Homework 2

S274563 Marco Gullotto

April 2020

1 Introduction

The premise to the homework that will be carried out later, consists of a brief introduction relating to the environment and the tools that will be used to perform the assigned task. We must first consider that in this case a great computational effort is needed to train deep models that will have to be trained using GPUs. The Google project called Colab, therefore, fits perfectly in this case because it allows you to use (although with some limitations) Google GPUs to train the models for free. Colab also allows you to take advantage of 12 GB of DDR5 RAM and about 70 GB of SSD. It may seem that we have many resources but, as we will find out later, the memory required to handle this type of problem is very huge, therefore many arrangements are necessary. Deep neural models such as AlexNet, ResNet etc. are very well-known models, and many frameworks further implement them. In this homework, we should use the Pytorch library developed by Facebook. It's not a common choice, since many developers prefer to work with Keras and TensorFlow which are developed by Google instead. The main advantage that derives from the use of Pytorch consists in the fact that, with this library, it is possible to work simultaneously at a high and low level. Vice versa, using the models produced by Google, if we want to work at a higher level we'd have to use Keras, otherwise it'd be necessary to use TensorFlow. The main drawback of the Facebook library is that, as it is not very well-known, it is difficult to find, online, feedback or suggestions if a problem arises. The dataset (ds) used in this homework is called Caltech-101, as it was developed by some researchers from the Californian Institute of Technology. In this ds, as we can imagine, there are images of objects belonging to 101 different categories. Most of the categories catalog about 50 images. The size of each image is approximately 300 x 200 pixels. It's not a huge dataset, so achieving a good accuracy score could be difficult. The model used at first is AlexNet: it is an old model designed designed by Alex Krizhevsky in 2012. AlexNet receives 224 X 224 X 3 images as input, and each of them passes through 8 different layers. The firsts are convolutional layers, followed by max-pooling layers, and the last three are fully connected layers. Non-linearity is introduced by the ReLU activation function. It is certainly not the state-of-art model with which we can deal with, but it will still represent our first baseline.

2 Data Preparation

A basic framework that can be used to obtain datas has already been provided. The code template used allows us to download all the images from the "GitHub" repository but, at this point, it is necessary to perform two operations:

1. divide the ds into two parts: train and test,
2. filter out the BACKGROUND images.

In order to do both at the same time, we adopt an ad-hoc dataset class called Caltech (which extends "torchvision.datasets.VisionDataset"). In the "init" method of this class, all the images except the background ones are retrieved. In this phase, it is also necessary to divide the ds into two parts following the instructions provided in the "train / test" file in the "GitHub" repository.

```
f = open(root + split,"r")

for image in f.read().splitlines():
    if image.split("/")[0] != "BACKGROUND_Google":
        self.array.append(image)
        self.labels.append(self.target_transform.transform \
            ([image.split("/")[0]])[0])

f.close()
self.labels = np.array(self.labels)
self.array = np.array(self.array)
```

The "get item" method therefore retrieves the image and establishes the link with the corresponding label. Since in the later stages we will deal with data augmentation, this method can apply a transformation to the image if a "torchvision.transforms.compose" is provided. But we will come back to this later.

```
image = pil_loader("./Homework2-Caltech101/101_ObjectCategories/"\
                   + self.array[index])
label = self.target_transform.transform([self.array[index] \
                          .split("/")[0]])[0]

if self.transform is not None:
    image = self.transform(image)

return image, label
```

3 Training from scratch

In the PyTorch library it is possible to load the CNN in pre-trained mode or not. This means that we can load a net with randomly chosen weights and,

in this case, we want to train our model from scratch. Otherwise, we can use a CNN that is already trained in the "imagenet" ds, so the weights are already initialized in an "clever" way. In this step, we must try to train an AlexNet from scratch. But before that, we have to choose which loss, optimizer and scheduler policy we should follow. A first attempt was made by adopting "CrossEntropyLoss" which is very similar to Negative log-likelihood loss, and is commonly used to train a classification problem with C classes. This can be described by the expression:

$loss(x, class) = -log(\frac{e^{x[class]}}{\sum_j e^{x[j]}})$

As optimizer, we implement the stochastic gradient descent with initial learning rate 1e-3, momentum equal to 0.9 and, for regularization, we reduce the weight-decay bringing it to a very low value such as 5e-5 . Finally, we choose a optimizer who decreases the value of the learning rate by 1/10 after 20 epochs. We are therefore ready to train the network. As shown in Figure 1, the basic structure of CNN has been adapted to our purpose; the last fully connected layer now has size 101, which corresponds to the number of classes in our problem. Now, using the code template provided by the homework, we can train our network.

