AML - Advanced Machine Learning
Assignment 2

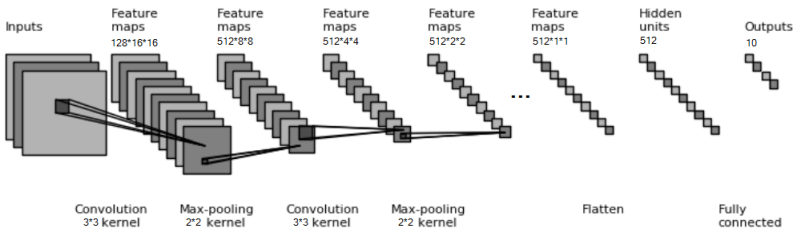
Instructor: Fabio Galasso

Giovanni Giunta, Marco Muscas,
Mehrzad Jafari Ranjbar, Zain Ullah

A.Y. 2021 - 2022

Question 1

As stated in the assignment, we will be implementing a Convolutional Neural Network (CNN) that has the following structure:



Question 1.a - The Architecture

The training model is composed by 5 convolution blocks to be followed by 2 fully connected layers. To go from the feature map of the last convolution (with shape $512 \times 1 \times 1$) to the output, we made use of a Flattening layer provided by PyTorch. Such layer, composed of 512 neurons, is connected to another linear layer with 10 neurons. Its output goes to an activation function (Softmax, in this case) so that its elements could be handled as class probabilities.

We used some specific methods from the PyTorch library to implement the 3 layers of each convolution block, namely:

1. 3×3 Kernel convolution over the input signal (convolutional layer)
2. 2×2 Kernel convolution for the Max Pooling
3. ReLU activation layer

Here is a representation of all the layers composing the Neural Network, as printed by our python script:

```
ConvNet(
 (layers): Sequential(
  (0): Conv2d(3, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (3): ReLU()
  (4): Conv2d(128, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (6): ReLU()
  (7): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (9): ReLU()
  (10): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (12): ReLU()
  (13): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (15): ReLU()
  (16): Flatten(start_dim=1, end_dim=-1)
  (17): Linear(in_features=512, out_features=10, bias=True)
 )
)
```

Figure 1: Network architecture printout.

The values we have chosen for the padding and stride of the convolutional layers are both equal to 1, to make sure that the matrices associated with each image have their dimensions preserved when applying the convolution. For the Max Pooling, we provided a stride value of 2, since our objective was to halve the input spatial resolution. After creating and training the model, we collected all the metrics achieved (epoch by epoch) in the following table:

Epochs	Training Loss	Validation Accuracy	Epochs	Training Loss	Validation Accuracy
1	1.4041	52.2%	26	0.0363	78.0%
2	1.0399	65.3%	27	0.0357	77.6%
3	0.8503	66.9%	28	0.0478	79.0%
4	0.7271	72.9%	29	0.0868	78.2%
5	0.5999	73.7%	30	0.0198	78.5%
6	0.6385	76.1%	31	0.0255	78.8%
7	0.6413	75.9%	32	0.0242	78.7%
8	0.4550	76.2%	33	0.0216	79.6%
9	0.3991	78.6%	34	0.0173	78.7%
10	0.3195	79.1%	35	0.0151	78.7%
11	0.2606	78.0%	36	0.0208	78.4%
12	0.1962	78.2%	37	0.0212	78.3%
13	0.2424	77.7%	38	0.0181	79.1%
14	0.1907	78.8%	39	0.0183	78.1%
15	0.1623	77.5%	40	0.0138	77.8%
16	0.1623	77.5%	41	0.0175	78.7%
17	0.1008	76.8%	42	0.0125	79.1%
18	0.1194	76.9%	43	0.0271	78.6%
19	0.0887	76.5%	44	0.022	78.3%
20	0.0990	78.0%	45	0.0123	78.4%
21	0.1293	77.3%	46	0.0094	78.1%
22	0.0394	79.5%	47	0.0151	78.4%
23	0.0341	78.3%	48	0.0152	78.5%
24	0.0416	78.9%	49	0.0121	78.6%
25	0.0288	77.9%	50	0.0147	77.8%

