Muffin vs Chihuahua classification

Dzmitry Kurch, DSE

Machine Learning Project, March 2023

Abstract

In the era of rapid technological advancements, image classification has emerged as a quintessential problem in computer vision, influencing diverse domains ranging from medical diagnostics to autonomous vehicles. One particularly intriguing facet of this field is the ability to discern between visually similar objects, a task that often demands nuanced analysis and sophisticated algorithms.

The primary goal of this project is to evaluate the effectiveness of Convolutional Neural Network architectures in accurately classifying images given a dataset of muffins and chihuahuas. Through systematic experimentation and analysis, we aim to discern the optimal model configuration for this challenging image classification task, thereby contributing to the advancement of computer vision research.

1. Introduction

The choice to employ Convolutional Neural Networks (CNNs) as the primary architecture for this classification task is motivated by their unparalleled success in handling spatially structured data, such as images. CNNs leverage hierarchical feature extraction through convolutional layers, allowing them to capture intricate patterns and relationships within the input data. Furthermore, their ability to automatically learn relevant features from raw pixel values alleviates the need for handcrafted feature engineering, rendering them particularly well-suited for complex image classification tasks.

The complexity of distinguishing between muffins and chihuahuas lies not only in their superficial similarities but also in the subtle nuances that define each class. Muffins, with their diverse shapes, textures, and toppings, present a myriad of visual variations that must be discerned amidst backgrounds of varying hues and lighting conditions. Similarly, chihuahuas, with their diverse coat colours and patterns, pose a formidable challenge in differentiation, especially when juxtaposed against backgrounds of similar hues or textures.

In this project we will focus on the Keras framework for building our deep learning architecture. The choice of Keras, built on top of TensorFlow, as the framework for data preprocessing is motivated by its user-friendly interface and extensive functionality for deep learning tasks. Keras provides a high-level abstraction layer that simplifies the implementation of complex neural network architectures, making it an ideal choice for rapid prototyping and experimentation. Furthermore, its seamless integration with TensorFlow offers scalability and performance optimizations, ensuring robust handling of large-scale datasets and computationally intensive operations.

2. The dataset and its variables

The dataset is taken from Kaggle, here you may find the original source and the description https://www.kaggle.com/datasets/samuelcortinhas/muffin-vs-chihuahua-image-classification.

By and large, we have 4733 images for train across which there are 2559 chihuahuas and 2174 muffins which we can consider as almost balanced dataset. There is also a separate test dataset containing 1184 images with 640 chihuahuas and 544 muffins. Most of the images illustrate exactly what you think when somebody tells you the word “chihuahua” or “muffin”, though there are some which might confuse even the human intelligence:

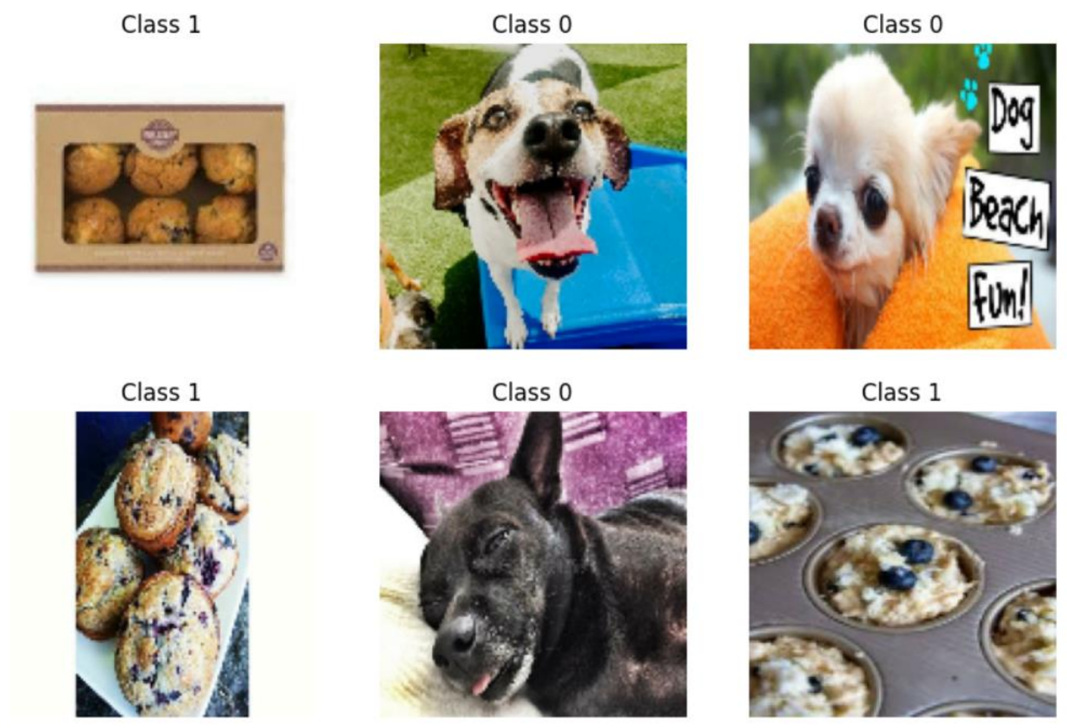


Figure 1 - Muffins and Chihuahuas pictures from training dataset

3. Modeling

In this section we introduce several convolutional neural network architectures (CNNs). Note that all computations and training are done using Google Colab processing engine.

3.1 Data preprocessing

The data preprocessing step begins with loading the dataset using the *image_dataset_from_directory* function provided by Keras, a high-level neural networks API that runs on top of TensorFlow, Google's open-source machine learning platform. This function allows for seamless loading of images directly from directories, simplifying the data ingestion process.

Following dataset loading, the images are resized from their original dimensions to a standardized size of 128x128 pixels. Resizing the images ensures uniformity in input dimensions across the dataset, facilitating efficient training of neural network models.

Subsequently, the dataset is divided into batches, each containing 128 images. Batching the data streamlines the training process, enhances computational efficiency and memory utilization.

3.2 Model quality estimates

To estimate the model quality, we will use the 5-fold cross validation and calculate the averaged accuracy of the prediction by folds. Since accuracy can be converted to zero-one loss just by subtracting it from 1, for the sake of simplicity and interpretability we will report the accuracy in our research. By choosing the best model hyperparameters with that estimation we finally fit the model on all available training data and estimate the quality on unseen test set.

It is important to note that the usage of cross validation when dealing with neural networks is a rare occurrence. That's why in the Keras interface there is no out-of-the-box functionality to deal with the folds, only basic train-validation split is supported as a parameter in function *image_dataset_from_directory.* Moreover, the limitation on RAM volume did not allow to load all dataset at once into memory to performs the spits in-place. I had to develop a workaround by creating ten temporary directories (five train and five validation) where each one contained two folders (chihuahua and muffins). That allowed me to benefit from the *lazy reading* provided by *image_dataset_from_directory* function.

3.3 Small model

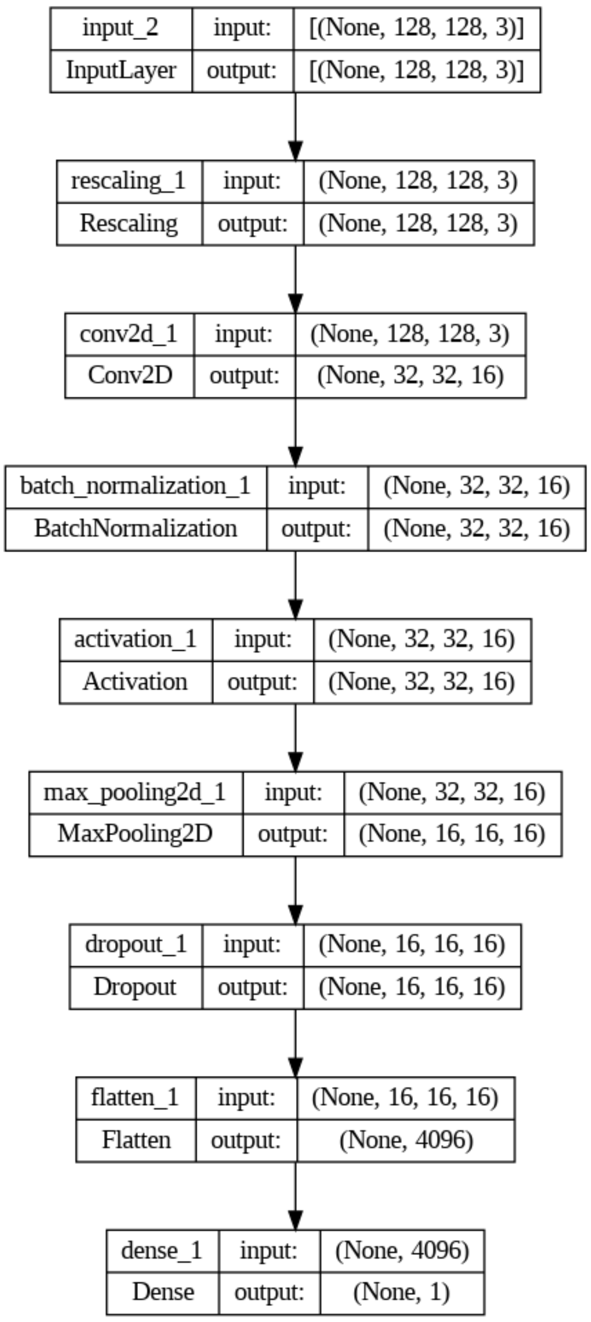
We start our journey with a very simple CNN by including all the basic layers. Here is a brief description of the architecture which maintains the order of the levels in the network:

1. *Rescaling Layer*: The first layer rescales the input image pixel values to a range between 0 and 1 by dividing them by 255. This normalization step is crucial as it ensures that all pixel values fall within the same numerical range, making optimization more stable and efficient.
2. *Convolutional Layer*: Following the rescaling step, the model employs a convolutional layer with 16 filters and a kernel size of 5x5. This layer performs feature extraction by convolving the input image with learnable filters, enabling the model to capture spatial patterns and local dependencies within the image. The use of convolutional layers is fundamental in deep learning for image processing tasks due to their ability to efficiently extract hierarchical representations from raw pixel data.
3. *Batch Normalization Layer:* After convolution, batch normalization is applied to standardize the activations of the previous layer across mini-batches. This technique helps in mitigating issues related to internal covariate shift, thereby accelerating convergence and improving the generalization capability of the model. Batch normalization ensures that

the model remains robust to changes in input distributions during training, enhancing its stability and performance.

4. *Activation Layer (ReLU):* Following batch normalization, the rectified linear unit (ReLU) activation function is applied element-wise to introduce non-linearity into the model. ReLU activation promotes sparse activation patterns and alleviates the vanishing gradient problem, facilitating faster convergence and more effective learning.
5. *MaxPooling Layer:* Subsequently, max-pooling with a pool size of 3x3 and a stride of 2 is employed to downsample the spatial dimensions of the feature maps. Max-pooling helps in reducing computational complexity and controlling overfitting by retaining only the most salient features while discarding redundant information. Additionally, it introduces translational invariance, making the model more robust to spatial translations within the input image.
6. *Dropout Layer:* Finally, dropout regularization with a dropout rate of 0.25 is applied to the output of the max-pooling layer. Dropout randomly deactivates a fraction of neurons during training, forcing the model to learn redundant representations and preventing over-reliance on specific features. This regularization technique enhances the model's generalization capability and reduces the risk of overfitting, thereby improving its performance on unseen data.
7. *Flatten Layer*: Following the last dropout layer, the feature maps are flattened into a one-dimensional vector. This operation reshapes the spatially structured feature maps into a format suitable for input to the subsequent fully connected layers. By flattening the feature maps, the spatial relationships between pixels are discarded, allowing the model to focus solely on learning high-level feature representations across the entire image.
8. *Dense Layer (Output Layer):* The final layer of the model is a densely connected layer with a single neuron, equipped with a sigmoid activation function. This layer serves as the output layer of the binary classification model, producing a probability score indicating the likelihood that the input image belongs to the class 1 (muffin) or class 0 (chihuahua). The sigmoid activation function squashes the output scores between 0 and 1, interpreting them as probabilities. During training, the model learns to minimize the binary cross-entropy loss between the predicted probabilities and the ground truth labels, thereby optimizing its ability to discriminate between muffins and chihuahuas.