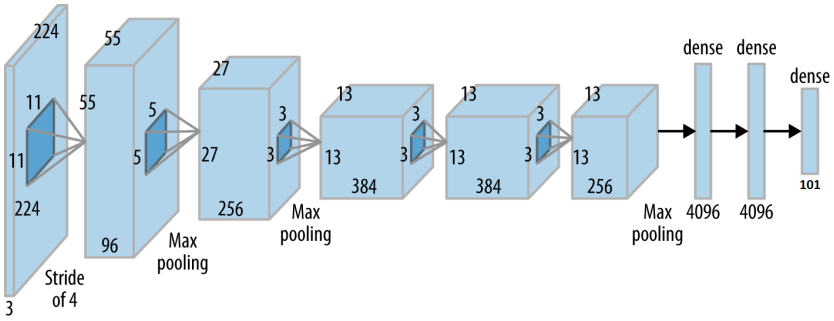

Figure 1: AlexNet structure

In general, to carry out the validation, we split the ds in train and validation set to obtain the best hyperparameters possible, and then train our model again on all the ds with this configuration. However, when we are dealing with CNN, it is difficult to take all these steps because this would lead to an excessive lengthening of the model's training time. In this case, therefore, we evaluate our best parameters on the validation set, then our best model is tested directly on the test set. It was necessary to be very careful so as not to eliminate entire classes from the sets, therefore half samples of each class were divided by train set and the other half in the validation set. These two sets are further divided into batches to avoid memory failures. This leads to longer computation times and it is less effective, but it is still strongly recommended especially for larger networks such as ResNet, VGG... Therefore we choose a batch of 256 images, which is a commonly used value. The result is shown in Figure 2.

Figure 2: Portion of a batch of images

Starting from here: let's try to use the aforementioned hyperparameters to have the first baseline, but the result obtained is a little bit poor. As shown in Figure 3, the loss function (the green one) starts from a value greater than 4.5 and ends approximately at 4.4. So the loss does not decrease quickly at all. This also implies that accuracy doesn't improve much during the 30 epochs. This result is confirmed in the accuracy chart, in which the orange line never exceeds the threshold of 0.1. This could mean that the network is fixed at a local minimum and the learning rate is too low to jump from that point. So, to improve the result, let's try to increase the learning rate. The question to ask, therefore, becomes: how should this hyperparameter be regulated? If we increase it too much, in fact, we may be faced with two different problems: either the accuracy becomes less than optimal, or the model diverges. Since

with LR = 1 the net diverges immediately, I tried with LR = 0.1. The result is not that bad. The loss (the blue one) decreases very quickly in the first 8 epochs, after which it starts to remain stable until the LR drops to 0.01 after 20 epochs. The final score obtained is quite good, considering that the net has not been trained, and in any case that the training phase lasts only 10 minutes; nevertheless, the accuracy reaches 0.1848. It is clear that the learning rate is a bit high because the accuracy and the loss fluctuated markedly upwards and downwards. So, the last attempt is to try a learning rate of 0.01. The result is quite remarkable. The loss decreases constantly during epochs and after the five initial steps, the accuracy score improves in any epoch. The final accuracy in the validation set is 0.3021 and a similar result is obtained in the test set where the accuracy is around 0.29. In both charts x axis represents the number of epochs and the y axis describes respectively accuracy and loss values (the same for all the graphs in this homework).

LR	N. Epochs	Step Size	Gamma	Optimizer	Scheduler	Acc.	Loss	Color
10^{-1}	30	20	0.1	SGD	StepLR	0.1848	4.11	Blue
10^{-2}	30	20	0.1	SGD	StepLR	0.3021	3.22	Pink
10^{-3}	30	20	0.1	SGD	StepLR	0.09	4.47	Green