Figure 2: Base model, trained up to 50 epochs

Below we also provide the loss-accuracy curves, as they were produced by the model, for a better consultation:

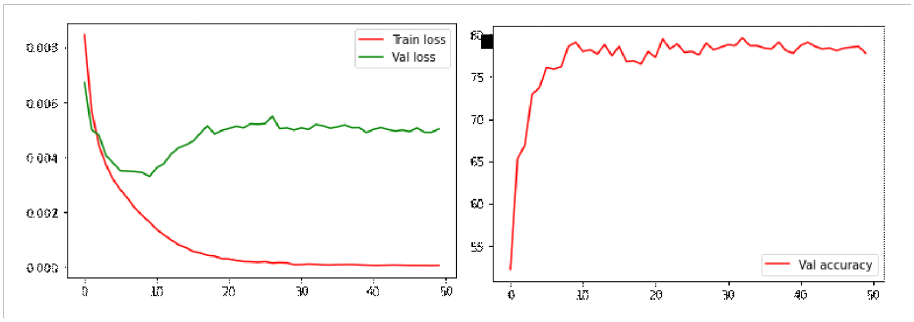


Figure 3: Loss-Accuracy curves for the base model.

Question 1.b - Model Size

As stated in the objective, the goal here is to calculate the size of our model, that corresponds to the number of trainable parameters. We implemented the requested function using the methods provided by PyTorch, so that when running `printModelSize` we get:

```
Model size | Number of parameters:  7682826
```

In other words, we have more than 7.5 millions trainable parameters.

Question 1.c - Filter visualization

From the assignment specifications we know that the first convolutional layer has to make use of 128 filters, each one of shape 3×3 due to each image having 3 input channels - which is actually a normalized RGB space. The weights for the filters were also generated by sampling from a Gaussian distribution. Below is a visualization of the filters that are used at the beginning of the training process.

Figure 4: (Pre-training) Filters in the first convolutional layer.

And here are the filters after finishing the training:

Figure 5: (Post-training) Filters in the first convolutional layer.

Now, to answer the question: do we see any patterns?

The main difference between the various filters we had after training was that some of them had bright colors, while other tended towards grey tints. That does not say much about their behaviour due to each kernel element being encoded in another 3 "color" dimensions; furthermore, we are dealing with very small filters indeed.

What was most interesting was the behaviour of the filters when performing a convolution: the apparently bright filters seemed to be recognizing edges on only some tints. Those greyer filters (even more in general, those with washed out colours) were blurring the images while also shifting the colors towards a specific tint.