Total number of model parameters to tune was 5377. Here is the visual representation of the network done with *keras.utils.plot_model()* function:


input_2 | input: | [(None, 128, 128, 3)]
InputLayer | output: | [(None, 128, 128, 3)]
rescaling_1 | input: | (None, 128, 128, 3)
Rescaling | output: | (None, 128, 128, 3)
conv2d_1 | input: | (None, 128, 128, 3)
Conv2D | output: | (None, 32, 32, 16)
batch_normalization_1 | input: | (None, 32, 32, 16)
BatchNormalization | output: | (None, 32, 32, 16)
activation_1 | input: | (None, 32, 32, 16)
Activation | output: | (None, 32, 32, 16)
max_pooling2d_1 | input: | (None, 32, 32, 16)
MaxPooling2D | output: | (None, 16, 16, 16)
dropout_1 | input: | (None, 16, 16, 16)
Dropout | output: | (None, 16, 16, 16)
flatten_1 | input: | (None, 16, 16, 16)
Flatten | output: | (None, 4096)
dense_1 | input: | (None, 4096)
Dense | output: | (None, 1)
Figure 2 - Small model architecture

The cross-validation accuracy score of that model turned out to be 0.89 on train and 0.81 on validation, whereas for test it was 0.81 – close to validation, fortunately no data leakage. But still the model was pretty simple, so I decided to move on with a more comprehensive one right away.

3.4 Medium model

Following the same reasoning of building a model architecture as I described above, I created a bigger model by just stacking more layers maintaining their best-practice order.

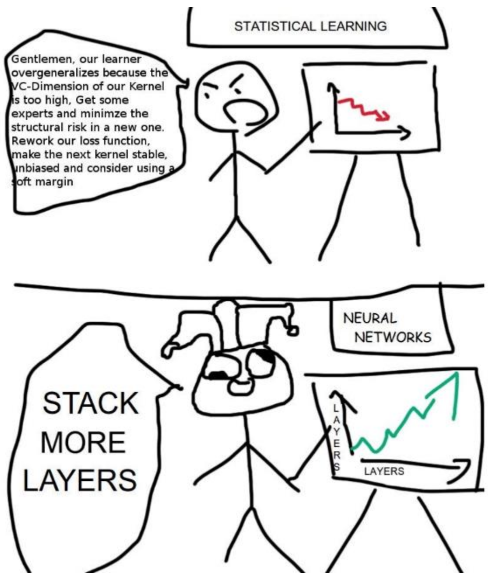

STATISTICAL LEARNING
Gentlemen, our learner overgeneralizes because the VC-Dimension of our Kernel is too high, Get some experts and minimze the structural risk in a new one. Rework our loss function, make the next kernel stable, unbiased and consider using a soft margin
NEURAL NETWORKS
STACK MORE LAYERS
LAYERS
LAYERS

