POLITECNICO DI TORINO

Master’s degree in Computer Engineering

Artificial Intelligence and Machine Learning

HOMEWORK 2
DEEP LEARNING

Student:
Enrico Loparco, s261072

Professor:
Tatiana Tommasi

ACADEMIC YEAR 2019-2020

Table of Contents

Introduction

This homework aims at getting hands-on experience with neural networks, trying to apply them to the Caltech-101 dataset.
Different techniques and transformations are used throughout the homework, to explore and compare the results obtained.
After the data preparation step, a basic neural network is trained, then transfer learning and data augmentation are applied, followed by an attempt to train a bigger network with a different structure and more layers.

For each approach, results are plotted and discussed, to interpret the changes in accuracy and loss and compare them with the expected behaviours studied during the course.

Dataset

The dataset, collected by the Caltech university, contains pictures of objects belonging to 101 categories, with about 40 to 800 images per category.
In practice, in the material provided there is an additional category ("BACKGROUND_Google"), that is discarded for the purpose of the homework.

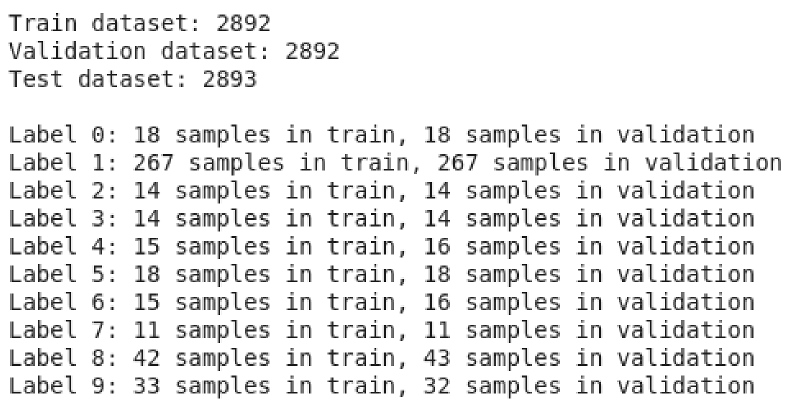
The size of each image is roughly 300 x 200 pixels.
The categories range from airplanes and electric guitars to butterflies and umbrellas.

In the initial data there are also a couple of files ("train.txt" and "test.txt") that contain a list of images belonging to either train or test dataset, to suggest the split to use.

1. Data Preparation

In this initial step, data is loaded from the GitHub repository, divided in train and test set, using the division suggested in "train.txt" and "test.txt" files.
As previously explained, images from the "BACKGROUND" category are not taken into account.
In order to be fed to the network, labels are encoded and saved with the associated images.

An additional split is made, by separating the initial train set in train and validation sets, used to tune hyperparameters.
To achieve that the "train_test_split" function from the scikit-learn library is exploited, allowing to split into two equal-sized sets ("test_size"=0.5).
Stratification provided by that function makes it possible to partition equally the sets, putting for each label half of the samples in the train set and the remaining half in the validation, as reported in the figure below:



```
Train dataset: 2892
Validation dataset: 2892
Test dataset: 2893

Label 0: 18 samples in train, 18 samples in validation
Label 1: 267 samples in train, 267 samples in validation
Label 2: 14 samples in train, 14 samples in validation
Label 3: 14 samples in train, 14 samples in validation
Label 4: 15 samples in train, 16 samples in validation
Label 5: 18 samples in train, 18 samples in validation
Label 6: 15 samples in train, 16 samples in validation
Label 7: 11 samples in train, 11 samples in validation
Label 8: 42 samples in train, 43 samples in validation
Label 9: 33 samples in train, 32 samples in validation
```

Figure 1: Data loading and processing results

This visualization was used to check the correctness of the operation applied. The sizes of the sets are also reported.

The next step consists in defining an ad-hoc PyTorch Dataset class, used as input for the subsequent operations.
For this purpose the template in GitHub repository is filled.
This elegant solution allows also to define custom transformations to be applied when retrieving data.
After that a few images are printed to get a taste of what the input looks like.

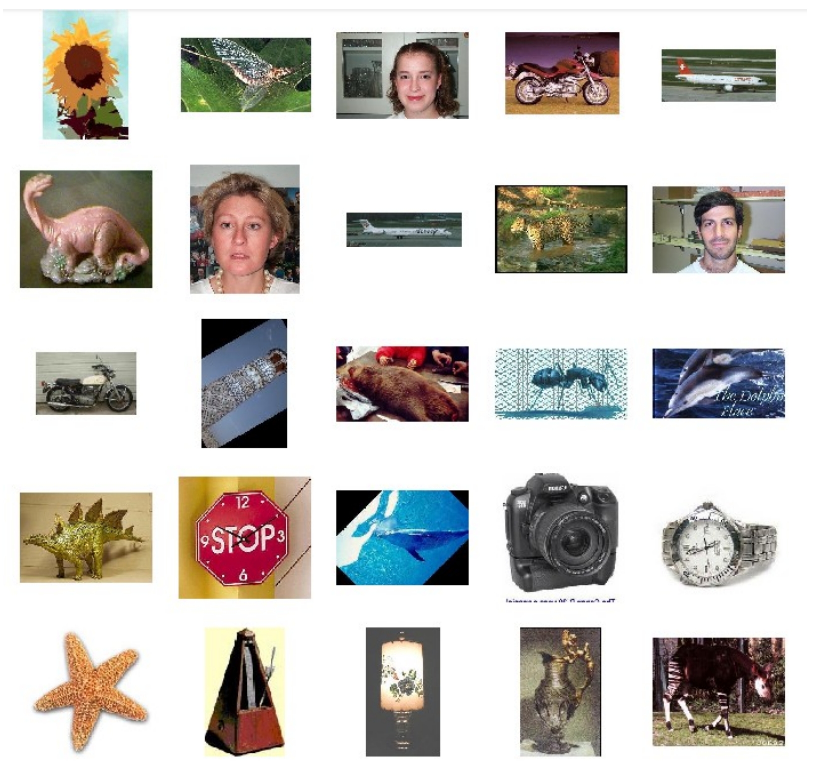


Figure 2: Images from Caltech101

As can be seen, images have different sizes. It will be taken care of in the transformation phase.

2. Train From Scratch

After loading the data, transformations to be used as input for the custom Dataset class should be defined.
In this first attempt, a neural network is trained from scratch, meaning that the net weights have not already been trained on other data in a previous stage.
The transformations defined are resizing and crop centering, allowing to re-scale the image and then crop a central square of size 224x224, as expected as input by the net that will be trained.
Then the result is converted into a tensor (multidimensional array) and normalized with the mean and standard deviation equal to 0.5 for each channel, yielding values in the range [-1,1].

