512, coupled with a dropout rate spanning from 0.2 to 0.5. Additionally, the learning rate of the Adam optimizer, is tuned presenting choices of 1e-2, 1e-3, and 1e-4.

MobileNet

The last model employed is the **MobileNet**[3], that adopt the structure illustrated in **Appendix A**: **Figure 8**.

The architecture is defined by loading the Keras model, with a custom classification head. The resulting Sequential model is composed of the "base model", which is initialized with weights pre-trained on the **ImageNet**[4] dataset. The latter refers to a vast set with millions of labeled images across diverse categories, that facilitates pre-training, enabling the model to grasp generic features and representations beneficial for a wide array of computer vision tasks.

The Sequential model integrates a flatten layer, a dense layer, a batch normalization layer, and a dropout layer. The dense layer's unit count is fine-tuned within the 32 to 512 range, while the dropout rate is varied between 0.2 and 0.5. Additionally, the model incorporates a final dense layer activated by sigmoid for binary classification. The Adam optimizer's learning rate is tuned selecting from the available choices of 1e-2, 1e-3, and 1e-4, contributing to optimization of the overall model architecture.

3.1.3 Tuning the Hyperparameters

The implementation of hyperparameter tuning employs the **KerasTuner API**[5], specifically utilizing the **Hyperband algorithm**[6]. The objective is to optimize the performance of a given neural network model through an efficient search of hyperparameter configurations.

The Hyperband algorithm, introduced in 2018 [5], represents an innovative and efficient approach to hyperparameter optimization, trying to speed up the Random-Search algorithm.

Hyperband operates through a series of brackets, each containing multiple configurations that are trained for a specified number of epochs. Configurations within a bracket are randomly sampled from a predefined search space. Resource allocation within Hyperband is dynamic, assigning fewer epochs to configurations with the aim of faster identification of promising candidates.

[3] `https://keras.io/api/applications/mobilenet/`
[4] `https://www.image-net.org/about.php`
[5] `https://keras.io/api/keras_tuner/`
[6] `https://keras.io/api/keras_tuner/tuners/hyperband/`

A key feature of Hyperband is its implementation of the **Successive Halving** (SH) procedure within each bracket. Configurations are grouped into successive subsets, and a fixed budget of resources is allocated to each subset. The best-performing configurations survive and move to the next round, while others are discarded.

The hyperparameter search is conducted using the **tuner.search**[7] method, exploring the hyperparameter space over a predetermined number of epochs (10 in this instance). The search process involves early termination based on the **EarlyStopping**[8] callback to enhance efficiency. This operation monitors the validation loss, interrupting training if no improvement is observed within a specified patience period, set to 3 epochs in this implementation. The collected data regarding this process are shown in **Appendix B**, within various tables labeled by model (MLP: Table 1 and Table 2; CNN: Table 3 and Table 4; MobileNet: Table 5 and Table 6).

After this step, the model is constructed and subjected to training for 10 epochs using the identified optimal hyperparameters. To gain a comprehensive insight into the training procedure and a preliminary understanding of the models capabilities, it is provided a representation of the training history. The **Figures 9** to **11** depict plots illustrating both training and validation accuracy, along with the corresponding loss values. The resultant tuned model is then serialized to JSON format and stored, along with its weights, in a designated directory for future use.

3.1.4 K-Fold Cross-Validation

During the previous process, the optimal hyperparameters and weights, are saved and used to build a model adopted as input for the **k-fold cross-validation** procedure (illustrated in **Figure 2**), used to evaluate the models performance.

This technique serves as a methodology to mitigate the risk of overfitting and ensure that the model's performance is not overly dependent on a specific subset of the data. It is extremely useful, especially in scenarios where the dataset size is limited. By partitioning the dataset into K equally sized folds, the technique enables the model to be trained K times, each time utilizing a different fold as test set and the remaining data for training. By averaging the performance metrics across multiple folds, the evaluation becomes less sensitive to overfitting on a specific split and provides a more comprehensive assessment of its generalization performance.

The adopted implementation begins by initializing and loading the model obtained from the previous tuning process. A key aspect stands in the definition of the **zero-one classification loss** function. Specifically designed to quantify misclassification,

[7] `https://www.tensorflow.org/tutorials/keras/keras_tuner?hl=en`

[8] `https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping`

returns a value of 1 when the true labels (y) do not align with the predicted labels ($\hat{y}$), and 0 otherwise, as formally defined below.

$$l(y, \hat{y}) = \begin{cases} 0 & \text{if } y = \hat{y} \\ 1 & \text{if } y \neq \hat{y} \end{cases} \tag{1}$$

The procedure is configured with a specific number of folds, set to 5 in the current analysis. This parameter governs the number of iterations, and for each iteration, accuracy, loss, and zero-one loss metrics are collected, providing a comprehensive evaluation of performance. An accurate look at these results can be taken by looking at the training history plot (**Appendix A**: Figure 12 to Figure 26) and data tables (**Appendix B**: Table 7, Table 8 and Table 9) generated during the process. The discussion of the performance comparisons is addressed in the subsequent chapter.

Figure 2: k-fold cross-validation scheme

3.2 Evaluation

The model evaluation process is a crucial step in assessing the performance and reliability of learning models. The implemented evaluation framework in this project uses diverse methods and metrics to comprehensively analyze the models.

- **Calculation of Evaluation Metrics:** Accuracy and loss provide a concise overview of each model's performance.

- **Classification Report:** Through the scikit-learn library is called the classification report function. This report provides precision, recall, and f1-score for each class, giving insight into how models perform across different categories.
- **Generation of Plots:** Three types of plots are defined to visually represent the classification capabilities.
  - **Confusion Matrix:** Illustrates the distribution of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) predictions.
  - **Prediction Evaluation Graph:** Compares true classes with predicted ones through a bar graph.
  - **Display of Prediction:** Define a grid-like structure showing pictures and the corresponding label, presenting an intuitive understanding of the models capabilities.

The first step involves the computation of the accuracy and loss metrics, defining an overview of the models performances. The outcomes reveal distinctive performances for each model as displayed in the **Appendix B:** Table 13.

For the **MLP** model , a loss of 0.752 indicates a notable disparity between predicted and actual values, highlighting a significant rate of misclassification. Comparatively, the **CNN** model displays improvement with a lower loss of around 0.29, signifying a reduced gap between predicted and true values, and higher accuracy value about 91% showing an improved correctness. Finally, **MobileNet** model stands out with exceptional performance results, featuring a remarkably low loss of 0.022 and an outstanding accuracy of around 99% showcases high correctness in predictions.

These results are more evident by observing the classification reports. Encapsulating key metrics such as **precision** (P), **recall** (R), and the **f1-score**, which provide additional insight for evaluating the effectiveness of models.

- **Precison**: representing the positive predictive value, should ideally be 1 (high) for a good classifier. Precision becomes 1 only when the numerator and denominator are equal , i.e., TP = TP + FP, this also means FP is zero. As FP increases the value of denominator becomes greater than the numerator and precision value decreases, representing an undesirable outcome.

$$P = \frac{TP}{TP + FP}$$

- **Recall**: also known as sensitivity is a measure of how many of the positive cases the classifier correctly predicted, and should ideally be 1 (high) for a good classifier. Recall becomes 1 only when the numerator and denominator are equal, i.e., TP = TP + FN, this also means FN is zero. As FN increases the value of denominator becomes greater than the numerator and recall value decreases, which is better to be avoided.

$$R = \frac{TP}{TP + FN}$$

- **F1-score**: is a metric which takes into account both precision and recall, that becomes high only when both are high. It represents the harmonic mean of the two values and is a better measure than accuracy, especially in the case of unbalanced class distribution.

$$F1 = 2 \times \frac{P \times R}{P + R}$$

Analyzing the classification reports, the MLP model exhibits trade-offs between precision and recall, with a large gap between the two classes (e.g., a high recall value is obtained for the chihuhua class, while a low value is obtained for the muffin class), resulting in a moderate f1-score. The CNN model showcases improvements, achieving better results and more balance between precision and recall, leading to higher f1-scores. Notably, the MobileNet model outclasses its counterparts, excelling in precision, recall, and F1-score, indicative of its superior classification capabilities.

The final evaluation phase, which involves the previously mentioned plots, provides further confirmation of the capabilities of the models under analysis. As observed in the results, presented in **Figures 27** to **32**, it becomes evident that the MLP model demonstrates suboptimal performance, characterized by outcomes that appear more random than methodical. Conversely, the CNN and MobileNet models exhibit, respectively, better and optimal results. The CNN model introduces a marginal degree of error (as highlighted before by the loss), while MobileNet excels, achieving a classification proficiency near perfection.