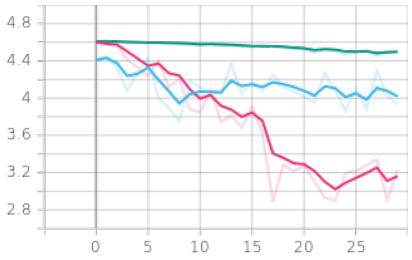

Figure 3: Loss/train alexnet

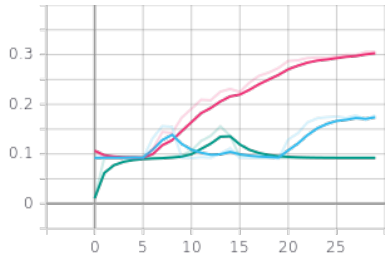

Figure 4: Accuracy/train alexnet

To try to improve the result once again, let's try to change the optimizer. This time it's up to the "torch.optim.ASGD" which implements the Average Stochastic Gradient Descent. This is a new approximate stochastic recursive algorithm based on the average of the trajectories. Using this optimizer, CNNs converge rather slowly in the first steps and both the accuracy and the loss functions have a worse behaviour than before.

LR	N. Epochs	Step Size	Gamma	Optimizer	Scheduler	Acc.	Loss	Color
10^{-1}	30	20	0.1	ASGD	StepLR	0.09	4.28	Orange
10^{-2}	30	20	0.1	ASGD	StepLR	0.2221	3.77	Blue
10^{-3}	30	20	0.1	ASGD	StepLR	0.03	4.5	Light-blue

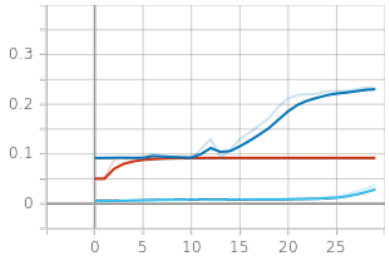

Figure 5: Accuracy/train alexnet

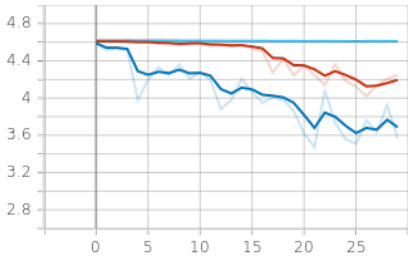

Figure 6: Loss/train alexnet

Choosing 1e-2 (blue line) as LR value leads once again to the best performance. In fact, the accuracy is above 0.2 and the loss decreases faster than the other cases. Instead, the result obtained with 1e-3 (light blue line) and 1e-1(orange line) is really poor. Both Cnns seems not to learn any meaningful features. In any case, to have a better understanding of the result, we trace the histogram related to the first convolutional layer of the Cnn during the epochs, both as regards the term "bias", the term "weight", and for the term "grad". Even though the gradient of AlexNet with ASDG (those in blue and red in Figures 7 and 8) has values similar to that with SDG (light blue), the terms of bias and weight do not change a lot during the different epochs. Furthermore, both terms have lower values than the one obtained with SDG. By virtue of this, the result shows that from now on it is better to keep the SDG optimizer.