Finally, data is loaded into Dataset objects, passing the transformations as parameters. The result obtained can be compared with the original images in the previous paragraph:

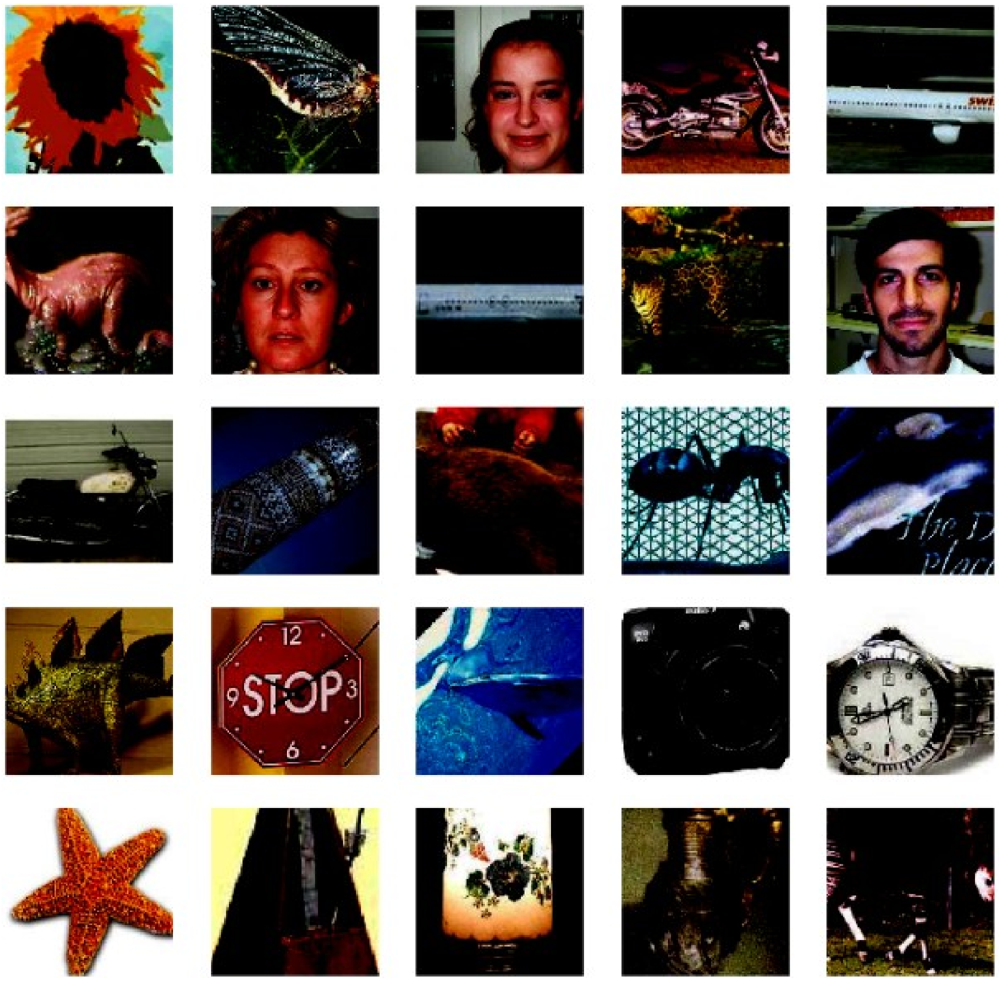
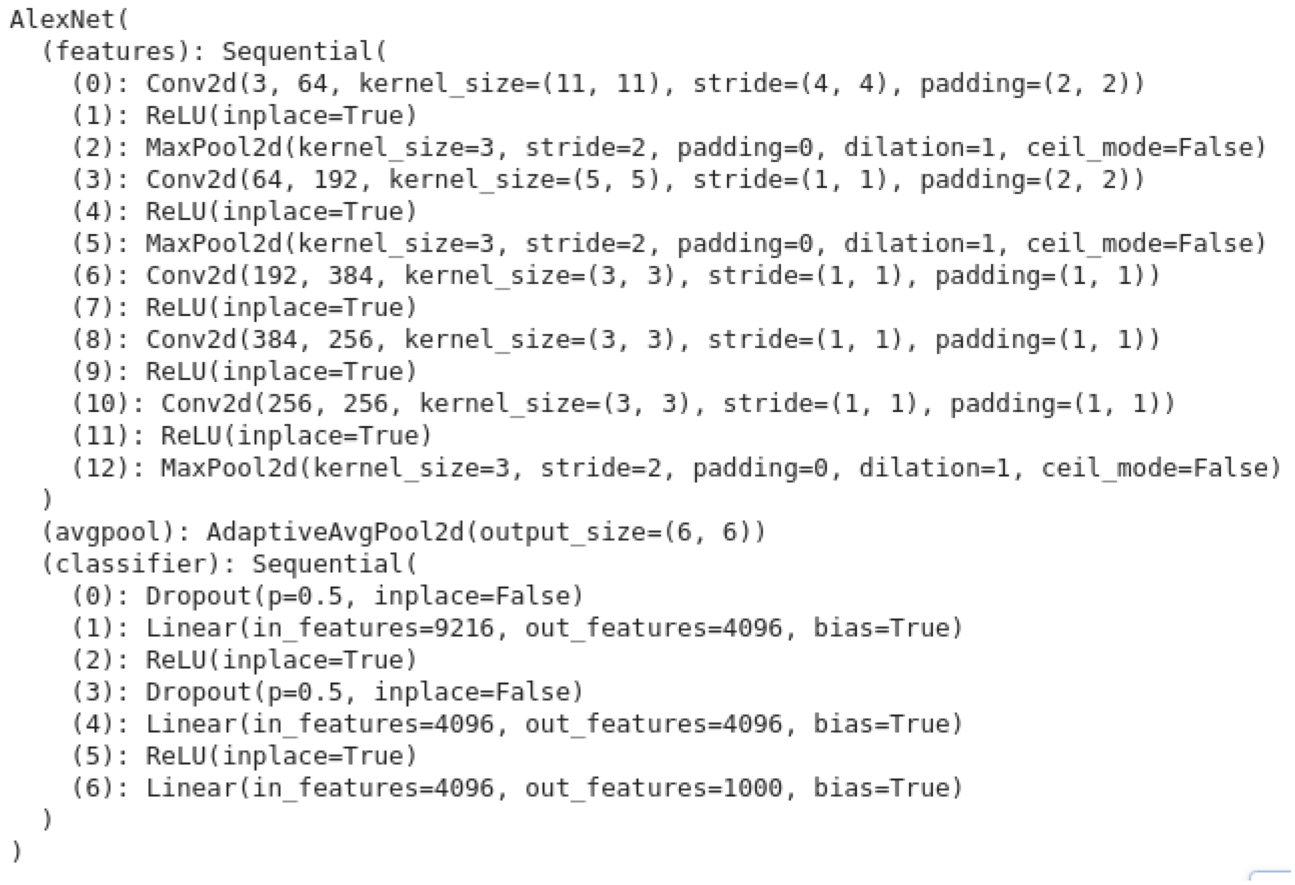


Figure 3: Images after transformations

At this point, since the input data is ready (all images have the same size), we focus on the network.

AlexNet is a neural network made up of 8 layers: 5 convolutional layers and 3 fully connected layers. It uses ReLU as activation function and Max Pooling to downsample. Dropout is applied as a form of regularization to reduce overfitting.
In the next figure the details of the architecture are displayed.



