Neural Network: Binary Classification

Andrei Skobtsov

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offenses in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

June 7, 2024

Chapter 1

Abstract

Convolutional Neural Networks (CNNs) have emerged as a powerful tool for image recognition and classification tasks. This research project aims to explore the application of CNNs in solving a binary classification problem: distinguishing between images of chihuahuas and muffins. By developing simple CNN models from scratch, I gain insights into how input manipulation and architectural modifications impact the model's performance. The project begins with an understanding of the fundamental components of CNNs, including convolutional layers, pooling layers, and dense layers. I discuss the role of backpropagation in optimizing the model's parameters and introduce the concept of data augmentation to mitigate overfitting. Through a series of experiments, I develop and evaluate three CNN models with increasing complexity. Starting with a base model, I progressively incorporate techniques such as data augmentation and dropout layers to improve the model's performance. The models are assessed using various metrics, including loss, accuracy, precision, recall, and F1-score. To further optimize the model, I employ hyperparameter tuning using the Bayesian optimization approach. This process helps identify the optimal combination of hyperparameters that maximizes the model's performance. Finally, I evaluate the tuned model's performance on a test set and present the results using a confusion matrix, classification metrics, and the Receiver Operating Characteristic (ROC) curve. This research project demonstrates the effectiveness of CNNs in solving binary classification problems and highlights the importance of careful model design, data augmentation, and hyperparameter tuning in achieving optimal performance. The insights gained from this project can be extended to more complex image classification tasks and serve as a foundation for further research in the field of deep learning.

Chapter 2

Introduction

Convolutional Neural Networks (CNNs) have revolutionized the field of computer vision, enabling machines to interpret and understand visual data with unprecedented accuracy. These powerful architectures have become the go-to solution for a wide range of tasks, including image classification, object detection, and semantic segmentation. At the heart of CNNs lies the ability to automatically learn hierarchical representations of visual features, allowing them to capture intricate patterns and abstract concepts within images.

2.1 Convolutional Neural Networks(CNN)

An image is represented as a three-dimensional array of pixels, characterized by its height, width, and depth (d). The depth dimension corresponds to the color channels, with grayscale images having a single channel (d=1) and RGB color images having three channels (d=3). Each pixel holds an intensity value, typically ranging from 0 to 255, which indicates the brightness or color at that specific location within the image. CNNs leverage this spatial structure of images by employing a hierarchical architecture composed of three primary types of layers: convolutional layers, pooling layers, and dense layers. Each layer plays a distinct role in processing and transforming the visual information, enabling the network to learn increasingly complex features and representations.

2.1.1 Convolutional Layers

Convolutional layers form the backbone of CNNs, responsible for detecting and learning local patterns and features within an image. These layers operate by applying a set of learnable filters, also known as kernels, to the input image through the mathematical operation of convolution. The filters slide across the image, computing the dot product between the filter weights and the corresponding pixel values at each position. This process allows the network to identify and capture relevant features, such as edges, textures, and shapes, at different locations within the image. The output of a convolutional layer is a feature map, which represents the presence and strength of the detected features at each position. By learning multiple filters, the network can extract a diverse set of features, enabling it to capture various aspects of the visual input. The spatial arrangement of the feature maps preserves the relative positions of the features, allowing the network to maintain the structure of the image. After each convolutional layer, an activation function is applied to introduce non-linearity into the network. The most

commonly used activation function in CNNs is the Rectified Linear Unit (ReLU), defined as $f(x) = \max(0, x)$. ReLU effectively sets negative values to zero while leaving positive values unchanged, introducing a non-linear transformation that allows the network to learn more complex representations.

2.1.2 Pooling Layers

Pooling layers play a crucial role in reducing the dimensions of the feature maps generated by the convolutional layers. The main purpose of pooling is to extract the most salient features while maintaining a degree of spatial invariance. By downsampling the feature maps, pooling layers help to reduce computational complexity and prevent overfitting by focusing on the most informative aspects of the features. The most common types of pooling operations are max pooling and average pooling. Max pooling selects the maximum value within a specified window size, typically 2x2 or 3x3, and discards the remaining values. This operation effectively captures the most prominent features and provides a degree of translation invariance. Average pooling, on the other hand, computes the average value within the window, providing a smoothed representation of the features.

2.1.3 Dense Layers

As the visual information propagates through the convolutional and pooling layers, the network gradually transitions from capturing low-level features to higher-level semantic concepts. The final stages of a CNN typically involve dense layers, also known as fully connected layers. These layers take the flattened feature maps as input and perform high-level reasoning and classification tasks. Dense layers learn to combine and interpret the extracted features, enabling the network to make predictions or decisions based on the input image. The neurons in dense layers are fully connected, meaning that each neuron receives input from all the neurons in the previous layer. This connectivity allows the network to capture complex relationships and dependencies among the features. The last dense layer in a CNN often has a number of neurons corresponding to the number of classes in the classification task. The outputs of this layer are typically passed through a sigmoid activation function, which produces a probability distribution over the classes. The class with the highest probability is considered the predicted class for the input image.

2.2 Training Convolutional Neural Networks

Training a CNN involves the process of adjusting the network's learnable parameters, such as the weights and biases of the convolutional filters and dense layers, to minimize a loss function that quantifies the discrepancy between the predicted outputs and the ground truth labels. This process is known as backpropagation, and it allows the network to learn from examples and improve its performance over time.