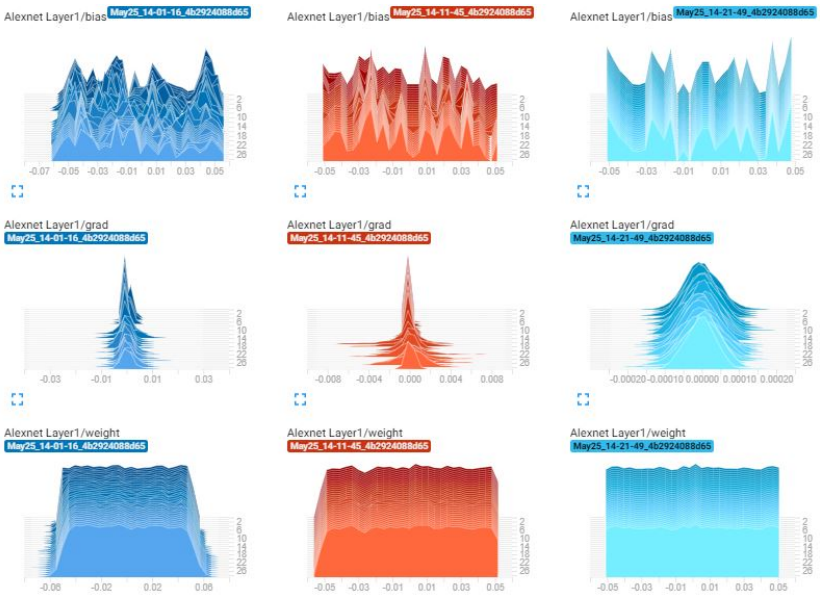

Figure 7: Comparison overlay histograms

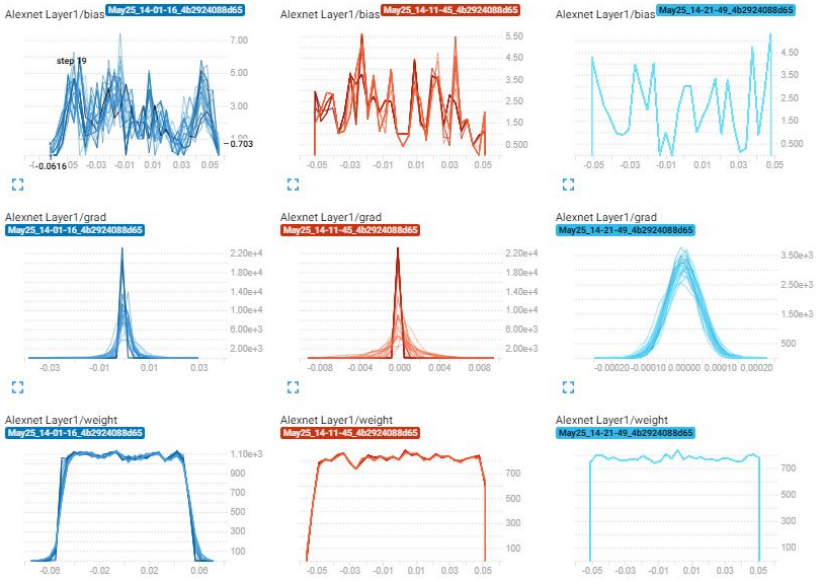

Figure 8: Comparison overlay histograms

It is very interesting to notice that the weight values do not have a Gaussian distribution. It may be a consequence of an insufficient training of the network. The gradient histogram of the SDG, instead, has a normal distribution: this could mean that the network has a good functioning and it is learning in a correct way. The value of the gradient also decreases during the various steps: possible indication of a slow convergence of the network.

4 Transfer Learning

To increase performance and reduce calculation times, it is possible to upload a pre-trained version of AlexNet on imagenet ds. This is a huge ds used in competitions with over 1000 classes. The weights of this network are better initialized than choosing values from a normal or an uniform distribution. Before starting to train the network, it is necessary to modify the Normalize function of the data preprocessing, in order to normalize it using the ImageNet mean and standard deviation. The new transformations are reported in the following code:

```
# Define transforms for training phase
train_transform = transforms.Compose([transforms.Resize(256),
transforms.CenterCrop(224),
transforms.ToTensor(),
transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
])
```

```
# Define transforms for the evaluation phase
eval_transform = transforms.Compose([transforms.Resize(256),
                                     transforms.CenterCrop(224),
                                     transforms.ToTensor(),
transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
])
```

Starting from the previous best parameter, we train the network again and the result this time is surprising. The loss (grey line) decreases immediately and the accuracy score starts at 0.46, which is much better than the old net after 30 epochs. So, the "learning transfer" this time dramatically increases CNN's performance. Let's try to move the training to other hyperparameters, and in particular:

1. LR = 0.001, Epochs = 40. When we use transfer learning, we do not want to change the weights of the network too much, therefore we decrease the learning rate but compensate to increase the number of epochs (orange line in Figure 9 and 10).
2. Since there is no significant improvement in the result, let's try back to the 30 epochs but change the "weight reduction" policy to 1/5, in an attempt not to slow down the network after 20 epochs (blue line in Figure 9 and 10).