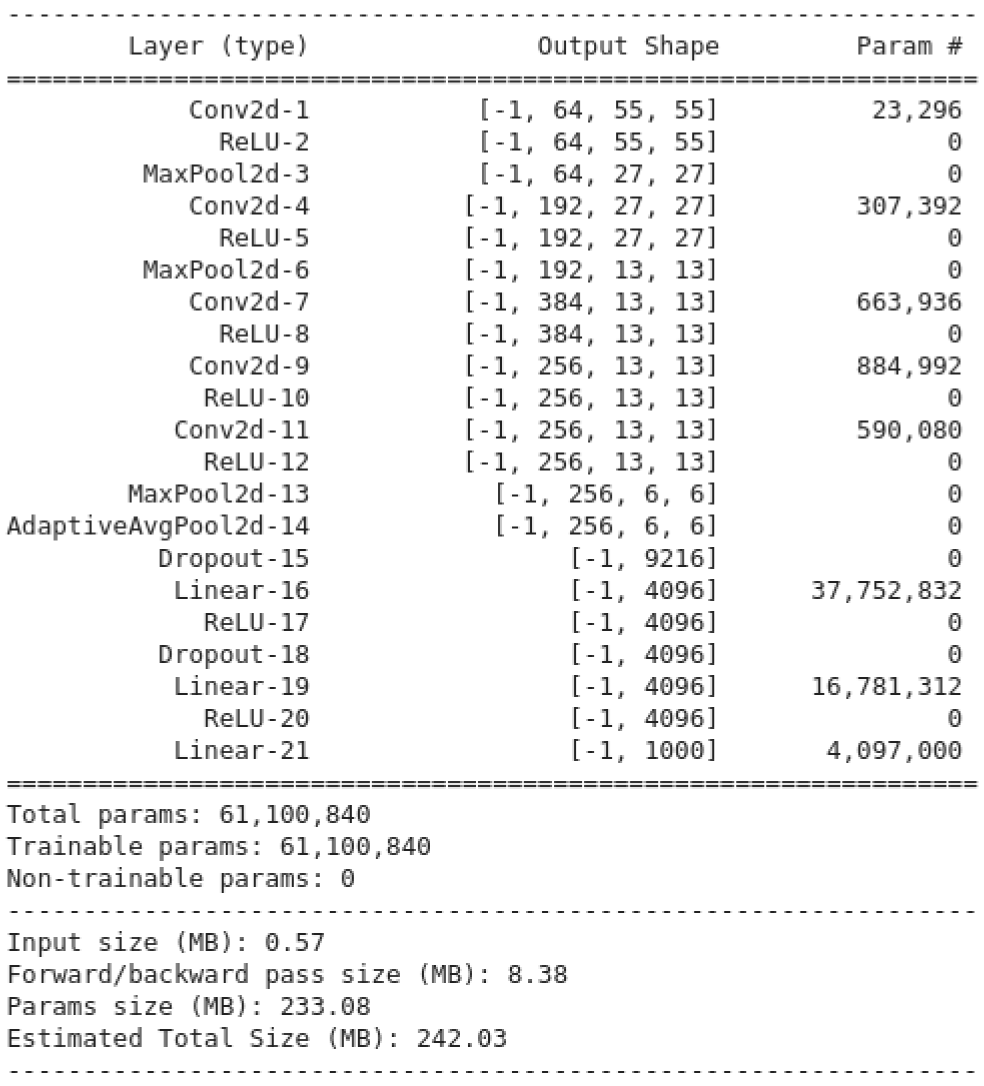
```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Figure 4: AlexNet architecture (with filter details)

While the previous figure shows the size, stride and padding of the filter, the one in the following page (Figure 5, obtained through the “torchsummary” library) highlights the output shape of each layer and the number of parameters.

It can be noticed that the PyTorch implementation is slightly different from the version seen during the course, but the main characteristics are still the same.

Before doing the training, the dataloaders for train, validation and test sets are initialized.
These objects will be used to retrieve batch of the dimension specified from the dataset.
Other then the batch size, other parameters to be specified are “shuffle”, set to True for train and validation to reshuffle the data at every epoch, and “drop last”, to decide if the last batch should be dropped if not divisible by the batch size.



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 55, 55]          23,296
              ReLU-2           [-1, 64, 55, 55]               0
         MaxPool2d-3           [-1, 64, 27, 27]               0
            Conv2d-4          [-1, 192, 27, 27]         307,392
              ReLU-5          [-1, 192, 27, 27]               0
         MaxPool2d-6          [-1, 192, 13, 13]               0
            Conv2d-7          [-1, 384, 13, 13]         663,936
              ReLU-8          [-1, 384, 13, 13]               0
            Conv2d-9          [-1, 256, 13, 13]         884,992
             ReLU-10          [-1, 256, 13, 13]               0
           Conv2d-11          [-1, 256, 13, 13]         590,080
             ReLU-12          [-1, 256, 13, 13]               0
        MaxPool2d-13            [-1, 256, 6, 6]               0
AdaptiveAvgPool2d-14            [-1, 256, 6, 6]               0
          Dropout-15                 [-1, 9216]               0
           Linear-16                 [-1, 4096]      37,752,832
             ReLU-17                 [-1, 4096]               0
          Dropout-18                 [-1, 4096]               0
           Linear-19                 [-1, 4096]      16,781,312
             ReLU-20                 [-1, 4096]               0
           Linear-21                 [-1, 1000]       4,097,000
================================================================
Total params: 61,100,840
Trainable params: 61,100,840
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 8.38
Params size (MB): 233.08
Estimated Total Size (MB): 242.03
----------------------------------------------------------------
```

Figure 5: AlexNet architecture (with parameter details)

Before proceeding with training, the last (fully connected) layer of the net should be changed to include 101 neurons (equal to the number of class labels) instead of the default 1000 (because the network was originally trained on the 1000 ImageNet's classes).
The Cross-entropy loss is chosen as loss function to estimate the difference between the prediction and the ground truth.
Stochastic gradient descent is the optimization technique used to calculate the gradient (that is then used to update the network weights).
The scheduler is the component responsible for the learning rate decay, that is, decreasing the learning rate after a defined number of epochs.

Specific functions have been implemented for training for one epoch, evaluate on the validation set and get the accuracy on the test set. The function "train" include both training and evaluation, since they are called many times over the notebook.

Model Tuning

For hyperparameter tuning there are different well-known techniques, among them: grid search, random search and “babysitting”.
Grid search is the simplest way to get hyperparameters, using brute force: different sets of parameters are combined together to see which ones work best.
Anyway, many times trying random hyperparameters leads to better results, especially in the case in which “important” parameters are combined with “unimportant” ones.
Unfortunately, both these techniques are computationally expensive.

For this reason, in this homework the last approach is proposed: parameters are obtained through trials and errors. In this manual method different parameters are tested in sequence, starting from the most important one (according to the literature) such as learning rate, learning rate decay, batch size and so on. Coarse to fine search is used when possible.

The most critical hyperparameter to tune in a neural network is the learning rate: the term that determines the step size used to move towards the minimum of the loss function. If it is too low it takes too long to reach the minimum, instead the larger the learning rate the higher the risk of diverging and making the accuracy decrease.