Figure 3 - Classic mem Statistical learning VS Neural Networks

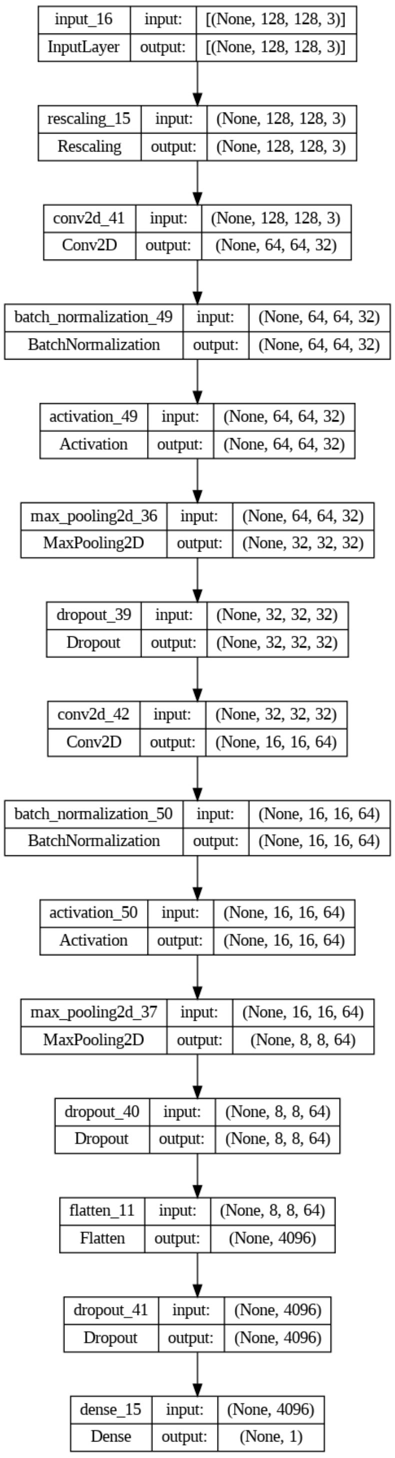

input_16 | input: | [(None, 128, 128, 3)]
InputLayer | output: | [(None, 128, 128, 3)]
rescaling_15 | input: | (None, 128, 128, 3)
Rescaling | output: | (None, 128, 128, 3)
conv2d_41 | input: | (None, 128, 128, 3)
Conv2D | output: | (None, 64, 64, 32)
batch_normalization_49 | input: | (None, 64, 64, 32)
BatchNormalization | output: | (None, 64, 64, 32)
activation_49 | input: | (None, 64, 64, 32)
Activation | output: | (None, 64, 64, 32)
max_pooling2d_36 | input: | (None, 64, 64, 32)
MaxPooling2D | output: | (None, 32, 32, 32)
dropout_39 | input: | (None, 32, 32, 32)
Dropout | output: | (None, 32, 32, 32)
conv2d_42 | input: | (None, 32, 32, 32)
Conv2D | output: | (None, 16, 16, 64)
batch_normalization_50 | input: | (None, 16, 16, 64)
BatchNormalization | output: | (None, 16, 16, 64)
activation_50 | input: | (None, 16, 16, 64)
Activation | output: | (None, 16, 16, 64)
max_pooling2d_37 | input: | (None, 16, 16, 64)
MaxPooling2D | output: | (None, 8, 8, 64)
dropout_40 | input: | (None, 8, 8, 64)
Dropout | output: | (None, 8, 8, 64)
flatten_11 | input: | (None, 8, 8, 64)
Flatten | output: | (None, 4096)
dropout_41 | input: | (None, 4096)
Dropout | output: | (None, 4096)
dense_15 | input: | (None, 4096)
Dense | output: | (None, 1)

Figure 4 - Medium model architecture

The resulted number of parameters to tune is 23873. This time as the model seemed to be more promising there was applied an additional step of hyperparameter tuning.

The tuning of the dropout rate in the neural network model is chosen to balance between underfitting and overfitting. When the dropout rate is small, there's a risk of overfitting as the model may rely too heavily on specific features. Conversely, when the dropout rate is large, there's a risk of underfitting as the model's ability to learn intricate patterns may be compromised. Therefore, selecting the optimal dropout rate is crucial to ensure that the model generalizes well to unseen data while capturing relevant features from the training data.

On the pictures below you may see the *learning curves* for the Medium (Base) model where we illustrated the dependency of model performance on the dropout rate.

