Daniele Genta, s276922
Machine Learning and Deep Learning
Homework 2 – Report
Academic Year: 2019-2020

0 - Introduction

The goal of this homework is to perform an image classification task by implementing AlexNet on a subset of the provided Caltech-101 dataset.
The main instrument used is Python3 (and its ML related libraries) executed on the Google Colab environment, due to the computational power requested (especially in terms of GPU).
The task takes up to 10/15 minutes to be executed, depending on the parameters chosen.

Architecture

A brief recap on the architecture: AlexNet (**fig. 1**) is a simple CNN (Convolutional Neural Network), that is made up of 5 convolutional layers and 3 fully connected layers (the total is of 8 layers).
Some of the convolutional layers are followed by max-pooling layers.
In the original architecture, the last layer outputs a 1000-dimensional vector, containing the score of each class.
Inside each neuron a ReLU function is applied.

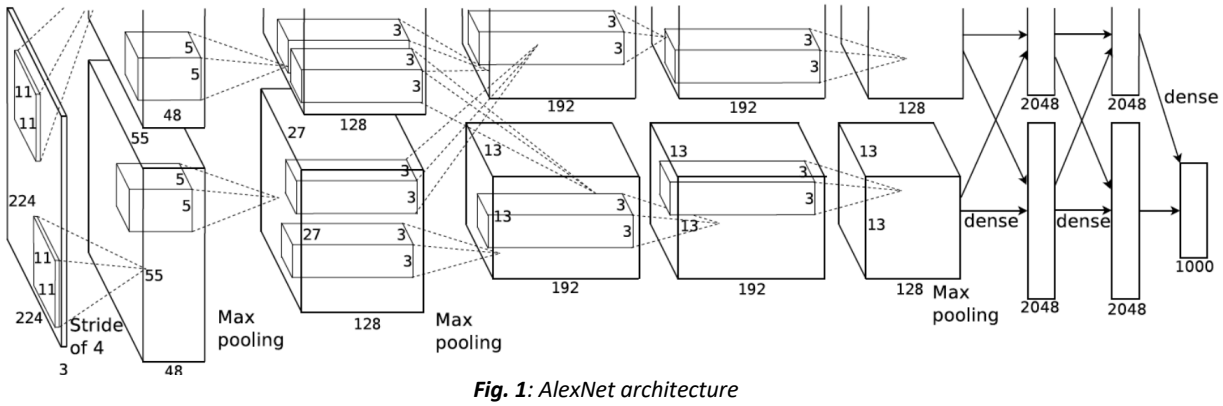

***Fig. 1**: AlexNet architecture*

Dataset

The provided dataset is Caltech-101, that is a dataset which contains images belonging to 101 categories. Each category contains from 40 to 100 images.
Roughly, each image has a size of 300 x 200 pixels (examples reported in **fig. 2**).
Actually, we've been provided with a dataset which contains 102 classes, in fact it includes an additional "BACKGROUND_Google" class, which will be deleted during the data preparation step.
Furthermore, we've been provided with a pair of text files, which guide us into the training/test set partition. To be more precise: the *train.txt* file reports 5784 instances, while the *test.txt* 2893.
The train set will be further split into train and validation set, which will be equally sized (2892 samples) and divided by exploiting stratification of the different categories.
Since AlexNet accepts as input 224x224 images, we've been provided with a snippet of code to transform and re-size the images adequately (data pre-processing step).

