Machine Learning - Image Classification with CNNs: Muffins vs Chihuahua

Doina Vasilev - n. 00080A

January 2024

1 Introduction

The following report contains an extensive explanation of the experimental project for the Machine Learning course. The purpose of my project is to implement several CNN architectures that can accurately classify images in 2 categories: *muffins* and *chihuahuas*. In the first section I'll be briefly describing the theoretical framework around Convolutional Neural Networks and their inner workings, also providing a brief explenation of the parameters. In the following section I'll be describing the dataset and the pre-processing steps I've undertaken for the purpose of the project. Finally, I'll be guiding you through the models I've implemented, and their results.

2 Convolutional Neural Networks

Convolutional neural network (CNN) is a regularized type of feed-forward neural network that learns feature engineering by itself via *filters optimization*. Its scope is to extract features from the grid-like matrix datasets, such as images, to classify them.[1]

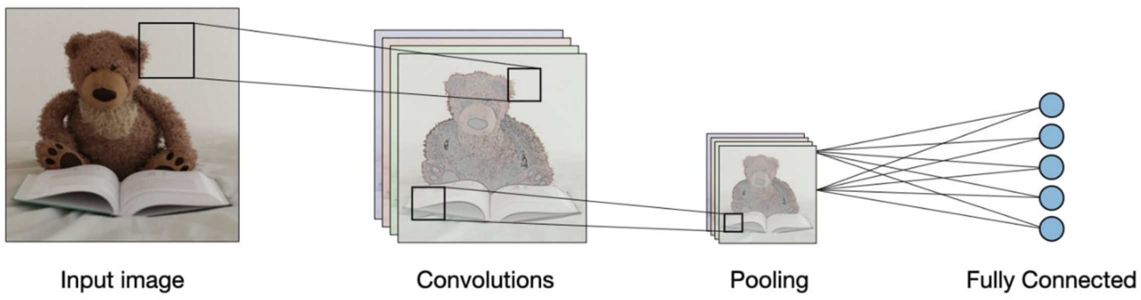


Figure 1: Representation of CNN layers (Source: Stanford University, CS 230 Convolutional Neural Networks)

The structure of a CNN typically comprises the following layers:

1. The **Input Layer**, which accepts the raw pixel data of images, provided in a 3-dimensional matrix format.

2. The **Convolutional Layers** are the core building blocks of a CNN. They are responsible for scanning through the input images with a series of learnable filters, or kernels, in order to identify relevant features such as edges in lower layers or complex patterns in higher layers. These filters move across the image, performing element-wise multiplication with the corresponding image patch, and store the results into a feature map. Important features are then 'activated' by passing each feature map value through a non-linear **activation function** such as the Rectified Linear Unit (**ReLU**) or Tanh. These activation functions help to capture the inherent non-linearities in image data.

3. The **Pooling Layer** follows the convolutional layer and simplifies the output by performing down-sampling, i.e., by reducing the dimensionality of the feature maps but retain the most important information. Common types of pooling include max pooling, which takes the maximum element from the feature map, and average pooling, which computes the average of the elements. This step significantly reduces computational complexity and the chances of overfitting.
   Up to now, these steps are essential for *feature learning*. By stacking by layer and layer, the network learns the features in higher level of precision each time. This is the feature learning part of the CNN.

[1] Wikipedia, *Convolutional Neural Networks.*

4. Finally, the **Fully-Connected Layers** take part in the *classification* phase of the CNN. At this stage, the high-level features extracted by the convolution and pooling layers are flattened into a one-dimensional vector. This vector is then processed through one or more dense layers which ultimately lead to a softmax function. The softmax function outputs a probability distribution over the target classes, offering a finalized prediction as output.

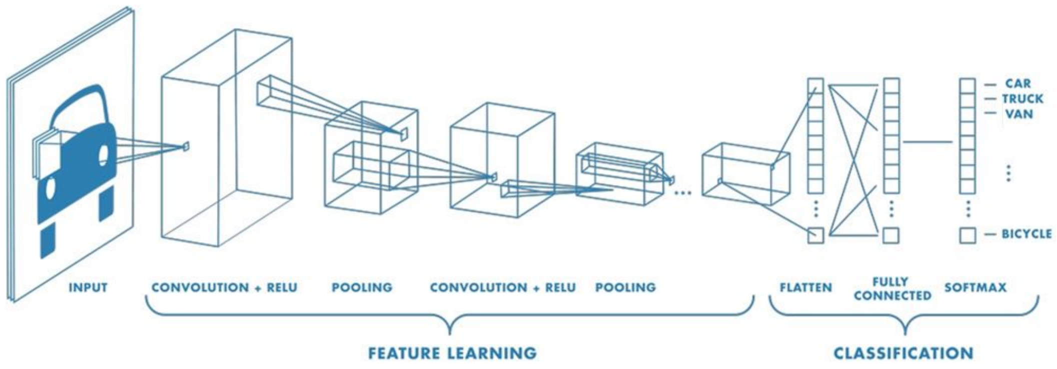


Figure 2: Steps of a CNN, split into **Feature Learning** and **Classification**.

