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

Deep Learning

Laboratory 1

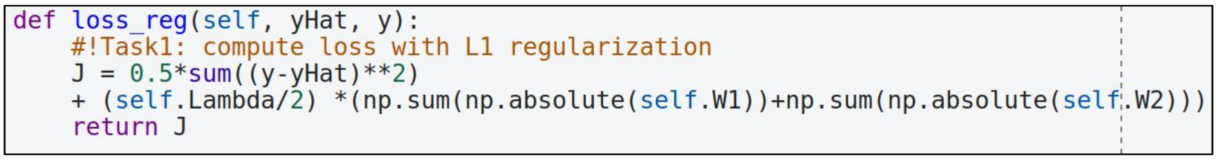
Q1. Implement a Basic Convolutional Neural Network using Pytorch in the ipython notebook : *SP2.1-1_ConvolutionBasic-Question.ipynb* (25 points).

In this first task we are asked to implement the loss function of a Neural Network with the L1 regularization. L1 regularization, also known as Lasso Regression, consists in adding to the Loss function a *lambda* parameter multiplied by a sum of the absolute value of the weights, this has also the name of *absolute value of magnitude*.

$$L(x,y) \equiv \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \boxed{\lambda \sum_{i=1}^{n} |\theta_i|}$$

Figure 1

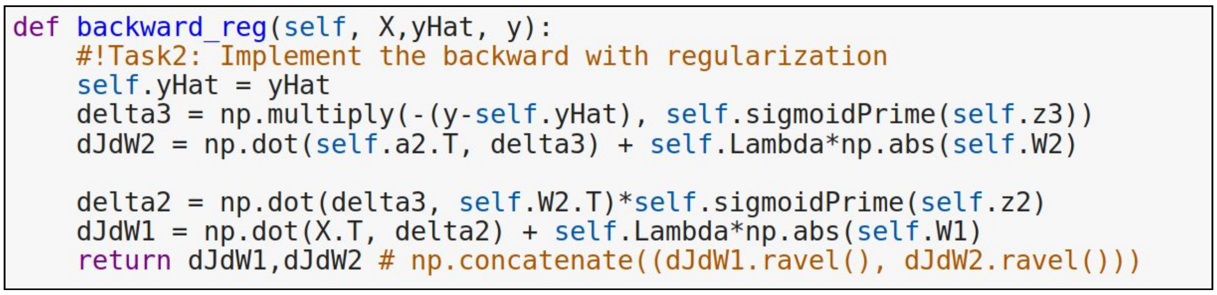
In the Jupyter Notebook we had to implement that function in the **loss_reg** function. Our implementation ended up being like is shown in Figure 2.



```
def loss_reg(self, yHat, y):
    #!Task1: compute loss with L1 regularization
    J = 0.5*sum((y-yHat)**2)
    + (self.Lambda/2) *(np.sum(np.absolute(self.W1))+np.sum(np.absolute(self.W2)))
    return J
```

Figure 2

After that, to fully implement the L1 regularization to the Neural Network we have to modify the backpropagation function, by updating the gradients accordingly. So our code ended up looking like this.



