UNIVERSITÀ DEGLI STUDI DI MILANO

Master's Degree in Computer Science

STATISTICAL METHODS FOR MACHINE LEARNING

CNN Binary Classification: Muffins and Chihuahuas

Students: Cannone Saverio – Rinaldi Alessandro

A.A. 2023-2024

Contents

Introduction

This project tackles a unique and interesting challenge: teaching a computer to tell the difference between pictures of muffins and chihuahuas. This task dives deep into the world of image recognition, a key area in artificial intelligence (AI), and to do so, we are going to be using two specific python libraries: Keras and Tensorflow. Keras makes building AI models less complicated, and TensorFlow gives us the power to handle lots of data and complex calculations. Additional libraries and tools that will be used are going to be discussed in detail over the next chapters.

Architectures

This project will experiment with various neural network architectures, each with its unique strengths and suitability for different aspects of the task at hand. The architectures to be explored are:

- **Multilayer Perceptrons (MLP):** Although traditionally more common for tabular data, exploring MLPs will provide a foundational understanding of neural network operations.
- **Convolutional Neural Networks (CNN):** Given their prowess in capturing spatial hierarchies in images, CNNs are expected to be particularly effective for this task.

- **EfficientNet:** Renowned for its ability to achieve high accuracy with significantly fewer parameters and computational expense.

Data Augmentation

Since the importance of a diverse dataset is crucial in training effective models, this project incorporates a data augmentation strategy to artificially expand our dataset. By applying transformations like rotation and flipping to our images, we aim to introduce a level of variation that mimics real-world scenarios, enhancing the model's ability to generalize from the training data to new, unseen images. This approach not only combats the risk of overfitting, where the model learns the training data too well and performs poorly on new, unseen data, but also enriches our dataset, providing a more comprehensive training experience for the neural network.

Training

The training process involves feeding the neural network with a large dataset of muffin and chihuahua images. Each image is labeled with the correct answer, allowing the model to learn by comparing its predictions against the true labels. Through a process of optimization, the model adjusts its parameters to minimize the difference between its predictions and the actual outcomes. We will employ Binary Cross-Entropy Loss as our primary loss function.

Tuning

The exploration of these architectures will be accompanied by rigorous hyperparameter tuning, especially employing two methods: random search and Bayesian optimization. These strategies are selected to strike a balance between exhaustive search and computational feasibility, with the aim of identifying optimal configurations that maximize the model's performance.

Risk Estimate

Moreover, the project will implement a 5-fold cross-validation approach as a risk estimate to ensure the robustness and generalizability of the findings. This method, by partitioning the data into five subsets and using each in turn for validation while training on the remainder, provides a comprehensive assessment of the model's performance across different data segments, thus mitigating the risk of overfitting and ensuring the reliability of the results.

Conclusion

Each decision made throughout the project—from the selection of neural network architectures to the choice of evaluation methods—is grounded in a strategic consideration of the trade-offs between computational efficiency, model complexity, and predictive accuracy. Every step will be thoroughly discussed in the following pages.

Chapter 1

Data Preprocessing

Data preprocessing plays a fundamental role in the success of neural network models, which are highly sensitive to the quality and structure of the input data. This section is going to showcase every technique used to improve the quality of the dataset and the models.

Figure 1.1: Image of a chihuahua from the training dataset

1.1 Dataset

The original dataset comprises 3199 photos of chihuahuas and 2718 photos of muffins, totaling 5917 images. To ensure a balanced representation of both classes, we opt to utilize around 80% of the dataset for training and reserve the remaining 20% for testing. Given the marginal difference in the number of images between the two classes, there is no immediate need to discard any images from either class.

To ensure the integrity and accuracy of the dataset, several preprocessing steps were undertaken. Firstly, a short script was developed to hash each image, allowing for the identification and removal of duplicate entries. This step helps prevent biases introduced by duplicate images, ensuring each unique image contributes equally to the training process. Additionally, a manual inspection was conducted to verify the labeling accuracy of each image. We actually found several misclassification errors, that would have led to a confusing training of the model, and erroneus validation.

To improve computational efficiency, it was decided to resize each image to dimensions of 150x150 pixels. This resizing strategy strikes a balance that minimizes computational overhead without significantly sacrificing image quality or model accuracy. This led to a dramatic improvement of the model's performance.

Figure 1.2: Image of a muffin from the training dataset

1.2 Data augmentation

The implementation of data augmentation in this project was carried out using methods from the `layers` class from TensorFlow's Keras API.

We will now discuss the steps we undertook:

Step 1: Configuration

Data augmentation was configured with a variety of transformations to simulate different possible variations in new images:

- **Rotation:** Images were randomly rotated within a range of $[-0.3\pi, 0.3\pi]$ degrees using `layers.RandomRotation`.
- **Contrast:** Images had their contrast randomly shifted by a factor of 0.2, using `layers.RandomContrast`
- **Zoom:** Random zooming was applied to images up to 10% to mimic object distance variations using `layers.RandomZoom`
- **Flip:** Images were randomly flipped horizontally to assume different orientations using `layers.RandomFlip`

Step 2: Load and Augment Images

This newly generated layer was then added to each model using `Sequential` with a vector as a parameter. An example:

```python
def get_model_data(model, filename="model", augmentation=True,
                                num_epochs=20, lr=0.001):
    augmented_model = Sequential([model])
    if augmentation:
        data_augmentation = Sequential([
            layers.RandomFlip("horizontal"),
            layers.RandomRotation(factor=(-0.15, 0.15)),
            layers.RandomContrast(0.2),
            layers.RandomZoom(-.1, 0.1),
        ], "Data_Augmentation")
        augmented_model = Sequential([data_augmentation, model]
                                                )
    return augmented_model
```

In the following training and validation accuracy/loss plots, our CNN implementation shows overfitting where there is no data augmentation: validation accuracy is 10% lower than training accuracy, and validation loss is 90% higher than training loss.

