Università degli Studi di Milano

Data Science and Economics

Statistical Methods for Machine Learning
Developing a CNN for image classication

Luca Perego
Registration numbers: 26590A
May 2024

ACADEMIC YEAR 2023-2024

1 Introduction

This project aims to develop a Convolutional Neural Network (CNN) suited for image recognition. More specifically, to train a CNN for discerning images of chihuahuas from muffins. This task is fulfilled through Tensorflow's Keras API. The code can be inspected on my GitHub repository[1].

2 Theoretical Framework

In this first section of the paper, I will briefly introduce the theory behind the models that will be implemented.
I will describe the overall structure of a Neural Network, the different layers involved and some regularization techniques.

2.1 Neural Network Fundamentals

Neural networks are predictors characterized by their complexity and versatility. They are composed of *nodes* (also called *neurons*) and links between the nodes, characterized by *weights*. The nodes are grouped into layers. Usually, a Neural Network is composed of three types of layers:

1. Input layer
2. Hidden layers
3. Output layer

If the data is processed from the input layer to the output layer without recursion or loops, the neural network is said to be feed-forward.

In a feed-forward neural work, the input layer receives the data over which a predictor will be produced (training data) and sends it to the next hidden layer. The hidden layers perform mathematical operations on the data. At the most fundamental level, for each node the input is scaled by a weight and a bias term is added to it. The result of this computation is used as input for an **activation function**. The outputs of the activation functions are summed together and are sent to the following nodes, a process called *forward propagation*. This process is repeated for each hidden layer until the output layer is reached and a final result is produced.

[1] I/We declare that this material, which I/We now submit for assessment, is entirely my/our 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/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences 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/us or any other person for assessment on this or any other course of study.

From this simplified sketch, two elements appear extremely important for the functioning of the neural network:

1. **Weights and biases**: these are updated at each iteration of the algorithm over the training set, through a mechanism called *back propagation*. This will be discussed in detail shortly.

2. **Activation function**: there are different types of activation functions, like *softmax, ReLU, sigmoid*. In this specific project, I will employ the following:

   (a) **ReLU** - Rectified Linear Unit: This activation function is piecewise linear and corresponds to:

   $$f(x) = max(0, x)$$

   The output of the ReLU is zero for inputs lower than zero, or the input itself. In the project, the ReLU function will be employed in all layers, except for the

   (b) **Sigmoid**: The sigmoid function is one of the first activation functions used in neural networks. It is characterised by the sigmoidal shape and can be represented by the following formula:

   $$f(x) = \frac{1}{1 + e^{-x}}$$

   Since the task is binary classification, the sigmoid will be employed in the very last layer to assess whether it's more likely for a given picture to be a muffin or a chihuahua.

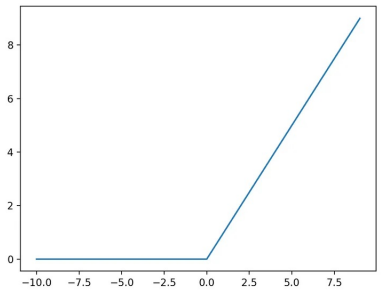

Figure 1: ReLU activation function

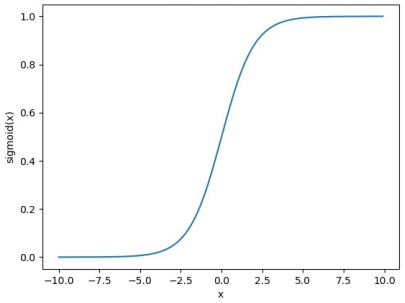

Figure 2: Sigmoid activation function

Neural networks learn by looking at the training data multiple times and by updating their weights for each cycle. These cycles are called **epochs**, and their number can be decided before training the model. In most of the project, I will

let the model train for 60 epochs.
During each epoch, the algorithm is "fed" the data in batches, which are subsets of the training data. The size of each batch is an attribute that can be defined by the user. I will employ batches of size equal to 32.
At the end of all the epochs, the results are provided in the form of *loss* and *accuracy*. For each model, a graph reporting both measures across epochs is presented.

The loss is a measure of the error of the predictor. There are different **loss functions** that can be adopted, but I will mainly use *binary cross entropy*. This loss function is particularly well-suited for binary classification. It is represented by the following formula:

$$\mathcal{L}_{BCE}(y_i) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

However, the loss used on the test set will be the 0-1 loss, which corresponds to:

$$\mathcal{L}_{01}(y, \hat{y}) = \begin{cases} 0 \, if \; y = \hat{y} \\ 1 \, otherwise \end{cases}$$

On the other hand, accuracy takes values in the interval [0,1] and represents the "precision" of the model.

Finally, in training the neural network it is necessary to define an *optimizer*. For this reason, I adopt the **Adam** optimizer. This optimization algorithm combines the advantages of *adaptive gradient algorithm* and *root mean square propagation*[3].