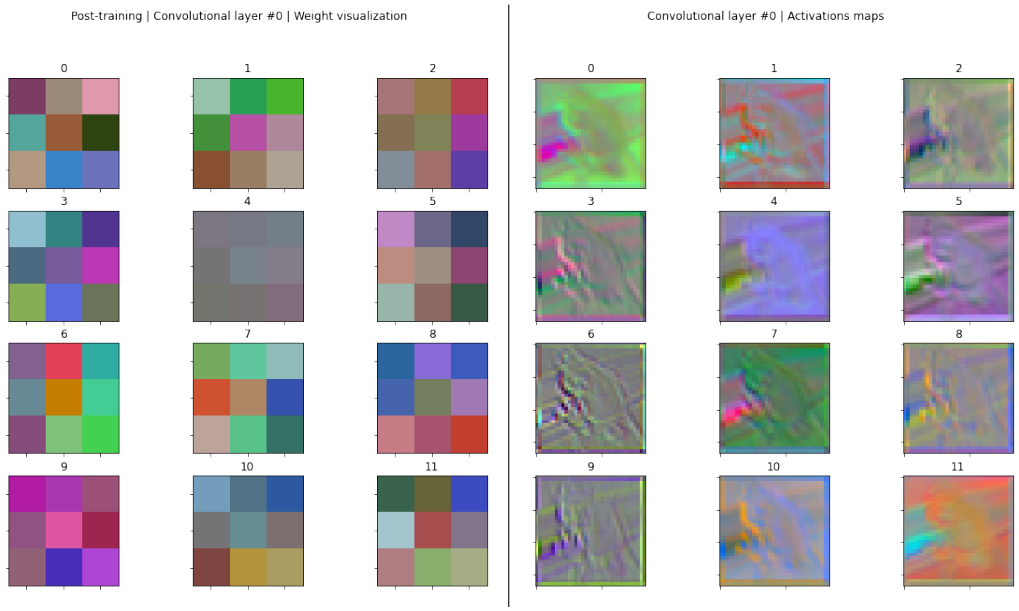


Figure 6: Learned filter effects on image.

To experiment a bit more, we also created a new script `ex3_Visualization_Load_From_Disk.ipynb`. There you can see the shift of tints by the greyer filters, and the unpredictable effects of the brighter ones!

Question 2

In this part of the exercise, we will try to implement a few techniques used to improve performance and classification accuracy.

Question 2.a - Batch Normalization

One problem we always face relates to limited computing resources. Luckily, for this exercise we are required to implement Batch Normalization! According to the paper from Ioffe & Szegedy, Batch Normalization should help us achieve better results while also speeding up computation. Surely that sounds interesting.
First let us redefine the batch normalization formula for a given mini-batch $\mathcal{B}$. First off, we need the batch mean and variance, respectively $\mu_{\mathcal{B}}$ and $\sigma^2_{\mathcal{B}}$. Let $\mathcal{B} = \{x_1, x_2, ..., x_m\}$, then:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$$

Now we can actually start normalizing the batch itself.

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$$

$$y_i = \gamma\hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$$

Side note, the epsilon is an arbitrary chosen parameter added for numerical stability. The equations were reported here from the paper.

The idea is to first calculate the batch mean and the batch variance. We then use them to perform normalization by modifying the values in the minibatch $\mathcal{B}$. As a result, $\mathcal{B}$ will have mean 0 and variance 1. That of course might not be representative of the actual distribution, so we added two **trainable** parameters γ and β to add scaling and shifting respectively.

Here is a table of the trials we did:

Epochs	Training Loss	Validation Accuracy	Epochs	Training Loss	Validation Accuracy
1	1.1779	50.9%	26	0.0254	81.6%
2	0.8602	66.7%	27	0.0600	80.2%
3	0.8239	64.3%	28	0.0406	81.0%
4	0.7859	73.8%	29	0.0269	82.3%
5	0.6373	71.0%	30	0.0150	82.8%
6	0.4691	79.6%	31	0.0169	83.1%
7	0.4669	75.1%	32	0.0313	82.5%
8	0.3768	78.3%	33	0.0090	83.8%
9	0.5503	81.7%	34	0.0046	84.2%
10	0.2654	80.4%	35	0.0021	83.9%
11	0.2835	81.6%	36	0.0058	83.5%
12	0.2138	80.7%	37	0.0034	84.4%
13	0.2433	83.8%	38	0.0039	84.5%
14	0.1819	79.4%	39	0.0063	84.4%
15	0.1247	81.8%	40	0.0341	75.5%
16	0.1260	80.1%	41	0.1169	82.2%
17	0.0586	82.8%	42	0.0107	83.5%
18	0.0846	81.4%	43	0.0041	83.9%
19	0.0586	82.8%	44	0.0033	84.6%
20	0.0846	81.4%	45	0.0030	84.2%
21	0.0648	81.4%	46	0.0029	84.1%
22	0.0332	81.1%	47	0.0022	84.3%
23	0.0735	81.1%	48	0.0029	84.3%
24	0.0377	82.7%	49	0.0027	84.6%
25	0.0417	81.2%	50	0.0023	84.0%

Figure 7: Batch normalization accuracies throughout epochs.