A clear differentiation between the the three models becomes manifest observing the **Figures** spanning from **33** to **35**. Depicting a “prediction view”, the plot is represented as a grid of nine images from the test dataset, indicating the true class to which they belong and the class predicted by the model. It is clear that the risk of misclassification becomes less and less relevant moving from the MLP model, where in the example shown, four out of nine images are misclassified, to the MobileNet model, which performs the process without making mistakes.

Chapter 4

Conclusion

4.1 Conclusion

In conclusion, the project successfully addresses the challenge of binary image classification, specifically distinguishing between images of chihuahuas and muffins. The implemented models, Multilayer Perceptron (MLP), Convolutional Neural Network (CNN), and MobileNet, undergo thorough preprocessing, data preparation, hyperparameter tuning, K-fold cross-validation, and evaluation processes.

The models exhibit varying degrees of performance, with MobileNet emerging as the standout performer, achieving near-perfect accuracy and classification proficiency. The CNN model also demonstrates notable results. The MLP model performs worse than its counterparts, exhibiting suboptimal performance characterized by higher loss resulting in a notable rate of misclassification, thus proving to be ineffective in this particular case.

Appendix A

Project Plots and Results

Figure 3: Brief Overview of Labeled Images Within the Dataset

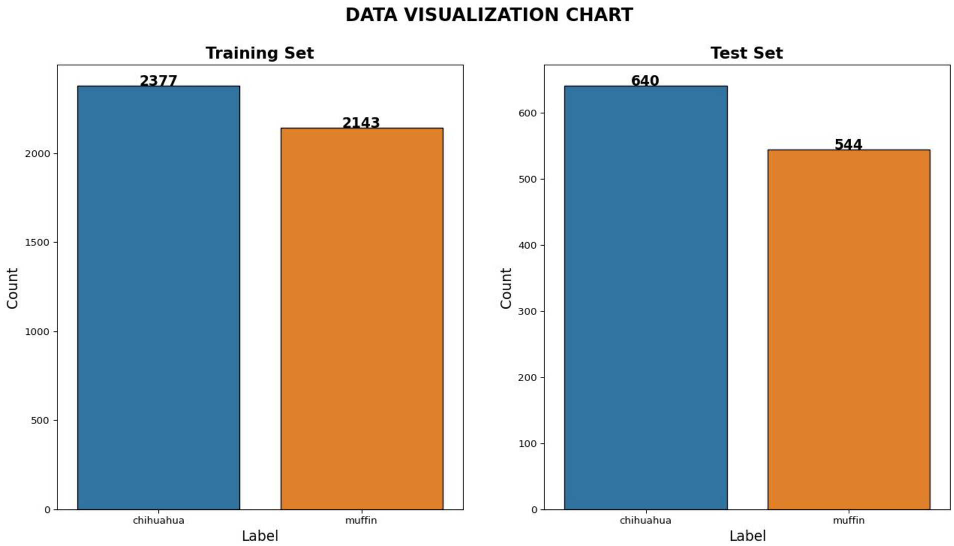

Figure 4: Dataset Class Distribution

Figure 5: Example of Data Augmentation

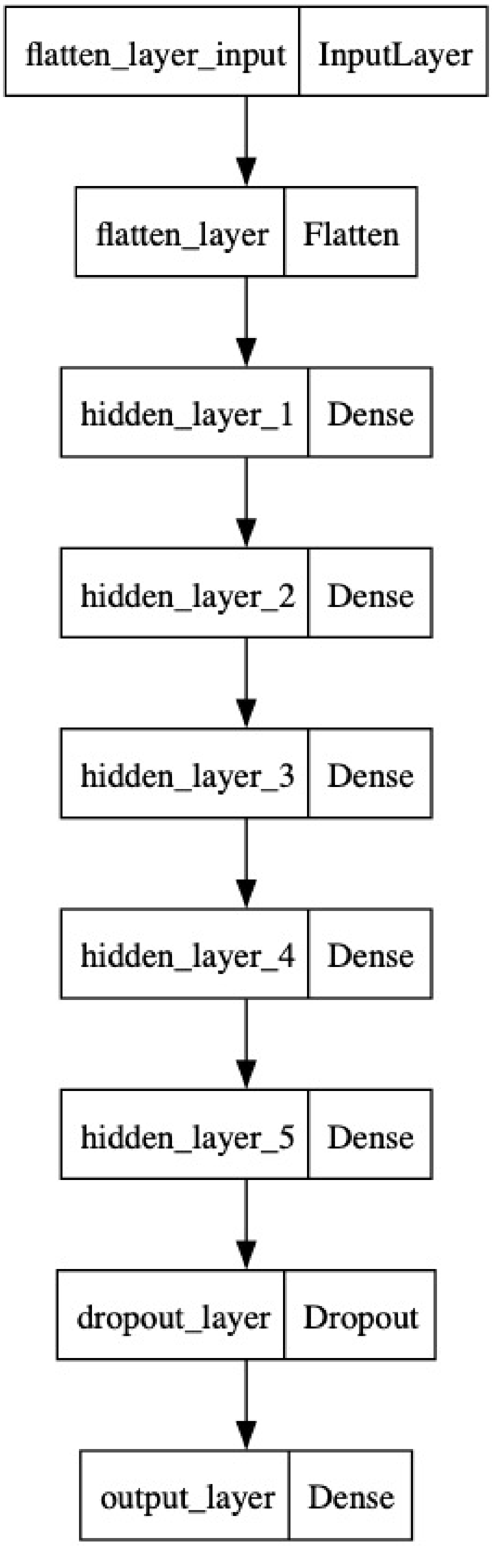

Figure 6: MLP Model Summary

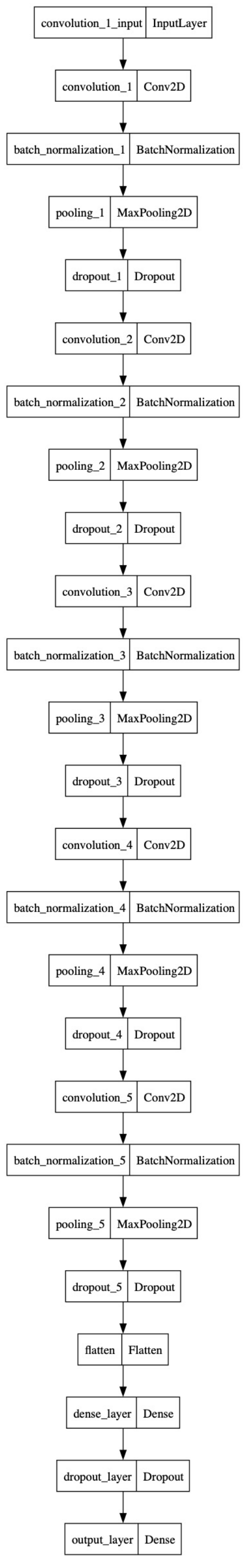

Figure 7: CNN Model Summary

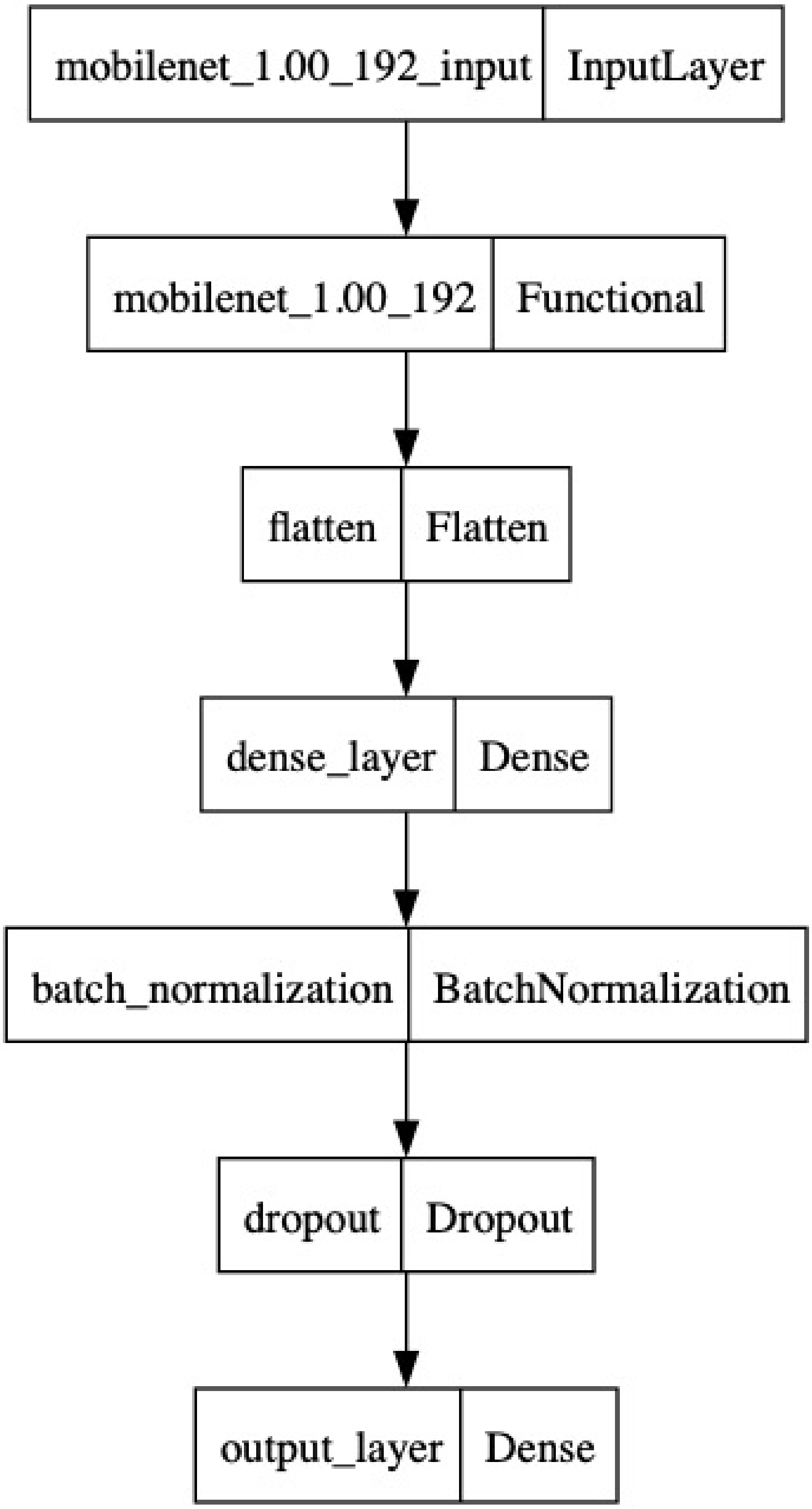

Figure 8: MobileNet Model Summary

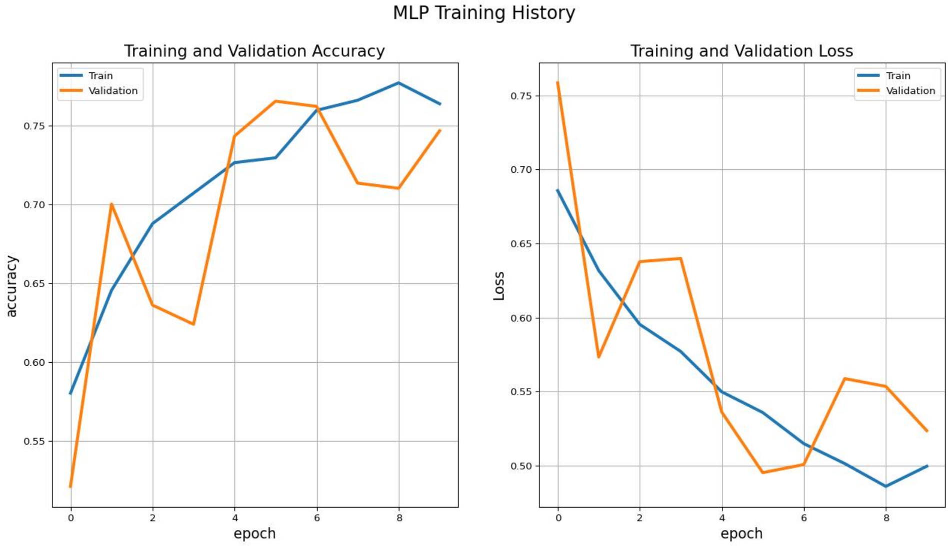

Figure 9: MLP Training History

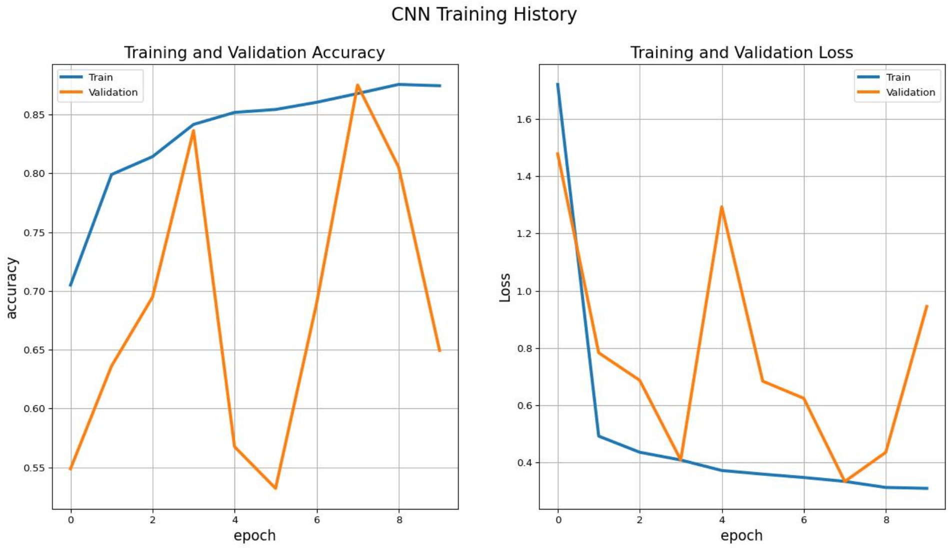

Figure 10: CNN Training History

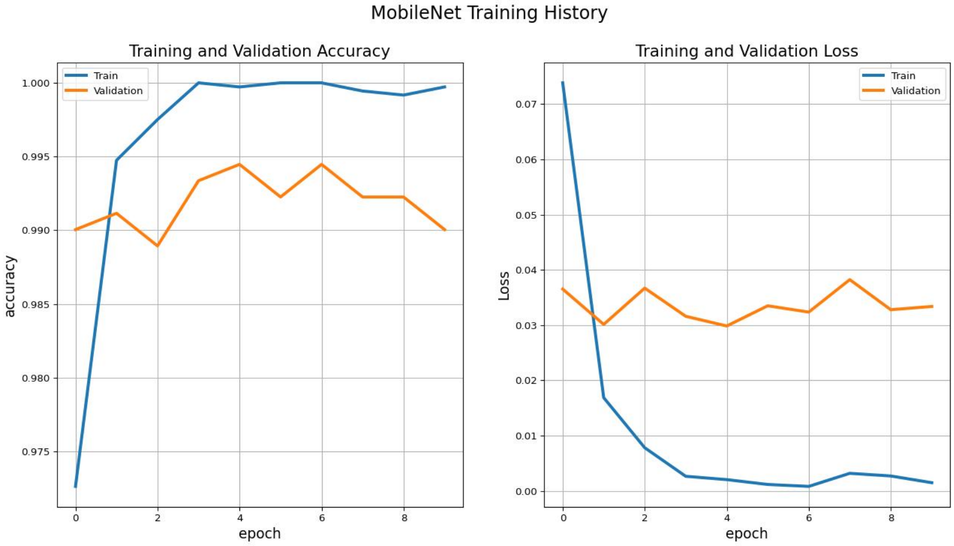

Figure 11: MobileNet Training History

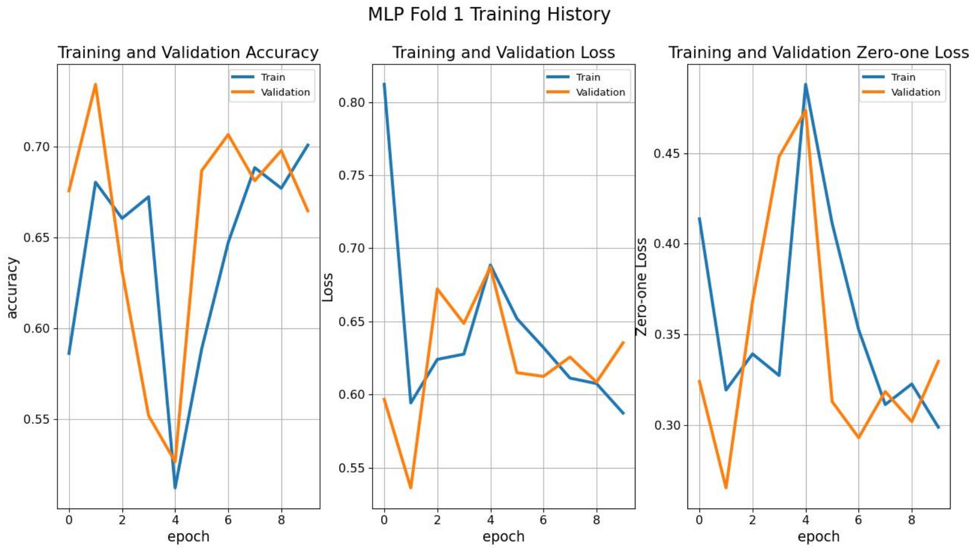

Figure 12: MLP Fold 1

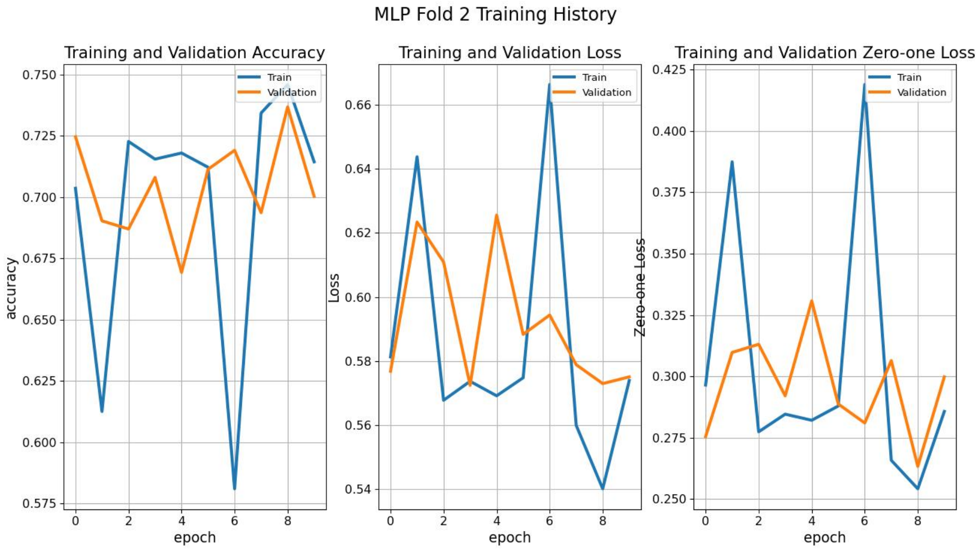

Figure 13: MLP Fold 2

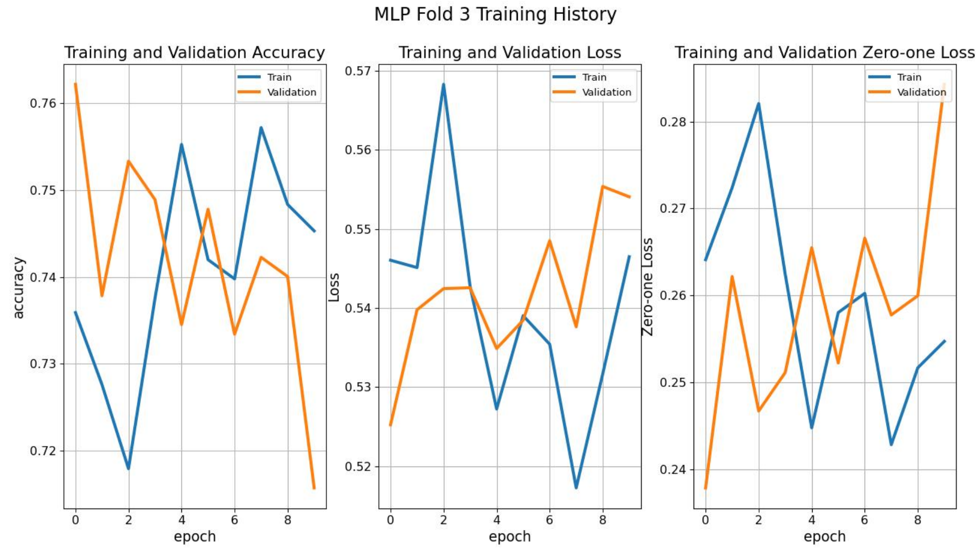

Figure 14: MLP Fold 3

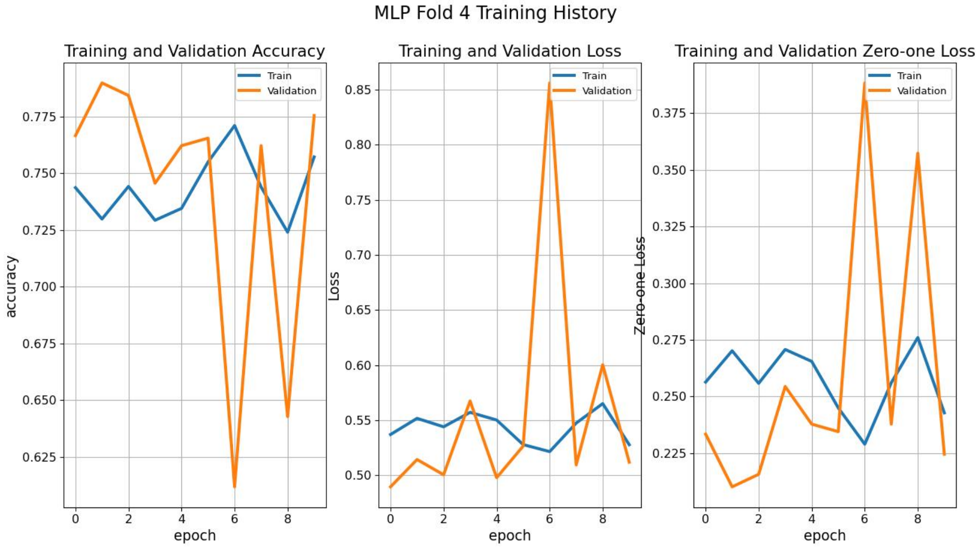

Figure 15: MLP Fold 4