2.2.1 Backpropagation

Backpropagation is the fundamental algorithm used to train CNNs. It involves computing the gradients of the loss function with respect to each learnable parameter in the network using the chain rule of differentiation. The gradients indicate the direction and magnitude

of the adjustments needed to minimize the loss function. During backpropagation, the gradients are calculated starting from the output layer and propagated backward through the network, layer by layer. The gradients are used to update the weights and biases of the network using optimization techniques. These optimization methods iteratively adjust the parameters in the direction that reduces the overall error, allowing the network to learn and improve its predictions. The learning rate is a crucial hyperparameter in the training process, as it determines the step size at which the parameters are updated. A higher learning rate allows for faster convergence but may lead to overshooting the optimal solution, while a lower learning rate ensures more stable updates but may result in slower convergence. Finding the right balance and adapting the learning rate during training is essential for efficient and effective learning. The backpropagation is named as such because the parameter values are computed through a procedure that starts from the end of the neural network: for each parameter, the gradient descent is calculated, which is the derivative of the loss function with respect to the parameter. The new value of the parameter is determined by the difference between the value taken in the previous step and the product of the derivative value and the learning rate. This allows me to move along the loss function. In the initial phase, the learning rate will be relatively high, but after several steps, as I approach the minimum of the function, it will assume relatively low values. In this project the Adam optimizer was employed due to its notable effectiveness. Without delving too deeply into the technical details, I can describe Adam as an algorithm capable of dynamically adapting learning rates for various weights in the neural network through the computation of a moving average of the gradient and its corresponding variance.

Chapter 3

Methodology

In this chapter, I present the methodology employed in my research project, which focuses on the application of CNNs for binary image classification. I provide a detailed description of the data preprocessing steps, model architecture, hyperparameter tuning, and evaluation metrics used throughout the study. The aim is to establish a comprehensive framework.

3.1 Data Preprocessing

Data preprocessing is a crucial step in any machine learning project, as it directly impacts the quality and performance of the trained models. To organize and manage the dataset effectively, I create a structured dataframe that contains the file paths and corresponding class labels for each image. This dataframe serves as a central reference for data splitting and generator creation.

3.1.1 Data Splitting

To ensure a robust evaluation of my CNN models, I split the dataset into three distinct subsets:

- **Training Set**: This subset contains the majority of the images and is used to train the CNN models. The models learn the underlying patterns and features from these images to make accurate predictions.
- **Validation Set**: The validation set is used to assess the performance of the models during training. It helps identify potential issues such as underfitting or overfitting and allows for model selection and hyperparameter tuning.
- **Test Set**: The test set is kept separate from the training and validation sets and is used to evaluate the final performance of the trained models. It provides an unbiased estimate of how well the models generalize to unseen data.

I allocated 75% of the datatset for training, 10% for validation, and the remaining 20% for testing.

3.1.2 Image Data Generators

To efficiently feed the images to the CNN models during training and evaluation, I utilize the Keras ImageDataGenerator class. This class provides a convenient way to load images from directories, apply data augmentation techniques, and generate batches of normalized images. The key parameters of the ImageDataGenerator include:

- **dataframe**: The reference dataframe containing file paths and class labels.
- **directory**: The directory where the images are stored.
- **x col and y col**: The dataframe columns specifying the file names and class labels, respectively.
- **target size**: The desired dimensions to resize the images.
- **batch size**: The number of images to include in each batch during training and evaluation.
- **class mode**: The type of class labels (e.g., binary for two classes).
- **color mode**: The color space of the images (e.g., rgb for color images, grayscale for black and white images).
- **shuffle**: Whether to shuffle the images after each epoch.

By configuring these parameters appropriately, I can ensure that the images are loaded efficiently and preprocessed consistently across the different subsets.

3.1.3 Data Augmentation

Data augmentation is used to artificially increase the size and diversity of the training dataset. It involves applying various transformations to the images, such as rotation, scaling, flipping, and shifting, to create new variations of the original samples. Data augmentation helps in reducing overfitting and improving the generalization ability of the models. In my study, I create additional image generators specifically for the training and validation sets, not for the test set.

- **Rotation**: Randomly rotating the images within a specified range of angles.
- **Width and Height Shift**: Shifting the images horizontally and vertically by a certain fraction of the image dimensions.
- **Shear and Zoom**: Applying shear and zoom transformations to the images.
- **Horizontal and Vertical Flipping**: Randomly flipping the images horizontally or vertically.
- **Channel Shift**: Randomly shifting the color channels of the images.

By applying these augmentations, I can create a more diverse and robust training datasets.

3.2 Model Architecture

The architecture of a CNN plays a crucial role in its ability to learn and extract meaningful features from images. In this section, I describe the different components and layers that make up my CNN models.

3.2.1 Convolutional Layers

Convolutional layers are the backbone of CNNs, responsible for learning local patterns and features from the input images. These layers consist of a set of learnable filters (also known as kernels) that convolve over the image, performing element-wise multiplication and summation to produce feature maps. The key parameters of convolutional layers include:

- **Number of Filters**: The number of filters determines the depth of the output feature maps. Each filter learns to detect specific patterns or features in the input.
- **Kernel Size**: The kernel size defines the dimensions of the filters.
- **Padding**: Padding adds extra pixels around the edges of the input to control the spatial dimensions of the output feature maps.
- **Activation Function**: An activation function is applied element-wise to introduce non-linearity into the network.