LR	N. Epochs	Step Size	Gamma	Optimizer	Scheduler	Acc.	Loss	Loss Type	Color
10^{-2}	30	20	0.1	SGD	StepLR	0.8301	≈ 0.002	CrossEntropy	Grey
10^{-3}	40	20	0.1	SGD	StepLR	0.8261	≈ 0.007	CrossEntropy	Orange
10^{-2}	30	20	0.2	SGD	StepLR	0.8291	≈ 0.003	CrossEntropy	Blue

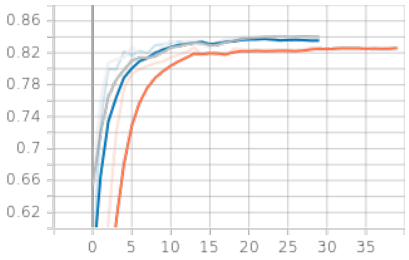

Figure 9: Accuracy/train trained alexnet

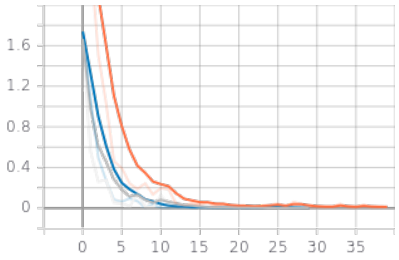

Figure 10: Loss/train trained alexnet

The three networks behave similarly. The loss and accuracy of CNN with a lower learning rate deteriorate slightly, but the end result, overall, is practically the same. An accuracy greater than 0.8 is an excellent result if we consider the previous networks. Furthermore, the loss for the last epochs is less than 0.01, a

truly impressive result. It is very interesting to underline that the weights and the bias in the first convolutional layers practically remain unchanged. This is clearly shown in histograms 11 and 12. This behavior is completely different from the net trained from scratch in which the values of weights change significantly during the epochs.

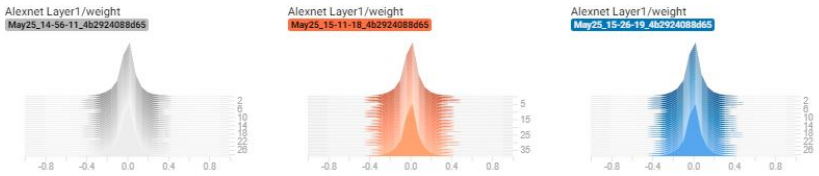

Figure 11: Comparison offset histograms

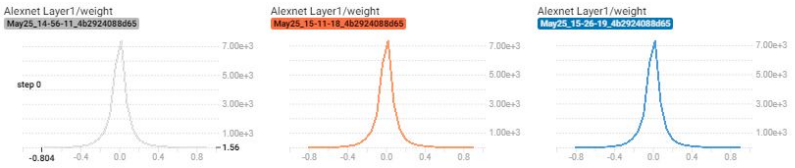

Figure 12: Comparison overlay histograms

We have also tried to reduce the number of epochs and the step-size and, at the same time, increase the value of gamma. This time the loss type is MultiMarginLoss and the optimizer is ASDG. The result obtained is poorer than before. This means that the gamma value it's not so relevant for the final result and the network should run at least 30 epochs because it's still learning something. Said that, the loss is still very low when we use 0.01 as learning rate. Both the ASDG and the MultiMarginLoss doesn't work so well so we come back to the SDG and CrossEntropy.

LR	N. Epochs	Step Size	Gamma	Optimizer	Scheduler	Acc.	Loss	Loss type	Color
10^{-2}	20	10	0.1	SGD	StepLR	0.8078	≈ 0.002	MultiMargin	Blue
10^{-3}	20	10	0.5	SGD	StepLR	0.7037	≈ 0.01	MultiMargin	Red
10^{-2}	20	10	0.5	SGD	StepLR	0.8076	≈ 0.001	MultiMargin	Light-Blue

