Introduction

In this homework we are going to train a CNN for image classification, we're using the pyTorch implementation of AlexNet, which is based on this paper[AlexNet].

The dateset on which the CNN will be trained is Caltech-101, which is made of 101+1 classes and a little more than 8500 images of roughly 300 x 200 pixels; each class has between 20 and 800 images, this has to be taken into consideration when training and testing.

First Step

We've been provided with a template code, for training and testing, and with a dataset class, with missing methods, to complete.

Training with this template code takes only a few minutes and it leads to an accuracy of 38.2% on the test set after approximately 20 epochs, than it starts to overfit.
Image1 represents loss and accuracy for each epoch.

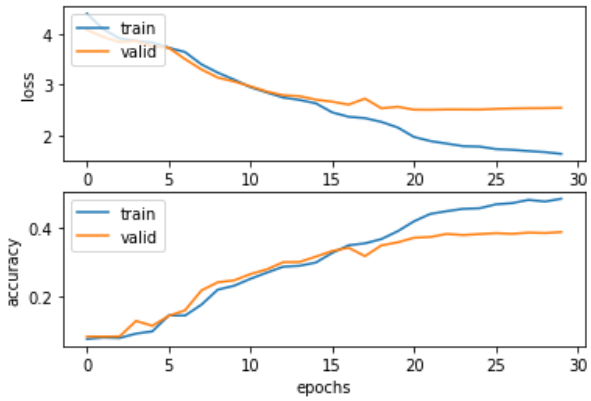

Figure 1: Train and valid loss and accuracy for each epoch.

Training from scratch

Validation and train set

In order to tune the hyper parameters and select a model, one needs to divide the training data in train and validation set.
K-fold cross validation, in fact, would be prohibitive due to the high number of iterations

required to perform the training.
I'm going to split the training set in half, trying not to remove entire classes from the training or validation set; the code for this operations can be found in the sections "Prepare the Dataset" and "Define the dataset class" of the notebook.

Hyper-parameters tuning and model selection

In order to choose the hyper-parameters of my model I need to evaluate it on the validation set after each training epoch.
In the next three sections I'm going to describe the three different sets of hyper-parameters that I decided to tune.

First set

Firstly, I decided to use the Adam optimizer and to tune learning rate and batch size. The update rule of the Adam optimizer is:

$$\begin{aligned}
\rho_1 &= \beta_1 * \rho_1 * (1 - \beta_1) * \Delta_{\theta_t} \\
\rho_2 &= \beta_2 * \rho_2 * (1 - \beta_2) * \Delta_{\theta_t} * \Delta_{\theta_t} \\
b_1 &= \frac{\rho 1}{(1 - \beta_1^t)} \\
b_2 &= \frac{\rho 2}{(1 - \beta_2^t)} \\
\theta_{t+1} &= \theta_t - \frac{\epsilon * b_1}{\sqrt{b_2} + 1e - 7}
\end{aligned}$$

I started by taking the learning rate from a uniform distribution U(10^{-3}, 10^{-5}) and by considering the batch size as a random variable with possible values 64, 128, 256 and 512.
At each iteration I draw a value for both learning and batch size and than trained the model for 15 epochs, after this initial phase I found that the best values for the learning rate lied between 10^{-4} and 10^{-5} and that those for the batch size were 64 or 128. Image2.
I than performed another search, this time with an uniform distribution U(10^{-4}, 10^{-5}) for the learning rate and by considering 64 and 128 as possible values for the batch size.

The best values found are 1e-4 for the learning rate and 64 for the batch size.

Second set

Having decided which values to use for learning rate and batch size, I then performed coarse and finer search for $beta_1$ and $beta_2$, the initial values were drawn from a U(0.8, 0.999) distribution.

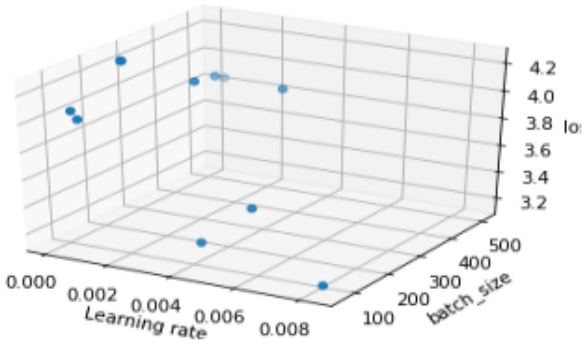

Figure 2: Best values for learning rate and batch size after first phase of hyperparameters tuning.