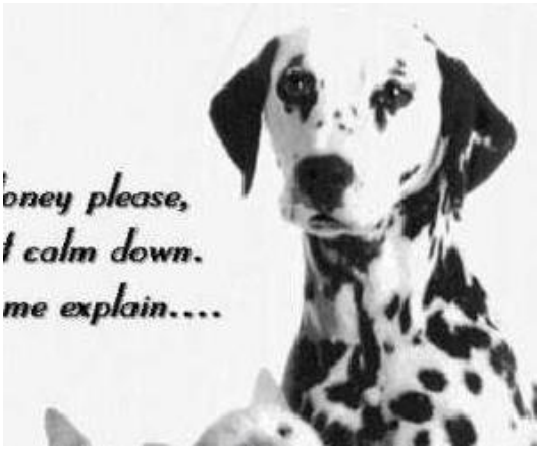

Fig. 2: *Examples of the content of Caltech-101*

1 - Data preparation

The first step of the pipeline, after having carefully studied the implementation of the given code, was to create the dataset for Caltech.
Starting from the code provided in the GitHub repository (*Caltech.py*), the first point was to create a train and a test set, according to the given text files.
Secondly, a split of the former training set into the actual train dataset and the validation set has been made, following a 50/50 proportion. In order to balance out the classes in the test and the validation set and, at the same time, shuffle the data the *StratifiedShuffleSplit* class has been used, provided by **Sklearn** library.
As a side note, it is important to acknowledge that the training set will be used to actually train the model, while the validation set will be used to properly set its the hyper-parameters.

2 – Training from scratch

In this section is reported the methodology that has been followed in order to implement the AlexNet and train it from scratch.

Network setup

The basic setup, which will serve as a baseline for all the following trials, involves a training over 30 epochs with an initial learning rate (LR) of 1e-3 and a loss function represented by the Cross-Entropy Loss function.
The gradients are updated, during the steps, via Stochastic Gradient Descent (SGD) with momentum (fixed at 0.9).
The learning rate is decaying by a factor of ten (GAMMA) after every 20 epochs (STEP-SIZE).
Furthermore, each step is executed on batches of 256 items, so each epoch involves 12 steps (which is the result of: training set size/batch size).
Finally, in the following sections of this homework some of the above parameters will be subject to a change in order to increase performances, those will be reported in detail.

A further step, prior to the training phase, is to adapt the AlexNet architecture being used to the actual dataset, that means that the last layer of the network has to be changed in order to restrict the output to the actual 101-classes (network preparation step).

Training, validation and hyper-parameters tuning

In this section training from scratch is discussed, so the methodology by which all the parameters of the network are learnt without previous knowledge.
During the training, and specifically at the end of each epoch, the model is evaluated on the validation set, by means of the accuracy score metric.
Recall that the accuracy, in a classification task, is measured as the ratio between correctly classified instances and total number of classified instances.
Implementing the validation phase makes it possible to keep track of the best scoring model in terms of accuracy and loss, during the epochs and use it in the test phase.
By analyzing this data, the hyper-parameters of the model can be tuned to increase the performances of the network.
Intuitively, in order to have a good performing neural network, it'd good to obtain a small gap between training and validation accuracy (that means that we're not overfitting the training set too much) and a smooth decreasing loss function.

By looking at **fig. 3** and **experiment 1**, some useful knowledge may be extracted.
First, the loss follows a decreasing path, this means that the model is actually converging.
The accuracy plot, though, suggests that the learning rate might be too small, so the model is converging too slowly and achieves a poor accuracy during the actual number of epochs.
So, **regarding experiment 1**, the model is not learning enough, the result is severe underfitting.
The strategy (**experiment 2**), to improve the results, consists in: increasing the learning rate, increasing the number of epochs and increasing the step-size. The other hyper-parameters will be kept fixed (batch_size, momentum, weight_decay and gamma).

Operatively, a sort of manual grid search has been performed on the three hyper-parameters of interest and the results are reported via the table and the plots below.
By increasing the learning rate, faster convergence is allowed. LR is actually the most important parameter of the model, yet changes of it must be carefully examined since a high LR could imply convergence to a sub-optimal solution or even divergence (**experiment 3**).
Increasing the number of epochs allows the model a higher number of the iterations over the whole dataset and so a higher chance to converge, but the training phase will be slower. However, as we can see from **fig. 4**, increasing too much the number of epochs is useless and actually lead to a greater overfitting.
Enlarging the step-size brings the model to converge faster, but may result is a less accurate model.
As it can be seen from the plots, in **experiment 2 and experiment 4**, the model is strongly overfitting the training set, as a solution the number of epochs could be decreased or the learning rate increased further, beside resorting in more advanced techniques as transfer learning (see **next section**).

Test phase

At the end of each training and validation phase, an evaluation of the best scoring model on the test set is performed. The operative methodology is pretty similar to the validation, but now the model is challenged with unseen data. What it can be extracted from the test phase is the overall goodness of our model, in terms of accuracy, and its ability to generalize, i.e. not to overfit.