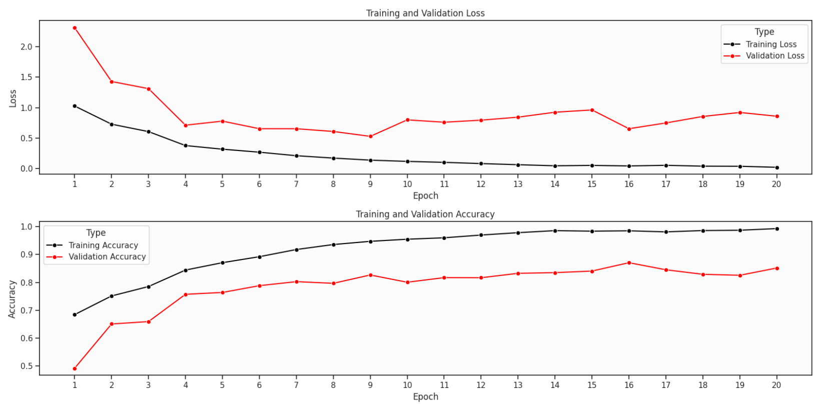

Figure 1.3: Non augmented CNN

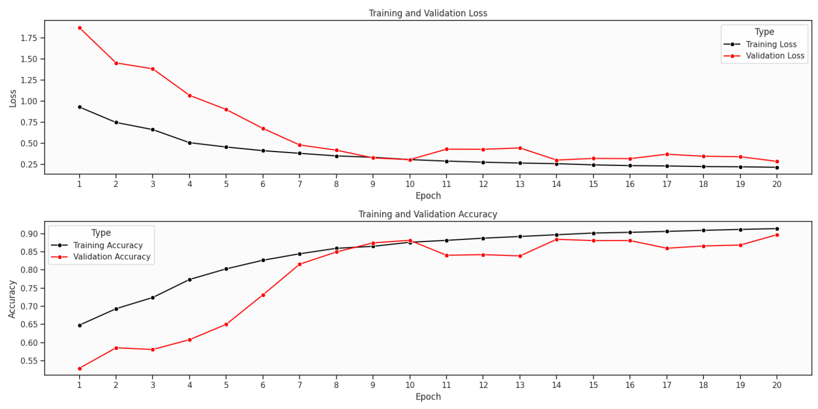

Figure 1.4: Augmented CNN

Chapter 2

Architectures of Neural Networks

2.1 Multi-Layer Perceptron

A Multilayer Perceptron (MLP) is a type of neural network that consists of fully connected layers. It's typically used for classification tasks where inputs are assumed to be vectors of features. However, using an MLP for image data requires flattening the image matrices into vectors, which can sometimes lead to a loss of spatial features that are critical for tasks like image classification. Generally, with special focus on what we implemented, the MLP has the following components:

- **Input Layer:** Flattened the image matrices into vectors, transforming 2D image data into a 1D array suitable for the MLP.
- **Hidden Layer:** It takes input from the previous layer, applies a weighted sum and an activation function, and passes its output to the next layer. An MLP can have multiple hidden layers.
- **ReLU activation function:** ReLU, short for Rectified Linear Unit, is a computationally efficient activation function defined

mathematically as:

$$\text{ReLU}(x) = \max(0, x)$$

- **Dropout Layer:** A dropout layer randomly sets a fraction of input units to zero during each update during training time, which helps prevent overfitting. This technique effectively reduces the network's reliance on any individual neuron, encouraging more robust learning of features across the network.
- **Sigmoid activation function:**

  $$\sigma(x) = \frac{1}{1 + e^{-x}}$$

  This function takes a real-valued number and compresses it into a range between 0 and 1. It is widely used since its gradient, $\sigma'(x) = \sigma(x)(1 - \sigma(x))$, is smooth and nonzero.
- **Output Layer:** A single neuron with a sigmoid activation function was used to predict the probability of the image being a chihuahua or a muffin.

Implementation: For our task of distinguishing chihuahuas and muffins, we used the following architecture:

- **Input Layer.**
- **Hidden Layer 1:** 512 neurons, ReLU activation.
- **Hidden Layer 2:** 256 neurons, ReLU activation.
- **Hidden Layer 3:** 512 neurons, ReLU activation.
- **Hidden Layer 4:** 256 neurons, ReLU activation.
- **Hidden Layer 5:** 512 neurons, ReLU activation.
- **Hidden Layer 6:** 256 neurons, ReLU activation.

- **Hidden Layer 7:** 512 neurons, ReLU activation.
- **Hidden Layer 8:** 256 neurons, ReLU activation.
- **Dropout Layer:** Dropout rate of 0.3.
- **Output Layer:** 1 neuron, Sigmoid activation.

2.2 Convolutional Neural Network

A Convolutional Neural Network (CNN) is specially designed for processing data that has a known grid-like topology, such as images, which are represented as a grid of pixels. The architecture of a CNN is characterized by the presence of convolutional layers that apply convolutional filters to the input. These filters are small matrices of weights that move across the input image to detect patterns such as edges, corners, and textures.

The main components of a CNN include:

- **Convolutional Layers:** These layers perform the convolution operation, which involves sliding a smaller matrix, known as a kernel, over a larger input image in a systematic way to produce a transformed output known as a feature map.
- **Activation Functions:** Typically, a ReLU activation function is used after each convolution operation to introduce non-linearities into the model, helping it to learn more complex patterns.
- **Pooling Layers:** Following convolutional layers, pooling layers (usually max pooling) reduce the spatial dimensions (height and width) of the input volume for the next convolutional layer. This subsampling or downsampling reduces the number of parameters and computation in the network, and helps achieve spatial invariance to input distortions.