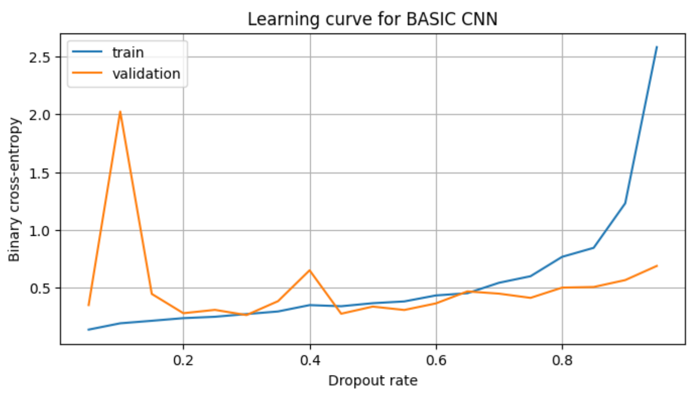


Figure 5 - Dropout rate tuning. Binary cross-entropy

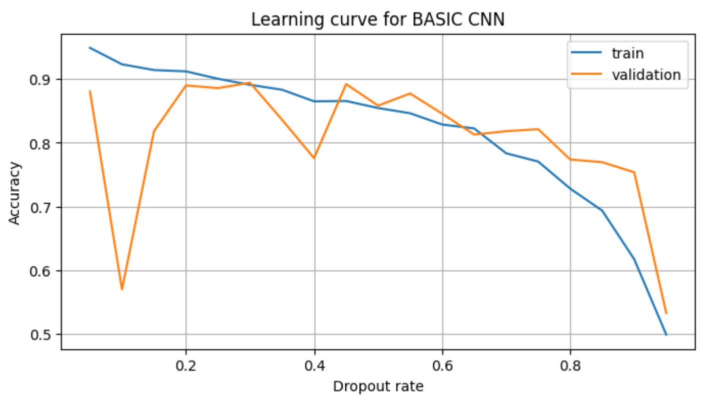


Figure 6 - Dropout rate tuning. Accuracy

Unfortunately, there is no “obvious” optimal value of the dropout rate for the given architecture, the sensitivity of the model to small changes in this parameter turned out to be unexpectedly high in some regions. But the general tendency is clear and aligned with the theory – too small value of dropout rate leads to overfitting whereas too large to clear underfitting. In the end I decided to stick to the value of 0.3 as it seemed to me the local minima in the stable with respect to the parameter region.

After choosing the value of dropout rate I ran a cross-validation on the dataset and the model ended up with the accuracy score of 0.89 on train and 0.87 on validation, with a test accuracy of 0.89. That is a clear improvement with respect to the result with a previous model. Let’s see if we can “go deeper” and improve that by adding more layers.

3.5 Large model

Large model was constructed by using the medium model as a basement on the top of which I added one more block of convolution – batch normalization – activation – max pooling – dropout, in total I ended up with 94657 parameters.

During the cross-validation epoch learning it was already possible to notice that such a large model fails to generalize the patters of chihuahuas and muffins since the validation accuracy remained unchanged up to the last epoch on each fold with the value 0.54. That is the same as the proportion of chihuahuas in the training dataset – as if the model was giving a constant prediction on unseen data!

My hypothesis was that this might have happened due to the extreme model complexity. To overcome that I added an additional preprocessing step of data augmentation which should have been enrich the training data by producing more images which seem to be obviously the same for the human eye but different for the model.

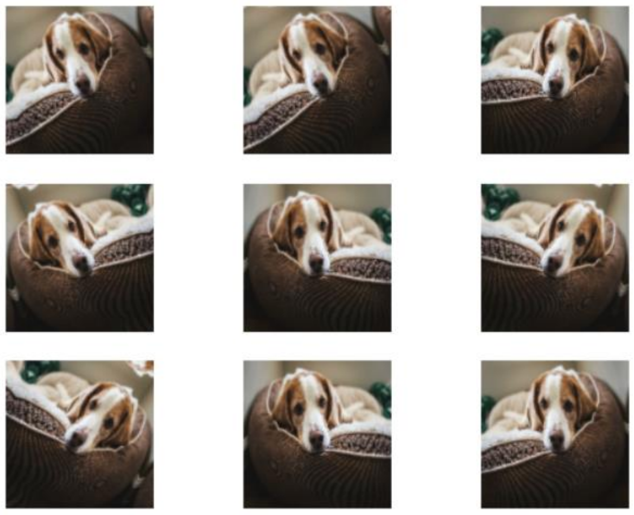


Figure 7 - Data augmentation example

Unfortunately, that didn't work well either and the validation error remained stable around 0.54 slightly increasing on some epochs which was most probably caused by the training images perturbation. The effect of data augmentation was not strong enough to fight the drawbacks of the large model architecture.

3.6 Getting all together

Here is an overview table containing the summary of my experiments.

Model type	*Number of parameters*	*CV train accuracy*	*CV validation accuracy*	*Test accuracy*
Small	5337	0.89	0.81	0.81
Medium	23873	0.89	0.87	0.89
Large	94657	0.90	0.54	0.54
Large + Data Augmentation	94657	0.89	0.54	0.54

4. Best model analysis

In this section we dive deeper into the medium model result analysis as it turned out to be the most successful one. On the right you will find the confusion matrix based on the test data evaluation.