Table 1: *Experiments in the training from scratch setting*

N	LR	#Epochs	Step-size	Accuracy result (test set)	Comments
1	1e-3	30	20	0.09	Baseline (strong underfitting)
2	1e-2	50	40	0.47	Best (still overfitting)
3	1e-1	30	20	/	Diverging
4	0.05	35	35	0.48	Strong overfitting

Fig.3: *baseline (experiment 1), accuracy and loss against epochs*

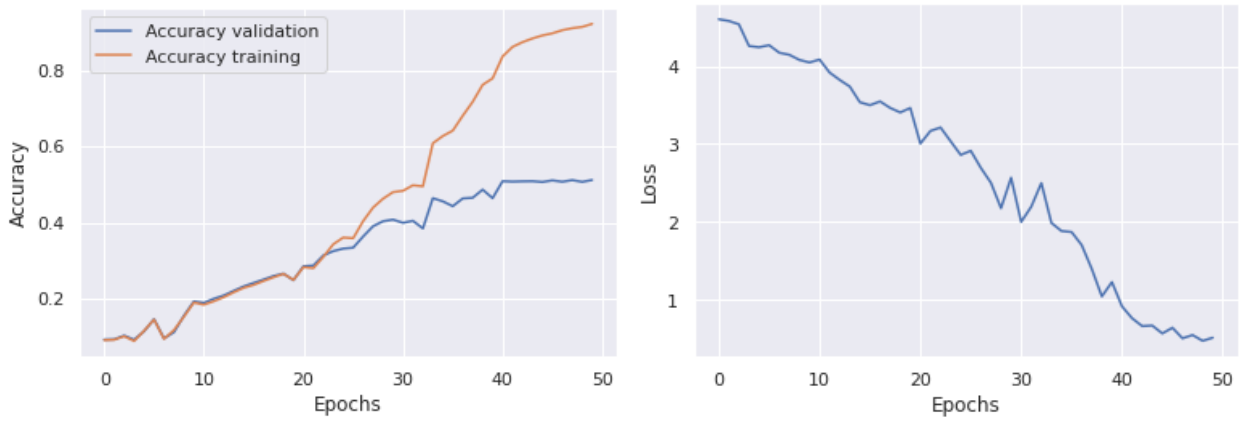

Fig.4: *experiment 2, accuracy and loss against epochs*

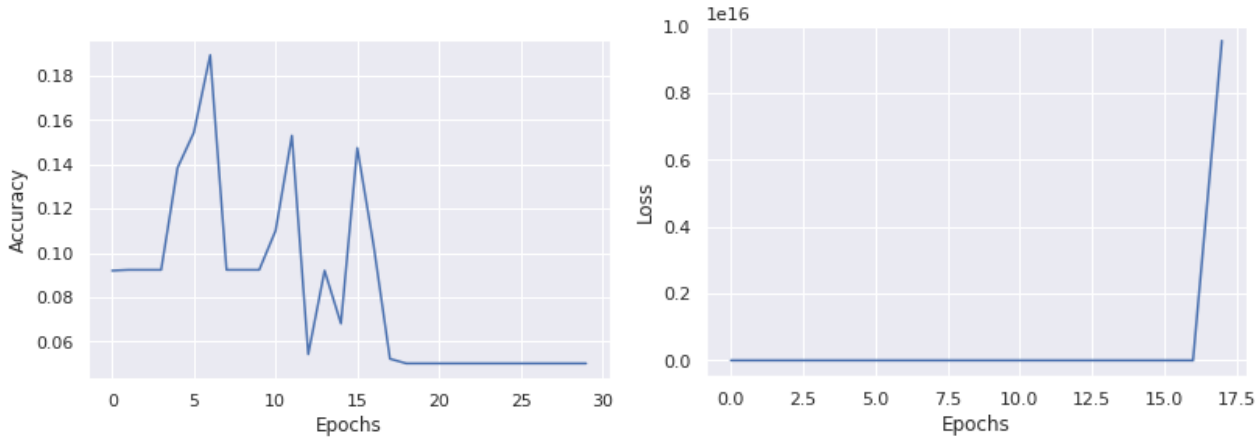

Fig.5: *experiment 3, accuracy and loss against epochs*

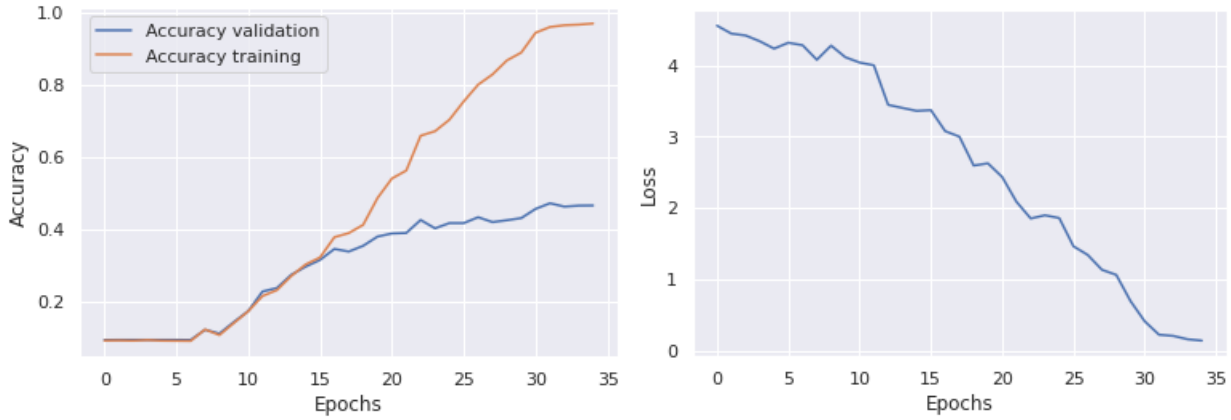

Fig.6: *experiment 4, accuracy and loss against epochs*

Based on the above consideration, what it'll be used as best reference in the next sections of the homework is **experiment 2**.
What can be seen that in this setting there's still a strong overfitting of the model that might be partially solved by increasing the size of the train dataset (see **section 4 – Data Augmentation**) or resorting to more advanced training techniques as **transfer learning**.

3 – Transfer learning

In this part of the homework the focus is on transfer learning.
Since usually neural networks, in particular CNN, need huge datasets in order to be more effective, it is likely, in the previous exposed setting, that the dataset size is the one of the bottlenecks of the problem, since the training set size contains less than 3000 items. In order to overcome this issue, an optimization technique called transfer learning is implemented.
The strategy consists in using weights that have been learnt by training neural networks on a large related dataset as a starting point for training on a small dataset.
The bottom line is that Caltech-101 is quite a small dataset and Deep Learning actually needs a very large one to train good features.

