Machine Learning and Artificial Intelligence
Homework 2

Ilio Di Pietro, s266393

Prof. Tatiana Tommasi, A.Y. 2019/2020

1. Introduction

Deep Learning is a subset of machine learning where artificial neural networks, algorithms inspired by the human brain, learn from large amounts of data. Similarly to how we learn from experience, the deep learning algorithm would perform a task repeatedly, each time tweaking it a little to improve the outcome. We refer to 'deep learning' because the neural networks have various (deep) layers that enable learning.
Deep Neural Networks (DNN) constitute a framework for different machine learning algorithms to allow them to process complex data inputs. In DNN, each level extracts features from the output of the previous one. In this homework we focused on particular types of DNN which are **Convolutional Neural Networks (CNN)** with **AlexNet** as a case study.

1.1. Homework Sub-tasks

- Prepare our dataset filtering all the images coming from the 'BACKGROUND_Google' class. Split it using the provided train/test splits for training and test set. Further divide the training set in train and validation;
- Train the AlexNet network, first evaluating the model on validation set after each epoch to choose the best hyperparameters, then using the best result on test set;
- Repeat using Transfer Learning to improve performances;
- Using the best hyperparameters obtained from Transfer Learning, repeat the training using only a subset of AlexNet's layers(freezing part);
- Apply Data Augmentation on training set to provide more data to the net;
- Use of ResNet to experiment other types of CNN.

2. Data Preparation

The dataset used for this homework is Caltech-101. It's a very small dataset for deep learning algorithms, containing pictures of object belonging to 101 classes (about 40 to 800 images per class).
The starting code that was given to us created the dataset using the ImageFolder class that generates the train and test set with an arbitrary criterion using the Subset class. Our task in this first part was to create a new class that implements a defined split, using the provided text files called train.txt and test.txt. In this class we also implemented the methods *__getitem__(self, index)* which applies transformation to images to resize, center crop and make them of the right size (224x224) for Alexnet(The original size of the images in the dataset is 300x200 pixel). We also filtered the 'BACKGROUND_Google' class.
The validation test was created taking half of the training set and being careful to not filter out entire classes: we first sorted our initial training set and after we piked up(using the Subset class) for validation set the images that had even indexes, while for training set images that had odd ones, in order to keep the number of images per category as balanced as possible.The resulting number of samples in the training and validation sets is 2892, 2893 for the test set.

2.1. Programming Environment

The proposed solution is obtained using Python 3.7 programming language. Google Colab framework is used to compute the requested data. Moreover, the most used libraries and packages are:

- PyTorch, an open-source machine learning library and deep learning framework for Python, based on Torch. The used modules and classes are nn and optim, specialized on creation and training of neural networks;
- matplotlib.pyplot: matplotlib is a Python 2D plotting library; pyplot provides a MATLAB-like interface;
- numpy, a package for scientific computing with Python;
- torchvision, a package made of popular datatsets and common image transformations for computer vision.

3. Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNN) are a class of Deep Neural Networks that are applied to analyzing visual images. They are usually composed by a set of layers that can be grouped by their functionalities in:

- **Convolutional layer**: compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume;
- **Non-Linear layer**: apply an elementwise activation function. This leaves the size of the volume unchanged;
- **Pooling layer**: perform a downsampling operation along the spatial dimensions (width, height);
- **Fully Connected layer**: compute the class scores, resulting in volume of size [1x1xC], where C is the number of classes. Each neuron in this layer will be connected to all the numbers in the previous volume.

In this way, CNN transform the original image from the original pixel matrix to the final class scores. It should be noted that some layers contain parameters and other don't. In particular, the convolutional and fully connected layers perform transformations that are a function of some parameters (the weights and biases of the neurons). On the other hand, the non linear and pooling layers will implement a fixed function.

3.1. AlexNet

AlexNet is a particular type of CNN which which has 60 million parameters and 500,000 neurons, consists of 5 convolutional layers, some of which are followed by max-pooling layers, and 3 fully connected layers with a final 1000-way softmax. Its architecture is summarized in Figure[1].
After each convolutional layers is applied the non-saturating ReLU activation function, defined as:

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

which showed improved training performance over tanh and sigmoid(x represent the input data).
The aim of this experience is to use this network on Caltech-101 dataset and to apply some special techniques to improve the performances of the net.

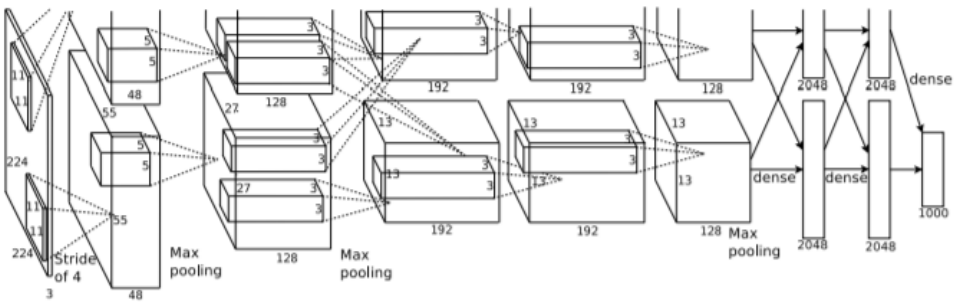

Figure 1: AlexNet architecture

4. Training from scratch

The first part of the homework concerns in training AlexNet using the train set(computed in the way described in the Data Preparation part) and tune in a proper way a set of hyperparameters, i.e. all of those parameters dealing with the structure of the net and of the training process; in particular we focused on:

- **Learning rate**: hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated;
- **Batch size**: number of samples processed before the model is updated;
- **Number of epochs**: total number of iterations over dataset;
- **Step size**: indicates how many epochs must pass before decreasing learning rate.