- **Fully Connected Layers:** After several convolutional and pooling layers, the high-level reasoning in the neural network is done via fully connected layers. Neurons in a fully connected layer have connections to all activations in the previous layer, and their activations can hence be computed with a matrix multiplication followed by a bias offset.

- **Batch normalization (BN):** It is applied to individual layers in a neural network. It normalizes the inputs of each layer for each mini-batch, meaning it adjusts and scales the activations. Typically, batch normalization is applied just after a convolutional or fully connected layer.

Mathematically, the convolution function can be expressed as:

$$Y[i,j] = \sigma\left(\sum_m \sum_n W[m,n] \cdot X[i-m, j-n] + b\right)$$

where:

- $Y[i,j]$ is the output feature map.
- σ represents the activation function.
- $W[m,n]$ is the kernel applied to the input.
- $X[i-m, j-n]$ is the input image or feature map.
- b is the bias term associated with the filter.
- i, j are the spatial indices for the output feature map.
- m, n are the indices used in the summation over the kernel dimensions.

Application and Implementation: We decided to employ two CNNs, one with a fixed architecture and one tuned with Random Search.

2.2.1 Basic CNN

A straightforward CNN that is easy to train and provides reasonable accuracy for simple image classification tasks.

Architecture:

- **Input Layer:** Standard image input size (150x150x3).
- **Conv Layer 1:** 128 filters, 3x3 kernel, ReLU activation.
- **Max Pooling Layer:** 2x2 pool size.
- **Dropout Layer:** Dropout rate of 0.2.
- **Conv Layer 2:** 64 filters, 3x3 kernel, ReLU activation.
- **Max Pooling Layer:** 2x2 pool size.
- **Dropout Layer:** Dropout rate of 0.2.
- **Conv Layer 3:** 32 filters, 3x3 kernel, ReLU activation.
- **Batch Normalization**
- **Max Pooling Layer:** 2x2 pool size.
- **Dropout Layer:** Dropout rate of 0.2.
- **Flatten Layer:** Flatten output to fit into dense layers.
- **Dense Layer 1:** 512 neurons, ReLU activation.
- **Dense Layer 1:** 256 neurons, ReLU activation.
- **Dropout Layer:** Dropout rate of 0.2.
- **Output Layer:** 1 neuron, Sigmoid activation.

2.2.2 Random search CNN

In our implementation of the random search CNN, we maintained the same architectural framework as the basic CNN, but introduced variations in the number of filters for both the convolutional and dense layers. We will explore these modifications in detail in the next chapter.

2.3 EfficientNet

EfficientNet is a family of CNN architectures designed to provide higher accuracy with fewer parameters and reduced computational cost. Introduced by Google researchers in 2019, EfficientNets use a systematic approach to scaling up CNNs in a more structured manner than previous methods. This approach, called compound scaling, uniformly scales all dimensions of depth, width, and resolution using a simple yet highly effective compound coefficient.

The key features of EfficientNet are:

- **Compound Scaling:** As anticipated, EfficientNet uses a compound coefficient ϕ to uniformly scale network width, depth, and resolution in a principled manner. The base model is scaled up to obtain the other models in the family (B1-B7) by applying this coefficient systematically.
- **Baseline Architecture:** The base model (EfficientNet-B0) is optimized with AutoML MNAS framework, which ensures the model is mobile-friendly with a good trade-off between latency and accuracy.
- **Performance:** EfficientNets have been shown to achieve much higher accuracy and efficiency than other CNNs on benchmarks like ImageNet and CIFAR-100, making them suitable for both high and low-resource environments.

Application and Implementation: For our task, we employed EfficientNet combined with a hyperparameter tuning method called Bayesian Optimization.

- **Base Architecture:** The model utilized was EfficientNetB0, initially leveraging pre-trained weights from ImageNet, which provided a robust foundation for feature extraction. It is the best base architecture of the 8 available, since it is suited for binary classification. Input images are resized to (224,224,3): the only format the architecture can accept.
- **Flatten Layer.**
- **Dense Layer:** Number of neurons searched through Bayesian Optimization.
- **Dropout Layer:** Dropout rate of 0.2.
- **Output Layer:** 1 neuron, Sigmoid activation function.

To implement, we used the class `EfficientNetB0`. Notice how the data augmentation is added beforehand, since the method `get_model_data()` is only called after the tuning process.

```
def search_en(hp):
    base_model = EfficientNetB0(include_top=False, input_shape=
                                    (img_height, img_width, 3),
                                     weights='imagenet')

    data_augmentation = Sequential([
            layers.RandomFlip("horizontal"),
            layers.RandomRotation(factor=(-0.15, 0.15)),
            layers.RandomContrast(0.2),
            layers.RandomZoom(-.1, 0.1),
        ], "Data_Augmentation")

    model_en = Sequential(
        [data_augmentation,
          base_model,
          Dense(hp.Int('dense_units', min_value=64, max_value=
                                                512, step=64),
                                                activation='relu'),
          Flatten(),
          Dense(1, activation='sigmoid')]
    )
    opt = keras.optimizers.Adam(learning_rate=hp.Choice('l_r',
                                        [1e-2, 1e-3, 1e-4]))
    model_en.compile(optimizer=opt, loss='binary_crossentropy',
                                         metrics=['accuracy'])

    return model_en
```

Chapter 3

Hyperparameter Tuning

Hyperparameter tuning is a crucial phase in the development of neural network models, as it helps to optimize their performance by systematically searching for the most effective network configurations. As anticipated, we will discuss the two methods used in our implementations: random search (on a CNN) and Bayesian Optimization (on EfficientNet).