Operatively, AlexNet has been loaded with weights trained on the ImageNet dataset (which is a pretty large image dataset).
Luckily, Pytorch makes it really convenient to use pretrained weights, in fact it is sufficient to instantiate it as follows:

```
pretrainednet = alexnet(pretrained=True)
```

Furthermore, in order to make things work, the data preprocessing step has to be changed.
In particular, the data should be normalized by using ImageNet's mean and standard deviation.
The values can be found in the documentation, in particular the following piece of code has been used:

```
…
transforms.Normalize(mean = (0.485, 0.456, 0.406),std= (0.229,
                        0.224, 0.225))
                        ])
…
transforms.Normalize(mean = (0.485, 0.456, 0.406),std = (0.229,
                        0.224, 0.225))])
```

Intuitively, since the starting features are already good, it makes sense to use a tinier learning rate.
The starting point was the very best settings achieved in the previous section, then the hyper-parameters (fine tuning) of the model have been tuned in order to achieve a better accuracy.
The results are reported in the table and plots below.

Table 2: *Experiments in the transfer learning setting*

N	LR	#Epochs	Step-size	Accuracy result (test set)	Comments
1	1e-2	30	20	0.82	Best
2	1e-3	30	20	0.80	Default Settings
3	1e-1	30	20	/	Diverging
4	0.05	30	20	/	Diverging

As it can be seen from the table, the best scoring model on the test set is the same as in the previous section. It is also interesting to point out that, using transfer learning, the baseline grows to an accuracy equal to 80%, the loss function becomes smoother and the network is less affected by overfitting (see **fig. 7**, **fig. 8**).
Unsurprisingly, if the LR becomes too high (**experiment 3, experiment 4)** the model is diverging, pretty quickly.