2.2 CNN layer types

Convolutional neural networks are instrumental in image recognition, although they also perform well in natural language processing. To develop the network I will employ keras'API. I will now discuss the different layers that compose a convolutional neural network:

- **Convolutional layer**: this layer is the basis of every CNN. Here the mathematical operation of 'convolution' takes place. Essentially a filter scans the image to check for a given feature. The filter (or kernel) is typically a 3x3 matrix. It is applied to a specific area of the image and the dot product between the input pixels and the filter is computed. Then, the filter shifts by a *stride*, and the process is repeated until the whole image has been scanned. The final output is called *feature map*.
  There are four hyperparameters involved in a convolutional layer:

  1. **Number of filters**: often either 16, 32, 64 or 128. The more filters the more depth the output has. Often in CNN, the number of filters

increases with the depth of the layer. This would enable the deeper layer to pick up on more specific details of the input.

2. **Kernel size**: The window of pixels that the filters analyze together. The most common kernel size is 3x3, and it has been shown that such size often outperforms greater ones (like 5x5 or 7x7).[5]
3. **Stride**: the distance that the kernel moves over the input matrix. The larger the stride the smaller the output.
4. **Zero-padding**: it acts when the filter does not fit the input size. It sets the elements that fall outside the matrix to zero. Padding can be of three types: Valid, Same or Full.[7]

- **Pooling Layer**: it performs dimensionality reduction. Like in the convolutional layer, a filter is passed across the input. Filters in the pooling layer have no weights but perform aggregation functions. There are two types of pooling:

  1. **Max pooling**: the pixel with the maximum value is sent to the output array.
  2. **Average Pooling**: the average value within the receptive field of the filter is computed and sent to the output array.

  When

- **Fully-connected layer**: based on its input, it performs a classification task. Usually, they are associated with *SoftMax* activation functions, unlike the convolutional layers that often use $ReLU$. However, in this project, the fully connected (dense) layers will be characterized by a *sigmoid* activation function.

- **Flatten layer**: it "flattens" the pooled feature map produced by the pooling layer into a 1-dimensional array. Its output is then sent to the last dense layer, that performs the final classification task.

2.3 Regularization/Optimization techniques

Nerual networks are prone to overfitting. This implies that the algorithm would model the statistical noise contained in the training data, which would hinder performance on new data points. For this reason, overfitting can be recognized by a large gap between the training and validation loss. To tackle this issue the following techniques will be implemented.

2.3.1 Dropout

Dropout is a regularization method articulated in a "Dropout layer" in our model. With "dropout" some nodes of the neural network are randomly dropped, basically creating a new architecture.[6]

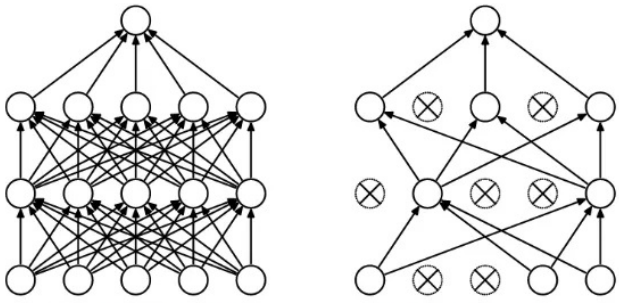

Figure 3: Graphical visualization of dropout. Source: Nitish et al. (2014

When dropout is implemented a "drop probability" must be defined. Assuming a hidden layer has 100 nodes, with a dropout probability = 0.5, 50 nodes will be randomly dropped in every batch, making the model more "generalised". According to the initial paper where Dropout has been proposed, the most adequate implementation would be inserting a dropout layer in between the dense layers, with a suggested dropout probability of 0.5.[6]. However, more recent studies[4] have found that applying dropout to convolutional layers, with a low dropout probability (e.g. 0.1 or 0.2) can benefit the model.

2.3.2 L2 regularization

L2 regularization is another regularization method useful for reducing overfitting. Through this technique, an extra term (regularization term) is added to the loss function of the network, much like in a Ridge regression. Another option would be L1 regularization, which is closely related to the Lasso regression. Both types of regularization require a positive λ parameter, which dictates the magnitude of the regularization.

2.3.3 Batch Normalization

Batch normalization (BN) is another method to tackle the issue of overfitting. The rationale behind BN is that by normalizing the output of a given layer, stochastic gradient descent will have an easier time converging during training. Thus, BN can be implemented as a layer between hidden layers. Its function is to normalize the output of the layer before it and feed the transformed output to the layer next to it.[1]

3 Dataset description

The dataset employed comes from Kaggle and is already divided into training and test sets. The training dataset is composed of 4733 images, while the test

dataset is composed of 1194 images.
We visualize some of these images to get an idea of how they are.