3.1 Random Search

Random Search is a method for hyperparameter tuning that randomly selects combinations of hyperparameters from a predefined search space. Each combination is used to train a model, and its performance is evaluated. To apply Random Search, the following steps were undertaken:

1. **Hyperparameter Definition:** The tuning process begins by defining the key hyperparameters of the CNN that will be adjusted. We worked on:
    - **Convolutional Layers.**
    - **Dense Layer.**
    - **Dropout Layers.**

2. **Search Space Setup:** A search space is established for each hyperparameter, defined to cover a broad range of values.

   - **Number of Filters in Each Convolutional Layer:** Min value: 32, Max value: 128, step: 32.
   - **Number of Neurons in the Dense Layers:**
     (a) **Dense Layer 1:** Min value: 64, Max value: 512, step: 64.
     (b) **Dense Layer 2:** Min value: 64, Max value: 256, step: 64.
   - **Dropout Rate:** Tested values between 0.3 and 0.5. Search done one time and applied to every dropout layer.

   This is done using the `RandomSearch` class from `keras_tuner.tuners`:

```
def search_cnn2(hp):
    model = Sequential([
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(factor=(-0.15, 0.15)),
    layers.RandomContrast(0.2),
    layers.RandomZoom(-.1, 0.1),
    Conv2D(hp.Int('conv_1_filters', min_value=32,
                                    max_value=128, step=32
                                    ), (3, 3), activation=
                                    'relu', input_shape=(
                                    img_width, img_height,
                                     3)),
    MaxPooling2D(2, 2),
    Dropout(hp.Choice("drop_r", [0.3,0.4,0.5])),
    Conv2D(hp.Int('conv_2_filters', min_value=32,
                                    max_value=128, step=32
                                    ), (3, 3), activation=
                                    'relu'),
    MaxPooling2D(2, 2),
    Dropout(hp.Choice("drop_r", [0.3,0.4,0.5])),
    Conv2D(hp.Int('conv_3_filters', min_value=32,
                                    max_value=128, step=32
                                    ), (3, 3), activation=
                                    'relu'),
    BatchNormalization(),
    MaxPooling2D(2, 2),
    Dropout(hp.Choice("drop_r", [0.3,0.4,0.5])),
    Flatten(input_shape=(img_width, img_height, 3)),
    Dense(hp.Int('dense_units_1', min_value=64, max_value=
                                  512, step=64),
                                  activation='relu'),
    Dense(hp.Int('dense_units_2', min_value=64, max_value=
                                  256, step=64),
                                  activation='relu'),
    Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer='adam', loss='
                                  binary_crossentropy',
                                  metrics=['accuracy'])
    return model

tuner = RandomSearch(
    search_cnn2,
    objective='val_accuracy',
    max_trials=15,
    executions_per_trial=1,
    directory='./',
    project_name='rs1',
    overwrite=True
)
```

3. **Random Sampling:** In each trial of the tuning process, random values are selected for each hyperparameter from the defined search space. Unlike grid search, random search does not systematically explore every possible combination, but instead randomly selects combinations. This approach can often lead to discovering effective configurations more quickly by avoiding less promising areas of the search space. This is done through the `search` function:

```
tuner.search(x_train,y_train, epochs=10, batch_size=32,
                                validation_data=(x_val,
                                y_val))
```

4. **Model Training and Evaluation:** Each set of randomly selected hyperparameters is used to configure and train a new CNN model. Model performance is assessed using accuracy on a validation dataset.

5. **Performance Tracking:** After each training session, the performance of the CNN is recorded. The primary objective is to determine which configurations of hyperparameters yield the best results on the validation dataset.

6. **Iteration and Optimization:** This process is repeated for a predefined number of trials (in our implementation we chose 15). With each iteration, there is the potential to uncover a more optimal set of hyperparameters, progressively refining the model's effectiveness.

7. **Best values found:** Using `tuner.get_best_hyperparameters()` we discovered the best hyperparameters:

```
'conv_1_filters': 32,
'conv_2_filters': 96,
'conv_3_filters': 64,
'drop_r': 0.3,
'dense_units_1': 384,
'dense_units_2': 256
```

3.2 Bayesian Optimization

Bayesian Optimization represents a sophisticated approach to hyperparameter tuning. It constructs a probabilistic model of the objective function, which, in our case, is the validation accuracy of the EfficientNet model. This method efficiently identifies promising hyperparameters for evaluation based on prior results, making it highly suitable for scenarios where each model evaluation is costly.

For large-scale problems with a manageable number of hyperparameters, Bayesian optimization is often the preferred method, potentially finding optimal solutions more quickly and with fewer function evaluations than random search.

Since EfficientNet is a pre-trained model, the hyperparameter space is relatively narrow. In our implementation we focused on:

- **Learning Rate:** Varied between 0.0001 and 0.01, this parameter will significantly influence the model's ability to converge to a global minimum efficiently.
- **Dense Layer:** Neurons ranging from 64 to 512 with `step`=64.

Similarly to random search, Bayesian optimization is done using a class from `keras_tuner.tuners`:

```
tuner = BayesianOptimization(
    search_en,
    objective='val_accuracy',
    max_trials=10,
    executions_per_trial=1,
    directory='./',
    project_name='chihuahua_muffin_ef',
    overwrite=True
)
```

Best hyperparameters found:

```
    'dense_units': 192,
    'l_r': 0.0001
```

Chapter 4

Training of Neural Networks

4.1 Training Process

Each model underwent a rigorous training process, utilizing the architectures and hyperparameters identified as optimal in the previous chapters. The training was conducted using a GPU-accelerated environment to handle the computational demands efficiently.