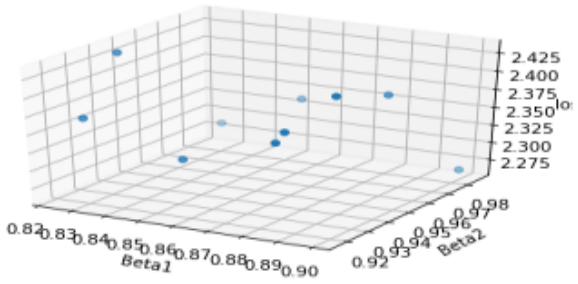

Figure 3: Best values for $beta_1$ and $beta_2$ after second phase of hyperparameters tuning.

Later, for the finer search the values for $beta_1$ were drawn from U(0.82, 0.90) and those for $beta_2$ from U(0.90, 0.99). Image3.

The best values found were 0.90 for $beta_1$ and 0.94 for $beta_2$, with them the model achieves an accuracy, on the validation set, of 51.4% after 7 epochs, than it starts to overfit the data. Image4 shows the loss and the accuracy for each epoch.

Third set

Lastly, I used SGD + Nesterov Momentum as optimizer and tuned again learning rate and batch size; the update rule for this optimizer is:

$$\begin{aligned} \widetilde{x}_t &= x_t + \rho v_t \\ v_{t+1} &= \rho v_t - \alpha \nabla f(\widetilde{x}_t) \\ \widetilde{x}_{t+1} &= \widetilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t) \end{aligned}$$

Here I started by considering the same values used for the Adam optimizer and then decreased the search space to U(10^{-3}, 10^{-4}) for the learning rate and to 64 or 128 as values for

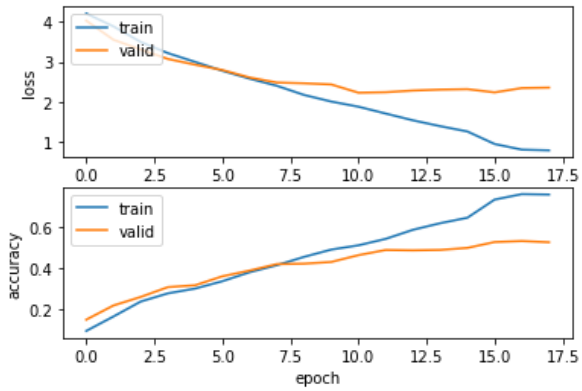

Figure 4: Accuracy and loss for both training and validation set during training. As the visual shows, after approximately 7 epochs the model starts to overfit.

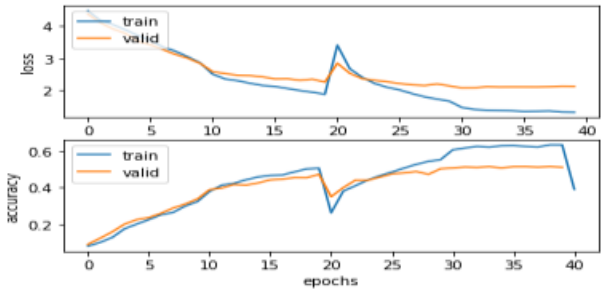

Figure 5: Accuracy and loss for both training and validation set during training. As the visual shows, after approximately 30 epochs the model starts to overfit.

the batch size.

I then tried to run the model with 8e-3 as learning rate and 64 as batch size, but the model started to overfit after approximately 20 epochs.
Using cyclic learning rate I was able to reach an accuracy of about 56% on the validation set in less than 30 epochs, as can be seen in Image5.

Testing

Before testing the best model found I tried to classify images from the validation set using the mean of the results found with the best model with Adam optimizer and the best with sgd optimizer, reaching 77.1% of accuracy.

Testing, in the same way, on the test set leads to an accuracy of 60.5%.

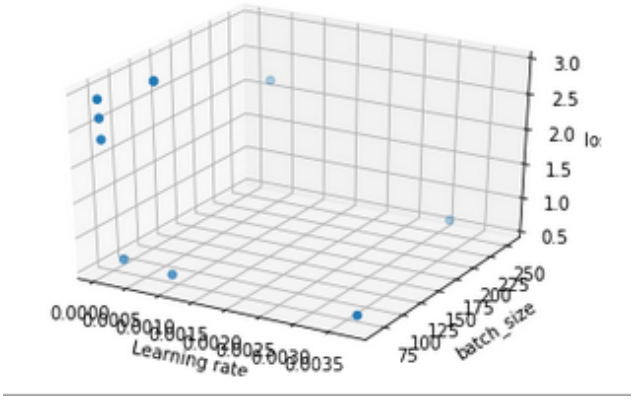

Figure 6: This visual represents the losses for each pair of learning rate and batch size used in the pretrained model.

