UNIVERSITÀ DEGLI STUDI DI MILANO

Convolutional Neural Networks for Binary Image Classification

Francesco Pineschi

23/06/2024

Contents

1 Introduction

In recent years, Convolutional Neural Networks (CNNs) have emerged as a powerful tool in the field of computer vision, with tasks such as image classification or object detection. CNNs are a class of deep neural networks specifically designed to process visual data in order to extract hierarchical features from images.

CNNs have the ability to automatically learn features from raw pixel data, eliminating the need for manual feature engineering. By leveraging convolutional layers, pooling layers, and fully connected layers, CNNs can effectively capture spatial hierarchies and patterns within images, enabling robust and accurate classification performance.

In this study, our objective is to realize the implementation of a CNN for binary image classification with two distinct classes: Chihuahua and Muffin. We seek to explore the CNN's performance and to interpret the effectiveness in terms of accuracy and loss.

2 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks represent a class of deep learning models specifically used for processing visual data. They have demonstrated remarkable performance in various computer vision tasks, ranging from image classification to object detection and segmentation.

An example of CNN application in the real world includes image classification tasks, where CNNs can accurately categorize images into predefined classes. For instance, in medical imaging, CNNs have been employed to diagnose diseases based on X-ray or MRI scans, showcasing their potential in critical healthcare applications.

Key components of CNNs include convolutional layers, pooling layers, and fully connected layers. Convolutional layers apply learnable filters to input data, capturing local patterns and features. Pooling layers reduce spatial dimensions, preserving important features while reducing computational complexity. Fully connected layers integrate extracted features for classification or regression tasks.

By leveraging these components, CNNs can effectively learn representations of image's input data, giving accurate predictions across various domains.

2.1 Convolutional Layers

Convolutional layers are fundamental components of Convolutional Neural Networks. They play a crucial role in feature extraction by applying convolution operations to input images using filters.

Filters: Filters, also known as kernels, are small matrices applied to input images during convolution. Each filter extracts specific features from the input by performing element-wise multiplication and summation operations.

Kernel Size: The kernel size refers to the spatial dimensions of the filter. It determines the receptive field of the filter and influences the types of features extracted. Common kernel sizes include 3x3 and 5x5.

Strides: Strides determine the step size of the filter as it traverses the input image during convolution. A stride of 1 means the filter moves one pixel at a time, while larger strides result in downsampling of the output feature map.

Padding: Padding is the process of adding additional border pixels to the input image before convolution. It helps preserve spatial information and prevent loss of information at the image boundaries. Common padding types include 'valid', which applies no padding, and 'same', which pads the input to ensure the output feature map has the same spatial dimensions as the input.

Activation Function: The activation function introduces non-linearity into the convolutional layer's output. Common activation functions include ReLU (Rectified Linear Unit), which introduces sparsity and addresses the vanishing gradient problem.

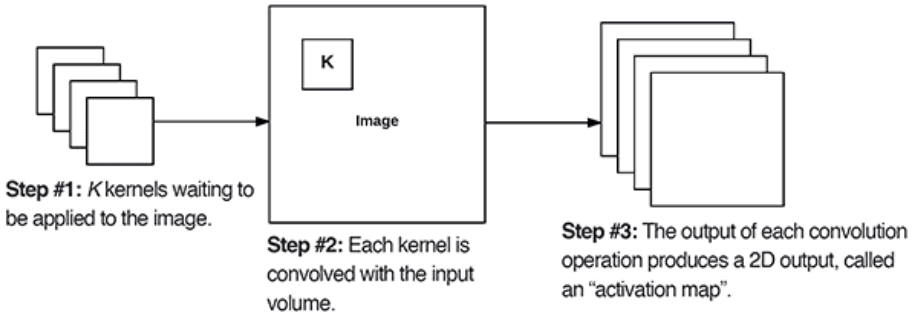

Figure 1: Illustration of a Convolutional Neural Network architecture.

2.2 Pooling Layers

Pooling layers are essential components in Convolutional Neural Networks used for down-sampling and feature reduction. They help reduce the spatial dimensions of the input feature maps, making the network more computationally efficient and reducing overfitting.

Pooling operations typically fall into three main categories:

Max Pooling: Max pooling selects the maximum value from each subregion of the input feature map. It identifies the most significant features while discarding less important ones. Max pooling is the most common type of pooling operation used in CNNs due to its simplicity and effectiveness.

Average Pooling: Average pooling computes the average value from each subregion of the input feature map. It provides a smoothed representation of the features and is less prone to overfitting compared to max pooling.

Global Pooling: Global pooling computes a single value for each feature map by applying a pooling operation across the entire map. It reduces the

spatial dimensions to a single value per feature map, often used as the input to the final classification layer of the network.

Among these categories, max pooling is exclusively chosen for this project due to its simplicity and capability.

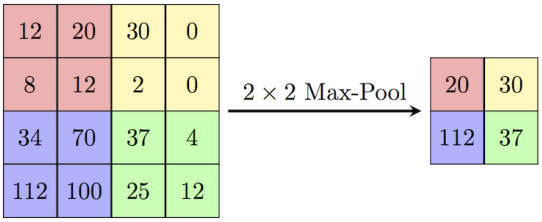

Figure 2: Illustration of Max Pooling (2x2) performed on a matrix (4x4) of integers.