In particular, for this homework many values are tested:

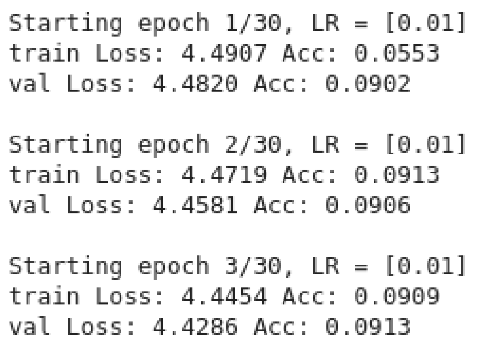
- lr = 1e-3, too small, was found to reach a low accuracy and a low reduction in loss during training;
- with lr = 1 (too big) the loss increase too much (it increases to NaN) in the first few steps;
- if lr = 1e-1 after half the iterations the model diverges too.

For this reason lr = 1e-2 is chosen.

0.9, 0.99 and 0.999 are tried for the momentum term, but 0.9 (default value provided with the homework template) gives the best results.

Still focusing on the learning rate, another parameter is the step-down policy, sometimes also called the learning rate decay: how much the learning rate is reduced over time. In particular with PyTorch the step_size tells after how many epochs the learning rate should be decreased.
The intuition behind this is that once we are approaching the minimum we want to slow down to avoid overshooting it.

This parameter is not tested intensively, since only two set of parameters were requested; anyway, using step size = 10 or 15 results in a worse accuracy, because reducing the learning rate too early. The default value (20) is chosen.

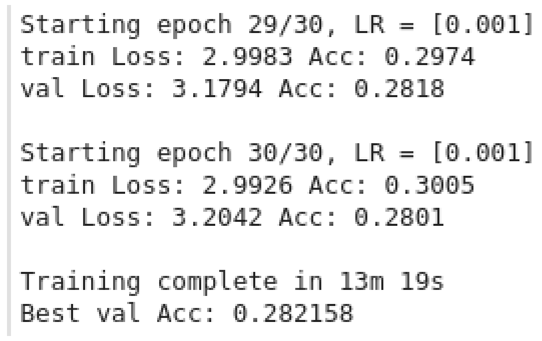


```
Starting epoch 1/30, LR = [0.01]
train Loss: 4.4907 Acc: 0.0553
val Loss: 4.4820 Acc: 0.0902

Starting epoch 2/30, LR = [0.01]
train Loss: 4.4719 Acc: 0.0913
val Loss: 4.4581 Acc: 0.0906

Starting epoch 3/30, LR = [0.01]
train Loss: 4.4454 Acc: 0.0909
val Loss: 4.4286 Acc: 0.0913
```

[...]



```
Starting epoch 29/30, LR = [0.001]
train Loss: 2.9983 Acc: 0.2974
val Loss: 3.1794 Acc: 0.2818

Starting epoch 30/30, LR = [0.001]
train Loss: 2.9926 Acc: 0.3005
val Loss: 3.2042 Acc: 0.2801

Training complete in 13m 19s
Best val Acc: 0.282158
```

Figure 6: Results obtained

In the figure, the plot of accuracy and loss for both train and validation, considering the network trained with lr=1e-2, step_size=20 and momentum=0.9.

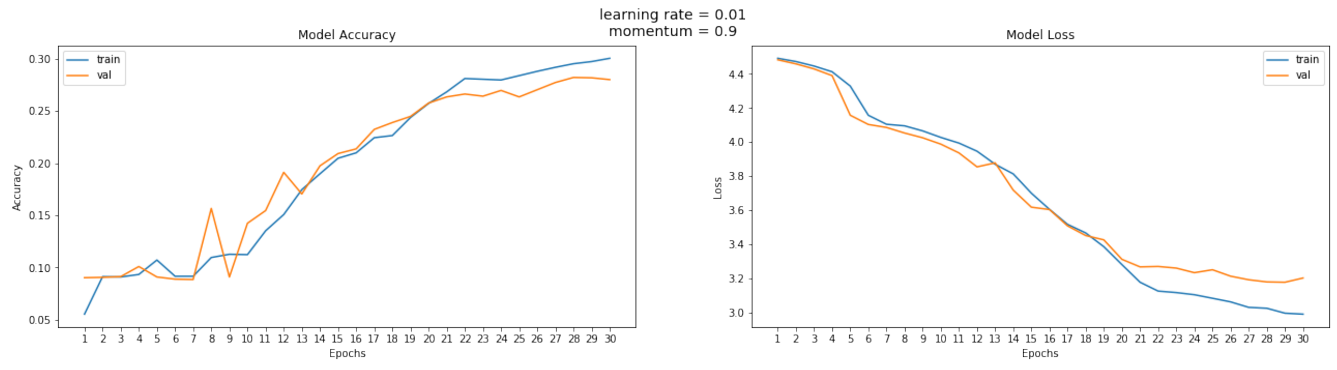


Figure 7: Model accuracy and loss

The final accuracy in the test set is about 0.3.

Even if the trial and error approach is discussed, code for grid search and random is also reported in the homework code. For random search log scale is adopted when selecting parameter values for learning rate and momentum, as suggested in the literature.

The final test accuracy is quite low, but looking at the plots it may be guessed that increasing the number of epochs would have further improved the model, considering that the last part of the curves is not flat but continues increasing for the accuracy and decreasing for the loss.

In this homework the number of epochs is not increased to let the training time remain less than 10 minutes.

3. Transfer Learning

Since it is relatively rare to have a dataset of sufficient size, many times transfer learning is used: instead of training a neural network starting from scratch (with random initialization), it is common to pre-train a net on a very large dataset.

In this case, PyTorch allow to load AlexNet with weights trained on the ImageNet dataset.
Being the weights related to ImageNet, it is necessary to change the normalization step by using the ImageNet's mean and standard deviation.

3.1 Finetuning

Once the transformations are updated with the new normalization factors and the dataset is loaded using the specified batch size (parameter tuned, this time), the train and test functions defined before are called.

For the tuning, the initial parameters chosen as a starting point (being the ones that led to good results in the training from scratch phase) are lr = 1e-2, step_size = 20, momentum 0.9 and batch size = 256.