4.2 Techniques Employed

Backpropagation: Used across all models to minimize loss functions, specifically using Adam optimizer for its adaptive learning rate capabilities.

Early Stopping: Monitored validation loss to stop training when improvement ceased, ensuring efficient use of computational resources. We chose `patience`=3.

4.3 Training Outcomes

The training process was monitored through loss and accuracy metrics:

- **MLP:** Reached a training accuracy of 74.51%, and training loss of 0.53.
- **Basic CNN:** Training accuracy: 91.5%, training loss: 0.20.
- **RandomSearch CNN:** Improved to 94.7% in training accuracy and 0.14 in training loss.
- **EfficientNet:** Demonstrated superior performance, with train accuracies of 99.7% and train loss of 0.012.

To monitor the training process, 20% of the dataset was allocated for validation. In the absence of overfitting, both training and validation accuracy should increase at similar rates after each epoch, while their respective losses should decrease. The following plots illustrate this pattern across all models:

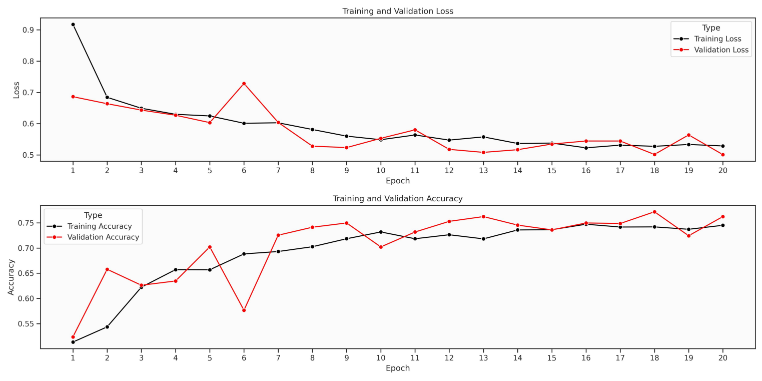

Figure 4.1: MLP training plot

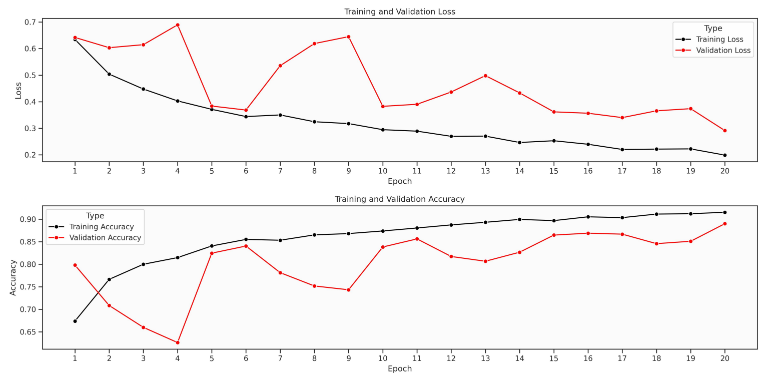

Figure 4.2: Basic CNN training plot

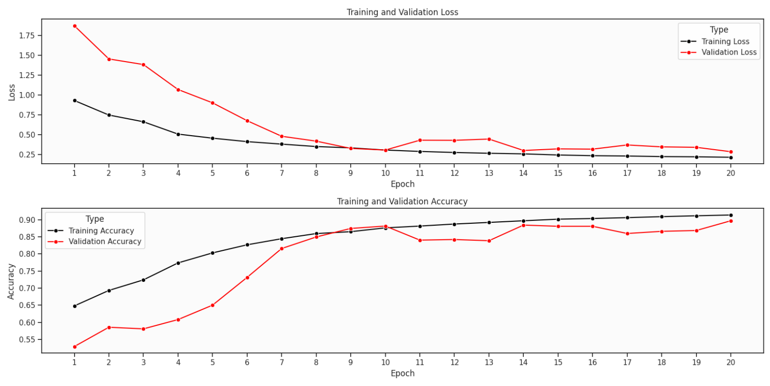

Figure 4.3: RandomSearch CNN training plot

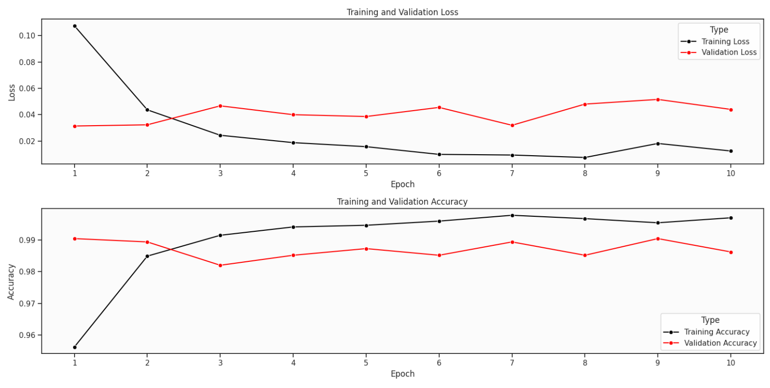

Figure 4.4: EfficientNet training plot

4.4 Risk Estimate

Risk estimate refers to predicting how well a model will perform on new data, outside of the training sample. The "risk" is often quantified as the expected value of some loss function over the distribution of all possible data sets.

In our implementation we used the zero-one loss function, mathematically defined as follows:

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

where y is the true label and $\hat{y}$ is the predicted label.