The other hyperparameters (that were left as constants) are:

- **Momentum**: hyperparameter for SGD[1], fixed at 0.9;
- **Weight Decay**: term of regularization fixed at $5 * 10^{-5}$;
- **Gamma**: multiplicative factor for learning rate step-down to apply after a number of epochs equal to Step Size. Fixed at 0.1.

The tuning of hyperparameters can be very tough since there is not a precise or analytic correlation between them:

- Choosing the learning rate is challenging as a value too small may result in a long training process that could get stuck, whereas a value too large may result in learning a sub-optimal set of weights too fast or an unstable training process;
- The choice of the best batch size is difficult, indeed having a small batch size requires less memory because, step by step, we train our net with few samples. Moreover the net training faster because we update the weights after each propagation. The disadvantages of a small value is that the smaller the batch is, the less accurate the estimation of the gradient will be;
- Tuning of epochs is important to reach a good level of accuracy on validation, but there is no guarantee that increasing the number of epochs provides a better result than a lesser number.

Moreover, since in deep learning compuational cost is not negligible, it is not possible to try all possible combinations to achieve the best result; it is mandatory to find a method to tune these hyperparameters. Whit the values already present in the code for batch size(256), number of epochs(30) and step size(20), we started to change the learning rate, which is the most influencing factor. At first, after some runs, the values that gave the best result in terms of accuracy on train-validation and loss were between 10^{-4} and 10^{-1}.
Batch size should be change accordingly to the learning rate, looking for equivalent increment to get comparable results between different combination so at this point we tried different values between 128 and 512. Coherently with this changes, we were able to decrease the range of possible good values for learning rate to $[10^{-3}, 10^{-1}]$.
Number of epochs has been lowered to decrease computational time but at the end we saw that with a value of 30 we reached a good compromise. The step size has been changed accordingly with the epochs.
At the end the 2 best set of values chosen were:

- **Learning rate**: $3 * 10^{-2}$;
- **Batch size**: 320;
- **Number of epochs**: 30;.
- **Step size**: 25.

[1]Stochastic Gradient Descent(SGD) is an optimization algorithm that estimates the error gradient for the current state of the model using examples from the training dataset, then updates the weights of the model using the back-propagation of errors algorithm, referred to as simply backpropagation.

and

- **Learning rate**: $2 * 10^{-2}$;
- **Batch size**: 256;
- **Number of epochs**: 30;
- **Step size**: 25.

The obtained results are shown in Figure[2] and Figure[3].
However, all the model show a consistent gap between training and validation accuracy, probably due to the small size of the training dataset used. This gap can show us the presence of overfitting if too high.

