



# DOG BREED CLASSIFICATION SYSTEM

*Capstone project for the Udacity Machine  
Learning Engineer Nanodegree*

## *What is included*

*This document describes the process of building a dog breed classification system from scratch, including the Data Science and Software Engineering tasks needed for the project*

*Luis Martín  
lumartmartin@gmail.com*

Problem definition .....	2
Data Analysis .....	3
Exploratory Data Analysis.....	3
Data cleaning .....	4
Data transformation and augmentation .....	4
Gaussian Blur.....	5
Superpixels .....	5
Flipping .....	5
Cropping .....	5
Image Rotation .....	5
New data gathering.....	6
Implementation.....	6
Benchmark: The basic notebook.....	6
From Scratch solution .....	6
Transfer Learning Solution.....	6
Techniques and Technologies.....	7
Experimentation framework .....	7
Code and resources.....	8
Experiments and Results .....	9
From-scratch approach .....	9
Experiment 1: Architecture comparison .....	10
Experiment 2: Learning rate comparison .....	10
Experiment 3: Optimizer comparison.....	11
Experiment 4: Augmentation techniques .....	12
Transfer learning approach .....	16
Conclusions.....	17

## Problem definition

**Image classification** is a traditional problem in Artificial Intelligence, belonging to the field of Computer Vision. A problem traditionally considered at the “human level”, which has been a test for different Machine Learning algorithms. More concretely, some of the benchmark problems include **character recognition or OCR** (MNIST<sup>1</sup>), and more recently **person and object identification**, tasks that have been tackled thanks to deep learning algorithms, such as **Convolutional Neural Networks**, with applications in many different fields, from Security to Autonomous Driving<sup>2</sup>.

**Convolutional neural networks** (LeCun, 1989<sup>3</sup>) changed the world. Although showing impressive results back in the XX century, mainly in the recognition of handwritten characters, it wasn't until 2011 were they became vastly popularized, during the era of Deep Learning. CNNs achieved superhuman performance in several tasks of image analysis, winning several image recognition competitions, like the popular IMAGENET challenge<sup>4</sup>.

A **convolutional network** makes use of backpropagation to find the parameters adjustment, but with the peculiarity of applying a convolution operation to the inputs. A convolution is a matrix operation that acts as a filter. The filter is applied to each input example, that is represented as a matrix, and gives us as output a transformation of the original example. In a convolutional network, the parameters that are learned through backpropagation are the numbers contained in the filters.

This problem is proposed as a capstone project candidate by Udacity. We want to make use of **convolutional networks** to classify dog images and figure out their breed. We are presented with a dataset containing images of dogs, and want to learn how to determine, given a new image, the breed of a dog in the image.

We assume that the images to be classified contain one and one only dog. Images without a dog are also not possible. Images are supposed to be colorful, and with a resolution like the ones in the dataset.

As mentioned, we will make use of convolutional neural networks. We will compare a **transfer learning** solution with other one developed **from scratch**.

---

<sup>1</sup> Lecun et al., “Gradient-Based Learning Applied to Document Recognition.”

<sup>2</sup> Al-Qizwini et al., “Deep Learning Algorithm for Autonomous Driving Using GoogLeNet.”

<sup>3</sup> LeCun et al., “Handwritten Digit Recognition with a Back-Propagation Network.”

<sup>4</sup> Krizhevsky, Sutskever, and Hinton, “ImageNet Classification with Deep Convolutional Neural Networks.”

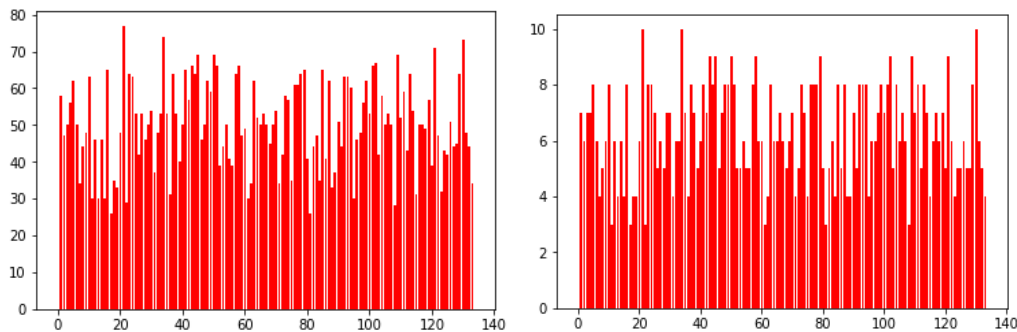
## Data Analysis

### Exploratory Data Analysis

We are provided with a fully categorized set of dog images. This dataset contains a total of 8351 dog images, with 133 different breeds. The images are organized in three different folders: train, valid, and test. Inside each of these folders, images are distributed the following way:

- Train set: 6680 images
- Validation set: 835 images
- Test set: 836 images

The distribution among classes is not homogeneous. For example, the following histograms shows the distribution of class representation in both the training and test set:



Although it is difficult to tell specific data heterogeneities, these graphs show clearly that 1) the distributions are different, and 2) there is a significant variance between most and least represented classes in both datasets. For example, the least represented class in the training set has 26 images (the Norwegian buhund), while the most represented has 77 (the Alaskan malamute).

The images have a variety of different resolutions, including widths going from a minimum of 121 pixels to 3456, and heights from 153 to 3072 pixels. The images are also colorful. However, they are taken from different angles and cover different parts of the dogs' bodies, so we will need to do some pre-processing/augmentation in order to learn the right attributes. Some images also present humans. Here are some examples of images in the dataset:



### Data cleaning

Some pictures have more than one dog, sometimes from different breeds. This type of information could be considered noisy. Some other pictures include humans, dressed dogs, or have printed legends. Here are some examples:



We could decide to remove these ones to avoid potential problems, but the dataset is already small, so we have decided to keep them.

### Data transformation and augmentation

The process of data augmentation is key when we are dealing with small datasets. In our case, due to the high number of classes, we are presented with few examples of pictures for each breed, which can be clearly insufficient for learning. When we use transfer learning, important features have been already learnt by the transferred network, but we definitely need more information if we want to build a classifier from scratch.