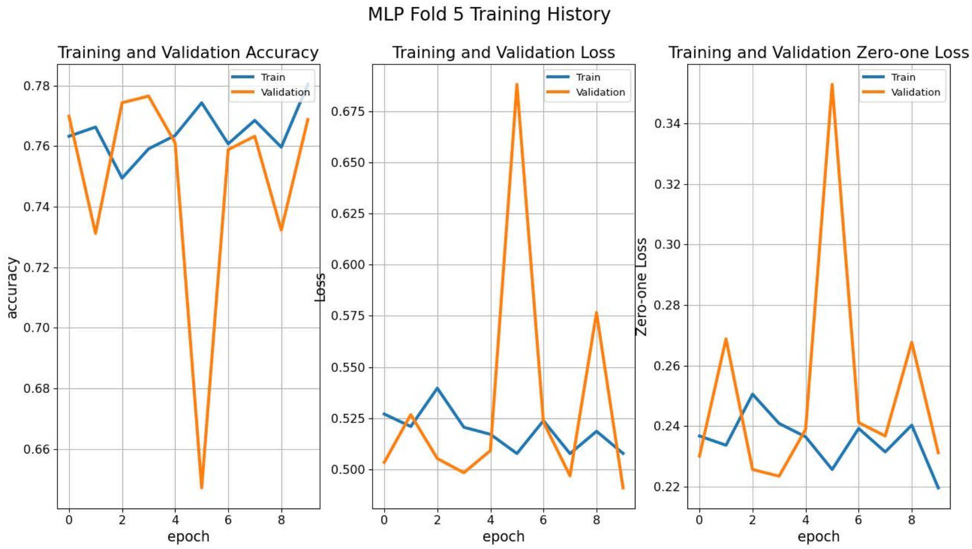

Figure 16: MLP Fold 5

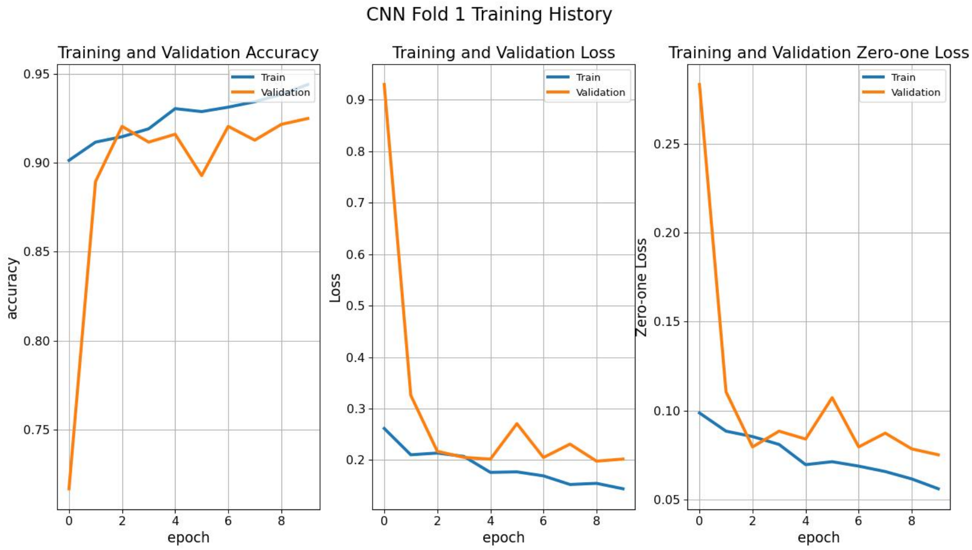

Figure 17: CNN Fold 1

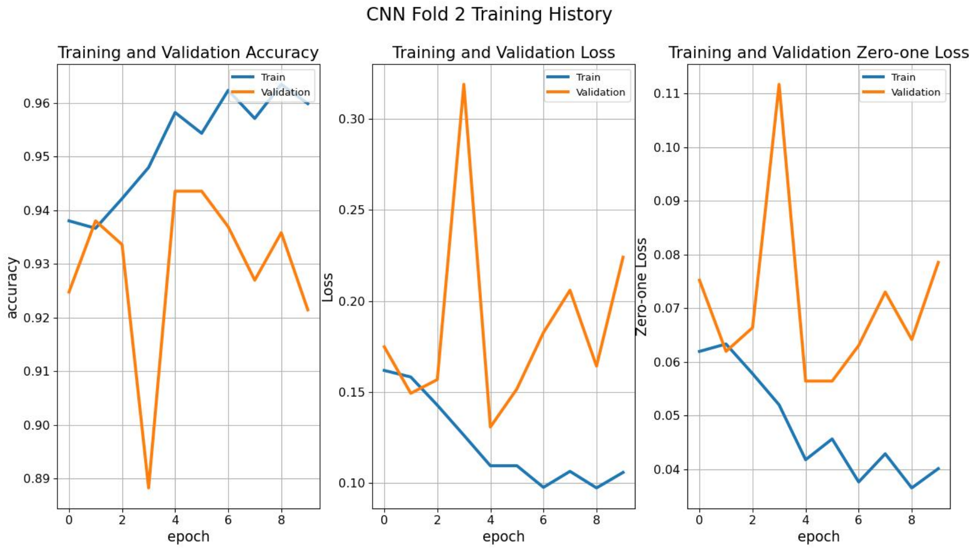

Figure 18: CNN Fold 2

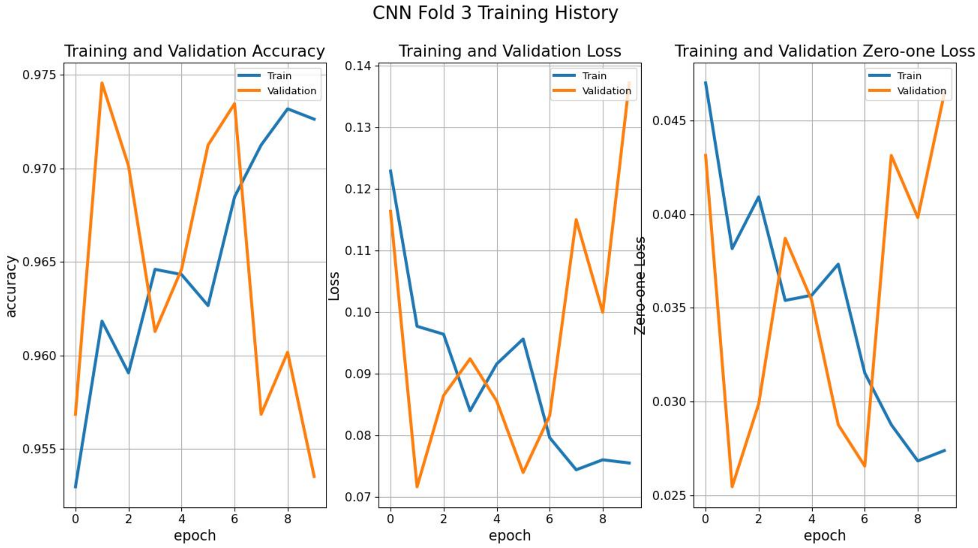

Figure 19: CNN Fold 3

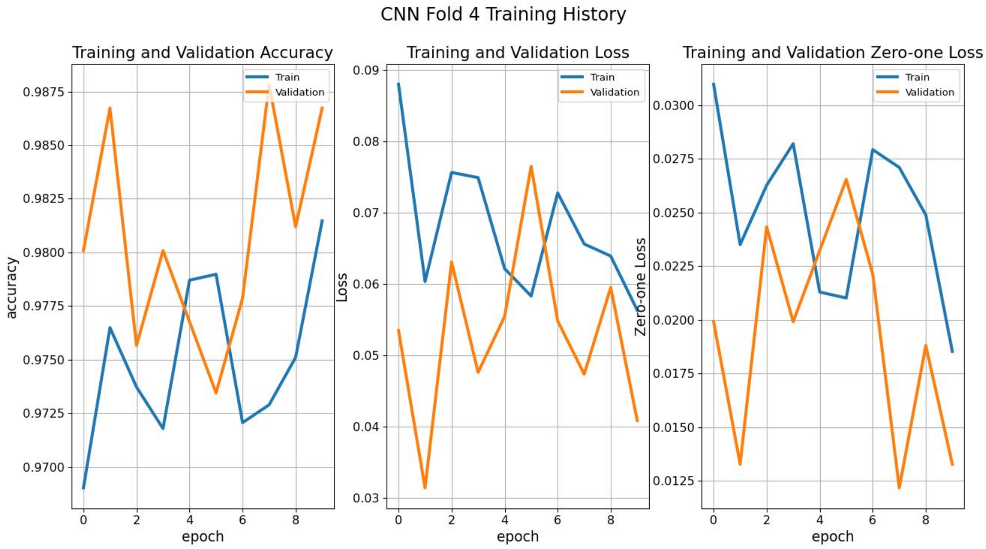

Figure 20: CNN Fold 4

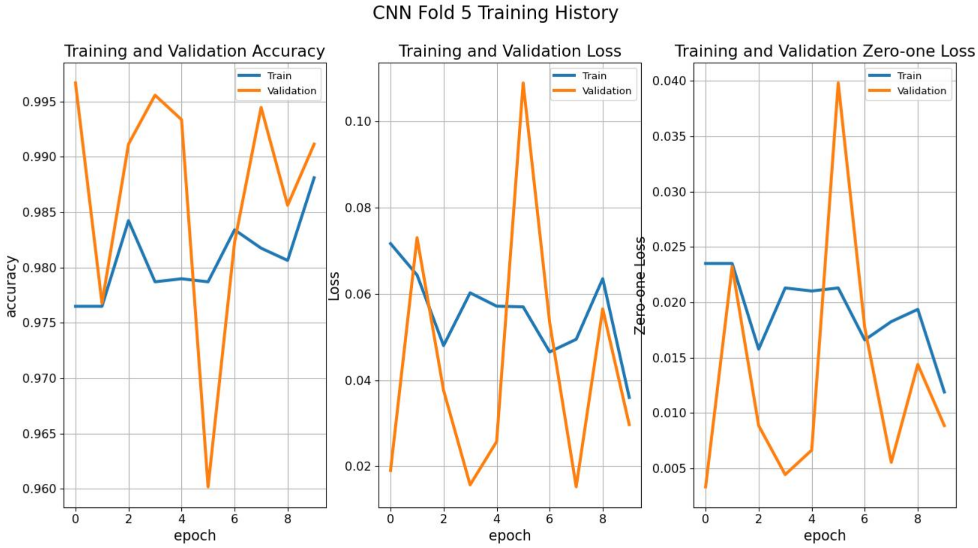

Figure 21: CNN Fold 5

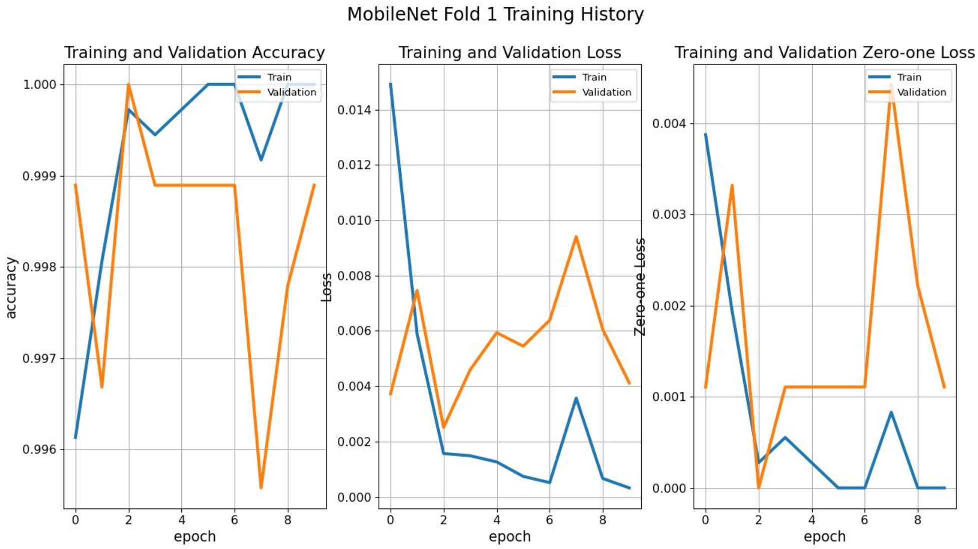

Figure 22: MobileNet Fold 1

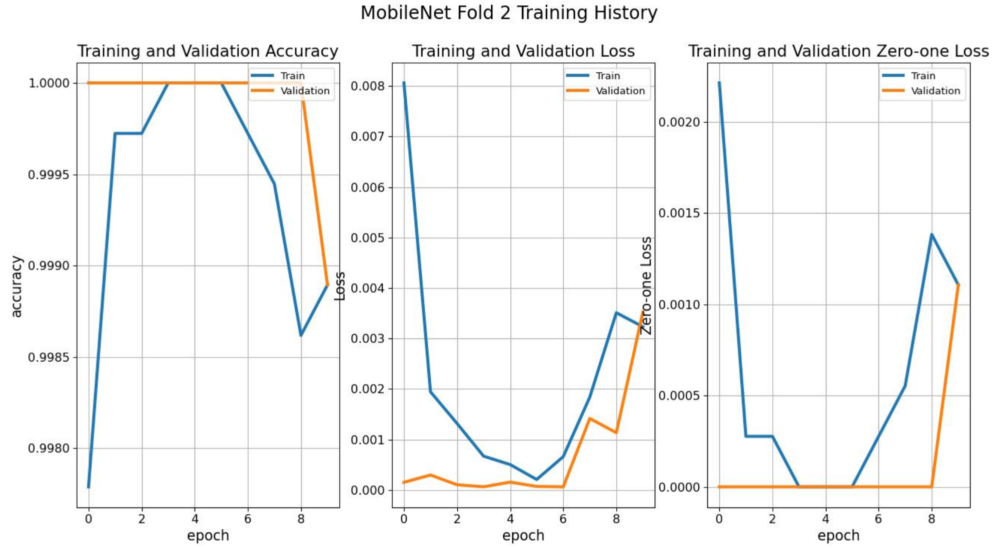

Figure 23: MobileNet Fold 2

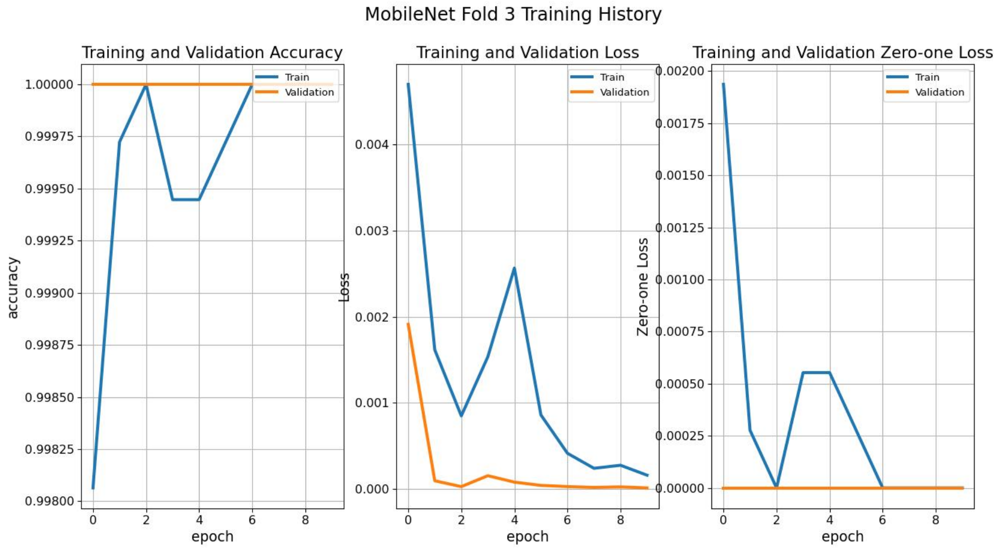

Figure 24: MobileNet Fold 3

Figure 25: MobileNet Fold 4

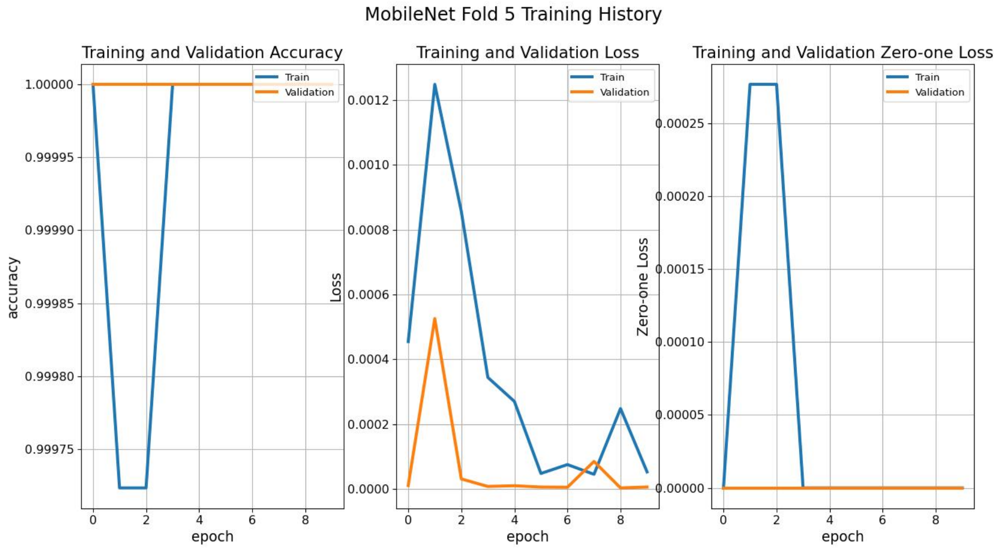

Figure 26: MobileNet Fold 5

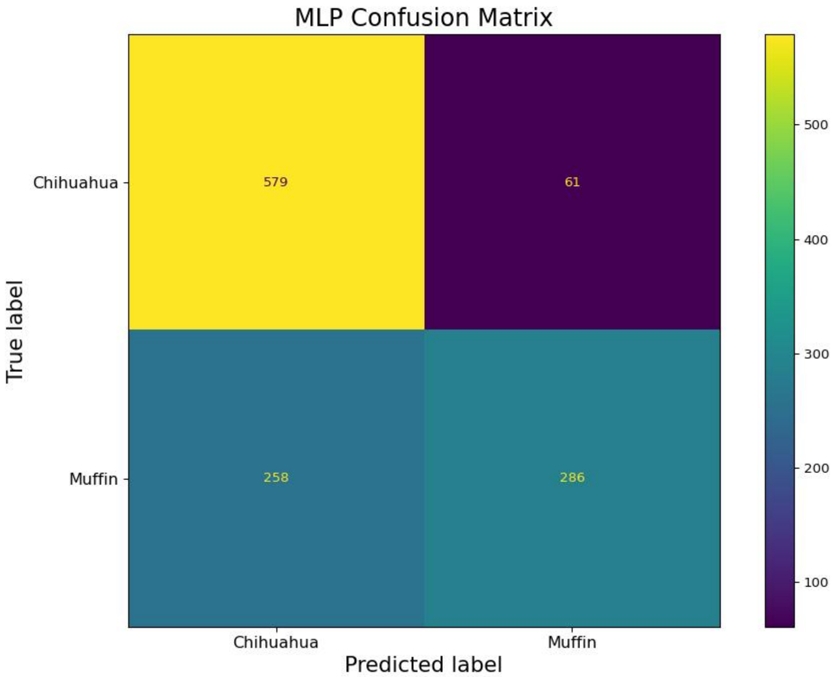

Figure 27: MLP Confusion matrix

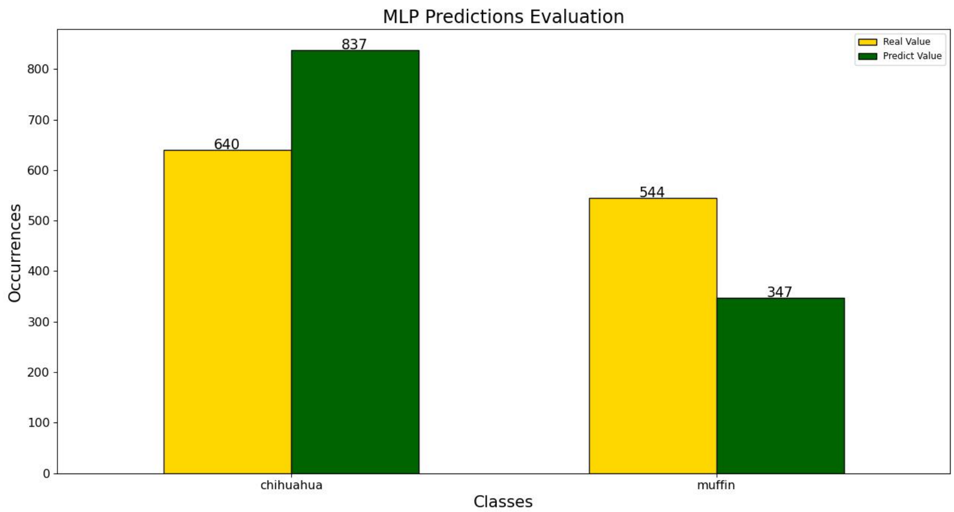

Figure 28: MLP Prediction Evaluation

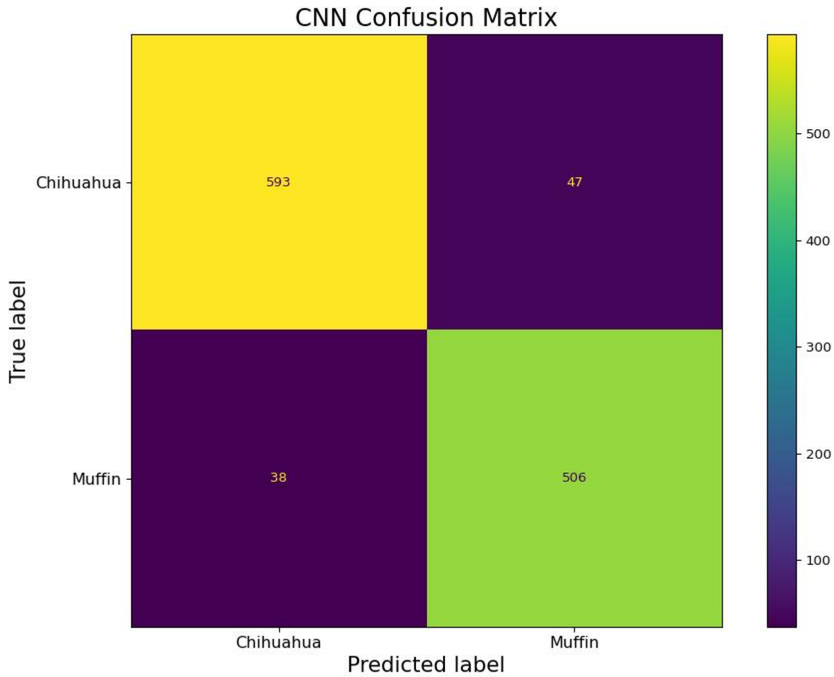

Figure 29: CNN Confusion matrix

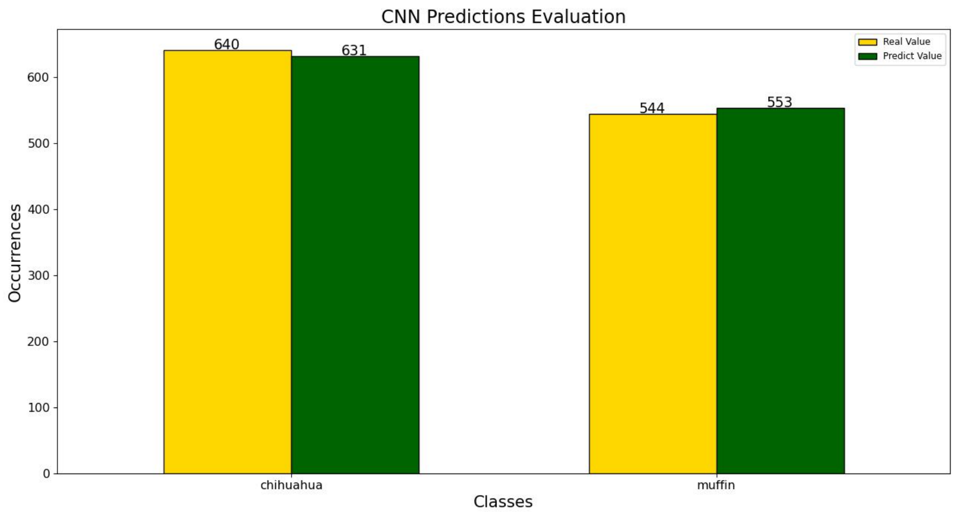

Figure 30: CNN Prediction Evaluation

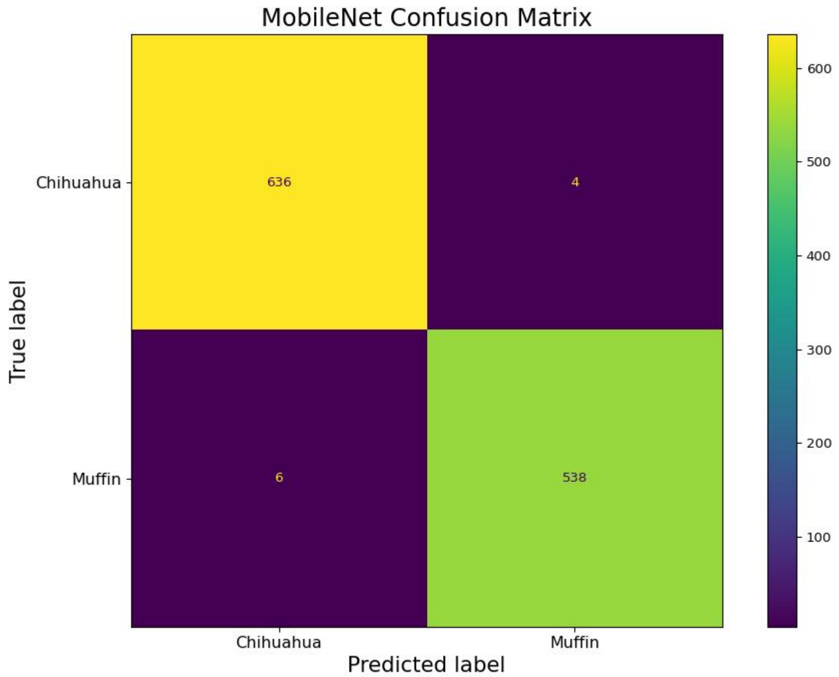

Figure 31: MobileNet Confusion matrix

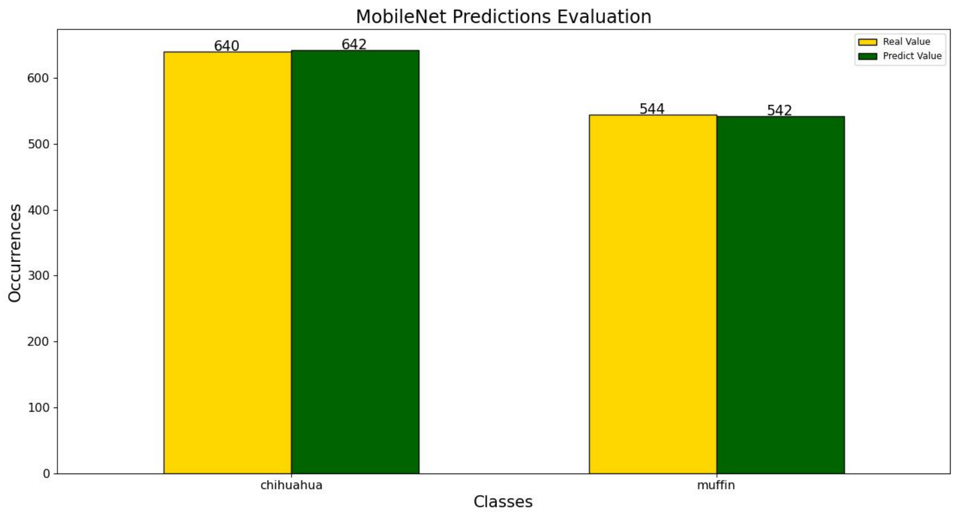

Figure 32: MobileNet Prediction Evaluation