2.3 Fully Connected Layers

Fully connected layers, also known as dense layers, integrate the features extracted by convolutional and pooling layers for final classification or regression tasks. Dense layers connect every neuron in one layer to every neuron in the next layer, enabling global information propagation through the network.

In Keras, fully connected layers are implemented using the `Dense` layer. These layers require input data to be in the form of a one-dimensional vector. However, the output of convolutional and pooling layers is typically a three-dimensional tensor. Therefore, before passing the output to the dense layers, the tensor is flattened into a one-dimensional vector using the `Flatten` layer.

The `Flatten` layer serves the purpose of reshaping the output of the preceding convolutional and pooling layers into a format suitable for input to the dense layers. It collapses the spatial dimensions of the feature maps into a single vector while preserving the relationships between the features.

3 Dataset Modeling

In this chapter, we discuss how the dataset is partitioned, the use of augmentation techniques, and the establishment of stopping criteria.

3.1 Dataset Partitioning

The dataset is partitioned into three subsets: training, validation, and test sets.

- **Training Set**:
  - Used for model training.
  - Contains examples of input (features) and output (labels/targets).
  - During training, the model adjusts its parameters (weights and biases in neural network models) to learn the relationship between input and output.
  - Proportion: 80% of the original dataset.
- **Validation Set**:
  - Used for evaluating model performance during training.
  - Helps in adjusting model parameters.
  - After each training iteration, the model is evaluated on this set to monitor overfitting and generalization.
  - Proportion: 12.5% of the training set.
- **Test Set**:
  - Used for final evaluation of model performance.
  - Contains data unseen during training or validation.
  - The model's performance on this set provides an estimate of its accuracy and predictive capability in real-world scenarios.
  - Proportion: 20% of the original dataset.

3.2 Augmentation

Data augmentation is a technique used to increase the diversity of the training dataset by applying various transformations to the original images. This helps in improving the model's generalization ability and reducing overfitting. Data augmentation methods include rotation, scaling, flipping, and other transformations.

It's important to note that augmentation should be applied to the training set and not the validation set to ensure unbiased evaluation of model performance. During my experiments, i have noticed that the validation accuracy and loss were slightly better if the augmentation was applied also on the validation

dataset. So i have driven my tests with an augmentated validation dataset. By training with augmented data, the models can better handle variations and complexities present in real-world data, leading to more reliable predictions.

In the code, I have applied augemntation using the followiong parameters:

- **rescale**: Normalizes the pixel values of the images.
- **rotation_range**: Specifies the range of rotation in degrees.
- **width_shift_range** and **height_shift_range**: Define the range of horizontal and vertical shifts, respectively.
- **shear_range**: Determines the range of shearing transformations.
- **zoom_range**: Sets the range of zooming transformations.
- **channel_shift_range**: Specifies the range of channel shifts.
- **horizontal_flip**: Enables horizontal flipping of the images.

Two data generators are then created for the training and validation sets. These generators take dataframes containing images and corresponding labels as input and generate batches of augmented images during training.

3.3 Early Stopping Callback

EarlyStopping is a technique used to stop the training of a model if no improvement in performance on the validation set is observed for a specified number of epochs. This helps to save computational resources by stopping training when further optimization is unlikely to yield better results.

- **Monitor**: The "val_loss" parameter is monitored to determine when to stop training. The validation loss is a metric that measures the model's performance on the validation set.
- **Patience**: The "patience" parameter specifies the number of epochs to wait before stopping training if no improvement in the validation loss is observed. In this case, training will be stopped if the validation loss does not improve for 5 consecutive epochs.
- **Start From Epoch**: The "start_from_epoch" parameter specifies the epoch from which to start monitoring the validation loss for early stopping. This allows the model to train for a certain number of epochs before early stopping is applied. In this example, monitoring starts from epoch 15.

The EarlyStopping callback is added to the list `stop_early`, which will be passed to the `fit` method of the model during training. This ensures that early stopping is applied during the training process if the specified conditions are met.

4 Model Implementation

In this chapter, we detail the implementation of three distinct CNN models for binary image classification of Chihuahua and Muffin images. Each model is progressively more complex, with additional layers and dropout regularization applied to improve generalization performance.

4.1 Model 1: Simplest Model

The first model is defined as a sequential stack of layers and is constructed as follows:

- **Convolutional Layers**: Two convolutional layers are employed, each followed by max-pooling layers for downsampling.
- **Flattening and Dense Layers**: The feature maps are flattened into a one-dimensional vector, followed by a dense layer with ReLU activation.
- **Output Layer**: The final layer consists of a single neuron with sigmoid activation for binary classification.

```
model1 = Sequential([
    Conv2D(filters=32, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), strides=2),

    Conv2D(filters=64, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), strides=2),

    Flatten(),
    Dense(units=64, activation='relu'),

    Dense(units=1, activation='sigmoid')
])
```

4.2 Model 2: Model with Dropout

In the second model, dropout regularization is introduced to prevent overfitting. Dropout randomly sets a fraction of input units to zero during training, preventing the network from relying too heavily on any individual feature. This encourages the network to learn more generalized patterns and reduces overfitting by promoting independence among neurons. It is constructed as follows:

- **Convolutional Layers**: Similar to Model 1, two convolutional layers with max-pooling are used.
- **Dropout Regularization**: Dropout layers with a dropout rate of 0.25 are inserted after each max-pooling layer and before the dense layer.