2.1 Hyperparameters of CNNs

Hyperparameters in CNNs are critical settings that influence the network's architecture and the training process. Tuning these hyperparameters is often key to achieving optimal performance. Some fundamental hyperparameters include:

- **Number of Filters (Kernels)**: Dictates the number of features a convolutional layer can extract. More filters can capture more complex features but increase computational complexity.

- **Filter Size**: The dimensions of the filters/kernels used in convolutional layers, typically 3x3 or 5x5. Smaller filters may capture fine details, while larger filters may capture more complete features.

- **Stride**: The step size with which a filter moves across the input. Larger strides result in smaller feature maps.

- **Padding**: Refers to the addition of pixels around the edge of the input image. Padding can be used to control the spatial size of the feature maps and to allow the filter to access the border pixels of the input.

- **Pooling Size**: Defines the dimension over which the pooling operation is applied. It impacts the downsampling factor and thus the spatial size of the resulting feature map.

- **Batch Size**: The number of training examples utilized in one iteration of model training. Larger batch sizes can lead to faster training but require more memory.

- **Number of Epochs**: The number of times the entire training dataset is passed forward and backward through the neural network. More epochs lead to more training, but also raise the risk of overfitting if not monitored properly.

- **Learning Rate**: A crucial parameter that affects how much the model weights get updated during training. A too-high learning rate may lead to missing the optimum, while a too-low learning rate can result in a prolonged training process.

Constructing the architecture of a CNN requires experimenting with different number of layers, with the size of filters in convolutional layers, the size of the pool in pooling layers, the number of neurons in fully connected layers, and more.
With each iteration and layer, the CNN becomes capable of capturing more abstract and detailed features from the images, and, although the more information the better, we must also ensure that the model does not overfit. To do so, we may also incorporate regularization techniques such as **dropout**, weight decay, **data augmentation** or **batch normalization** techniques.

3 Data and data pre-preprocessing

The dataset was obtained from Kaggle at the following link: Muffin vs chihuahua.
The initial step in preprocessing was to partition the dataset into three distinct subsets: the training set, validation set, and test set—each serving a specific role in the model development and evaluation process. Following this, I used the *removal_corrupted_files* function, passing the *category* parameter to iterate over the directories and eliminate any images that were either corrupted or had a file size of zero bytes. Fortunately, the dataset revealed no such defective files, allowing me to use the entire original image collection for subsequent processing steps. In terms of dataset composition, there are a total of 3,199 images of chihuahuas and 2,718 images featuring muffins. To facilitate effective model training and unbiased evaluation, these images were distributed across the training, validation, and test sets in an 80:10:10 ratio, respectively. This distribution ensures a sufficient amount of data for the model to learn from, while also providing adequate datasets to fine-tune model hyperparameters and to test the model's ability to generalize to new, unseen data.