Figure 4: Example of chihuahua images

Figure 5: Example of muffin images

By visualizing these examples we can already extrapolate some information. First, it can be noted that the images have different sizes, thus it will be necessary to transform them in a common format. Additionally, some images are not proper 'chihuahuas' or 'muffins', but can be other objects linked to the categories (e.g. dog food with pictures of chihuahuas).
To train the CNN it's necessary to make all pictures the same format. I decided to make all images 150x150 pixels and in greyscale. I decided to use this solution for efficiency purposes. At 150x150 pixels the images retain information (at least to the human eye) regarding whether they portray chihuahuas or muffins. The lower average resolution merged with the greyscale, should make training less time-consuming. Finally, I normalized the pixel values so that they are between 0 and 1, to make convergence faster during training. The following is a subset of the transformed input data:

Figure 6: Sample of the processed data. This is the data that will be used as input to train the CNN.

It can be seen that it is still easy to discern muffins from chihuahuas. Furthermore, it can be noted that our dataset labels the chihuahuas as "0" and muffins as "1".
The dataset is divided into three parts: training, validation and test set.
The training set is the data through which the algorithm will be trained and through which a predictor will be produced. The validation set will be employed to assess the performance of the model. The validation set is made of 15% of the pictures of the training data, randomly chosen. Finally, the test set is composed of images never seen before by the algorithm, which will be used to assess its true performance on novel data.

To train the model, I initially intended to develop a data pipeline with TensorFlow's dataset API. This would have avoided loading all the data into memory, making it a more scalable and transferable solution. I started by using `tf.keras.utils.image_dataset_from_directory`, which allowed for quick preprocessing (like resizing images and converting them to greyscale). The output of using the `keras.utils` is not a dataset per se, but a *generator*, thus I then created a data iterator through the `.as_numpy_iterator()` method and through the `.next()` method I could access the data pipeline itself.

However, this process was revealed to be time-consuming. Therefore, my solution was to use Google Colab, download the data from the Kaggle API and create training and test sets through 'for' loops. At the end of this procedure, I serialized these objects through 'pickle', to access the necessary data quickly during the analysis. This process can be evaluated through the `data_prep.ipynb` file in my GitHub repository.
This process, plus the GPU runtime of Google Colab, drastically reduced the training time of the models (from two minutes for each epoch to two seconds each)[2].

4 Model Implementation

In this section, three architectures will be presented. For each model, I will define the layers and the associated parameters. The 'provisional'[3] results will be presented. Then, hyperparameter tuning will be carried out, and the final results associated with each model will be reported.
Before displaying the models, I briefly recap the layers used and their respective parameters in the order they have on the Keras code.

- **Conv2D**: Convolutional layer.
  - Number of filters (powers of 2)
  - Kernel size (a tuple of two integers)
  - Strides (either an integer or a tuple of two integers)
  - Activation function (either ReLU or Sigmoid)
  - kernel_regularizer (L1 or L2 regularization, with respective λ value)
- **MaxPooling2D**: Pooling layer, with MaxPooling.
  - pool size: a tuple of two integers representing the area of pixels over which the layer has to select the maximum. The default value is (2,2).
- **Flatten**: Flatten layer.
- **Dense**: Fully connected layer.
  - Units (Positive integer, power of 2): dimensionality of the output space.[2]
- **Dropout**: Dropout layer

[2] Arguably, this improvement was due to the use of GPUs. While this is probably true, I found my previous solution for loading data more time-consuming, although with a better scalability. In the end, I think the final solution proved adequate to the task.

[3] With 'provisional' I refer to the results obtained by the model without hyperparameter tuning.

  - Dropout probability.

- **BatchNormalization**: BN layer

For each architecture, I will report the layer attributes as they are in the code. This implies that some attributes will be left blank since I will employ the default values. For instance, in MaxPooling I will always use a `pool_size` of (2,2), which corresponds to the default setting.
All the models are trained with the following characteristics:

- Epochs = 60
- Batch size = 32
- Validation split = 0.15
- Adam optimizer
- Binary Cross Entropy loss function
- Strides = (1,1)

4.1 First model

The first model I implemented has the following architecture:

1. **Conv2D**: 16 filters, (3x3) Kernel Size, ReLU
2. **MaxPooling2D**: all default, (2,2) pool size
3. **Conv2D**: 32 filters, (3x3) Kernel Size, ReLU
4. **MaxPooling2D**
5. **Conv2D**: 64 filters, (3x3) Kernel Size, ReLU
6. **MaxPooling2D**
7. **Flatten**
8. **Dense**: 256 units, ReLU
9. **Dense**: 1 unit, Sigmoid

4.1.1 Provisional results

The following is the result of this first architecture. It can be noted that there is an overfitting problem.

