Sapienza Università di Roma

ADVANCED MACHINE LEARNING

REPORT HW 2: Convolutional Networks

Students

Mert Yildiz
1951070
Ali Reza Seifi Mojaddar
1900547
Gianmarco Ursini
1635956
Mohammadreza Mowlai
1917906

Professor

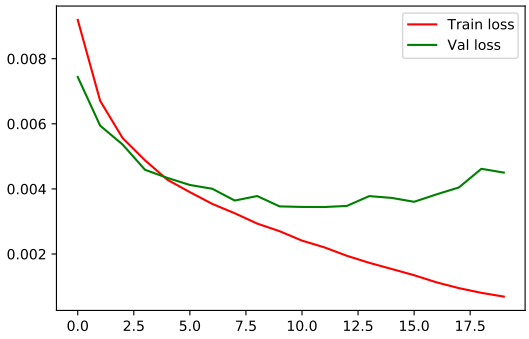
Prof. Fabio Galasso

Accademic Year 2021/2022

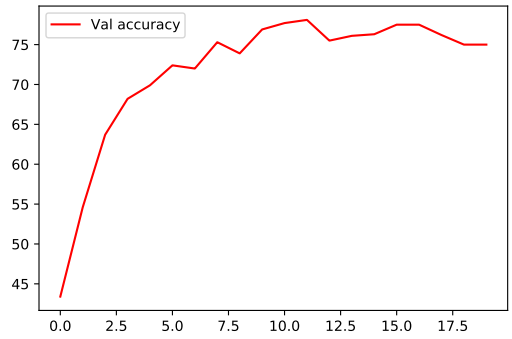
Question 1: Implement a Convolutional Neural Network

Report training and validation loss curves and validation accuracy.

Below we report the requested training and validation loss curves and validation percentage accuracy trend. As we can clearly see, **overfitting** is occurring: the validation loss reaches a minimum around epoch 11 (a maximum of the validation accuracy can be spotted at the same epoch) and then suddenly start to increase as the training keeps going.



(a) Training and validation losses.



(b) Validation accuracy.

Report the number of trainable parameters in the model implemented.

After implementing our own `PrintModelSize()` function, its output (i.e. the number of trainable parameters in our model) turns out to be equal to `7678474`.

Let's try to compute theoretically the number of parameters in our model. As clear, both the `ReLu()` and the `MaxPool2d()` layers contain no parameters (all they have to do is execute a deterministic task), while the number of parameters N_{par}^{Conv} in a convolutional layer is given by the following expression (being K_{size} the kernel filter size adopted in the convolutional layer, N_{ch}^{in} the number of channels in the input of the convolutional layer, N_{ch}^{out} the number of channels in the output of the convolutional layer):

$$N_{par}^{Conv} = (K_{size} \cdot K_{size} \cdot N_{ch}^{in} + 1) \cdot N_{ch}^{out} \tag{1}$$

Note that the additive unit in the parenthesis recall the bias parameter we are adding to each of the convolutional layer's filter used.

The number of parameters N_{par}^{FC} in a fully connected layer will be given instead by the following (being N_{in} the number of neurons in the input layer, N_{out} the number of neurons in the output layer):

$$N_{par}^{FC} = N_{out} \cdot (N_{in} + 1) \tag{2}$$

Using (1) and (2) we obtain:

- $N_{par}^{Conv_1} = 3584$;
- $N_{par}^{Conv_2} = 590336$;
- $N_{par}^{Conv_3} = 2359808$;
- $N_{par}^{Conv_4} = 2359808$;
- $N_{par}^{Conv_5} = 2359808$;
- $N_{par}^{FC} = 5130$;

Summing over the number of parameters of all the layers, we obtain `7678474`, that matches our `PrintModelSize()` output.

Visualize and compare 1^{st} convolutional layer's weights before and after training. Through the implementation of the `VisualizeFilter()` function, we are able to visualize 1^{st} convolutional layer weights. In particular, we aim to compare those weights before and after the training procedure. Giving a bird's-eye view to the filters before and after the training, we can see that in the first case weights look randomly distributed: indeed, looking at those weights we somehow experience a "white-noise" feeling (i.e. no particular shape or pattern stand out from the average behaviour of the filter set).

Figure 2: Weigths of 1^{st} convolutional layer before training.

Weights after training have a completely different appearance: it's indeed enough to briefly look at them to spot geometrical and color patterns. Some filters resemble to be activated by horizontal lines (e.g. filters n°25, 111, 112), oblique lines (e.g. filter n°54), vertical lines (e.g. filters n°27, 72, 128, 93), green patches (e.g. filters n°1, 34, 94), pink patches (e.g. filters n°47, 122), transition between different color patches (e.g. filters n° 86, 102,113, 7, 15).

Figure 3: Weigths of 1^{st} convolutional layer after training.

Question 2: Improve training of Convolutional Networks

Q2.a Batch Normalization

In this part we keep all the hyper-parameters the same with the Q1.a, and add batch normalization layer after the Conv. Net and before the max pooling.
Here you can find the differences between the previous model and the one with BN.

Figure 4: Validation accuracy without BN and filter visualization.

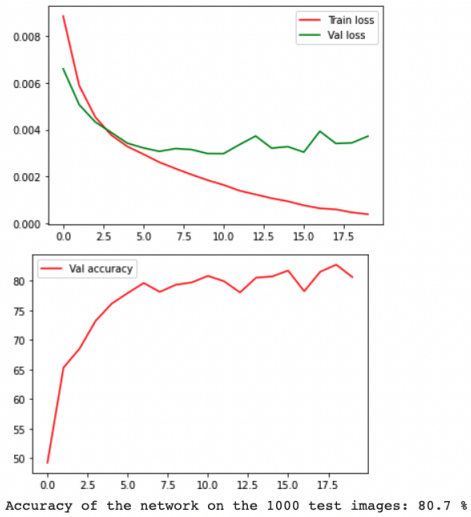


Figure 5: Test accuracy without BN = 80.7% (Training and Val. Loss, Val. accuracy).