With the Batch Normalization implemented we were able to outperform the previous model after just 6 epochs (before, we needed 10 epochs to reach the same level of validation accuracy and training loss). Additionally, after 13 epochs, we scored a validation accuracy of 83.8%, an accuracy we could not reach before.

Nonetheless, at the end of the training process, the model still presented some overfitting.

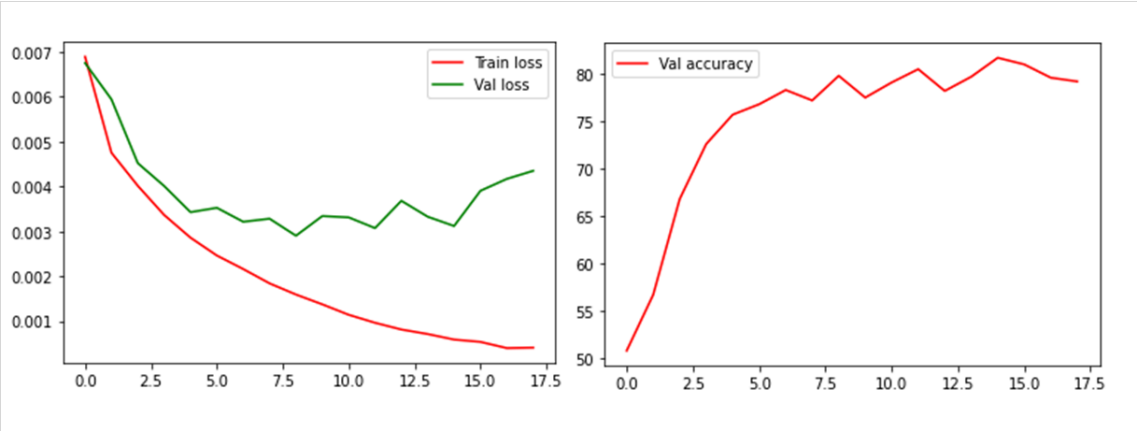
Question 2.b - Early stopping

A lesson we learned so far is that it is pointless to keep training a model over time, if after a number of consecutive epochs such model is not showing any sensible improvements in terms of validation accuracy. Moreover, by letting the model train under such circumstances, we might end up with an overfitted model (that means, the training loss converges to zero while the validation accuracy is not improving significantly, if not decreasing).
It is requested by the assignment to handle this kind of situation by implementing an Early Stopping mechanism.

The code we implemented is structured so that if after a given number of consecutive epochs we do not gain much in terms of validation accuracy, we stop the training process. Such a number of epochs is passed as a parameter called *PATIENCE*. The value we chose for *PATIENCE* is 4, increased to 8 for any number of epochs greater than 10. The metrics returned by the model after implementing the Early Stopping are summarized below:

Epochs	Training Loss	Validation Accuracy
1	1.2037	50.8%
2	0.8463	56.7%
3	0.7989	66.8%
4	0.5719	72.6%
5	0.5018	75.7%
6	0.5436	76.8%
7	0.4020	78.3%
8	0.4277	77.2%
9	0.4072	79.8%
10	0.3592	77.5%
11	0.3537	79.1%
12	0.2451	80.5%
13	0.2256	78.2%
14	0.1292	79.7%
15	0.1302	81.7%
16	0.1631	81.0%
17	0.1022	79.6%
18	0.1308	79.2%
19	x	x
20	x	x
21	x	x
22	x	x
23	x	x
24	x	x
25	x	x

Early stopping - Performance



Early Stopping - Loss-Accuracy Curves

Question 2.b - Bonus

To get a more in depth look at the performance differences between an early stopped model against a model trained on a higher number of epochs, we performed an additional experiment.
We tried to see, across different numbers of training epochs (and other hyperparameters), how the performance of a model would change. Therefore, we implemented a GridSearch over an arbitrary space of hyperparameters.