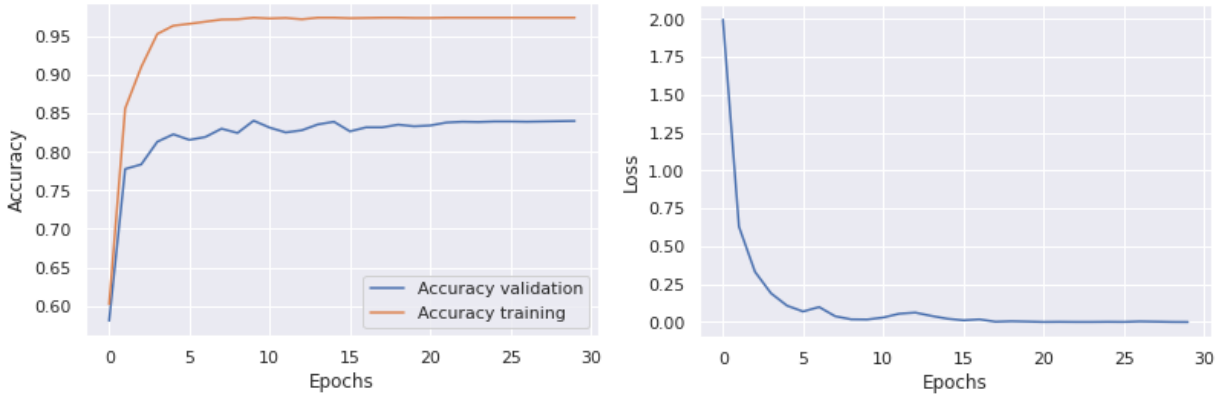

Fig.7: *experiment 1, loss and accuracy against epochs*

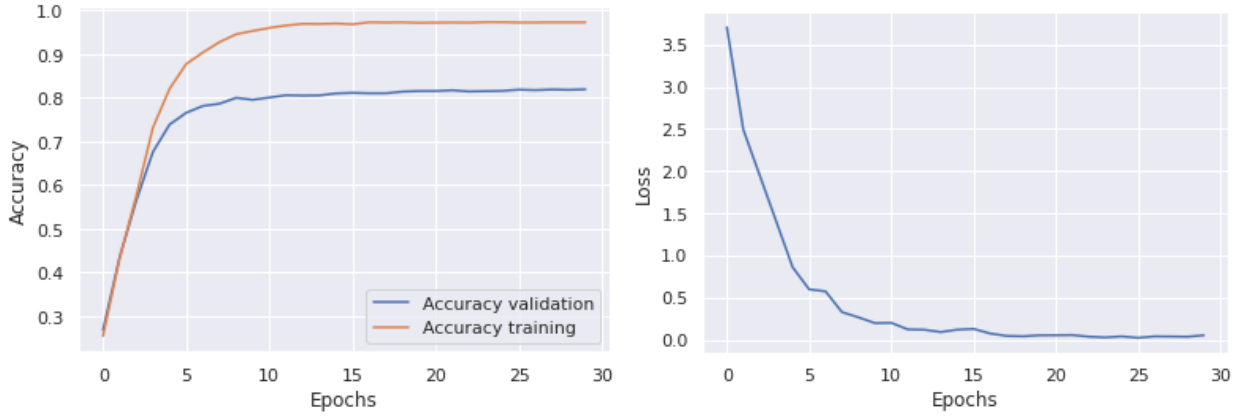

Fig.8: *experiment 2, loss and accuracy against epochs*

Freezing layers of the network

The focus of this section is to train and evaluate the best scoring model, achieved in the transfer learning setting, by freezing some parts of AlexNet.
As mentioned in the introduction, AlexNet is made up of two kind of layers that are respectively convolutional and fully connected.
Operatively, freezing a part of the network carry the consequence of speeding up the training phase, since obviously there's a smaller number of parameters to learn.
In order to train a model just on the convolutional layers or the fully connected ones, the following two lines of code could be exploited.

```
parameters_to_optimize = net.features.parameters()
parameters_to_optimize = net.classifier.parameters()
```

These parameters to optimize are used to tailor made the optimizer and scheduler objects and basically represent which parameters of the AlexNet architecture should be optimized.
The results of such experiments are reported in the following table (**table 3**).