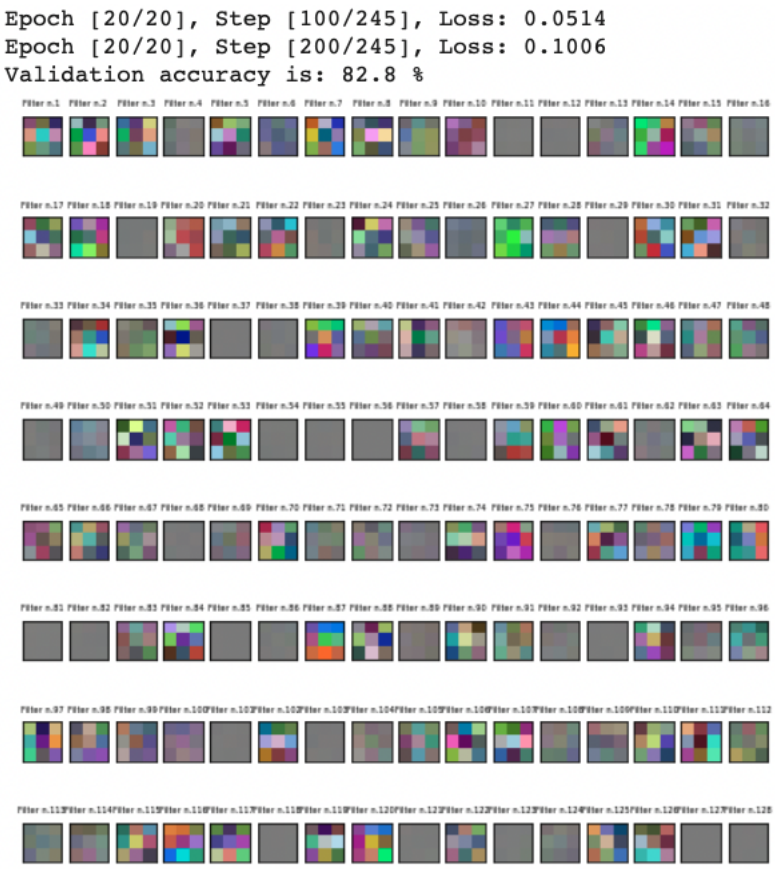


Figure 6: Validation accuracy with BN and filter visualization.

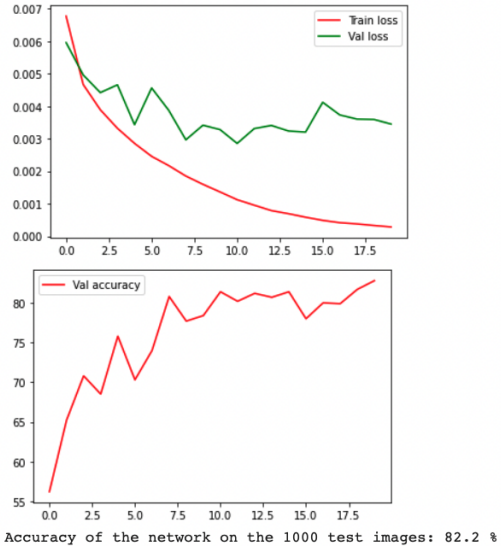


Figure 7: Test accuracy with BN = 82.2% (Training and Val. Loss, Val. accuracy).

As you can see in the pictures, the validation and test accuracy increased around 2% when we added the Batch Normalization layer.

Q2.b Early Stopping

In order to avoid overfitting, early stopping mechanism has been implemented with 50 epochs and different patience numbers. Here you can find the results:

Figure 8: Val. accuracy without BN, 50 ep. and patience of 5(and filter visualization).

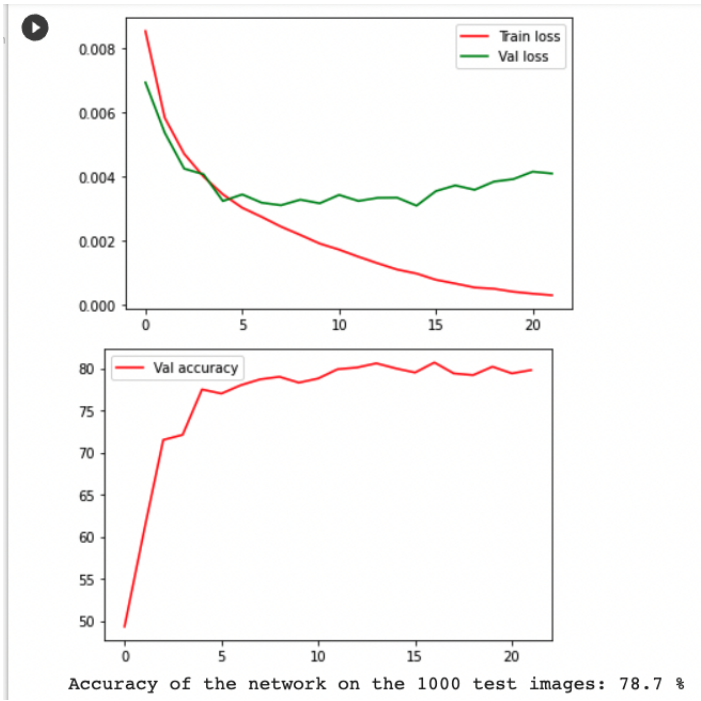


Figure 9: Test accuracy without BN, 50 ep. and patience of 5(Training and Val. Loss, Val. accuracy).

```
Extracting datasets/cifar-10-python.tar.gz to datasets/
ConvNet(
  (Convlayers): Sequential(
    (0): Conv2d(3, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (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): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (7): ReLU()
    (8): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (11): ReLU()
    (12): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (15): ReLU()
    (16): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (19): ReLU()
  )
  (Fconnected): Linear(in_features=512, out_features=10, bias=True)
)
Total number of parameters in the model is: 7682826
```

Figure 10: Model

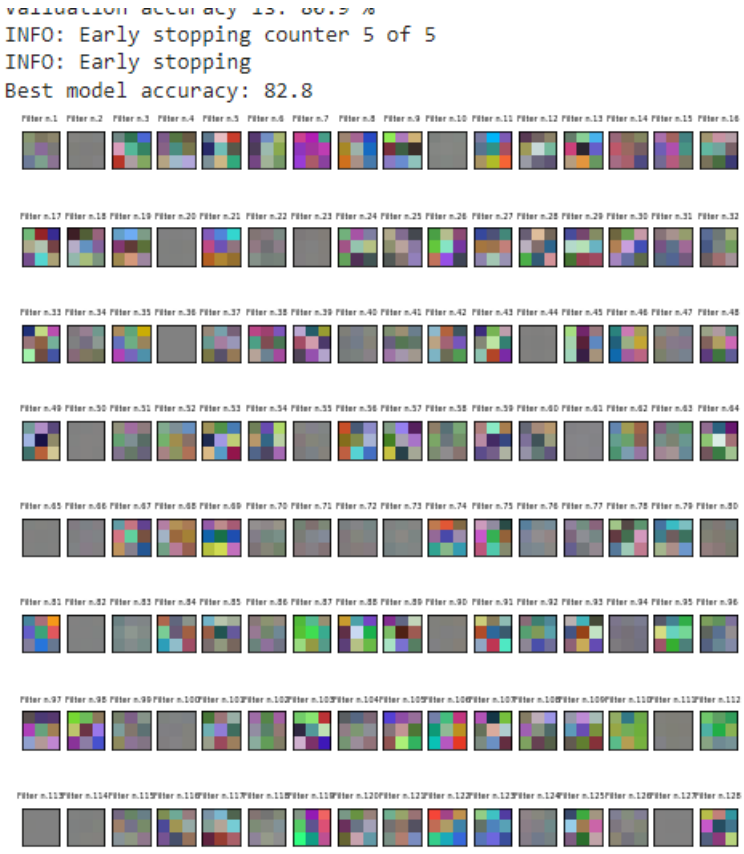


Figure 11: Val. accuracy with BN, 50 ep. and patience of 5 = 82.8% (and filter visualization).

