# Optimization of Convolutional Neural Networks for binary image classification

Rossi Lorenzo (Student ID 982595)

University of Milan, MSc in Data science and Economics

Project for the module of Statistical Methods for Machine Learning

Course of

Machine Learning, Statistical Learning, Deep Learning and Artificial Intelligence

15 November 2022

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious 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 or any other person for assessment on this or any other course of study.

**Abstract**

In this work different Convolutional Neural Network architectures have been tested for a binary image classification task. Models have been tested with different layer structures and parameters in order to find the best one. Risk estimates of the final predictive model have been computed with K-Fold Cross Validation (using 5 folds) in order to obtain a robust estimate of the final predictive performance. In the end, the best results were found with a Convolutional Neural Network with 12 convolutional layers divided in six blocks made by two layers each and batch size of 32, bringing the highest accuracy and lowest loss among the other architectures.

## 1 Introduction

Simple machine learning algorithms work very well on a wide variety of subjects. However, they don't succeed in central AI problems, such as recognizing speech patterns or recognizing objects. The development of deep learning was motivated also by the need to find algorithms that are able to solve those tasks. Deep learning provides many solutions in the aforementioned fields, such as image analysis and natural language processing, and it is seen as a key method for various future applications, especially regarding AI development.

This paper will focus on the problem of Image Classification, which consists in the task of attempting to comprehend an entire image as a whole. The goal is to classify the image by assigning it to a specific label. Typically, Image Classification refers to images in which only one object appears and is analyzed. In contrast, object detection involves both classification and localization tasks, and is used to analyze more realistic cases in which multiple objects may exist in an image using different network architectures. However, this last application won't be the focus of this work.

For this project the task was to develop different Neural Network architectures in order to find the best model for a binary classification task related to cats and dogs images. The first part of this work will cover the pre-processing of data and how the dataset is structured. Consequently, the theoretical foundations of deep learning will be explained, with a focus on the functioning of Convolutional Neural Network. The following step will be the analysis of the network architectures that has been developed and tested. Finally, the results of the analysis will be discussed, along with some considerations on to improve the performances of the aforementioned models.

# 2 Dataset and data preparation

The dataset consists in 25.000 images of cats and dogs (12.500 for each animal class) with different size. It is a popular dataset used for many competitions involved in computer vision tasks[1]. Data preparation and processing were done using the OpenCV library for Python and this part of the work consisted in the following steps: all the images have been converted from JPEG to RGB and scaled down to 100x100 pixels in order to obtain a dataset of images with the same dimensions. Moreover, colors were converted to black and white (gray-scale), because the challenge with images having multiple color channels is that we have huge volumes of data to work with which makes the process computationally intensive and, unless color is a principal discriminant in the classification task, it doesn't always lead to better results. Moreover, to improve model performances the gray-scale values have been re-scaled from the [0, 255] range to the [0, 1] range.

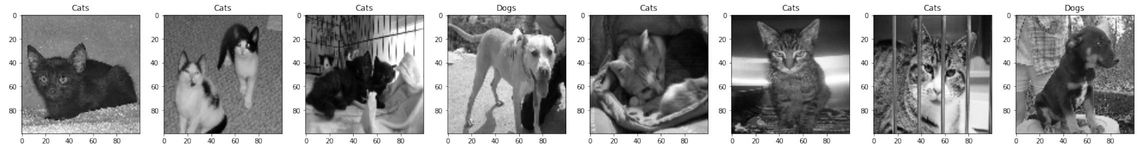


Figure 1: Sample of processed images.

# 3 Theoretical Framework

## 3.1 Deep Learning and Neural Networks

Neural Networks are a class of predictors inspired by the structure of the brain, which consists (oversimplifying) in a large and complex network of interconnected computing devices, the neurons, through which the brain can perform highly complex computations. The simplest neural network architecture is the feedforward neural network.

### 3.1.1 Graph Structure of a Network

The definition of network comes from the fact that they are typically represented as a directed acyclic graph $G = (V, E)$[Fig. 2], where each node (or unit) $j \in V$ computes a function $g(\mathbf{v})$ where $\mathbf{v}$ is the output of the nodes $i$ such that $(i, j) \in E$. The nodes are divided in three subsets, such that $V = V_{in} \cup V_{hid} \cup V_{out}$ where $V_{in}$ refers to the input nodes (with $|V_{in}| = d$ where $d$ is the number of input features or dimensions), $V_{hid}$ are the hidden nodes which have both incoming and outgoing edges and finally $V_{out}$ are the output nodes (with $|V_{out}| = n$ where $n$ is the number of output features, e.g. for a binary classification task $n = 2$).