Table 3: *Experiments in the transfer learning setting, by freezing out a part of the net*

Freeze	LR	#Epochs	Step-size	Accuracy result (test set)
Convolutional	0.01	30	40	0.82
Fully Connected	0.01	30	40	0.57

The results, which are reported in the previous table, contain some interesting insight.
Since a transfer-learning setting is being used, freezing out the convolutional layer does not harm the performances of the network, this can be explained by imaging that the first layers of the network are working with very low level features of the images (e.g. edges, gradients…), that probably have already been learnt by ImageNet properly in the pre-training phase.
Moreover, freezing the convolutional layer might be a strategy to reduce overfitting during the learning phase.
On the other hand, by freezing out the fully connected layers, i.e. the last layers, the result is poor performances. This should not come as a surprise since the very last layers provide the actual classification according to our specific dataset, Caltech, and work with very high-level features of the images.

As a bottom line: if operating in a transfer learning setting, freezing the convolutional layers of our network could be very useful, since the training time is reduced, and the performances are not negatively affected while the overfitting might be reduced.

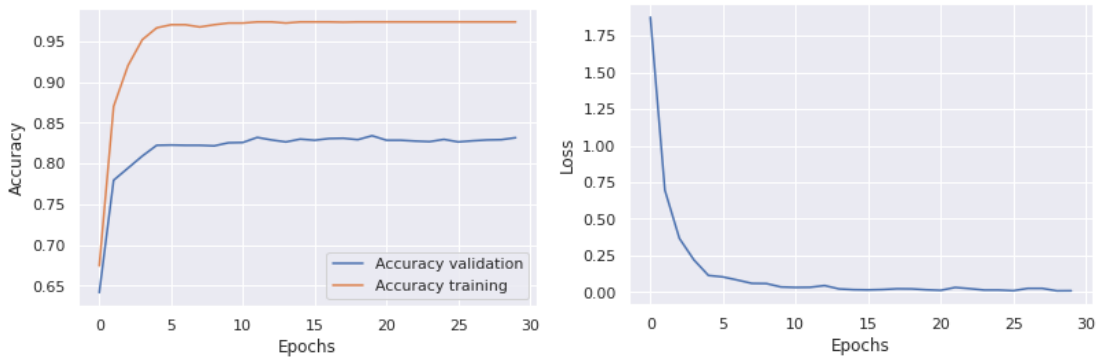

Fig.9: *Best pretrained net, convolutional layers frozen*

3 – Data augmentation

Another instrument to increase the size of the initial dataset and therefore improve the performances of the model is to exploit the images of our starting dataset by adding to it transformations of the images themselves. This strategy is called *data augmentation*.
A first implementation was already provided in the code, that performed a transformation and cropped a central patch of 224x224 pixels per each image, in order to feed proper sized pictures to the AlexNet architecture.
That has been taken a step further, in order to increase the size of our dataset.
The first transformation implemented is just slightly different one that the above-mentioned default transformation.

What has been done is to augment the dataset by inserting random crop of each image, beside the central one already performed. The ratio behind this is to teach the network to learn an image label by distinguish a part of it (occlusion).
Secondly, the size of our training set could be enhanced by flipping the images, for example horizontal flips or vertical flips are possible. Considering the nature of this dataset, a horizontal flip was assumed to be more realistic, since the likelihood of having a horizontal flipped image is higher than the one of an upside-down one.
Finally, a gray scale transformation has been implemented that, provided as input a certain picture, is able to transform it to a grays scale image.
All of the above-mentioned transformations are applied with a specified probability, that enables to choose the likelihood of performing the transformation on a given image and add it to the training set.

Operatively, the images that come from data augmentation are added to the dataset prior to the splitting into train and validation set, therefore the data augmentation is performed on the validation set too. Instead, images obtained through data augmentation are not added to the test set.
In the following table (**table 4**) are reported the result of pretrained AlexNet obtained by freezing the convolutional layers and implementing data augmentation.