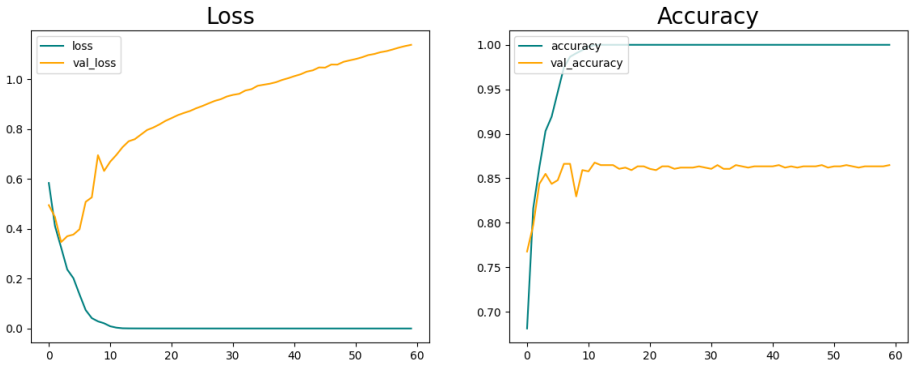

Figure 7: Performance plot of the first model. The gap between the validation and training loss/accuracy hints at an overfitting problem. This aside, the model achieves a validation accuracy more or less stable around 0.85

4.1.2 Refining the model

Overfitting can be observed by the large gap between training and validation loss. For this reason, different variations of this first model are developed.
First, we introduce a dropout layer on the dense layer before the output, using a value of p = 0.5. This is done following the methodology of Hinton (2012). Additionally, I include another dense layer after the dropout layer. I reduce the number of nodes in each fully connected layer, from 256 to 64 nodes. However, these values will be optimized in the hyperparameter tuning section. We obtain the following results:

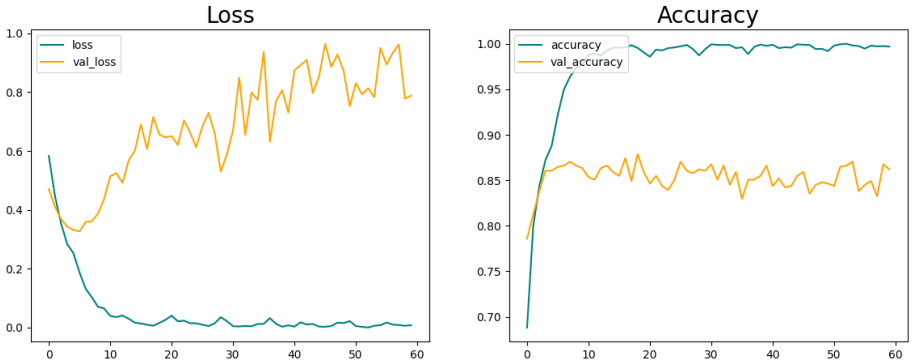

Figure 8: The validation accuracy seems even more stable around the 0.85 mark, however, there is still a large gap between the validation and training loss. It cannot be said that the dropout layer has solved the overfitting issue.

Since the gap between the two accuracies is persistent, I increment the dropout probability. I perform a grid search, by testing the effects of multiple values and eventually settle on 0.8. The results are the following:

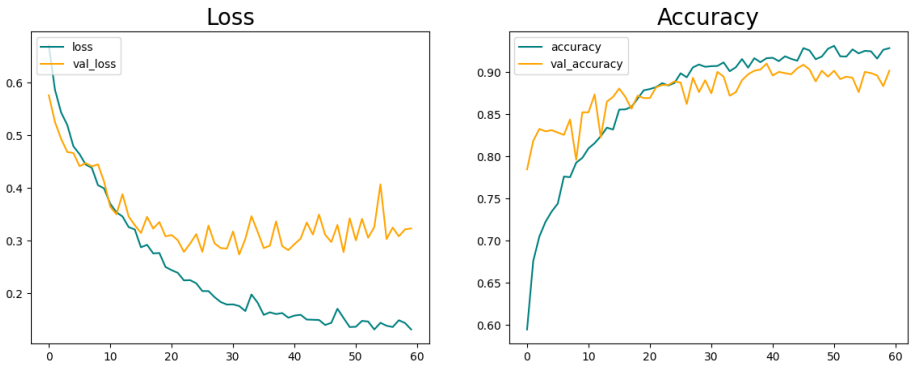

Figure 9: By incrementing the dropout probability, the overfitting problem is much less pronounced.

Another model is developed. Here the dropout layer is preserved and L2 regularization is applied to all convolutional layers and the first dense layer. The λ parameter is set to 0.01. The results are the following:

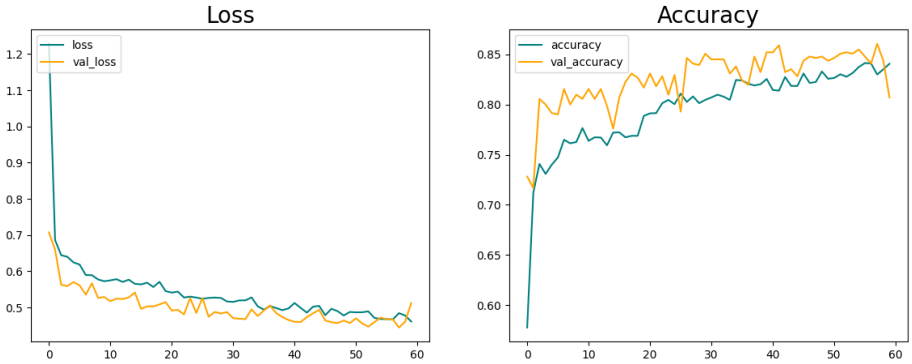

Figure 10: By implementing both dropout as well as L2 regularization, the model seems to not overfit anymore. However, the validation accuracy is reduced, being now between 0.80 and 0.85

4.2 Second model

For the second architecture, a deeper network is developed.