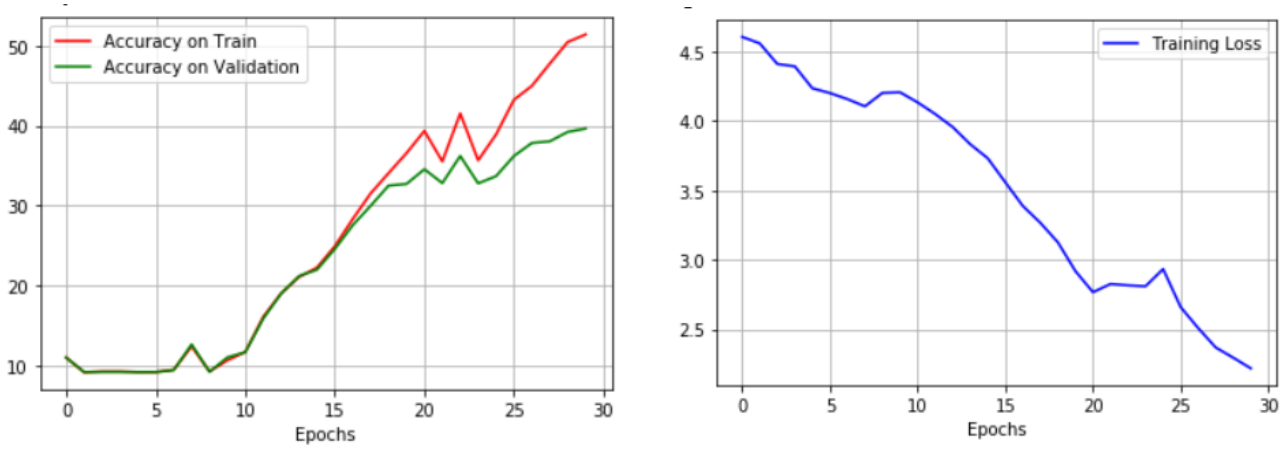

Figure 2: Accuracy and loss with first set of hyperparameters

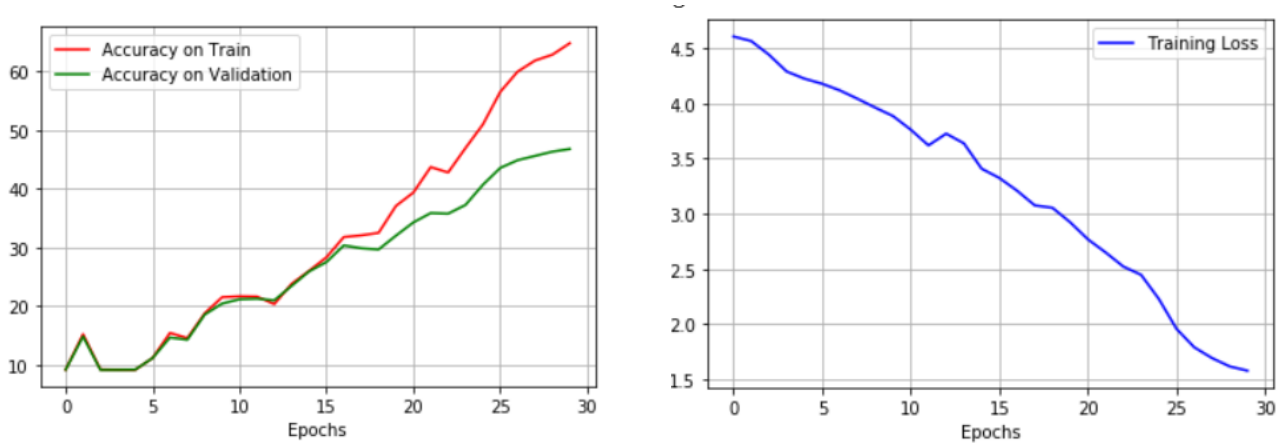

Figure 3: Accuracy and loss with second set of hyperparameters

The implementation of the code is such that at the end there is a part dedicated to testing of the best model on the test set. At first we set the correct values of the chosen hyperparameters, then we trained a new net in such a way that weights inside it will be consistent with them and at the end it is performed an evaluation on the test set. Table[1] shows the the numeric values obtained.

	Accuracy on train(%)	Accuracy on val(%)	Loss
first set	51	39	2.29
second set	64	46	1.71

Table 1: Accuracy and Loss for the 2 set of hyperparameters

From these results the best set of hyperparameters are the second ones, and the relative accuracy on test set is 50.6%.

5. Transfer Learning

The purpose of Transfer Learning is to take advantage of previous knowledge to improve the performances of the network. Deep Learning needs a large amount of data to achieve good results but in our case Caltech-101 it's a too small dataset and the low accuracy obtained in the previous point proves it. The solution is to apply Transfer Learning on AlexNet as a starting point to start the training phase on Caltech-101. AlexNet is trained on a very big dataset called ImageNet, which contains 1.2 million high-resolution images. Only By setting the flag *pretrained* to True is possible to pre-train weights of the net on it.
Using this technique is extremely powerful but now AlexNet needs as input ImageNet-like images, which means that we have to apply on our dataset a new kind of normalization with different mean and standard deviation. At this point the hyperparameter tuning was performed in the same way as explained in the previous point. the 3 set chosen are:
1

- **Learning rate**: $1 * 10^{-3}$;
- **Batch size**: 150;
- **Number of epochs**: 30;
- **Step size**: 25.

2

- **Learning rate**: $1 * 10^{-4}$;
- **Batch size**: 256;
- **Number of epochs**: 30;
- **Step size**: 25.

3

- **Learning rate**: $6 * 10^{-3}$;
- **Batch size**: 512;
- **Number of epochs**: 30;
- **Step size**: 25.

The results are shown in Figure[4], Figure[5] and Figure[6]. As we can see now the accuracy becomes higher and loss becomes lower, meaning that the model can apply in a good way its previous knowledge to our dataset. Table[2] shows the numeric values obtained.