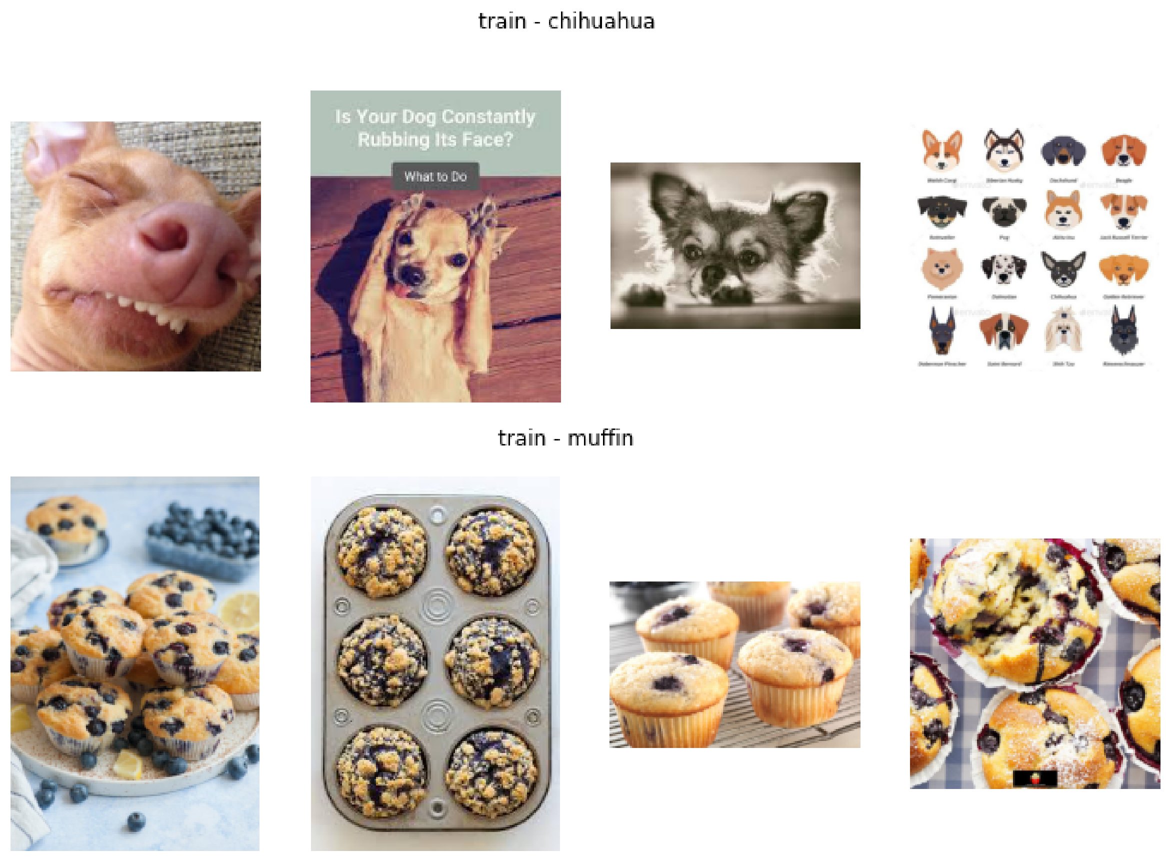


Figure 3: A random selection of images from the *training set*

.

Finally, the *preprocessing_step* standardizes the images by resizing them to a consistent shape of (90, 90) and ensures that they are all formatted in the RGB color space, which consists of the three color channels—red, green, and blue. Consequently, each image is represented as an array of shape (90, 90, 3), where '3' denotes the three distinct color channels. Each pixel within an RGB image is described by three numeric values, ranging from 0 to 255, which encode the intensity levels of the respective color channels—0 indicates no intensity, while 255 signifies maximum intensity. The preprocessing includes a normalization step, where all pixel values are scaled down to the range [0, 1] by dividing them by 255. Normalizing the image data is an essential practice in machine learning because it brings the input features to a common scale. This practice enhances the efficiency of the optimization algorithms used during training by facilitating a more consistent and faster convergence of the models.

4 Models and Experimental Results

In this section, I'll describe the various network architectures that I've implemented, along with the outcomes of each. To start, I experimented with two simple baseline models, Models 1 and 2, to get a feel for how deepening the architecture and adding dropout layers can help control overfitting. With some insights from these initial models, I moved on to include a data augmentation layer in Models 3, 3.1, and 4, aiming to make the models more robust by exposing them to a more varied set of training data. I also added more layers to these models to increase their complexity. The final step was to find the best performing model through hyperparameter tuning. I ran 20 rounds of this, with each round consisting of 50 epochs. For building all these models, I used the **Sequential()** function from Keras TensorFlow, which allows you to stack layers easily, one after the other. This approach helped me to systematically improve the model structures and achieve better results.

4.1 Baseline Models - Model 1

My first model, referred to as Model 1, includes a minimal architecture comprising two convolutional layers with 32 and 64 filters respectively, followed by a dense layer with 32 nodes. Each layer is regularized using a dropout layer with a rate of 0.5. This model serves as the baseline for my experiments.

Architecture

- Convolutional layer with 32 filters, kernel size (3, 3), relu activation.
- MaxPooling layer with (2, 2) pool size.
- Dropout layer with 0.5 rate.
- Flatten layer.
- Dense layer with 32 nodes, relu activation.
- Dropout layer with 0.5 rate.
- Output layer with sigmoid activation.

4.1.1 Training

The training phase involved compiling the model with the **Adam** optimizer—short for Adaptive Moment Estimation. This optimizer is an extension of stochastic gradient descent that leverages adaptive calculations of moment estimates, and it's mostly recognized for its efficiency. For evaluating the performance of our model, I used a **binary**

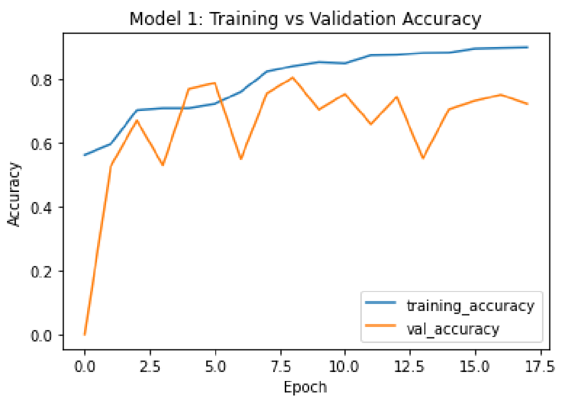
crossentropy loss function, which is particularly suitable for binary classification tasks. I monitored **accuracy** as metric to gauge the model's predictive capabilities. Training was performed in over 50 epochs, with a 20% validation split to assess the model's generalization on unseen data. To prevent overfitting and reduce unnecessary computation, I implemented early stopping with a patience parameter set to 9 epochs, thus halting the training process if no improvement in validation loss is observed over this interval. All models, from 1 to 5, have been compiled using the same parameters, but varying number of patience parameters.