- **Dense Layers**: The architecture concludes with a dense layer with ReLU activation and an output layer with sigmoid activation.

```
model2 = Sequential([
    Conv2D(filters=32, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), strides=2),
    Dropout(0.25),

    Conv2D(filters=64, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), strides=2),
    Dropout(0.25),

    Flatten(),
    Dense(units=64, activation='relu'),
    Dropout(0.3),

    Dense(units=1, activation='sigmoid')
])
```

4.3 Model 3: Final Model with Additional Convolutional Layer

In the third model, an extra convolutional layer is added to further enhance feature extraction:

- **Convolutional Layers**: Three convolutional layers with increasing filter counts are utilized, each followed by max-pooling and dropout layers.
- **Flattening and Dense Layers**: The architecture is concluded with flattening, a dense layer with ReLU activation, dropout regularization, and an output layer with sigmoid activation.

```
model3 = Sequential([
    Conv2D(filters=32, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), strides=2),
    Dropout(0.25),

    Conv2D(filters=64, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), strides=2),
    Dropout(0.25),

    Conv2D(filters=128, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), strides=2),
    Dropout(0.25),

    Flatten(),
    Dense(units=64, activation='relu'),
```

```
    Dropout(0.3),

    Dense(units=1, activation='sigmoid')
])
```

4.4 Training and Augmentation

All three models are trained using a training dataset with and without augmentation techniques to enhance accuracy. During training, the data augmentation methods are applied to the images, effectively increasing the dataset's diversity and aiding the model in learning more robust features. This approach significantly improves the model's ability to generalize to unseen data, as observed from the analysis showing a notable improvement in validation accuracy and a reduction in validation loss.

5 Results

The training results for the three models (Model 1, Model 2, and Model 3) are summarized below, including their performance with and without data augmentation techniques.

5.1 Model 1: Simplest Model

The accuracy and loss of Model 1 without data augmentation are suboptimal, with an accuracy of approximately 70% and a relatively high validation loss.

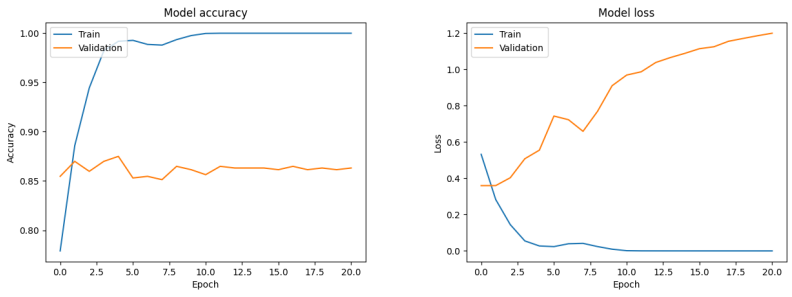

Figure 3: Accuracy and Loss of Model 1 (without augmentation)

5.2 Model 2: Model with Dropout

Model 2 demonstrates improved performance compared to Model 1, achieving a slightly higher validation accuracy and lower validation loss. The introduction of dropout regularization helps mitigate overfitting.

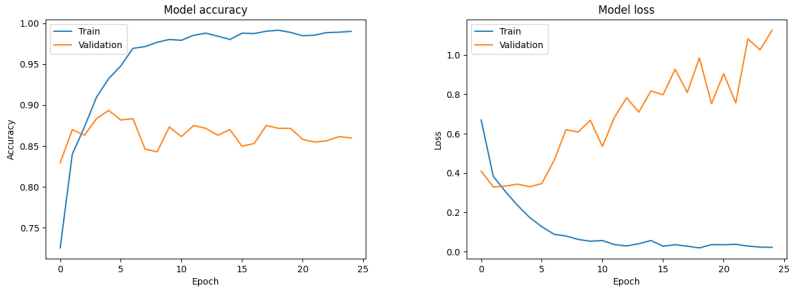

Figure 4: Accuracy and Loss of Model 2 (without augmentation)

5.3 Model 3: Additional Convolutional Layer

Model 3, the most complex architecture among the three models, exhibits slightly superior performance compared to Model 2. The additional convolutional layer improves the validation accuracy and loss.