We computed the risk estimate using 5-fold cross validation, which works as follows:

1. **Dividing the Data:** The dataset is split into five distinct subsets or "folds."
2. **Training and Validation Cycles:** For each fold:
   - The model is trained on four of the folds (80% of the data).
   - The model is then tested on the remaining fold (20% of the data) that the model hasn't seen during training.
   - The loss rate for each validation phase is recorded.
3. **Averaging Results:** The error rates from each of the five validation cycles are averaged. This average error rate l^{CV} is considered an estimate of the model's generalization error or risk.

Since every data point gets to be in both the training and validation set exactly once, the risk estimate is less biased towards any particular subset of the data, and averaging the outcomes across multiple folds helps in smoothing out variability due to any anomalies or outliers in a single fold, providing a more robust estimate of the model's performance.

5-fold cross is implemented through the method `KFold` from `sklearn.model_selection`:

```
def run_kfold(augmented_model, num_folds = 5):
    kf = KFold(n_splits=num_folds, shuffle=True, random_state=
                                    42)
    fold_histories = []
    fold_scores = []
    early_stop = keras.callbacks.EarlyStopping(monitor='loss',
                                    patience=3)
    fold_no = 1
    for train_index, test_index in kf.split(train_images):
        x_train, x_val = train_images[train_index],
                                        train_images[test_index
                                        ]
        y_train, y_val = train_labels[train_index],
                                        train_labels[test_index
                                        ]
        opt = keras.optimizers.Adam(learning_rate=learningrate)
        augmented_model.compile(optimizer=opt, loss='
                                        binary_crossentropy',
                                        metrics=['accuracy'])
        print(f'Training on fold {fold_no}...')
        history = augmented_model.fit(x_train, y_train,
                                        batch_size=32, epochs=
                                        10, validation_data=(
                                        x_val, y_val), verbose=
                                        2, callbacks=[
                                        early_stop])
        scores = augmented_model.evaluate(test_images,
                                        test_labels)
        fold_histories.append(history)
        fold_scores.append(scores)

        fold_no += 1
    for idx, h in enumerate(fold_histories):
        fold_histories[idx] = pd.DataFrame(h.history)

    return (fold_histories, fold_scores)
```

Chapter 5

Results

5.1 Validation

Post-training, each model underwent a validation phase using the reserved test set, which was not exposed to the models during training:

- **MLP Model:** Achieved a test accuracy of 76.35%, and test loss of 0.4932 reflecting its limitations in handling image data.
- **Basic CNN:** Test accuracy: 89.70%, test loss: 0.2917, showing how a CNN handles well the task of image recognition.
- **Random Search CNN:** Showed better performance, reaching 91.22% accuracy and 0.2403 test loss.
- **EfficientNet:** Stood out once again with a test accuracy of 99.49% and test loss of 0.0135.

5.2 Risk Estimates

As anticipated, we are going to use the zero-one loss as a metric to predict the variance in performance of a model. Here are the results obtained:

- **MLP:** $l^{CV} = 0.300$
- **Basic CNN:** $l^{CV} = 0.233$
- **Random Search CNN:** $l^{CV} = 0.076$
- **EfficientNet:** $l^{CV} = 0.008$

These results clearly show how precise and reliable both RandomSearch CNN and EfficientNet are. Graphically:

