Víctor Pérez *183950*
Diego Vincent *174458*
Óscar Font *183826*

Deep Learning

Laboratory 2

Q1. Implement and analyse a Neural Network with L1 and L2 regularizer in the ipython notebook: *SP1.2-1_BasicMLPRegularizer(Question).ipynb* (20 Points).

In this first task we are asked to implement a Convolutional Neural Network with Pytorch. The first step is to complete the CNN class and we did it as follows.

```
class CNNFC(nn.Module):

    def __init__(self, num_classes=10):
        super().__init__()
        default_num_filters = 8
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, default_num_filters, kernel_size=2),
            nn.Tanh(),
        )

        self.fc = nn.Sequential(
            nn.Linear(32, num_classes),
            nn.Sigmoid()
        )

    def forward(self, x):
        #dont forget to reshape your output of CNN to be one vector
        #ex : out.reshape(out.size(0), -1)
        out = self.conv1(x)
        out = out.reshape(out.size(0), -1)
        print(out.size())
        out = self.fc(out)
        return out
```

Figure 1

We have used the order Sequential to store in the attribute conv1 the result of the convolution and then the result of the Tanh function in this order. After that we have the fully connected (fc) attribute with another Sequential to store the result of applying: first the Linear function and then the Sigmoid activation function. Finally, as it can be seen in the definition of the forward function we call first the convolution attribute, we reshape the result and we then pass these data to the fully connected, whose result is the one to return.

The next step is to define the parameters to be able to train the Network. Our way to define them can be seen in Figure 2.

Víctor Pérez *183950*
Diego Vincent *174458*
Óscar Font *183826*

```
#! Task 2, copy the old weight filter of CNN using np.copy()

cw1 = np.copy(list(CNN.conv1.parameters()))
cw2 = np.copy(list(CNN.fc.parameters()))

#! Task 3, define the learning rate and optimizer
learning_rate = 0.01
optimizer = torch.optim.Adam(params=CNN.parameters(), lr=learning_rate)

# Device configuration
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
num_epoch = 5
```

Figure 2

After defining those parameters we just need to train the net. We train the CNN and we get the corresponding costs before and after the training. Before the training we compute the cost with the following formula.

```
cost = np.square(- Y[i]).sum() * 0.5
```

Figure 3

The cost after the training is computed as shown in the examples provided by the teacher. The results we get are:

- Cost before Training: tensor(1.4291)
- Cost after Training: tensor(0.1351)

Q2. Implement a Pytorch dataset using Pytorch to load CatDog data in ipython notebbok : *SP2.1-2_PyTorchDatasetExamples-Question.ipynb* (10 points).

In this task we were asked to complete a class to load certain data from folders in order to use them in the future with some Networks. We have completed the class as follows.

```
import os


default_directory = os.path.join(myDrive, 'CatDog/train')


class CatDog(torch.utils.data.Dataset):

    def __init__(self, transform, *, dataDir = default_directory):
        #Initialize the data and label list
        self.labels = []
        self.data = []

        #First load all images data
        listImage = os.listdir(dataDir)

        for x in listImage:
            #Second filter according name for labelling : cat : 1, dog : 0
            lbl = 1 if 'cat' in x else 0
            # Append to our class attribute :)
            path = os.path.join(dataDir, x)
            self.data.append(path)
            self.labels.append(lbl)


    def __getitem__(self, index):
        img = Image.open(self.data[index])
        img = transform(img)
        return img, self.labels[index]


    def __len__(self):
        return len(self.data)
```

Figure 4

As can be seen in Figure 4 in the constructor we create two attributes one for the files and the other for the respective labels (cat 1 or dog 0). We then access the directory passed as parameter and iterate through the list of files in the folder. In each iteration we store the path of the file in one attribute and the label in the other. After that the getter, *getItem()* returns the image from the path of one of the attributes with the corresponding label. Finally the length attribute returns the length of the data attribute which is the number of files in the folder.

Q3. Implement Basic CNN with Fully Connected Bottleneck layer for Cat/Dog dataset analysis in the ipython notebook: *SP2.1-3_PyTorchDatasetAndTrainCross-Question.ipynb* and analyse (15 points).

