# Image Classification: Muffins vs. Chihuahuas

Emre Kalkan

## Contents

## 1 Introduction

This report discusses an experiment focused on differentiating between images of muffins and chihuahuas using machine learning techniques. The following provides an outline of the methodology:

1. **Tools Utilized**: The Keras library was employed to construct the machine learning models, while Keras Tuner was used for fine-tuning.
2. **Data Processing**: Data augmentation techniques, such as rotation, zooming, and shifting, were applied to the images.
3. **Validation Technique**: Stratified K-Fold cross-validation was used to evaluate the models' robustness, involving training and testing on multiple, distinct sets of images.
4. **Model Variations**: Three distinct neural network architectures were explored to determine the optimal model for this task.
5. **Performance Metrics**: After training, the models were evaluated based on accuracy, AUC, and F1 Score. Additionally, the learning process was visualized to analyze the training progression.

## 2 Dataset Details

- **Total Images:** 6278
- **Training-Testing Split:** 80-20 (both in hyperparameter tuning phase and 5-fold cross validation phase)
- **Source of Images:** `https://www.kaggle.com/datasets/samuelcortinhas/muffin-vs-chihuahua-image-classification`
  (Dataset was in pristine condition and it didn't need to be cleaned and labeled)
- **Directory Structure:**

```
root/
|-- train/
|   |-- muffin/
|   |   |-- image1.jpg
|   |   |-- image2.jpg
|   |-- chihuahua/
|       |-- image1.jpg
|       |-- image2.jpg
|-- test/
    |-- muffin/
    |   |-- image1.jpg
    |   |-- image2.jpg
    |-- chihuahua/
        |-- image1.jpg
        |-- image2.jpg
```

## 3 Model Architectures Details

### 3.1 Simple Sequential Model

The simple sequential model begins with a flattening layer that converts the input image of shape (IMG_HEIGHT, IMG_WIDTH, 1) into a 1-dimensional array. This is followed by a dense (fully connected) layer with a number of units ranging from 192 to 256 (based on hyperparameter tuning) and uses the ReLU activation function. The final layer is another dense layer with 1 unit and uses the sigmoid activation function, which is suitable for binary classification tasks. The optimizer used is the Adam optimizer with variable learning rates, and the loss function is binary cross-entropy.

```python
from keras.optimizers.legacy import Adam
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

IMG_HEIGHT, IMG_WIDTH = 224, 224

def build_simple_sequential_model(hp):
    """
    This function returns a simple sequential model for hyperparameter tuning.
    Args:
        hp (keras_tuner.HyperParameters): Hyperparameters to tune.

    Returns:
        keras.Model: The compiled model.
    """
    model1 = Sequential()
    model1.add(Flatten(input_shape=(IMG_HEIGHT, IMG_WIDTH, 1)))
    model1.add(Dense(hp.Int('units', min_value=192, max_value=256, step=32), activation='relu'))
    model1.add(Dense(1, activation='sigmoid'))
    model1.compile(loss='binary_crossentropy', optimizer=Adam(hp.Choice('learning_rate', values=[1e-3, 5e-4, 1e-4])), metrics=['accuracy'])
    return model1
```