First off, let us take a look at the possible values that the parameters can get during training:

- Epochs: $[5, 10, 30, 50]$
- Batch size: $[200, 500]$
- Learning rate: $[0.001, 0.002, 0.01]$
- Learning rate decay: $[0.9, 0.95, 0.99]$
- Regularization parameter: $[0.001, 0.005]$

The idea here is to explore all the possible combinations of hyperparameters and see which one performs better. Side note, the results are partially reproducible: the initialization of the model's parameters, being random, influences the model's overall performance.

Below is a sample of the best models we trained, you will find more in the file `ex3_convnet_gridsearch_results_complete.csv`.

Epochs	Batch Size	LR	LR Decay	Reg	Best Model Validation Accuracy	Early Stopped Validation Accuracy	Best Model Test Accuracy	Early Stopped Test Accuracy
30	200	0.001	0.9	0.005	85.6	84.2	86.7	86.7
30	400	0.001	0.9	0.001	87.5	86.2	86.3	86.3
30	200	0.001	0.9	0.001	86.4	85.3	85.9	85.9
50	400	0.001	0.9	0.001	87.6	86.1	85.83	85.83
30	400	0.001	0.95	0.001	85.9	-	85.76	-
30	200	0.001	0.95	0.001	86.7	-	85.3	-
30	400	0.002	0.9	0.001	86.7	85.4	85.3	85.3
30	400	0.001	0.95	0.005	85.6	-	85.16	-
50	400	0.002	0.9	0.001	87.6	86.1	85.08	85.08
30	400	0.001	0.9	0.005	87	84.6	84.91	84.91

From what we can see, the best models required a minimum of 30 epochs to train until they were able to make somewhat reliable predictions, though some less performing models only needed 10 epochs to train. Funnily enough, some of the worst models had 50 epochs to train, but although patience was set at 8, they were luckily early stopped to proceed with possibly better performing models. As stated before, initialization of the parameters is important.
One thing we can notice though, is how crucial the Learning Rate Decay is: the "softer values" of 0.95 to 0.99 (so almost no decay) did bring the model to perform worse, while a stronger decay would generally improve performance.

Question 3

Question 3.a - Data Augmentation

For this task we are going to do a little bit of tests with data augmentation. Luckily PyTorch has its own implementation of various techniques, that seamlessly get applied when defining the dataset.

We experimented a bit with various ones, namely rotation, flipping the image, changing the perspective of the image (artificially of course) and jittering, which is another way of saying manipulation of the pixels as to apply changes in brightness, contrast and more.

The first thing to say is: we did not expect such good results. We ran a few tests when solving the exercise, and of course we were not sure that data augmentation was actually helping. Later on, as we tested the model more, we found values for the different hyperparameters that got us impressive values for accuracy. We have also tried some combinations of hyperparameters on jittering, however they did not improve the validation and test accuracy of the model.
Here are some of them:

Transformations (in sequence)	Best Model Validation Accuracy	Best Model Test Accuracy
Horizontal Flip ($p = 0.3$), Random Rotation (degrees $\in [-20, 20]$)	86.4	88.7
Horizontal Flip ($p = 0.6$), Random Rotation(degrees $\in [-20.0, 20.0]$)	85.5	88.4
Horizontal Flip($p = 0.6$)	88.9	87.7
Horizontal Flip ($p = 0.3$)	88.6	87.6
Random Rotation (degrees $\in [-20.0, 20.0]$)	84.3	87.3
Random Perspective($p = 1.0$)	87.2	86.3

We noticed that some of the transformations were not useful at all, getting us accuracy values lower that 70%. This again is to show that data augmentation, if necessary, will be a trial-and-error process: some hyperparameters will bring better results than others, as we have discovered with the gridsearch.
To run more experiments, please refer to our script `ex3_Data_Augmentation_Tests.ipynb`.

Question 3.b - Dropout

As suggested, we have disabled data augmentation in order to clearly visualize all the different performances returned by the model when using each p value for the dropout layer. Since we incremented by 0.1 at each iteration, we ended up having 9 different values for p, going from 0.1 to 0.9. In the table below we show the validation accuracies we got for each epoch, after using all of the chosen dropout values.