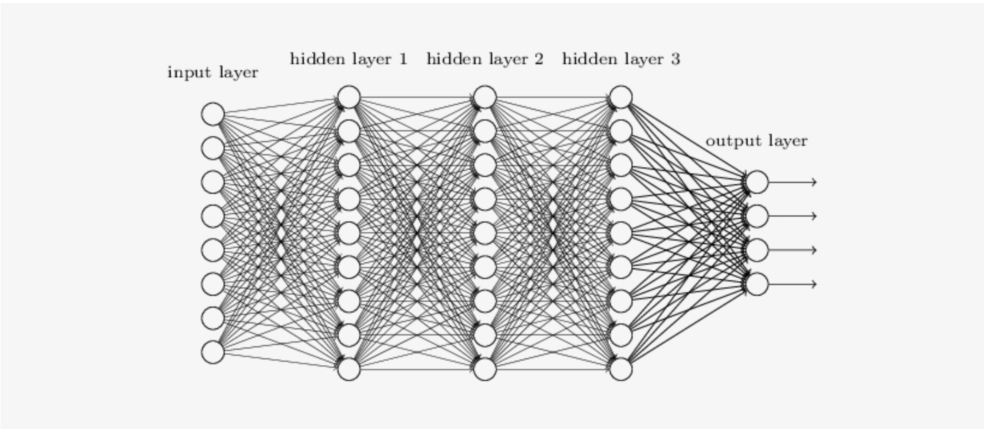


Figure 2: Graphical structure of a network.

[1] https://www.kaggle.com/c/dogs-vs-cats

Networks are composed by many different functions. For example, we might have three functions connected in a chain to form the following system:

$$g^{(1)}, g^{(2)}, g^{(3)} \rightarrow g(\mathbf{v}) = g^{(3)}(g^{(2)}(g^{(1)}(\mathbf{v}))) \tag{1}$$

Where each of the $g^{(x)}$ is called a "layer", thus having $g^{(1)}$ as first layer, $g^{(2)}$ as second layer and so on. Each layer is composed by many nodes that compute the functions. Moreover, to every edge $(i, j) \in E$ a parameter (called "weight)" $w_{i,j} \in \mathbb{R}$ is associated. Considering all the other edges, this creates a weight matrix $W = |V| * |V|$. This gives the idea of networks as a complex sequence of matrix operations.

### 3.1.2 Non-linearity and activation functions

The goal of a network is to approximate some function $f$*and in order to understand its functioning it is important to recall how linear models work. In the case of a linear classifier $y = f$*$(\mathbf{x})$ the function maps $\mathbf{x}$ to a class $y$. However, linear models have the defect to depend only on linear functions. Neural Networks solve the problem of learning from non-linearity by applying a non-linear transformation to $\mathbf{x}$:

$$y = g(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + \mathbf{c}) \tag{2}$$

Where $\mathbf{x}$ is the input, $\mathbf{w}$ is the weight (or parameter), $\mathbf{c}$ is the (possible) bias and $\sigma$ is a non-linear function such that $\sigma : \mathbb{R} \rightarrow \mathbb{R}$, and it is called "activation function". There is broad range of activation functions which have to be chosen carefully depending on the task. Each layer has its own activation function. In this work we'll refer to two of the most used functions:

**ReLU**: denoted by $f(z) = max(0, z)$. It is mainly used in input and hidden layers and its advantages comes from the fact that the gradients remain large and consistent in the nodes where it is applied.

**Sigmoid**: denoted by $\sigma(z) = 1/(1 + e^{-z})$. It is commonly used to produce the parameter of a Bernoulli distribution because its range is (0,1). It's mainly applied in binary classification where the task is to determine whether a data point belongs to a class or not and because of this it is usually applied in the output nodes.

Basically, when building a Neural Network one must take into consideration how many layers to include, the number of nodes per layer, how these layers are connected, the choice of the activation functions, i.e. a set of many parameters and structural components that need to be tuned finely in order to find the best learning model.

## 3.2 Convolutional Neural Networks

Convolutional Networks (or ConvNets) are a special kind of neural networks used to process data that has a grid-like topology, such as forecasting time series (which can be considered as a 1-D grid of samples at regular intervals of time) or classifying images, interpreted as a 2-D grid of pixels.

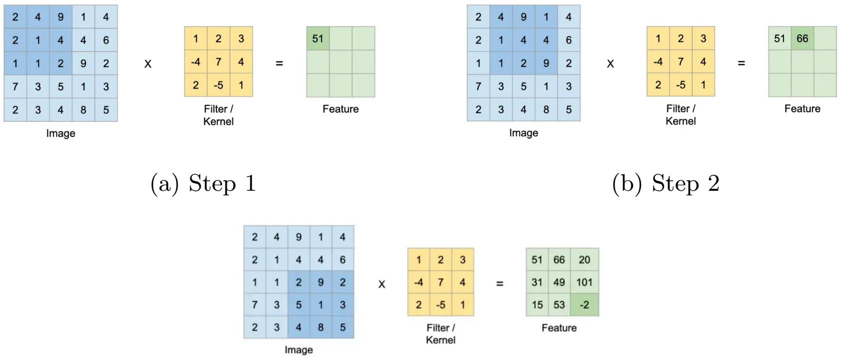
### 3.2.1 Convolutional Layers

The name of this particular structure of neural networks comes from the "convolution", a mathematical linear operation that the network employs instead of general matrix multiplication inside the layers. Convolution is used to obtain more accurate estimates by averaging The operation is denoted by the following equation:

$$s = (x * w) \tag{3}$$

Where $x$ is the input and $w$ is the kernel and they are usually multidimensional arrays of data and parameters respectively. The kernel, or feature detector, can be set to different dimensions, for example, 3x3. The kernel works as a filter and to perform convolution, it goes over the input image (in case of image classification), doing matrix multiplication element after element [Fig. 3]. Usually, these operations are performed with respect to $t$ (in this case we write $s(t) = (x * w)(t)$), which is an index and integer value. It can be the position of a pixel or a date of measurement and in this case the kernel performs a weighting operation that gives more weight to the values that are "closer" to the one we're estimating.

Another element that composes a convolutional layer is the stride. It represents the slide of the kernel when shifting from one tile of data points to another on which the convolution is computed. When the stride is 1 then we move the filter one pixel at a time. For computer vision and image processing task, it's unusual to use a stride bigger than 1.



(a) Step 1

(b) Step 2

(c) Step $n$

Figure 3: Convolution operation.

### 3.2.2 Pooling Layers

Consequently to Convolution there is Pooling: a pooling function is usually applied after the convolution layer, in order to further modify and down-sample the output of the layer. The filter of a pooling layer is always smaller than a feature map. In this case it has been used the Max Pooling function with a 2*2 window, which reports the maximum output in a square-shaped neighborhood [Fig. 4]. Pooling operations help to make representations almost invariant from translation of the input, reducing computational costs and allowing the learning of high-level pattern by deeper layers of the network.

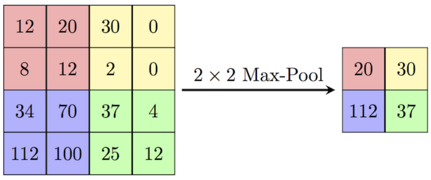


Figure 4: Pooling operation.

### 3.2.3 Flattening and Dense Layers

Blocks of Convolution and MaxPooling layers are usually followed by a Flatten layer, which transforms the 3D or 2D input into a one-dimensional vector. The one-dimensional vector created by the Flatten layer can then pass through one or a series Dense Layers, also referred to as Fully Connected layers. These layers can learn complex relationships between the high-level features learnt by the preceding blocks of Convolution and MaxPooling Layers. This layer provides the model with the ability to finally understand images: there is a flow of information between each input pixel and each output class. The hyperparameters of this layer are the number of nodes and the activation function, which will be Sigmoid since for this task we're dealing with binary classification.

Generalizing, we can conclude that the typical structure of a ConvNet is the following [Fig. 5]:

1. **Convolutional layer** (including Input layer)

2. **Pooling layer**

3. **Flattening**

4. **Fully connected layer** (including Output layer)

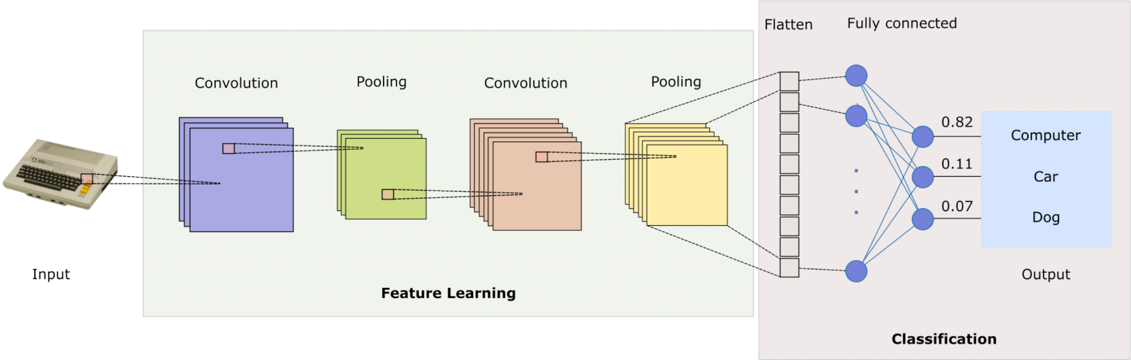


Figure 5: ConvNet structure in case of classification.

The aforementioned layers are not the only ones that compose ConvNets. There are other two types of layers which are not essential but they help improving learning performances. These layers will be implemented in this work.

### 3.2.4 Dropout

Dropout layers are placed after the Pooling layer and they function as masks that nullify the contribution of some neurons towards the next layer and leaves unmodified all others. We can apply a Dropout layer to the input vector, in which case it nullifies some of its features; but we can also apply it to a hidden layer, in which case it nullifies some hidden neurons. Dropout layers are important in training ConvNets because they prevent overfitting on the training data. If they are not present, the first batch of training samples influences the learning in a disproportionately high manner. Dropout can also be applied after a Dense layer.