Test precision is 0.915, test recall is 0.824. Our model has slightly less False Positives with respect to the False Negatives.

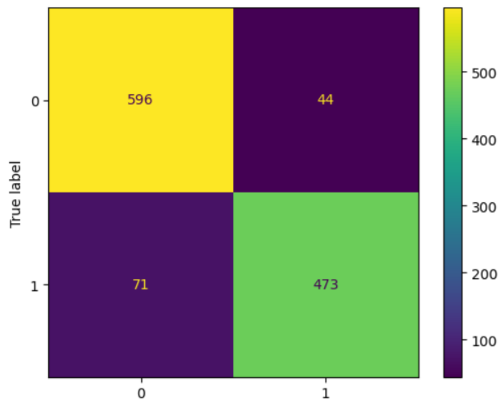


Figure 8 - Test confusion matrix best model

To study deeper the model behaviour, I have listed the top images in each category (TP, FP, TN, FN) where by “top” I mean the ones which have the extremely “right” or “wrong” probability score received from the model.

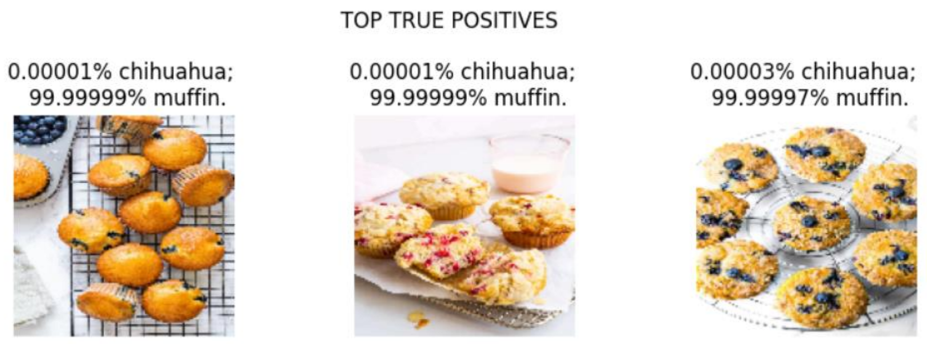

TOP TRUE POSITIVES
0.00001% chihuahua; 99.99999% muffin.
0.00001% chihuahua; 99.99999% muffin.
0.00003% chihuahua; 99.99997% muffin.

Figure 9 - Top test TP best model

Top True Positives are the images which shows muffins as they are imagined by the the most people – clear and unambiguous.

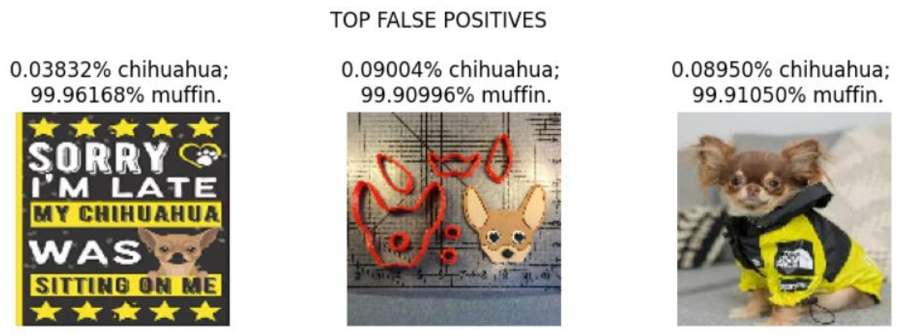

TOP FALSE POSITIVES
0.03832% chihuahua; 99.96168% muffin.
SORRY I'M LATE MY CHIHUAHUA WAS SITTING ON ME
0.09004% chihuahua; 99.90996% muffin.
0.08950% chihuahua; 99.91050% muffin.

Figure 10 - Top test FP best model

Top False Positives were different from the ones suggested by that mem in the description of this dataset. Our model confused some obvious to the human eye images of chihuahuas with muffins probably to some learned patterns of small and colourful details present on the picture.