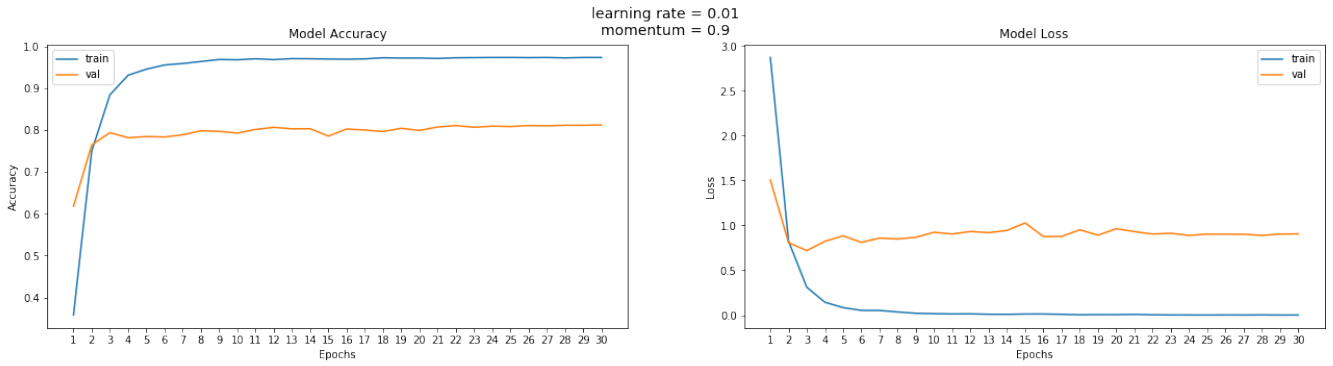


Figure 8: Transfer learning with learning rate = 1e-2, weight decay = 5e-5, batch size = 256

Two main considerations can be made by looking at the plots obtained: finetuning has increased substantially performance (from about 0.3 0.7 accuracy on test set), but at the same time overfitting can be detected quite easily because the two curves, in both plots, are too far away from each other: the model does a good job on the train set, but is not good enough on the validation set (does not generalize well).

Starting from this consideration, it is useful to investigate and try to tune the hyperparameters to deal with this problem.
The usual way to overcome overfitting is collecting more data (not feasible in our case) and add regularization.

For this reason the weight decay is tuned: starting from the initial value of 5e-5, it is experimented with 5e-3, leading to similar results and not much improvement; then 1e-1, resulting this time in underfitting (degradation in accuracy already on train set).
Then, going from coarse to fine, 1e-2 is another value tested, but again the two curves are still distant and with this update they are very unstable (continue to fluctuate, moving up and down).
So trying to penalize our model decreasing the weights is not the solution.

Changes in the learning rate are not of much help, leading to situations similar to training from scratch.

The last hyperpatameter tuned (trying to solve the overfitting problem) is the batch size. Among the trials the one with batch size 512 leads to an improvement in accuracy and a reduction in the gap among the two curves.

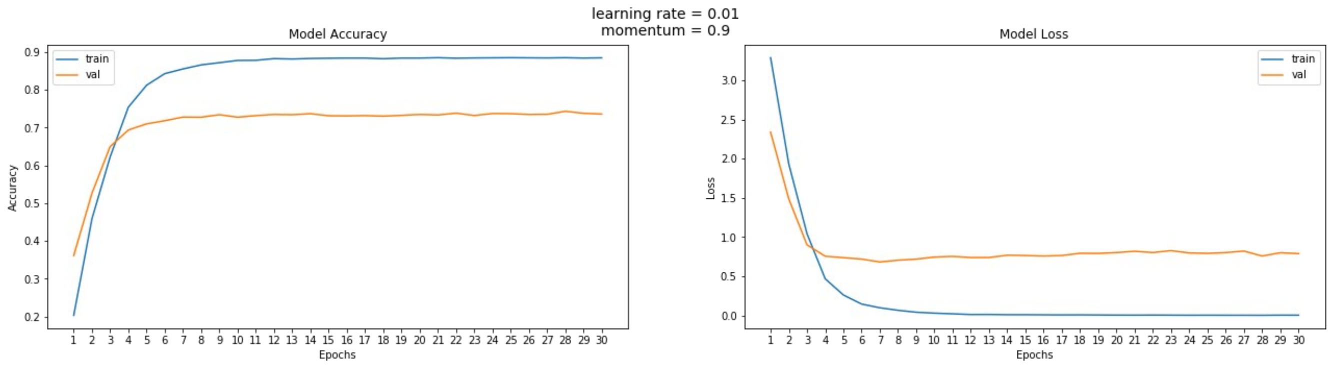


Figure 9: Transfer learning with learning rate = 1e-2, weight decay = 5e-5, batch size = 512

The final setup is ls = 1e-2, step_size = 20, momentum = 0.9, weight_decay = 5e-5 and batch_size = 512, leading to a test accuracy of about 0.84.

Maybe an alternative good solution (to overcome overfitting) would have been to add more samples in the training with respect to the validation set, but the half split was a constraint of the homework.

3.2 Freeze layers

Another strategy used in transfer learning consists in “freezing” part of the network and only train the remaining part, since it is faster and can prevent overfitting in some cases.
In this homework two strategies are adopted: first only fully connected layers are trained, then only convolutional ones.

3.2.1 Train only fully connected layers

When freezing convolutional layers and training fully connected layers, gradients are calculated in the backward propagation, but only the weights of the fully connected layers are updated.
This approach is more straightforward than the dual one (train only convolutional layers) and is sometimes referred to as using the network as a *fixed feature extractor*, because it fixes the convolutional layers (that act as feature extractors in a neural network).

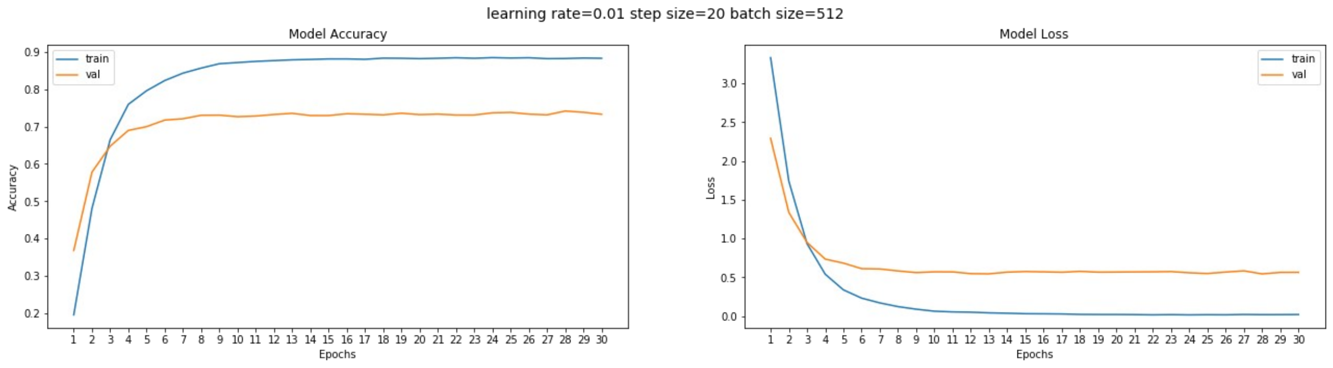


Figure 10: Freeze convolutional layers

The model still suffers of overfitting and only a few seconds are gained in terms of training speed (maybe the difference would have been greater training for a larger amount of epochs).
It achieves good performances, with an accuracy close to 0.85, similar to the one obtained before.

