AML Assignment 3 - report

May 4, 2020

Fabio Montello (1834411), Francesco Russo (1449025), Michele Cernigliaro (1869097)

1 Overview

In this report we proceed to answer as requested all the questions of the homework 3. For each point we are going to write a brief description using figures and formulas whenever needed.

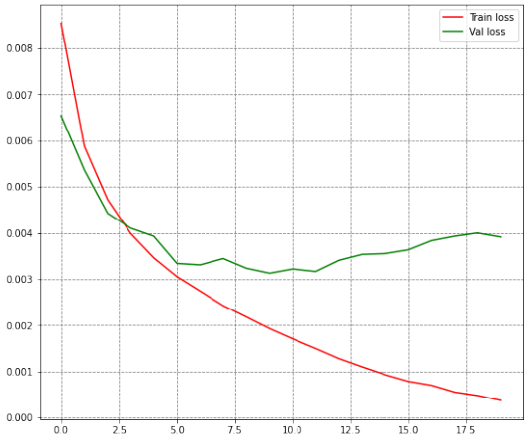
2 Question 1a

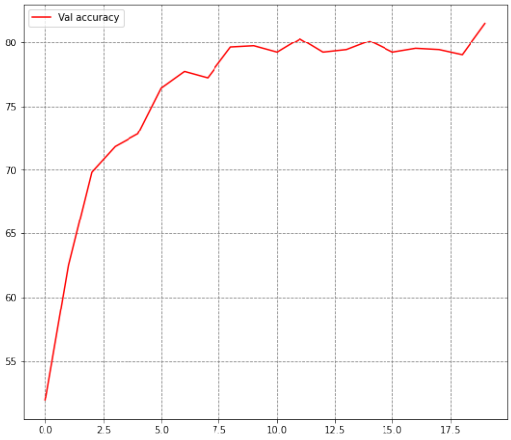
Here below we present the ConvNet implementation with pytorch:

```
class ConvNet(nn.Module):
    def __init__(self, input_size, hidden_layers, num_classes, norm_layer=None):
        super(ConvNet, self).__init__()
        layers = []
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        # First ConvBlock with input size (i.e. C=3) and first hidden layer(i.e. 128)
        layers.append(nn.Conv2d(input_size, hidden_layers[0], kernel_size=3, stride=1, padding=1))
        if norm_layer=="BN":
          layers.append(nn.BatchNorm2d(hidden_layers[0], eps 1e-05, momentum 0.1,
                                       affine=True, track_running_stats=True))
        layers.append(nn.ReLU())
        layers.append(nn.MaxPool2d(kernel_size 2, stride 2))

        # Adding the other blocks
        for Din, Dout in zip(hidden_layers[:-1], hidden_layers[1:]):

          layers.append(nn.Conv2d(Din, Dout, kernel_size=3, stride=1, padding=1))
          if norm_layer=="BN":
            layers.append(nn.BatchNorm2d(Dout, eps=1e-05, momentum=0.1,
                                         affine=True, track_running_stats=True))
          layers.append(nn.ReLU())
          layers.append(nn.MaxPool2d(kernel_size=2, stride=2))

        # stacking convolutional blocks
        self.ConvBlocks = nn.Sequential(*layers)
        self.Dout = hidden_layers[-1]

        # fully connected layer
        self.Dense = nn.Linear(hidden_layers[-1], num_classes)
```

Notice that we included also the Batch Normalization as requested in the point 2a, but the code

has been executed without any normalization layer set ("norm_layer" flag set to None). We will plot the traces of the loss and accuracy values along 20 training epochs:



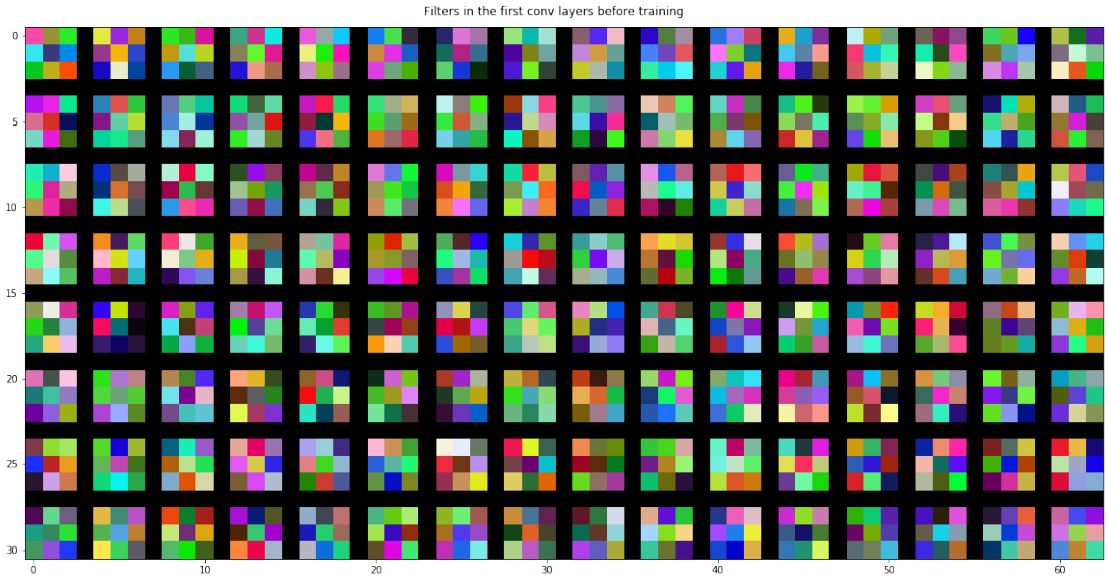


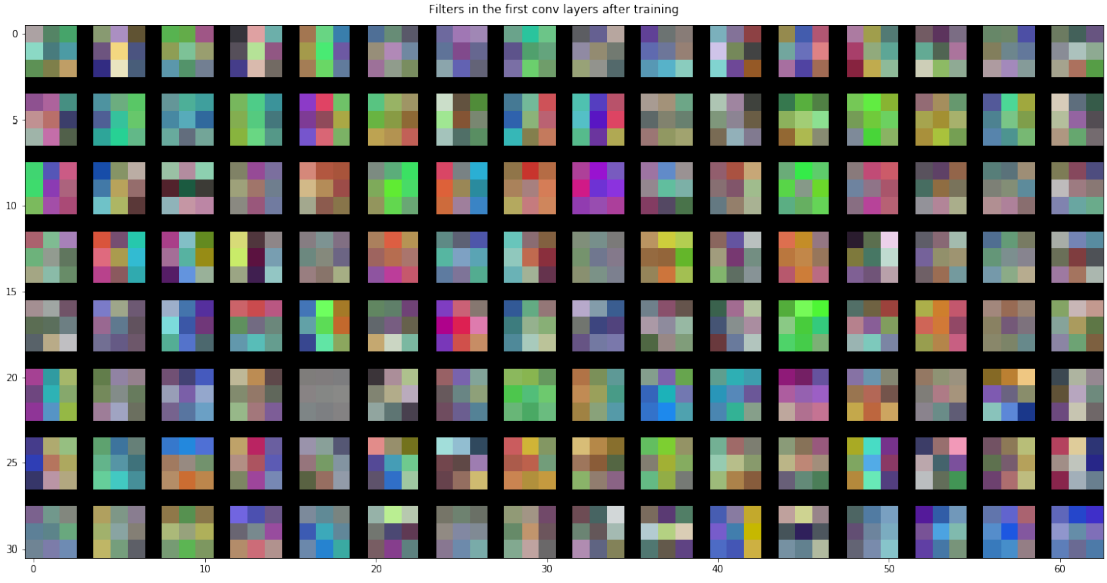
Our best result (see early stopping at question 2b) obtained 81.4% of accuracy on the validation and 79.3% on the test set. According to the loss graph, it seems that model is slightly overfitting starting from the seventh epoch. We will try to reduce the overfitting in the next steps with data augmentation and dropout, as requested from the homework.

3 Question 1b

Implementing the function 'PrintModelSize' we found out that our convnet has 7,678,474 parameters without the batch layers, and 7,682,826 considering also the new scale (γ) and shift (β) parameters to be learned in each BN layer.