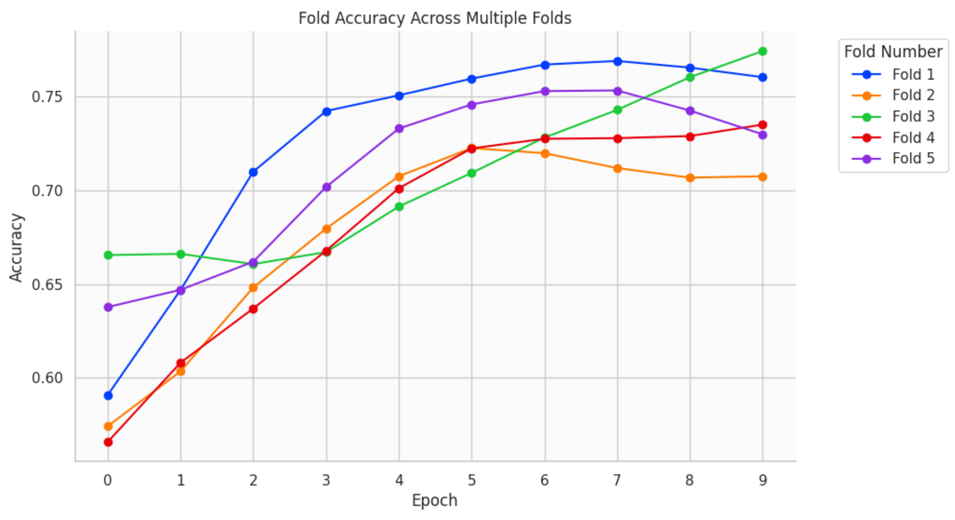

Figure 5.1: 5-fold on MLP

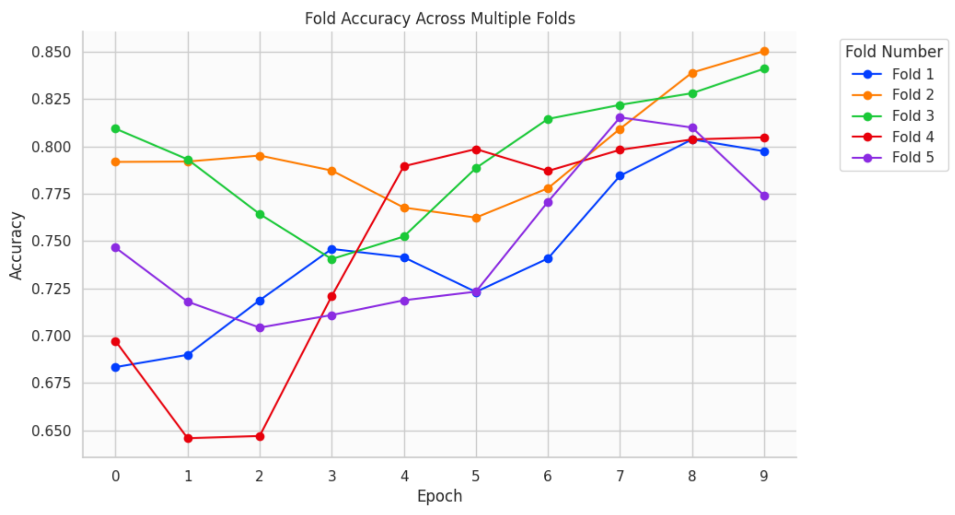

Figure 5.2: 5-fold on Basic CNN

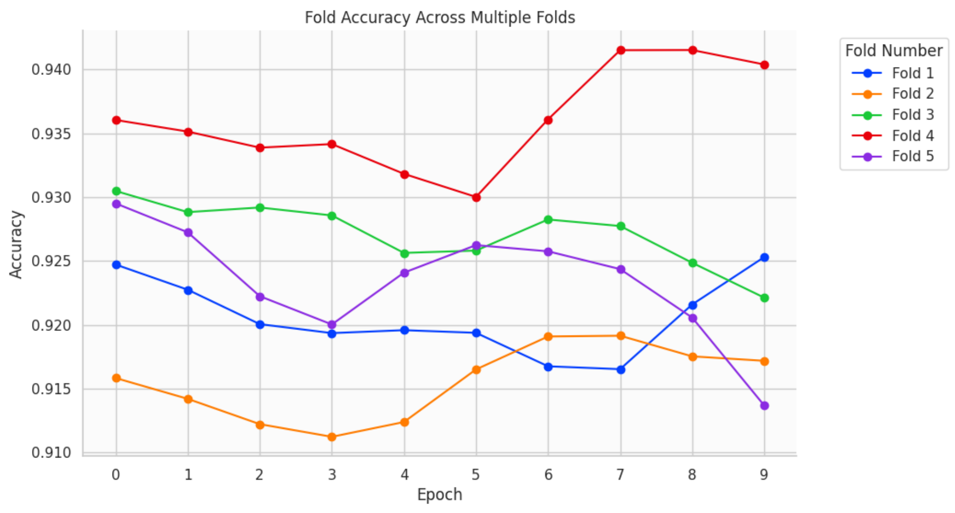

Figure 5.3: 5-fold on RandomSearch CNN

EfficientNet, being extremely precise, was the only model that triggered early stopping, with its cross-validation accuracy consistently ranging between 98.9% and 99.5%.

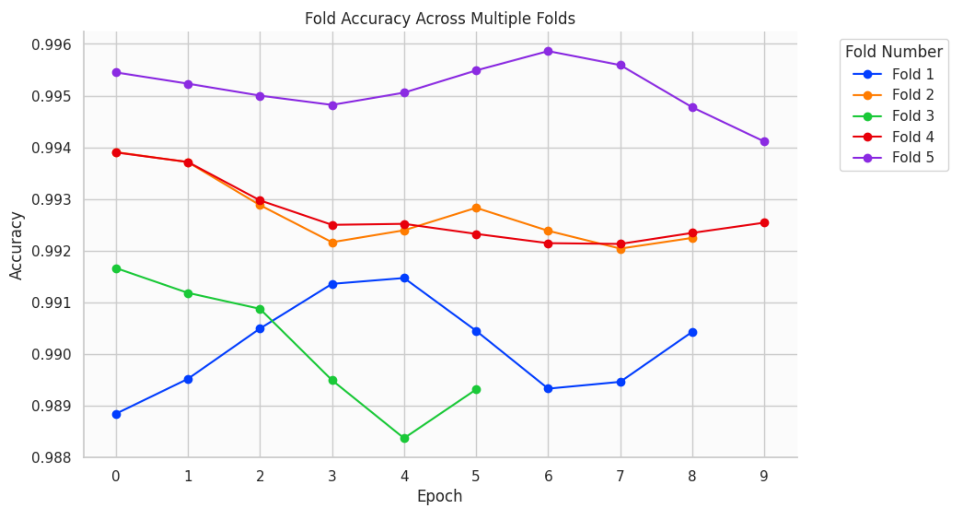

Figure 5.4: 5-fold on EfficientNet

5.3 Evaluation

There are several ways of evaluating a model. We will focus on the following three:

- **Precision:** It measures the accuracy of positive predictions made by the model. It answers the question: "Of all the instances that the model predicted as positive, how many are actually positive?"

  Mathematically, precision is defined as the ratio of true positives (TP) to the sum of true positives and false positives (FP):

  $$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

- **Recall:** Recall measures the ability of the model to correctly identify all positive instances. It answers the question: "Of all the actual positive instances, how many did the model correctly identify as positive?"

  Mathematically, recall is defined as the ratio of true positives (TP) to the sum of true positives and false negatives (FN):

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

- **F1 Score:** F1 Score is the harmonic mean of precision and recall. It provides a balance between precision and recall and is often used as a single metric to evaluate the performance of a classifier.