Listing 1: Simple Sequential Model

### 3.2 Simple CNN Model

The simple CNN model starts with a convolutional layer with filters ranging from 192 to 256, a kernel size of 3x3, ReLU activation function, and an input shape of (IMG_HEIGHT, IMG_WIDTH, 1). This is followed by a max-pooling layer with a 2x2 pool size. Another convolutional layer is added next, similar to the first, but without specifying the input shape. This is followed again by a 2x2 max-pooling layer. After these layers, the model uses a flattening layer, followed by a dense layer with units ranging from 128 to 192 and a ReLU activation function. The model concludes with a dense layer with 1 unit and a sigmoid activation function. Again, the Adam optimizer with variable learning rates is used, paired with a binary cross-entropy loss function.

```python

from keras.optimizers.legacy import Adam
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Set constants for reproducibility and ease of use
IMG_HEIGHT, IMG_WIDTH = 224, 224


def build_simple_cnn_model(hp):
    """
    This function returns a simple CNN model for hyperparameter tuning.

    Args:
        hp (keras_tuner.HyperParameters): Hyperparameters to tune.

    Returns:
        keras.Model: The compiled model.
    """
    model2 = Sequential()
    model2.add(Conv2D(hp.Int('input_units', min_value=192, max_value
   =256, step=32), (3, 3), activation='relu', input_shape=(IMG_HEIGHT,
   IMG_WIDTH, 1)))
    model2.add(MaxPooling2D((2, 2)))
    model2.add(Conv2D(hp.Int('hidden_units', min_value=192, max_value
   =256, step=32), (3, 3), activation='relu'))
    model2.add(MaxPooling2D((2, 2)))
    model2.add(Flatten())
    model2.add(Dense(hp.Int('dense_units', min_value=128, max_value
   =192, step=32), activation='relu'))
    model2.add(Dense(1, activation='sigmoid'))
    model2.compile(loss='binary_crossentropy', optimizer=Adam(hp.Choice
   ('learning_rate', values=[1e-3, 1e-4])), metrics=['accuracy'])
    return model2
```
Listing 2: Simple CNN Model

## 3.3 Complex CNN Model

The complex CNN model is an extension of the simple CNN model. It starts with a convolutional layer similar to the simple CNN model, followed by a 2x2 max-pooling layer. This pattern is repeated three more times, meaning there are four convolutional layers and four max-pooling layers in total. The number of filters in these convolutional layers can range from 128 to 256. After these convolutional operations, the model flattens the data and feeds it into a dense layer with units ranging between 128 and 256, using the ReLU activation function. The final layer is a dense layer with 1 unit and a sigmoid activation function for binary classification. The model is compiled using the Adam optimizer with variable learning rates and the binary cross-entropy loss function. The code does not specify the use of dropout or batch normalization layers in this model.

```python
from keras.optimizers.legacy import Adam
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

IMG_HEIGHT, IMG_WIDTH = 224, 224

def build_complex_cnn_model(hp):
    """
    This function returns a complex CNN model for hyperparameter tuning
    .
    Args:
        hp (keras_tuner.HyperParameters): Hyperparameters to tune.
    Returns:
        keras.Model: The compiled model.
    """
    model3 = Sequential()
    model3.add(Conv2D(hp.Int('input_units', min_value=128, max_value
    =256, step=32), (3, 3), activation='relu', input_shape=(IMG_HEIGHT,
    IMG_WIDTH, 1)))
    model3.add(MaxPooling2D((2, 2)))
    model3.add(Conv2D(hp.Int('hidden_units_1', min_value=128, max_value
    =256, step=32), (3, 3), activation='relu'))
    model3.add(MaxPooling2D((2, 2)))
    model3.add(Conv2D(hp.Int('hidden_units_2', min_value=128, max_value
    =256, step=32), (3, 3), activation='relu'))
    model3.add(MaxPooling2D((2, 2)))
    model3.add(Conv2D(hp.Int('hidden_units_3', min_value=128, max_value
    =256, step=32), (3, 3), activation='relu'))
    model3.add(MaxPooling2D((2, 2)))
    model3.add(Flatten())
    model3.add(Dense(hp.Int('dense_units', min_value=128, max_value
    =256, step=32), activation='relu'))
    model3.add(Dense(1, activation='sigmoid'))
    model3.compile(loss='binary_crossentropy', optimizer=Adam(hp.Choice
    ('learning_rate', values=[1e-3, 1e-4])), metrics=['accuracy'])
    return model3
```

Listing 3: Complex CNN Model

## 4 Evaluation Metrics Discussion

### 4.1 Accuracy

In the realm of classification models, one of the main evaluation metrics is accuracy. This metric offers an intuitive grasp of the frequency with which a model's predictions align with the actual outcomes [2]. Accuracy is equal to the proportion of accurate predictions relative to the entire set of predictions made:

$$\text{Accuracy} = \frac{\text{Correctly Identified Instances}}{\text{Total Predicted Instances}}$$

However, it's essential to note that while accuracy can offer a general understanding of model performance, it can sometimes be misleading. That's why we need other evaluation metrics.