### 3.2.5 Batch normalization

Batch normalization layers are placed after the Convolutional layer and before the Pooling layer, are proposed as a technique to help coordinate the update of multiple layers in the model by scaling the output of the layer, specifically by standardizing the activations of each input variable per mini-batch, such as the activations of a node from the previous layer (standardization refers to rescaling data to have a mean of zero and a standard deviation of one, e.g. a standard Gaussian). Standardizing the activations of the prior layer means that assumptions the subsequent layer makes about the distribution of inputs during the weight update will not change, or at least not dramatically. This has the effect of stabilizing and speeding-up the training process.

# 4 Tested Models and Parameters

## 4.1 The VGG Architecture

The base architecture that has been implemented in this problem takes inspiration from the VGG architecture described in the research paper by Simonyan and Zisserman. VGG stands for Visual Geometry Group; it is a standard deep ConvNet architecture with multiple layers[Fig. 6]. The "deep" refers to the number of layers with VGG-16 or VGG-19 consisting of 16 or 19 convolutional layers. The VGG presents a structure made of blocks of two and three convolutional layers. Each pair or triplet of layers is then followed by a pooling layer, ending in three fully connected layers after flattening.

Although the exact same number of layers and parameters have not been replicated for computational costs, the models presented in this work present a similar structure, based on blocks of convolutional layers. Six models have been developed and tested, each with a different composition of blocks and number of layers.

Note: all the models have been developed and trained using the TensorFlow framework.

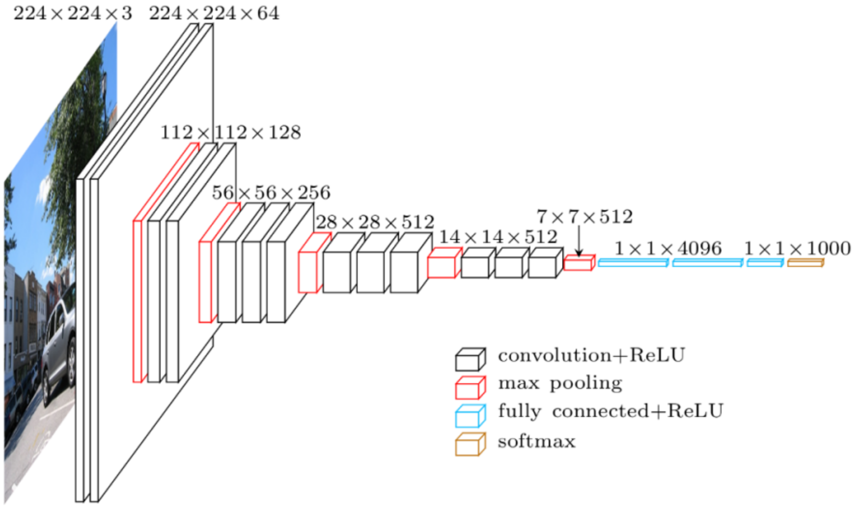


Figure 6: The VGG architecture.

## 4.2 Architecture variants and layers

Three variants of the network structure applied to the models have been developed and each of the six models have been trained and tested using these variants. This was done in order to study how adding different regularization layers (such as dropout and batch normalization) affects learning performance. The variants are the following:

1. **Base architecture** [Fig. 7.a]: it consists in blocks made of two consecutive convolutional layers, followed by a pooling layer and terminating in a Flatten and two Dense layers. For the convolutional layers, the kernel size was set to 3*3 and the stride was set to 1. The pooling layer was set to Max Pooling with a 2*2 window. To each of the convolutional layer have been applied the ReLU activation function, as well as to the first Dense layer (which also presents a different number of nodes depending on the model) while for the final Dense layer (which returns the output) the Sigmoid activation function has been chose, with a number of nodes equal to 1.

2. **Architecture + Dropout** [Fig. 7.b]: Dropout layers have been added to the previous architecture, more specifically, a layer with dropout rate of 0.25 have been applied after each pooling layer and another one with dropout rate of 0.5 has been inserted after the first Dense layer.

3. **Architecture + Dropout + Batch Normalization** [Fig. 7.c]: this third variant adds one Batch Normalization layer after each convolutional layer (and before the pooling layer) in order to stabilize the learning process and verify the eventual improvements of the models.

Note: the architectures shown in Fig. 7 present the number of layers of model 1. It is an explicative example of how the architecture variants apply to any model presented in this paper.

## 4.3 Models

We'll then explore how the models are built and their layer blocks composition.

1. **Model 1**: one block made by a pair of convolutional layers with 64 nodes each, followed by flattening and two Dense layers with 128 and 1 (the output layer) nodes respectively.

2. **Model 2**: two blocks made by one pair of convolutional layers each, with increasing number of nodes per block {[32,32],[64,64]}, followed by flattening and two Dense layers with 128 and 1 (the output layer) nodes respectively.