Results

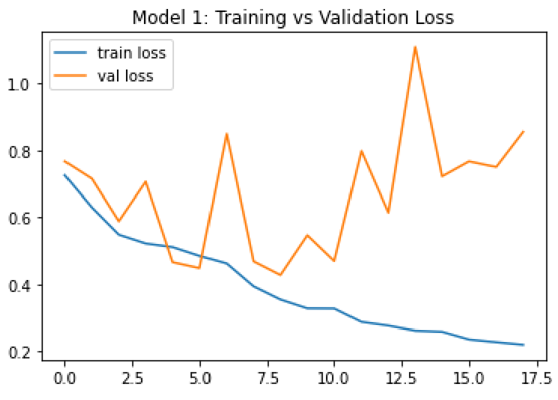
The model halted training at its optimum performance on epoch 9, with the outcomes illustrated in Table 1. Despite the relative simplicity of the architecture, it exceeded my initial expectations. Model 1 appears to marginally underfit, evidenced by its modest accuracy of 0.839 on the training dataset; it hasn't extracted substantial insights from the data provided. Interestingly, the model delivers a marginally higher accuracy of 0.84 on the test set, which might be attributed to the inherent variability in the models. While such stochastic effects mean consistent replication of these results isn't guaranteed, it suggests that the model has the capacity to identify key features that facilitate balanced performance across all datasets.
Moving forward, the plan is to refine the model's performance by incorporating additional layers and increasing the number of filters. This aims to enhance the model's capability to assimilate more complex patterns in the data.

Table 1: Model Performance Summary

Set	**Accuracy**	**Loss**
Training	0.8394	0.3541
Validation	0.8036	0.4273
Test	0.8418	0.3802



(a) Model 1 Accuracy



(b) Model 1 Loss

Figure 4: Training and validation curves for Model 1.

4.2 Baseline Models - Model 2

4.2.1 Architecture

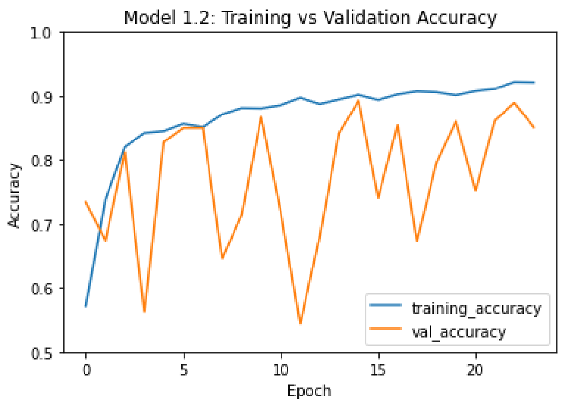
- Convolutional layer with 32 filters, kernel size (3, 3), relu activation.
- MaxPooling layer with (2, 2) pool size.

- Dropout layer with 0.5 rate.
- Convolutional layer with 64 filters, kernel size (3, 3), relu activation.
- MaxPooling layer with (2, 2) pool size.
- Dropout layer with 0.5 rate.
- Flatten layer.
- Dense layer with 32 nodes, relu activation.
- Dropout layer with 0.5 rate.
- Output layer with sigmoid activation.

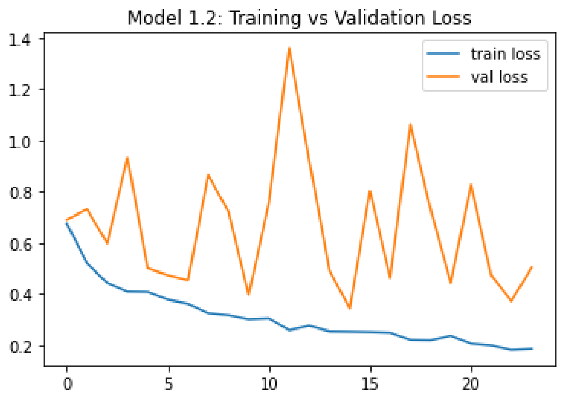
4.2.2 Results

Table 2: Model Performance Summary

Metrics	**Accuracy**	**Loss**
Training	0.9015	0.2530
Validation	0.8923	0.3434
Test	0.8586	0.3451



(a) Model 2 Accuracy



(b) Model 2 Loss

Figure 5: Training and validation curves for Model 1.2.

Model 2 has stopped after 15 epochs, and the weights are restored to the best epoch, whose results are shown in 2. Thanks the integration of an additional layer with 64 filters, the model has increased its capability to extract more complex information from the training set. This has led to a significant decline in loss across all datasets; however, there is a negligible increase in test set accuracy when comparing it to the first model. Moreover, the models' performance over the training and validation sets are higly unstable, which makes the model unreliable.

4.3 Model 3

To all next models I have added a **data augmentation** layer. Keras data augmentation sequential model includes layers that apply random transformations to enhance the dataset's diversity without collecting new data. It horizontally flips images, and introduces random rotations to a small degree (0.1 in this case), which helps the model to be insensitive to variations in image orientation. Moreover, it features random zoom, which slightly

(0.1) adjusts the scale of the image, aiding the model to recognize objects of different sizes and at varying distances. Collectively, these augmentations contribute to better model robustness and generalization by preventing overfitting and ensuring model exposure to a broader range of image variations during training.