By stacking multiple convolutional layers, the CNN can learn hierarchical features, starting from low-level edges and textures to high-level semantic concepts.

3.2.2 Pooling Layers

Pooling layers are used to downsample the spatial dimensions of the feature maps, reducing the computational complexity and providing a form of translation invariance. Pooling layers are typically inserted between convolutional layers to progressively reduce the spatial dimensions while maintaining the depth of the feature maps.

3.2.3 Dropout Layers

Dropout is a regularization technique used to prevent overfitting in neural networks. It randomly sets a fraction of the input units to zero during training, forcing the network to learn more robust and generalizable features.

- **Dropout Rate**: The dropout rate determines the fraction of input units to be randomly dropped out. The value I chose is 0.5, which means that half of the units are dropped out in each training iteration.

3.2.4 Dense Layers

Dense layers (also known as fully connected layers) are used in the final stages of the CNN to perform high-level reasoning and classification. These layers take the flattened output from the previous layers and apply a linear transformation followed by an activation function.

- **Number of Units**: The number of units in a dense layer determines the dimensionality of the output space. It can be adjusted based on the complexity of the classification task.
- **Activation Function**: For binary classification, I used the sigmoid activation, producing a probability value between 0 and 1.

3.3 Model Training and Evaluation

Once the CNN architecture is defined, the next step is to train the model using the prepared dataset and evaluate its performance. This section describes the key aspects of model training and evaluation.

3.3.1 Loss Function

For binary classification i chose the binary cross-entropy as my loss function. It measures the dissimilarity between the predicted probabilities and the true labels, penalizing incorrect predictions more heavily. The binary cross-entropy loss is defined as:

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(p_i) + (1 - y_i)\log(1 - p_i)] \tag{3.1}$$

where N is the number of samples, y_i is the true label (0 or 1) of the i-th sample, and p_i is the predicted probability of the i-th sample belonging to class 1.

3.3.2 Optimizer

The optimizer is responsible for updating the model's parameters based on the computed gradients during training. In this study, I employed the Adam optimizer, which adapts the learning rate for each parameter based on the historical gradients and their squared values.

3.3.3 Metrics

Evaluation metrics provide quantitative measures of the model's performance. These metrics include:

- **Accuracy**: The proportion of correctly classified samples out of the total number of samples.
- **Precision**: The proportion of true positive predictions among all positive predictions.
- **Recall**: The proportion of true positive predictions among all actual positive samples.
- **F1 Score**: The harmonic mean of precision and recall, providing a balanced measure of the model's performance.
- **Specificity**: The proportion of true negative predictions among all actual negative samples.

3.3.4 Training and Validation Curves

During training, it is essential to monitor the model's performance on both the training and validation sets. The training curve represents the model's performance on the training data, while the validation curve represents its performance on the unseen validation data. By plotting the loss and accuracy curves for both the training and validation sets, we can gain insights into the model's learning behavior and identify potential issues such as overfitting or underfitting. Overfitting occurs when the model performs well on the training data but poorly on the validation data, indicating that it has memorized the training examples instead of learning generalizable patterns. Underfitting, on the other hand, occurs when the model performs poorly on both the training and validation data, suggesting that it lacks the capacity to capture the underlying patterns. To mitigate overfitting Early stopping is employed, where the training is halted if the validation loss stops improving for a specified number of epochs, preventing the model from overfitting to the training data.

3.4 Hyperparameter Tuning

Hyperparameters are the adjustable parameters of a model that are set prior to training and can significantly impact its performance. In CNNs, hyperparameters include the learning rate, batch size, number of filters, kernel size, dropout rate, and more. To find the optimal combination of hyperparameters, we employ hyperparameter tuning techniques. In this study, I utilized Bayesian optimization for hyperparameter tuning. Bayesian optimization is a sequential model-based optimization technique that intelligently explores the hyperparameter space by balancing exploration and exploitation. It constructs a probabilistic surrogate model (often a Gaussian process) to approximate the relationship between hyperparameters and the model's performance metric (e.g., validation accuracy). The Bayesian optimization process iteratively selects the next set of hyperparameters to evaluate based on an acquisition function, which quantifies the trade-off between exploring new regions and exploiting promising regions. The surrogate model is updated with the observed performance metric, and the process continues until a predefined number of iterations or a convergence criterion is met.

3.5 Model Evaluation and Interpretation

After training and tuning the CNN models, it is crucial to evaluate their performance on the test set, which consists of unseen data. This evaluation provides an unbiased estimate of how well the models generalize to new instances.

3.5.1 Confusion Matrix

The confusion matrix is a tabular summary of the model's predictions compared to the actual labels. It provides a detailed breakdown of the true positive (TP), true negative (TN), false positive (FP), and false negative (FN) predictions.

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

The confusion matrix allows us to calculate various performance metrics and gain insights into the model's strengths and weaknesses.

3.5.2 ROC Curve and AUC

The Receiver Operating Characteristic (ROC) curve is a graphical representation of the model's performance at different classification thresholds. It plots the true positive rate (TPR) against the false positive rate (FPR) as the threshold varies. The TPR (also known as sensitivity or recall) measures the proportion of actual positive samples that are correctly predicted as positive. It is calculated as:

$$TPR = \frac{TP}{TP + FN} \tag{3.2}$$

The FPR measures the proportion of actual negative samples that are incorrectly predicted as positive. It is calculated as:

$$FPR = \frac{FP}{FP + TN} \tag{3.3}$$

An ideal classifier would have a TPR of 1 and an FPR of 0, corresponding to the top-left corner of the ROC plot. A random classifier would have a diagonal line from the bottom-left to the top-right corner, indicating equal TPR and FPR at all thresholds. The Area Under the ROC Curve (AUC) is a single scalar value that summarizes the model's performance across all thresholds. An AUC of 1 represents a perfect classifier, while an AUC of 0.5 represents a random classifier. A higher AUC indicates better discriminative power of the model.