3.2.2 Train only convolutional layers

This dual method is less common in machine learning practices and the reason can be easily understood looking at the model accuracy plot:

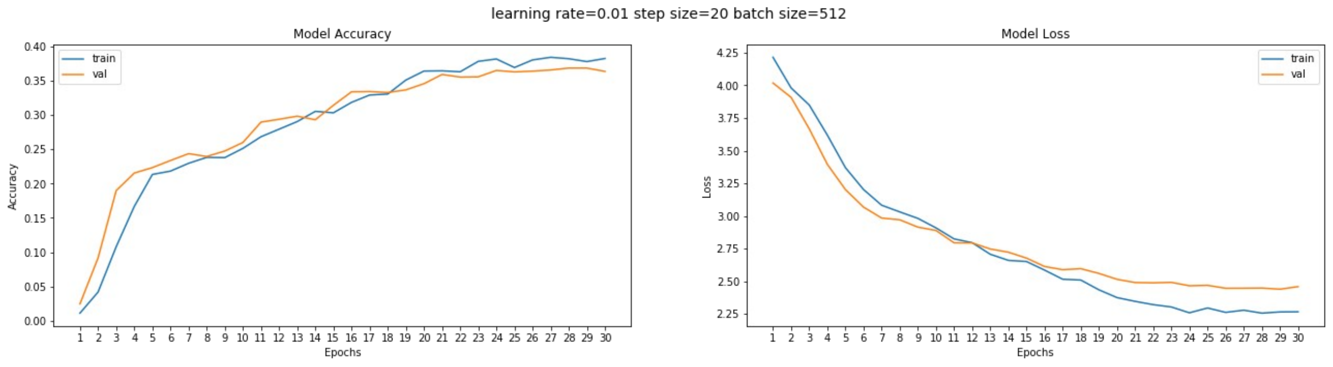


Figure 11: Freeze fully connected layers

Overfitting is not a problem anymore, but the model is affected by underfitting, reaching a low final accuracy of about 0.41.

4. Data Augmentation

Data augmentation is another path that can be pursued when the initial dataset is not big enough: it allows to artificially increase the dataset size by applying transformations.
Training using this additional data generated from the existing one helps the network generalize and prevents overfitting.

The transformations applied in this homework are:

- RandomResizedCrop: instead of cropping the center of the image, this time the crop position is random
- RandomHorizontalFlip: horizontally flip the image (with a probability of 50% in the provided implementation)
- RandomRotation: the image is rotate by 90 degrees clockwise or counterclockwise

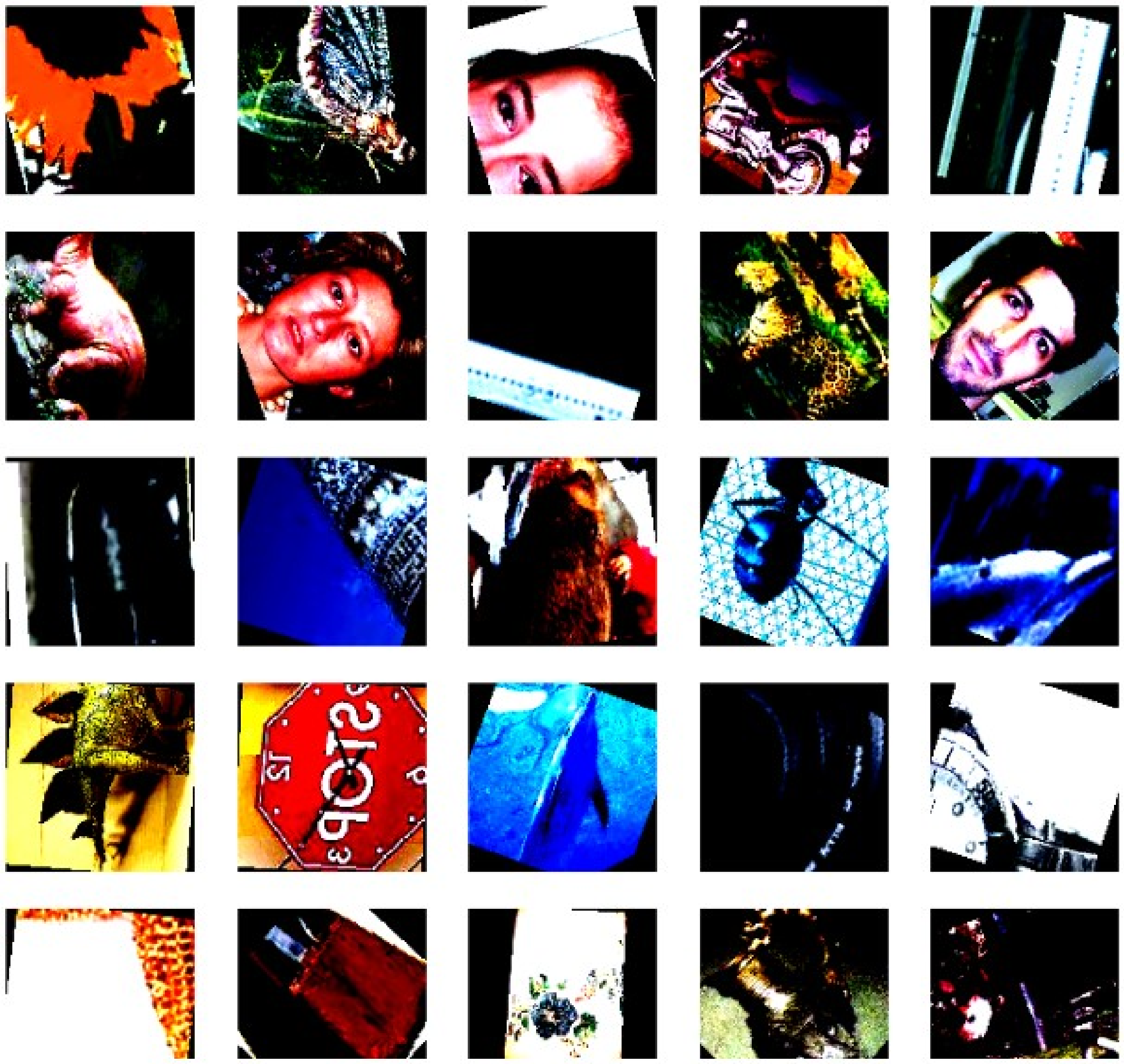


Figure 12: Images after data augmentation

In the figure above, the modification with respect to the initial images can be appreciated and the kinds of transformation applied can be easily guessed.

In practice, using transformations, the dataset size does not increase (creating new images) but the operations are applied to the original dataset at every batch generation.
Therefore, the dataset is left unchanged, only the batch images are copied and transformed at every iteration.

The parameters used are the ones that yield the best results in the transfer learning phase: lr = 1e-2, step_size = 20, batch size = 256, momentum = 0.9 and weight_decay = 5e-5.

The final accuracy is about 0.75 and the plots obtained are smooth and without large fluctuations.