4.3.1 Architecture - 3

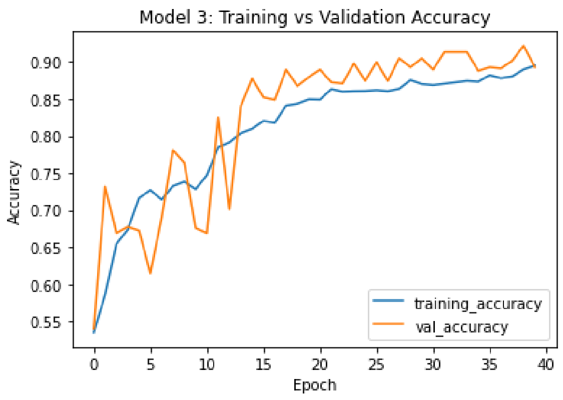
- Data augmentation layer.
- Convolutional layer with 32 filters, kernel size (3, 3), relu activation.
- MaxPooling layer with (2, 2) pool size.
- Dropout layer with 0.5 rate.
- Convolutional layer with 64 filters, kernel size (3, 3), relu activation.
- MaxPooling layer with (2, 2) pool size.
- Dropout layer with 0.5 rate.
- Convolutional layer with 128 filters, kernel size (3, 3), relu activation.
- MaxPooling layer with (2, 2) pool size.
- Dropout layer with 0.5 rate.
- Flatten layer.
- Dense layer with 32 nodes, relu activation.
- Dropout layer with 0.5 rate.
- Output layer with sigmoid activation.

4.3.2 Results

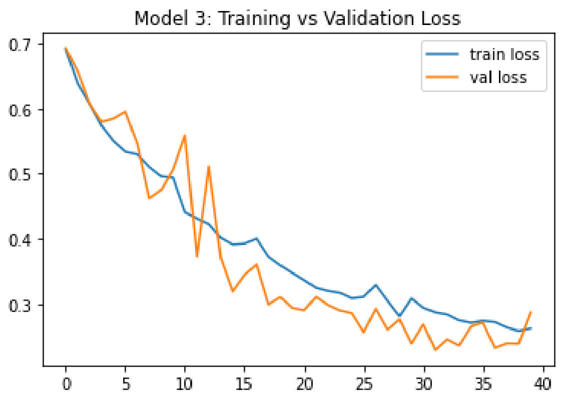
Model 3 stopped after 32 epochs, and the results are shown in 4. Training accuracy is slightly lower than what we saw in previous models, but the validation and test accuracies have seen a significant rise. Conversely, although the training loss is slightly higher than in model 2, validation and loss are significantly lower. This suggests that even though the model isn't performing as well on the training data, it's doing better at generalizing to new data, which is ultimately more important. Another positive aspect is that the model is showing consistent performance. The training and validation metrics are moving together, indicating a stable learning process without any erratic jumps or drops in performance. This stability, combined with the closer alignment between training and validation results, hints that the model is learning effectively and is less likely to be overfitting.

Table 3: Model Performance Summary

Metrics	**Accuracy**	**Loss**
Training	0.8707	0.2872
Validation	0.9134	0.2295
Test	0.8737	0.2783



(a) Model 3 Accuracy



(b) Model 3 Loss

Figure 6: Training and validation curves for Model 3.

4.3.3 Architecture - 3.1

For Model 3.1, I experimented with a different regularization technique, setting aside the dropout layer and integrating batch normalization layers into the architecture. Specifically, I placed a batch normalization layer after each convolutional layer and before the activation function, following a practice recommended in *Batch normalization: Accelerating deep network training by reducing internal covariate shift,* by Ioffe, S., and Szegedy, C., (June 2015). Batch normalization is a technique that normalizes the inputs of each layer of the network, with the purpose of speeding up training by using higher learning rates and it reduces the dependence on careful weight initialization. It can also be used as regularization technique, sometimes also replacing drop out (just like in this case).

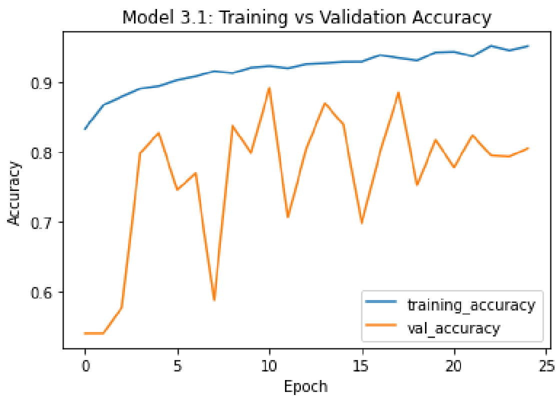
- Data augmentation layer.
- Convolutional layer with 32 filters, kernel size (3, 3).
- Batch Normalization layer.
- Relu activation layer.
- MaxPooling layer with (2, 2) pool size.
- Convolutional layer with 64 filters, kernel size (3, 3).
- Batch Normalization layer.
- Relu activation layer.
- MaxPooling layer with (2, 2) pool size.
- Convolutional layer with 128 filters, kernel size (3, 3).
- Batch Normalization layer.
- Relu activation layer.
- MaxPooling layer with (2, 2) pool size.
- Flatten layer.
- Dense layer with 32 nodes.
- Batch Normalization layer.
- Relu activation layer.
- Output layer with sigmoid activation.