Dropout Level	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Epoch 1	46.6%	43.3%	46.3%	34.3%	34.7%	39.7%	26.7%	24.1%	18.2%
Epoch 2	58.4%	49.3%	48.9%	39.6%	54.5%	43.3%	42.4%	25.7%	21.9%
Epoch 3	68.0%	68.1%	58.9%	57.8%	56.0%	57.2%	50.1%	28.3%	20.5%
Epoch 4	69.1%	63.1%	65.3%	69.4%	66.8%	66.0%	53.7%	46.1%	35.6%
Epoch 5	71.7%	70.0%	74.1%	74.9%	72.7%	69.0%	66.8%	49.7%	38.1%
Epoch 6	78.8%	71.4%	75.2%	74.6%	68.8%	69.0%	66.5%	58.7%	48.1%
Epoch 7	78.0%	77.1%	78.1%	76.0%	76.1%	73.8%	72.3%	66.2%	50.2%
Epoch 8	76.9%	75.2%	78.9%	77.1%	77.4%	75.0%	72.9%	63.2%	54.0%
Epoch 9	80.2%	80.5%	81.5%	76.3%	75.7%	76.5%	69.7%	68.7%	51.3%
Epoch 10	81.2%	80.7%	77.9%	77.0%	79.2%	74.4%	73.5%	66.1%	59.7%
Epoch 11	80.0%	80.3%	82.5%	80.5%	79.1%	76.2%	73.4%	68.8%	62.3%
Epoch 12	77.8%	78.2%	78.9%	82.0%	80.5%	80.0%	73.5%	68.1%	59.1%
Epoch 13	79.3%	81.3%	80.5%	80.0%	80.9%	75.7%	76.6%	66.4%	63.0%
Epoch 14	81.2%	82.3%	79.3%	81.9%	79.3%	78.6%	74.7%	72.4%	56.0%
Epoch 15	80.7%	79.5%	79.1%	81.3%	80.3%	78.3%	79.3%	73.5%	X
Epoch 16	80.2%	81.2%	79.6%	80.1%	82.4%	81.4%	78.9%	73.5%	X
Epoch 17	81.5%	82.4%	80.7%	81.3%	80.7%	80.1%	81.0%	74.3%	X
Epoch 18	81.5%	82.5%	81.7%	82.3%	81.1%	80.3%	78.4%	75.5%	X
Epoch 19	80.9%	80.0%	82.6%	80.9%	82.2%	81.5%	80.5%	76.8%	X
Epoch 20	83.3%	81.5%	81.1%	81.2%	80.8%	81.2%	79.9%	76.5%	X
Epoch 21	79.9%	81.9%	82.3%	79.6%	82.3%	80.9%	79.8%	76.9%	X
Epoch 22	X	82.0%	83.2%	X	83.7%	80.0%	82.5%	76.6%	X
Epoch 23	X	82.5%	79.7%	X	83.0%	X	77.3%	77.4%	X
Epoch 24	X	80.7%	X	X	82.1%	X	X	78.4%	X
Epoch 25	X	X	X	X	82.3%	X	X	77.3%	X
Epoch 26	X	X	X	X	82.3%	X	X	78.2%	X
Epoch 27	X	X	X	X	83.0%	X	X	76.4%	X
Epoch 28	X	X	X	X	84.1%	X	X	X	X
Epoch 29	X	X	X	X	84.2%	X	X	X	X
Epoch 30	X	X	X	X	83.2%	X	X	X	X
Epoch 31	X	X	X	X	83.2%	X	X	X	X
Epoch 32	X	X	X	X	83.2%	X	X	X	X
Test Accuracy	79.7%	82.3%	81.2%	81.1%	83.4%	79.8%	77.9%	77.1%	57.1%

In the above table, the red values indicate the epoch in which the model accuracy started to decrease, while the green values are associated with the best validation accuracies (corresponding to the best model saved). We immediately noticed that each dropout level has triggered the early stopping mechanism at a different epoch. For example, with a dropout level of 0.1,

the model has stopped after 21 epochs. For this reason, we have placed a (x) in all of the subsequent cells in the provided table. As revealed by the following chart, the best result was achieved when setting p equal to 0.5.

