IFT6135 - Assignment 2 - Programming

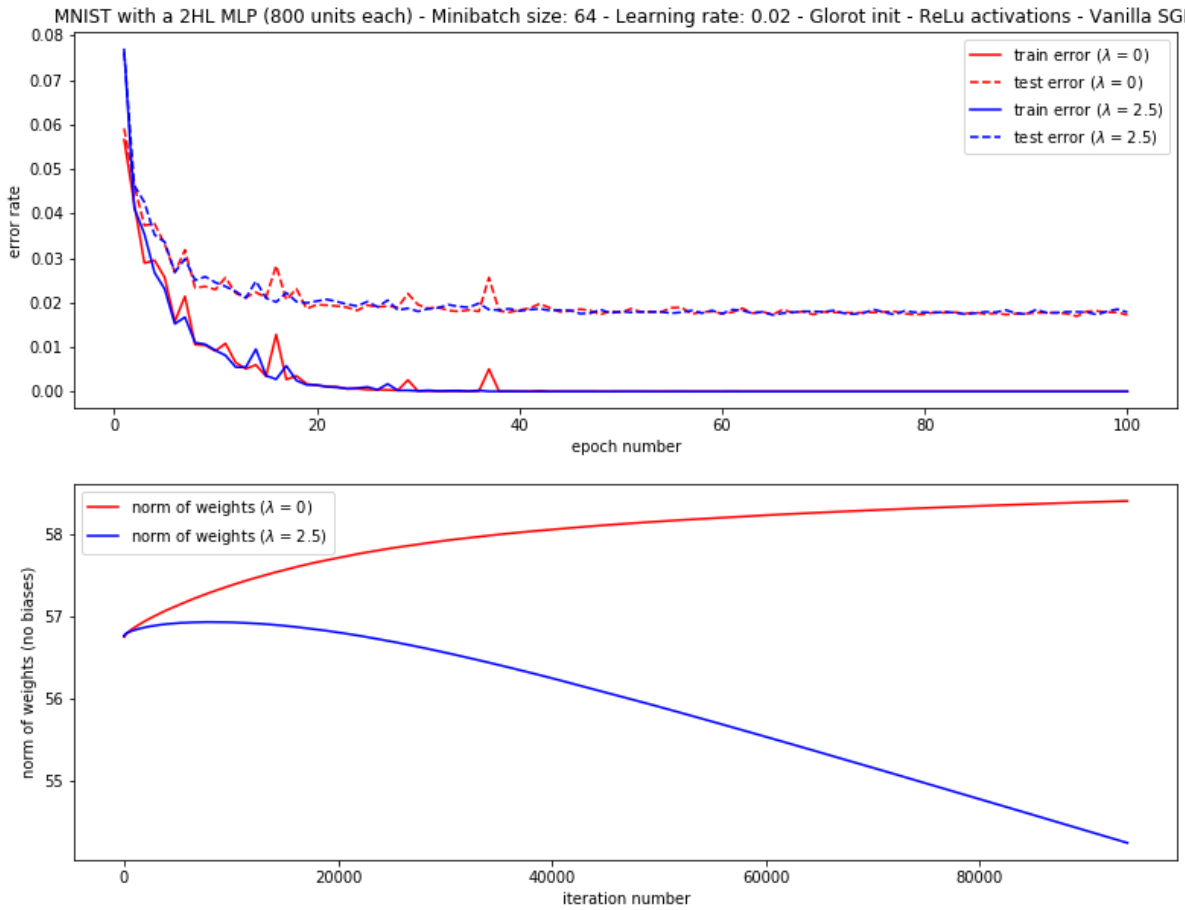
Problem 1

Early stopping and weight decay

We use a Multi-Layer Perceptron with 2 hidden layers of 800 units each. We train this model using SGD on the usual training set of the MNIST database. For this purpose, we use a learning rate of 0.02 and a minibatch size of 64. We compare two versions of the model:

- A non regularized model
- A model with $L2$ regularization with weight decay coefficient $\lambda = 2.5$

We observe the training and test losses for both models during 100 epochs, and the evolution of the norm of the weights of the MLP (Without considering the biases, i.e. the norm of the vector of size $800 \times 784 + 800 \times 800 + 10 \times 800$), at each iteration of the training process.



The first observation we can make is that the regularization does not improve the test accuracy at all. Both models (regularized or not) achieve a perfect training accuracy, but regularization fails to decrease the generalization gap.

We actually were expecting the non-regularized model to overfit after a few epochs, and that early stopping would have come to the rescue, but it doesn't look like so.