4.3.4 Results

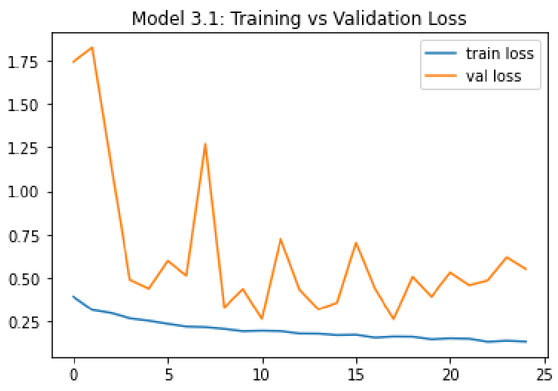
In testing different network designs, I found that substituting dropout layers with batch normalization seemed to yield better results than using both together. However using batch normalization did not improve the model's performance when I compare it to my baseline, model 3. This suggests that for this particular problem, the specific benefits of batch normalization aren't as pronounced as they are in some other cases.
Compared to model 3, the current network shows higher training accuracy, and based on the stored weights at epoch 18, it also demonstrates promising results in validation and test accuracy as well as loss. However, as it can be seen in 6, the model is highly unstable, thus unreliable in making predictions.

Table 4: Model Performance Summary

Metrics	**Accuracy**	**Loss**
Training	0.9339	0.1642
Validation	0.8846	0.2634
Test	0.8872	0.2569



(a) Model 3.1 Accuracy



(b) Model 3.1 Loss

Figure 7: Training and validation curves for Model 3.1.

4.4 Model 4

Finally, model 4 is my last experiment before conducting hyperparameter tuning to identify the best-performing model. Building upon the structure of model 3, I have introduced an extra convolutional layer that houses 256 filters, enhancing the network's capacity to capture even more complex features from the data.

4.4.1 Architecture

- Convolutional layer with 32 filters, kernel size (3, 3), relu activation.
- MaxPooling layer with (2, 2) pool size.
- Dropout layer with 0.5 rate.
- Convolutional layer with 64 filters, kernel size (3, 3), relu activation.
- MaxPooling layer with (2, 2) pool size.
- Dropout layer with 0.5 rate.
- Convolutional layer with 128 filters, kernel size (3, 3), relu activation.

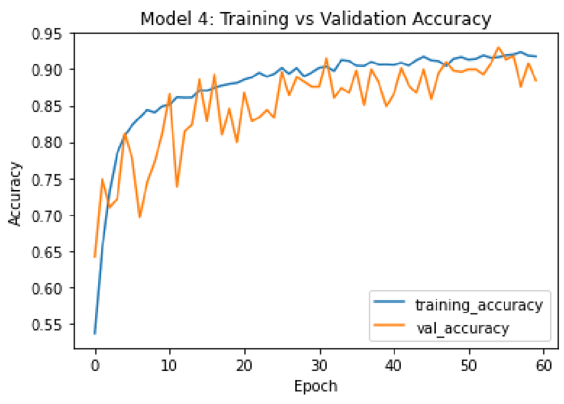
- MaxPooling layer with (2, 2) pool size.
- Dropout layer with 0.5 rate.
- Convolutional layer with 256 filters, kernel size (3, 3), relu activation.
- MaxPooling layer with (2, 2) pool size.
- Dropout layer with 0.5 rate.
- Flatten layer.
- Dense layer with 32 nodes, relu activation.
- Dropout layer with 0.5 rate.
- Output layer with sigmoid activation.

4.4.2 Results

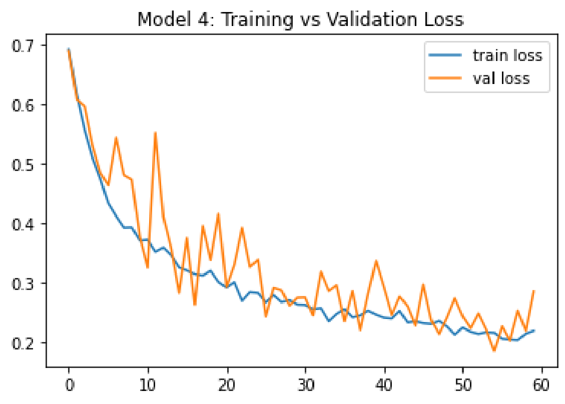
Comparing model 4 to model 3, it seems that incorporating an additional layer with 256 filters may have pushed the model towards overfitting. Training ran for a predefined 60 epochs without triggering early stopping, indicating the possibility of further loss reduction if training had continued. The validation loss was on a downward trend, settling between 0.18 and 0.25; however, a slight increase to 0.28 was observed towards the end, as shown in Table 4.

Table 5: Model Performance Summary

Metrics	**Accuracy**	**Loss**
Training	0.9178	0.2185
Validation	0.8846	0.2847
Test	0.8653	0.3131



(a) Model 4 Accuracy



(b) Model 4 Loss

Figure 8: Training and validation curves for Model 4.

