Homework 2: Tranfser Learning

Lucia Innocenti

April 2020

Contents

1 Introduction

This project is about training a neural network by using the dataset Caltech101. The neural network used for the first part of the project is AlexNet. In order to manage the network and the dataset, the first step is to define methods and attributes to access the dataset itself. The model of Caltech101 contains the following methods:

- **Class Constructor:** it creates the main class; the attributes assigned in the method are:
  - root
  - split
  - transform
  - classes
  - items
  - items_as_string
- **_find_classes:** the method returns the sorted list of classes available into the dataset;
- **__getitem__:** given an index, it returns the correspondent element at index position in the dataset, both image and label;
- **__getSubsets__:** it returns two lists of indexes, each of them containing a subset of elements from each class; the dimension of the subset is defined by the parameter "percentage" takes as input;
- **__len__:** it returns the dimension of the items list

So, after having defined the core class, in the Colab Notebook I have work on the structure provided for the project that is made as follow:

- **Set Arguments:** in this section, all the hyper-parameters are defined and initialized;
- **Define Data Preprocessing:** in this section are defined transformers for the dataset; they will be used to adapt images to the network and also to do data augmentation;
- **Prepare Dataset:** the class constructor for the Caltech101 are called here; from the original splits available in the dataset, I have retrived the train and the test set; by the getSubsets method I have also split the train set in two parts: the validation set and the train set;
- **Prepare Dataloaders:** a dataloader makes a dataset iterable;

- **Prepare Network:** the network choose for this project is alexNet; in a first phase, the network is trained by scratch and, in a second moment, I will use a pretrained version, so in this section I also choose which layers I want to freeze or to unfreeze;
- **Prepare Training:** in order to evaluate the performances, the project need a loss estimator, an optimizer and a scheduler;
- **Train - Validation - Test:** the images of the data set are passed over the network to train it and subsequently to evaluate its performance

1.1 Evaluation methods

Being the goal of this project to improve the performances of the network, it is necessary to define some evaluation methods. The network need a loss function in order to compute the gradient and improve itself, so we can use it also to evaluate if the network is well-doing and, especially, if it's overfitting results. Pytorch provides different loss-function; the chosen one is the cross-entropy loss, useful when training a classification problem with N classes. Given as input a matrix $N * C$, where C is the batch size, each of the N rows has, in each of the C elements, the probability that the element belong to the correspondent class. The target array contains the correct class for all the C elements, so the Cross Entropy loss is obtained by computing:

$$-\log \frac{exp(x[class])}{\sum_{j=1}^{N} exp(x[j])} \tag{1}$$

Another important approach is to evaluate the accuracy, computed as:

$$\frac{Correctly\_predicted\_elements}{Total\_number\_of\_elements} \tag{2}$$

Results on the test set are reported in the Table 2

2 Learning from scratch

The goal of a good Network is to find the best hyper-parameters. So, in order to do this, I've try some approaches.

The parameters in which I have decided to focus my analysis are the learning rate and the number of epochs. In fact, given a perfectly estimated learning rate, the network will learn the best function given available resources. By reading different papers, I've notice that common values for LR are the ones belonging to $[10^6, 1]$.

In the provided template, starting values were:

