POLYTECHNIC OF TURIN

MACHINE LEARNING & DEEP LEARNING HOMEWORK 2

MAURIZIO VASSALLO, S276961

Contents

INTRODUCTION **3**
DATA EXPLORATION **3**
PREPROCESSING **4**
MODELS TRAINING **5**
TRAINING FROM SCRATCH 5
TRANSFER LEARNING 7
CNN FREEZED 9
FC LAYERS FREEZED 10
DATA AUGMENTATION **11**
(Extra) Beyond AlexNet **12**
VGG16 12
RESNET18 13

INTRODUCTION

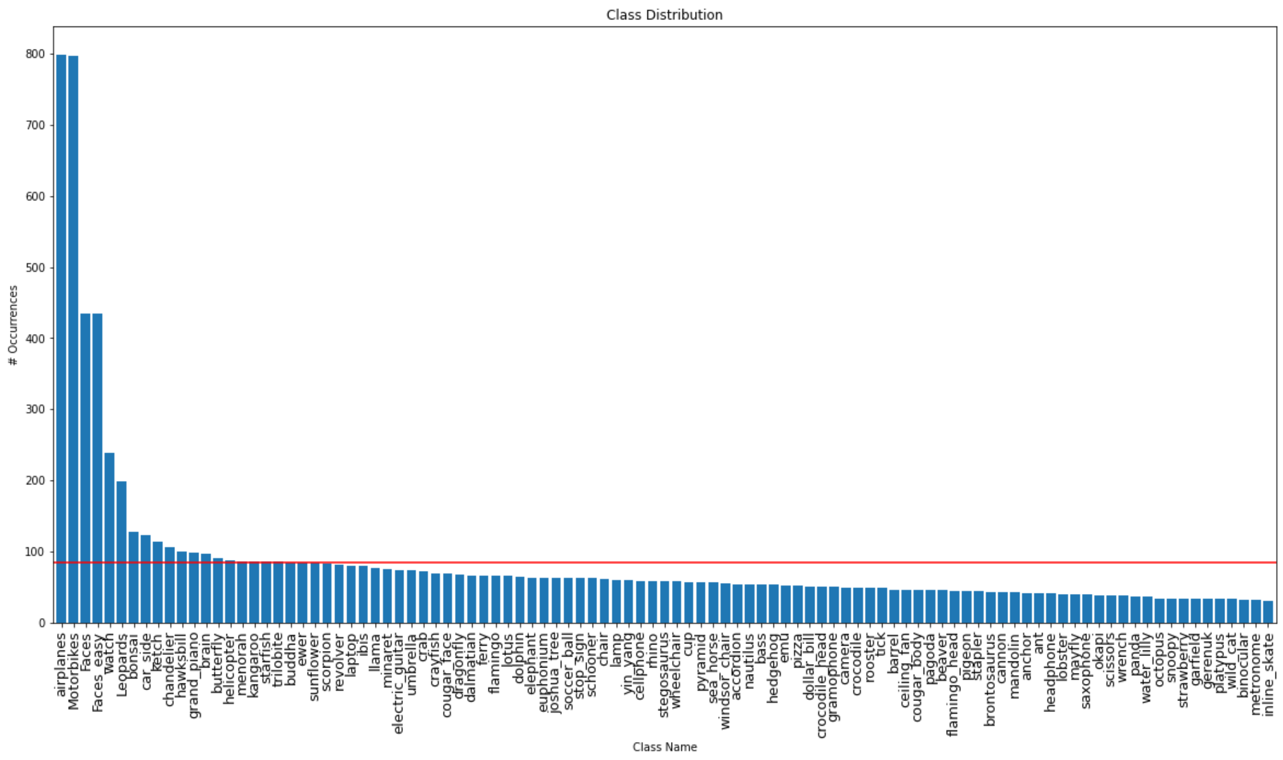
This is the **homework 2** report of the course of Machine Learning & Deep Learning course at polytechnic of Turin.

The objective is to train a Convolutional Neural Network for image Classification. The focus will be on the **Caltech-101** dataset and the usage of a Convolution Neural Network: **Alexnet.**

DATA EXPLORATION

This dataset contains 9145 images from 102 different classes, the important ones are 101 and one category is for not useful images. The size of each image is roughly 300x200 pixels with 3 channels: RGB.

Each class contains from 40 to 800 images and most categories contain 50 images.



Some statistics are:

```
Total Images: 8677
Max # Images in a class: 799
Min # Images in a class: 30
Mean Images for classes: 84.91
STD Images for classes: 117.75
```

As it is possible to see the categories are unbalanced, as one may expect the most common categories are about vehicles and faces and while the least common one is about objects like skates, binoculars and metronome.

Even if the classes are unbalanced the performances should not be bad, since in this case the test set is taken from the same dataset in a way that keeps proportions; and in the real case, generally, one may expect, for example, to have more images of faces than the images of metronome.

PREPROCESSING

For the pre-processing part there are some steps.

For the cleaning process there is a category that must be removed when the data is loaded, this is the *BACKGROUND_Google* class.