Data augmentation techniques are also used in large datasets to avoid overfitting, when other techniques (such as regularization) are not enough.

Data augmentation techniques are diverse, especially in the domain of images. Images can be processed in a variety of ways, and the way we do it can have significant impact in the process of training our model. In order to decide which image augmentation techniques we should use for our problem, it is good to perform an exploratory data analysis in first place. Luckily, images are normally "understandable" by a human being by just looking at them.

In our case, we have made use of **imgaug**<sup>5</sup>, a popular image augmentation library for the python language. *Imgaug* provides a large set of transformation and augmentation

---

<sup>5</sup> "Imgaug — Imgaug 0.4.0 Documentation."

techniques for the images in the dataset. Next we present some of these techniques applied to images in our dataset.

### **Gaussian Blur**

---

The image is blurred using a Gaussian function. This technique reduces image noise, at the cost of reducing detail.



### **Superpixels**

---

Superpixels technique groups pixels that are similar in color and other properties.



### **Flipping**

---

Flipping is a common and very simple technique that has very good results in a variety of problems.



### **Cropping**

---

By cropping an image we are removing some areas of it. Random cropping is commonly used in Machine Learning.



### **Image Rotation**

---

Pictures can be taken from different perspective and showing the object to be detected can be a burden generalization.





These and other examples can be found inside the [dog breeds augmentation examples](#) notebook, in this project Github repository.

### New data gathering

Another common thing that can be done to solve the problem of dealing with a small dataset is leveraging the image availability over the Internet to make our dataset bigger. Gathering more data is a common technique used for real life problems. In this particular case, we are interested in investigating how to get a good performance with few images, so we have decided to keep the dataset as it is.

### Implementation

This section describes how I have approached the implementation of the different components needed to find the best classifier. One can find below details about benchmarking, techniques and technologies, concrete developments and the basement for the experiments performed.

### Benchmark: The basic notebook

This project has a base notebook included (**dog\_app.ipynb**) that somehow guides the student in the process of applying the correct techniques for the problem. My first step in the project was completing this notebook. Although my research has gone further, I have kept the original notebook as a benchmark for the rest of the project.

In the benchmark notebook I have achieved the following results.

### From Scratch solution

Using basic augmentation provided by PyTorch <sup>6</sup> and a quite simple convolutional network, I achieve **an accuracy of 15%**, which is not particularly good, but is way better than the random solution.

Code available at: [https://github.com/lumartin/cnn-dog-breed-classifier/blob/master/dog\\_app.ipynb](https://github.com/lumartin/cnn-dog-breed-classifier/blob/master/dog_app.ipynb)

### Transfer Learning Solution

---

<sup>6</sup> "PyTorch."

For transfer learning I have made use of a pretrained ResNet50<sup>7</sup> with a very simple classifier on top. This leads to an accuracy of 82%.

Code available at: [https://github.com/lumartin/cnn-dog-breed-classifier/blob/master/dog\\_app.ipynb](https://github.com/lumartin/cnn-dog-breed-classifier/blob/master/dog_app.ipynb)

## Techniques and Technologies

As stated before, given that this is an image classification problem, I have decided to use Convolutional Neural Networks for the solution, as they are the state-of-the-art technology for this matter.

I have explored several ways to approach the problem, divided in two main groups: **from-scratch** implementation and **transfer learning** implementation. This type of problem is clearly better solved using a **transfer learning** approach, but I think it worth it to make some research in the other approach. Most of the work I have made for this project has been in this context.

The main technology used has been PyTorch, the Facebook Neural Networks Library. It gives me the ability to build and train neural networks from scratch, as well as use common transfer learning solutions. Together with PyTorch, I have made use of common Python libraries like Numpy<sup>8</sup> and Matplotlib<sup>9</sup>.

The main execution and development tool used has been Jupyter Notebook<sup>10</sup>, but I decided to make some pure Python development to isolate some code, as the programming style proposed by Jupyter, though very useful for some tasks, ends up being tightly coupled to implementation details and difficult to maintain and reuse.

More concretely, I have developed an experimentation framework specially built for this problem, but that I plan to separate in an independent package in the future.

I have made use of Google Colab<sup>11</sup> (a PRO subscription) to have access to a good GPU. Also, I own a 1070ti, but I cannot find justification to execute the experiments locally nowadays if you have a common equipment like mine.

## Experimentation framework

In order to facilitate the executions of the experiments, I have developed a library that wraps the low-level details. This library exposes a single method called **"run\_experiments"**, that receives a dictionary of hyperparameters and runs experiments making combinations of the values received. For example, we can specify that we want a set of experiments that combine two different optimizers, such as Adam<sup>12</sup> and

---

<sup>7</sup> He et al., "Deep Residual Learning for Image Recognition."

<sup>8</sup> "NumPy — NumPy."

<sup>9</sup> "Matplotlib: Python Plotting — Matplotlib 3.2.1 Documentation."

<sup>10</sup> "Project Jupyter."

<sup>11</sup> "Google Colaboratory."

<sup>12</sup> Kingma and Ba, "Adam."



Adagrad<sup>13</sup>, and two learning rates, say 0.01 and 0.03. With these settings, the framework will perform the following experiments:

1. Optimizer: Adam, Learning Rate: 0.01
2. Optimizer: Adam, Learning Rate: 0.03
3. Optimizer: Adagrad, Learning Rate: 0.01
4. Optimizer: Adagrad, Learning Rate: 0.03

More concretely, the allowed hyperparameters that can be included in the configuration are the following:

- **augmentations**: Specify a list of **imgaug** augmentation methods.
- **learning\_rates**: List of learning rate values
- **epochs**: Number of training epochs
- **optimizers**: List of optimizers
- **models**: List of models

This is the signature for the “**run\_experiments**” function:

```
def run_experiments(paths, hyperparameters, model_file='model')
```

where **paths** is a dictionary specifying, **hyperparameters** is the dictionary described previously, and **model\_file** is the name of the file that we want for storing our model.

Everything else is hidden (preprocessing, training and testing), but we can still use the generated model to make calculations.

This framework is designed to work within a Jupyter notebook environment, so it directly prints a report to the console.