### 4.2 AUC (Area Under the Curve)

The ROC (Receiver Operating Characteristic) Curve's AUC (Area Under the Curve) is a crucial metric in binary classification tasks. It graphically illustrates a classifier's ability over different decision thresholds by contrasting the True Positive Rate (TPR) with the False Positive Rate (FPR).

The essence of AUC lies in its singular numerical value, highlighting the model's adeptness at distinguishing between positive and negative outcomes. A perfect model achieves an AUC score of 1. On the contrary, a model making random guesses would result in an AUC of 0.5. One of AUC's primary strengths is its resistance to the effects of data imbalance, making it a preferable metric compared to straightforward accuracy in binary classification scenarios.

### 4.3 The F1 Score

An indispensable metric in the classification domain is the F1 score, which encapsulates the balanced harmony between precision and recall [1]. Precision, in this context, quantifies the proportion of true positive predictions among all positive predictions. On the other hand, recall, often labeled as sensitivity, gauges the fraction of true positive predictions in relation to the entirety of the actual positive class.

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

In datasets exhibiting class imbalance, where one class vastly dominates the other, the traditional accuracy measure might not capture the true model efficacy. Herein lies the importance of the F1 score, which offers a holistic view by considering both false positives and false negatives. Ideally, an F1 Measure of 1 denotes impeccable precision and recall, whereas its least desirable value is 0.

## 5 Challenges and Solutions

### 5.1 Hardware Limitations

The most pronounced challenge was the hardware limitations. Our infrastructure posed constraints on the computational resources available, impacting several facets of our modeling phase:

- **Nested Cross-Validation**: We initially intended to use nested cross-validation to ensure the model's generalization to new data. However, nested cross-validation requires significant computational power and time, as it involves multiple layers of training and validation loops. Due to our hardware limitations, we couldn't implement this approach.

- **Training Epochs**: A higher number of epochs usually allows the model to learn and adapt better to the data, but it comes at the cost of increased computational time. Given our constraints, we had to strike a balance by selecting a limited number of epochs that offered reasonable performance without overburdening our hardware.

- **Tuning Trials**: Ideally, extensive hyperparameter tuning should be performed to ensure the best model configuration. This requires multiple trials, each with a different set of hyperparameters. However, our hardware constraints limited the number of tuning trials we could feasibly run, compelling us to prioritize and choose a select few promising configurations.

### 5.2 Solutions

- **Grayscale Images**: To further alleviate the computational burden, we converted all input images to grayscale. By reducing the three RGB channels to a single channel, we significantly decreased the data's size. This not only minimized the memory footprint but also reduced the training time.
- **Cloud-based Platform**: To partially overcome these hardware limitations, we leveraged Google Colab, a cloud-based platform that offers free GPU access, to run some of our computations. This strategy allowed us to handle more complex computations.

## 6 Preprocessing Phase

### 6.1 Image Loading and Resizing

The images are loaded from specified directories, resized to a standard dimension of 224x224 pixels, and converted to grayscale. The grayscale conversion simplifies the dataset and reduces computational demand. Every image is normalized by dividing each pixel value by 255. This ensures all pixel values lie between 0 and 1, aiding in stabilizing and speeding up the convergence during training.

```python
def load_data(path):
    images = []
    labels = []
    for first_folder_name in ['train', 'test']:
        first_folder_path = os.path.join(path, first_folder_name)
        for second_folder_name in ['muffin', 'chihuahua']:
            second_folder_path = os.path.join(first_folder_path, second_folder_name)
            for img_name in os.listdir(second_folder_path):
                img_path = os.path.join(second_folder_path, img_name)
                img = load_img(img_path, target_size=(IMG_HEIGHT, IMG_WIDTH), color_mode='grayscale')
                img_array = img_to_array(img) / 255.
                images.append(img_array)
                labels.append(1 if second_folder_name == 'muffin' else 0)
    return np.array(images), np.array(labels)

# Load all data
X, y = load_data('../MLproject/archive')
```