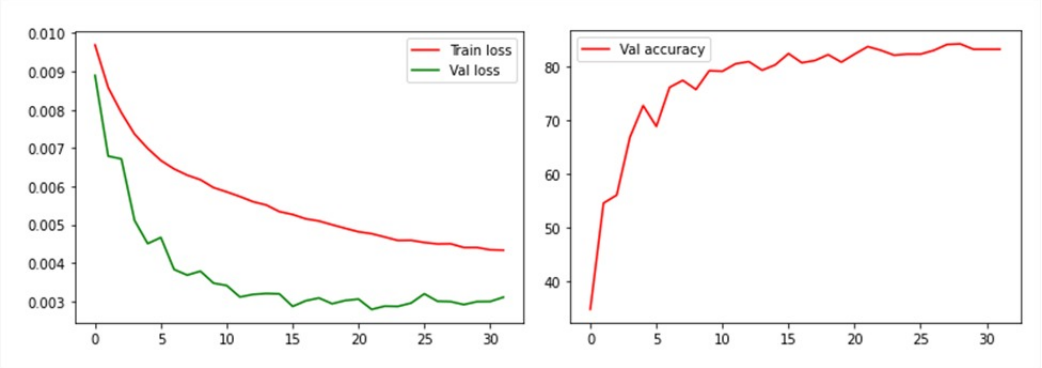


Figure 8: Loss-accuracy curves throughout epochs (Dropout).

This comes with no surprise: we already know that with low values of p the model might overfit, while with high values of p we might decrease the model capacity due to an excessive amount of regularization.

In our specific case, a perfectly balanced value for p, that is exactly in between the minimum and maximum thresholds, turned out to be the best possible choice.

For the sake of completeness, we also provide, attached to our submission, an excel file with multiple sheets, each describing in detail all the results obtained at each epoch for all the given dropout values.