- $LR = 1e - 3$
- $NUM\_EPOCHS = 30$

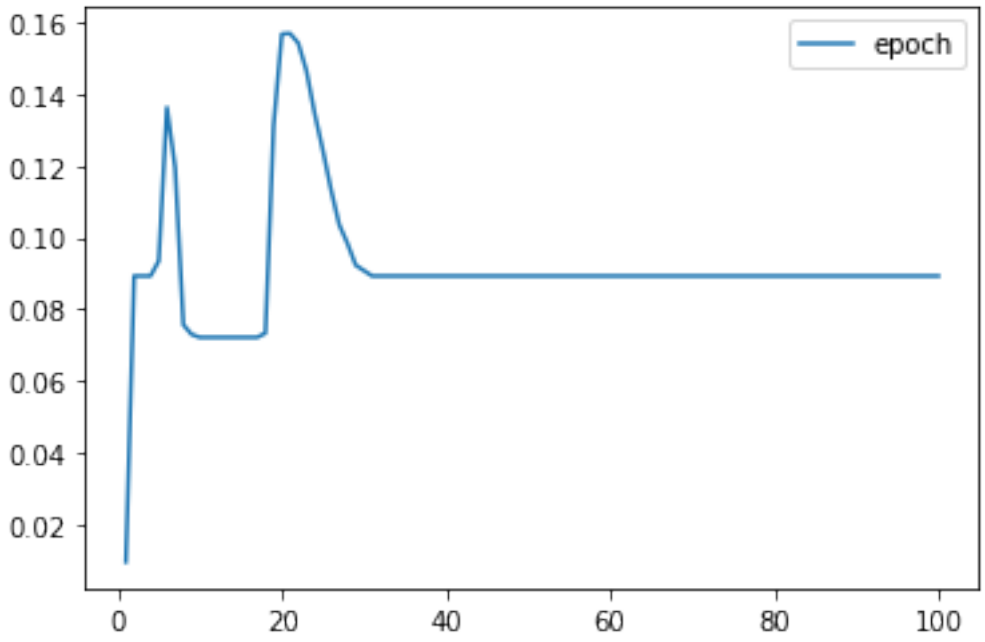

Figure 1: Accuracy - Epoch

We cannot analytically calculate the optimal LR for a given model on a given dataset. Instead, a "good enough" value can be find by trial and error. Some relations have to be considered when tuning the learning rate:

- Smaller LR -> More Epochs
- Smaller Batch Size -> Smaller LR

In order to tune the hyper-parameters, I've used diagnostic plot: to investigate how the LR impacts the learning phase, I've create a line plot of loss and accuracy over epochs during training and a line plot of LR and accuracy during training.

I've set the number of epochs = 100. Figures 1 and 2 shows how the accuracy and the loss fluctuate over epoch. From this, I have see that at $epochs = 70$ I have a minimum for the loss and a maximum for the accuracy.

Then, I've plotted how the accuracy changed with the learning rate. This plot is available at figure 3 and we can see that the best value of accuracy is for a learning rate = 0.001.

So, I've tried to decrease the number of epochs and increase the learning rate. The best combination found, after a few attempts, was

- $LR = 0.01$
- $NUM\_EPOCHS = 70$

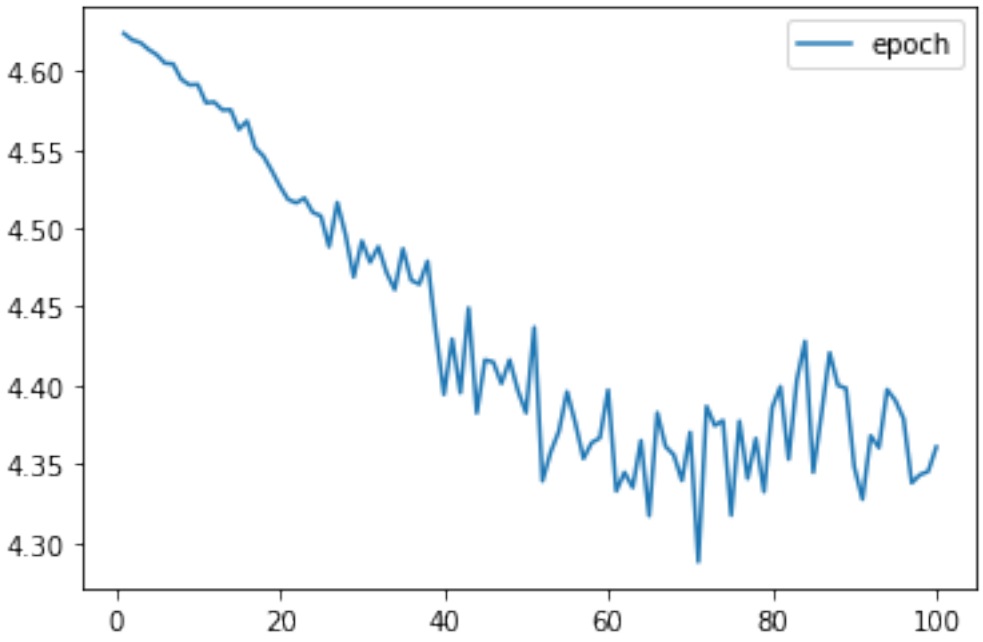

Figure 2: Loss - Epoch

If we take a look to the plot in figure 4 we can see that since the number of epochs is equal to 40, there is not an increase of performances. So, to optimize the algorithm execution and avoid overfitting, I've scaled the number of epochs from 70 to 40.