3.5.3 Cross-Validation

Cross-validation is a technique used to assess the model's performance and robustness by evaluating it on multiple subsets of the data. We used 5-fold cross-validation, where the data is divided into 5 equal-sized folds. In each iteration of k-fold cross-validation, one fold is used as the validation set, while the remaining k-1 folds are used for training. The model is trained and evaluated k times, with each fold serving as the validation set exactly once. The performance metrics are then averaged across all iterations to obtain a more reliable estimate of the model's performance. Cross-validation helps in assessing the model's ability to generalize and reduces the risk of overfitting to a specific train-test split.

Chapter 4

Results

In this chapter, I present the results obtained from the implementation of the methodology described in the previous chapter. I discuss the outcomes of data preprocessing, model architectures, hyperparameter tuning, and model evaluation.

4.1 Data Preprocessing Results

The dataset consisted of 3,199 images of chihuahuas and 2,718 images of muffins. Although there was a slight imbalance, with the chihuahua class having more samples, the difference was not considered significant enough to warrant specific balancing techniques. The dataset was organized into a structured dataframe, which was then split into three subsets: training set (75%), validation set (10%), and test set (20%). Image data generators were created using the Keras ImageDataGenerator class, with the following parameters:

- target size: (224, 224)
- batch size: 16
- class mode: "binary"
- color mode: "rgb"

The training and validation generators had the shuffle parameter set to True, while it was set to False for the test generator. Data augmentation techniques were applied to the training and validation images. The augmentation operations included random rotations, width and height shifts, stretching, zooming, channel shifting, and horizontal flipping. Figure 1 and Figure 2 showcase examples of the normalized and augmented training images, respectively.

4.2 Model Architectures and Results

I experimented with four different architectures to evaluate their performance.

Figure 4.1: Normalized training images

Figure 4.2: Normalized and augmented training images

4.2.1 First Model: Base Model

The base model consisted of the following layers:

- First convolutional layer:
  - 32 filters of size 3×3 with padding
  - ReLU activation function
- First pooling layer: Pooling size of 2×2 with a stride of 2
- Second convolutional layer: 64 filters of size 3×3 with padding
- Second pooling layer: Same dimensions as the first pooling layer
- Dense layer: 64 neurons with ReLU activation

- Final layer: 1 neuron with sigmoid activation for output

The base model was trained for 21 epochs using the normalized training images. Figure 1 and Figure 2 display the loss and accuracy curves for the base model.

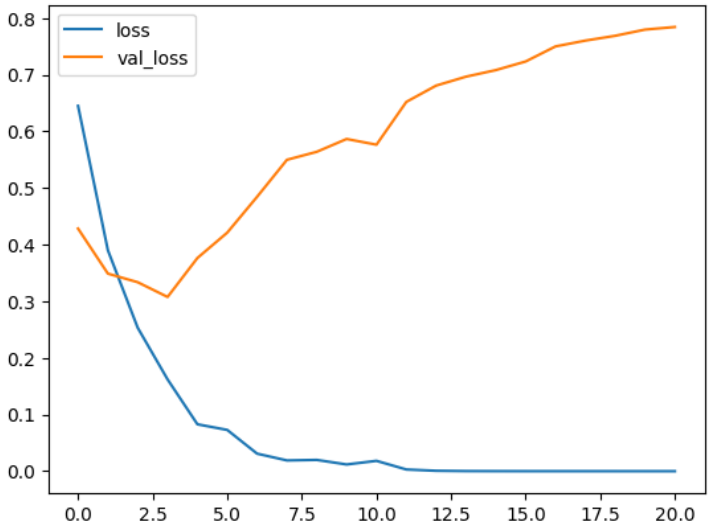

Figure 4.3: Loss and validation loss for Model 1

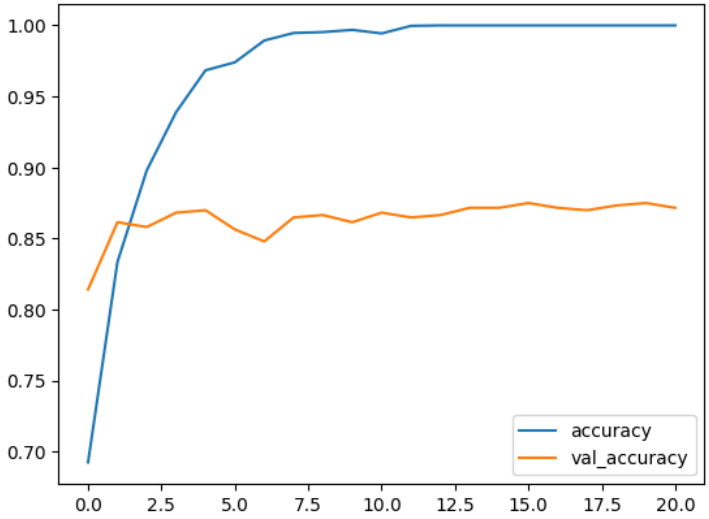

Figure 4.4: Accuracy and validation accuracy for Model 1

The results indicated that the base model suffered from overfitting, as evident from the increasing gap between the training and validation loss curves.