Listing 4: Simple Sequential Model

## 7 Data Augmentation

To increase the robustness of the model and prevent overfitting, various data augmentation techniques were applied:

- **Rotation:** Images were randomly rotated within a range of 0 to 20 degrees.
- **Shift:** Random horizontal and vertical shifts, within 20% of the image width and height.
- **Shear:** Random shearing with a shear angle up to 20 degrees.
- **Zoom:** Random zooming of images within a range of 20%.
- **Horizontal Flip:** Images were randomly flipped horizontally.
- **Fill Mode:** Points outside the boundaries are filled based on the nearest boundary values.

```
# Create a ImageDataGenerator instance with augmentation options.
datagen = ImageDataGenerator(
    rotation_range=20,  # Randomly rotate images in the range (degrees, 0 to 180)
    width_shift_range=0.2,  # Randomly shift images horizontally (fraction of total width)
    height_shift_range=0.2,  # Randomly shift images vertically (fraction of total height)
    shear_range=0.2,  # Set range for random shear
    zoom_range=0.2,  # Set range for random zoom
    horizontal_flip=True,  # Randomly flip inputs horizontally
    fill_mode='nearest'  # Points outside the boundaries of the input are filled according to the given mode
)
```

Listing 5: Simple Sequential Model

## 8 K-Fold Cross Validation

Stratified 5-fold cross-validation was implemented. Stratification ensures that each fold has a good representation of the entire dataset, maintaining the ratio of muffins to chihuahuas. This approach:

- Reduces Overfitting: Using different training-test splits helps in ensuring the model doesn't overfit to a specific set.
- Enhances Reliability: The model's performance is evaluated multiple times on different data, ensuring the results are reliable.

## 9 Model Building and Hyperparameter-Tuning

Three model architectures were considered:

1. Simple Sequential Model: A straightforward neural network with dense layers.

2. Simple CNN Model: A basic convolutional model with a couple of convolution and pooling layers.
3. Complex CNN Model: A more intricate convolutional model with multiple convolution and pooling layers.

### 9.1 Hyperparameter Tuning

For each architecture, hyperparameters (number of units in layers and learning rate) were tuned using `RandomSearch` from Keras Tuner:

- **Search Space:** Defined the range and choices for hyperparameters.
- **Objective:** Minimize validation loss.
- **Trials:** Total of 5 trials for each architecture.

## 10 Results and Interpretation

### 10.1 Performance Metrics

| Model | Accuracy | AUC | F1 Score |
|---|---|---|---|
| build_simple_sequential_model | $0.64 \pm 0.06$ | $0.72 \pm 0.02$ | $0.53 \pm 0.27$ |
| build_simple_cnn_model | $0.88 \pm 0.02$ | $0.95 \pm 0.02$ | $0.88 \pm 0.02$ |
| build_complex_cnn_model | $0.93 \pm 0.03$ | $0.98 \pm 0.02$ | $0.93 \pm 0.03$ |

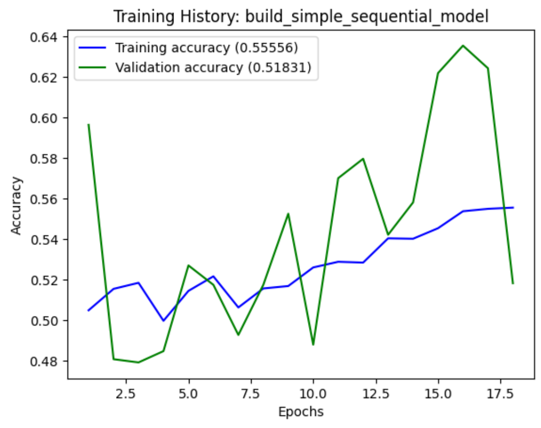
Table 1: Performance metrics for the three models.