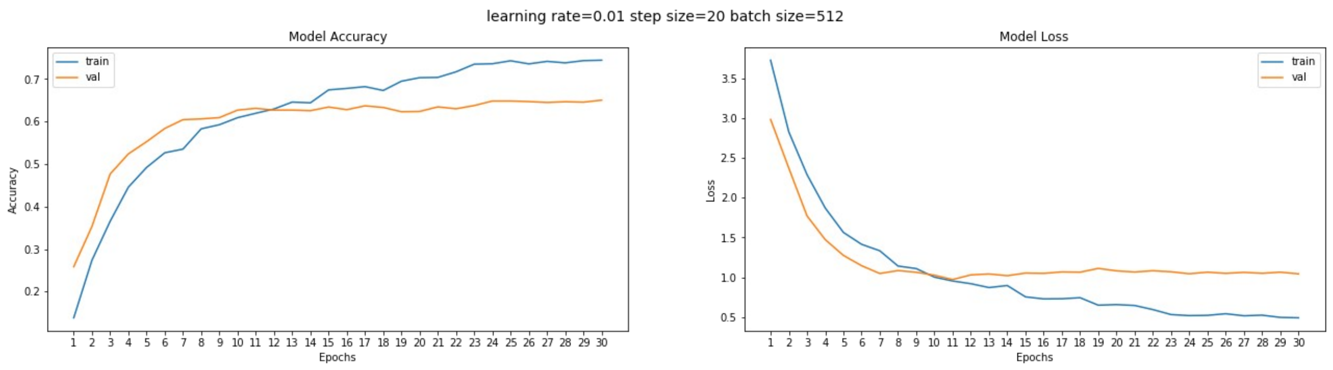
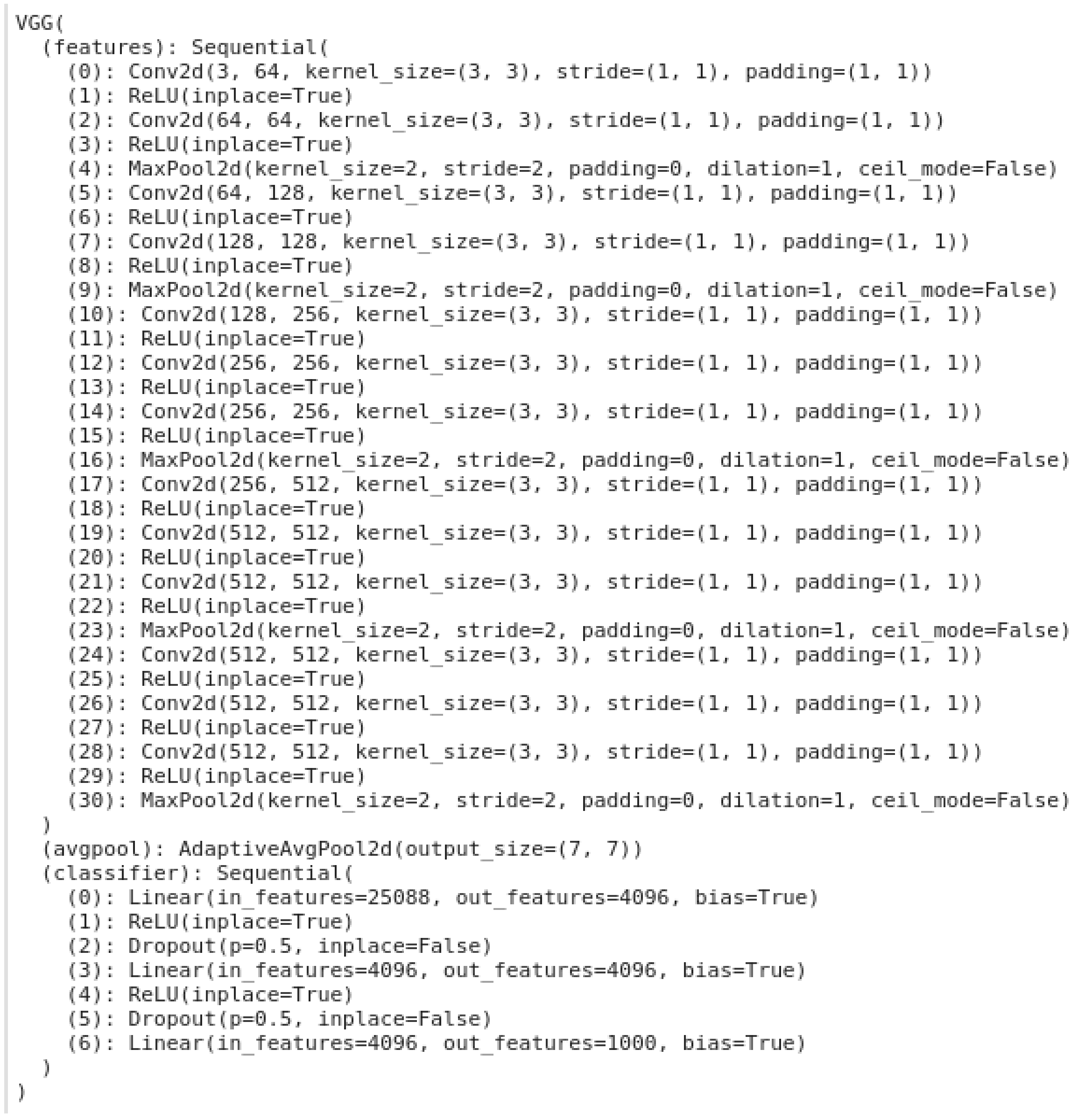


Figure 13: Plot of accuracy and loss using data augmentation

5. (Extra) Beyond AlexNet

In this section we switch to another neural network: VGGNet.
In particular, the 16-layers version, containing 13 convolutional layers and 3 fully connected layers.