1. **Conv2D**: 16 filters, (3x3) Kernel Size, ReLU
2. **MaxPooling2D**: all default,pool size = (2,2)
3. **Conv2D**: 32 filters, (3x3) Kernel Size, ReLU
4. **MaxPooling2D**
5. **Conv2D**: 64 filters, (3x3) Kernel Size, ReLU
6. **MaxPooling2D**
7. **Conv2D**: 128 filters, (3x3) Kernel Size, ReLU
8. **MaxPooling2D**
9. **Flatten**
10. **Dense**: 256 units, Relu
11. **Dropout**: 0.5 dropout probability
12. **Dense**: 1 unit, Sigmoid

4.2.1 Provisional results

The results of the first version of the second model are the following:

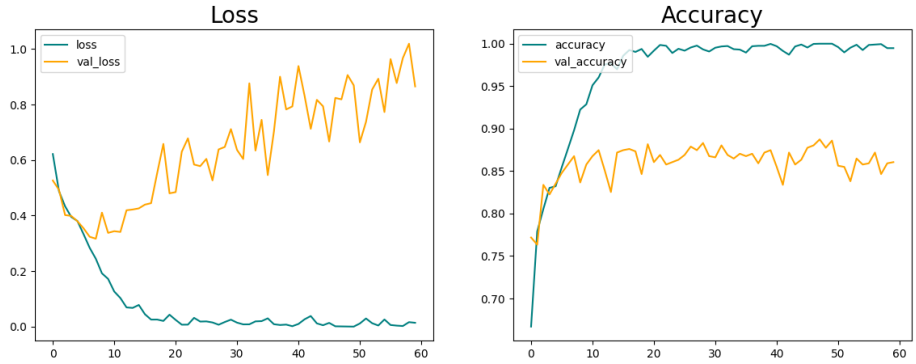

Figure 11: While there is a gap between the validation and training loss, the validation accuracy appears to be more or less stable above the 0.85 mark. This could be considered an improvement on the first family of models.

4.2.2 Refining the model

The baseline of the second model already foresees a dropout layer to prevent overfitting. Since the problem appears to be persistent, other solutions are tried on top of the dropout layer.

First, L2 regularization is applied to the first dense layer of the second model, while keeping everything else equal. Its results are the following:

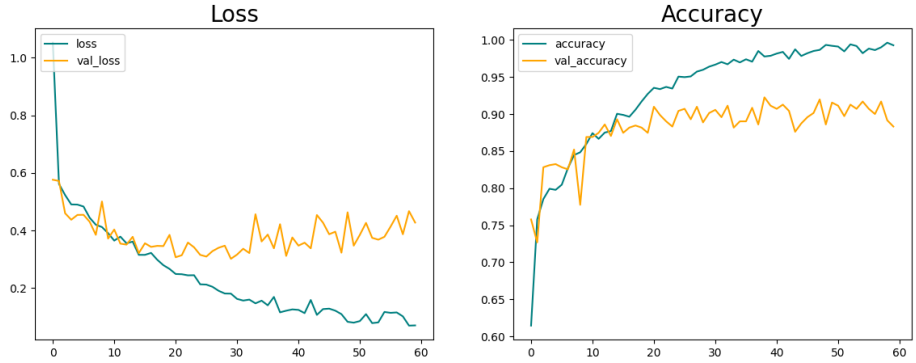

Figure 12: There is a drastic change in the gap between training and validation loss compared to the previous model. Validation accuracy appears to be more or less stable (especially after the 20th epoch) between 0.85 and 0.90

Alternatively, a model where BN is applied instead of L2 regularization is trained on the data. A BatchNorm layer is added after each Pooling layer and after the Dropout Layer. The results are the following:

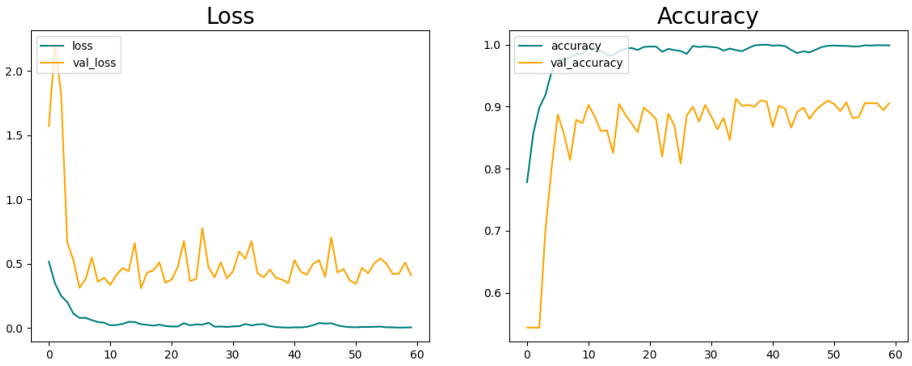

Figure 13: There is a systematic gap between training and validation loss, although it is not a drastic one. Validation accuracy appears less stable compared to the previous model, but it is contained between 0.80 and 0.90