Then, the dataset is split in train, the proportions are: 2-1 for training and validation sets and 2-1 for training and validation sets; so, the overall proportions are almost: 4-2-3. This is made first implementing the *Caltech* class, which helps load the images and divide them in train and test set and it takes care to filter out the *BACKGROUND_Google* class. The index of the train and test set are given in two files: *train.txt* and *test.txt.* Then the train set is divided in actual train set and validation set using a method inside the *caltech_dataset* class, that split the given data using an array representing the data index. Train and validation are split such that train set has odd indices and validation set has even indices, in this way it is avoided that a whole class in filtered out and the samples for each class are half and half for train and validation sets.

```
Train Dataset: 3856
Valid Dataset: 1928
Train/Valid Dataset Proportion: 2.0 (Required: 2)
Test Dataset: 2893
```

Also, the images have to be processed using a composition of transformations, in particular data has to be: resized to 256 px, perform a centre crop with 224 px since Alexnet needs an image with 224x224 px size, transform the image in a tensor and normalize this tensor using a specific mean and standard deviation. To know mean and standard deviation means to have a prior knowledge about the dataset or calculate them.

Given the 3 datasets, generate the dataloaders using the *DataLoader()* method from the *torch.utils.data* library. This method divides the dataset in batches given a batch size, in this case 256.
Choosing a good batch size is difficult: the gradient of the loss function is computed over an entire batch. So, a very small batch size means: small amount of data to be storage, fast processing but the gradient would be all over the place and almost random, because learning will be image by image or a small set of them. A larger batch size means more amount of data to be storage, slower processing but more 'accurate' gradients because now the optimization is over a larger set of images. So, the trade-off is: many 'bad' updates versus few 'good' updates.
I tried different values for the batch size, but Google Colab GPUs limited memory size does not allow too large dimension since it would be out of memory after some epochs. So, I have to remain to a batch size of 256, that is not too small actually.

Finally, the model needs to be initialized. *Torchvision* library allows to import different models, in this case we are interested in the **Alexnet** model. AlexNet is a convolutional neural network that is 8 layers deep and this has 100 neurons at the last fully connected layer but we have only 101 classes, so we have to change this layer with a 'custom' one using the *Linear(in,out)* method in the *torch.nn* library, the parameters are the input number of neurons (from the previous layer) and the output number of neurons (our number of classes in this case).

Initialized the model, some parameters must be defined:

- A loss function, in this case CrossEntropyLoss function,
- The parameter of the network to be optimized, this allows to 'freeze' some parts of the network so that the weights will not be updated;
- An optimizer, for example Stochastic Gradient Descent which needs to know the parameter to be optimized, the learning rate used, the momentum and the weight decay;
- A scheduler to decrease on a factor *gamma* the learning rate every performed number of steps.

MODELS TRAINING

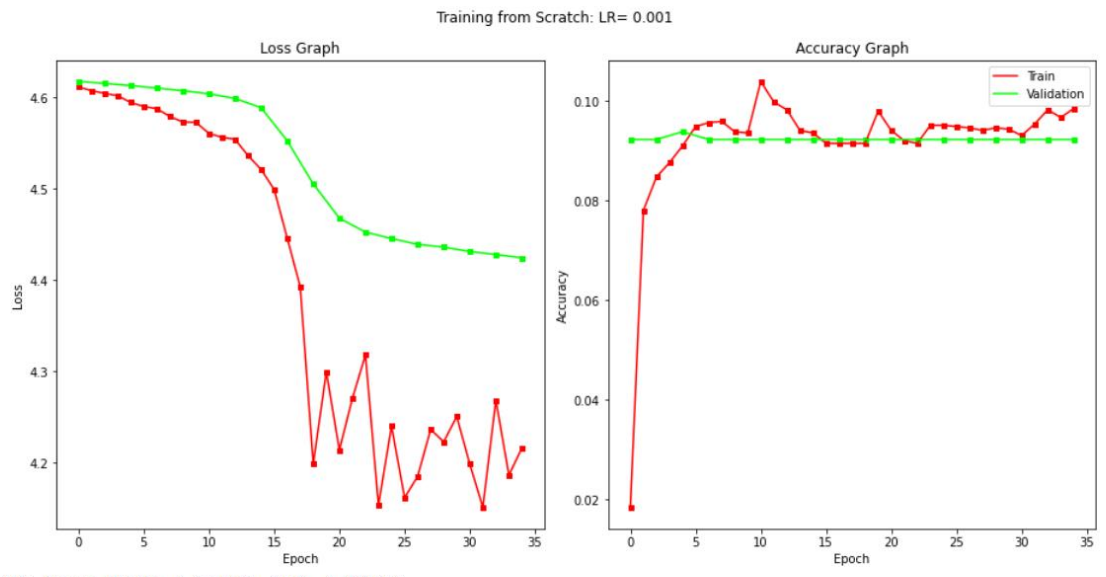
Since the training for deep neural networks is expensive, the models will train on Google GPUs. The GPUs available in Colab often include Nvidia K80s (memory: 24 GB), T4s(meemory: 16 GB), P4s (memory: 8 GB) and P100s (memory: 16GB). So, the model before training must be instructed to use GPU, this is made with *model.to('GPU').*

Then the model will train for each epoch on all the batches of the train set, the model will predict the classes of the images and accuracy and loss are calculated comparing the outputs with the labels. Then the loss in back propagated in order to update the weights and improve the performances in the next iteration. After a fixed number of epochs, the model is valuated on the validation set in the same way accuracy and loss are calculated. During evaluation the weights are not updates. This evaluation, if from one side it slows down the training process, it gives more control on the model and how the training is going: e.g. the model overfit the data, etc.