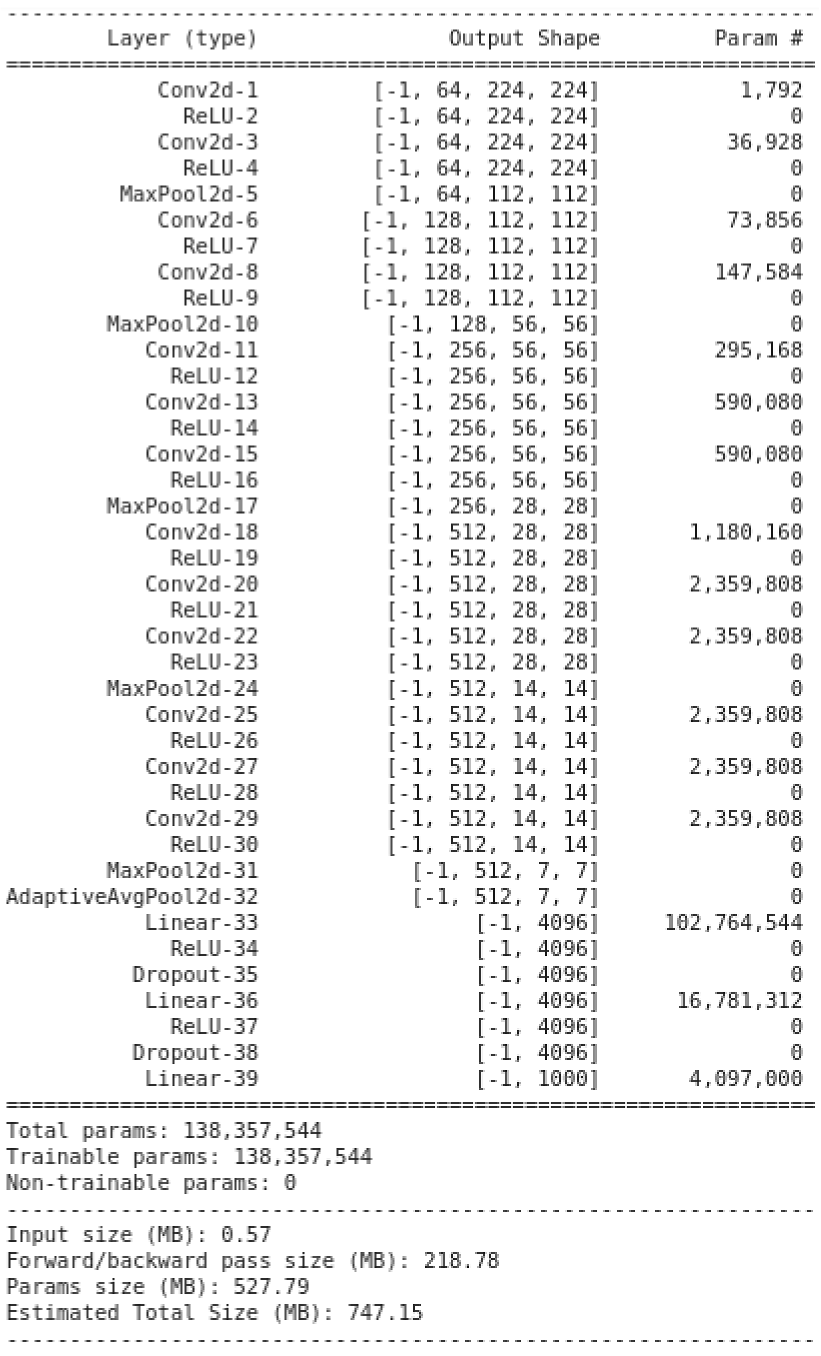
```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Figure 14: VGGNet architecture (with filter details)

It is a deeper network than AlexNet and its main characteristic is to use small filters (3x3), with the advantage of getting a deeper network with more non-linearity, but maintaining the same effective receptive field of 7x7 layers (where the receptive field is the size of the input that a particular neuron is influenced by).

Parameters lr = 1e-2, step_size = 20 and momentum = 0.9 are maintained from previous steps, only batch size is changed, set to 64.
Being the network a bigger one ,more memory is required and a batch size of 256 is not fitting in RAM, as signaled by the Colab runtime error *“CUDA out of memory”*.

The maximum power of 2 that Colab can handle in this configuration is 64.
An approximation of the amount of memory required can be seen in the next figure (about 3x with respect to AlexNet).



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 224, 224]           1,792
              ReLU-2         [-1, 64, 224, 224]               0
            Conv2d-3         [-1, 64, 224, 224]          36,928
              ReLU-4         [-1, 64, 224, 224]               0
         MaxPool2d-5         [-1, 64, 112, 112]               0
            Conv2d-6        [-1, 128, 112, 112]          73,856
              ReLU-7        [-1, 128, 112, 112]               0
            Conv2d-8        [-1, 128, 112, 112]         147,584
              ReLU-9        [-1, 128, 112, 112]               0
        MaxPool2d-10          [-1, 128, 56, 56]               0
           Conv2d-11          [-1, 256, 56, 56]         295,168
             ReLU-12          [-1, 256, 56, 56]               0
           Conv2d-13          [-1, 256, 56, 56]         590,080
             ReLU-14          [-1, 256, 56, 56]               0
           Conv2d-15          [-1, 256, 56, 56]         590,080
             ReLU-16          [-1, 256, 56, 56]               0
        MaxPool2d-17          [-1, 256, 28, 28]               0
           Conv2d-18          [-1, 512, 28, 28]       1,180,160
             ReLU-19          [-1, 512, 28, 28]               0
           Conv2d-20          [-1, 512, 28, 28]       2,359,808
             ReLU-21          [-1, 512, 28, 28]               0
           Conv2d-22          [-1, 512, 28, 28]       2,359,808
             ReLU-23          [-1, 512, 28, 28]               0
        MaxPool2d-24          [-1, 512, 14, 14]               0
           Conv2d-25          [-1, 512, 14, 14]       2,359,808
             ReLU-26          [-1, 512, 14, 14]               0
           Conv2d-27          [-1, 512, 14, 14]       2,359,808
             ReLU-28          [-1, 512, 14, 14]               0
           Conv2d-29          [-1, 512, 14, 14]       2,359,808
             ReLU-30          [-1, 512, 14, 14]               0
        MaxPool2d-31            [-1, 512, 7, 7]               0
AdaptiveAvgPool2d-32            [-1, 512, 7, 7]               0
           Linear-33                 [-1, 4096]     102,764,544
             ReLU-34                 [-1, 4096]               0
          Dropout-35                 [-1, 4096]               0
           Linear-36                 [-1, 4096]      16,781,312
             ReLU-37                 [-1, 4096]               0
          Dropout-38                 [-1, 4096]               0
           Linear-39                 [-1, 1000]       4,097,000
================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 218.78
Params size (MB): 527.79
Estimated Total Size (MB): 747.15
----------------------------------------------------------------
```

Figure 15: VGGNet architecture (with parameter details)

As expected, it takes much longer to train a bigger network (more than 50 minutes), but the advantages can be seen in the final accuracy achieved on the test set of about 0.83.

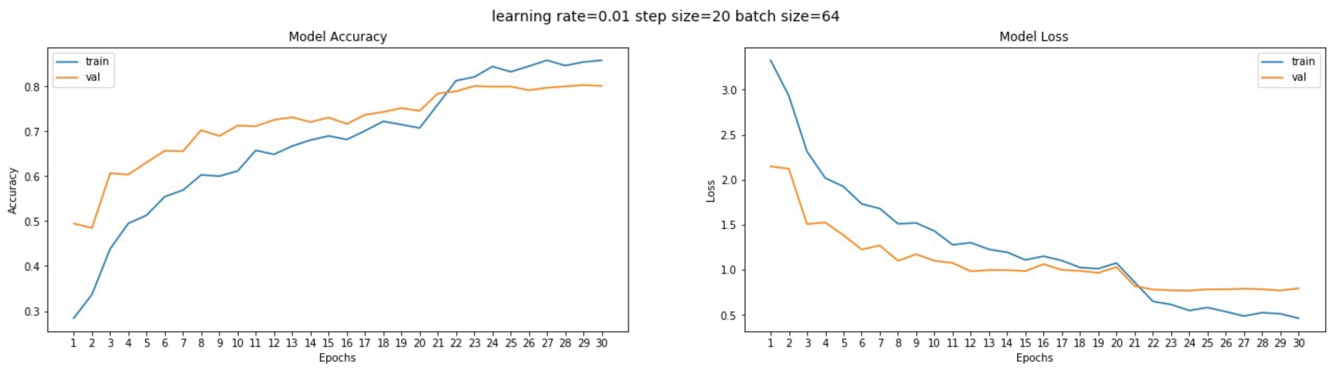


Figure 16: caption