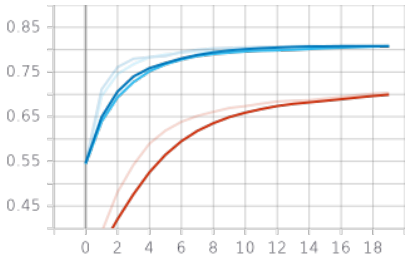

Figure 13: Accuracy/train trained alexnet

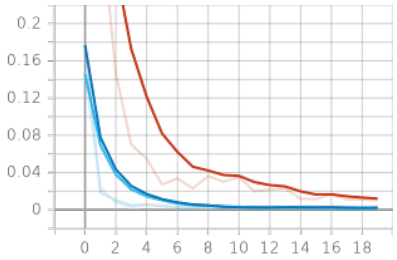

Figure 14: Loss/train trained alexnet

When we use transfer learning, we can also "freeze" part of the network and train only certain levels to speed up the train phase. First of all, we start to freeze the convolutional layers, which represent the largest part of the network. As illustrated by the graph below (the blue line in this case) the situation is very similar to the previous one. This could mean that the Caltech classes are present in the Imagenet ds, so the convolutional layers are already well trained. To have a better understanding about a possible correspondence between what we have seen and the factual truth, we try to freeze only the fully connected levels. The result obtained may suggest that what we have seen previously is not exactly correct. This result suggests that the network can still learn important features to improve the final result. At the beginning, both loss and accuracy are a bit worse than before, but after 10 epochs the situation is exactly the opposite. The final result is quite impressive, with an accuracy of 0.84 both on the validation and test set. Since the result is a consequence of the data shuffle, it may also simply depend on the "luck" we had in the shuffle phase.

LR	N. Epochs	Step Size	Gamma	Optimizer	Scheduler	Acc.	Loss	Color	Freeze Layer
10^{-2}	30	20	0.1	SGD	StepLR	0.8402	≈ 0.003	Blue	Convolutional
10^{-2}	30	20	0.1	SGD	StepLR	0.8355	≈ 0.002	Pink	Fully connected

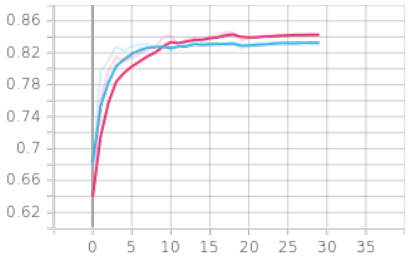

Figure 15: Accuracy/train freeze convolutional or fully connected layers layers

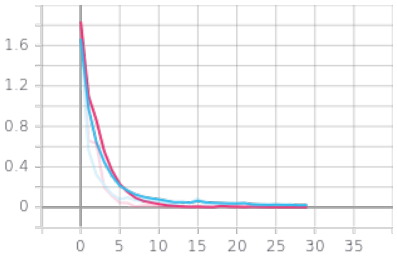

Figure 16: Loss/train freeze convolutional or fully connected layers

5 Data Augmentation

In order to make the network able to operate a better generalization, a commonly used approach is represented by Data Augmentation. In this case, CNN should be able to correctly classify an image within a class, regardless of its rotation, point of view, size or color of the image itself. But often our DS is too small to represent all the possible facets of a class. So, in order to obtain "more data", you could think of proceeding by taking all the images you have available, and inserting some noise in them (for example: turn the image upside down, resize the image, etc ...) to prevent the network from learning irrelevant pattern and improving its overall performance. The most commonly used transformations are represented by random horizontal / vertical flip and color jitter. With the first, we try to rotate the image with respect to its horizontal or vertical axis with a probability of 0.5. With the second, however, we change the brightness, contrast, hue and saturation of the image. The final goal is to try to understand if a vertical rotation is better than a horizontal one, using the same color jittering.

LR	N. Epochs	Step Size	Gamma	Optimizer	Scheduler	Acc.	Loss	Color	Effects
10^{-2}	30	20	0.1	SGD	StepLR	0.8519	≈ 0.006	Light-blue	Horizontal flip + jittering
10^{-2}	30	20	0.1	SGD	StepLR	0.8179	≈ 0.008	Pink	Vertical flip + jittering

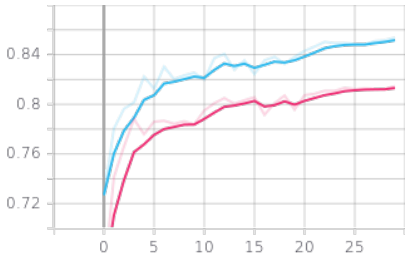

Figure 17: Accuracy/train with alexnet and data augmentation

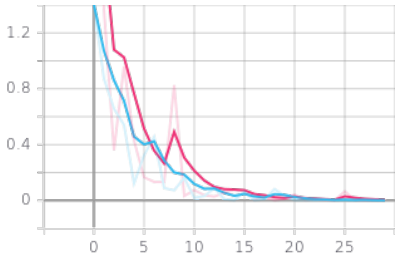

Figure 18: Loss/train with alexnet and data augmentation

As shown in the graphs above, the horizontal rotation (in light blue in Figures 15 and 16) helps the net to reduce the loss and to increase the accuracy in a relevant way. The network could therefore have a better understanding of what the important characteristics of each class are, regardless of where they are in the photo. It is important to underline that the final score achieved is better than the one obtained without data augmentation. A different result, however, is obtained with the vertical flip: CNN does not improve at all. After all, it is not precisely what is logical to expect: a horizontal rotation of the image changes it less than a vertical one. For example, if we rotate a human face around the vertical axis, the result is roughly the same as the initial image. Vice versa, by making a vertical rotation, we get an upside down face and this may suggest to

the net that a face is characterized by some particular characteristics and not by the positions it has in the photo. But this not happen. This can depend by the fact that in the validation set we don't have upside-down photos. So, even if the image changes less with a horizontal flip it is disturbed enough to let the net learn important features.