## Code and resources

---

All the code and resources developed for this project can be found at:

<https://github.com/lumartin/cnn-dog-breed-classifier>

The repository is organized in several folders, containing the following components:

- Benchmark: [https://github.com/lumartin/cnn-dog-breed-classifier/blob/master/dog\\_app.ipynb](https://github.com/lumartin/cnn-dog-breed-classifier/blob/master/dog_app.ipynb)
- Analysis: <https://github.com/lumartin/cnn-dog-breed-classifier/tree/master/analysis>
- Experiments: <https://github.com/lumartin/cnn-dog-breed-classifier/tree/master/experiments>
- Utils: <https://github.com/lumartin/cnn-dog-breed-classifier/tree/master/utils>

---

<sup>13</sup> Lydia and Francis, “Adagrad - An Optimizer for Stochastic Gradient Descent.”

## Experiments and Results

The experiments I introduce in this section are divided into two main groups: from-scratch and transfer learning. Of course, most of the work have been performed within the scope of the from-scratch approach. Next sections describe this process in detail.

### From-scratch approach

In the experiments performed to find a better classifier for the **from-scratch** approach, I leverage the little framework I have developed, described in the previous section. Next, I describe the various features involved and how I performed this task.

First, I will describe the environment used and the metrics. Both are common for the whole set of experiments.

### Environment

More concretely, I have uploaded the dataset and tools to my personal Google Colab environment, with a PRO subscription enabled, and used the computing power provided by this environment. I have decided to use Colab because access to GPU is cheaper than other providers (such as AWS), and it integrates seamlessly with my Google Drive, a tool I am comfortable using. To be more specific, this is the description of the GPU provided by Google Colab:

NVIDIA-SMI 440.64.00				Driver Version: 418.67		CUDA Version: 10.1	
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile Uncorr. ECC		
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.	
0	Tesla P100-PCIE...	Off	00000000:00:04.0	Off		0	
N/A	36C	P0	32W / 250W	16215MiB / 16280MiB	0%	Default	
Processes:							
GPU	PID	Type	Process name	GPU Memory Usage			

I can run 4 processes simultaneously, so actually have access to 4 of these GPUs. Given that training of these networks is especially slow, this has been extremely useful.

### Metrics

In Machine Learning, it is normal to have a metrics or set of metrics appropriate for the specific learning problem. For example, in a classifying problem like this one, we would like to use something like **Accuracy**<sup>14</sup> or **AUC**<sup>15</sup>. However, from an engineering point of view, there are other things we would like to measure, as they may have significant impact on performance, memory or even energy consuming.

In this case I have selected **accuracy** as metric for comparing performance of experiments, but I am also measuring the impact in **time per epoch** and **total time** for training. Additionally, I will present some information on the **size of the model**.

<sup>14</sup> "Classification."

<sup>15</sup> "Classification."

## Experiment 1: Architecture comparison

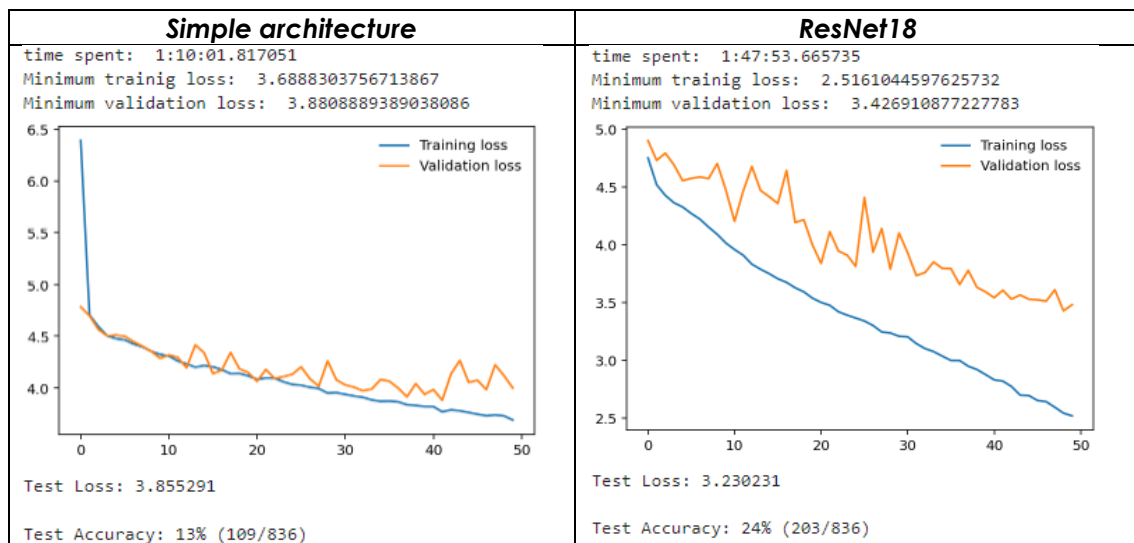
This experiment has the goal of testing the capabilities of two architectures: Basic Convolutional Neural Network and Resnet18. Technical specifications for both networks can be found in the `utils` directory of the code repository, inside `simple_model.py` and `resnet_model.py` respectively.

This experiment has been executed using parameters specified below:

- Augmentations: **Crop**
- Optimizer: **Adam**
- Learning Rate: **0.001**
- Epochs: **50**

With this experiment, I just wanted to have a taste of the power of an overly simplistic approach versus one of the state-of-the-art architectures. ResNet was designed to be able to stack lots of convolutional layers without being affected to the "vanishing gradient" problem. I wanted to know if this type of architecture was useful for this problem.

The results, are as follows:



As we can see, ResNet is able to fit to the data much faster. With very low number of epochs and few preprocessing techniques applied, is able to reach an accuracy of 24%.