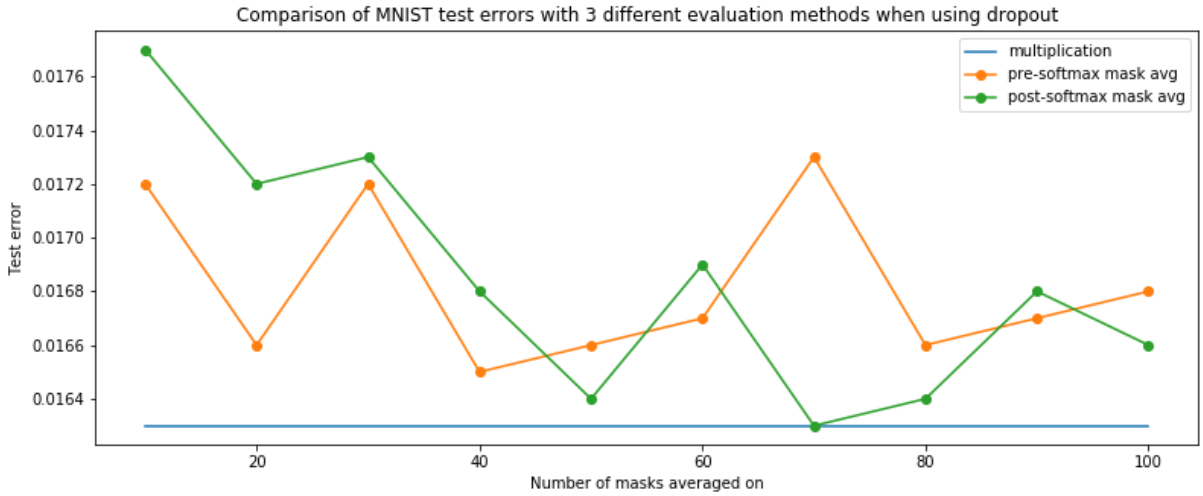
However, one striking difference between both models is the evolution of the norms of the weight. The norm of the weights of the $L2$ regularized model decreases very smoothly, as expected. But it looks like these 100 training epochs are not enough for the weights' norm to converge, so it might be that training the regularized model longer yields slightly better generalization gaps.

Dropout

We use another regularization technique here instead of $L2$ regularization. We use dropout with a rate of $p = 0.5$ on the second hidden layer.
During evaluation, we compare three methods:

- Multiply the last hidden layer activations by the dropout rate (0.5)
- Averaging the pre-softmax values of N evaluations obtaines using N different dropout masks
- Averaging the post-softmax values of N evaluations obtaines using N different dropout masks

We do this for different values of N.



This plot of test error clearly shows that regularizing with dropout has significant positive effect on the test accuracy/error. In fact, the previous non-regularized and $L2$ regularized models achieved a 2% test error, where is dropout here achives something in between 1.6% and 1.7% depending on the method used for inference.

The inference using N masks is actually an approximation of ensemble methods where the average is done over all possible masks. While it is not possible to do it here (800 units in the last layer we apply dropout on), we restrict ourselves to averaging over small but reasonable values of N. The test errors with the (pre-softmax or post-softmax) mask averaging are thus very noisy, and the plot only presents **one** sample.

Multiplying the last hidden layer by the dropout rate can be seen as an approximation (because we have more than one layer) of an ensemble method in which we consider the geometric mean over **all** the possible classifiers defined by the diffrent possible masks.

The averaging methods can be understood as a voting classifier over the set of the N classifiers trained with the different masks.

Convolutional Networks

Convolutional Neural Networks are more suited for classifying images. We train a CNN with the following architecture on the inputs of size $(3 \times 28 \times 28)$:

- 16 kernels of size $(3, 3)$ with padding 1, stride 1, ReLU, MaxPool with kernel of size $(2, 2)$