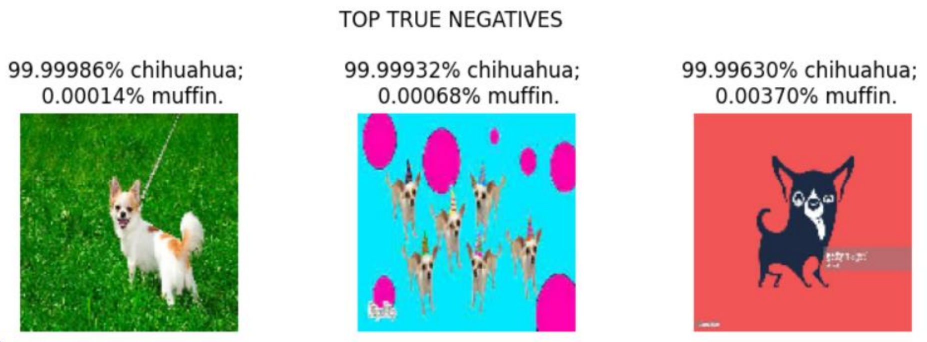

TOP TRUE NEGATIVES
99.99986% chihuahua; 0.00014% muffin.
99.99932% chihuahua; 0.00068% muffin.
99.99630% chihuahua; 0.00370% muffin.

Figure 11 - Top test TN best model

Top True Negatives turned out to be quite strange. My expectations were that I would see some clear and big images of chihuahuas. Instead, we observe some animated and small in its dimensions chihuahuas, sometimes even duplicated multiple dogs.

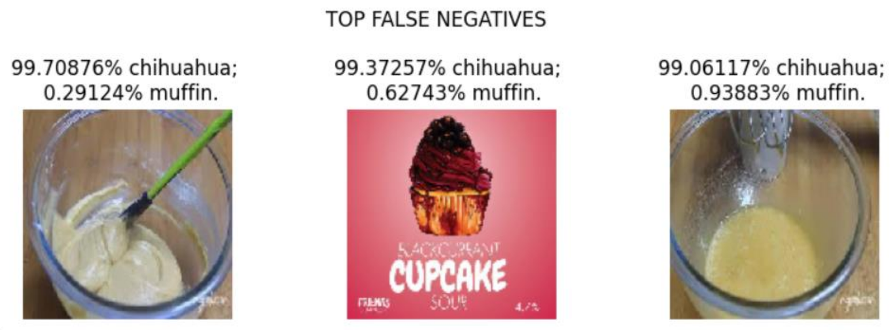

TOP FALSE NEGATIVES
99.70876% chihuahua; 0.29124% muffin.
99.37257% chihuahua; 0.62743% muffin.
99.06117% chihuahua; 0.93883% muffin.
CUPCAKE

Figure 12 - Top test FN best model

Top false negatives are represented by some muffins in the format of not yet cooked mixtures and one colourful animated image of muffin all confused with chihuahuas. I guess we cannot even blame our model here as these are standing-out images which either should be excluded both from train and test or included in a higher quantity – it depends if we expect from our model to recognize mixtures of muffins as muffins at all.

5. Conclusions

In this study, we explored the efficacy of Convolutional Neural Network (CNN) architectures for classifying images of muffins and chihuahuas. Through rigorous experimentation, we investigated the impact of model architecture, dropout regularization, and hyperparameter tuning on the classification performance.

Our findings underscore the critical importance of striking a delicate balance between model complexity and regularization to mitigate the risk of overfitting and underfitting. Specifically, we observed that the size of the model architecture plays a pivotal role in this trade-off: while deeper and more complex architectures may afford greater representational capacity, they also increase the risk of overfitting, especially when trained on smaller datasets. Conversely, shallower architectures may mitigate overfitting but may struggle to capture the intricacies of the data, leading to underfitting.

Moreover, our experiments highlight the significance of selecting an appropriate dropout rate to effectively regularize the model. By systematically tuning the dropout rate, we can enhance the model's generalization performance while preventing it from memorizing noise and irrelevant patterns in the training data.

In conclusion, this study underscores the multifaceted nature of image classification tasks and highlights the importance of thoughtful model design and hyperparameter tuning. By leveraging insights gained from this research, future endeavours in computer vision can continue to push the boundaries of classification accuracy and robustness, ultimately advancing the state-of-the-art in image recognition technologies.