Figure 33: MLP Prediction Visualization

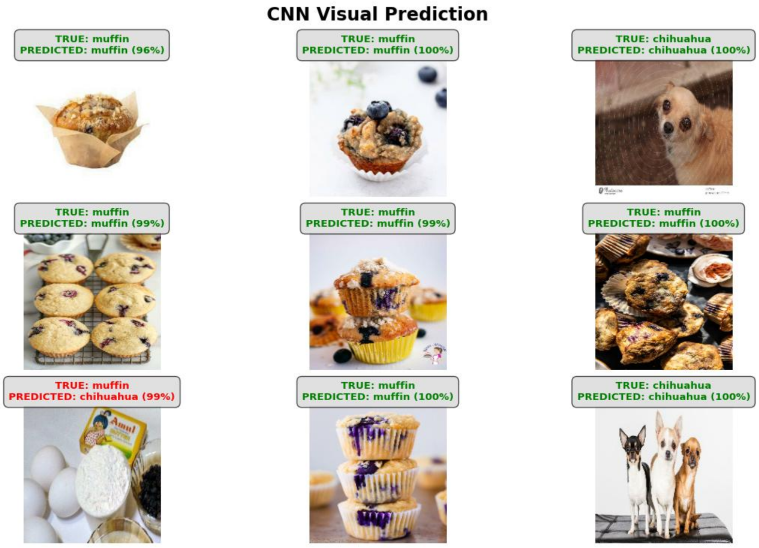

Figure 34: CNN Prediction Visualization

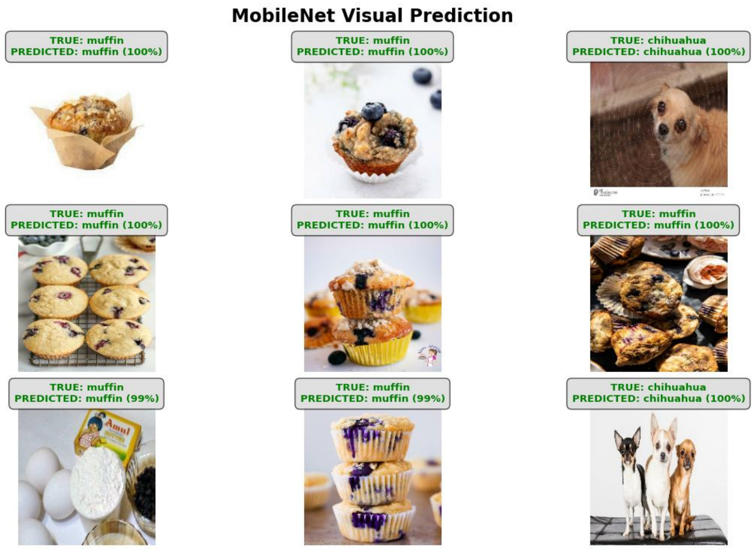

Figure 35: MobileNet Prediction Visualization

Appendix B

Tables and Data

Trial	**Units**	**Dropout Rate**	**Learning Rate**	**Validation Accuracy**
1	480	0.4000	0.0100	0.5100
2	480	0.3000	0.0100	0.5100
3	224	0.3000	0.0100	0.5100
4	224	0.3000	0.0010	0.5100
5	416	0.3000	0.0010	0.5100
6	96	0.3000	0.0010	0.6327
7	96	0.3000	0.0010	0.5973
8	480	0.4000	0.0100	0.5100
9	480	0.3000	0.0100	0.5100
10	96	0.3000	0.0010	0.5133
11	480	0.4000	0.0100	0.5100
12	96	0.4000	0.0010	0.5100
13	256	0.3000	0.0001	0.6914
14	64	0.4000	0.0100	0.5100
15	128	0.2000	0.0010	0.5100
16	256	0.3000	0.0001	0.7157
17	96	0.4000	0.0010	0.5100
18	96	0.2000	0.0100	0.5122
19	192	0.2000	0.0100	0.5100
20	352	0.2000	0.0001	0.7190
21	352	0.4000	0.0001	0.7080

Table 1: MLP hyperparameters tuning process

Model	**Units**	**Dropout Rate**	**Learning Rate**
MLP	352	0.2000	0.0001

Table 2: MLP optimal hyperparameters

Trial	**Units**	**Dropout Rate**	**Learning Rate**	**Validation Accuracy**
1	128	0.4000	0.0010	0.5100
2	384	0.3000	0.0100	0.5852
3	512	0.4000	0.0010	0.5100