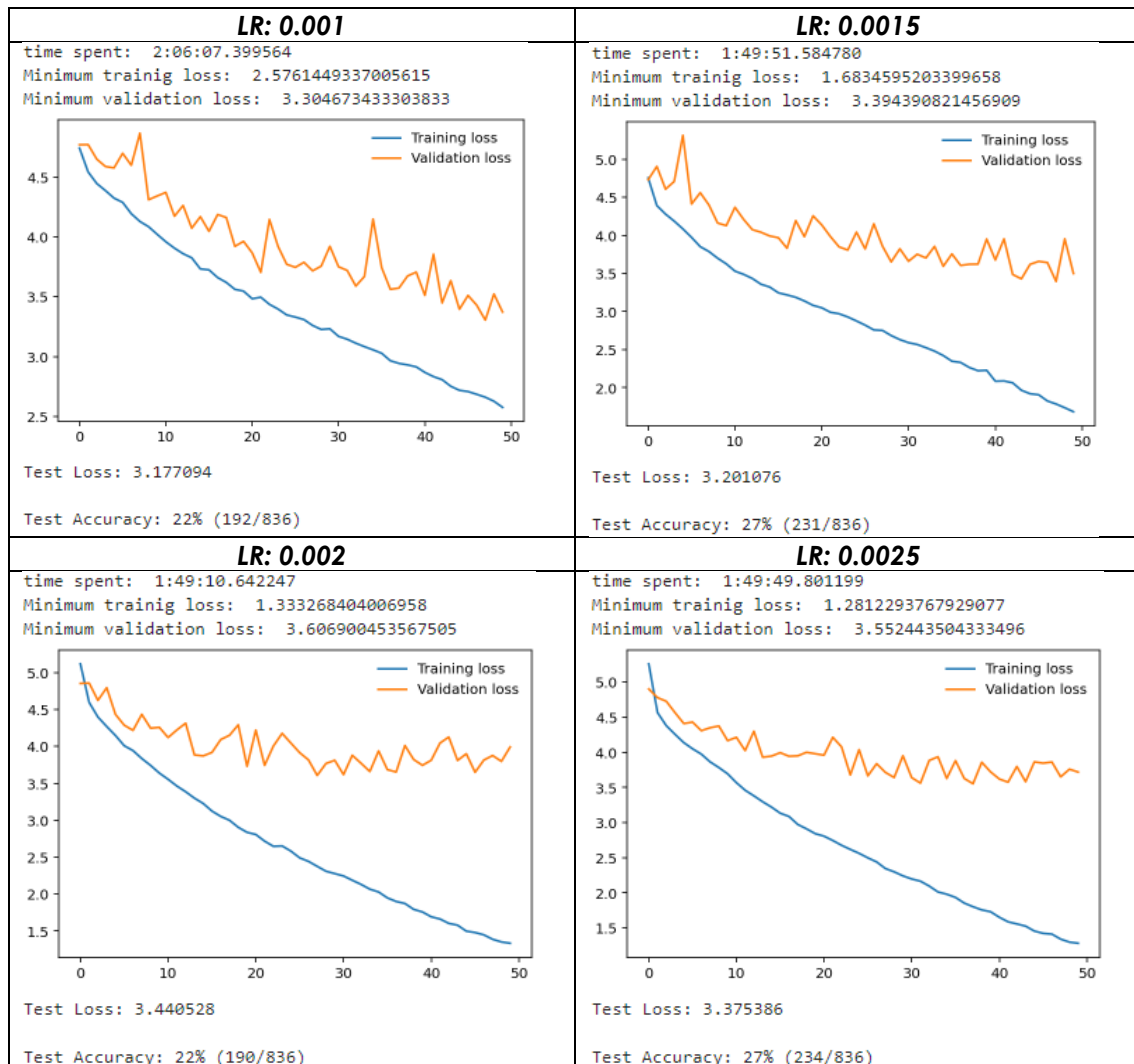
## Experiment 2: Learning rate comparison

This experiment has the goal of comparing different learning rates for the Adam optimizer. This experiment has been executed using parameters specified below:

- Augmentations: **Crop**
- Optimizer: **Adam**
- Learning Rates: **0.001, 0.0015, 0.002, 0.0025**

- Epochs: **50**
- Architecture: **ResNet18**

The results, are as follows:



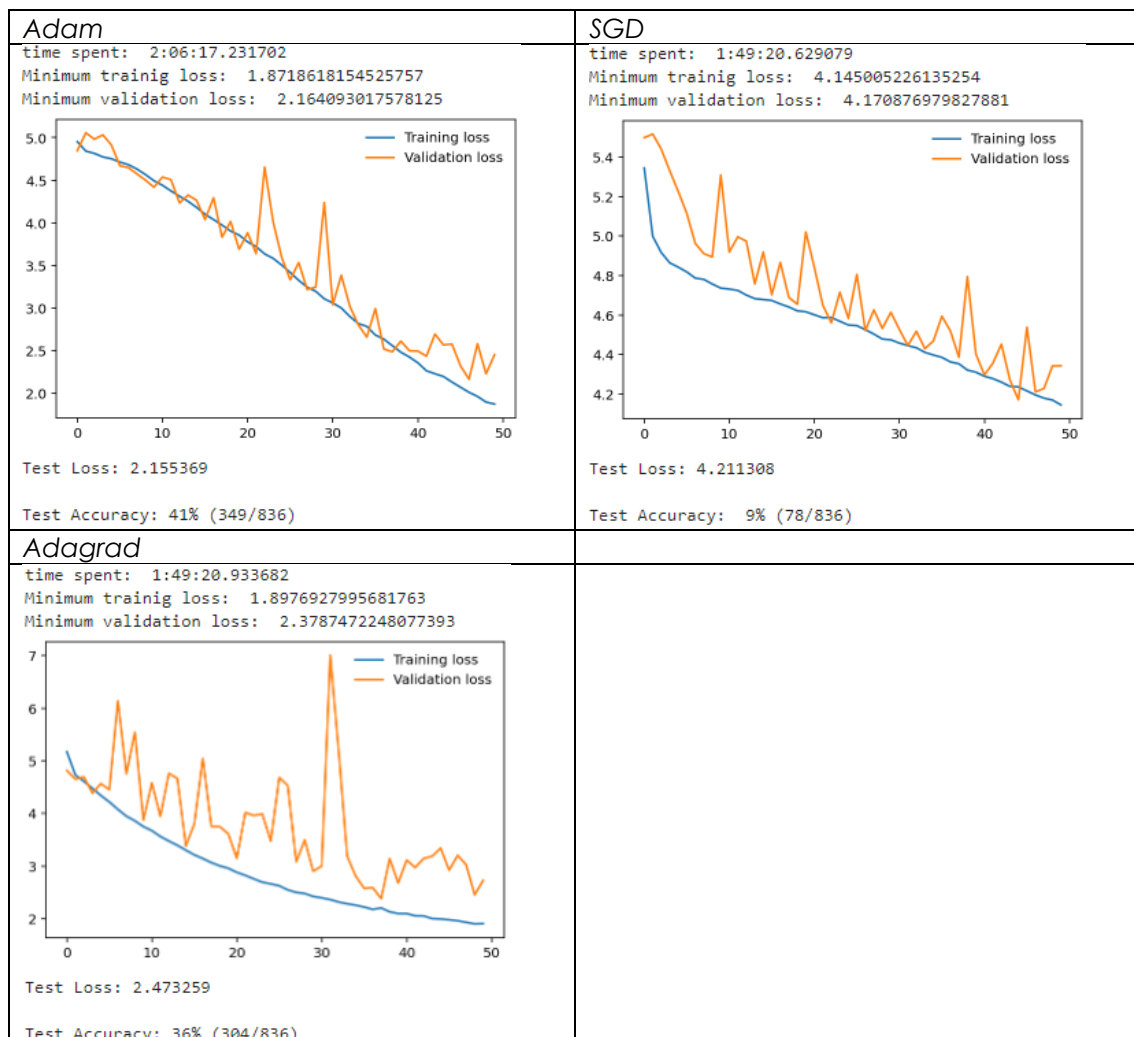
The results are a bit weird, since the most promising values are 0.0015 and 0.0025, but I see a little decrease with 0.002. However, it seems that 0.0015 tends to overfit a bit less, so I will stick to this value in further experiments.