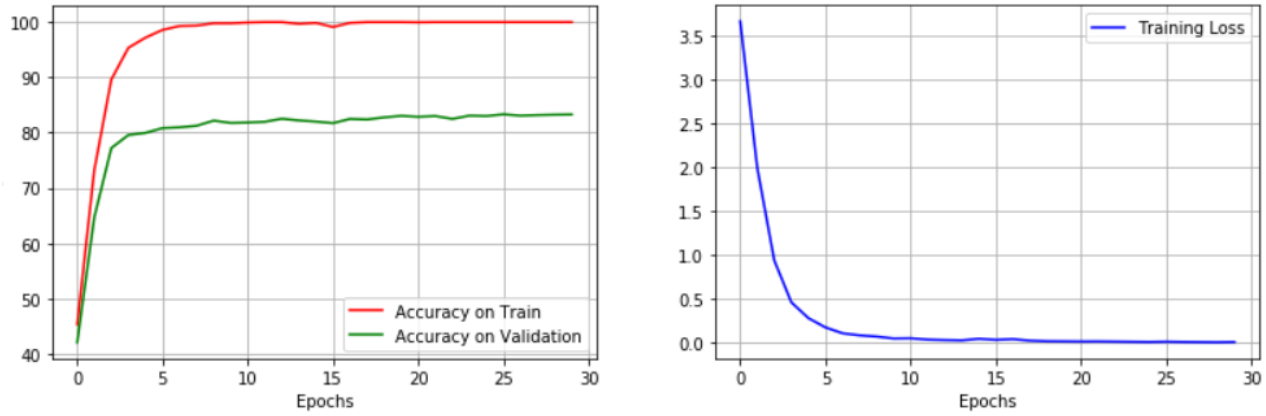

Figure 4: Accuracy and loss with the first set of hyperparameters with Transfer Learning

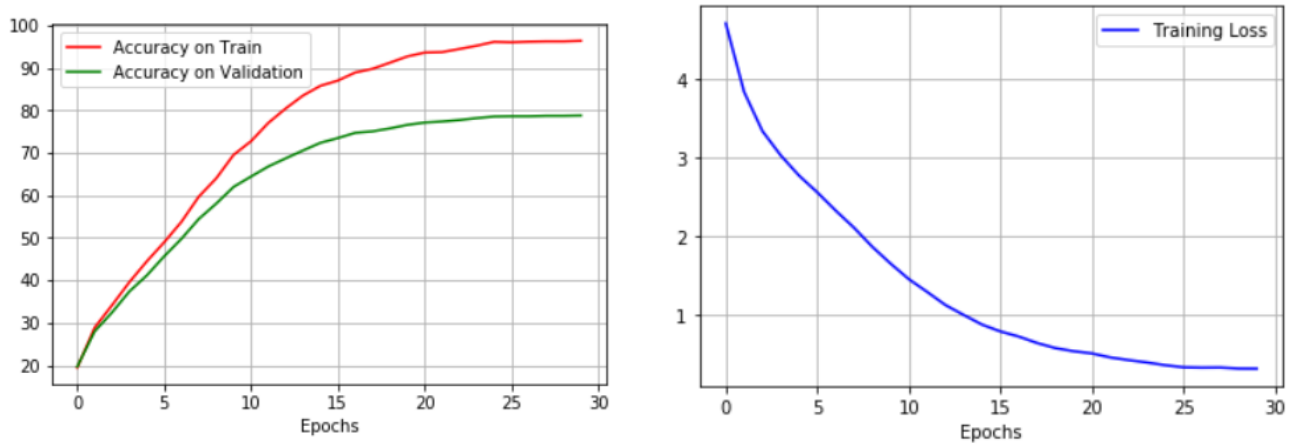

Figure 5: Accuracy and loss with the second set of hyperparameters with Transfer Learning

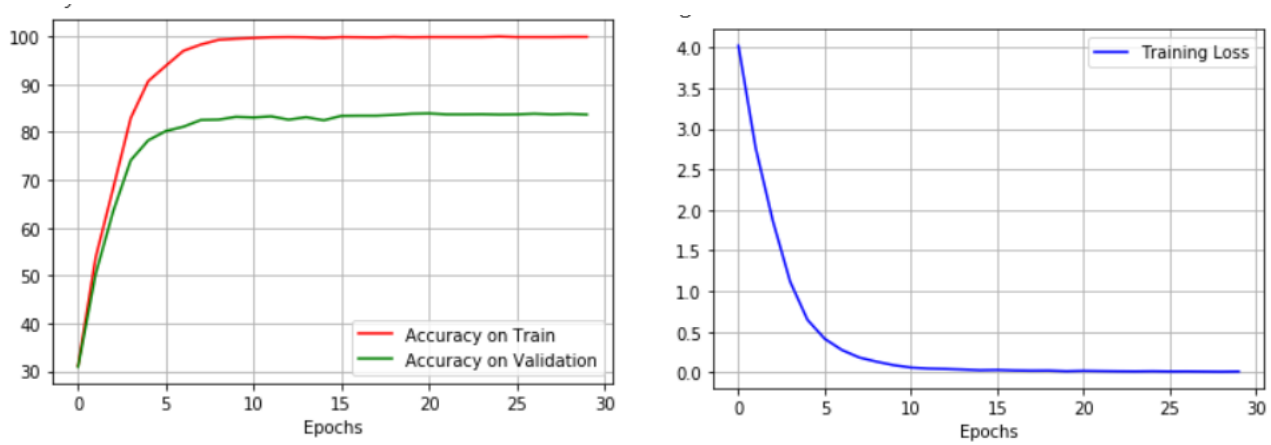

Figure 6: Accuracy and loss with the third set of hyperparameters with Transfer Learning

	Accuracy on train(%)	Accuracy on val(%)	Loss
first set	99	83	0.008
second set	96	78	0.312
third set	99	83	0.010

Table 2: Accuracy and Loss for the 3 set of hyperparameters (Transfer Learning)

From these results the best set of hyperparameters are the first ones, and the relative accuracy on test set is 83.1%.