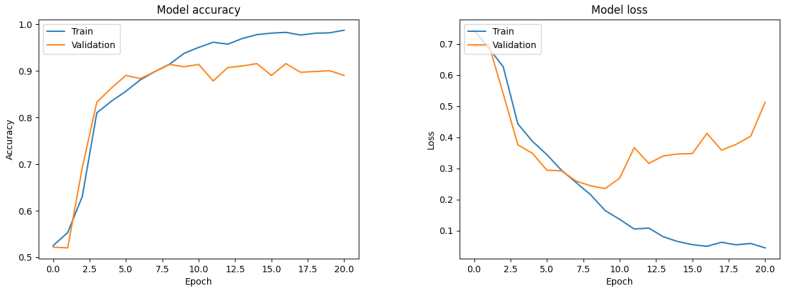

Figure 5: Accuracy and Loss of Model 3 (without augmentation)

5.4 Impact of Data Augmentation

When training the models using augmented data, the performance significantly improves across all models in terms of both accuracy and loss. The improvement of performances can be seen clearly in the following diagrams:

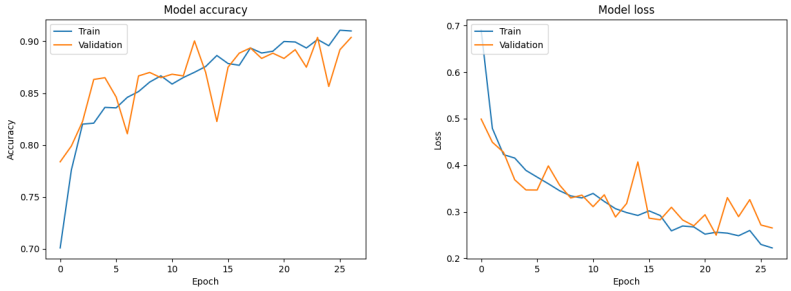

Figure 6: Impact of Data Augmentation on Model 1 Performance

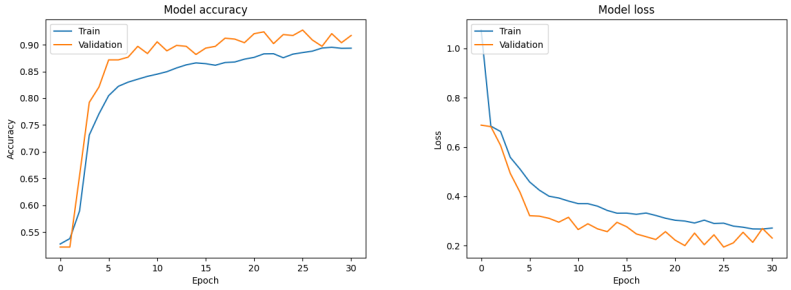

Figure 7: Impact of Data Augmentation on Model 2 Performance

5.5 Best Model: Model 3 with Augmentation

Among the three models trained with data augmentation, Model 3 emerges as the top performer. Its validation accuracy and loss demonstrate the most significant improvements, showcasing the effectiveness of the additional convolutional layer and dropout regularization in handling complex patterns within the dataset.

In summary, Model 3 trained with data augmentation exhibits the highest validation accuracy and the lowest validation loss, making it the optimal choice for binary image classification tasks involving Chihuahua and Muffin classes.

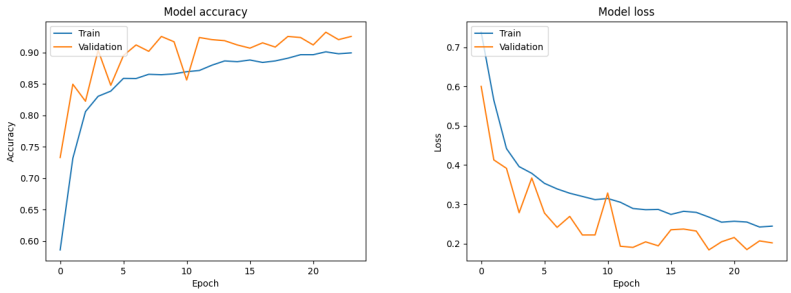

Figure 8: Impact of Data Augmentation on Model 3 Performance

6 Hyperparameter Tuning

Hyperparameter tuning is a critical step in optimizing the performance of a convolutional neural network (CNN) model. It involves the systematic search for the best combination of hyperparameters that maximize the model's performance on validation data. In this context, hyperparameters refer to parameters that are set before the learning process begins and are not updated during training.

To perform hyperparameter tuning, a Bayesian optimization approach is employed using the Keras Tuner library. The objective is to maximize the validation accuracy of the CNN model.

First, a model builder function is defined (`model_builder`) that constructs a convolutional neural network based on specified hyperparameters. The function takes an instance of `HyperParameters` (denoted as `hp`) as input, which allows for tuning of various model architecture parameters such as the number of filters, kernel size, dropout rates, and number of units in dense layers.

The model constructed by `model_builder` closely resembles the architecture of the third model (`model3`) with data augmentation described earlier, which consists of multiple convolutional layers followed by dropout regularization and dense layers.

Next, a Bayesian optimization tuner (`hp_tuner`) is initialized with the following parameters:

- **Objective**: The goal is to maximize the validation accuracy ("val_accuracy") of the model.
- **Max Trials**: The maximum number of trials (model configurations) to evaluate during the search.
- **Directory and Project Name**: These parameters specify the location on google drive to save the results of the optimization process.
- **Overwrite**: If set to `False`, existing optimization results will not be overwritten.

The search space for hyperparameters is defined within the tuner, specifying the ranges and step sizes for each hyperparameter (e.g., filters, dropout rates, number of units).