In this third task we are asked to implement a basic Neural Network with a fully connected bottleneck layer for the CatDog dataset try to get maximum accuracy and analyse the results.

First we defined our network as can be seen in Figure 5.

```
class ConvNetSig(nn.Module):
    def __init__(self, num_classes=2):
        super().__init__()
        default_num_filters = 32
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, default_num_filters, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(default_num_filters, default_num_filters, kernel_size=3,stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        self.fc = nn.Sequential(
            nn.Linear(131072, num_classes),
            nn.Sigmoid()
        )

    def forward(self, x):
        #dont forget to reshape your output of CNN to be one vector
        #ex : out.reshape(out.size(0), -1)
        out = self.conv1(x)
        out = out.reshape(out.size(0), -1)
        #print(out.size())
        out = self.fc(out)
        return out
```

Figure 5

In the class constructor we apply two convolutions to the data, followed each by a ReLU function and then by a Max Pooling so we get the bottleneck effect desired, because it will reduce the number of parameters. Then we have the fully connected net with the Linear function and the Sigmoid in the last layer.

After that we define the CNN, loss function and parameters (learning rate, number of epochs, etc.) as can be seen in Figure 6.

```
CNNS = ConvNetSig()
CNNS = CNNS.to(device)
criterion = nn.CrossEntropyLoss()

#optimizer
learning_rate = .001
optimizer = torch.optim.Adam(CNNS.parameters(), lr = learning_rate)


# Device configuration
num_epochs = 10
```

Figure 6

Once everything is defined, when we train and test the model, we obtain the loss plots that can be seen in the following Figures.

Víctor Pérez *183950*
Diego Vincent *174458*
Óscar Font *183826*

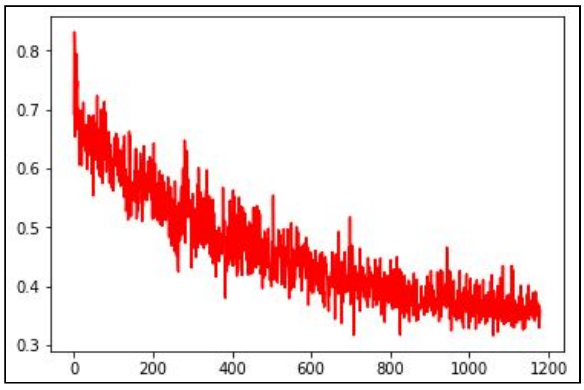

Figure 7

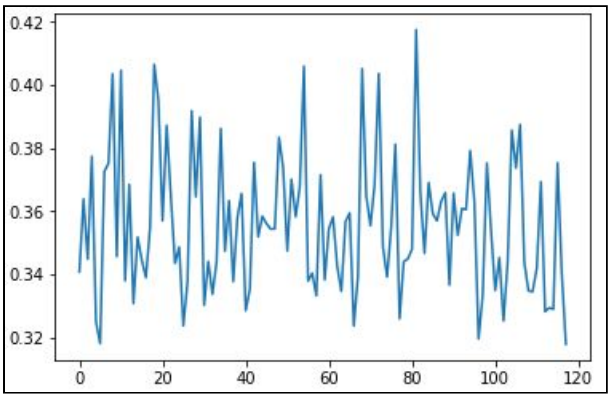

Figure 8

On the one hand, on the Figure 7 we have the loss during the training, which in the end converges to a 0.3 value of error. On the other hand, on Figure 8, we have the loss during the test, which ends up converging to a 0.3 value as the other plot. Both graphs can make us think that the model might perform well, but not perfectly, given that both of the losses converge to a number close to 0.

To try to go ahead and confirm or refuse our theory we just need to check the accuracy computed during the test phase. We actually obtained the following results:

- Correct classified: 7140
- Total tested images: 7500
- Accuracy of the model: **95.2 %**

Finally we were asked to visualize the filters, and we do so. We obtain the following plots.