TRAINING FROM SCRATCH

The first point is to evaluate the performance of the Alexnet on the dataset, training it from scratch. The parameter used are:

```
NUM_EPOCHS = 35
LR = 1e-3
MOMENTUM = 0.9
BATCH_SIZE = 256
GAMMA = 0.1
STEP_SIZE = 20
DECAYING_POLICY: decrease the LR of a factor GAMMA after STEP_SIZE
OPTIMIZER = SGD
```



```
Min Loss: Train: 4.150970, Val: 4.424387
Max Accuracy: Train: 0.103994, Val: 0.093880
```

Here are the values for loss and accuracy for the training and evaluation sets.

As expected, the performances are not good since the dataset is not too large to train the model: we have 57,417,637 weights to updates and only ~3800 different images to train on for every epoch.

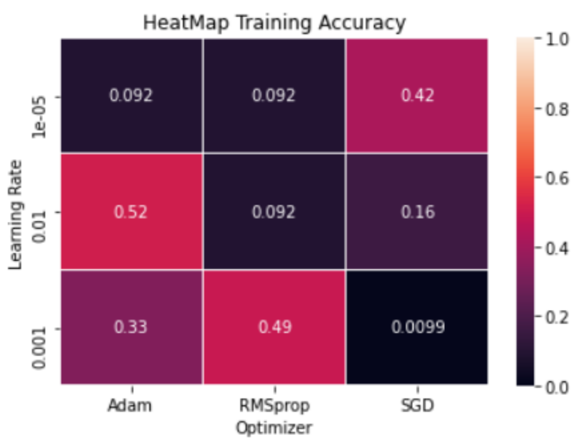
Then, the request is to train from scratch and evaluate the model using different parameters. The chosen parameters are:

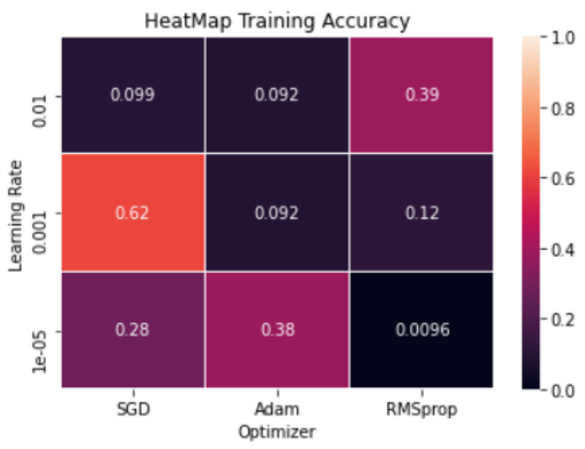
```
LR_values = [0.01, 0.001, 0.00001]
Optimazers = ['RMSprop', 'Adam', 'SGD']
```

Brief explanation of RMSprop and Adam:

RMSprop is optimization algorithm designed for neural networks. It tries to resolve the problem that gradients may vary in magnitudes: it can be tiny or be huge, which result in difficult updates. It attempts to solve this problem with trying to find a single global learning rate for the algorithm. The algorithm after a full-batch is interested only in the sign of the gradient, with that it can guarantee that all the weights updates are of the same size. With this adjustment it better deals with saddle points and plateaus as since it takes big enough steps even with tiny gradients (This cannot be solved just increasing the learning rate because the steps it takes with large gradients are going to be even bigger, which will result in divergence). So, RMSprop combines the idea of only using the sign of the gradient with the idea of adapting the step size for each weight. The step is adjusted considering the 2 previous gradients for the weights, is they are the same sign it means that that weight is going in the right direction and it should accelerate (increase learning rate), if they are different it must decelerate (decrease learning rate).

Adam is an adaptive learning rate optimization algorithm that's been designed specifically for training deep neural networks. It can be seen as a combination of RMSprop and Stochastic Gradient Descent with momentum, in fact, it uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using moving average of the gradient instead of gradient itself like SGD with momentum.
Adam is an adaptive learning rate method, which means, it computes individual learning rates for different parameters using estimations of first (mean) and second moments (uncentered variance, don't subtract the mean during the calculation of the variance) of gradient to adapt the learning rate for each weight of the neural network.
The main reasons to choose it are: the update of the weights is invariant of the gradient magnitude, which helps a lot in overcome areas with small gradients (like saddle points). In these areas SGD struggles to overcome them and it requires less memory to compute.