5.1. Freeze of layers

Sometimes, to reduce training time, one way is to freeze some layers. Freezing a layer prevents its weights from being modified. This technique is often used with transfer learning, where the base model(trained on some other dataset)is frozen. In this way the overall model is already trained on ImageNet dataset but we will not update weights using Caltech-101.
With the best hyperparameters obtained with transfer learning we have experimented 2 different ways of freezing the layers:

- **Freeze of convolutional layers**: we have trained only the fully connected layers by passing to the optimizer function as parameters to optimize only *new_net.classifier.parameters()*. This means that the convlolutional layers will not update weights or extract features from our dataset. Optimizing only a subset of our net means that we will reach a good compromise on accuracy only if the dataset already used by AlexNet is comparable with Clatech-101; in this case the fully connected layers are able anyway to perform a good classification, so we obtained a comparable accuracy with respect to simple transfer learning, with the advantage of having reduced the training time. Figure[7] shows these results;

- **Freeze of fully connected layers**: we are now training only the convolutional layers by passing to the optimizer function as parameters to optimize only *new_net.features.parameters()*. AlexNet will now update the weights of the convolutional layers but not the parameters of the fully connected ones. The results (see Figure[8]) show us that there is a worsening of the accuracy and loss, because the last layer need to be optimized with respect to our specific dataset to perform a good classification.

Table[3] shows the numeric values obtained.

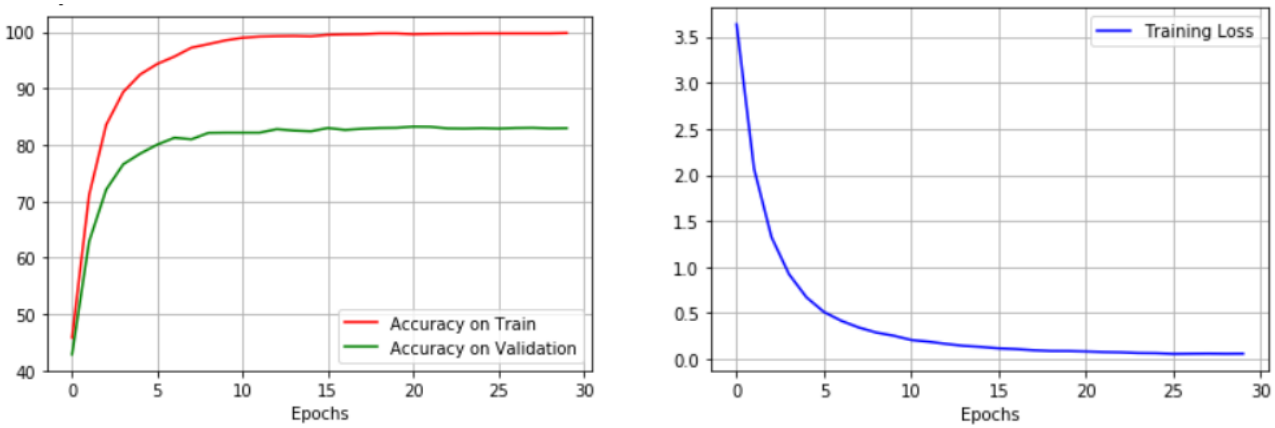

Figure 7: Accuracy on train and validation set and Loss with freeze of convolutional layers

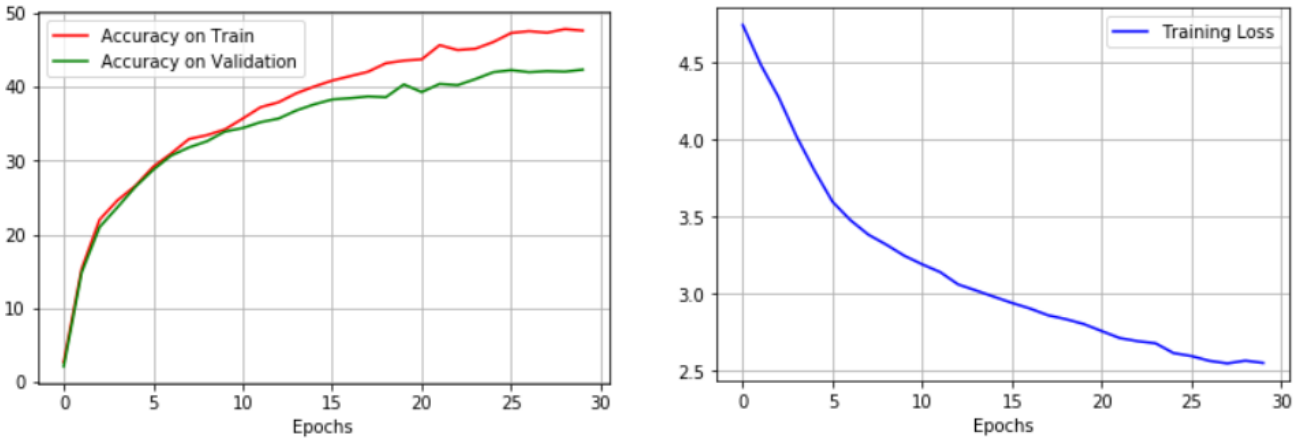

Figure 8: Accuracy on train and validation set and Loss with freeze of fully connected layers