By looking at this plots we can notice that, even when the hyper-parameter change, the starting point for both accuracy and loss is not so good; for the network is very challenge to reach good performances: the best reached accuracy is around 0.5, not really a good result.

3 Transfer Learning

Also after have tuned the hyper-parameters, the performances in the test test are not so good. An interesting approach might be to use a pretrained network: instead of starting the learning process from scratch, it's possible to start from patterns that have been learned when solving a different problem. The network are loaded with a pretrained version and the last fully connected layer is changed in order to adapt the last classification at the needed task.

In this step, the hyper-parameters are set at the best values detected by learning from scratch.

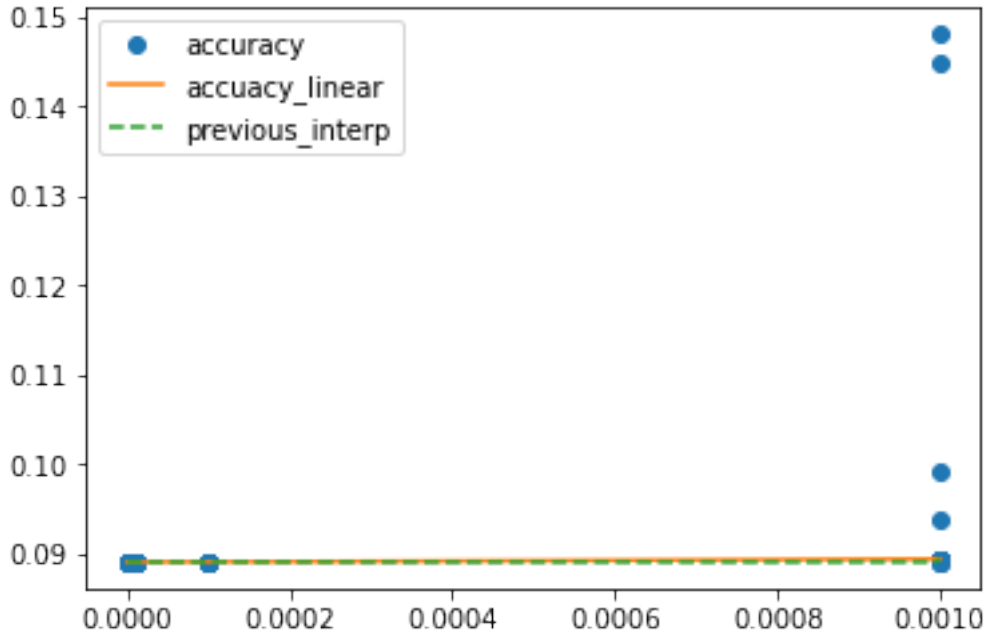

Figure 3: Accuracy - Learning Rate

The results obtained are reported in figure5, where we can see that we can notice that:

- The accuracy starts, for epoch = 0, from more or less 0.675 and reach a maximum of 0.85
- The loss has significantly decreased

Looking at the graphs, another thing to highlight is that the accuracy increase until the epoch 20 and that, from that moment, the loss start to increase. It could suggest an overfitting. So I've tried two different strategies to improve performances:

- Early Stopping: it is a regularization method that consist in setting a large number of epochs and stop training at the point when performance on a validation dataset starts to degrade. In this case I've changed the optimizer removing the weight decay, which are another regularization method which would gone against early stopping. I've implemented it by keeping at each epoch the loss value and, if that value increase for at least three times in a row, the network stop the train phase. By applying this method the accuracy in the training set increases by a few points
- Adam Optimizer: it is an improved version of the Stochastic gradient descent optimizer; but, in this network, it don't provide improvements.