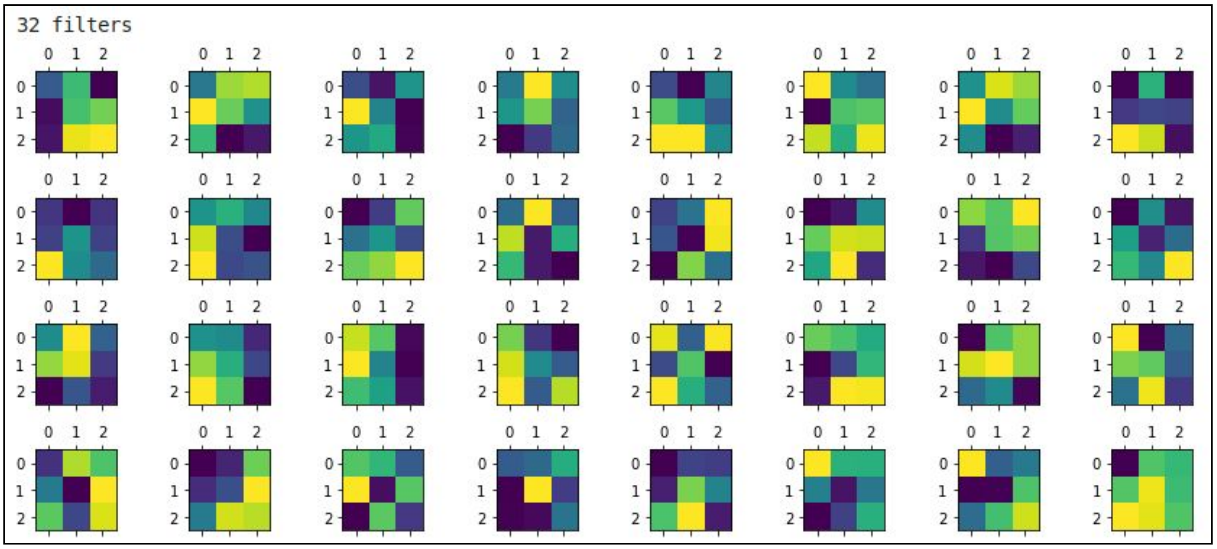

Figure 9

Víctor Pérez *183950*
Diego Vincent *174458*
Óscar Font *183826*

Q4. Design custom CNN based on current state of the art architecture and try to obtain highest accuracy possible with CatDog dataset in the ipython notebook : *SP2.1-4_ConvolutionExtended-Question.ipynb* and explain the process (50 Points).

In this task we were asked to implement a custom network based on the current state of the art architectures. After having seen the example provided by the teacher, the two best performing architectures with the MNist dataset were HourGlass followed by ResNet. So for this task we decided to implement this architectures to see which one performs the best with the CatDog dataset.

Here in Figures 10 and 11 we show our implementations of the ResNet and HourGlass architectures for the task.

```
class HourGlassSim(nn.Module):
    def __init__(self, num_classes=10):
        super(HourGlassSim, self).__init__()

        self.convols11_12 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=5, stride=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        self.intermediates = nn.Sequential(
            nn.Linear(31, 512),
            nn.Linear(512,31)
        )

        self.convols21_22 = nn.Sequential(
            nn.ConvTranspose2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Upsample(scale_factor = 2),
            nn.ConvTranspose2d(64, 64, kernel_size=5, stride=1, padding=1),
            nn.ReLU(),
            nn.Upsample(scale_factor = 2),
        )

        self.fc = nn.Linear(1048576, num_classes)

    def forward(self, x):

        out11_12 = self.convols11_12(x)

        intermediate = self.intermediates(out11_12)

        out21_22 = self.convols21_22(intermediate)

        out21_22 = out21_22.reshape(out21_22.size(0), -1)
        out = self.fc(out21_22)

        return out
```