	Accuracy on train(%)	Accuracy on val(%)	Loss	Accuracy on test(%)
conv. freeze	99	82	0.054	83.7
fully conn. freze	47	42	2.427	41.7

Table 3: Accuracy and Loss for different types of freezing

6. Data Augmentation

The easiest and most common method to reduce overfitting is to artificially enlarge the dataset using label-preserving transformations. Their introduction should make the classification problem harder for the network during the training session because more various inputs are provided; at the same time, the testing phase will be easier.
The set of transformations that we applied are:
1

- **RandomResizedCrop(224)**:Crop the given PIL Image to random size and aspect ratio;
- **RandomGrayscale()**: Randomly convert image to grayscale.

2

- **RandomResizedCrop(224)**: Crop the given PIL Image to random size and aspect ratio;
- **RandomHorizontalFlip()**: Horizontally flip the given PIL Image randomly.

3

- **RandomResizedCrop(224)**:Crop the given PIL Image to random size and aspect ratio;
- **RandomErasing()**: Randomly selects a rectangle region in an image and erases its pixels.

The Dataloader method applies the transformation when it's called and randomly generates new images without stores it.
The effect of those transformations was tested separately and the results are depicted in Figure[9], Figure[10] and Figure[11]. The accuracy and loss values are shown in Table[4].

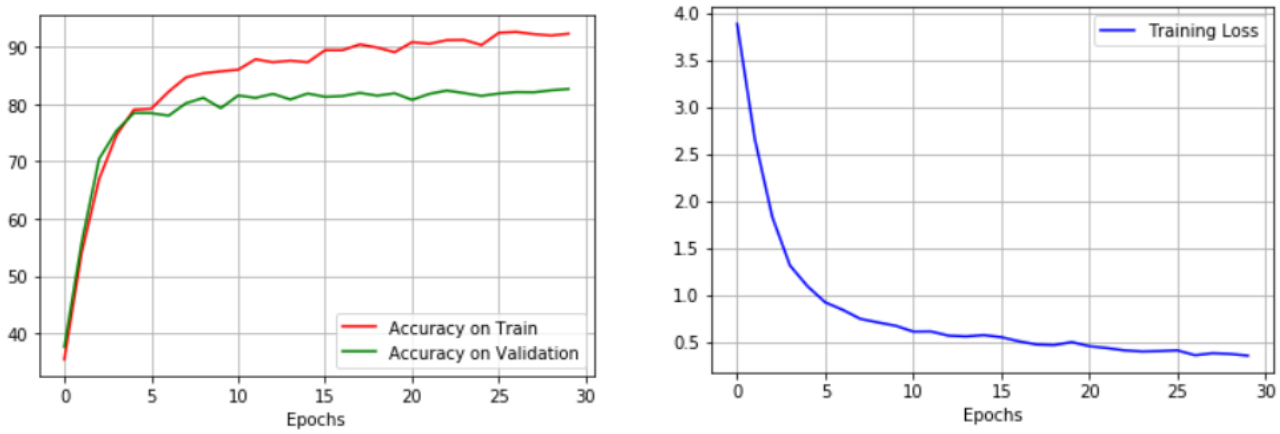

Figure 9: Accuracy on train and validation set and Loss with Data Augmentation (first transformation)

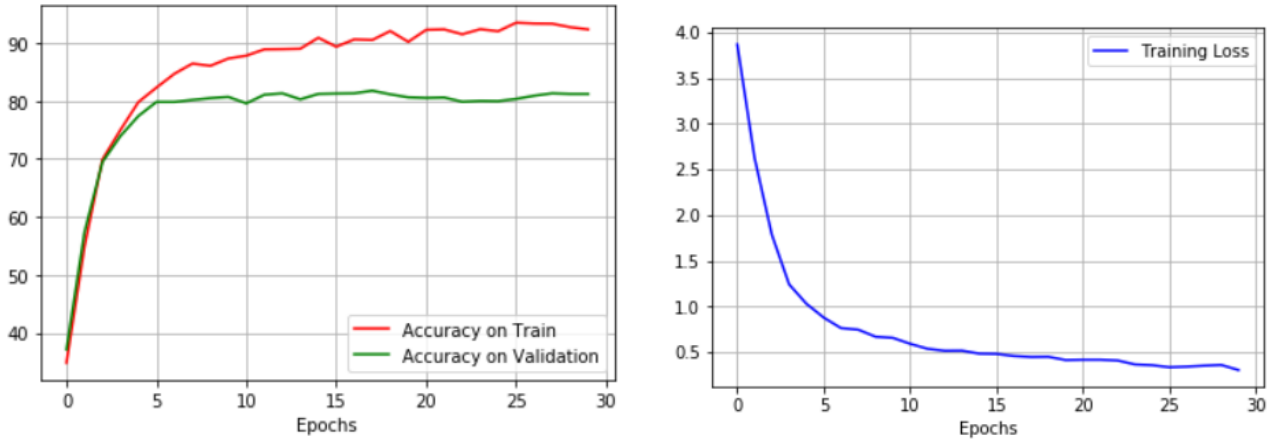

Figure 10: Accuracy on train and validation set and Loss with Data Augmentation (second transformation)