The tuner then conducts the search (`hp_tuner.search`) using the training data generator (`train_gen_aug`) and validation data generator (`val_gen_aug`) with augmentation. During the search, the model is trained for a fixed number of epochs (50 epochs in this case) while evaluating its performance on the validation set.

After the search completes, the best set of hyperparameters is retrieved (`best_hp`) based on the highest validation accuracy achieved. These hyperparameters are then used to build the final optimized model (`hypermodel`) by calling `hp_tuner.hypermodel.build`.

The resulting `hypermodel` is then trained using the `train_model` function with augmentation enabled.

The best hyperparameters found through the hyperparameter tuning process are as follows:

- **Convolution 1 Filters**: 16
- **Convolution 2 Filters**: 96
- **Convolution 3 Filters**: 192
- **Dropout Hidden Layer**: 0.05
- **Num Units (Dense Layer)**: 96
- **Dropout Flatten Layer**: 0.4

These hyperparameters represent the optimal configuration discovered through the Bayesian optimization search, aiming to maximize the performance of the CNN model for binary image classification.

7 Confusion Matrix and ROC Curve

7.1 Confusion Matrix

The confusion matrix is a table used in classification problems to describe the performance of a classification model on a test dataset where the true classes are known. The confusion matrix consists of four terms:

	Predicted Positive	Predicted Negative
Actual Positive	TP	FP
Actual Negative	FN	TN

Now, regarding the metrics derived from the confusion matrix:

- **Precision**: Precision is a measure of the correctness of positive predictions made by the model. It is defined as the ratio of true positives (TP) to all instances predicted as positive (TP + FP). In simpler terms, it represents the model's ability to not label a negative class as positive.

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

- **Recall (or Sensitivity or True Positive Rate)**: Recall is a measure of the completeness of positive predictions made by the model. It is defined as the ratio of true positives (TP) to all actual positive cases in the dataset (TP + FN). In other words, it indicates the model's ability to find all positive cases.

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

- **F1 Score**: The F1 score is the harmonic mean of precision and recall. It is useful when you want a single measure that combines precision and recall into one metric. The F1 score is calculated using the following formula:

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

- **Support**: The support of a class is the number of occurrences of that class in the test dataset. It is used to calculate the weighted average of precision and recall.

In summary, the confusion matrix and metrics like precision, recall, F1 score, and support provide a detailed assessment of a classification model's performance across different classes in the test dataset. Below are the precision, recall, F1-score, and support for the chihuahua and muffin classes obtained by my final model:

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Chihuahua	0.92	0.97	0.95	624
Muffin	0.97	0.90	0.94	560

Here's the interpretation of the confusion matrix:

	Predicted Chihuahua	Predicted Muffin
Actual Chihuahua	608(TP)	16(FP)
Actual Muffin	54(FN)	506(TN)

- **True Positive (TP)**: The model correctly predicted Chihuahua (608 instances) and Muffin (504 instances).
- **False Positive (FP)**: The model incorrectly predicted Muffin as Chihuahua (16 instances).
- **False Negative (FN)**: The model incorrectly predicted Chihuahua as Muffin (54 instances).
- **True Negative (TN)**: The model correctly predicted not Muffin (TN = 506 instances).

The overall accuracy of the model is 94%, calculated as the ratio of correct predictions (TP + TN) to the total number of predictions.

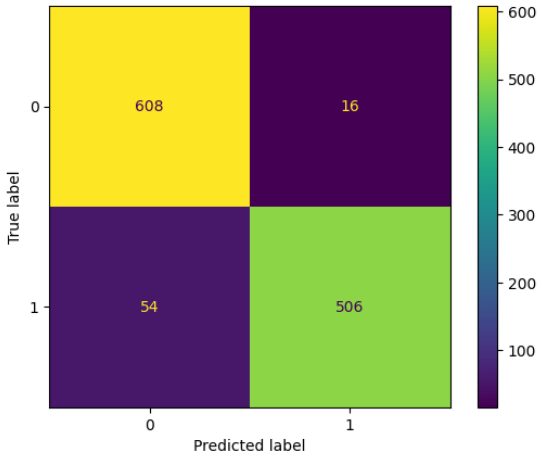

Figure 9: Confusion Matrix

The precision, recall, and F1-score metrics provide a comprehensive evaluation of the model's performance for each class, taking into account both positive and negative predictions.

7.2 ROC Curve

The **ROC curve** (Receiver Operating Characteristic curve) is a graphical representation that illustrates the performance of a binary classification model as

the decision threshold varies. This curve is widely used to evaluate and compare classification models.

The ROC curve is constructed by plotting the **True Positive Rate (TPR)** on the y-axis and the **False Positive Rate (FPR)** on the x-axis. Here's how the ROC curve is obtained and interpreted:

- **True Positive Rate (TPR)**: Indicates the percentage of positive examples correctly classified as positive by the model. It is calculated as:

$$\text{TPR} = \frac{\text{TP}}{\text{P}} = \frac{\text{Number of true positives}}{\text{Total number of positives in the dataset}}$$

- **False Positive Rate (FPR)**: Represents the percentage of negative examples incorrectly classified as positive by the model. It is calculated as:

$$\text{FPR} = \frac{\text{FP}}{\text{N}} = \frac{\text{Number of false positives}}{\text{Total number of negatives in the dataset}}$$

- **Constructing the ROC Curve**:
  1. Using the classification model, calculate the predicted probabilities of belonging to the positive class for each example in the test dataset.
  2. Order the examples based on the predicted probabilities.
  3. Vary the decision threshold from 0 to 1 and, for each threshold, calculate the TPR and FPR.
  4. Plot the TPR vs FPR points to create the ROC curve.