4	32	0.3000	0.0100	0.6670
5	512	0.2000	0.0100	0.6184
6	192	0.4000	0.0010	0.5100
7	32	0.3000	0.0100	0.5100
8	512	0.2000	0.0100	0.5509
9	384	0.3000	0.0100	0.7633
10	384	0.3000	0.0100	0.7301
11	512	0.2000	0.0100	0.7434
12	512	0.4000	0.0100	0.5686
13	32	0.4000	0.0010	0.5100
14	288	0.4000	0.0001	0.5100
15	224	0.4000	0.0010	0.5100
16	512	0.4000	0.0100	0.6803
17	32	0.4000	0.0010	0.5100
18	320	0.4000	0.0100	0.8650
19	192	0.2000	0.0100	0.7500
20	320	0.4000	0.0010	0.8551
21	512	0.3000	0.0010	0.8252

Table 3: CNN hyperparameters tuning process

Model	**Units**	**Dropout Rate**	**Learning Rate**
CNN	320	0.4000	0.0100

Table 4: CNN optimal hyperparameters

Trial	Units	Dropout Rate	Learning Rate	Validation Accuracy
1	256	0.2000	0.0100	0.9900
2	288	0.3000	0.0100	0.9878
3	224	0.3000	0.0010	0.9912
4	416	0.2000	0.0001	0.9923
5	512	0.3000	0.0001	0.9912
6	32	0.2000	0.0100	0.9923
7	416	0.2000	0.0001	0.9912
8	32	0.2000	0.0100	0.9923
9	224	0.3000	0.0010	0.9923
10	32	0.2000	0.0100	0.9923
11	224	0.3000	0.0010	0.9923
12	480	0.3000	0.0001	0.9945
13	64	0.4000	0.0001	0.9923
14	160	0.4000	0.0010	0.9956
15	384	0.3000	0.0001	0.9912
16	160	0.4000	0.0010	0.9945
17	480	0.3000	0.0001	0.9923