Here are the heatmaps of the result for train and validation accuracies for the different pairs LR-Optimizer. The pair who performs better in the validation set is (0.01, Adam)

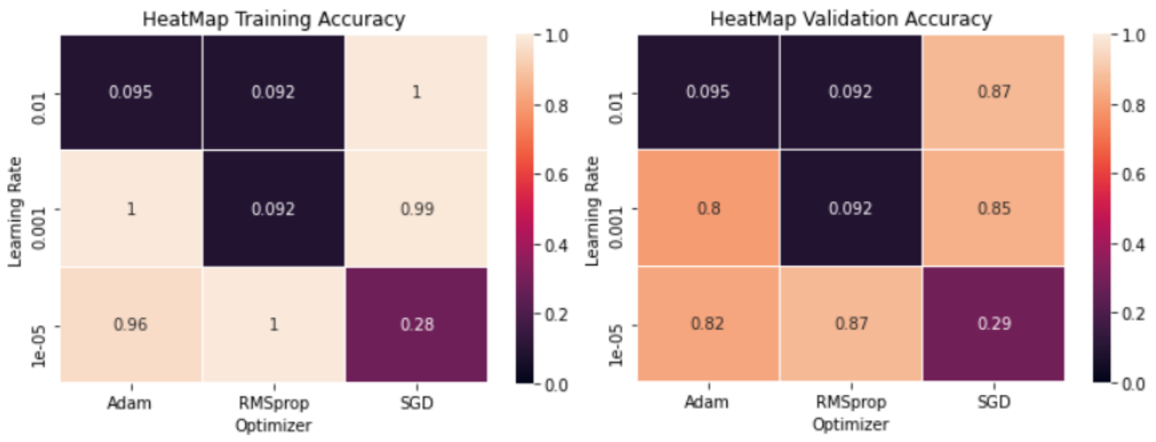
TRANSFER LEARNING

Since Caltech dataset is too small to train Alexnet neural network from scratch, it is common to use a strategy: **Transfer Learning**. This means that a pre-trained model is imported, this model is trained on a large dataset, in this case ImageNet dataset which contains more than 15 million high-resolution images from 22,000 different categories. Given this already trained model, we will train it on the dataset we are working on, freezing (not updating) some parts of it. There are some strategies to train it on our dataset: if the dataset is small freeze only the Convolution Neural Network and update fully connected layer, instead if the dataset is large it is possible also to update some layers before the fully connected one.

Since the ImageNet is different from the Caltech dataset and the models are pretrained on that dataset, it is usually better to use mean and standard deviation of the ImageNet dataset for the transformation, that are:

```
Mean = (0.485, 0.456, 0.406),
Std = (0.229, 0.224, 0.225).
```

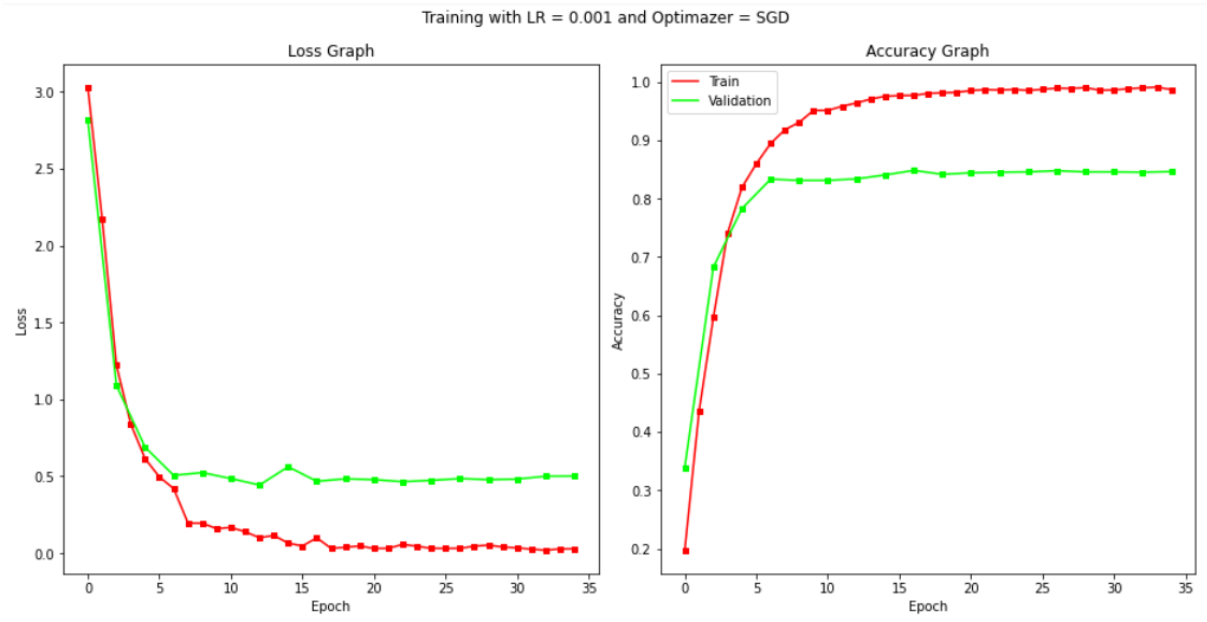
It is required to try both: freeze the CNN and freeze the FC layer and using the best parameters for LR and Optimizer found so far, which are:



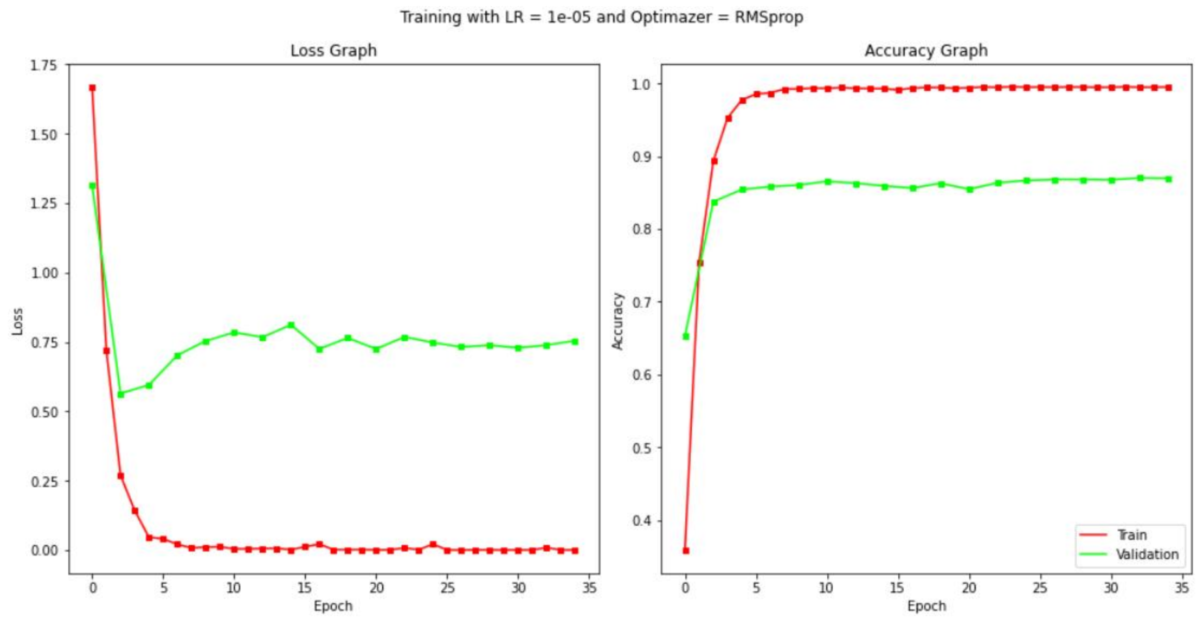
Learning Rate	Adam	RMSprop	SGD
0.01	0.095	0.092	1
0.001	1	0.092	0.99
1e-05	0.96	1	0.28

Learning Rate	Adam	RMSprop	SGD
0.01	0.095	0.092	0.87
0.001	0.8	0.092	0.85
1e-05	0.82	0.87	0.29

From the heatmap it is possible to see that there are different pairs which performs very well in the test set: (0.01, SGD), (0.001, Adam), (0.001, SGD), (0.00001, Adam), (0.00001, RMSprop). Some pairs overfitted since they perform 'bad' in the validation set: (0.001, Adam) and (0.00001, Adam). I would exclude (0.01, SGD) since the learning rate may be too high and may diverges in some cases.



```
Min Loss: Train: 0.018996, Val: 0.443116
Max Accuracy: Train: 0.991442, Val: 0.848548
```



```
Min Loss: Train: 0.000513, Val: 0.564287
Max Accuracy: Train: 0.995851, Val: 0.870332
```

Since usually do grid search on parameters is not feasible due to long training time, I decided to train on only 25 epochs also because the model seems to reach its max peak after 10/20 epochs.

It is possible to see from the graphs that RMSprop converges in less epochs than the SGD so would choose (0.00001, RMSprop) over (0.001, SGD), even if the LR=0.00001 is a bit too low and it may not converge.

```
        Best_LR = 0.00001
        Best_Optimizer = RMSprop
```

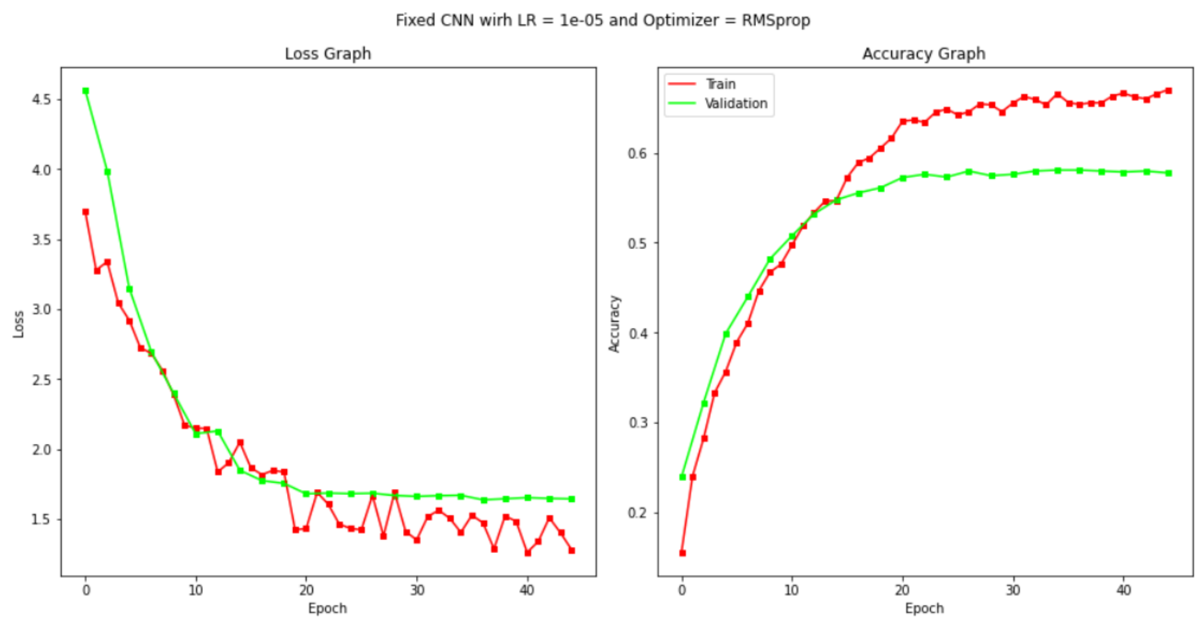
The score accuracy of the best params net with the test set is:

```
        Test Accuracy: 0.872796 (On Validation: 0.870332)
```