- The `build_simple_sequential_model` performs the poorest among the three in terms of all metrics: accuracy, AUC, and F1 score. The high variance in F1 score also indicates potential overfitting or inconsistency in model performance.
- The `build_simple_cnn_model` shows a marked improvement from the sequential model. It has high accuracy, AUC, and F1 score with low variances, indicating consistent performance.
- The `build_complex_cnn_model` performs the best with the highest values in all metrics and low variances, suggesting it's not only the most accurate but also the most consistent of the three.

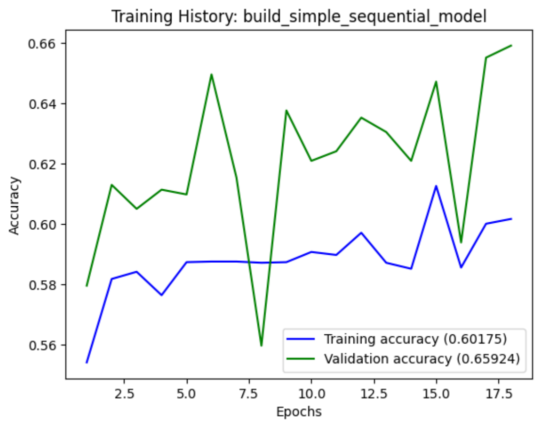
Based on these results, the `build_complex_cnn_model` is the best choice as it has the highest accuracy, AUC, and F1 score. It demonstrates the highest ability to correctly classify samples, distinguish between the classes, and maintain a balance between precision and recall. The added complexity of this model seems to be beneficial for this particular classification task. However, in practice, one should also consider computational costs, the possibility of overfitting with more complex models, and the interpretability of the model when making a final decision.

## 10.2 Training History Visualization

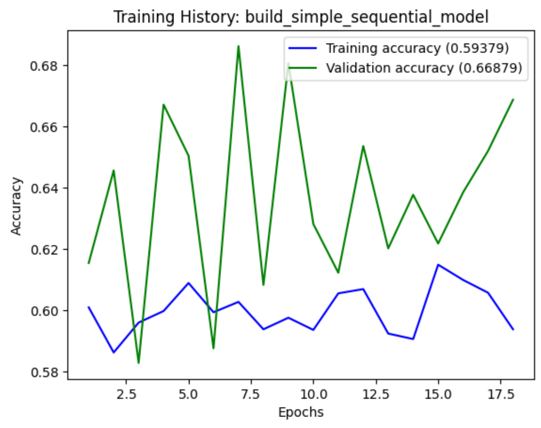
For each of our three model architectures, we provide the training history over various cross-validation iterations. The y-axis represents accuracy, while the x-axis represents the epoch count.



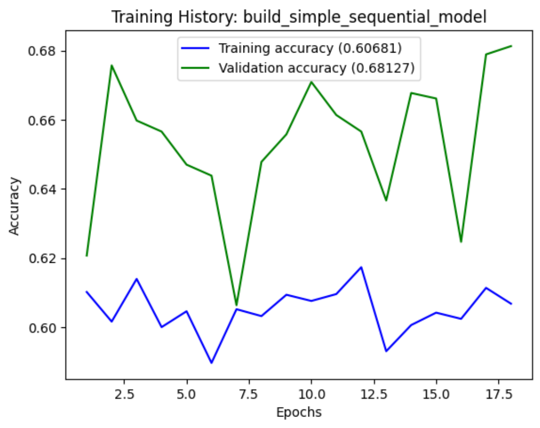
(a) Simple Sequential Model, Iteration 1