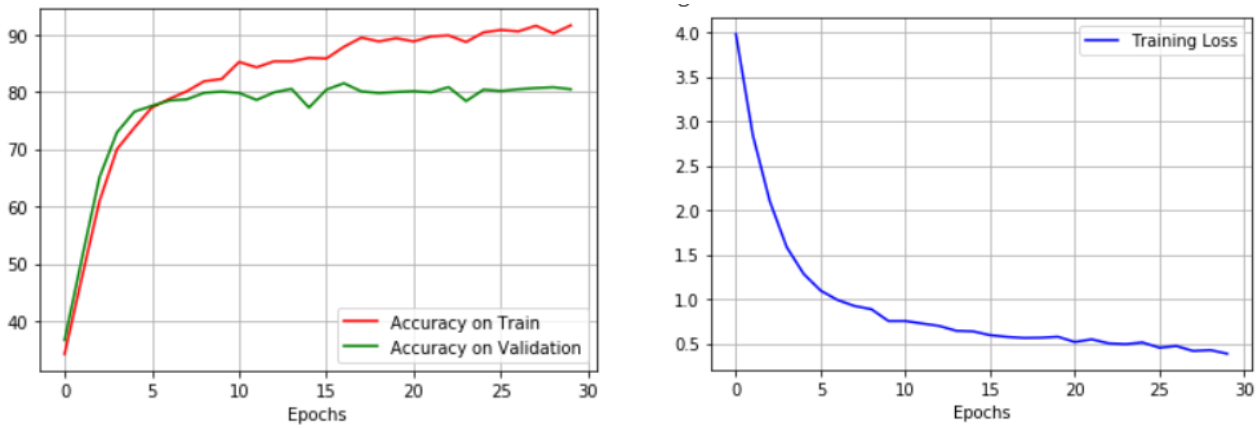

Figure 11: Accuracy on train and validation set and Loss with with Data Augmentation (third transformation)

All the three networks seems to have a common behavior with respect to the previous solution without data augmentation, probably due to the fact that they all are trained on a dataset which is already large enough to achieve good results on classification.

	Accuracy on train(%)	Accuracy on val(%)	Loss	Accuracy on test(%)
first transf.	92	82	0.311	82.3
second transf.	92	81	0.314	81.3
third transf.	91	80	0.427	81.8

Table 4: Accuracy and Loss for the 3 set of transformation (Data Augmentation)

7. Extra: ResNets

Residual Neural Network (ResNet) is a Convolutional Neural Network (CNN) architecture which can support hundreds or more convolutional layers. It introduce a different architecture from what we have seen since now. Inside there are two main paths the input can follow: the first one is a traditional convolutional network; the second one skips all the connections and connects the input to the output produced by the current layer so that they can be summed. In this way, the resulting total output is simply an update of the initial input. This implementation can be explained as a solution to the "vanishing gradient", a problem that concerns all Neural networks. They train via backpropagation, which relies on gradient descent to find the optimal weights that minimize the loss function. When more layers are added, repeated multiplication of their derivatives eventually makes the gradient infinitesimally small, meaning additional layers won't improve the performance. In ResNet, during training process, the identical layers are skipped, reusing the activation functions from the previous layers.
The strength of the ResNet architecture is its ability to generalize well to different datasets and problems.
In this homework we used in particular ResNet-101 on our dataset, doing some of the precedent tasks to test the overall performance with respect to AlexNet. In the code we implemented ResNet everytime calling the function *torch.hub.load('pytorch/vision:v0.4.2', 'resnet101')* instead of *alexnet()*.
Note that due to the fact that ResNet is a very deep net, it will consume many GPU resurces, so to avoid an out of memory error, we drastically reduce the batch size, choosing 32 as a good compromise.
The results obtained are shown below.

7.1. Training from scratch

In this first case ResNet is used without pre-train it on ImageNet. The set of hyperparameters chosen was:

- **Learning rate**: 10^{-2};
- **Batch size**: 32;
- **Number of epochs**: 30;
- **Step size**: 25.