CNN FREEZED

Freezing the CNN, there are 2 million weights to optimize:

```
        57,417,637 total parameters.
        2,469,696 training parameters.
        0.04 ratio training/total.
```



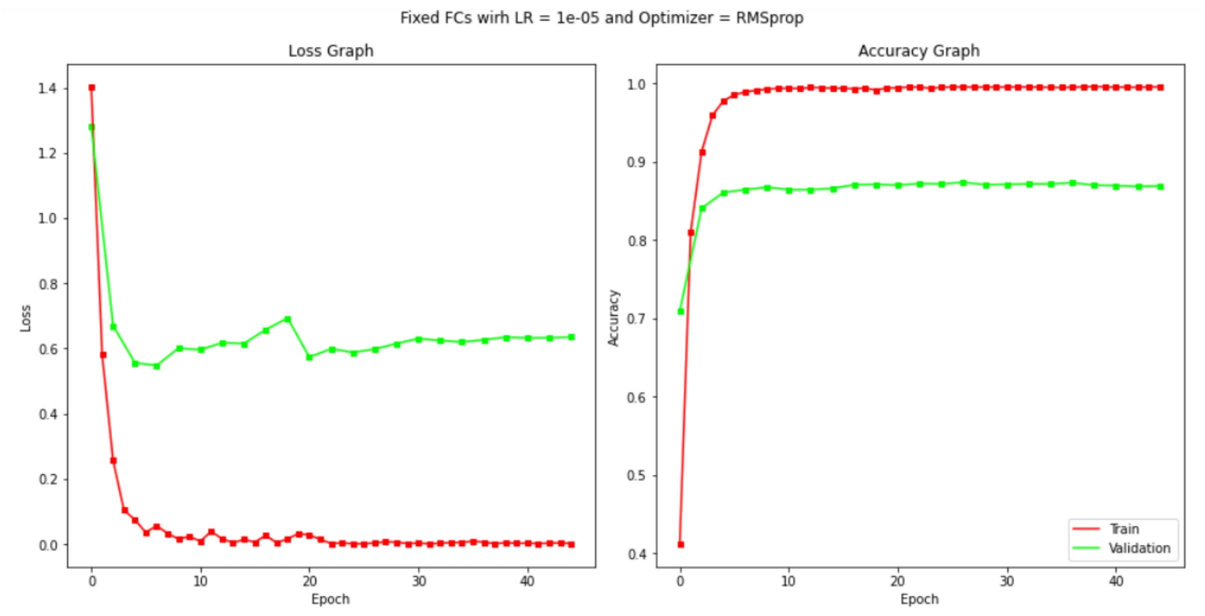
```
Min Loss: Train: 1.260867, Val: 1.636194
Max Accuracy: Train: 0.670384, Val: 0.580913
```

It is possible to see that the model does not learn very well since the two dataset, Caltech and ImageNet, are two different datasets and freezing the CNN, that is the all first part of the model, the one which receives the raw pixels, does not allow the model to 'adapt' to the new dataset. This brings to a decreasing in the validation accuracy, the previous accuracy (freezing nothing) was around 0.87, while now it is around 0.60. I also used more epochs since in some previous tests the model seems not to converge.

FC LAYERS FREEZED

Freezing the FC layers, there are 55 million weights to optimize:

```
        57,417,637 total parameters.
        54,947,941 training parameters.
        0.96 ratio training/total.
```



```
Min Loss: Train: 0.001229, Val: 0.547968
Max Accuracy: Train: 0.995851, Val: 0.873444
```