3. **Model 3**: three blocks made by one pair of convolutional layers each, with increasing number of nodes per block {[32,32],[64,64],[128,128]}, followed by flattening and two Dense layers with 256 and 1 (the output layer) nodes respectively.

4. **Model 4**: four blocks made by one pair of convolutional layers each, with increasing number of nodes per block {[32,32],[64,64],[128,128],[256,256]}, followed by flattening and two Dense layers with 512 and 1 (the output layer) nodes respectively.

5. **Model 5**: five blocks made by pair of convolutional layers each, with increasing and then decreasing number of nodes per block {[32,32],[64,64],[128,128],[256,256],[128,128]}, followed by flattening and two Dense layers with 512 and 1 (the output layer) nodes respectively.

6. **Model 6**: six blocks made by one pair of convolutional layers each, with increasing and then decreasing number of nodes per block {[32,32],[64,64],[128,128],[256,256],[128,128],[64,64]}, followed by flattening and two Dense layers with 512 and 1 (the output layer) nodes respectively.

To all the convolutional layers and the first dense layers the ReLU function has been applied, while the Sigmoid function was used on the last Dense layer. Plus, as already explained in 4.2, a 3*3 kernel was applied to all the convolutional layers, as well as a Max Pooling with 2*2 shape layer after each block.

### 4.4 Hyperparameters and Compilation

The initial hyperparameters and model compilation setting is the following:

1. **Epochs**: 50

2. **Battch size**: 64

3. **Optimizer**: Adam

4. **Metrics**: Binary Accuracy

5. **Loss**: since we're dealing with binary classification, the choice for the loss function is the Binary Crossentropy, which computes the average of the log losses of predicted class probabilities (the log is used as a penalization term for the classifier):

$$Loss = -\frac{1}{N}\sum_{i=1}^{N} y_i log(p(y_i)) + (1-y_i) log(1-p(y_i)) \tag{4}$$

## 5 Results

### 5.1 Evaluation of Architectures and Models

We know discuss the results of the networks. It is clear that the base architecture fails at learning in all the models [Tab. 1, Fig. 8], reaching high levels of loss and the maximum accuracy is the one in Model 5. This network presents only the convolutional and pooling layers without any further regularization. To notice is the difference between the training and validation loss, where the latter steeply increases after a few epochs. This means that the current architecture tends to overfit the data in the training.

The second architecture variant which include Dropout layers presents better results for Models 3, 4 and 5 [Table 2 and Fig. 9], which reach an accuracy of more than 0.9 and a loss slightly higher than than 0.2. Model 6 still fails to improve its learning performance, while Models 1 and 2 still present overfitting.

The third architecture, as expected, presents the best scores [Tab. 3, Fig. 10]. Despite the first two models, validation loss and accuracy behave similarly in the other four. Model 6 outperforms the others in terms of loss and accuracy, even though Model 4 and 5 still present highly performing results. Even overfitting in the first three models tends to decrease. These results confirm the positive effect that regularization through dropout and batch normalization layers has on the networks.

### 5.2 Hyperparameter Optimization

Following the previous results, the three best models (Model 4, 5 and 6) have been selected in order to search for better performances by using different batch sizes. The batch size defines the number of samples that will be propagated through the network and it has been noted that using a larger batch that the quality of the model significantly decreases. In this case, the other batch sizes that will be tested are 32, 128 and 256. As expected again, a lower batch size of 32 brings better results than the initial batch of 64. Model 6 is still the most accurate one.

Another aspect that can improve or decrease model performance is the choice of the optimizer. Other two optimizers have been tested and compared: SGD (with standard learning rate of 0.1, momentum if 0.9 and decay equal to 0.1/50, i.e. the number of epochs; RMSprop (with learning rate of 0.0001). However, using these different optimizers didn't bring better results [Tab. 5]. From the comparison of the different models, we can conclude that the most accurate one is Model 6 with batch size 32 and Adam optimizer.

### 5.3 5-Fold Cross validation using Zero-One Loss

Model 6, which presented the best loss and accuracy after optimization, was trained (with the best hyperparameters) one last time using 5-Fold Cross Validation. This time, instead of the Binary Crossentropy, the loss was computed according to the zero-one loss. Since TensorFlow doesn't present a way to compute automatically this type of loss, predictions have been rounded in order to become 0 or 1 (since the output of the sigmoid function is a probability in (0,1)) and then converted into a Numpy array, allowing for the computation of the zero-one loss with the actual values [Tab. 6]. The reason why the averaged score for the loss is lower than the Binary Crossentropy is that the latter adds a penalization for each wrong prediction. The accuracy is slightly lower than in the model with Binary Crossentropy loss but still manages to reach an highly performing score.