Figure 10

Víctor Pérez *183950*
Diego Vincent *174458*
Óscar Font *183826*

```
class ResNetSim(nn.Module):
    def __init__(self, num_classes=10):

        super(ResNetSim, self).__init__()

        self.convols11_12 = nn.Sequential(
          nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
          nn.ReLU(),
          nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
          nn.ReLU()
        )

        self.convols21_22 = nn.Sequential(
          nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
          nn.ReLU(),
          nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
          nn.ReLU(),
        )

        self.MaxPooling = nn.MaxPool2d(kernel_size=2, stride=2)

        self.fc = nn.Linear(262144, num_classes)

    def forward(self, x):

        convols11_12 = self.convols11_12(x)
        convols21_22 = self.convols21_22(convols11_12)

        pooling = self.MaxPooling(convols21_22)

        pooling = pooling.reshape(pooling.size(0), -1)
        out = self.fc(pooling)

        return out
```

Figure 11

Then we define define the CNN, optimizers, loss function and parameters (learning rate, number of epochs, etc.). Our approach is the following:

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
fName = 'HourGlass' #'ResNet'
CNN = HourGlassSim() #ResNetSim()
CNN = CNN.to(device)

#criterion
criterion = nn.CrossEntropyLoss()

#optimizer
learning_rate = .001
optimizer = torch.optim.Adam(CNN.parameters(),lr = learning_rate)

# epochs
num_epochs = 10
```

Figure 12

As can be seen in Figure 9 we have CNN and fName variables to be able to train first with both structures and save both the models and the loss results in a file, so then we can compare the losses during the training of both HourGlass and ResNet.

The next step after training and generating both models with both CNN architectures is to test both models and save the losses to different files. Different accuracies are also obtained and then we compare both training and test losses. So once all those are done, we get the following loss plots.

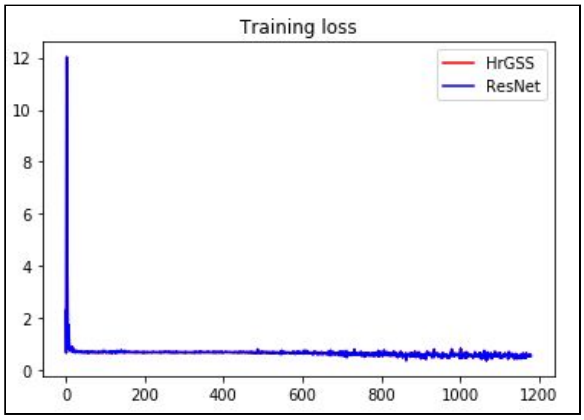

Figure 13

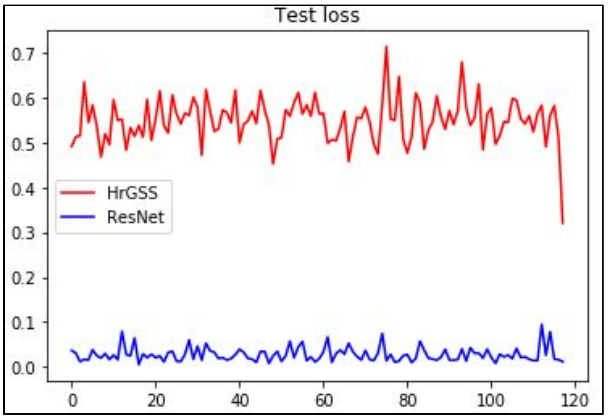

Figure 14

On the one hand, when talking about the training loss we can see that both models have very similar losses, both converging very close to zero, which is a good signal. Given that the HourGlass was the first one plotted and the second one plotted was the ResNet one, and that both plots are almost identical, we can just see the blue one.

On the other hand, on the Test loss plot we can clearly see that ResNet has less loss in the testing part rather than HourGlass, which can start telling us that ResNet model performed better than HourGlass.

Finally to check our predictions we obtain the accuracies after the testing of both models. The accuracies obtained are:

- Accuracy of HourGlass: 72.70666666666666 %
- Accuracy of ResNet: **99.46666666666667 %**

In conclusion, for the example task of MNist dataset HourGlass outperformed the other CNN architectures, but to tackle the CatDog dataset problem, as seen in the above results, ResNet performed more accurately than HourGlass.