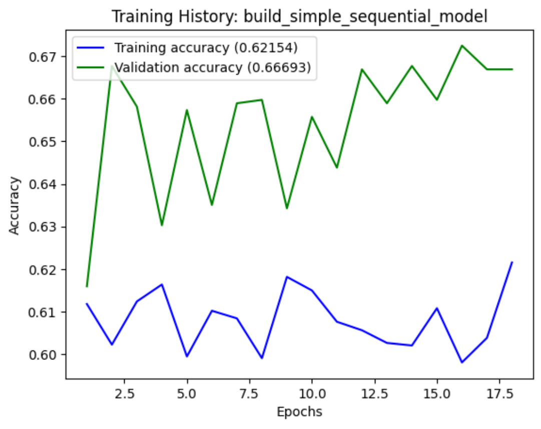
(b) Simple Sequential Model, Iteration 2



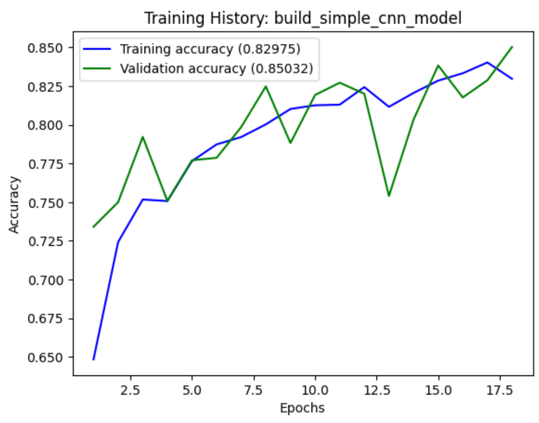
(c) Simple Sequential Model, Iteration 3



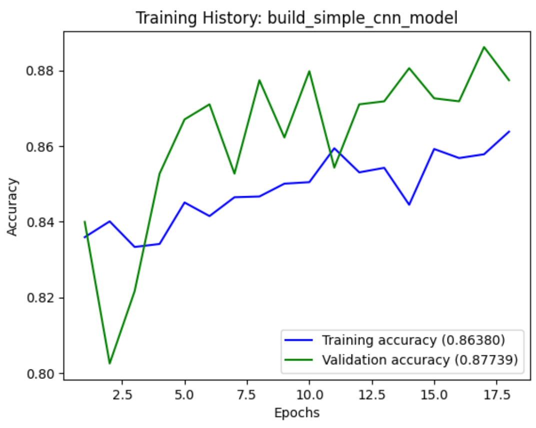
(d) Simple Sequential Model, Iteration 4



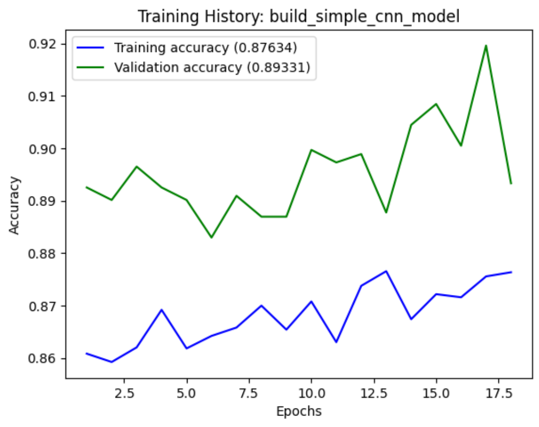
(e) Simple Sequential Model, Iteration 5