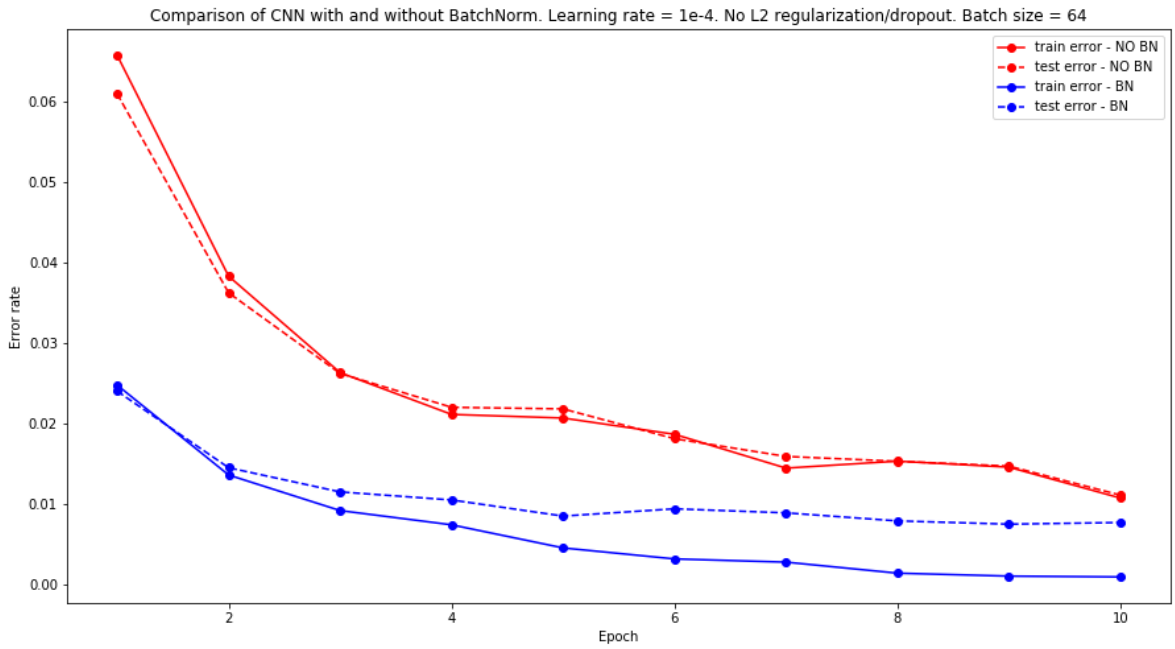
- 32 kernels of size $(3, 3)$ with padding 1, stride 1, ReLU, MaxPool with kernel of size $(2, 2)$
- 64 kernels of size $(3, 3)$ with padding 1, stride 1, ReLU, MaxPool with kernel of size $(2, 2)$
- 128 kernels of size $(3, 3)$ with padding 1, stride 1, ReLU, MaxPool with kernel of size $(2, 2)$
- Fully connected layer mapping 128 inputs to 10 outputs

We train two models with this architecture:

- One without BatchNorm
- One with BarchNorm after each convolution

We change the learning rate to 0.0001 and keep the same minibatch size.

We train the model for 10 epochs only.



First thing we can observe is that whether or not we use BatchNorm, the performance of the CNN is better than that of the MLP, especially that we train only during 10 epochs.

The fact that that there is an almost 0 generalization gap when not using BatchNorm kind of corroborates the fact that CNNs are inherently regularized because of parameter sharing. This by itself, added to the nature of convolutions and how they are adapted to images (invariance to translations, etc...), explains why we get better results than the previous MLP that was trained on 10 times more epochs.

What's more striking is the effect BatchNorm has on this. We obtain a 99% accuarcy after training on 10 epochs only.

Problem 2

For this problem we use the provided script that resizes the images to 64×64 pixels. We could probably get a better accuracy by using a larger image size.
We use the default validation set (size 5000) as test set and partition the training in two parts

- the actual train set (size 16,000, 80% of the original size)
- the validation set (size 4,000, 20% of the original size)

(a) Architecture

A first simple model

For the first model, we took inspiration from the DCGAN's discriminator architecture which only has convolutional layers from beginning to end.
This has a fairly low number of parameters and its training is faster that with big linear layers.

The network has 5 convolutional layers and uses ReLU activations.
The simplest way to explain this architecture is to say that each convolutional layer reduces the feature map size by 2 on each dimension (because of the stride of 2, and the padding of 1), and we augment the number of channels by 2 everytime.

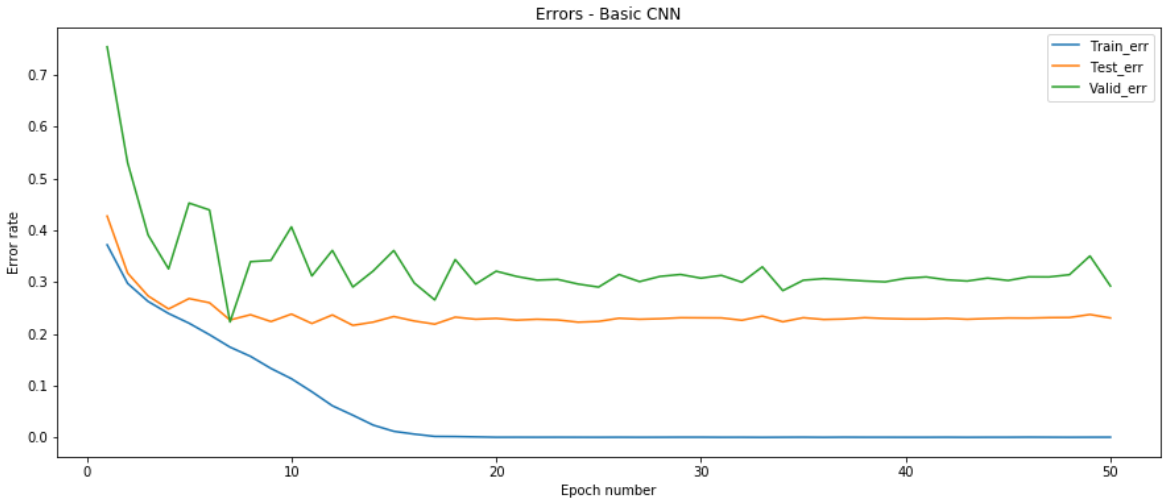
We do not use dilations, and we do not use batchnorm or any regularization here.