4.2.2 Model 2: Data Augmentation

To address the overfitting issue, I trained the second model using the augmented training images while keeping the architecture identical to the base model. The loss and accuracy curves for Model 2 are shown in Figure 1 and Figure 2.

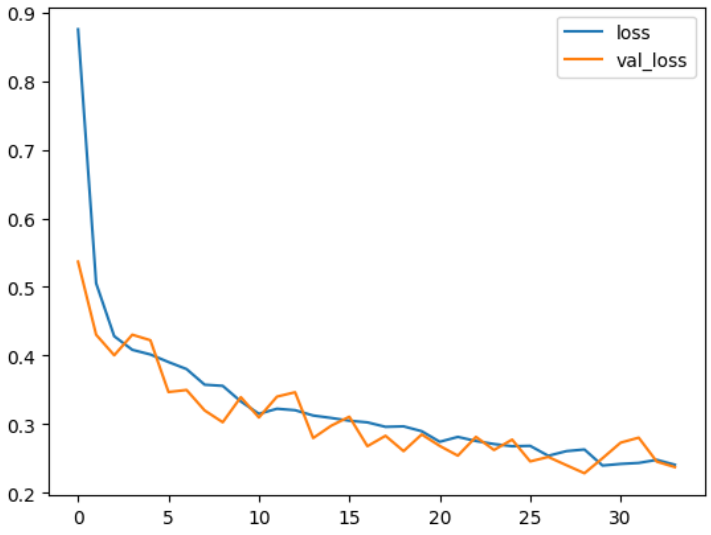

Figure 4.5: Loss and validation loss for Model 2

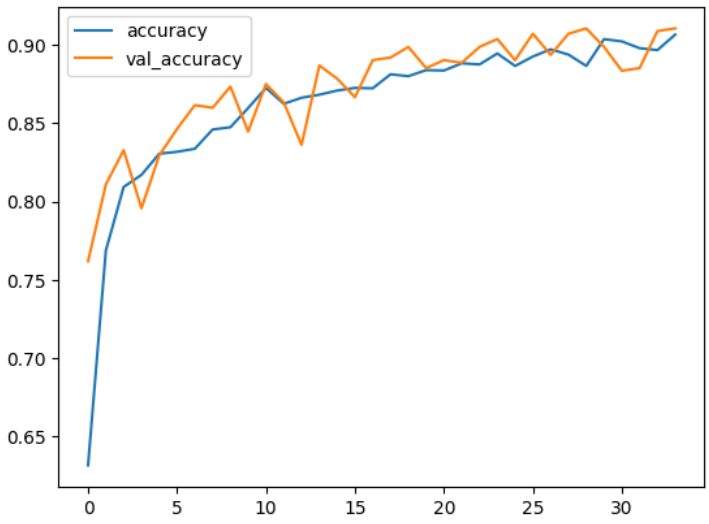

Figure 4.6: Accuracy and validation accuracy for Model 2

The results demonstrated a significant improvement in the model's performance. The training and validation loss curves were closer together, indicating a reduction in overfitting.

4.2.3 Model 3: Dropout Layers

In the third model, I introduced dropout layers to further mitigate overfitting. The model architecture remained the same as the first and second models, with the addition of dropout layers after each max-pooling layer (25% dropout) and before the final output layer (30% dropout). Figure 1 and Figure 2 present the loss and accuracy curves for Model 3.

Although Model 3 did not exhibit overfitting, its performance was slightly worse compared to Model 2 in terms of loss and accuracy.