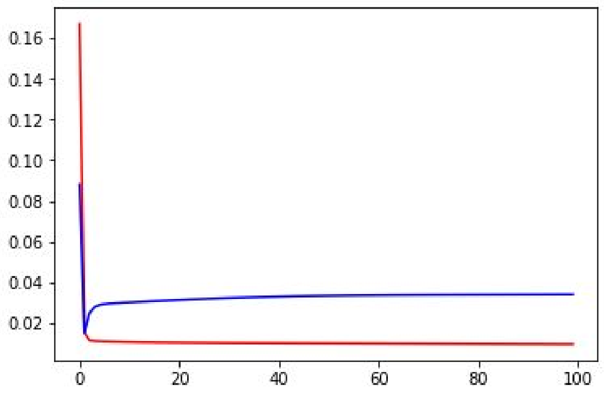
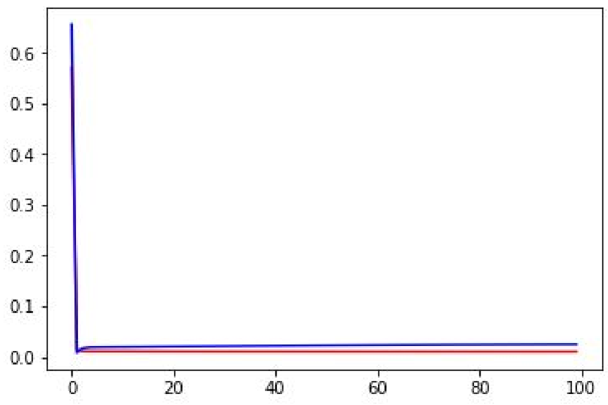
```
def backward_reg(self, X,yHat, y):
    #!Task2: Implement the backward with regularization
    self.yHat = yHat
    delta3 = np.multiply(-(y-self.yHat), self.sigmoidPrime(self.z3))
    dJdW2 = np.dot(self.a2.T, delta3) + self.Lambda*np.abs(self.W2)

    delta2 = np.dot(delta3, self.W2.T)*self.sigmoidPrime(self.z2)
    dJdW1 = np.dot(X.T, delta2) + self.Lambda*np.abs(self.W1)
    return dJdW1,dJdW2 # np.concatenate((dJdW1.ravel(), dJdW2.ravel()))
```

Figure 3

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

Finally we were asked to train the Neural Network with the regularization several times and analyze what happens with different lambda values. In our experiment we tried it with different lambda values and we will show you one overfitting case and one case which the regularization works correctly. All the graphs represent the loss vs the epochs.

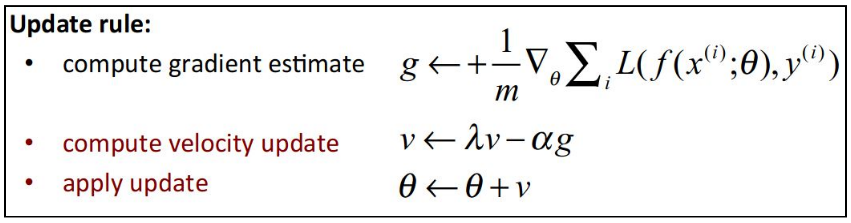
Lambda	**Training set loss and Test set loss**
0.00001	
0.0001	

We can observe that in the first graph, due to an almost negligible lambda value, overfitting takes place since the loss for the test set is higher than for the training set. However, as we regularize our model, the test set performs better (better generalization due to regularization).

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

Q2. Implementation and explanation of a NN's Backpropagation with Momentum in the ipython notebook: SP1.2-2_BasicMLPOptimizer(Question).ipynb (10 Points).

Here we are asked to apply Momentum Optimization to a Neural Network and than compare it to another already implemented which is the Stochastic Gradient Descent Optimizer.
Momentum is designed to accelerate the learning, especially for high curvature, small but consistent gradients or noisy gradients. Momentum uses the following update rule (where alpha denotes the learning rate).