5 Hyperparameter tuning - Model 5

Hyperparameter tuning is an optimization process to find the best set of hyperparameters for a machine learning model. It involves testing different combinations of hyperparameters, evaluating their performance, and selecting the combination that yields the best results based on a defined metric, such as validation loss. The **MyHyperModel** is a class I created to systematically test different network configurations. The model class defines a range of values for parameters like the number of filters in convolutional layers, dropout rates, and learning rates. The tuning is performed using **Bayesian Optimization**, which is a method that uses the outcomes of past trials to inform which set of hyperparameters should be tried next. This process typically results in more efficient and effective searching than random or grid search methods.
After 20 trials, and 50 epochs each, the optimized parameters are the following:

Table 6: Optimized Hyperparameters

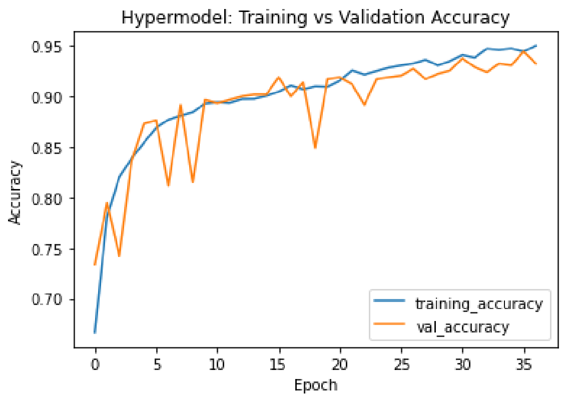
Hyperparameter	Value
filters_1	64
dropout_1	0.4
filters_2	32
dropout_2	0.5
filters_3	128
dropout_3	0.6
units_1	64
dropout_4	0.3
learning_rate	3.3681e-4

5.0.1 Results

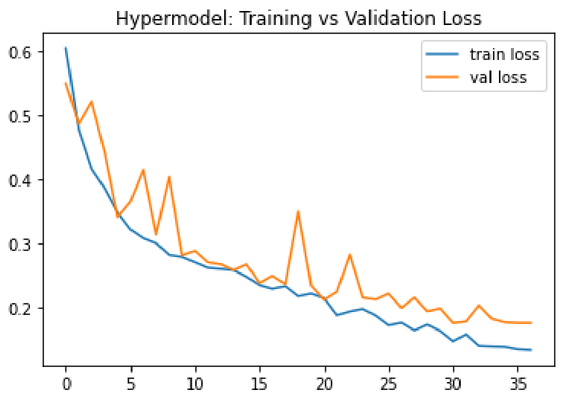
As expected, the implementation of Bayesian Optimization has decisively enhanced the model's capacity to process and learn from the dataset. The model has stopped after 42 epochs, and as reported in Table 7, the model not only attained a good training accuracy of 0.9548 but did so while maintaining robustness against overfitting. This is confirmed by the substantially high validation accuracy of 0.9508, which closely mirrors the training performance, proving the model's generalization ability. Although there is an expected decrease in accuracy on the test set, with a value of 0.9007, it still represents a strong performance. The modest increase in loss figures from training to test datasets (0.1205 to 0.2306) further echoes this sentiment.

Table 7: Model Performance Summary

Metrics	**Accuracy**	**Loss**
Training	0.9548	0.1205
Validation	0.9508	0.1565
Test	0.9007	0.2306



(a) *Hypermodel* Accuracy



(b) *Hypermodel* Loss

Figure 9: Training and validation curves for Model 4.

6 5-fold Cross Validation with zero-one loss

Finally, the last step of the process is to evaluate the performance of the models using 5-fold cross validation. K-fold cross-validation is a statistical method that evaluates the model with the following approach: first, it starts by randomly splitting the dataset into k equal-sized subsets or folds, then the model is trained and evaluated k times, each time using a different k fold as the validation set, while the remaining k-1 folds form the training set. In the following results, I'll be using three different metrics to evaluate the model: accuracy, binary cross-entropy as before, and zero-one loss.
The zero-one loss function is a very convenient loss function due to its simplicity, as it is counts every time the predicted label and the true label match, and divides them by number of observations to compute the average:

$$\text{Zero-One Loss} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}\left(\hat{y}_i \neq y_i\right) \tag{1}$$

6.1 Comparative Analysis

I performed 5-fold cross-validation on the last 2 models, model 4 and the *hypermodel*. While Model 4 offers a somewhat lower average accuracy of 0.8524 according to Table 8, its performance across the folds of cross-validation is quite stable. This is an important characteristic, as it implies reliable performance even when the model is exposed to varying subsets of the dataset. The Binary Loss and Zero-One Loss metrics, at 0.3557 and 0.1475 respectively, although higher than those of the hypermodel, reflect a consistent predictive behavior. Conversely, the hypermodel boasts a higher average accuracy of 0.9555, as seen in Table 9. However, this comes at the cost of greater variation in performance across the different folds, suggesting a potential overfitting issue or sensitivity to the data it is trained on. The Binary Loss and Zero-One Loss are lower, standing at 0.1235 and 0.0444 respectively, but the fluctuation in results could point to a lack of robustness when the model encounters new or diverse data scenarios. The visual representations in Figures 10 and 11 demonstrate these discrepancies in performance stability, with Model 4 showing more consistent results and the hypermodel presenting a broader range of outcomes across folds. In summary, although the hypermodel has demonstrated a high level of accuracy, its variability suggests that further investigation is needed to improve its consistency. Model 4, while slightly less accurate, offers a steadiness that may be more desirable in real-world applications where predictability and dependability are crucial. It is possible that a compromise or a hybrid approach could combine the high accuracy of the hypermodel with the stability of Model 4, resulting in a robust solution that also performs well across different data samples.