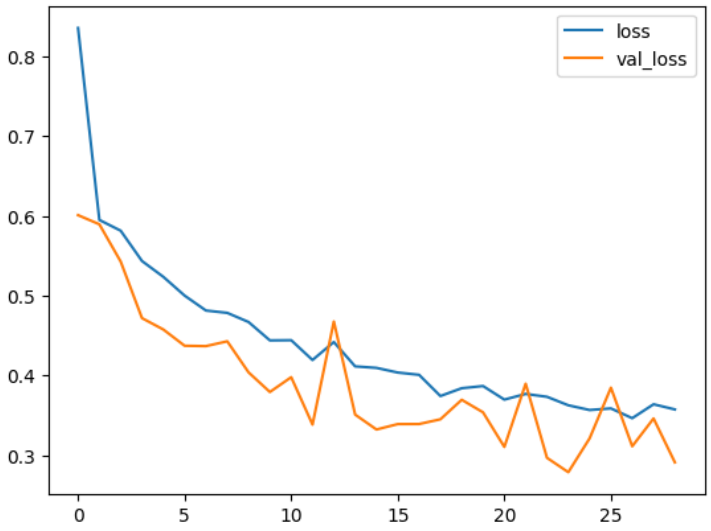

Figure 4.7: Loss and validation loss for Model 3

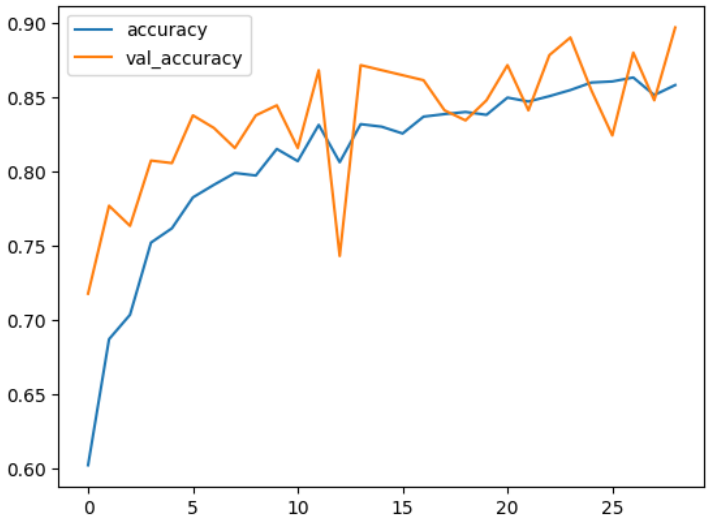

Figure 4.8: Accuracy and validation accuracy for Model 3

4.2.4 Model 4: Additional Convolutional Layer

The fourth model extended the architecture of the third model by adding an additional convolutional layer. The model architecture consisted of the following layers:

- First convolutional layer:
  - 32 filters of size 3×3 with padding
  - ReLU activation function
- First pooling layer: Pooling size of 2×2 with a stride of 2
- First dropout layer: 25% dropout
- Second convolutional layer: 64 filters of size 3×3 with padding
- Second pooling layer: Same dimensions as the first pooling layer

- Second dropout layer: 25% dropout
- Third convolutional layer: 128 filters of size 3×3 with padding
- Third pooling layer: Same dimensions as the previous pooling layers
- Third dropout layer: 25% dropout
- Dense layer: 64 neurons with ReLU activation
- Dense dropout layer: 30% dropout
- Final layer: 1 neuron with sigmoid activation for output

Figure 1 and Figure 2 display the loss and accuracy curves for Model 4.

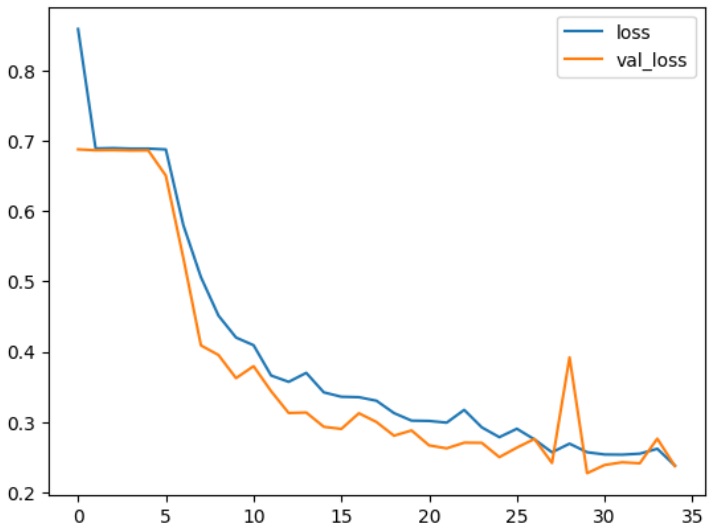

Figure 4.9: Loss and validation loss for Model 4

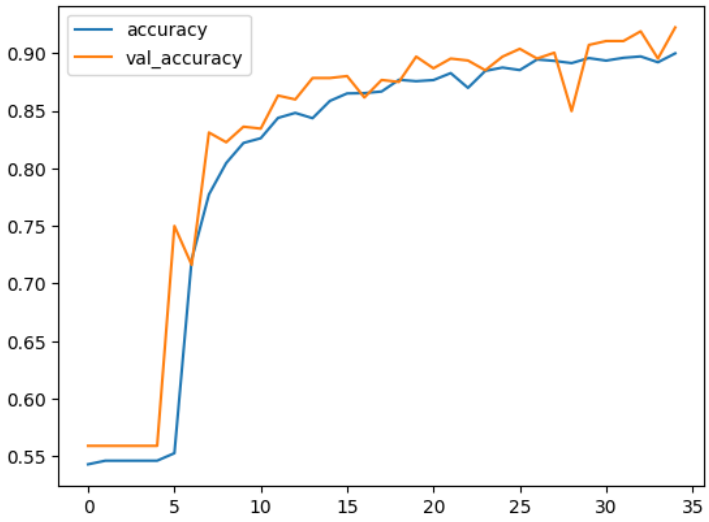

Figure 4.10: Accuracy and validation accuracy for Model 4

The results showed that Model 4 achieved a good fit, with the training and validation curves closely aligned. The additional convolutional layer and dropout regularization helped in improving the model's performance and generalization ability.

4.3 Hyperparameter Tuning Results

To further optimize the performance of the final model, I conducted hyperparameter tuning using the Bayesian optimization approach. The tuning process involved defining ranges and step sizes for various hyperparameters, such as the number of filters, dropout rates, and the number of neurons in the dense layer. After three iterations of the Bayesian optimization process, the following optimal hyperparameter values were obtained:

- First Convolutional Layer: Number of filters: 48
- First Dropout Layer: Proportion of dropped neurons: 0.05
- Second Convolutional Layer: Number of filters: 96
- Second Dropout Layer: Proportion of dropped neurons: 0.05
- Third Convolutional Layer: Number of filters: 192
- Third Dropout Layer: Proportion of dropped neurons: 0.05
- First Dense Layer: Number of neurons: 224
- First Dense Dropout Layer: Fraction of dropped neurons: 0.4

The tuned model's performance was evaluated, and the loss and accuracy curves are shown in Figure 4.11 and Figure 4.12.