### Experiment 3: Optimizer comparison

This experiment goes a little bit further by combining different optimizers and learning rates. More concretely, the parameters used have been:

- Augmentations: **Crop**
- Optimizer: **Adam, SGD, Adagrad**
- Learning Rates: **0.0015**
- Epochs: **15**
- Architecture: **ResNet18**

Given the cost of each training, I had to reduce the number of epochs significantly, but I think it worth the effort seeing how different optimizers behave in these conditions. Here are the results:



As we can see, clearly Adam seems to be a good choice. SGD is not performing good, and Adagrad, though close, does not seem to worth the try. I decide to keep Adam as my main choice for the rest of the experiments.

#### Experiment 4: Augmentation techniques

In this experiment set, I have made use of the augmentation library **imgaug** to try to ease the main inner lack this dataset has: number of examples. As image augmentation techniques number is big, I have tried to make an exploration to find which of them have a significant impact on performance.

More concretely, I have tried the following set of augmentation techniques:

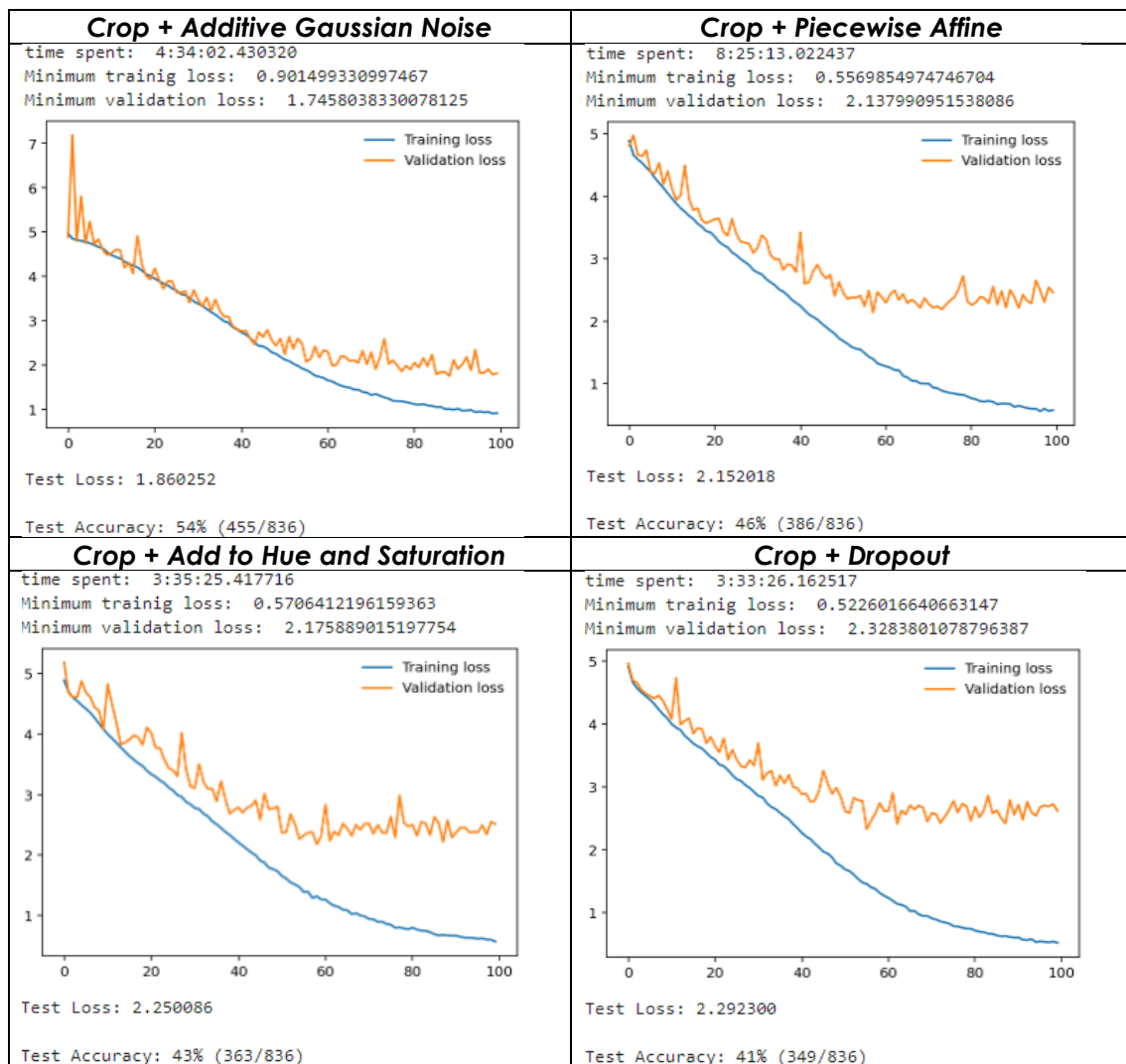
- Crop
- Additive Gaussian Noise
- Add & Multiply
- Add to Hue and Saturation