- **Interpreting the ROC Curve**:
  - An ideal ROC curve approaches the top-left corner of the graph, indicating a model that achieves a high TPR while maintaining a low FPR across different decision thresholds.
  - A random model has an ROC curve that follows the diagonal line from (0,0) to (1,1), indicating predictive performance equivalent to random guessing.
  - The **Area Under the Curve (AUC)** of the ROC curve provides an overall measure of the model's performance. AUC value close to 1 indicates an excellent model, while a value close to 0.5 indicates a random model.

The ROC curve is a powerful visualization for understanding and evaluating the performance of a binary classification model across different levels of sensitivity and specificity, helping to select the optimal threshold for classification decisions.

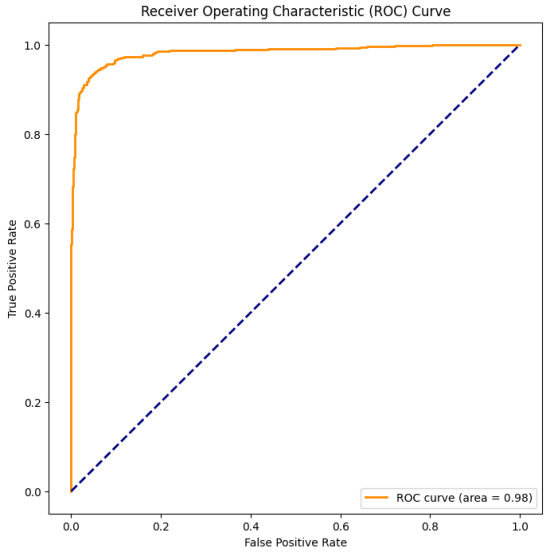

Figure 10: Receiver Operating Characteristic (ROC) Curve with ROC AUC = 0.983067

8 N-fold Cross-Validation Algorithm

n-fold cross-validation is a technique used in machine learning to evaluate the performance of a predictive model. The main purpose of n-fold cross-validation is to assess how well a model performs by repeatedly splitting the dataset into multiple subsets for training and validation.

In n-fold cross-validation:

- The dataset is divided into n equal-sized folds.
- The model is trained n times, each time using a different fold as the validation set and the remaining folds as the training set.
- Performance metrics, such as accuracy, loss, etc., are computed and averaged over the n iterations to provide an estimate of the model's overall performance.

Purpose of n-fold Cross-Validation:

- **Model Evaluation**: It provides a more robust estimate of a model's performance compared to a single train-test split.
- **Generalization Assessment**: It helps assess how well a model generalizes to unseen data, which is crucial for ensuring that the model performs well in real-world scenarios.

- **Hyperparameter Tuning**: It is commonly used for tuning hyperparameters and selecting the best model configuration by comparing performance across different folds.

Overall, n-fold cross-validation is a valuable technique for model evaluation and selection, especially when working with limited data or when robust performance estimates are required.

Suppose we have a dataset D consisting of N training examples. The n-fold cross-validation algorithm can be mathematically represented as follows:

Dataset Division:

$$D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$$

Divide the dataset D into n disjoint subsets:

$$D_1, D_2, \ldots, D_n$$

where $|D_i| = \frac{N}{n}$ for $i = 1, 2, \ldots, n$.

Training and Validation Iterations: For each i-th iteration (where $i = 1, 2, \ldots, n$):

$$\text{Training Set} : D \setminus D_i = \bigcup_{j \neq i} D_j$$

$$\text{Validation Set} : D_i$$

Model Training and Evaluation: Train a model M_i using the training set $D \setminus D_i$ and evaluate it on the validation set D_i to obtain performance metrics.

Performance Aggregation: Compute the average performance metrics across all n models:

$$\text{Performance}_{\text{avg}} = \frac{1}{n} \sum_{i=1}^{n} \text{Performance}(M_i)$$

Performance Metrics Across Folds:

After performing n-fold cross-validation and calculating accuracy and loss metrics for each fold, we can graphically represent these metrics to assess the model's performance on each validation set. In Figure 11, the lines show the accuracy (solid) and loss (dashed) trends for each fold. This visualization highlights performance variations across folds, evaluating model consistency and stability. Aggregating metrics across folds gives an overall performance estimate, helping detect overfitting or underfitting and showcasing the model's generalization on unseen data.