## 6 Conclusion

This project was aimed to test different Neural Network architectures on a binary classification task and find the best predictive model. The resulting tests applied to the six network structures that have been used confirm what is known from the state of the art: deeper networks, composed by many layers, tend to obtain better performances than "shallow" ones. Moreover, using regularization and optimization techniques such as adding Dropout and Batch Normalization layers can help the network to improve its learning. Finally, one must not forget that hyperparamenter tuning is crucial for network functioning. To improve the results even more, it may be considered to use Data augmentation layers, add other convolutions or change the number of nodes in each layer. Neural Networks are not pre-built models and they must be carefully tuned depending on the task.

## Bibliography

Bengio, Y., Courville, A., Goodfellow, I. (2016). *Deep Learning*, The MIT Press.

Simonyan, K., Zisserman, A. (2015). *Very Deep Convolutional Networks for Large Scale Image Recognition*, conference paper at ICLR 2015

## Appendix

### Tables

Table 1: Base architecture

| Model | Accuracy | Loss |
|---|---|---|
| Model 1 | 3.220 | 0.75 |
| Model 2 | 2.346 | 0.807 |
| Model 3 | 1.068 | 0.890 |
| Model 4 | 0.634 | 0.879 |
| Model 5 | 0.489 | 0.891 |
| Model 6 | 0.693 | 0.504 |

Table 2: Dropout layers

| Model | Accuracy | Loss |
|---|---|---|
| Model 1 | 1.687 | 0.642 |
| Model 2 | 0.772 | 0.796 |
| Model 3 | 0.275 | 0.916 |
| Model 4 | 0.232 | 0.921 |
| Model 5 | 0.216 | 0.914 |
| Model 6 | 0.693 | 0.495 |

Table 3: Dropout and batch normalization layers

| Model | Accuracy | Loss |
|---|---|---|
| Model 1 | 1.093 | 0.768 |
| Model 2 | 0.560 | 0.862 |
| Model 3 | 0.321 | 0.916 |
| Model 4 | 0.272 | 0.935 |
| Model 5 | 0.274 | 0.938 |
| Model 6 | 0.227 | 0.942 |

Table 4: Model 4, 5, and 6 for different batch sizes

| Model | Batch 32 | Batch 64 (Initial) | Batch 128 | Batch 256 |
|---|---|---|---|---|
| Model 4 | 0.230, 0.945 | 0.272, 0.935 | 0.227, 0.927 | 0.230, 0.932 |
| Model 5 | 0.243, 0.938 | 0.274, 0.938 | 0.308, 0.927 | 0.279, 0.932 |
| Model 6 | 0.210, 0.945 | 0.227, 0.942 | 0.248, 0.940 | 0.277, 0.929 |

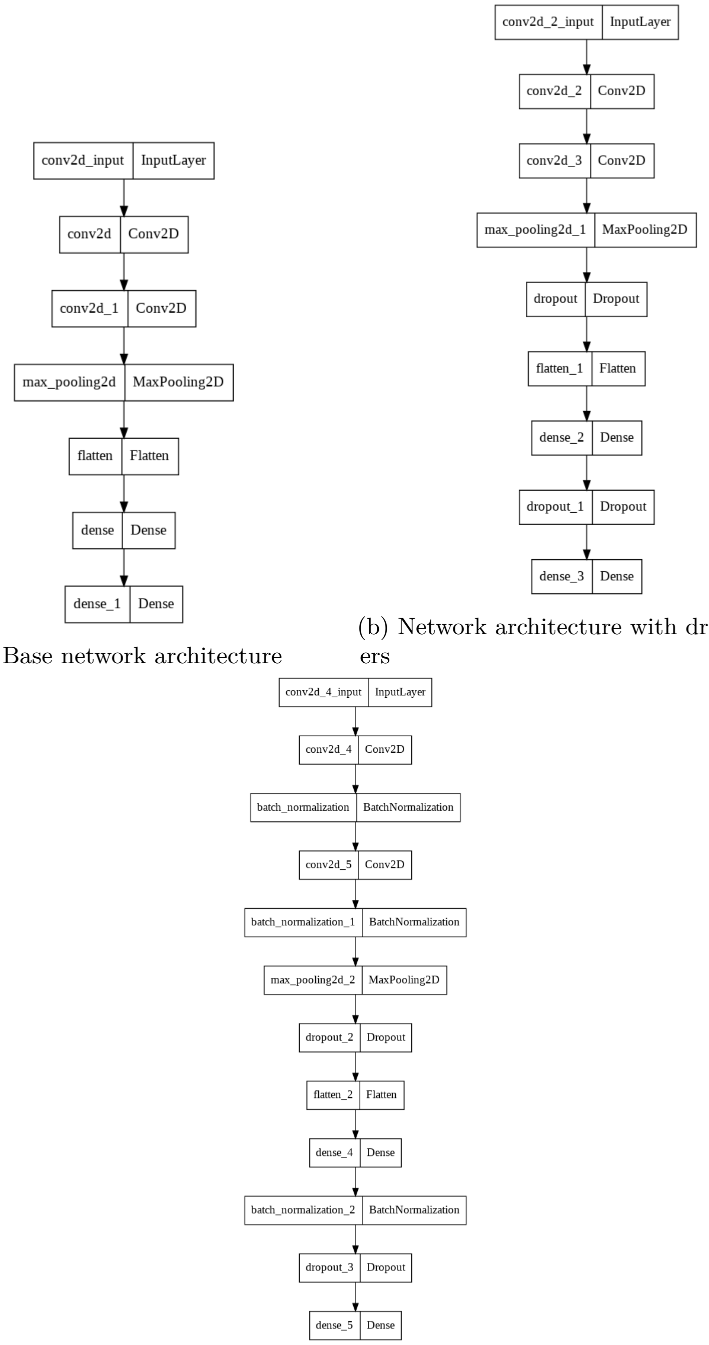
Table 5: Model 4, 5, and 6 for different optimizer