4 Question 1c



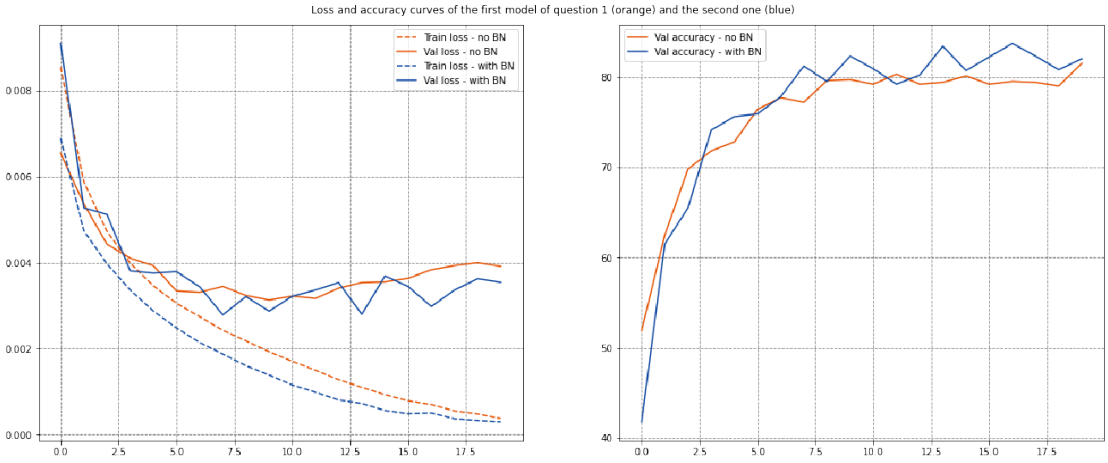


In the first image we cannot distinguish any distinct pattern, as we expected since the weights are initialized randomly. In the second image, altough the filters are 3x3, we are able to recognize low level feautures such as edge and/or color detection.

5 Question 2a: batch normalization

In order to add batch normalization layers in each Convolutional block, it is sufficient to set the "norm_layer" flag to "BN". As we designed the ConvNet class, it will add BN's automatically (see the architecture in the first image in question 1a).

As suggested by the question 2a, we performed the training with the same hyperparameters used in the tasks in question 1. We report below the loss and accuracy curves comparing the two models (with and without batch normalization).



The loss curves are slightly better for the model with batch normalization (even if its validation curve seems a bit more fuzzy, but almost always lower than the one of the model without BN). The same trend is observable in the accuracy plot, where the model with batch normalization exceeds the normal one starting from the 3rd epoch.

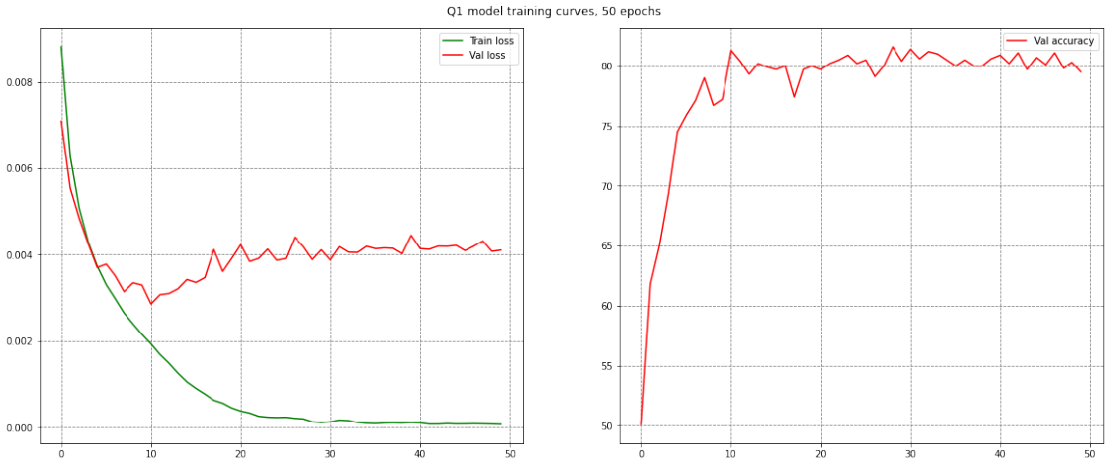
6 Question 2b: Early stopping

As already mentioned, we performed early stopping (to all the models implemented in this homework) in order to save the best model weights (according to the validation set) for the final predictions on the test set.