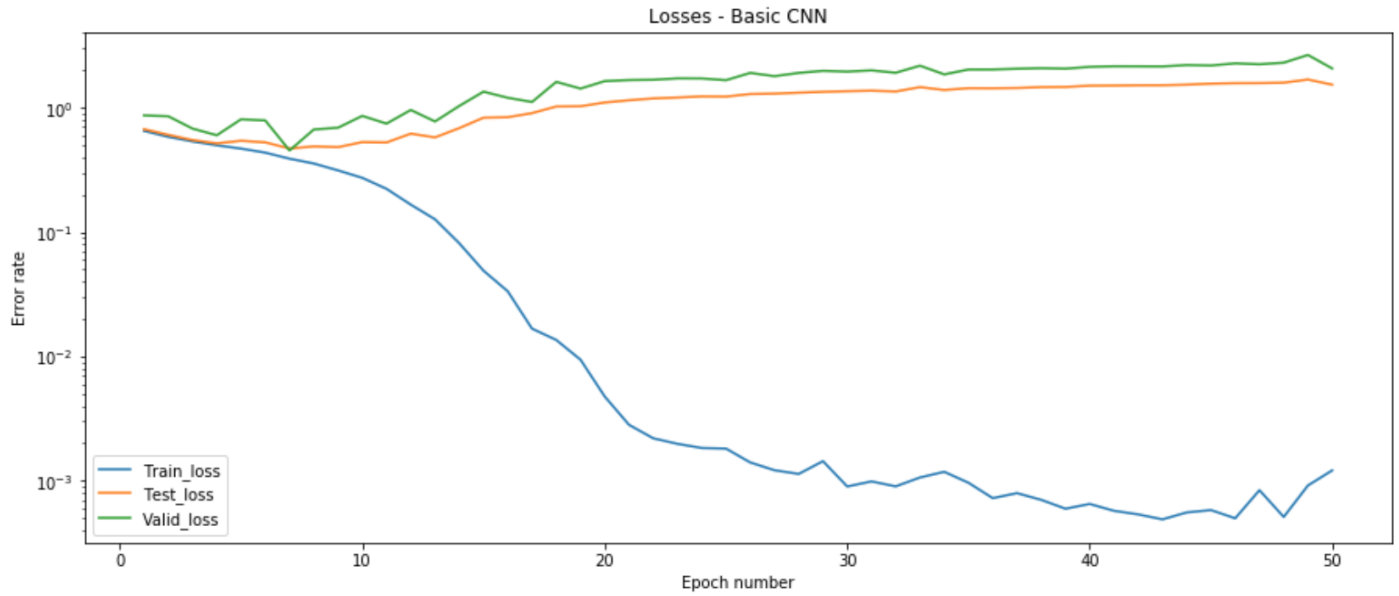
```
class netCNN5(nn.Module):
    def __init__(self):
        super(netCNN5, self).__init__()
        self.main = nn.Sequential(
            # input is 3 x 64 x 64
            nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1 bias=True),
            nn.ReLU(inplace=True),
            # size 64 x 32 x 32
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1, bias=True),
            nn.ReLU(inplace=True),
            # size 128 x 16 x 16
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1, bias=True),
            nn.ReLU(inplace=True),
            # size 256 x 8 x 8
            nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1, bias=True),
            nn.ReLU(inplace=True),
            # size 512 x 4 x 4
            nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=0, bias=True),
            # size 1 x 1 x 1
            nn.Sigmoid()
        )

    def forward(self, input):
        output = self.main(input)
        return output.view(-1, 1)
```

The number of parameters in this model is 2,764,737

In this part, we use the Adam optimizer with default β parameters and a learning rate of 1e-4. While we could probably achieve better generalization (and accuracy!) with plain SGD, this would require a bit of tuning and Adam works pretty well out of the box





Observations

First thing to notice is that we reach the reasonable accuracy of 80.8%. However, we also notice that we overfit from the beginning: while we do not see our validation accuracy rise, the validation loss grows over time and the generalization gap is pretty high.