| Model | Adam | SGD | RMSprop |
|---|---|---|---|
| Model 4 | 0.230, 0.945 | 0.331, 0.917 | 0.354, 0.920 |
| Model 5 | 0.243, 0.938 | 0.371, 0.911 | 0.303, 0.924 |
| Model 6 | 0.210, 0.945 | 0.392, 0.894 | 0.315, 0.921 |

Table 6: 5-Fold Cross Validation

| Model | Loss | Accuracy |
|---|---|---|
| Fold 1 | 0.066 | 0.933 |
| Fold 2 | 0.065 | 0.934 |
| Fold 3 | 0.065 | 0.934 |
| Fold 4 | 0.059 | 0.940 |
| Fold 5 | 0.059 | 0.940 |
| Avg. scores | 0.063 | 0.936 (+-0.003) |

## Images

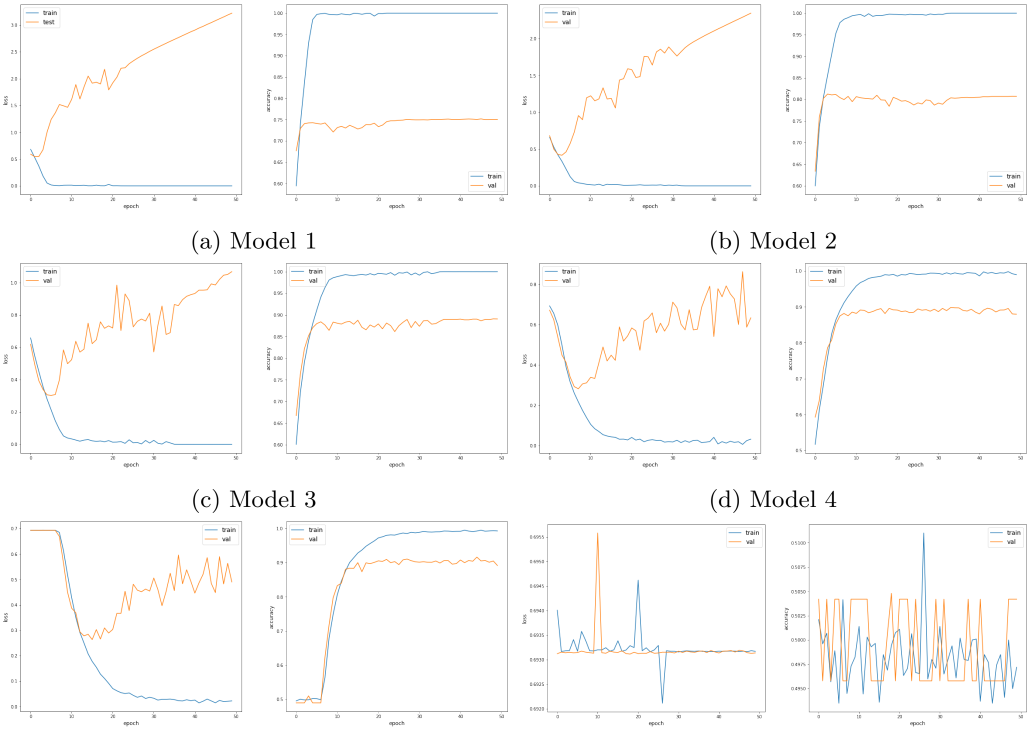


(a) Base network architecture

(b) Network architecture with dropout layers

(c) Network architecture with dropout layers and batch normalization