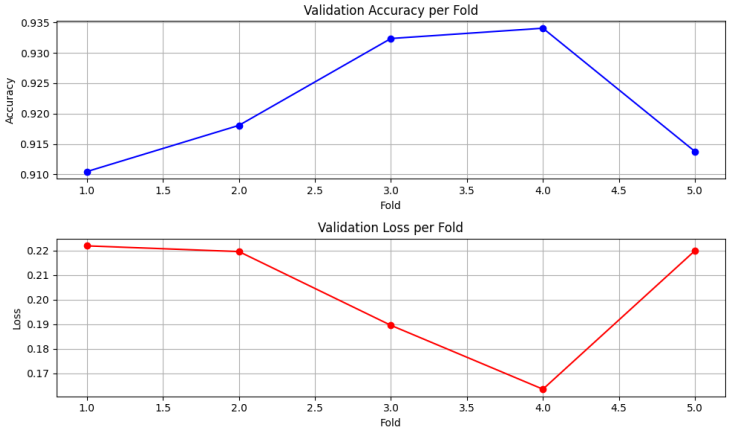

Figure 11: Trend of Accuracy and Loss Across Folds

9 Error Analysis

In this chapter, we show how and in what ways it is possible to conduct a more detailed error analysis. The objective is to understand in which layers and, most importantly, in which regions of the images the classifier makes mistakes. It allows us to identify and understand the reasons behind the model's incorrect predictions, providing valuable insights into how to improve its performance. This helps to identify structural issues in the model, errors in the training data, or specific aspects of the model that need further improvement. In the context of our work, we have employed three main techniques to analyze the errors of our binary classifier. These methods were applied to the best model with optimal hyperparameters obtained through hyperparameter tuning:

1. **Kernel Weights Analysis**: This technique involves visualizing and interpreting the kernel weights in the convolutional layers of the neural network. The kernel weights represent the features learned by the model during training. Analyzing these weights can help understand which patterns the model is trying to recognize and can reveal issues such as overfitting or underfitting.

2. **Activation Function Analysis**: The activation functions in the various layers of the neural network show how information is transformed and propagated through the model. By visualizing the activations, it is possible to identify if certain features of the image are not being correctly detected or if certain activations are too weak or too strong, indicating a potential issue in the model's structure or the quality of the training data.

3. **Grad-CAM Analysis for Each Layer on Misclassified Images**: Grad-CAM (Gradient-weighted Class Activation Mapping) is a technique that allows us to visualize the regions of the image that most influenced the model's decision. For images that were misclassified, we applied Grad-CAM to various convolutional layers to obtain heatmaps showing where the model focused its attention. This analysis helps to better understand which parts of the image led to an incorrect prediction and can suggest how to modify the model's architecture or improve the quality of the data to reduce such errors.

Using these techniques, we can gain a more detailed understanding of the performance of our binary classifier and take specific measures to improve its accuracy and robustness.

9.1 Kernel Weights Analysis

Kernel weights analysis in the convolutional layers of a neural network allows us to understand the features learned by the model during training. The kernels act as filters that extract specific patterns from the images, such as edges and textures.

Explanation of the Code

The code extracts the kernel weights from a specified convolutional layer of the model and normalizes them for visualization. The kernel weights are then displayed as images, separated by color channel. This technique allows us to directly observe the patterns that the model has learned to recognize, providing insights into the model's capabilities and potential areas for improvement.

Limitations of the Technique

Despite its advantages, kernel weights analysis has some limitations. This technique is primarily useful for the early convolutional layers, where the kernels extract simple features such as edges and textures. In the deeper layers, the kernels become more complex and difficult to interpret visually. Additionally, this analysis does not provide information on how these features are combined to make predictions, limiting the overall understanding of the model's behavior.

9.2 Activation Function Analysis

Activation function analysis in the various layers of a neural network shows how information is transformed and propagated through the model. By visualizing the activations, it is possible to identify if certain features of the image are not being correctly detected or if certain activations are too weak or too strong.

Explanation of the Code

The code extracts and visualizes the activation functions of a specified layer in the model. This involves creating an intermediate model that ends at the layer of interest. By passing an image through this intermediate model, we can observe how the neurons in the layer are activated. Each filter in the convolutional layer generates an activation map, which is displayed as an image.

This allows us to see which parts of the original image most influence the model's decisions.

Limitations of the Technique

While useful, activation function analysis has some limitations. The activation maps of deeper layers can be difficult to interpret as they represent complex and abstract features. Moreover, this technique does not provide a quantitative understanding of the importance of different activations, making it necessary to combine it with other techniques to get a complete picture of the model's behavior.

9.3 Grad-CAM Analysis for Each Layer on Misclassified Images

Grad-CAM (Gradient-weighted Class Activation Mapping) analysis allows us to visualize the regions of an image that most influenced the model's decision. This approach provides a heatmap overlaid on the original image, indicating which parts of the image were most relevant for the classification. Applying Grad-CAM to various convolutional layers, we can gain a detailed understanding of how the model interprets images, especially those that result in misclassifications.

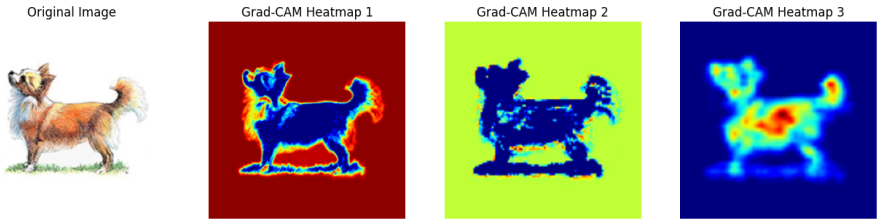

Figure 12: Heatmap for each Convolutional Layer

Explanation of the Code

The code extracts Grad-CAM heatmaps for an image that the model misclassified. These heatmaps show the areas of the image that most influenced the model's decision. The analysis is performed for three convolutional layers of the model, allowing us to observe how different features are detected at various depths in the network.