After that, we also tried to change the optimizer again, using ASDG, to check if something different happened, but the result turned out to be the same. This result emphasizes that horizontal rotation (blue in the graphs) and color jittering help CNN to generalize better.

LR	N. Epochs	Step Size	Gamma	Optimizer	Scheduler	Acc.	Loss	Color	Effects
10^{-2}	30	20	0.1	ASGD	StepLR	0.8345	≈ 0.01	Blue	Horizontal flip + jittering
10^{-2}	30	20	0.1	ASGD	StepLR	0.8067	≈ 0.02	Orange	Vertical flip + jittering

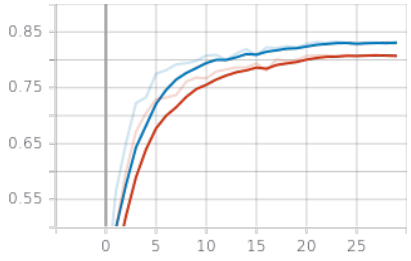

Figure 19: Accuracy/train with alexnet(ASDG) and data augmentation

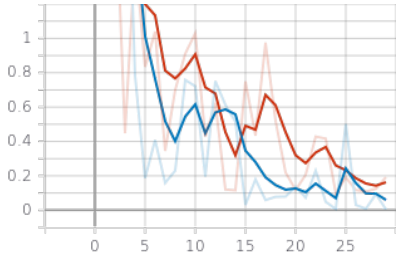

Figure 20: Loss/train with alexnet(ASDG) and data augmentation

Using the ASDG optimizer let's try to change the "center crop" with a random one. During all these trials we always use the centre part of the image because it's common that the subject we are interested is located in the middle of the photo. It's not always true. To consider all the details of each photo we

choose every time a different part of the image. So, we apply Horizontal and Vertical flip to check which one performs better.

LR	N. Epochs	Step Size	Gamma	Optimizer	Scheduler	Acc.	Loss	Color	Effects
10^{-2}	30	20	0.1	ASGD	StepLR	0.7818	≈ 0.003	Orange	RandomCrop + Horizontal flip
10^{-2}	30	20	0.1	ASGD	StepLR	0.7593	≈ 0.005	Gray	RandomCrop + Vertical flip

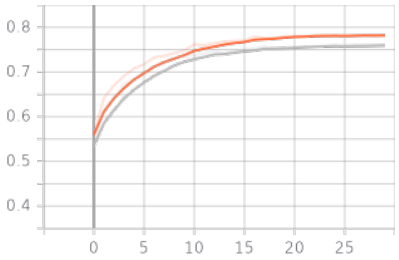

Figure 21: Accuracy/train with alexnet and data augmentation

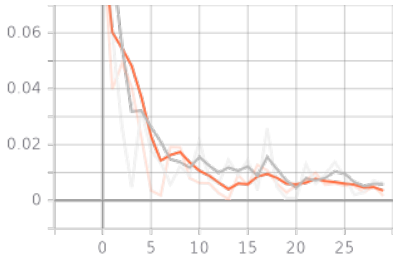

Figure 22: Loss/train with alexnet and data augmentation

The final result confirms what seen until now. The only thing to underline is that the accuracy never decreases. This means that a different crop every time allows the net to discriminate better at every iteration. Finally, removing the colour jittering slow down the perfomances in a drastic way.