18	160	0.2000	0.0100	0.9923
19	480	0.4000	0.0001	0.9934
20	320	0.4000	0.0100	0.9923
21	416	0.4000	0.0100	0.9923

Table 5: MobileNet hyperparameters tuning process

Model	Units	Dropout Rate	Learning Rate
MobileNet	160	0.4000	0.0010

Table 6: MobileNet optimal hyperparameters

Fold	Loss	Accuracy (%)	0-1 Loss
1	0.635	66.482	0.335
2	0.575	70.022	0.300
3	0.554	71.571	0.284
4	0.512	77.544	0.225
5	0.491	76.881	0.231
Average	0.553	72.500	0.275

Table 7: MLP Validation per Fold

Fold	Loss	Accuracy (%)	0-1 Loss
1	0.202	92.478	0.075
2	0.224	92.146	0.079
3	0.137	95.354	0.046
4	0.041	98.673	0.013
5	0.030	99.115	0.009
Average	0.127	95.553	0.044

Table 8: CNN Validation per Fold

Fold	Loss	Accuracy (%)	0-1 Loss
1	0.004	99.889	0.001
2	0.004	99.889	0.001
3	0.000	100.000	0.000
4	0.000	100.000	0.000
5	0.000	100.000	0.000
Average	0.002	99.956	0.000

Table 9: MobileNet Validation per Fold

index	precision	recall	f1-score	support
chihuahua	0.692	0.905	0.784	640.000
muffin	0.824	0.526	0.642	544.000
accuracy			0.731	1184.000
macro avg	0.758	0.715	0.713	1184.000
weighted avg	0.753	0.731	0.719	1184.000

Table 10: MLP Classification Report

index	precision	recall	f1-score	support
chihuahua	0.940	0.927	0.933	640.000
muffin	0.915	0.930	0.923	544.000
accuracy			0.928	1184.000
macro avg	0.927	0.928	0.928	1184.000
weighted avg	0.928	0.928	0.928	1184.000

Table 11: CNN Classification Report

index	precision	recall	f1-score	support
chihuahua	0.991	0.994	0.992	640.000
muffin	0.993	0.989	0.991	544.000
accuracy			0.992	1184.000
macro avg	0.992	0.991	0.991	1184.000
weighted avg	0.992	0.992	0.992	1184.000

Table 12: MobileNet Classification Report

Model	Loss	Accuracy (%)
MLP	0.752	73.057
CNN	0.292	92.821
MobileNet	0.022	99.155

Table 13: Test Accuracy and Loss per Model

Bibliography

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

[2] Wikipedia. Multilayer perceptron — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Multilayer_perceptron&oldid=1182254097.

[3] Wikipedia. Convolutional neural network — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Convolutional_neural_network&oldid=1191271173.

[4] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. https://arxiv.org/abs/1704.04861.

[5] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. https://arxiv.org/abs/1603.06560.

Online Resources

The sources for this project are available via Github.