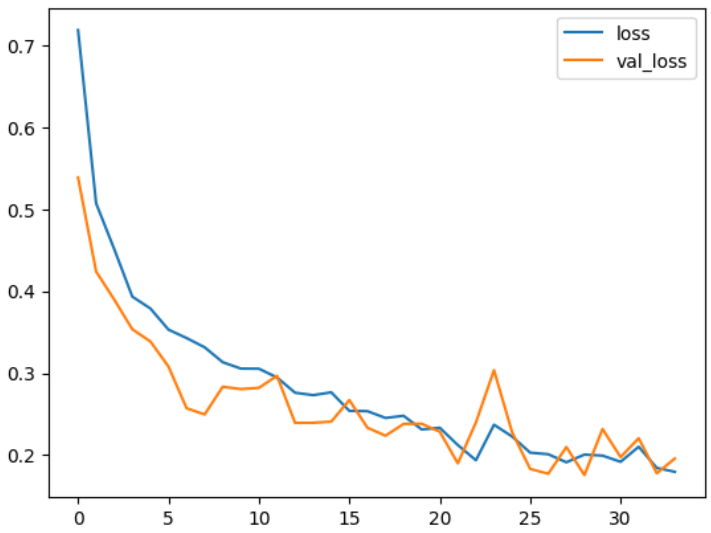

Figure 4.11: Loss and validation loss for the tuned model

The tuned model exhibited a good fit, with the training and validation curves closely aligned, indicating the effectiveness of the hyperparameter tuning process.

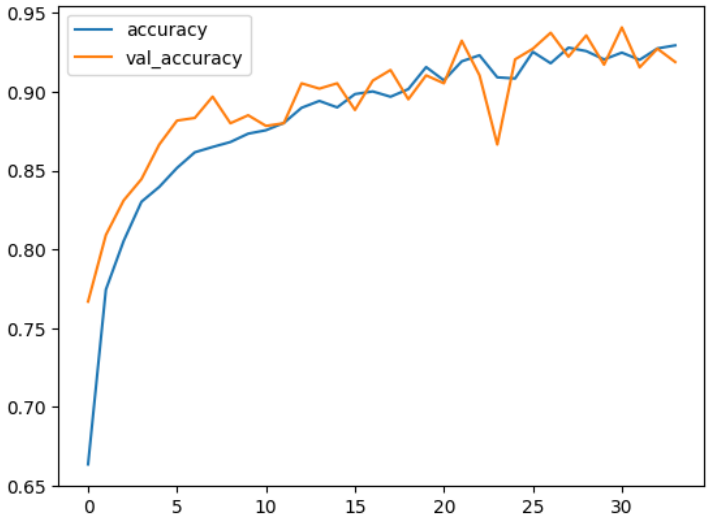

Figure 4.12: Accuracy and validation accuracy for the tuned model

4.4 Model Evaluation Results

To assess the performance of the tuned model on unseen data, I evaluated it using the test set. The trained model was used to predict the labels for the test images, and the predicted labels were compared with the actual labels to compute various evaluation metrics.

4.4.1 Confusion Matrix

The confusion matrix for the test set predictions is shown in Figure.

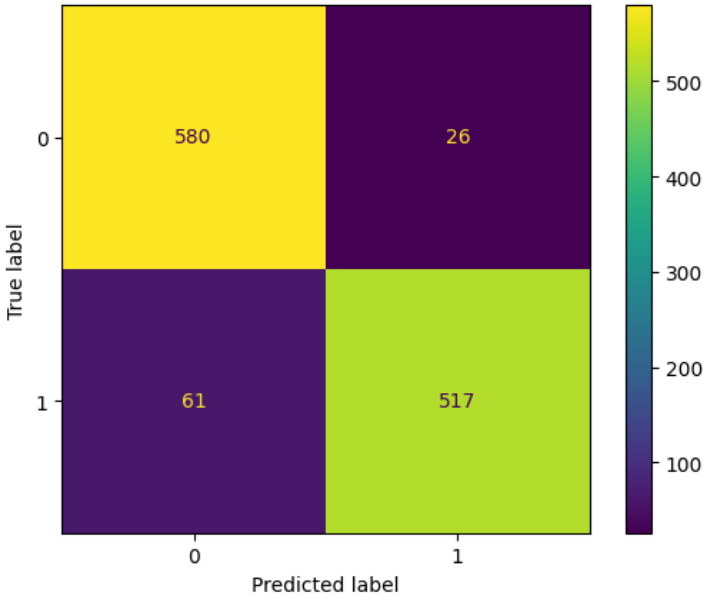

Figure 4.13: Confusion matrix for test set predictions

The confusion matrix provides a detailed breakdown of the model's predictions:

- True Positive (TP): 580 images of muffins correctly classified as muffins
- False Positive (FP): 61 images of chihuahuas misclassified as muffins
- True Negative (TN): 517 images of chihuahuas correctly classified as chihuahuas
- False Negative (FN): 26 images of muffins misclassified as chihuahuas

4.4.2 Classification Metrics

Table presents the classification metrics computed based on the confusion matrix. The

	Precision	Recall	F1-score	Support
Chihuahua	0.94	0.89	0.92	578
Muffin	0.90	0.96	0.93	606
Accuracy			0.93	1184

Table 4.1: Classification metrics

model achieved an overall accuracy of 0.93, indicating that it correctly classified 93% of the test images. The precision for the chihuahua class (0.94) was higher than that for the muffin class (0.90), suggesting that the model had a slight tendency to predict chihuahuas more accurately. On the other hand, the recall for the muffin class (0.96) was higher than that for the chihuahua class (0.89), indicating that the model had a slight tendency to misclassify chihuahuas as muffins.