6 Beyond AlexNet

Alexnet is a very simple and old fashioned model. Over the years, much more complex and specific CNNs have emerged. One of the most recent and well-made network is ResNet. Its working principle is based on skip connections between the layers, which allows to prevent "vanishing gradients", which is a big problem when we have multiple convolutional layers. The Pytorch Library implements 5 different types of this network: from a Resnet with 18 levels to one with 152. As you can imagine, a greater number of layers implies greater accuracy, but this also involves a greater computational and memory effort.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 23: PyTorch ResNets structures

Since Colab, as previously said, has a limited amount of memory, only the smallest ones are considered: ResNet18 and ResNet34. In both cases, using a batch of 256 images you get a memory error, therefore we reduce it to 32. Since we reduce batch-size, we also reduce the learning and, after some tests, we decide to use LR = 0.005. The result is fantastic, much better than the one obtained before. ResNet18 (orange lines) works properly, in fact the final accuracy exceeds 0.92 and the loss, especially in the last steps, is very similar to the one obtained with Resnet34. The latter reaches a score above 0.947, an excellent result. During these tests with ResNet, we also used data augmentation, but we believe that perhaps with complex structures and with more tuning of the hyper-parametrs it is possible to obtain an accuracy very near to 1.0. In fact our result in the test set is 0.941 with the ResNet34 and 0.912 with the ResNet18.

ResNet type	LR	N. Epochs	Step Size	Gamma	Optimizer	Scheduler	Acc.	Loss	Color
34	$5 * 10^{-3}$	30	20	0.1	SGD	StepLR	0.947	≈ 0.001	Grey
18	$5 * 10^{-3}$	30	20	0.1	SGD	StepLR	0.921	≈ 0.004	Orange

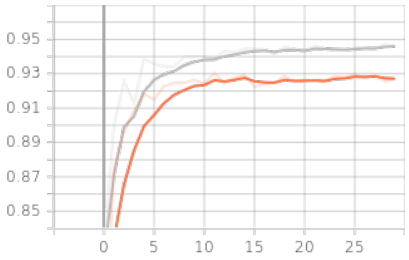

Figure 24: Accuracy/train ResNet

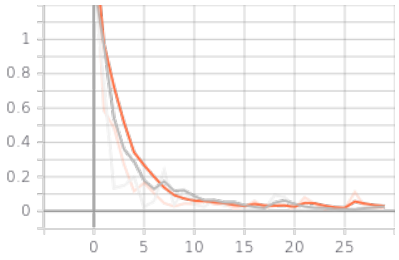

Figure 25: Loss/train ResNet

Finally, a further test was carried out, and it was significant. In fact, we compared a non-pre-trained ResNet with an AlexNet. To understand if every-

thing went well we chose "torch.nn.MultiMarginLoss" which is very similar to the loss of the hinge, as can be seen from the formula:

$$loss(x, y) = \frac{\sum_i (max(0, margin - x[y] + x[i]))}{x.size(0)}$$

When the network is not initialized, $x[y]$ and $x[i]$ are very similar, so the loss for a specific class will be approximately 1. The total initial loss will therefore be approximately $101/100 \cong 0.99$. As illustrated by the graph, this approximation is quite correct: after some steps, ResNet34 is even better than AlexNet, therefore the result obtained previously is confirmed even with non-pre-trained networks.

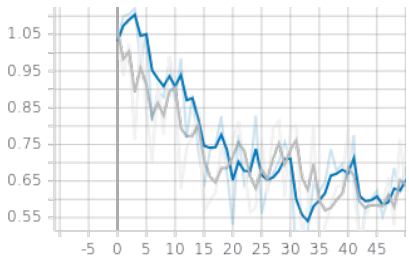

Figure 26: Loss/train of ResNet34 and AlexNet

Finally, the histogram of the weights of the first convolutional layer of AlexNet and ResNet is plotted. Interestingly, the former has a uniform distribution and the later has a normal one.

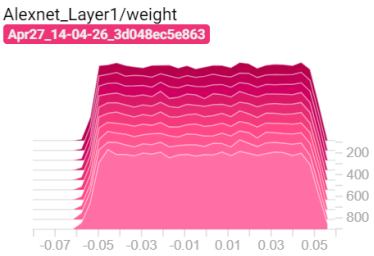

Figure 27: AlexNet's weights of the first convolutional layer

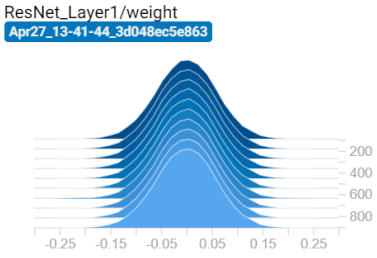

Figure 28: ResNet's weights of the first convolutional layer