(b) Improving the model

A better model

We now consider an improvement over the previous model. To tackle the generalization issue, we will

- add explicit regularization (Dropout here)
- use data augmentation on the training set

We do also make use of classical deep learning techniques such as MaxPooling and BatchNormalization. We also use LeakyReLU instead of ReLU as it can increase a bit our accuracy and use kernel sizes of 3×3 (except for the last layer)

```
class betterNet(nn.Module):
    def __init__(self):
        super(betterNet, self).__init__()
        self.main = nn.Sequential(
            # input is 3 x 64 x 64
            nn.Conv2d(3, 64, 3, 1, 1, bias=True),
            nn.LeakyReLU(0.2, inplace=True),
            nn.MaxPool2d(2),
            # size 64 x 32 x 32
            nn.Conv2d(64, 128, 3, 1, 1, bias=True),
            nn.LeakyReLU(0.2, inplace=True),
            nn.BatchNorm2d(128),
            nn.Dropout2d(0.2),
            nn.MaxPool2d(2),
            # size 128 x 16 x 16
            nn.Conv2d(128, 256, 3, 1, 1, bias=True),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout2d(0.2),
            nn.MaxPool2d(2),
            # size 256 x 8 x 8
            nn.Conv2d(256, 512, 3, 1, 1, bias=True),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),
            #nn.Dropout2d(0.2),
            nn.MaxPool2d(2),
            # size 512 x 4 x 4
            nn.Conv2d(512, 1, 4, 1, 0, bias=True),
            # size 1 x 1 x 1
            nn.Sigmoid()
        )

    def forward(self, input):
        output = self.main(input)
        return output.view(-1, 1)
```

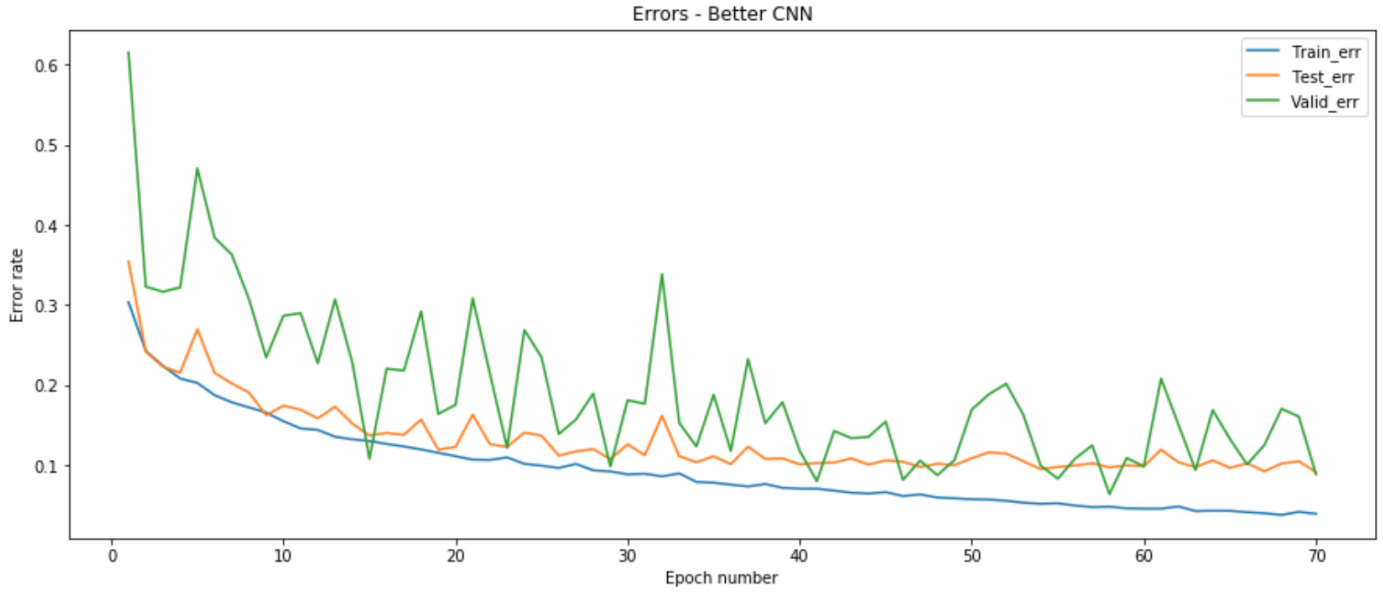
This model has approximately 2 times fewer parameters: 1,560,961.