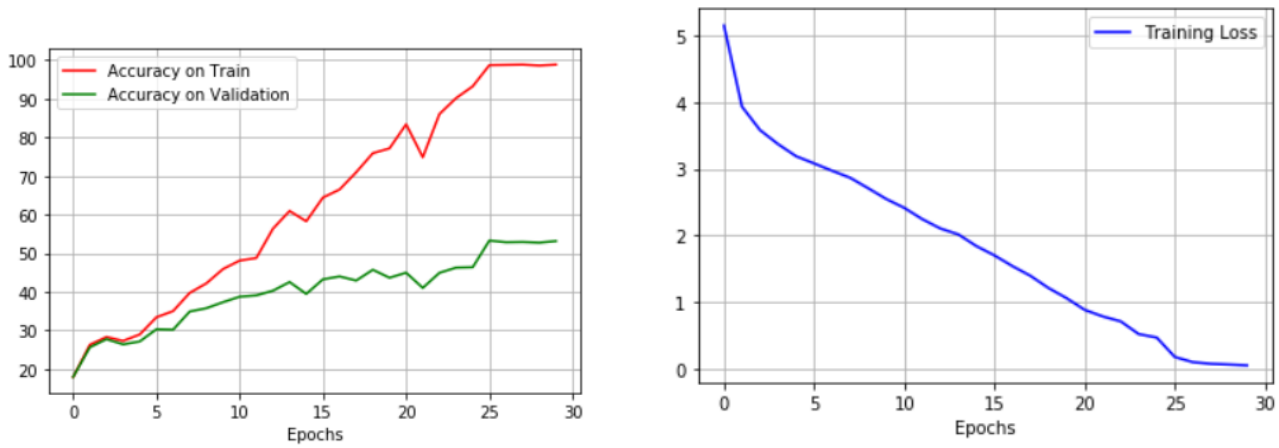

Figure 12: Accuracy on train and validation set and Loss with ResNet

An analysis of the results shows that ResNet reach levels of accuracy higher with respect to AlexNet, despite the gap between accuracy on train and on validation which is very large, making us think of a probable overfitting, due to the fact that ResNet is a more complex architecture, which needs a larger dataset to tune its parameters. Regarding the loss, its trend reach lower level with respect to AlexNet.

Accuracy on train(%)	Accuracy on val(%)	Loss	Accuracy on test(%)
98	53	0.0199	52.7

Table 5: Accuracy and Loss with ResNet

7.2. Transfer Learning

Transfer Learning is now applied and the training is repeated: the hyperparameters chosen was:

- **Learning rate**: 10^{-2};
- **Batch size**: 32;
- **Number of epochs**: 15;
- **Step size**: 12.

Note that the number of epochs, and consequently the step size, have been decreased, given the rapid convergence of the net. Moreover the gap between train and validation accuracy is much more lower with respect to AlexNet: the model is more accurate as we expected, thanks to the large amount of data taken from ImageNet.

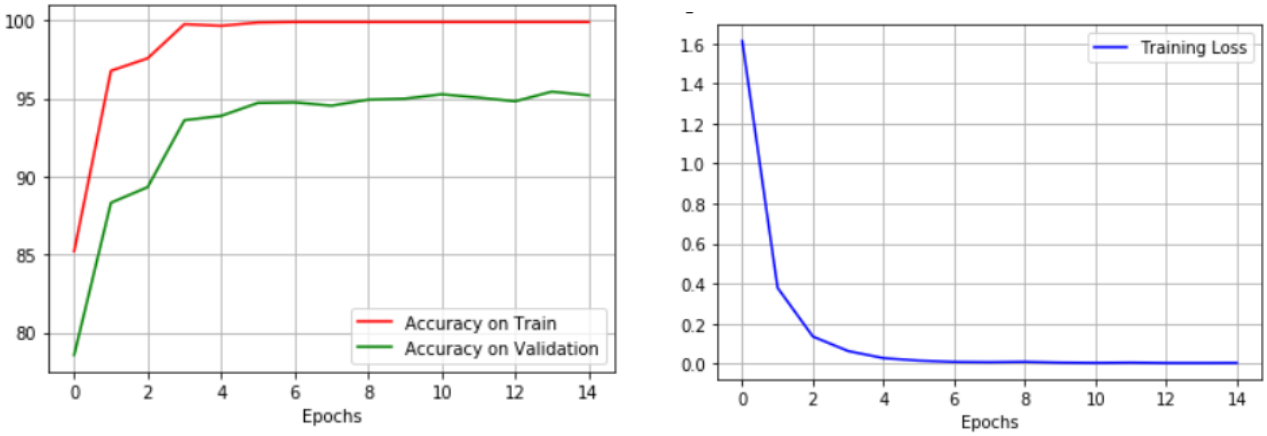

Figure 13: Accuracy on train and validation set and Loss with ResNet (Transfer Learning)

Accuracy on train(%)	Accuracy on val(%)	Loss	Accuracy on test(%)
99	95	0.0004	94.9

Table 6: Accuracy and Loss with ResNet (Transfer Learning)

8. Conclusions

The different tasks of this homework allowed us to better understand advantages and disadvantages related to the wide world of deep learning. Convolutional Neural Networks have been used to try to understand tough aspects for the training procedure, such as the tuning of the hyperparameters or the small amount of training data available. Different solution to this last problem have been implemented, such as Transfer Learning, which has greatly improved the results, or Data Augmentation, which allowed us to increase even more our dataset. Moreover we reached a deeper knowledge about the architecture of AlexNet, thanks to the study of its layers and how to freeze part of them in order to decrease training time without lose accuracy.
At the end we have also learned how other more complex CNNs works,(in particular ResNet) and tried to make comparison with respect to AlexNet.