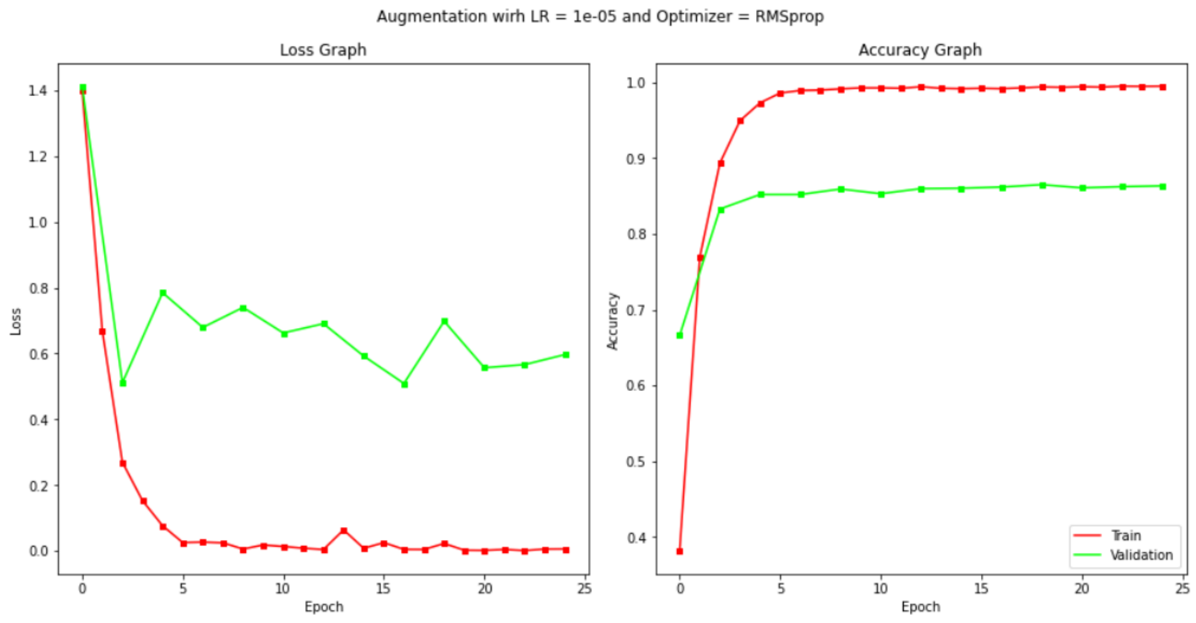
It is possible to see that the model does learns quite well. The results are very similar to the case of training all the weights (previous accuracy 0.8703), since the proportion of freezed weights is low, 4%

These results may be explained making some comparisons between the two datasets. Caltech is not that small dataset and has many categories so instead of freezing the CNN or the FC layers a solution may be something between, like freezing the firsts CNN layers, since these layers look for globally features like edges or curves (present in almost every images) and make the other layers of the CNN and the FC layers. Another option using the pre-trained weights as starting point and train all them with a very low learning rate so that the gradient at the first layers is almost all vanished.

DATA AUGMENTATION

As discussed before, to train a deep neural network there is a need of a large amount of data. It is possible to transform the images so that they are similar but the category of the image does not change. This allow to prevent overfitting and increasing the amount of data.

In order to increase the diversity of the avaible data it is possible to apply transformations to the data, example of transformations is: position augmentation like Scaling, Cropping, Flipping, Rotation, Translation but also colour augmentation like change Brightness, Contrast, Saturation, etc. It must be said that these transformation does not increase the memory used to store the images, since the transformations are applied 'on the fly' at each epoch by the data loader.



```
Min Loss: Train: 0.000688, Val: 0.508848
Max Accuracy: Train: 0.995332, Val: 0.865145
```

It is possible to see that the score does not change much from the normal train (without augmentation). The score is 0.865 and previous accuracy was 0.870. This result could explained noticing that the validation set has all images centered and vertical so augmentation does not help in this case but the model should be more general in this case and using a different dataset for validation should improve the performances.

(Extra) Beyond AlexNet

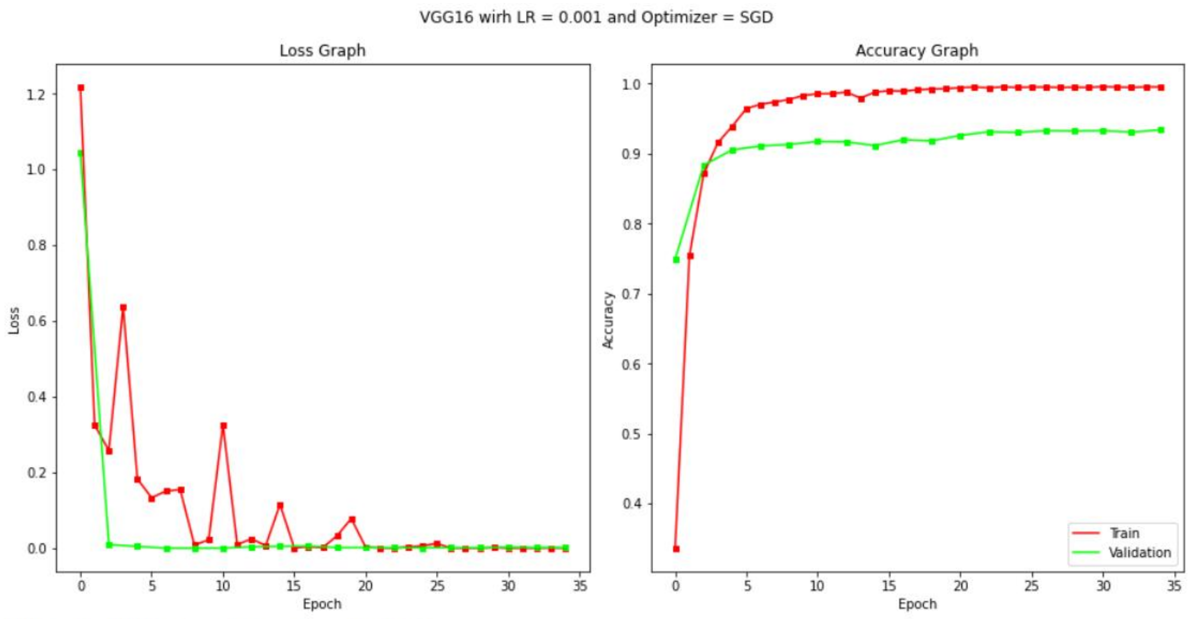
In the *pytorch* library there are different models. It is possible to use them and decided if initialized the model as pre-trained on the ImageNet dataset or not.