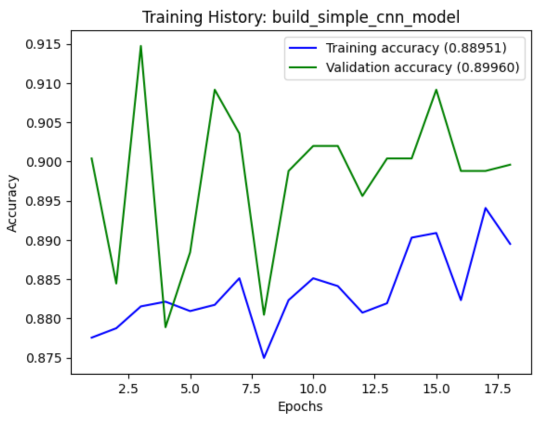
(a) Simple CNN Model, Iteration 1



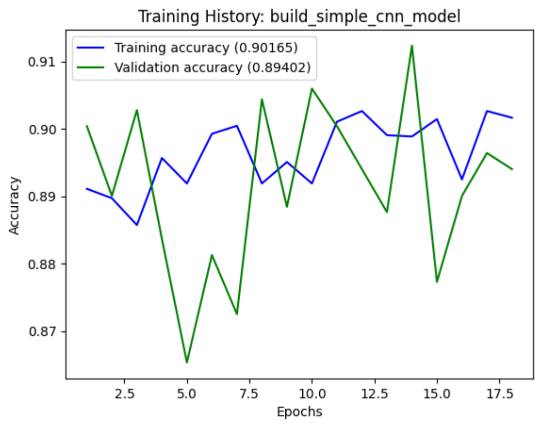
(b) Simple CNN Model, Iteration 2



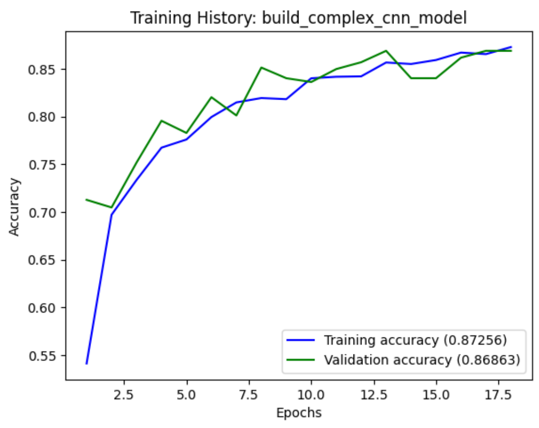
(c) Simple CNN Model, Iteration 3



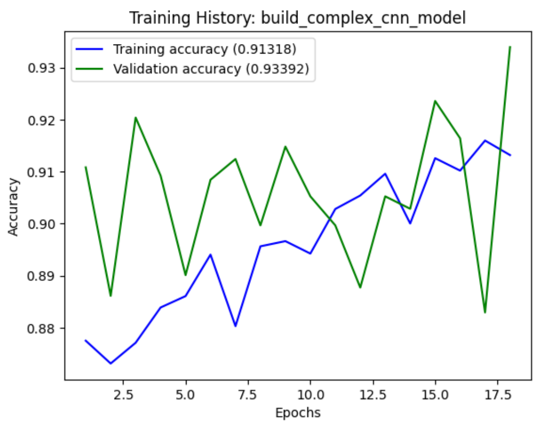
(d) Simple CNN Model, Iteration 4



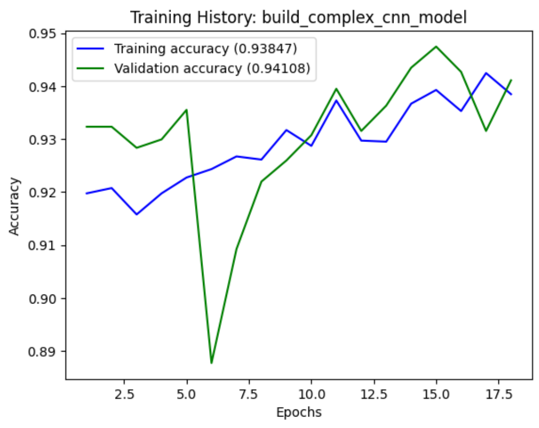
(e) Simple CNN Model, Iteration 5



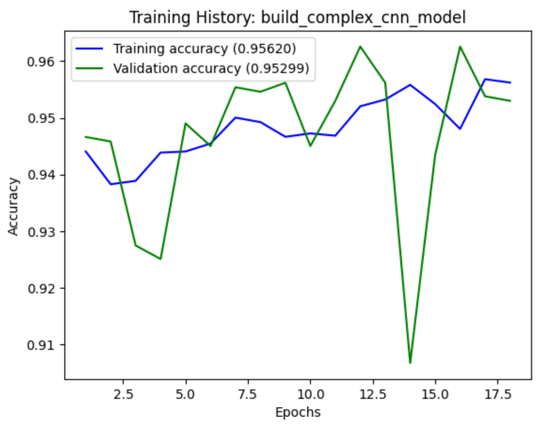
(a) Complex CNN Model, Iteration 1



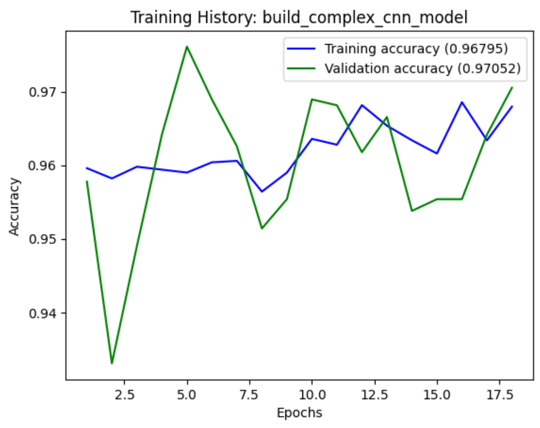
(b) Complex CNN Model, Iteration 2



(c) Complex CNN Model, Iteration 3



(d) Complex CNN Model, Iteration 4



(e) Complex CNN Model, Iteration 5

## 10.3 Interpretation of Training History Plots

Upon inspection of our model's training history plots, several key observations and interpretations can be made:

1. **Validation Accuracy Surpassing Training Accuracy:** Analyzing our models' training plots, we noticed that the validation accuracy often exceeded the training accuracy. This likely happens because we only used data augmentation on the training set. Augmentation changes and complicates the training data, making it harder for the model to achieve high accuracy during training. However, the untouched

validation set is easier for the model to predict, resulting in higher accuracy. This trend shows our model's strength in handling both augmented and original data.

2. **Complexity and Accuracy:** As the complexity of the model increases, both the validation and training accuracies tend to improve. This suggests that a more sophisticated model can potentially capture intricate patterns and relationships in the data which simpler models might overlook.