Transfer Learning

I then used the weights, given by pytorch, of a model trained on ImageNet and performed the same search used above to tune the hyper-parameters.

Optimizers

Firstly, I tried to understand which optimizer could give me the best result, the three possible choices were Adam, SGD + Momentum and SGD + Nesterov Momentum.
Each one of them has been used without optional parameters and with the non-optional hyper-parameters equal to those given at the start of the notebook.

Once finished the search, 15 epochs for each optimizer, I find out that both the SGD optimizers have rached a loss around 0.62 in 7-8 epochs, while Adam started overfitting in just a few epochs.
This led me to use SGD + Nesterov Momentum for the rest of this homework.

Learning rate and batch size

Using that optimizer I then proceeded to tune learning rate and batch size, starting with an uniform distribution U(10^{-2}, 10^{-5}) and from 64, 128 and 256 as possible batch sizes. The results are shown in Figure6.

The finer search was then conducted with U(10^{-3}, $4*10^{-4}$) and between 64 and 256.
The search returned as a result a learning rate equal to 10^{-3} and 64 as batch size.

Gamma

Using the previously found hyper parameters I tried to tune γ, starting from U(0.001, 0.99) and then U(0.04, 0.30).

The best value I find is 0.11, and with this I then tried to trained only part of the model, firstly I considered only the classifier's layers and then only the convolution's ones.

Training only classifier's layers

Initially I trained the model using the best values find for both learning rate and batch size, obtaining an accuracy of approximately 76.5% on the validation set after only 3 epochs.
Successively I tried to reduce both hyper-parameters by a factor of 2, increasing an accuracy of about 82.9% on the validation set after 7 epochs.

Training only convolution's layers

In order to train the model freezing the classifier's layers I found that using a batch size and a learning rate equal to those find previously needed a lots of time, so I tried to increase the batch size to 128 and the learning rate to $10^(-2)$; this, merged with a cyclic learning rate, between $10^(-2)$ and $10^(-3)$, helped me to reach an accuracy of 48.7% on the validation set in 40 epochs.

This accuracy is significantly lower than the one found in the previous section, this may be caused by the fact that we are training only the first half of the model, forcing it to adapt on the weights of the classifier's layers, even though those of the last layer have never been trained.

Test

I chose to test the model with highest accuracy on the validation set, reaching a result of 83.5% on the test set.

Data Augmentation

Often, when the train dataset is not big enough, data augmentation is used to train the model with modified images, while keeping their labels.
I'm going to try three different sets of image transformations using the same set of hyper-parameters find above.

Random rotation and random horizontal flip

While keeping all of the transformation given to us I applied random rotation with probability 20% and random horizontal flip with probability 50%.
This set of transformation led to an accuracy of 76.8% on the validation set.

Random horizontal flip and random gray scale

As with the first set of transformation I applied a random horizontal flip and a random gray scale with probability 10%.
This set led to the best accuracy, for a single model, of 81.8% on the validation set.

Random rotation, random horizontal flip, random gray scale and random perspective

I then applied the same set of transformation used for the first set and added also a random perspective with default values.
This led to an accuracy of 79.2% on the validation set.

Model ensembles

I tried to classify images of the validation set using different sets of models, the results are shown on table1.
As it can be seen the best results are given by using all three models, reaching an accuracy of 85.5% on the validation set

I test this model on the test set reaching an accuracy of 79.7%.

Set	Accuracy
1+2	83.8%
1+3	83.2%
2+3	83.5%
1+2+3	85.5%

Table 1: Accuracy for each set of models.

VGG

I'm going to train three models using pretrained weights of configurations A, E and E with batch normalization of VGG.

I'll use the hyper-parameters found before, except for step and batch size, set respectively to 5 and 32, and the second set of transformation.

The model A reached an accuracy on the validation set of 87.2%, while the deeper model E gained 1.4% more of accuracy on the same set, needing only a few more minutes of training. The best model found is E with batch normalization, even though in the paper it wasn't gaining accuracy with respect to E, while requiring more memory and computation, I decided to try it anyway on this dataset and it reached an accuracy of 89.7%.

Lastly, I tried, as before, with model ensembles, using all the three models, and reached an accuracy of 90.3% on the validation set.
On the test set this led to an accuracy of 91.2%.

With a few more hyper-parameters tuning this models could have the possibilities to reach even higher accuracies, so I tried to decrease the learning rate in order to respect the empirical rule of reducing batch size and learning rate by the same factor, but it led to an accuracy lower than the one find before.

Ultimately I tried with greater batch size and learning rate, but it caused problems due to the amount of gpu's memory available.