- Affine
- Dropout
- Elastic Transformation
- Grayscale
- Invert
- Linear contrast
- Piecewise Affine
- Flip

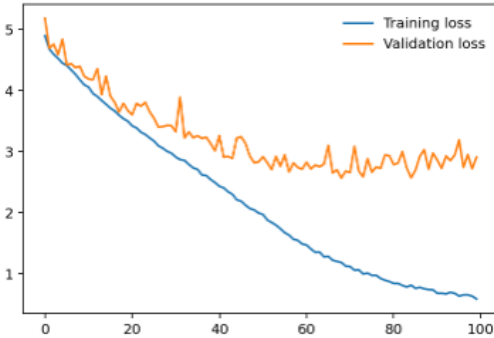
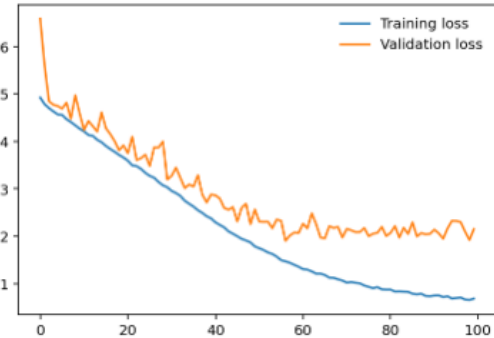
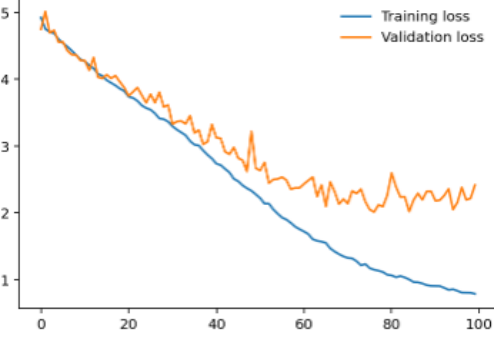
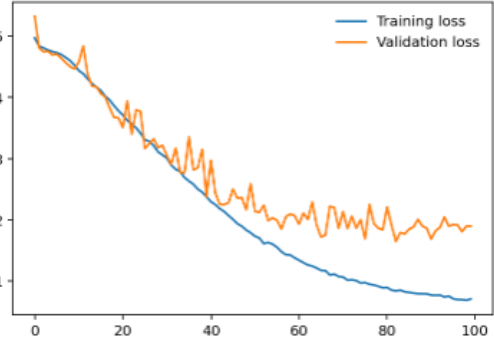
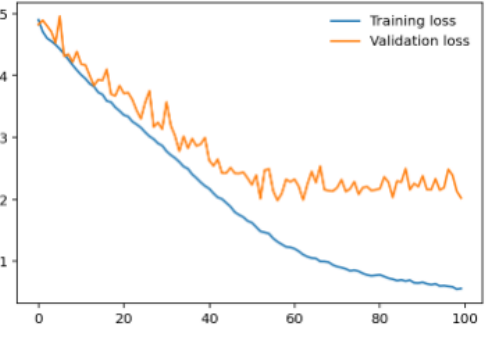
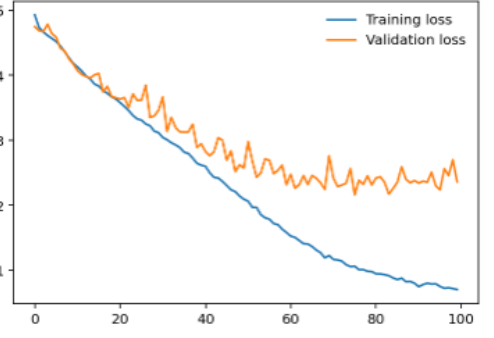
The fixed parameters have been:

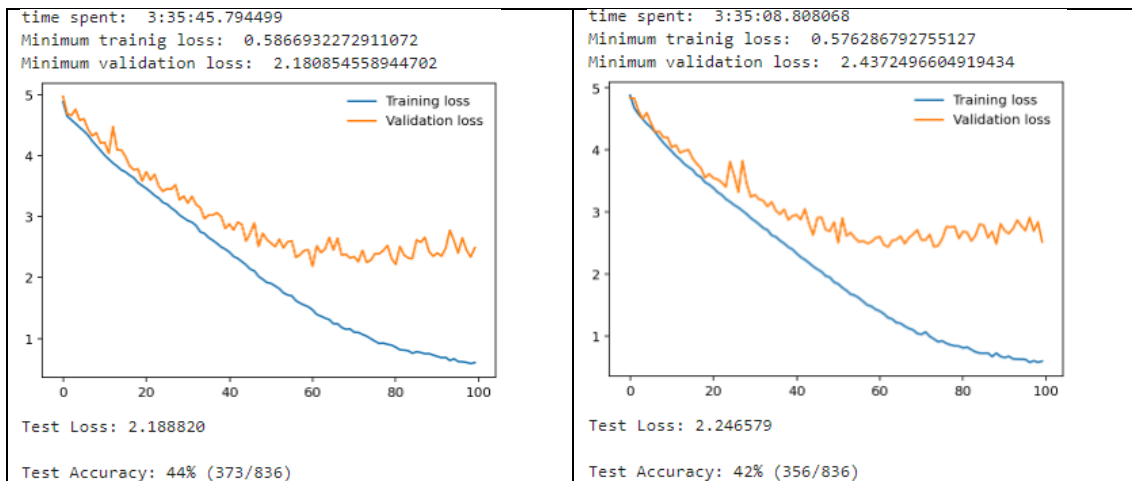
- Optimizer: **Adam**
- Learning Rates: **0.001**
- Epochs: **100**
- Architecture: **ResNet18**