Finally, BN is combined with L2 regularization. The architecture is the following:

- **Conv2D**, 16 filters, (3,3) kernel size, ReLU activation function
- **MaxPooling2D**
- **BatchNormalization**
- **Conv2D**, 32 filters, (3,3), ReLU
- **MaxPooling2D**
- **BatchNormalization**
- **Conv2D**, 64, (3,3), ReLU
- **MaxPooling2D**
- **BatchNormalization**
- **Conv2D**, 128, (3,3), ReLU
- **MaxPooling2D**
- **BatchNormalization**
- **Flatten**
- **Dense**, 256 units, L2 regularization with $\lambda = 0.01$

- **Dropout**, $p = 0.5$
- **Dense** 1 unit, Sigmoid

Its results are the following:

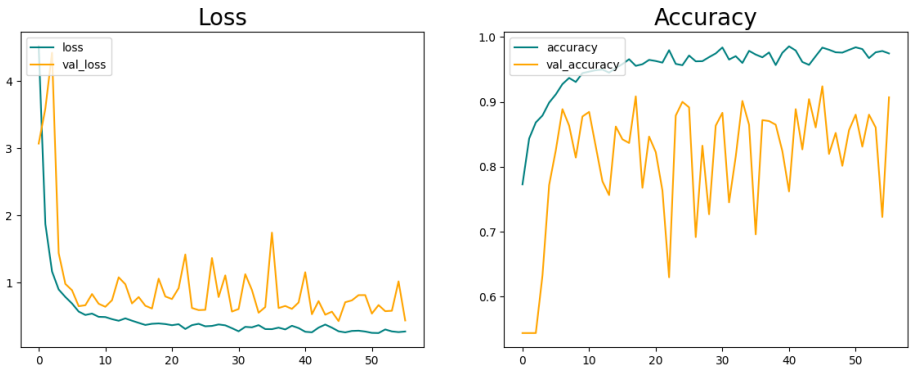

Figure 14: It can be observed that while the difference betweent training and validation loss is reduced, the validation loss (and therefore accuracy) is strongly unstable. Validation accuracy exhibits positive peaks above 0.90 and negative peaks around 0.65.

4.3 Third model

For the final architecture I try to implement a network inspired by the VGG-16 architecture. This architecture was proposed by the Visual Geometry Group at Oxford University, and presented 13 convolutional layers and 3 dense layers. In the convolutional layers, same-padding is introduced. The architecture (restructured for our classification task) is the following:

1. **Conv2D**, 32, (3,3), ReLU, same-padding
2. **Conv2D**, 32, (3,3) ReLU, same-padding
3. **MaxPooling2D**
4. **Conv2D**, 64, (3,3), ReLU, same-padding
5. **Conv2D**, 64, (3,3) ReLU, same-padding
6. **MaxPooling2D**
7. **Conv2D**, 128, (3,3), ReLU, same-padding
8. **Conv2D**, 128, (3,3) ReLU, same-padding
9. **MaxPooling2D**

10. **Conv2D**, 256, (3,3), ReLU, same-padding
11. **Conv2D**, 256, (3,3) ReLU, same-padding
12. **MaxPooling2D**
13. **Flatten**
14. **Dense**, 512 units, ReLU
15. **Dropout**, $p = 0.5$
16. **Dense**, 256 units, ReLU
17. **Dropout**, $p = 0.5$
18. **Dense**, 1, Sigmoid

Unfortunately the results were sub-par, even compared to the previous models. They can be observed in the following graph:

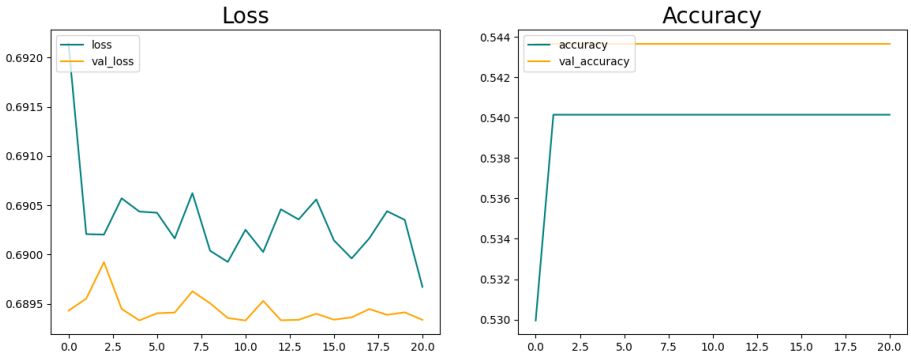

Figure 15: Evidently, the algorithm gets stuck in a local minimum and becomes unable to improve the accuracy.