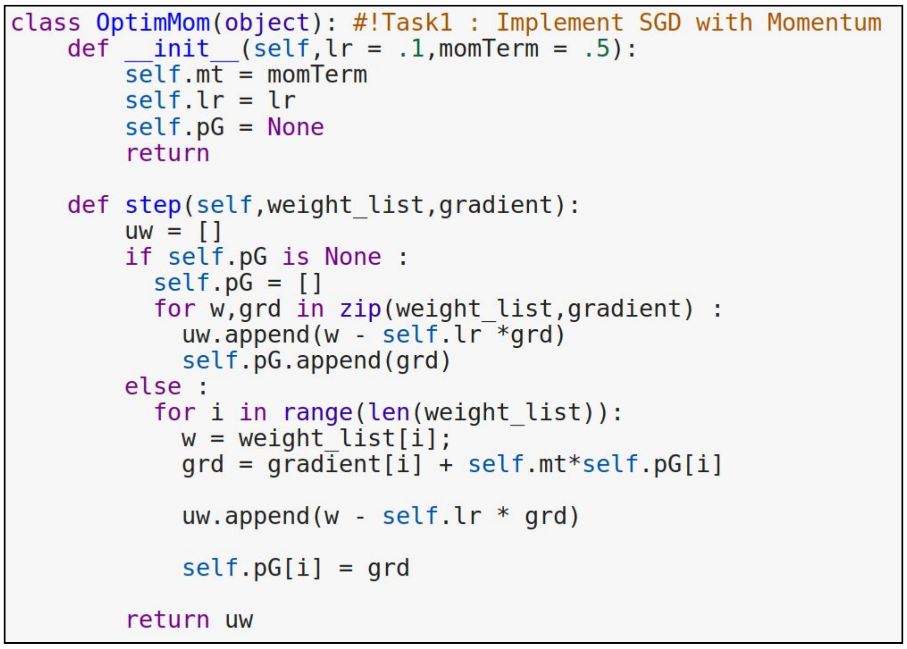


Update rule:

- compute gradient estimate $g \leftarrow +\frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)})$
- compute velocity update $v \leftarrow \lambda v - \alpha g$
- apply update $\theta \leftarrow \theta + v$

Figure 4

Having seen that, we have implemented this idea in the OptimMom and the result is the following.



```
class OptimMom(object): #!Task1 : Implement SGD with Momentum
    def __init__(self,lr = .1,momTerm = .5):
        self.mt = momTerm
        self.lr = lr
        self.pG = None
        return

    def step(self,weight_list,gradient):
        uw = []
        if self.pG is None :
          self.pG = []
          for w,grd in zip(weight_list,gradient) :
            uw.append(w - self.lr *grd)
            self.pG.append(grd)
        else :
          for i in range(len(weight_list)):
            w = weight_list[i];
            grd = gradient[i] + self.mt*self.pG[i]

            uw.append(w - self.lr * grd)

            self.pG[i] = grd

        return uw
```

Figure 5

Once implemented, further in the notebook we two different trainings of the Network, one for each Optimization method. The idea is to execute it and see the difference between both. In the resulting Figure 6 we have plotted in blue the SGD trained NN and in red the Momentum trained NN. In the graph we see the loss in front of the epochs.

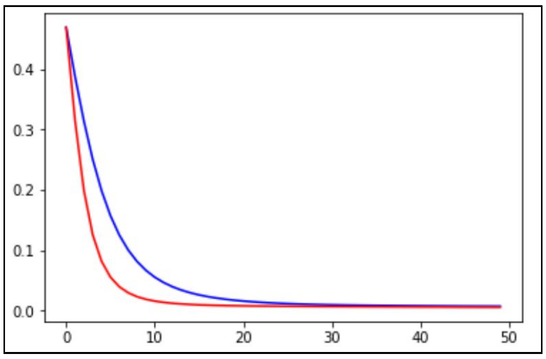


Figure 6

As it can be seen in the Figure, the loss decreases faster in the Neural Network trained with the Momentum optimizer as it does with the Neural Network trained with the SGD Optimizer.

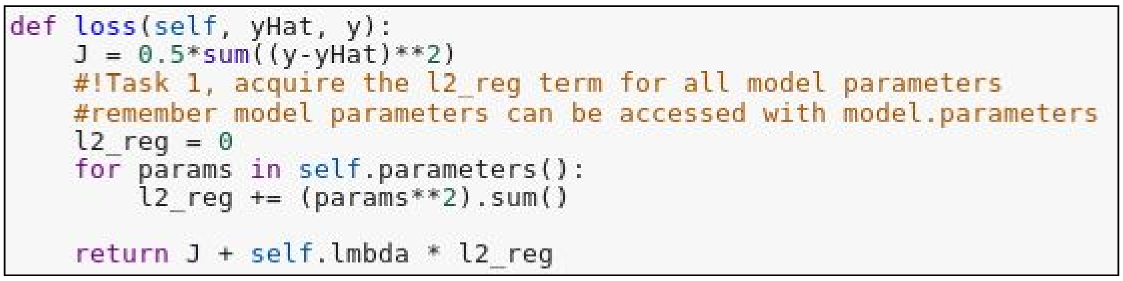
Q3. Implement the models on native pytorch architectures, with regularizer and momentum optimizers: SP1.2-3_BasicMLPPytorch(Question).ipynb (20 Points).