As performing experiments involving all the combinations is too expensive, I began trying different techniques in isolation. Soon I realized that **Cropping** enables a significant improvement in the results, so I tried the other techniques together with this one. Here are the results:

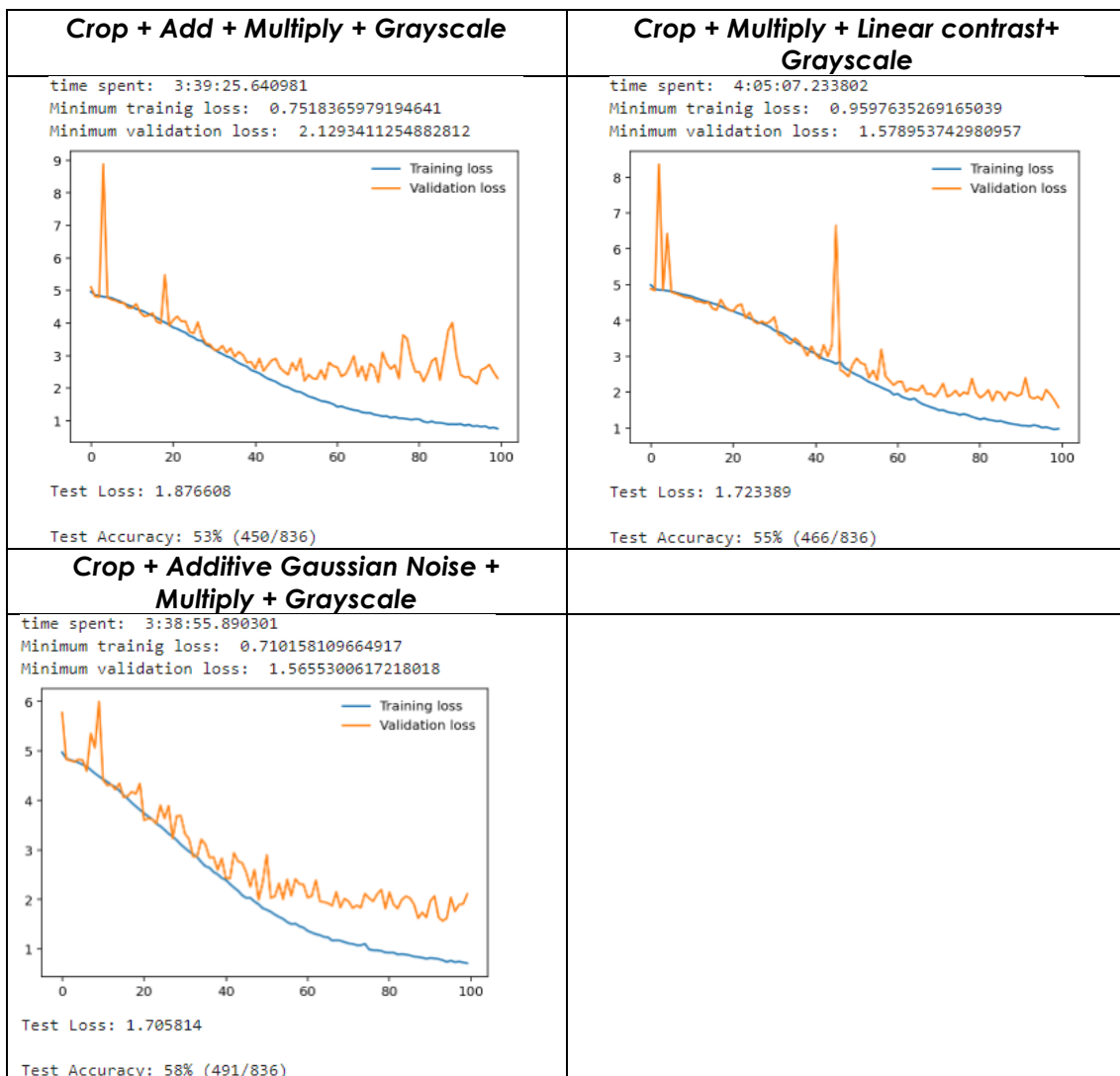




<p><b>Crop + Elastic Transformation</b></p> <p>time spent: 3:35:49.421438  Minimum trainig loss: 0.5783833265304565  Minimum validation loss: 2.560019016265869</p>  <p>Test Loss: 2.454370  Test Accuracy: 40% (338/836)</p>	<p><b>Crop + Grayscale</b></p> <p>time spent: 3:35:31.355132  Minimum trainig loss: 0.6517424583435059  Minimum validation loss: 1.9011183977127075</p>  <p>Test Loss: 1.937640  Test Accuracy: 48% (409/836)</p>
<p><b>Crop + Linear Contrast</b></p> <p>time spent: 3:37:56.518314  Minimum trainig loss: 0.7866325974464417  Minimum validation loss: 2.011298894882202</p>  <p>Test Loss: 2.075913  Test Accuracy: 46% (392/836)</p>	<p><b>Crop + Multiply</b></p> <p>time spent: 3:34:00.925216  Minimum trainig loss: 0.684609055519104  Minimum validation loss: 1.6453732252120972</p>  <p>Test Loss: 1.574850  Test Accuracy: 58% (492/836)</p>
<p><b>Crop + Add</b></p> <p>time spent: 3:37:32.206281  Minimum trainig loss: 0.5460805892944336  Minimum validation loss: 1.9816585779190063</p>  <p>Test Loss: 2.016611  Test Accuracy: 49% (414/836)</p>	<p><b>Crop + Invert</b></p> <p>time spent: 3:32:56.417019  Minimum trainig loss: 0.7008154988288879  Minimum validation loss: 2.1588082313537598</p>  <p>Test Loss: 2.123769  Test Accuracy: 47% (394/836)</p>
<p><b>Crop + Affine</b></p>	<p><b>Crop + Flip</b></p>



I have also performed some more complex combinations of techniques, such as the ones introduced below.