In question 2b it was asked moreover to increase the training epochs to 50 in both Q1a and Q2a, and compare for each one of them the best model and latest model. Here below we report the results we obtained after the training procedure:

Q1a - No Batch Normalization

We report below the training curves for 50 epochs.



The table below indicates the differences between the latest and the best model:

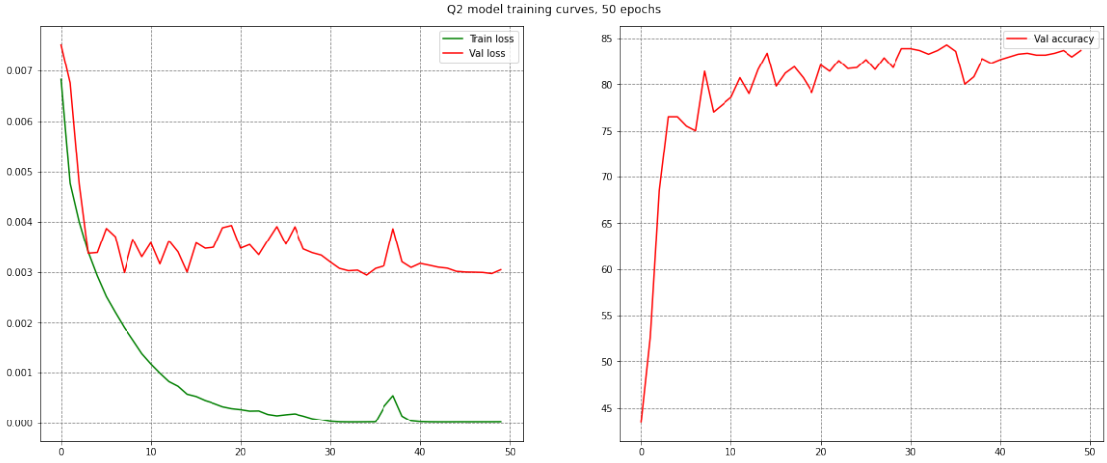
[8]:

	latest_model	best_model
train_loss	7.408e-5	1.33e-4
val_loss	4.11e-3	1.34e-4
val_acc	79.5%	81.6%

At the end, the latest model has 79.4% of accuracy on the **test set** while the best one 79.1%

Q2a - Batch Normalization

We report the training curves for 50 epochs and the table for the latest and best models comparison:



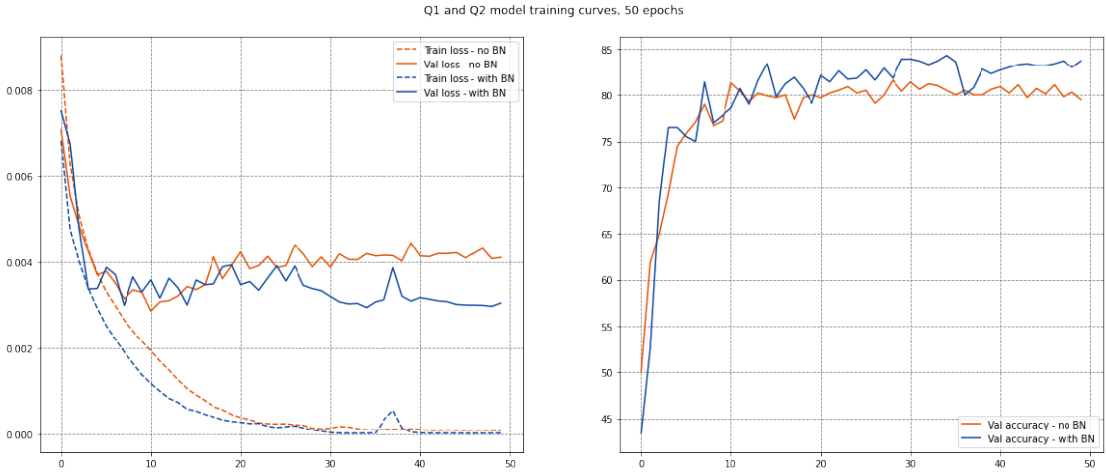
[10]:

	latest_model	best_model
train_loss	1.650e-5	1.6e-5
val_loss	0.003	1.6e-5
val_acc	83.7%	84.3%

At the end, the latest model gives 82.7% of accuracy on the **test set** while the best model reaches 83.7%

Q1a and Q2a - overwiew

As we've done before, we can visualize the training curves together. The result is the following:



To sum up, from all of this results we clearly see that:

- Even if the batch normalization does not helps directly with overfitting, it clearly enhances the model performances, allowing us to increase the model accuracy up to 2-3% (in our case, ~+4% on the final accuracy over the test set).
- Moreover, the early stopping helped our ConvNet model with batch normalization to save the parameters which **generalize well**, therefore giving us better results also on the test set.