3. **Epochs and Accuracy Convergence:** With an increasing number of epochs, the gap between validation and training accuracy diminishes. This convergence can be attributed to the model becoming more familiar with the training data, thus improving its performance over time. However, one must be cautious, as too many epochs could lead to overfitting, where the model might perform exceptionally well on the training data but poorly on unseen or new data.

4. **General Interpretations:** Typically, training plots provide insights into the model's learning progression. Sharp fluctuations in accuracy or loss might indicate a learning rate that's too high, while plateaus could suggest that the learning rate is too low or that the model has reached its performance limit for the given architecture and data.

## 11 Conclusion

In our quest to differentiate muffins from chihuahuas, we explored various machine learning designs. The complex convolutional neural network clearly stood out, excelling in accuracy, AUC, and F1 score for our dataset.

Data augmentation, despite complicating training, boosted the model's adaptability, reflected in superior validation results. As we used more sophisticated models, they detected finer image details better. The reducing gap between training and validation accuracies with more epochs highlighted the model's learning prowess.

Yet, challenges like potential overfitting with complex models and computing limits reminded us to maintain balance in our approach.

In essence, our study underscores the significant role of model design in achieving desired outcomes, emphasizing machine learning's potential in complex image tasks.

## Declaration

## References

[1] Jesse Davis and Mark Goadrich. "The Relationship Between Precision-Recall and ROC Curves". In: *Proceedings of the 23rd International Conference on Machine Learning.* 2006, pp. 233–240.

[2] John D. Kelleher, Brian Mac Namee, and Aoife D'Arcy. *Fundamentals of Machine Learning for Predictive Data Analytics: Algorithms, Worked Examples, and Case Studies.* MIT Press, 2015.