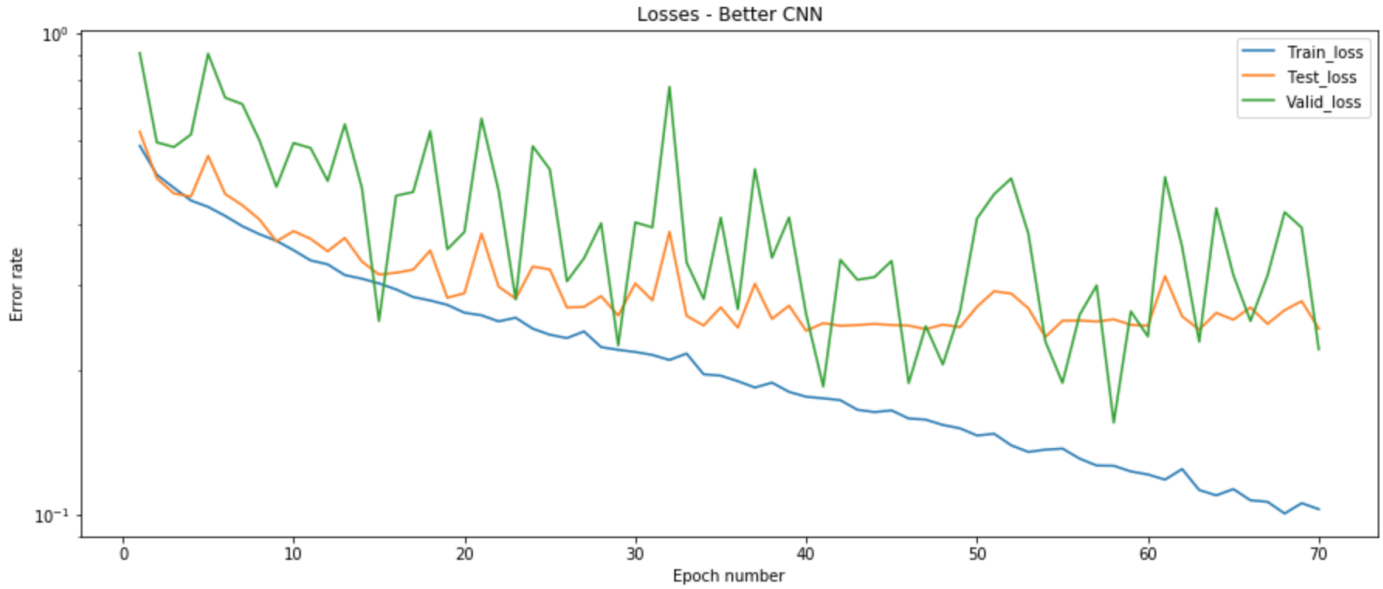
Notice that to increase the performance of our model, we add some data augmentation to the training set. Namely we add some random horizontal flippings, some random brightness change and small random rotations to the input.

```
train_set = datasets.ImageFolder(root=datapath+'/train_64x64',
                                 transform=transforms.Compose([
                                     transforms.Resize(64),
                                     transforms.CenterCrop(64),
                                     transforms.RandomHorizontalFlip(),
                                     transforms.ColorJitter(),
                                     transforms.RandomRotation(15),
                                     transforms.ToTensor(),
                                     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                                 ]))
```

We tested different architectures, learning rates, optimizers, and dropout rate, and this one was consistently performant.

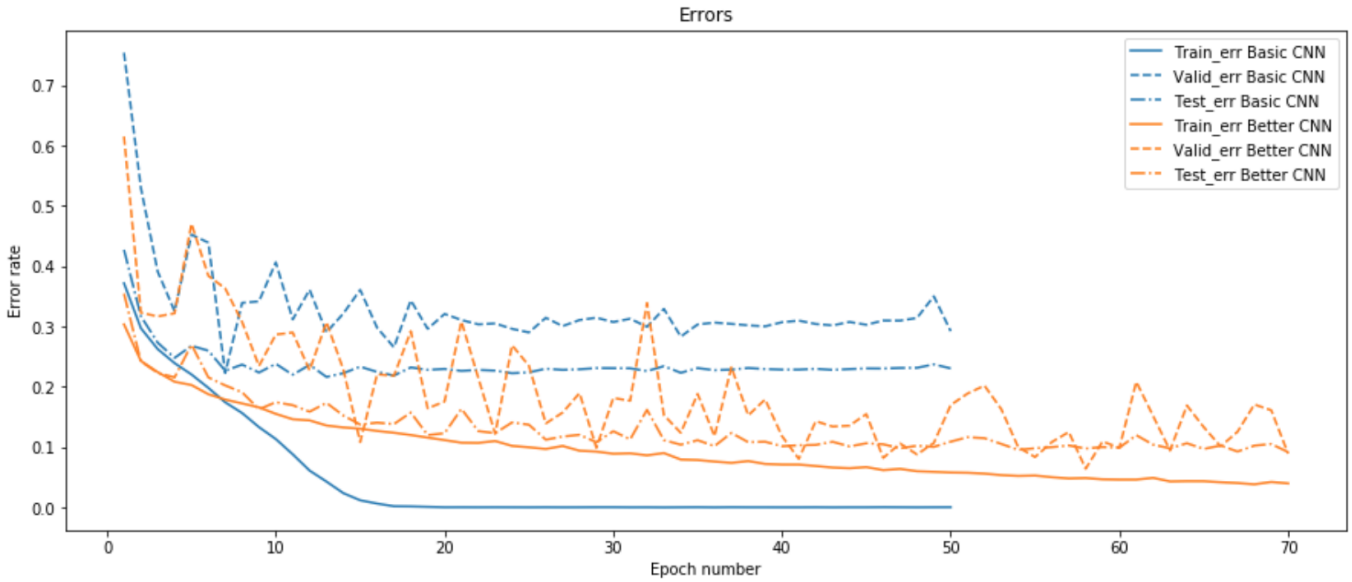
We get the following learning curves for the new model:





Our new model is indeed more performant. The accuracy gap is lower and we reach a better accuracy (around 93.5%).