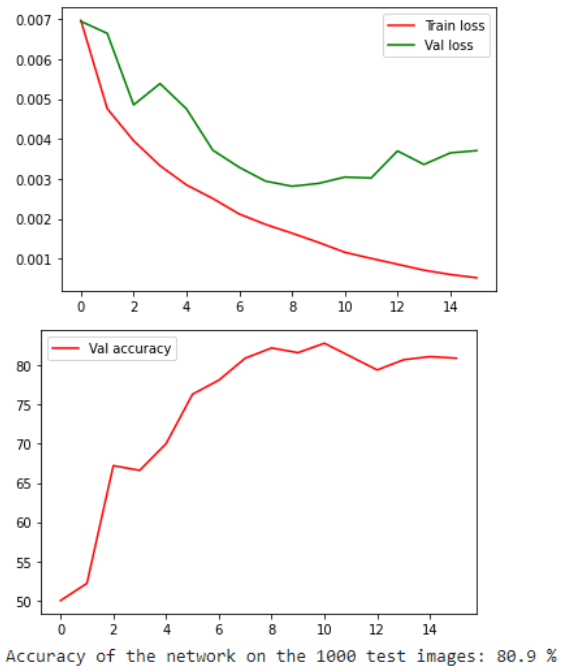


Figure 12: Test accuracy with BN, 50 ep. and patience of 5 = 80.9% (Training and Val. Loss, Val. accuracy).

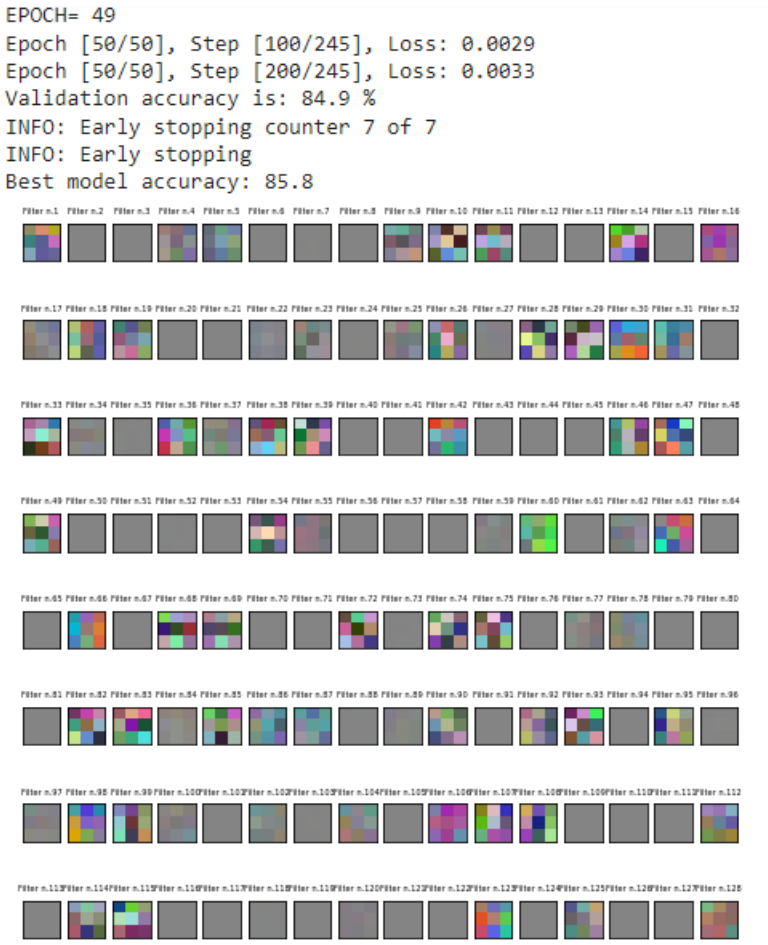


Figure 13: Val. accuracy with BN, 50 ep. and patience of 7 = 85.8% (and filter visualization).

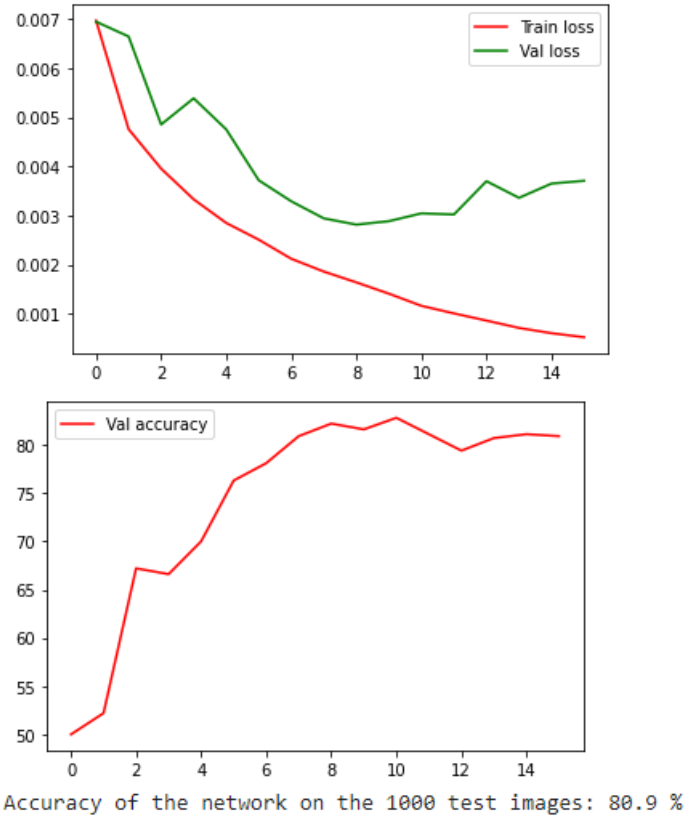


Figure 14: Test accuracy with BN, 50 ep. and patience of 7 = 80.9% (Training and Val. Loss, Val. accuracy).

In this part we observed a 80.7% val. accuracy with 50 epochs(without BN) where the val. accuracy with BN and the same epochs was 82.8%.
After trying some different patience numbers, the best model was with batch normalization, 7 patience number and 50 epochs, where we got an **85.8% val accuracy** and **80.9% test accuracy**.