  Mathematically, F1 Score is defined as:

$$\text{F1 Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

The results obtained can be synthesized in the following table:

MLP	**Precision**	**Recall**	**F1 Score**
Chihuahua (0)	0.752809	0.837500	0.792899
Muffin (1)	0.779661	0.676471	0.724409

Basic CNN	**Precision**	**Recall**	**F1 Score**
Chihuahua (0)	0.933110	0.871875	0.901454
Muffin (1)	0.860068	0.926471	0.892035

RS CNN	**Precision**	**Recall**	**F1 Score**
Chihuahua (0)	0.929487	0.906250	0.917722
Muffin (1)	0.892857	0.919118	0.905797

EfficientNet	**Precision**	**Recall**	**F1 Score**
Chihuahua (0)	0.993769	0.996875	0.995320
Muffin (1)	0.996310	0.992647	0.994475

To encapsulate everything graphically, we will use the confusion matrix, plotted using `matplotlib`:

```
def plot_confusion_matrix(model, filename="cm.png"):
  sns.set(style="whitegrid")
  test_predicted_labels = np.around(model.predict(test_images))
  test_conf_matrix = confusion_matrix(test_labels,
                                      test_predicted_labels)

  plt.figure(figsize=(8, 6))
  sns.heatmap(test_conf_matrix, annot=True, fmt='d', cmap='Reds
                                      ', xticklabels=["Chihuahua",
                                      "Muffin"], yticklabels=["
                                      Chihuahua", "Muffin"])
  plt.xlabel('Predicted Labels', fontsize=16)
  plt.ylabel('True Labels', fontsize=16)
  plt.title('Confusion Matrix')
  plt.show()
  plt.savefig(filename, dpi=300)
```

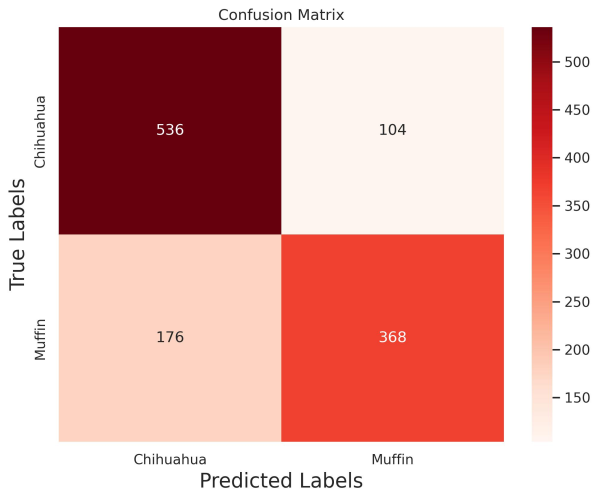

Figure 5.5: MLP Confusion Matrix

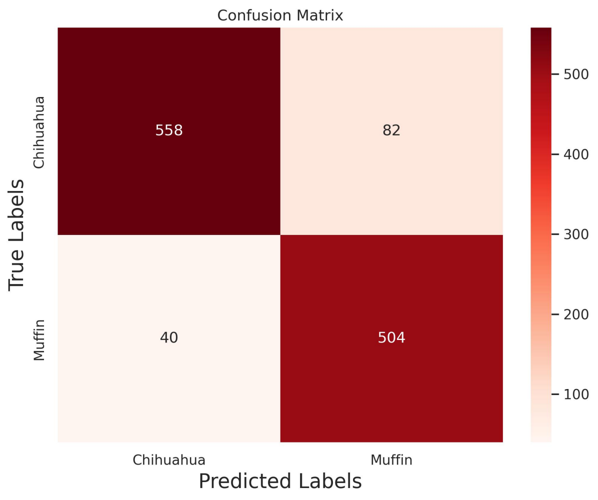

Figure 5.6: Basic CNN Confusion Matrix

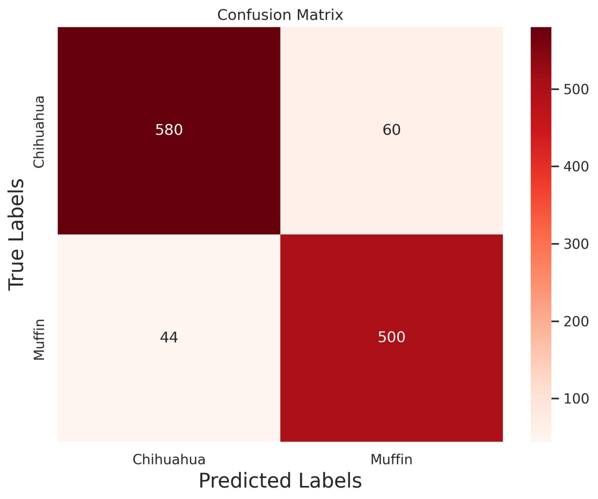

Figure 5.7: RandomSearch CNN Confusion Matrix

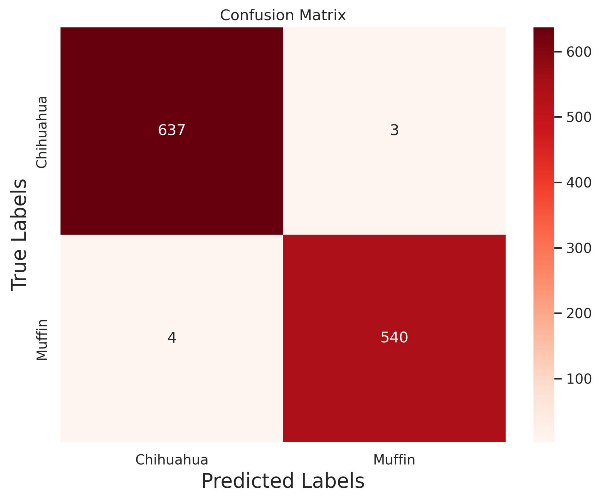

Figure 5.8: EfficientNet Confusion Matrix

5.4 Conclusion

The comprehensive approach to building, tuning, and training neural networks has culminated in robust models capable of accurately classifying images into categories of muffins and chihuahuas. This project not only highlights the effectiveness of various neural network architectures but also underscores the importance of systematic hyperparameter tuning and rigorous training protocols in achieving high-performing AI systems.