Question 4

Question 4.a - Testing `VGG11-BN`

As stated in the exercise, this time we will be performing a bit of transfer learning. We will be working with a known neural network, namely VGG11-BN. We cannot use VGG11-BN directly though, as although being in a similar problem domain, it does not apply to our specific goal. We will need a few adjustments: at the end of the original VGG11-BN there will be two additional linear fully connected layers, so the structure becomes as illustrated below:

```
VggModel(
  (features): Sequential(
    (0): Sequential(
      (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (6): ReLU(inplace=True)
      (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (8): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (10): ReLU(inplace=True)
      (11): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (12): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (13): ReLU(inplace=True)
      (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (15): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (16): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (17): ReLU(inplace=True)
      (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (19): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (20): ReLU(inplace=True)
      (21): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (22): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (23): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (24): ReLU(inplace=True)
      (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (26): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (27): ReLU(inplace=True)
      (28): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
  )
  (classifier): Sequential(
    (0): Linear(in_features=512, out_features=512, bias=True)
    (1): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

We will not then train the entire network (i.e. just keeping the architecture and hoping it works), but we will rather do something more efficient: the weights of the layers belonging to the VGG11-BN will be "frozen" using the function `set_parameter_requires_grad` onto the VGG layers. The rest of the layers, to be specific those we just added, will be trained as usual.

As expected, the results are quite disappointing, we just achieved a mere 60%-65% validation accuracy. More plots below:

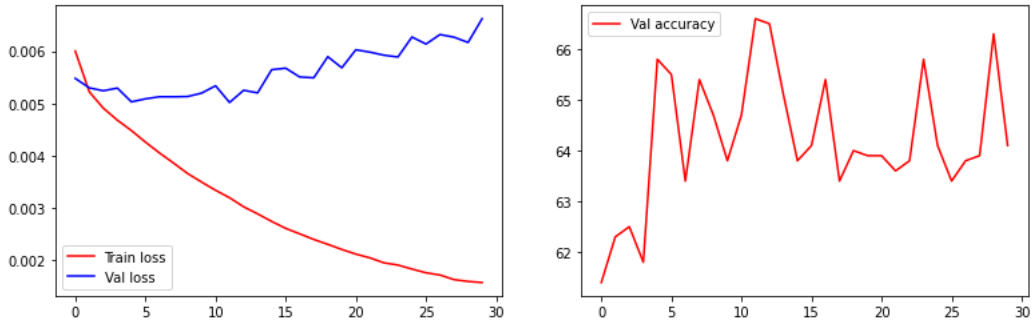


Figure 9: Loss-Accuracy curves.

Question 4.b - Improving our newer model!

We saw how the domain shift from the Imagenet dataset to the CIFAR10 dataset decreased the overall performance of our previous model, so as suggested by the exercise, we will now fine tune the VGG to better apply it to our problem.

Firstly, we "unfroze" the weights in the VGG11, then re-trained it: basically the weights from VGG11 were kept and modified during the new training iteration, while the additional layers were trained as usual.

It was also required to compare our model with a baseline model with the same architecture but trained entirely from scratch. Here are the results:

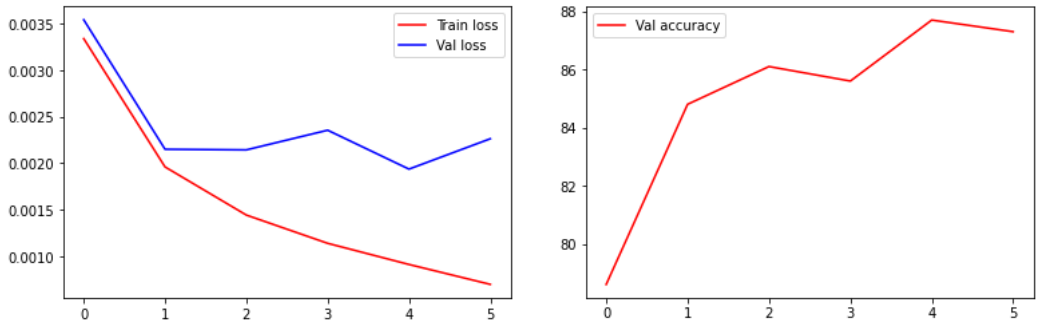


Figure 10: Loss-Accuracy curves for the fine tuned model.

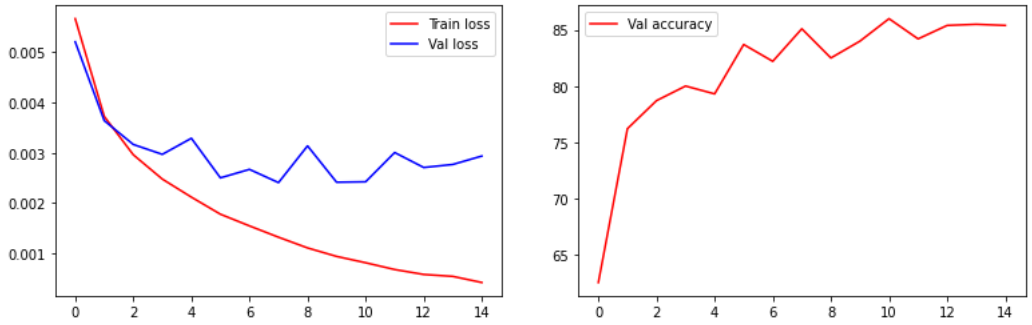


Figure 11: Loss-Accuracy curves for the model trained from scratch.

As we can see, the results we now achieved significantly improved if compared with those obtained in **Question 4.a**, although with the fine-tuned

model (as opposed to the one trained from scratch), we did require way fewer epochs to achieve good results: around 85% accuracy, up to 88% on the validation set. For the Test results accuracy we had 84% and 80% for the Model 1 (with Pre-trained weights of Image net and Custom Classifier) and Model 2 (without Pre-trained weights of Image-Net and Custom Classifier) respectively.