6.1.1 Results - Model 4

Table 8: Model Performance Summary

Average Metrics on 5-fold cv	
Accuracy	0.8524
Binary Loss	0.3557
Zero-One Loss	0.1475

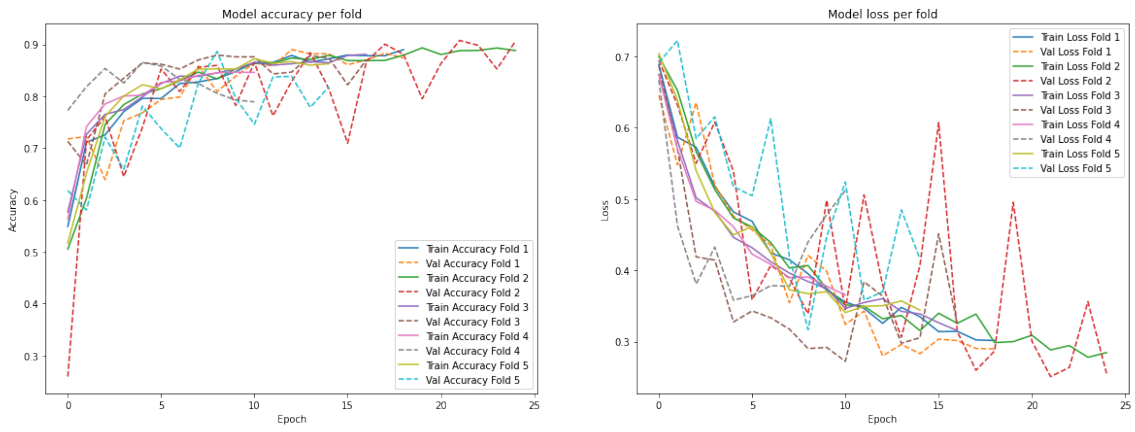


Figure 10: Model 4 performance on 5-fold cv.

6.1.2 Results - Hypermodel

Table 9: Model Performance Summary

Average Metrics on 5-fold cv	
Accuracy	0.9555
Binary Loss	0.1235
Zero-One Loss	0.0444

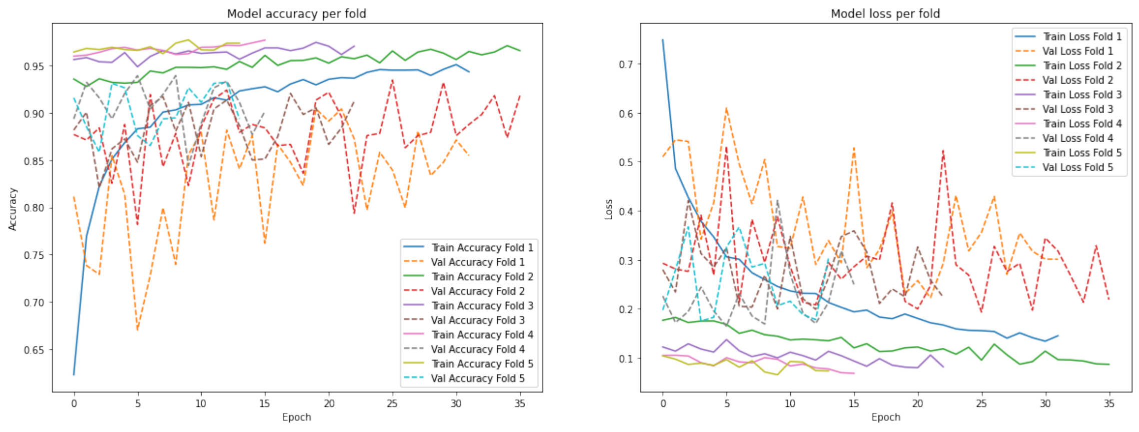


Figure 11: Hypermodel performance on 5-fold cv.

7 Bibliography

All the resources I have used for the purpose of this project are hereby listed:

1. Professor Cesa-Bianchi's lectures.

2. Keras Tensorflow Documentation (at github.com/tensorflow/tensorflow)

3. MIT Introduction to Deep Learning: *Convolutional Neural Networks for Computer Vision.*

4. Christian Versloot on GitHub (at github.com/christianversloot/machine-learning-articles/blob/main/how-to-use-k-fold-cross-validation-with-keras.md)

5. Ioffe, S., & Szegedy, C. (2015, June). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In International conference on machine learning (pp. 448-456). pmlr.