To start with this task, we are first asked in the loss function to get all the parameters and add them to the function as L2 regularization term. L2 regularization, also known as Ridge regression, adds the *squared magnitude* at the end of the loss function, which is a lambda term multiplied by the sum of the squared weights.

$$L(x,y) \equiv \sum_{i=1}^{n}(y_i - h_\theta(x_i))^2 \boxed{+ \lambda \sum_{i=1}^{n} \theta_i^2}$$

Figure 7

To apply this to the loss function in our code it ended up looking like seen in FIgure 8.



```
def loss(self, yHat, y):
    J = 0.5*sum((y-yHat)**2)
    #!Task 1, acquire the l2_reg term for all model parameters
    #remember model parameters can be accessed with model.parameters
    l2_reg = 0
    for params in self.parameters():
        l2_reg += (params**2).sum()

    return J + self.lmbda * l2_reg
```

Figure 8

Once this task was completed, we just had to execute the code and compare both plots of the losses in the training and in the test sets. The plot we obtained is the following of Figure 9, in which the blue line is the test loss and the red line is the training loss. In the graph we plotted loss vs epochs.

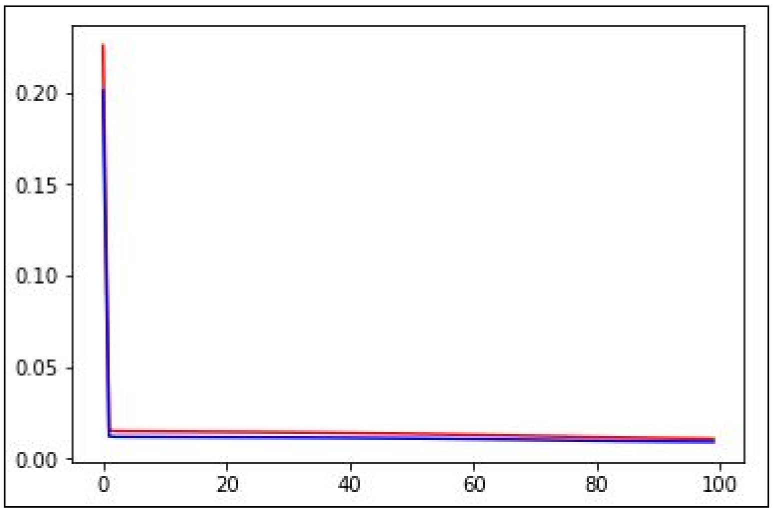


Figure 9

As can be seen in Figure 9, both the loss in the training and in the test set converge to a number very close to 0, which means that between the prediction of the network and the real values both in the training set and in the test set there is no much difference. This means the Network generalizes and predicts quite well.

Q4. Study-case on Height-Weight datasets : SP1.2-4_MLP_HW(Question).ipynb (50 Points)

We have implemented this linear model using the Sequential module from *pytorch*. We have first used a Linear layer with the number of inputs and hidden units set by the user. We have then implemented a Sigmoid activation function on the output, then we have added another Linear module with the hidden and the output units set also by the user and we have finally added a Sigmoid activation function at the end.

For the loss function, as this is a binary classification problem, we decided to use the binary cross entropy loss, which is already implemented in pytorch and works well to obtain losses of this kind of classification problems. We have also added an L1 regularizer for the parameters of the net to the loss function weighted by a lambda set by the user. We used the *l1_loss* function from pytorch for this purpose (we added the parameters and zeros so it just takes into account the value of the parameters).

We have also initialized the weights sampling random values from a normal distribution with an std of 0.02. We have done that using the *randn* function of pytorch (which give us values sampled from a normal distribution with mean 0 and a std of 1) and multiplying this values by 0.02.