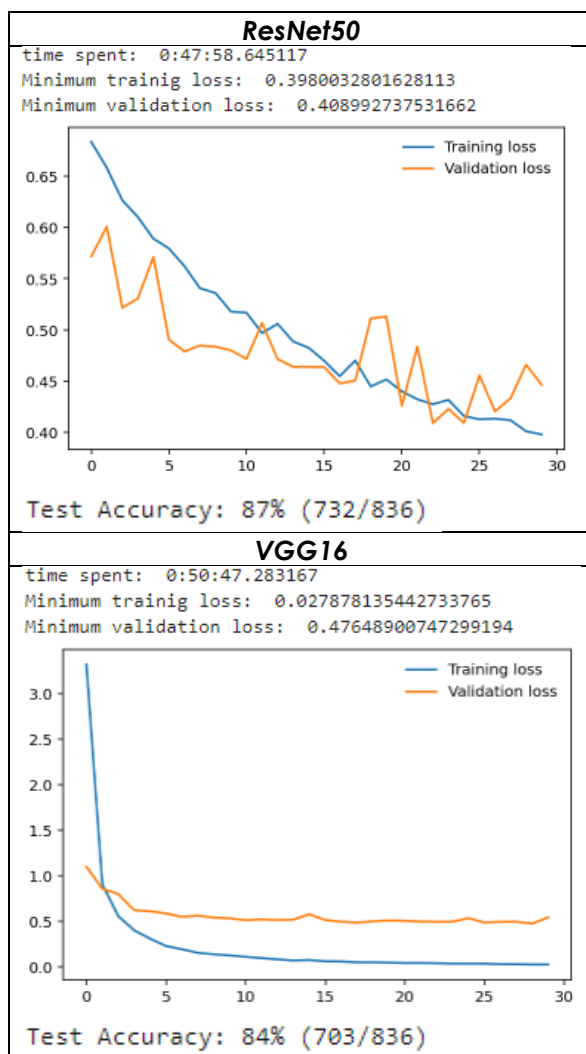
As we can see, the combination of Crop, Multiply, Additive Gaussian Noise and Grayscale produces the best results. I think that iterating this way could lead us to find an even better result, but for the purpose of this work I think this is enough.

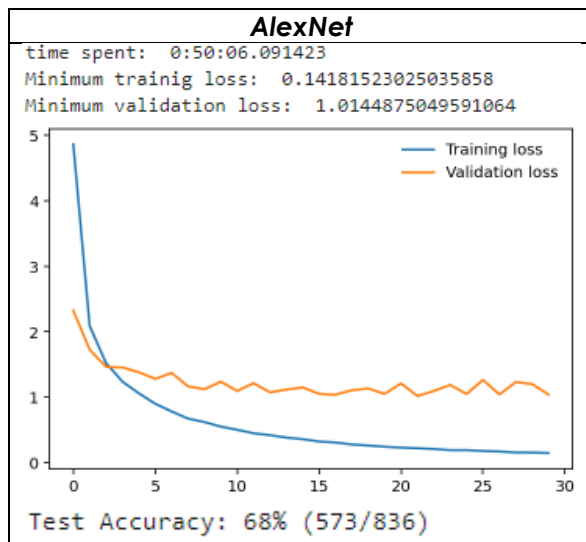
### Transfer learning approach

For the transfer learning experiments I have leveraged some of the already available pretrained models provided within PyTorch. More concretely, in this case I have made use of:

1. ResNet50
2. VGG16
3. AlexNet

I have put a little classifier on top, the same in the three cases. The results are of course much better than what I have achieved in the **from-scratch** solution. This makes sense, since these models have a huge force of computing training stored in their parameters. Here are the results:





## Conclusions

The comparison between benchmark and final models is described in the following table:

FROM SCRATCH	
Benchmark	Final
Accuracy: <b>15%</b>	Accuracy: <b>53%</b>
Hyperparameters:	Hyperparameters:
<ul style="list-style-type: none"> <li>Optimizer: <b>Adam</b></li> <li>Learning Rates: <b>0.001</b></li> <li>Epochs: <b>15</b></li> <li>Architecture: <b>Simple CNN</b></li> <li>Augmentation: <b>Yes, simple Cropping, Flipping and Rotation.</b></li> </ul>	<ul style="list-style-type: none"> <li>Optimizer: <b>Adam</b></li> <li>Learning Rates: <b>0.001</b></li> <li>Epochs: <b>100</b></li> <li>Architecture: <b>ResNet18</b></li> <li>Augmentation: <b>imgaug Cropping, Additive gaussian noise, Multiply and Grayscale</b></li> </ul>
TRANSFER LEARNING	
Benchmark	Final
Accuracy: <b>82%</b>	Accuracy: <b>87%</b>
Hyperparameters:	Hyperparameters:
<ul style="list-style-type: none"> <li>Optimizer: <b>Adam</b></li> <li>Learning Rates: <b>0.001</b></li> <li>Epochs: <b>15</b></li> <li>Architecture: <b>ResNet50 + 1 layer classifier</b></li> <li>Augmentation: <b>No.</b></li> </ul>	<ul style="list-style-type: none"> <li>Optimizer: <b>Adam</b></li> <li>Learning Rates: <b>0.001</b></li> <li>Epochs: <b>30</b></li> <li>Architecture: <b>ResNet50 + 2 layers classifier + dropout</b></li> <li>Augmentation: <b>No.</b></li> </ul>

After doing the experiments described above, there are several bullets I can highlight clearly:

1. Transfer learning is way better and easier for problems with few examples. However, it needs a little more work to have better results.
2. Augmentation is key. In absence of a more representative dataset, augmentation techniques can strongly elevate performance.
3. Optimizers do not seem to be that relevant in low on data problems, but some are clearly not working (SGD).
4. It would be interesting to test whether balancing examples per class by including new data would improve performance.
5. The problem of generalization is a difficult one. Every single network I have tried overfitted after epoch 50-70. This clearly needs more work.

Regarding the exploration process itself, I have experimented the tendency to chaos that a tool like Jupyter Notebooks can generate if is not effectively managed. As more lines of code are needed, the management gets more difficult. I propose to take some time and build a little library that help us organize our code and use notebooks for what they are designed: exploration.