Thus, I decide to modify heavily the architecture, removing the "double" convolutional layers. Thus, the final model becomes:

1. **Conv2D**, 32, (3,3), ReLU, same-padding
2. **MaxPooling2D**
3. **Conv2D**, 64, (3,3), ReLU, same-padding
4. **MaxPooling2D**
5. **Conv2D**, 128, (3,3), ReLU, same-padding
6. **MaxPooling2D**

7. **Conv2D**, 256, (3,3), ReLU, same-padding
8. **MaxPooling2D**
9. **Flatten**
10. **Dense**, 512 units, ReLU
11. **Dropout**, $p = 0.5$
12. **Dense**, 256 units, ReLU
13. **Dropout**, $p = 0.5$
14. **Dense**, 1, Sigmoid

The results are the following:

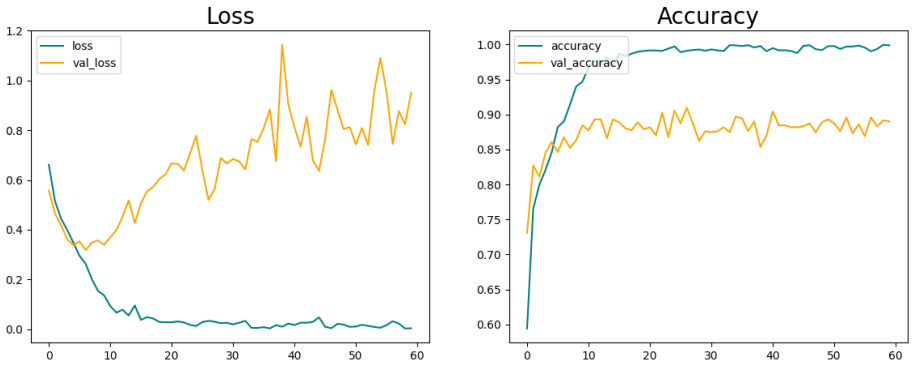

Figure 16: There is a large gap between training and validation loss. However, validation accuracy is more or less stable between 0.85 and 0.90, especially after the 10th epoch.

5 Hyperparameter tuning

I decided to carry out hyperparameter tuning on the first two models. In particular, I will focus on the variation of the first model that includes Dropout, and the variation of the second model that includes Dropout and L2 regularization. I carry out hyperparameter tuning manually, although solutions like keras-tuner exist.

To optimize the values of the hyperparameters, I define discrete options for each of them and create all possible combinations. Then, I train the model with all combinations of hyperparameters. I select as 'optimal' parameter values the ones that result in the highest average validation accuracy across the last five training epochs.

I first focused on the following hyperparameter values, which comprise 72 combinations.

- learning rate = 0.005, 0.001, 0.0005[4]
- batch size = 32, 64
- dropout probability = 0.5, 0.7, 0.8, 0.9
- λ for L2 regularization = 0.1, 0.01, 0.001

After performing this tuning, I focused on optimizing the number of filters for each convolutional layer. In particular, I used the following combinations of filters:

- filters_layer1 = [16, 32, 64]
- filters_layer2 = [32, 64, 128]
- filters_layer3 = [64, 128, 256]
- filters_layer4 = [64, 128, 256]

The number of filters increases with the depth of the network. This is because early layers should capture low-level features, while the patterns identified by deeper layers are more complex. Indeed, many successful architectures, like VGG and ResNet exhibit this pattern.

Lastly, I focus on the units of the dense layers, where the possible values were 64, 128, 256.

I realize this is not an optimal solution, as my "grid" search is mostly limited to three possible values for each hyperparameter. Another fundamental mistake in my hyperparameter tuning is the lack of cross-validated estimates. Unfortunately performing cross-validation was not sustainable for my hardware.
In training the models for the tuning I set the epochs = 30 with an early stop callback with patience = 10 epochs.

5.1 Tuning the first model

I tune the first model with the dropout layer. The following values are obtained:

- Learning rate = 0.001
- Batch size = 64,
- Dropout rate = 0.9

In the last 5 epochs of training, this model has:

[4]Actually, I first implemented hyperparameter tuning with learning rate values = 0.01, 0.001, 0.0001. However, I noticed only the results with lr = 0.001 were satisfactory. For this reason, I carried out hyperparameter tuning again, with these different potential values. I also dropped the possibility of batch size = 16 and added the dropout probability = 0.8, since it appeared to have good results on the first model.

- Average validation accuracy: 0.895
- Std. dev. of accuracy: 0.004

With these hyperparameters, the best filter values are 32, 64, 64 respectively. The final version of the first model exhibits the following performance:

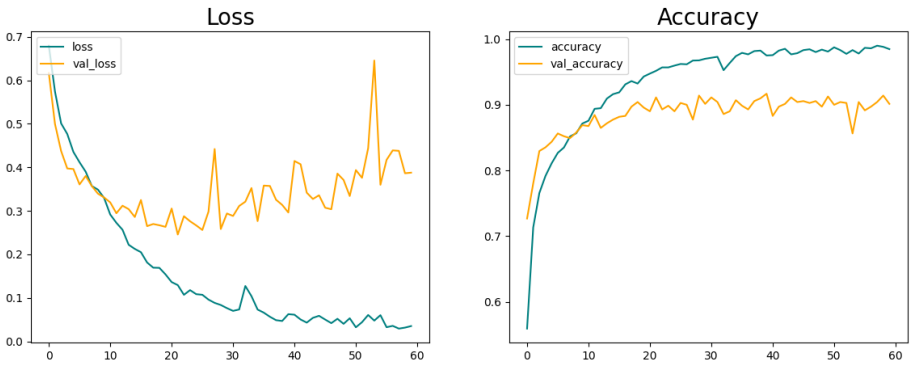

Figure 17: Results of the fully tuned version of the first model.