Figure 7: Network architectures variants applied to the models

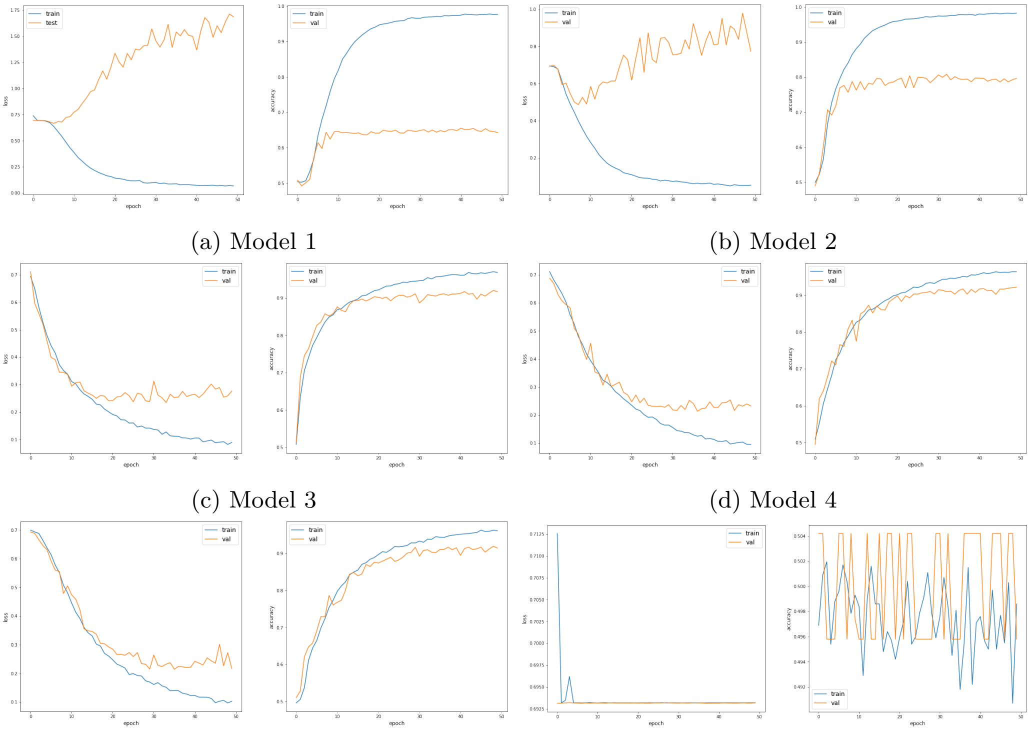


(a) Model 1 (b) Model 2

(c) Model 3 (d) Model 4

(e) Model 5 (f) Model 6

Figure 8: Accuracy and loss of the six models with base architecture.

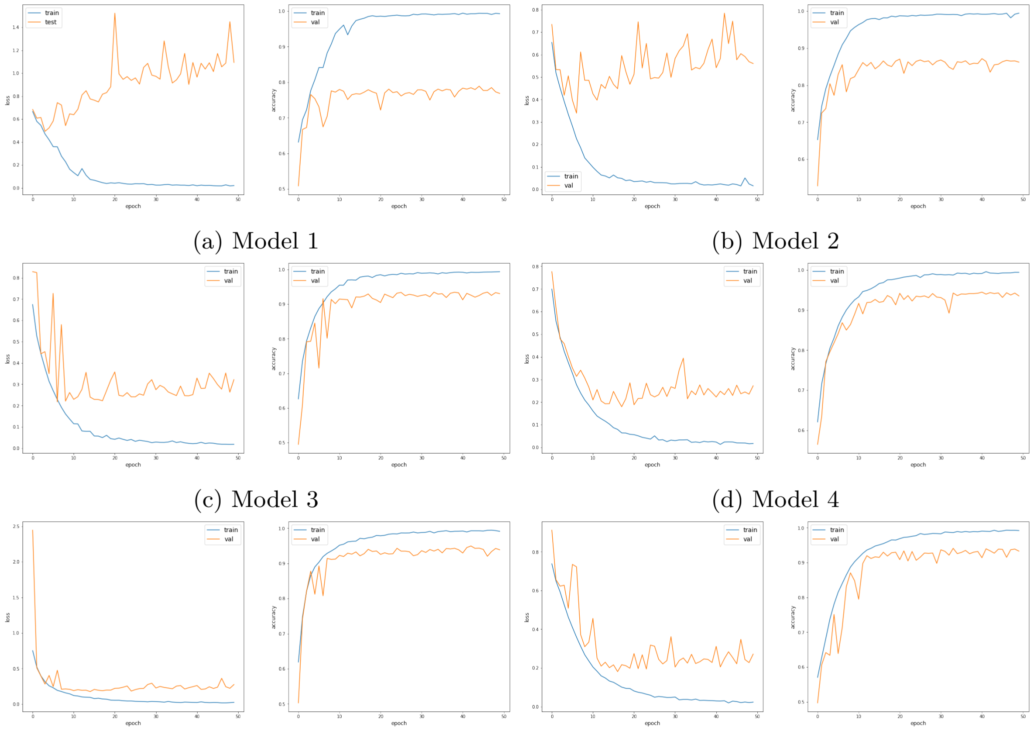


(a) Model 1 (b) Model 2

(c) Model 3 (d) Model 4

(e) Model 5 (f) Model 6

Figure 9: Accuracy and loss of the six models with dropout layers.



(a) Model 1

(b) Model 2

(c) Model 3

(d) Model 4

(e) Model 5

(f) Model 6

Figure 10: Accuracy and loss of the six models with dropout layers and batch normalization layers.