Table 4: *Experiments in the transfer learning setting, by freezing out a part of the net and performing Data Augmentation*

Freeze	Additional Transformations	LR	#Epochs	Step-size	Accuracy result (test set)
Convolutional	Random crop	0.01	50	40	0.83
Convolutional	Horizontal flip	0.01	50	40	0.83
Convolutional	Random crop and Horizontal flip	0.01	50	40	0.85
Convolutional	Gray scale	0.01	50	40	0.85
Convolutional	Gray scale and Random crop	0.01	50	40	0.83
Convolutional	Gray scale and Horizontal flip	0.01	50	40	0.82

As it can be seen, the test set accuracy is slightly better if data augmentation is used and in particular some specific transformations, but the changes with respect to the previous section aren't substantial. This fact could suggest that the AlexNet architecture is the bottleneck of the pipeline and a potential improvement will be presented in the **next section**.

Extra – Beyond AlexNet

The scope of this part of the homework is to repeat the previous steps by using a convolutional neural network architecture different from AlexNet, since **Torchvision** implements lots of them.
I have chosen to perform this homework by using VGG as well.
In this homework, a 16 layers VGG, which is double as deep as the standard AlexNet analyzed in the previous sections, has been used. This network is expected to be more accurate and more costly in terms of memory consumption.
So, the network setup must be chosen accordingly to these characteristics, in fact the number of epochs has been reduced from 50 to 30 and the batch size from 256 to 64.
The scope of this section is to evaluate if a different architecture may achieve better results than AlexNet so, what it has been done, was to immediately use Transfer Learning as well as data augmentation using VGG, instead of repeating all the steps of the homework.
To be more specific, VGG using transfer learning from ImageNet has been implemented (see **section 2** of this homework) and together with data augmentation, in detail the combination that yielded the best results on AlexNet in the **section 3** has been chosen (random crop and horizontal flip).
Furthermore, as for the network setting, the best parameters found in AlexNet were used (see section 1).
The pipeline has remained the same as for the data preprocessing, data preparation, data augmentation, data split, train, validation and testing.
The results are reported below, by means of quantitative data and plots.

Table 4: *experiment using VGG architecture, transfer learning, data augmentation and keeping the convolutional layers frozen*

Architecture	Freeze	Additional Transformations	LR	#Epochs	Step-size	Accuracy result (test set)
VGG	Convolutional	Random crop and Horizontal flip	0.01	30	40	0.89

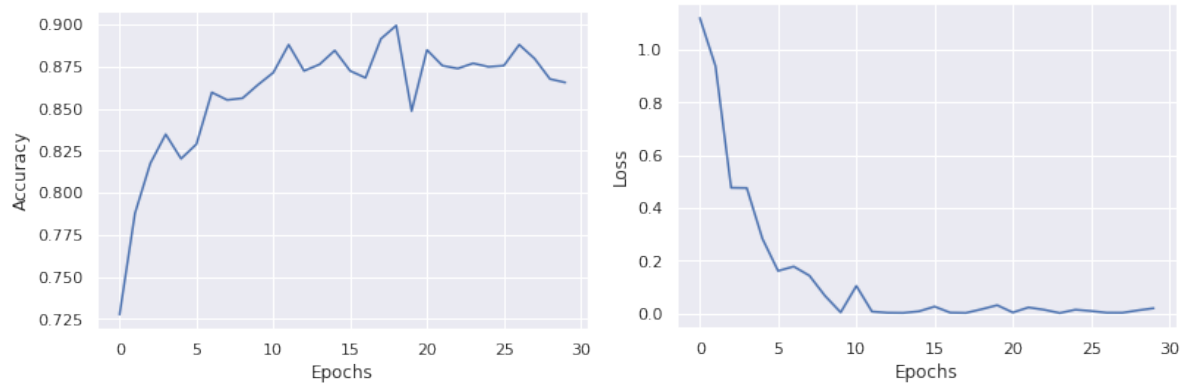

Fig.10: *VGG setting, loss and accuracy against epochs*

As expected, VGG outperforms AlexNet in terms of classification accuracy, this should not surprise us since it is a more powerful, deeper and more modern Neural Network. However, since its computational cost is very high some parameters of the network had to be changed accordingly, as the batch size and the number of epochs.
Using pretrained VGG together with data augmentation and fine tuning 89% accuracy on the test set has been reached, whereas with AlexNet the best result was 85%.
Further studies may comprehend hyper-parameters tuning and ad-hoc transformations, to enhance the capabilities of VGG on this particular homework.