7 Question 3a: Data Augmentation

Using data augmentation techniques allows to expand the original dataset by adding to it random transformations of the original images. In our task, we chose to include random transformations such as crops, horizontal flips, vertical flips, translations, rotations, grayscale, color jittering.

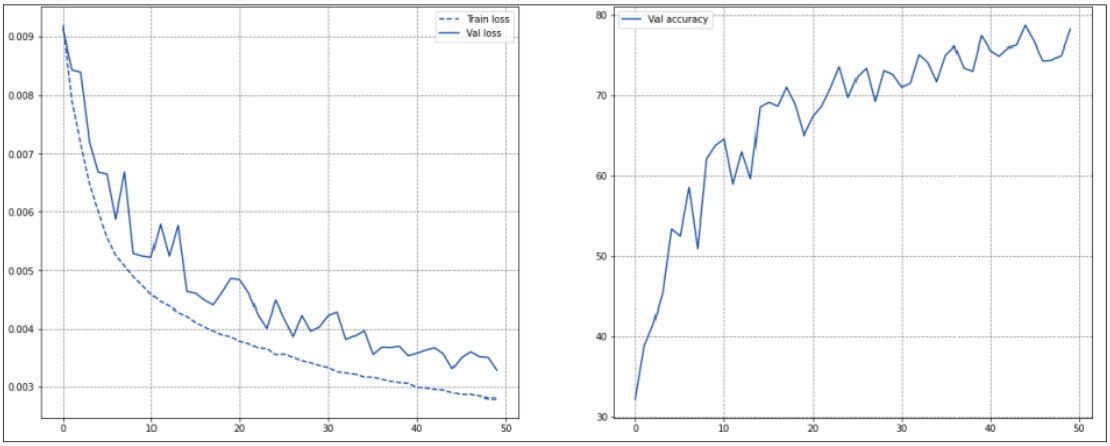
Here's the formal implementation of such operations in our code:

```
data_aug_transforms = [transforms.RandomCrop(32, padding=4),
                       transforms.RandomHorizontalFlip(),
                       transforms.RandomVerticalFlip(),
                       transforms.RandomRotation(5),
                       transforms.RandomGrayscale(),
                       transforms.RandomAffine(0, translate=[0.2,0.2], scale=None, shear=0, resample=False, fillcolor=0),
                       transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.05)]
```

We chose to set the cropping hyperparameter to 32 px, while the translation is sampled in [**-img_width * 0.2, img_width * 0.2**] on the x-axis and in [**-img_height * 0.2, img_height * 0.2**], and the rotation is in the range [**-5°, +5°**].

The relevant parameters in the color jittering are sampled from [**max(0, 1 - value), 1 + value**], where **value** is the value we passed to the function.

We then performed the training over 50 epochs, in order to take into account the increased size of the dataset, as specified in the assigment. The train and validation loss, and the validation accuracy, have the following behaviours:

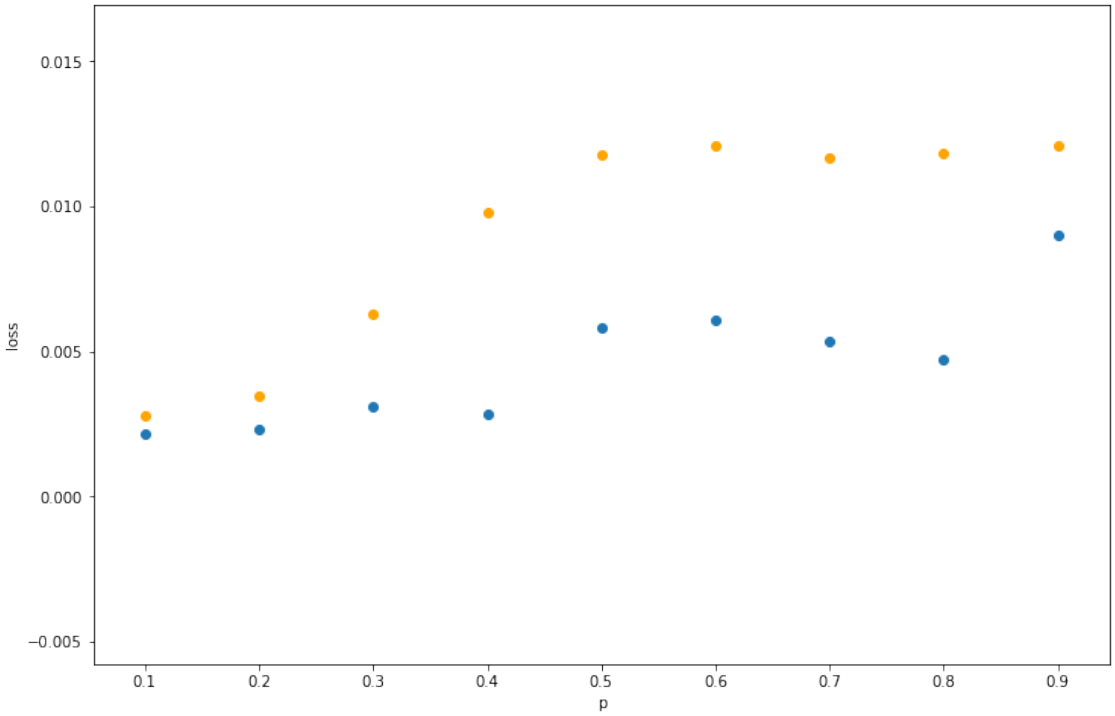


The final accuracy on validation is 78.2% while the accuracy on test is 82%

8 Question 3b: Dropout

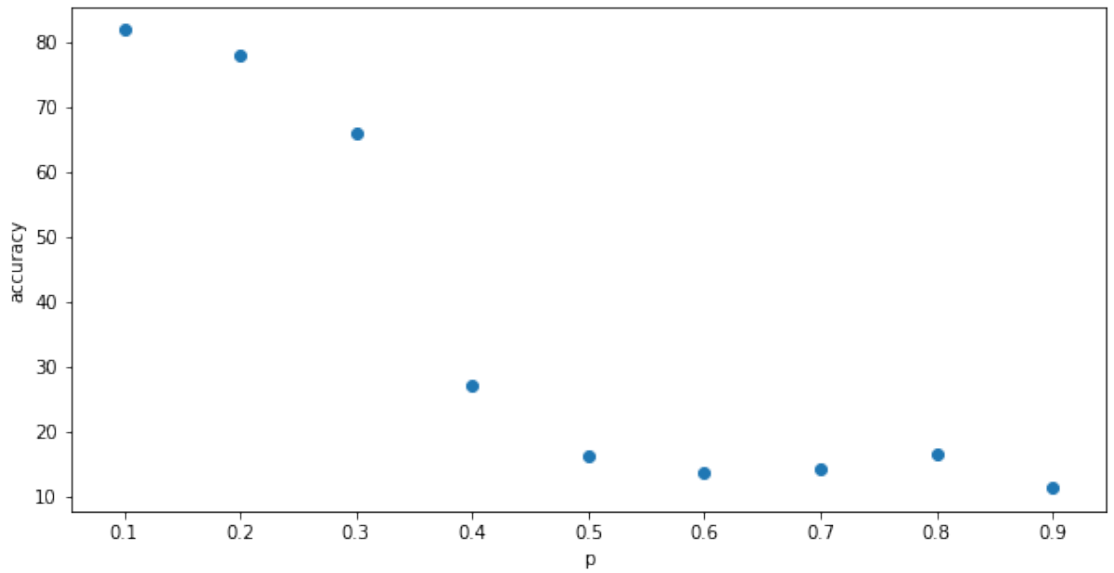
The dropout allows to try variations of the network by dropping units with a probability p. We tried to perform this for several different values of p, more specifically for the values in the set $[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]$

The following plot compares the best loss on train (blue) and validation (orange), for the different values of p