5.2 Tuning the second model

The second model has been optimized on the learning rate, the dropout probability, the λ for L2 regularization and in batch size. The number of filters for the convolutional layers are 32, 64, 128, 256 respectively. The single dense layer has 256 nodes.

The following hyperparameter values are obtained:

- Learning rate=0.001
- Batch size=32
- Dropout rate=0.7
- L2 lambda=0.01

In the last five epochs of training, the model exhibited a:

- Avg. validation accuracy: 0.904
- Std. dev. of validation accuracy: 0.004

The final version of the second semi-tuned model has the following performance on the training set:

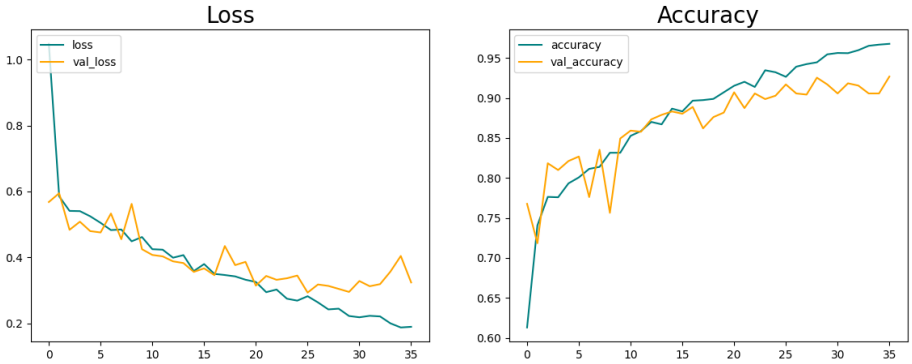

Figure 18: Results or the semi-tuned version of the second model.

6 Final results and Cross Validated Risk Estimates

We now focus on the following three models:

- Tuned first model with dropout
- Semi-Tuned second model with dropout and L2 regularization
- Un-tuned third model

First, 5-fold cross-validation is carried out to compute risk estimates with zero one loss. I get the following results:

Validation Loss	Tuned first model	Semi-tuned second model	Un-tuned third model
1st fold	0.04012	0.08025	0.10242
2nd fold	0.02006	0.05596	0.01689
3rd fold	0.01055	0.00422	0.00739
4th fold	0.00528	0.00634	0.00001
5th fold	0.00317	0.01162	0.00001
Average	0.0158	0.03168	0.02534

Then, we apply the models to the test set. As evaluation metrics, I investigate: recall, precision and accuracy. The following are the results of the models:

Model	0-1 Loss	Accuracy	Precision	Recall
1st tuned	0.464	0.901	0.92	0.88
2nd semi-tuned	0.344	0.907	0.92	0.87
3rd un-tuned	0.326	0.855	0.90	0.86

Table 1: Results on the test set of all the three final models. The performances are similar. It appears that tuning can make up for limited model complexity.

7 Conclusion

The models increase in complexity from the first to the third. However, they decrease in number of parameters tuned: the first one had all its hyperparameters tuned, the second only half, whereas the third didn't undergo hyperparameter tuning at all. Their results are pretty similar, as if the additional tuning can make up for the lack of complexity.
In the end, the models predict correctly 9 out of 10 new images of chihuahuas or muffins. It is probably possible to increase the performance by tuning the more complex models.
While a more complex architecture provided more accurate estimates, hyperparameter tuning has been instrumental in reducing overfitting. For instance, in the first model it could be clearly observed how the increase in dropout probability reduced the gap between training and validation accuracy. L2 regularization on the dense layers helped as well, while Batch Normalization did not prove beneficial.

References

[1] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. pmlr. 2015, pp. 448–456.

[2] *Keras documentation: Dense layer*. URL: `https://keras.io/api/layers/core_layers/dense/`.

[3] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: `1412.6980 [cs.LG]`.

[4] Sungheon Park and Nojun Kwak. "Analysis on the dropout effect in convolutional neural networks". In: *Computer Vision–ACCV 2016: 13th Asian Conference on Computer Vision, Taipei, Taiwan, November 20-24, 2016, Revised Selected Papers, Part II 13*. Springer. 2017, pp. 189–204.

[5] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: `1409.1556 [cs.CV]`.

[6] Nitish Srivastava et al. “Dropout: a simple way to prevent neural networks from overfitting”. In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[7] IBM website. *What are convolutional neural networks?* URL: `https://www.ibm.com/topics/convolutional-neural-networks`.