- **Intermediate Model Construction**:

```
grad_model = Model(
[model.inputs],
[model.get_layer(last_conv_layer_name).output,model.output]
)
```

This creates an intermediate model that returns the output of the specified last convolutional layer and the final output of the model.

- **Gradient Calculation**:

```
with tf.GradientTape() as tape:
    last_conv_layer_output, preds = grad_model(img_array)
    if pred_index is None:
        pred_index = tf.argmax(preds[0])
    class_channel = preds[:, pred_index]

grads = tape.gradient(class_channel, last_conv_layer_output)
```

  GradientTape is used to record operations and compute the gradients of the class output with respect to the last convolutional layer's output.

- **Pooling Gradients**:

```
pooled_grads = tf.reduce_mean(grads, axis=(0, 1, 2))
```

  The gradients are averaged across all spatial dimensions to obtain an average value for each channel.

- **Heatmap Generation**:

```
heatmap = last_conv_layer_output @ pooled_grads[..., tf.newaxis]
heatmap = tf.maximum(heatmap, 0) / tf.math.reduce_max(heatmap)
```

  The averaged gradients are combined with the last convolutional layer's output to generate the heatmap, which is then normalized to have values between 0 and 1.

- **Heatmap Visualization**:

```
heatmaps = [np.uint8(255 * heatmap) for heatmap in heatmaps]
jet = cm.get_cmap("jet")
jet_heatmap = tf.keras.preprocessing.image.array_to_img(jet_heatmap)
jet_heatmap = jet_heatmap.resize((img.shape[1], img.shape[0]))
axes[i + 1].imshow(jet_heatmaps[i].astype("uint8"))
```

  The heatmaps are converted to uint8 for correct display with 'imshow'. The "jet" colormap is used to represent the heatmap, which is converted to an image and resized to match the original image's size. The heatmaps are then displayed next to the original image.

- **Clipping Input**

  Clipping input is performed when converting the heatmap values to uint8, ensuring that the values are within the range [0, 255]. This is important because Matplotlib's 'imshow' expects values in this range to correctly display RGB images.

- **Heatmap Normalization**

  The heatmap normalization ensures that the heatmap values are between 0 and 1. This process improves the heatmap's visibility, making the distinction between high and low activation regions clearer.

- **Image Conversion**

  Image conversion is performed to transform the normalized heatmap into an image format that can be easily displayed. This step ensures that the heatmaps are correctly resized and displayed next to the original image, providing an immediate visual comparison.

Advantages of Grad-CAM over Other Methods

Grad-CAM analysis offers several advantages over kernel weights and activation function analysis:

1. **Intuitive Visual Interpretation**: Grad-CAM provides a heatmap overlaid on the original image, making it immediately clear which areas influenced the model's decision. This is more intuitive compared to interpreting kernel weights or activations, which can be abstract and hard to understand visually.

2. **Multi-Level Analysis**: By applying Grad-CAM to different convolutional layers, it is possible to see how features are detected at various depths in the network. This offers a more comprehensive understanding of the model's decision-making process compared to analyzing individual weights or activations.

3. **Focus on Misclassified Images**: Grad-CAM is particularly useful for analyzing images that the model misclassified. It provides specific insights into where the model went wrong, allowing targeted modifications to the model's architecture or the training data.

4. **Combines Quantitative and Qualitative Information**: While kernel weights and activation function analysis provide quantitative information, Grad-CAM combines this with a qualitative visual representation, offering a more complete and detailed view of the model's performance.

In summary, Grad-CAM analysis represents a more complete and detailed approach to understanding the model's behavior, especially for images that result in misclassifications, compared to other analysis methods.

10 Conclusion

In this document, we have explored the implementation and optimization of convolutional neural network (CNN) models for binary image classification of Chihuahua and Muffin images. We started by designing and progressively enhancing three distinct CNN architectures: Model 1, Model 2 with dropout regularization, and Model 3 with an additional convolutional layer. Each model was evaluated with and without data augmentation techniques to improve generalization performance.

Through hyperparameter tuning using Bayesian optimization with the Keras Tuner library, we identified the optimal set of hyperparameters that maximized the validation accuracy of our CNN model.

The final CNN model, trained with the optimized hyperparameters and data augmentation, achieved an overall accuracy of 94% on the test dataset. The confusion matrix analysis revealed high precision and recall for both Chihuahua and Muffin classes, demonstrating the model's ability to effectively distinguish between the two categories.

Furthermore, the ROC curve analysis showed performance with an Area Under the Curve (AUC) of 0.983, indicating strong predictive capabilities across different decision thresholds.

In summary, our study underscores the significance of model design, hyperparameter optimization, and evaluation techniques such as confusion matrix and ROC curve in developing robust CNN models for image classification tasks.