Among the models, I choose **VGG16** and **RESNET18**. For both the parameter of LR and Optimizer are the best ones of the previous tests.

VGG16

The VGG16 name refers to different models that present the same architecture and differ only in the depth: from 11 weight layers (8 conv. and 3 FC layers) to 19 weight layers (16 conv. and 3 FC layers).

Due to its depth and number of fully-connected nodes, VGG16 is over 533MB, for roughly 138 million parameters (almost 3 times Alexnet size).



```
Min Loss: Train: 0.000020, Val: 0.000078
Max Accuracy: Train: 0.995591, Val: 0.933610
```

The model performs very well even better than Alexnet, where the accuracies are 0.933 for VGG16 and 0.870 for Alexnet. It seems also to overfit less than Alexnet. These can be explained considering that VGG16 has a deeper configuration than Alexnet so has more learning power, due to his depth VGG16 requires a lot of memory, it requires to decrease the batch size of lot, from 256 to 32, to avoid that Google GPUs were out of memory.

Score on Test dataset:

```
Accuracy: 0.930521
```

RESNET18

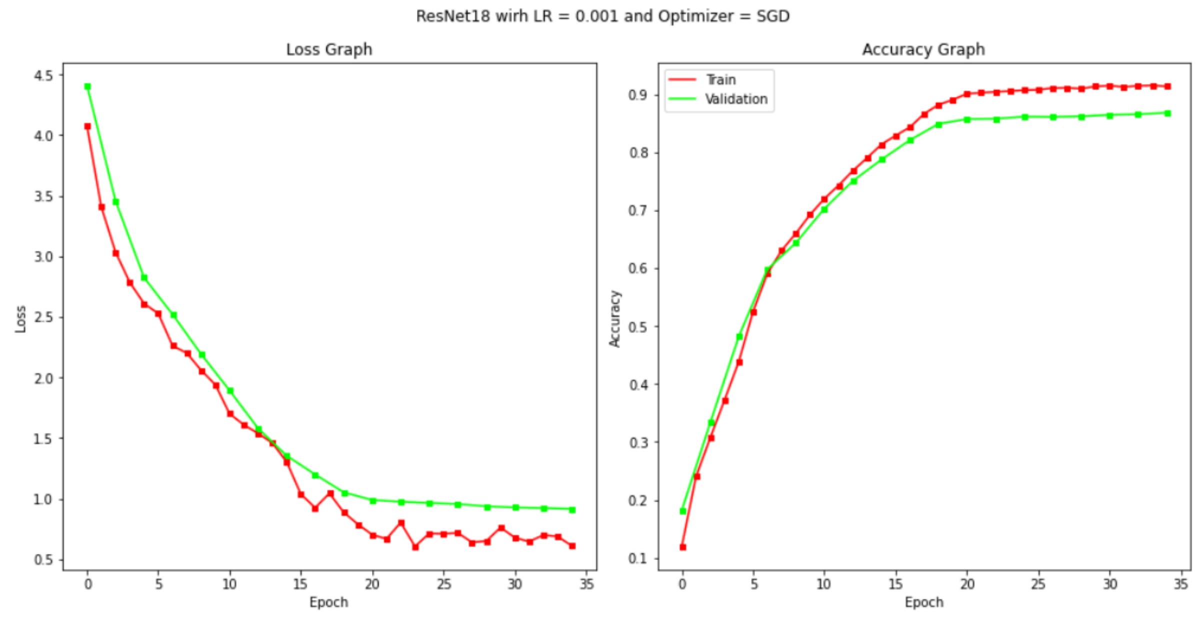
ResNet-18 is a convolutional neural network that is 18 layers deep. The network has an image input size of 224-by-224.

Even though ResNet is much deeper than VGG16, the model size is smaller due to the usage of global average pooling rather than fully-connected layers — this reduces the model size down to less than 102MB, for roughly 25.5 million parameters.

The last FC layer is added creating a new fully connected layer, with *Sequential()* method which accept an list or a OrderedDict of Modules:

```
net.fc = nn.Sequential(OrderedDict([
    ('fc1', nn.Linear(512,4096)),
    ('relu', nn.ReLU()),
    ('fc2', nn.Linear(4096,NUM_CLASSES)),
    ('output', nn.LogSoftmax(dim=1))
]))
```

The last Softmax() is used to normalize the probabilities (log probabilities) in the feature dimension, so the sum of all the probabilities will be 1.



```
Min Loss: Train: 0.605167, Val: 0.915262
Max Accuracy: Train: 0.915456, Val: 0.868257
```

It is possible to see that even with a lower number of weights ResNet18 reach performances that are similar to Alexnet, with an accuracy of 0.868 for ResNet and 0.870 for Alexnet. Since the model is smaller I used a batch size of 256 without porblems.

Score on Test dataset:

```
        Accuracy: 0.872450
```