The comparison is striking



The generalization gap is much smaller and we reach rapidly very good accuracies!

Visualizations

Features maps

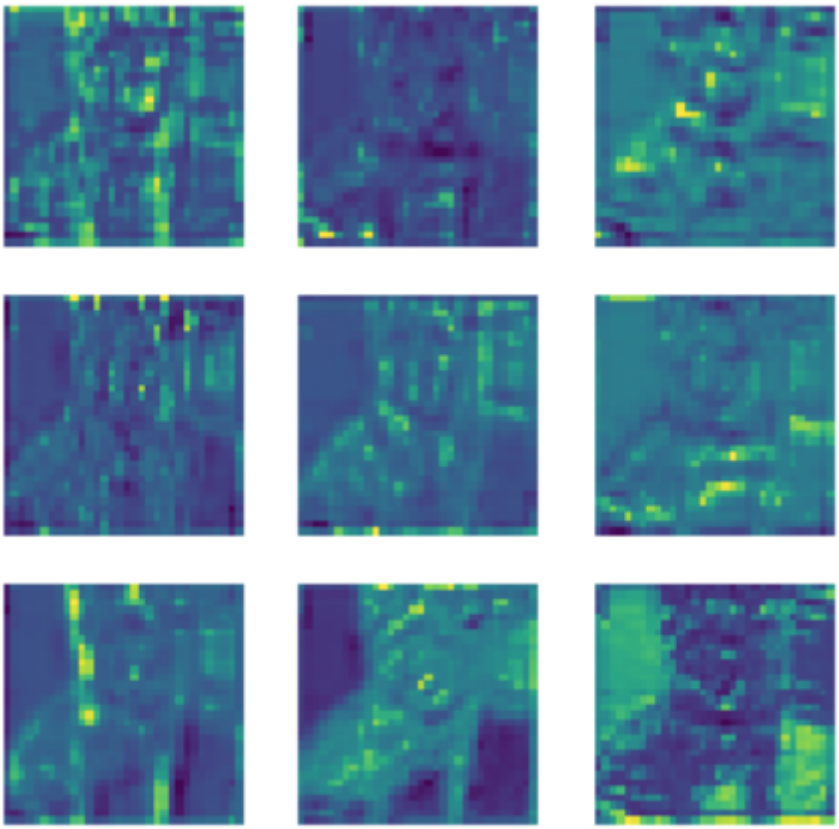
Let us consider the following validation image

We will plot the features maps in 9 channels of the feature map at each layer.

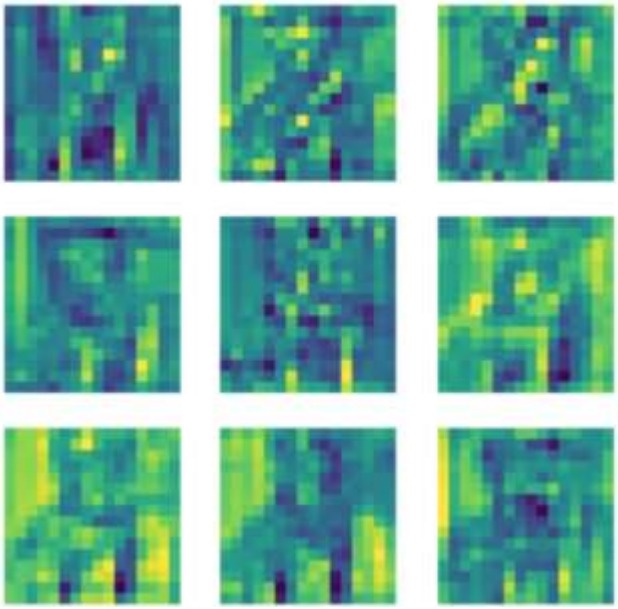
In the first layer

We observe that some channels seem to differentiate foreground and background, some seem to be detecting edges while some others seem to be sensitive to the color of the dog's hair.