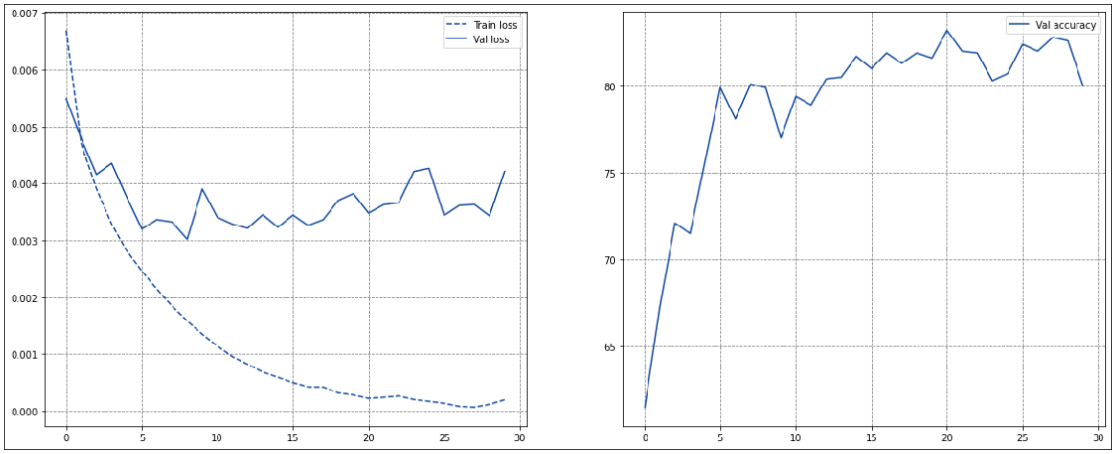
The next plot shows instead the best value of the validation accuracy for the different values of p

[17]: `<matplotlib.collections.PathCollection at 0x11e17bcd0>`



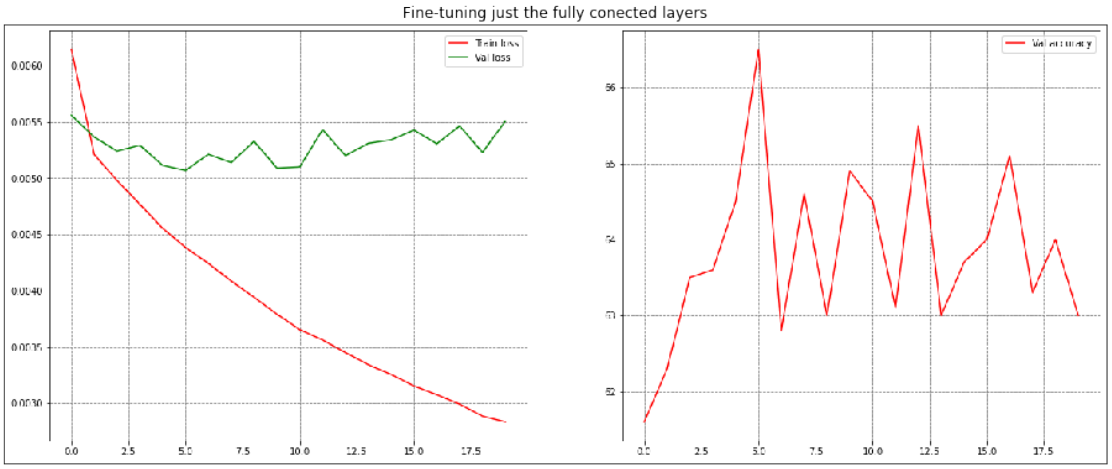
We can notice that we get the lowest value of the train and validation loss, and the highest value

of the validation accuracy, for $p = 0.1$. We therefore train the model again using this value of the hyperparameter, obtaining the following behaviours:



The final validation accuracy is 80% and the test accuracy is 81.6%

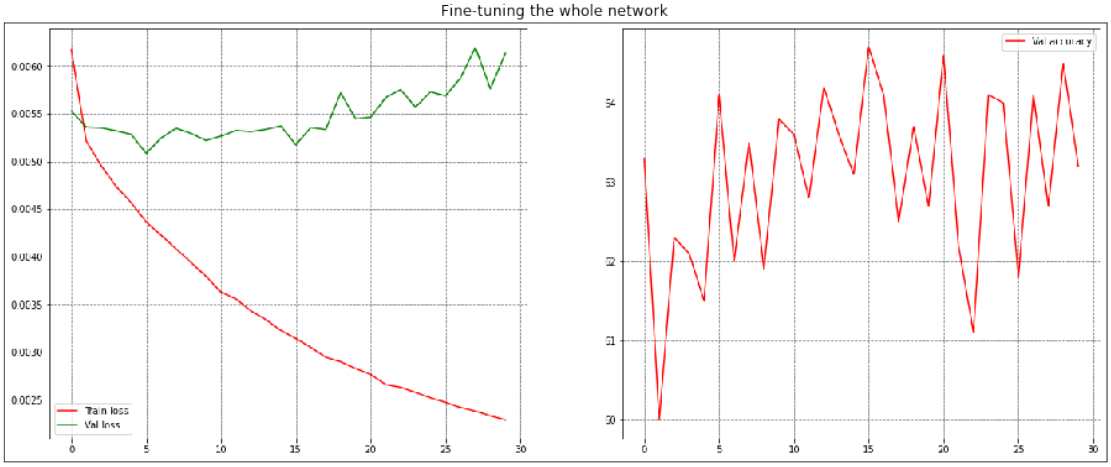
9 Question 4a



The accuracy we obtained by applying two fully connected layers on top of the pre-trained VGG11_bn model is: 63.2 %. This was obtained after 20 epochs of training only on the fully connected layers. The network seems to be overfitting, since training loss keeps decreasing but the validation loss and accuracy seems to be invariant.

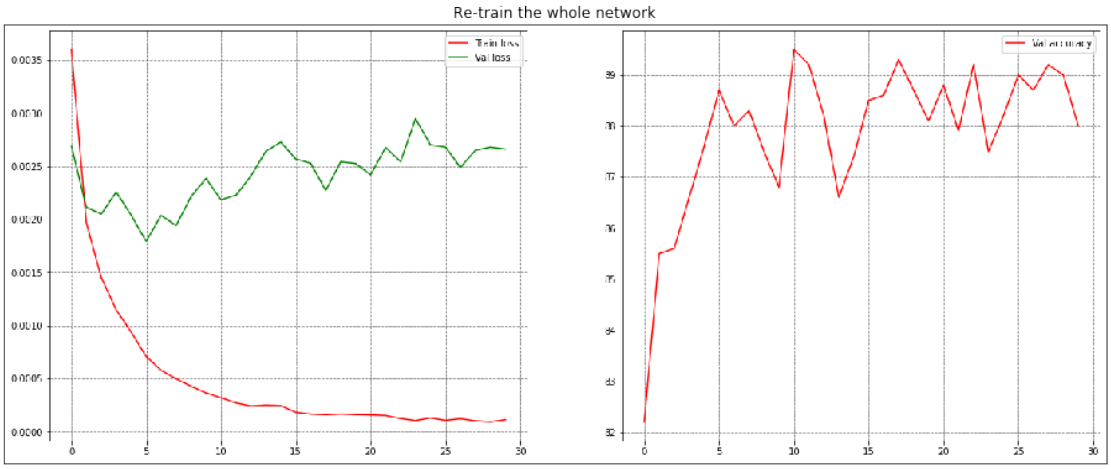
10 Question 4b

As requested from the excercise, we tried fist to fine-tune the whole network on the CIFAR-10 dataset, starting from the ImageNet initialization in order to improve the quality of the accuracy on the test set. The following are the results of our implementation on the training and validation sets:



The final accuracy on the test set is 62.9%, which is not an improvement to the results in the previous point, as confirmed also by the trend charts of train and validation sets plotted above. The reason for this result can still be due to the domain-shift between the ImageNet dataset (224x224 resolution images) and the CIFAR-10 dataset (32x32 images), which may be too big to just fine tune the network.

We are expecting much better results in the training from scratch of the whole network. Let's observe the trend plot of training and validation sets for this case:



As it is possible to observe, the results improves much more the network, bringing the validation accuracy almost at 90% and giving a final result on the test set of 88.8%, which is an huge improvement with respect to the only fine-tuning of the whole network or only the fully connected layers.