Question 3: Implementing the feedforward model

Neural Networks are heavily reliant on big data to avoid overfitting which refers to the phenomenon when a network learns a function with very high variance to perfectly model the training data. And since many application domains do not have access to big data, augmentation techniques will be useful, which is a data-space solution to the problem of limited data. To build useful learning models, the validation error must continue to decrease with the training error. Data Augmentation is a powerful method of achieving this, which minimizes the distance between the training and validation set, as well as testing sets. In this part, we use Data wrapping augmentations which transform existing images such that their label is preserved. This includes augmentations such as geometric and color transformations which is the main focus in this question.

3.1. Analysis and Inference of different Augmentation methods

To have a better grasp of the impact of different augmentations, each of them will be applied and analyzed independently. And after that, the combined effects will be tested.

Flipping:

Horizontal flipping is applied with p=0.1 over 30 epochs and the results are as follows. The test accuracy is 81.7 and the maximum validation accuracy is 82.5 which is an improvement over the base model. This is an indication that Flipping might be a good augmentation technique over this Data set.

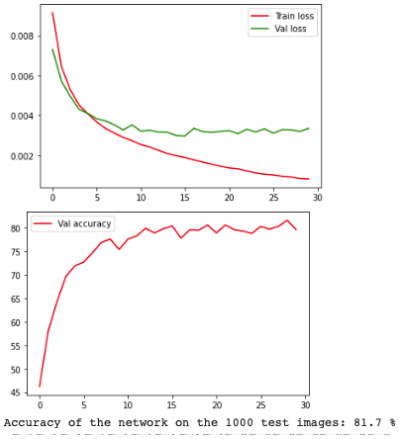


Figure 15: Horizontal flipping with p=0.1

In this setup Horizontal flipping with p=0.3 is applied. The test accuracy is 82.6 and the validation accuracy is 84.3, which indicates a little boost over the initial one.

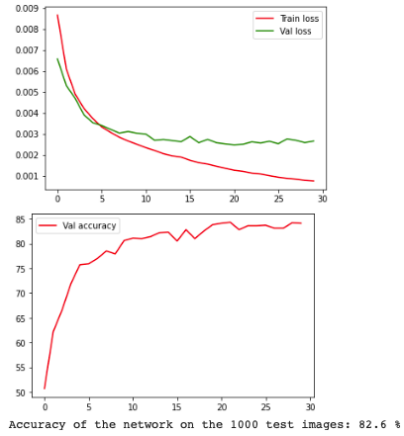


Figure 16: Horizontal flipping with p=0.3

In this setup Horizontal flipping with p=0.5 is applied. The test accuracy is 83 and the validation accuracy is 84.7

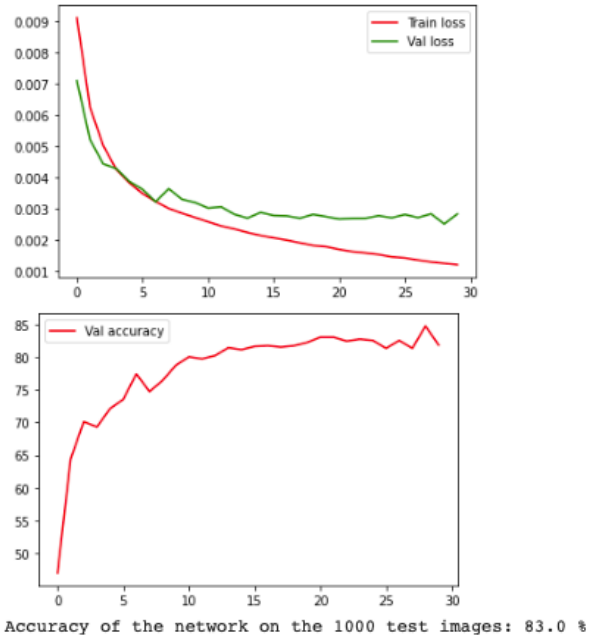


Figure 17: Horizontal flipping with p=0.5

In this setup Horizontal flipping with p=0.7 is applied. The test accuracy is 83.8 and the validation accuracy is 84.4

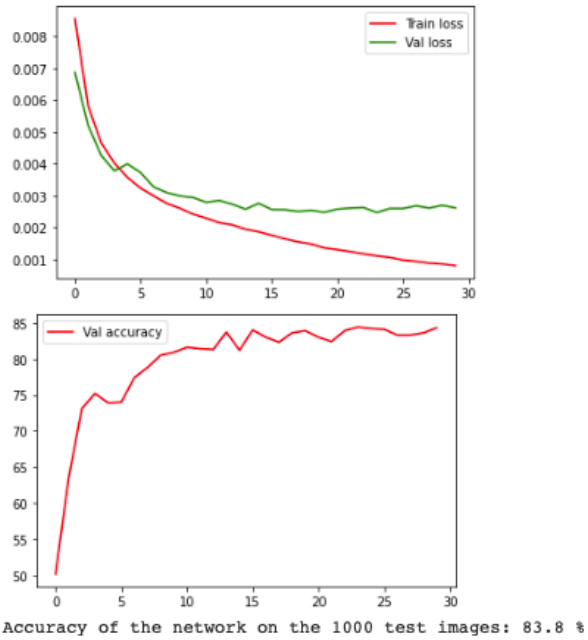


Figure 18: Horizontal flipping with p=0.7

Next Horizontal flipping with p=0.9 is applied. The test accuracy is 78.7 and the validation accuracy is 83.1 which shows the decrease in test accuracy.

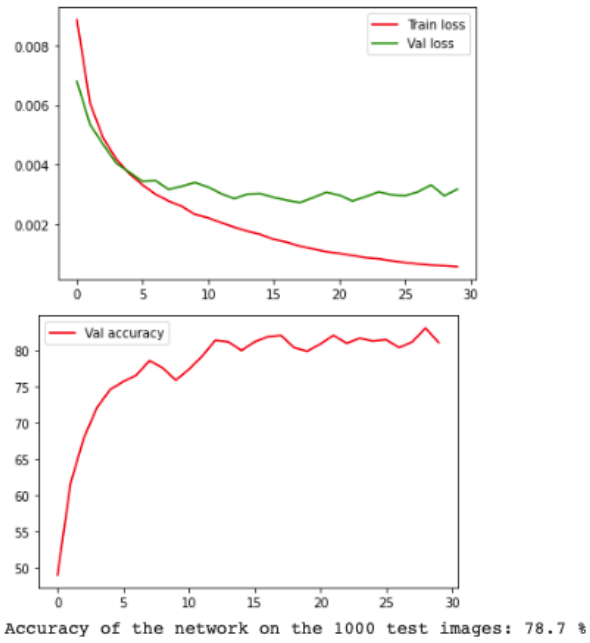


Figure 19: Horizontal flipping with p=0.9

To conclude, this augmentation is one of the easiest to implement and has proven useful in CIFAR-10 dataset.