4.4.3 ROC Curve and AUC

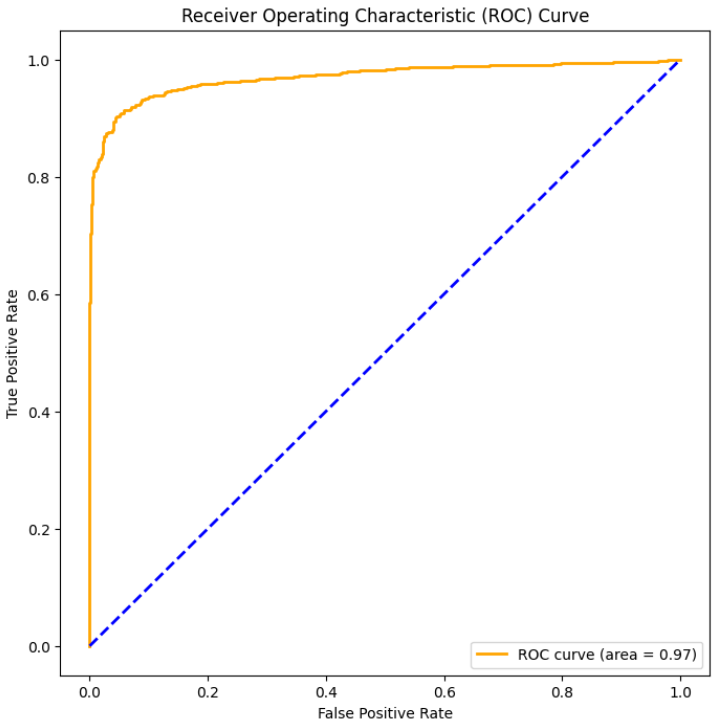

Figure 4.14: ROC curve for the tuned model

The Receiver Operating Characteristic (ROC) curve and the Area Under the Curve (AUC) were used to assess the model's ability to discriminate between classes across different probability thresholds. Figure 4.14 displays the ROC curve for the tuned model. The ROC curve for the tuned model approached the top-left corner, indicating excellent discrimination ability. The AUC value of 0.96 further confirmed the model's robustness in distinguishing between chihuahuas and muffins.

4.5 Cross-Validation Results

To assess the model's performance and robustness, I performed 5-fold cross-validation on the training set. Table presents the validation accuracy and validation loss for each fold, along with the average values.

Fold	Validation Accuracy	Validation Loss
1	0.9502	0.0498
2	0.8801	0.1199
3	0.9341	0.0659
4	0.5638	0.4362
5	0.9138	0.0862
Average	0.8484	0.1516

Table 4.2: 5-fold cross-validation results

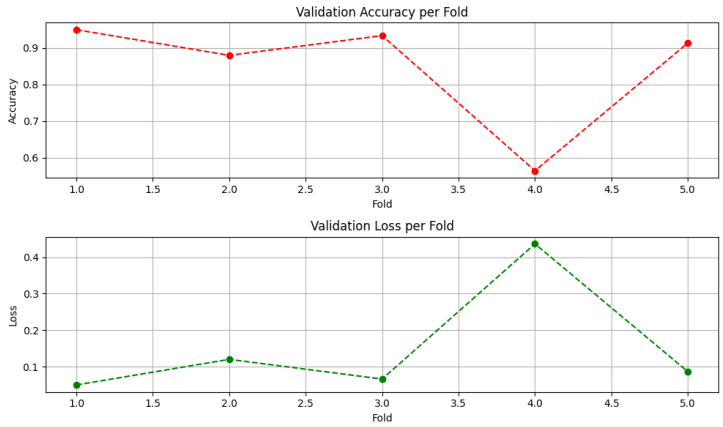

Figure 4.15: Validation Accuracy and Loss per Fold

The 5-fold cross-validation results demonstrate the model's overall good performance, with an average validation accuracy of 0.8484, indicating that the model correctly classified approximately 84.84% of the images in the validation sets across all folds. The average validation loss of 0.1516 suggests that the model's predictions are generally aligned with the true labels. However, it is important to acknowledge an oddity in the results, where fold 4 exhibits a significantly lower validation accuracy of 0.5638. This discrepancy raises concerns about the model's performance on the specific subset of data in fold 4. The low validation accuracy could be attributed to the fact that the data in fold 4 contains more challenging or diverse examples that the model struggles to classify accurately. This could be due to variations in image quality, object appearance, or the presence of outliers

that differ from the patterns learned from the other folds. To address this issue and improve the model's performance, collecting a larger and more diverse dataset could be a potential solution. Despite the lower performance in fold 4, the cross-validation results for the other folds show promising performance, with validation accuracies above 0.88. This suggests that the model is capable of learning meaningful features and patterns from the training data and generalizing effectively to unseen examples in most cases. The consistency in performance across the majority of the folds indicates that the model is not severely overfitting to specific subsets of the data and can handle variations in the input images. While the average validation accuracy of 0.8484 is lower than the accuracy achieved on the separate test set (0.93) it is still a succesful result.