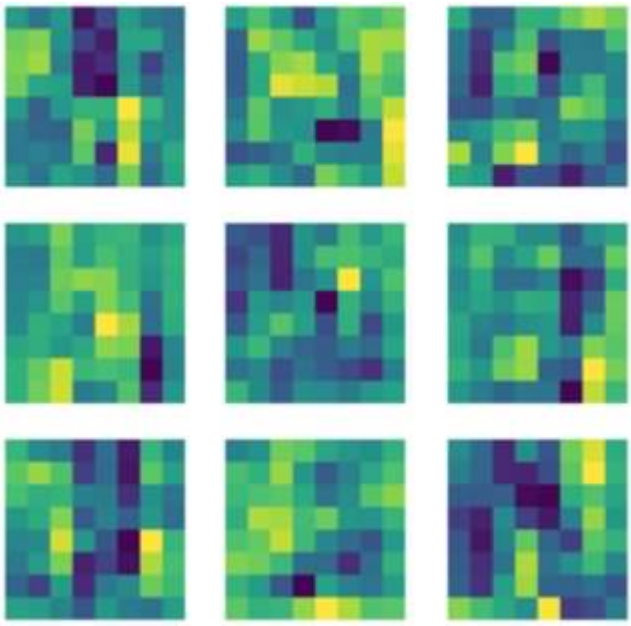
In the second layer, the feature map is getting smaller and harder to interpret, but it seems some channel are now looking at specific places in the input (like dog's ears, nose and legs).



Feature maps of the layer 3 and 4 become low resolution and very hard to interpret.



Layer 4



Misclassified images with high confidence

Here are some example of misclassified image with really high confidence (>0.95)

We observe that in most cases the head of the animal is cut, there is even one image where we both have a dog and a cat!

Neutral images

Here we plot images where our neural network add a balanced score around 50/50 for each class.

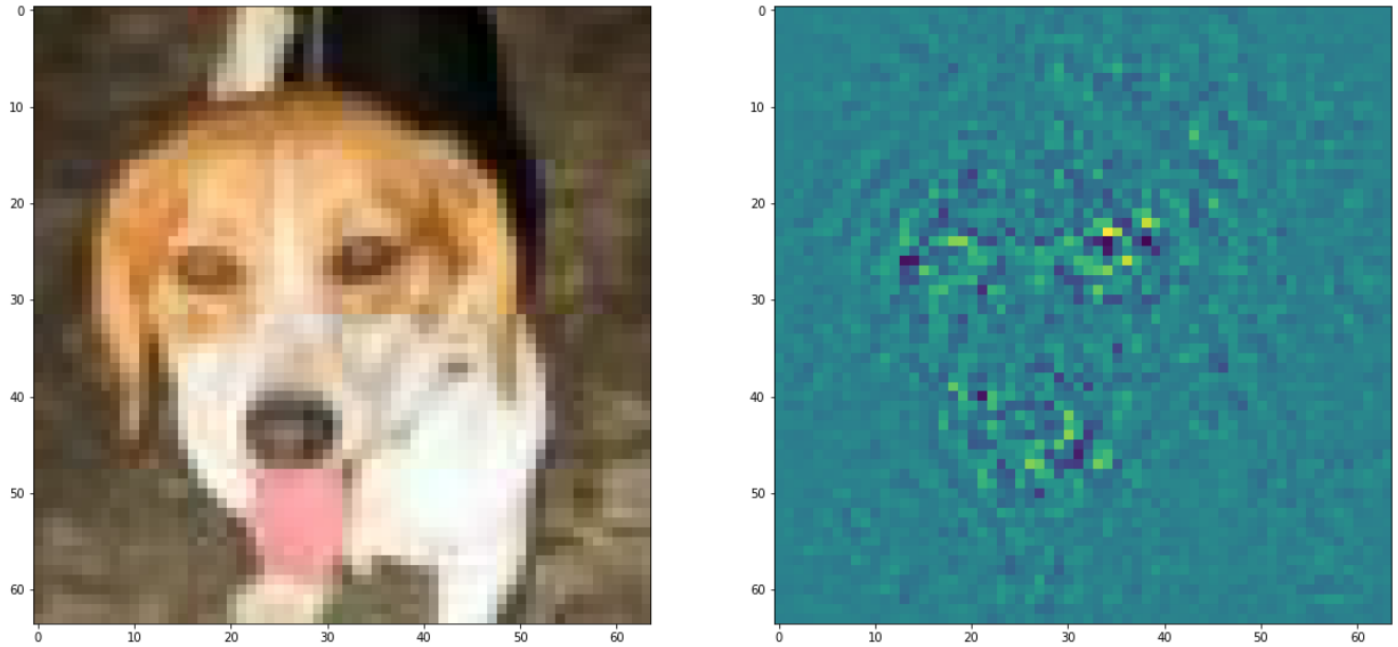
These are clearly mistakes which are not as interesting to analyze as the previous ones.

Saliency map

We here plot the saliency map ie the heatmap of the gradient norm of each pixel with respect to the target score.
More precisely, this is the heatmap (averaged over channels) of

$$\nabla_x f_\theta(x)[y]$$

Where x is an input image, f_θ is our trained classifier (without the sigmoid/softmax layer) and y is the target class.



We observe that our classifier puts a lot of weights on the pixels of the dog's eyes and mouth and they will probably be very important for the classification.

Other examples