Random-Crop:

Each image is padded by 4 pixels, and then a random crop of size 32 x 32 pixels is taken. Here the test accuracy is 82.9 and the max validation accuracy is 82.7 which shows that this augmentation method also looks promising.

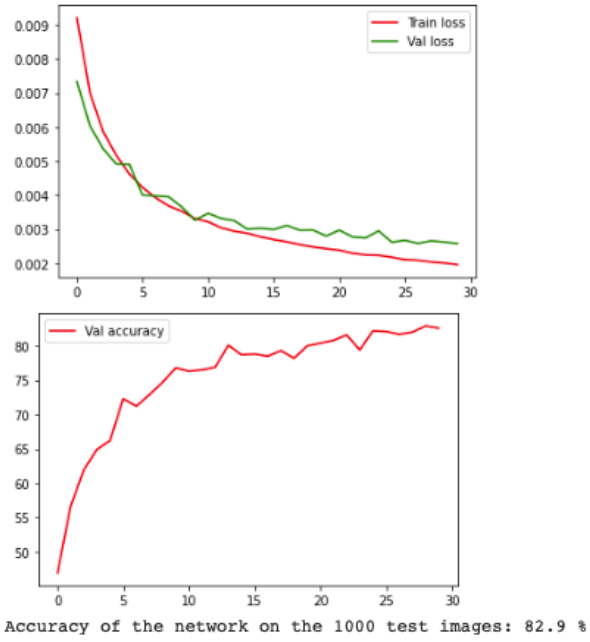


Figure 20: Random-Crop

Combination of random-crop and flipping:
Each image is padded by 4 pixels, and then a random crop of size 32 x 32 pixels is taken , and then the image is flipped horizontally with a 50 percent probability. Since the transformation will be applied randomly and dynamically each time a particular image is loaded, the model sees slightly different images in each epoch of training, which allows it to generalize better. Here the test accuracy is 84.6 and the maximum validation accuracy is 85, which is an improvement over the previous steps. and also it is obvious from the graph that the gap between training and validation has been reduced significantly.

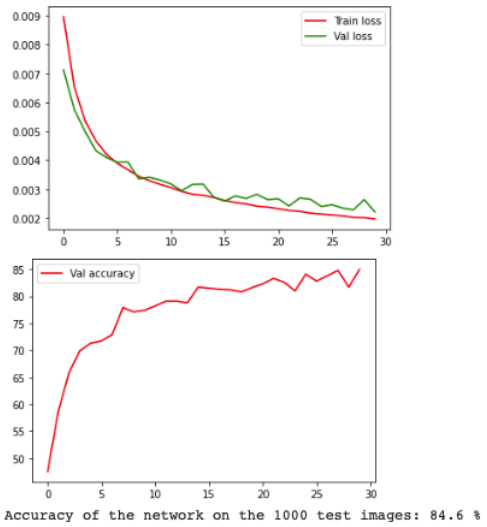


Figure 21: Random-Crop

Gray Scale:
Gray Scale augmentation will force the model to learn from even less information and can boosts the test accuracy. The result is as follows, The test accuracy is 79 and the maximum validation accuracy is 80.7, which shows a little boost in validation accuracy.

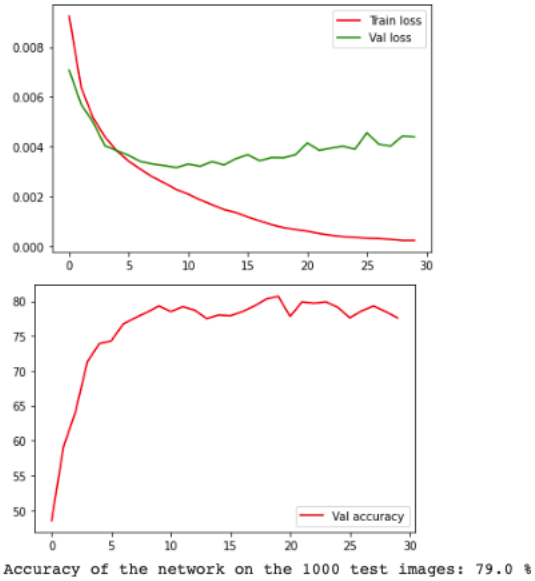


Figure 22: Gray Scale with p=0.2

Combination of RandomCrop, RandomHorizontalFlip and RandomGrayScale:
Mixture of Random Crop, Random Horizontal Flip and, Random Gray Scale is applied and the test accuracy is 81.6 and maximum validation accuracy is 82.

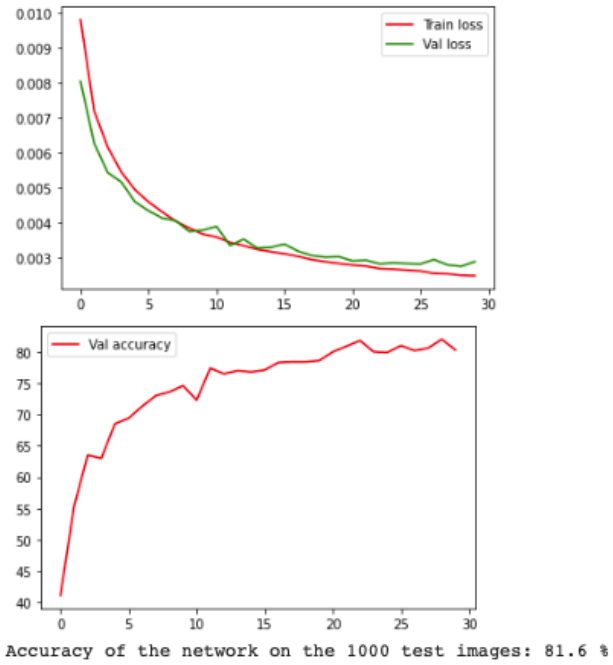


Figure 23: Combination of Random-Crop, RandomHorizontalFlip and RandomGrayScale

Color Jitter:
In this setup color jitter augmentation is applied with the probability of 0.4 and with brightness, contrast, saturation, hue parameters set to [0.4, 0.4, 0.4, 0.1]. The test accuracy is 81.1 and the maximum validation accuracy is 82.1