Our best accuracy has been using an SGD optimizer with a learning rate of 0.01 and training the model for 2000 epochs with a batch size of 16. We reported a training accuracy of 89% and a test accuracy of 95%.

We can observe the increasing trend of our model for the accuracy as the number of iterations augment in the figure below:

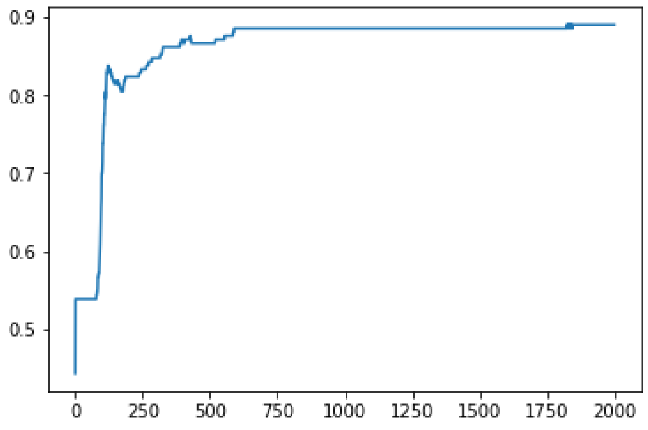


Figure 13.

The classification discrimination contour can be seen in the following figure:

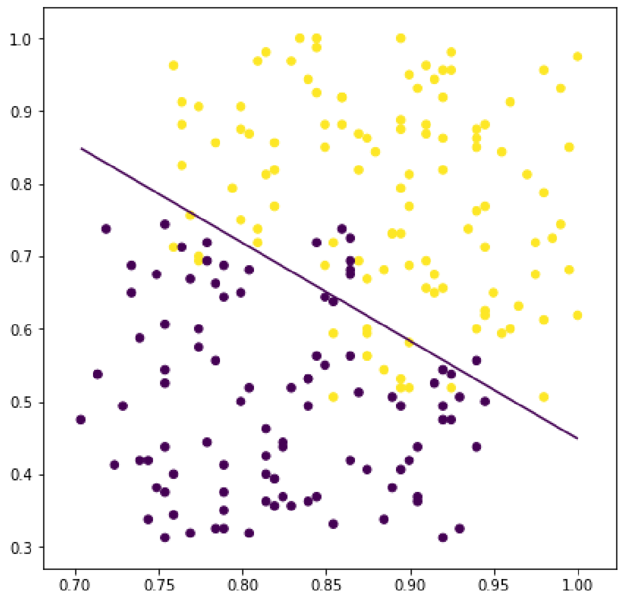


Figure 12

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

Although we can clearly see some mistakes misclassified close to the (0.85, 0.7) point, the accuracy is quite good, and if we tried to make the boundary account for those samples we might end up overfitting the model and not making it as generalized as it looks in the figure.

We have written a code with the ability of run the model on the GPU by setting a variable called "device" with the value "cuda:0" in the case that a compatible GPU was available. For this experiments we run the experiments on the free TPU that colab offers.

We also made experiments adding more layers and hidden units to the network and we achieved better results. This results can be reproduced by calling the class called *Custom_Neural_Network()*. Here we present the results:

- 2 hidden linear layers with 10 hidden units each one using Tanh activation functions in all the model but in the last layer where we left the Sigmoid function. The model was trained with SGD with a learning rate of 0.001 for 400 epochs:
    - Train accuracy: 89%
    - Test accuracy: 95%
- 2 hidden layers with 10 hidden units each one using ReLU activation functions in all the model but in the last layer where we left the Sigmoid function trained with SGD with a learning rate of 0.001 for 2000 epochs:
    - Train accuracy: 89%
    - Test accuracy: 93%

So in the end, although is not the norm, with this train and test example a more complex model is not giving us better results. This can be because the samples where our model is failing they're actually outliers and as our implementation it's not overfitting it performs bad on them. Maybe we could improve the performance of our model adding components like batch normalization and applying more regularization to our weights for example.