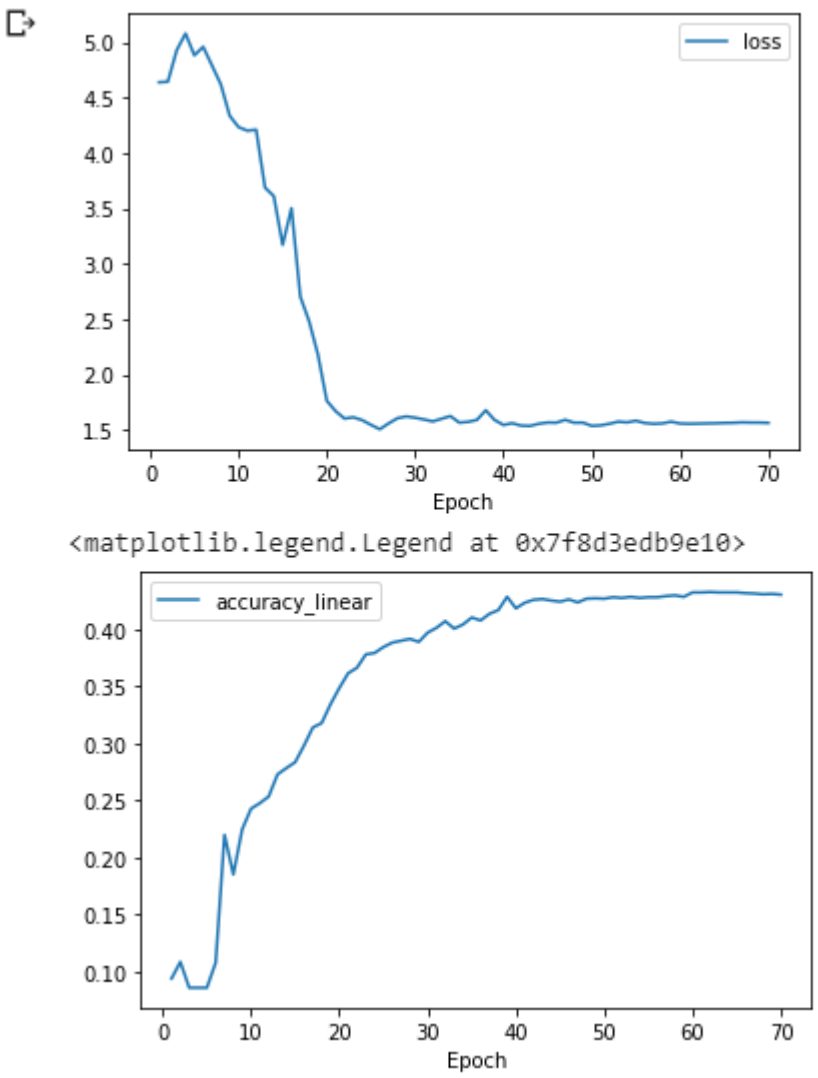

Figure 4: Epochs - Accuracy - Loss

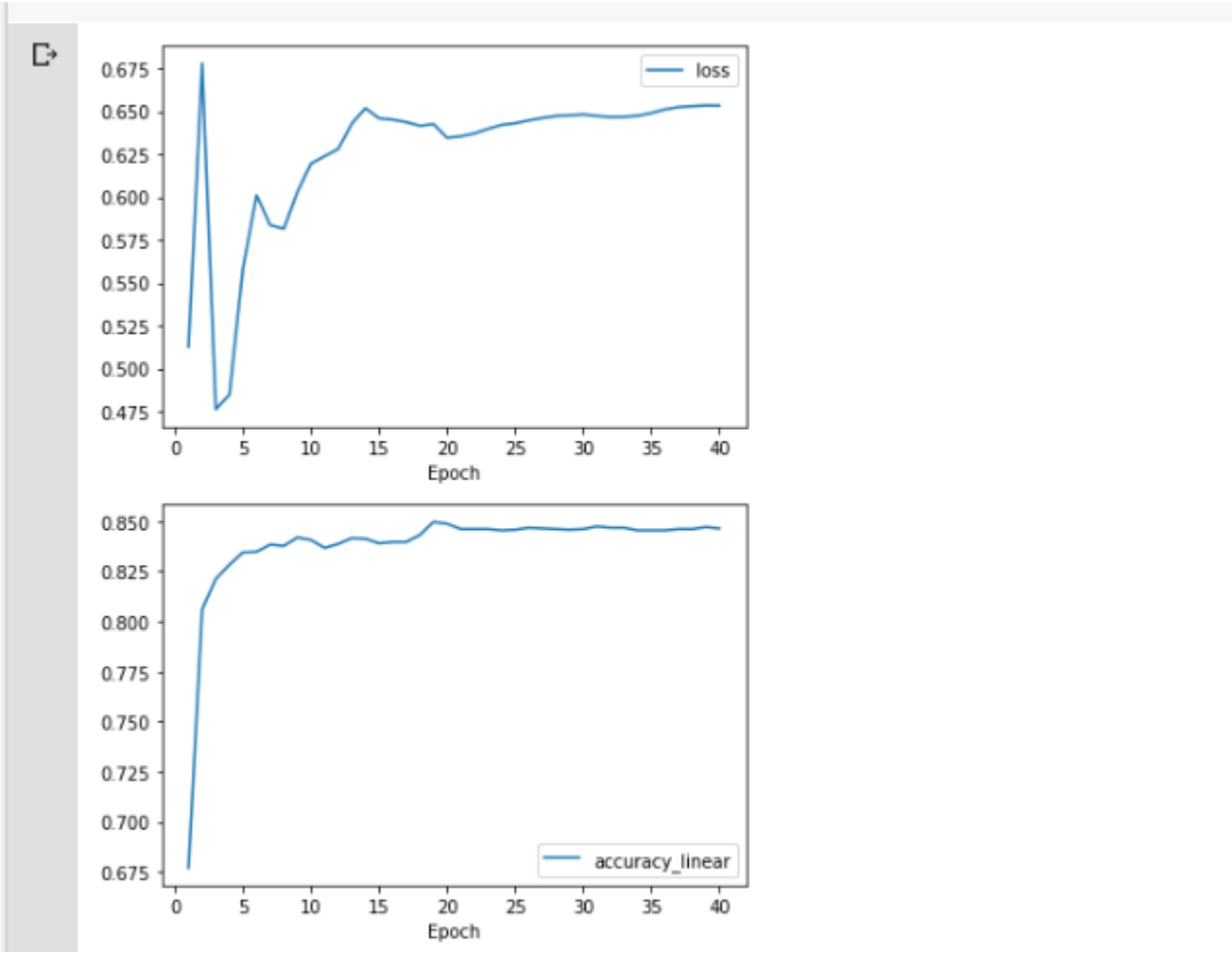

Figure 5: Loss - Epoch

3.1 Partially freeze layers

When a pretrained model is loaded, it's possible to re-train a subset of the network in order to better adapt it to the task in exam.

So, in this step I've first unfreeze only the fully connected layers. As expected,the performances are slightly increased. The fully connected layers are the last in the network, so that layers that learns the features more adapted to the dataset. For that reason, the second step (freeze the fully connected layers) was not so good: in this case the performances decrease with respect to the version with all the layers freeze but the last one.

4 Data augmentation

We can use image augmentation to increase the number of images used to train the network. In order to define which methods apply, I've take a look to the following aspects in the classes:

- **Colors**: some classes, like Dalmatian, are strictly defined by their colors, so I've decided to take particular attention in transform colors.
- **Flips**: in this case, horizontal flip can be a good idea, and also rotate images

After this analysis, I've choose the following methods:

- **ColorJitter**:Randomly change the brightness, contrast and saturation of an image.
- **RandomRotation**: Rotate the image by angle from 0 to 30 degrees.
- **RandomOrizontalFlip**: Horizontally flip the given PIL Image randomly with a probability 0.5.

Results are in the Table 2. All the methods for data augmentation have been applied to the Pretrained Network with unfreeze fully connected layers.

5 Results

Learning from scratch	
LR , Num Epochs	*Accuracy*
0.01, 40	0.389
0.01, 70	0.408
Pretrained Network - train last level	
Method	*Accuracy*
Weight decay	0.838
Early Stopping	0.842
Optim Adam	0.815
Pretrained Network - train FC	
Method	*Accuracy*
Weight decay	0.852
Early Stopping	0.841
Pretrained Network - train notFC	
Method	*Accuracy*
Weight decay	0.804
Early Stopping	0.216
Pretrained Network - train FC - Data Augmentation	
Method	*Accuracy*
RandomRotation	0.829
ColorJitter	0.853
RandomHorizontalFlip	0.848

Table 2: My table