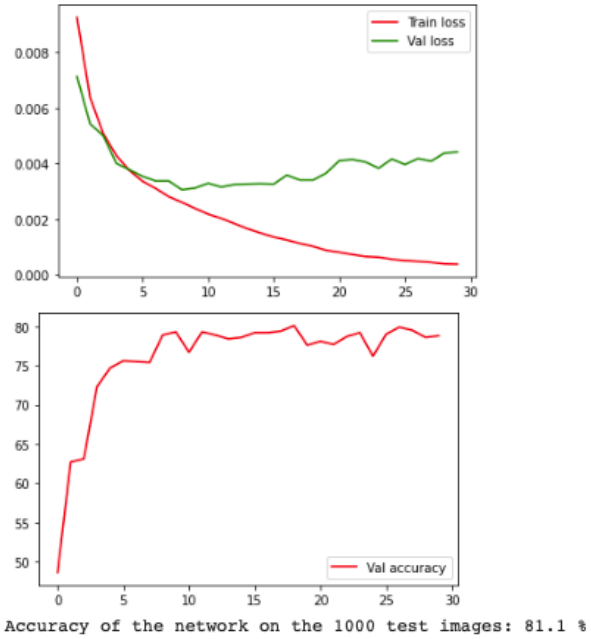


Figure 24: Color Jitter Augmentation

Random-Crop, Random Horizontal Flip, Random Gray Scale and Random Applying of Guassian Blur:
In this setup mixture of Random-Crop, Random Horizontal Flip, Random Gray Scale, and Random Applying of Gaussian Blur is applied. The test accuracy is 81.8 and, the maximum validation accuracy is 82.9

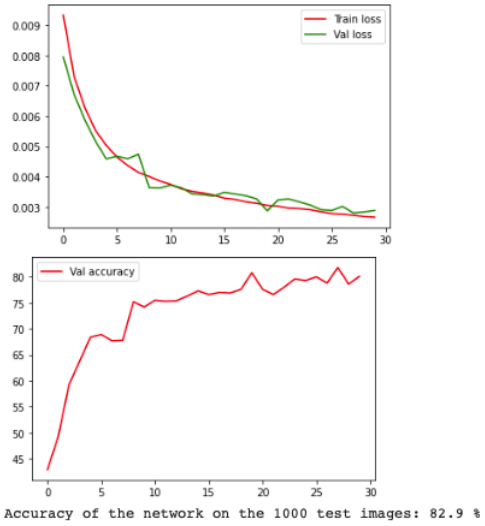


Figure 25: Color Jitter

Random-Crop, Random Horizontal Flip, Random Gray Scale and Random Applying of Guassian Blur plus color jitter:
In this setup mixture of Random-Crop, Random Horizontal Flip, Random Gray Scale, and Random Applying of Gaussian Blur plus color jitter is applied. The test accuracy is 83 and, the maximum validation accuracy is 84.9.

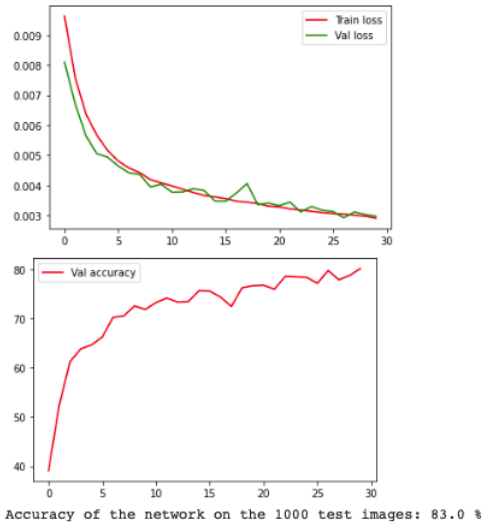
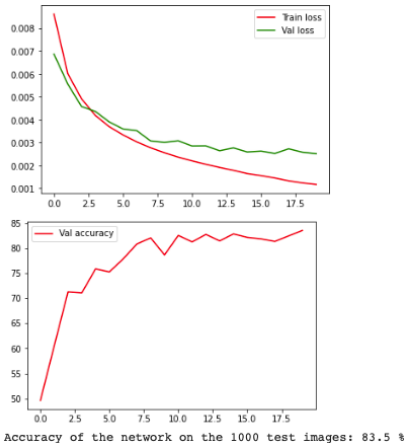


Figure 26: Color Jitter

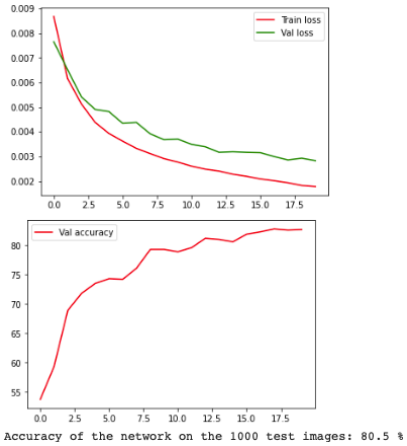
To conclude mixture of RandomCrop, RandomHorizontalFlip, Random GrayScale, and Random Applying of Gaussian Blur plus color jitter and combinations of random-crop and flipping were amongst the best augmentations tested in this exercise.

3.2 Analysis and Inference of Dropout

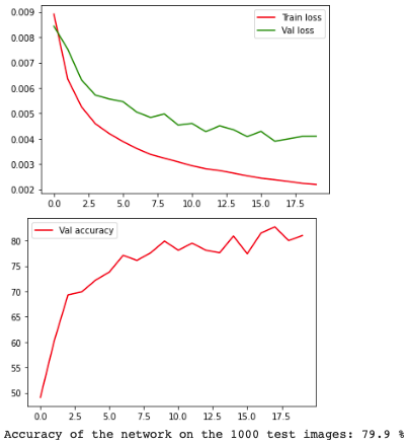
Dropout refers to ignoring units during the training phase of certain set of neurons which is chosen at random. At each training stage individual nodes are dropped out of the net with probability 1 - p and kept with probability p. And the overall aim is to prevent Over-fitting. Dropout forces a neural network to learn more robust features that are useful in conjunction with many different random subsets of the other neurons. Also higher numbers of iteration is needed for the network to converge, However it is quite obvious that the training time for each epoch is less. From the Below graphs we can conclude that with increasing the dropout, there is some increase in validation accuracy and decrease in loss however this trend starts to go down at some point. We can also see the best models are with dropouts 0.1, 0.2, and 0.3 however the gap between train and validation sets is the least for dropout 0.2 and the accuracy is almost the same with 0.1. For this 0.2 is a safe point to pick.



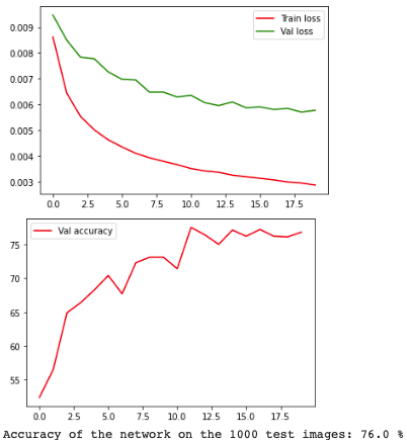
(a) Dropout set to 0.1



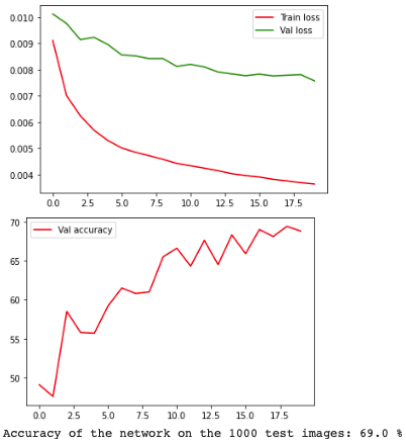
(b) Dropout set to 0.2



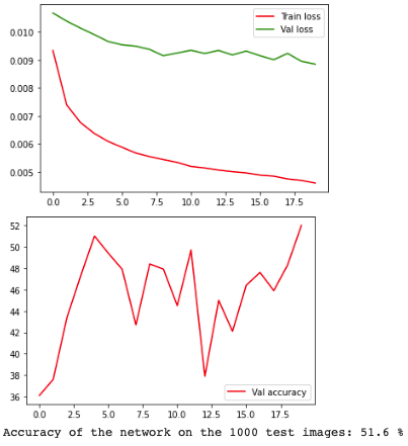
(c) Dropout set to 0.3



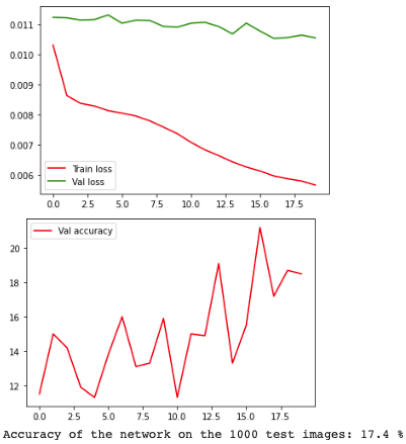
(a) Dropout set to 0.4



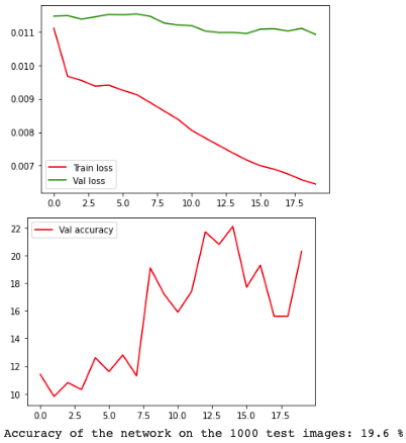
(b) Dropout set to 0.5



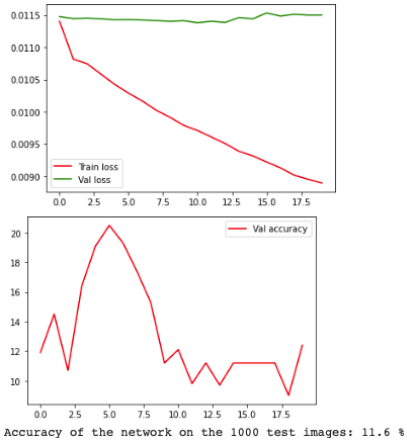
(c) Dropout set to 0.6



(a) Dropout set to 0.7



(b) Dropout set to 0.8



(c) Dropout set to 0.9

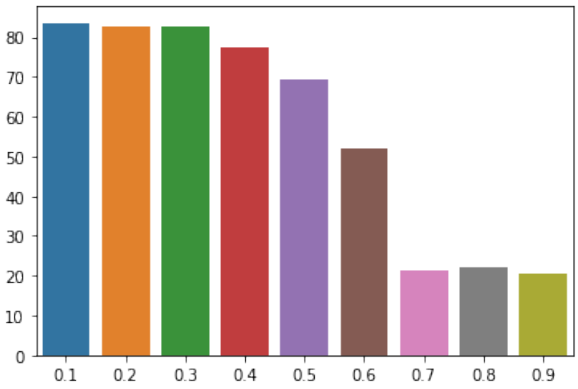


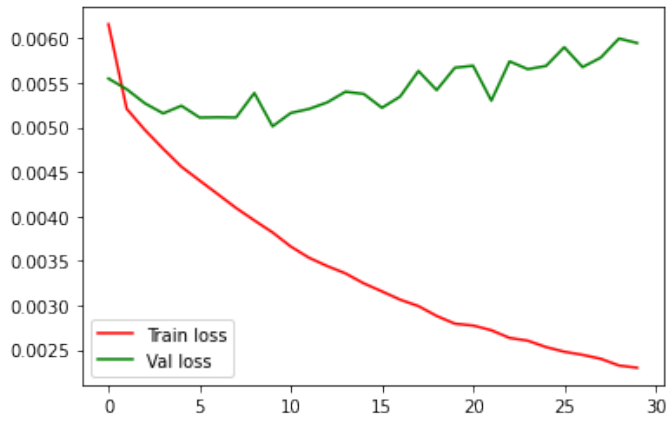
Figure 30: Validation accuracy over different Dropouts

Question 4: Implementing the feedforward model

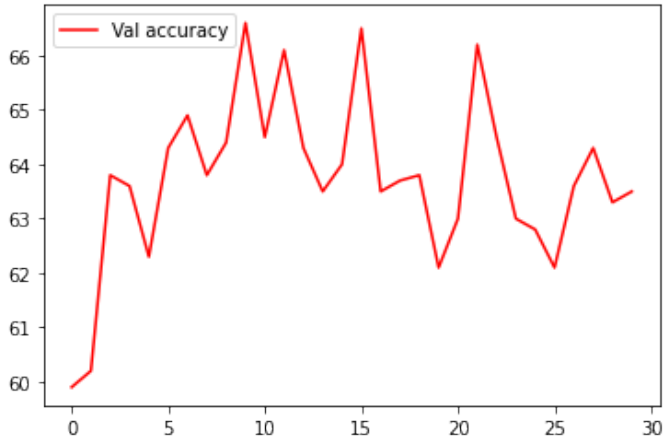
4-a) CIFAR 10 Classification with `VGG_11_bn` model with ImageNet pre-trained weights and disabled gradient:

In this part we have used a pre-trained convolutional network as the backbone feature extraction network and train new layers on top for the target task. We have used the `VGG_11_bn` network from the `torchvision.models` library as our backbone network. `VGG_11_bn` model has been trained on ImageNet achieving top-5 error rate of 10.19%. It consists of 8 convolutional layers followed by adaptive average pooling and fully-connected layers to perform the classification. We got rid of the average pooling and fully-connected layers from the `VGG_11_bn` model and attach our own fully connected layers to perform the CIFAR-10 classification.

For this purpose we have instantiated the pre-trained version of the `VGG_11_bn` model with ImageNet pre-trained weights and we have added two fully connected layers on top, with BatchNorm and ReLU layers between them to build the CIFAR-10 10-class classifier. Our first layer that we have added on top has `in_features` = 512, `out_features` = 256 and the second layer has `in_features` = 256, `out_features` = 10 since we have the 512 as the out of the `VGG_11_bn` and 10 classes for the classification. As it has been requested to set the correct mean and variance in the data-loader, to match the mean and variance which the data was normalized with when the `VGG_11_bn` was trained, we have checked from the internet and it is just matching with the data-loader. Also we have disabled the gradient for the rest of the network. According to the **pytorch** documentation, `requires_grad` is a flag, defaulting to false unless wrapped in a **"nn.Parameter"**, that allows for fine-grained exclusion of subgraphs from gradient computation. It takes effect in both the forward and backward passes. During the forward pass, an operation is only recorded in the backward graph if at least one of its input tensors require grad. During the backward pass **(.backward())**, only leaf tensors with `requires_grad`=True will have gradients accumulated into their **.grad** fields. After all this setups we have trained the model and confirm the validation accuracy oscillate in range 61-66%.



(a) Train and Validation data loss with pretrained `VGG_11_bn` model with ImageNet pre-trained weights.

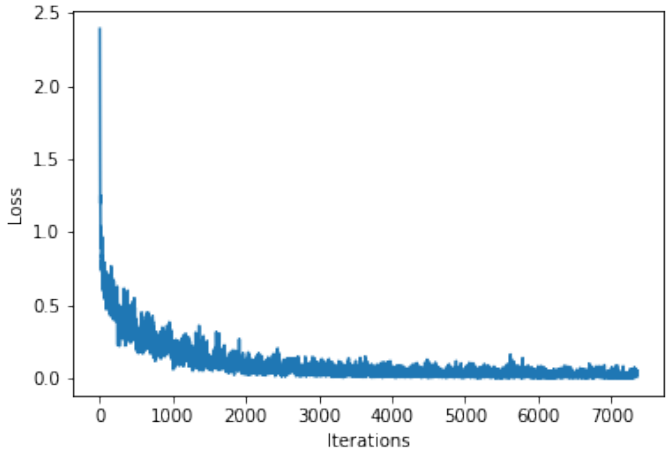


(b) Validation accuracy with pretrained `VGG_11_bn` model with ImageNet pre-trained weights.

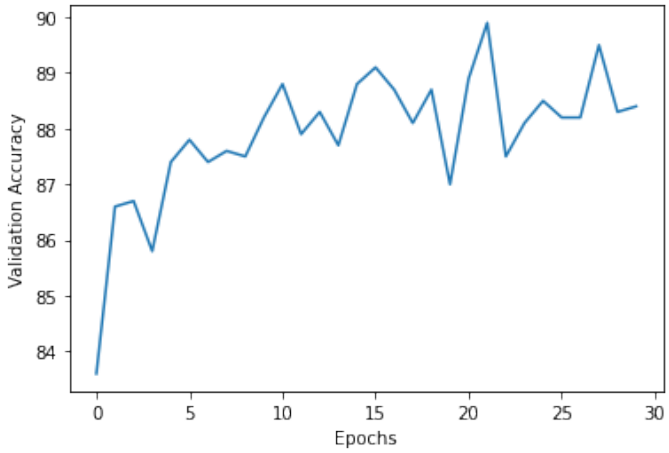
4-b) CIFAR 10 Classification with `VGG_11_bn` model with and without ImageNet pre-trained weights and enabled gradient:

By looking to the loss and validation accuracy plots below we can see that while the ImageNet features are useful, just learning the new layers does not yield better performance than training our own network from scratch. This is due to the domain-shift between the ImageNet dataset (224x224 resolution images) and the CIFAR-10 dataset (32x32 images). To improve the performance we have fine-tuned the whole network on the CIFAR-10 dataset, starting from the ImageNet initialization. To do this, we have enabled gradient computation to the rest of the network, and updated all the model parameters. Additionally we have trained a

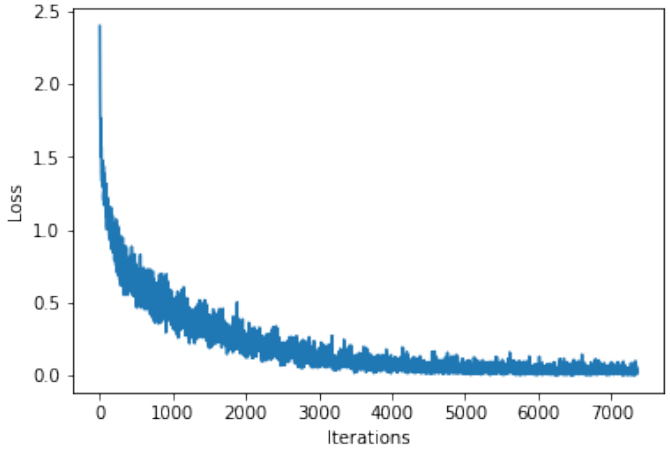
baseline model where the same entire network is trained from scratch, without loading the ImageNet weights. The first models validation accuracy starts from around 83.5% and rich to the 89.9% since we have enabled the gradient for the rest of the network and test set accuracy is 89.2%. In the second model since we have trained a baseline model where the same entire network is trained from scratch, without loading the ImageNet weights the starting validation accuracy point is around 70% as expected and it has reached to 87.7% and the test set accuracy is 87.3%. Which lead us to the proof of the statement we have made at the beginning of this part.



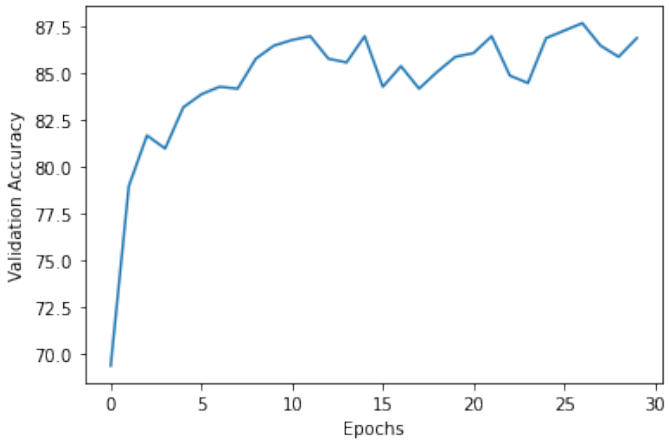
(a) Val loss with pretrained `VGG_11_bn` with ImageNet pre-trained weights and enabled gradient.



(b) Val accuracy with pretrained `VGG_11_bn` with ImageNet pre-trained weights and enabled gradient.



(a) Val loss for the network trained from scratch without loading the ImageNet weights.



(b) Val accuracy for the network trained from scratch without loading the ImageNet weights.