Deep Learning : Mini-Project 2

Massonnet Julien
Saini Anand Prakash

December 17, 2021

1 Introduction

The goal of this project is to design a small deep learning framework without using "torch.nn", autograd and over advance library.
We want to create a module that given a points in $[0,1]^2$ should determine if the points is in the disk of center $(0.5, 0.5)$ and radius $1/\sqrt{2\pi}$.
The network should have 3 hidden layer of 25 units and can work with 2 possible activation function (ReLU and Tanh). To train the network, we're using stochastic gradient descent.

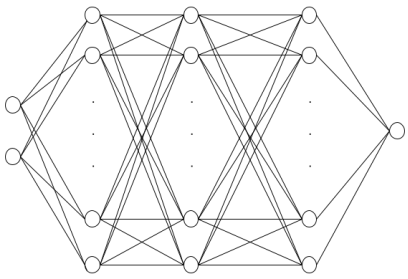

Fig : achitecture of the neural network

2 different Class

To implement such a network we're gonna need a few module, lets run through each one :

2.1 Linear

Linear is a simple module that takes an input of dimension "$in\_features$" and return an output of dimension $out\_features$.
It consist of doing the linear operation $y = x \cdot w^T + b$ where x is the input, y the output, w the weight and b the bias.
Hence it has w ($in\_features \times out\_features$) and b ($out\_features$) as parameters to train.
When doing back propagation, the module needs to compute every partial derivative of the loss for a given gradient ∇y of the output (∇k is the partial derivative of the loss with respect to k).

weight : We get the derivative of the weight as follow :

$$\nabla w = \nabla y \cdot x^T$$

We're doing the outter product between ∇y and x.

bias : We get the derivative of the bias as follow :

$$\nabla b = \nabla y$$

input : We get the derivative of the input as follow :

$$\nabla x = w^T \cdot \nabla y$$

So when doing bacwardation, we should have the gradient of the ouput for each training point as argument, and compute every gradient for each point, then return all the gradient with respect to the input.

2.2 Activation function

We have two possible activation function for this Framework, either ReLU or tanh.

ReLU : ReLU work as follow : it will take each component and take the maximmum between zero and its value. So the derivative of the output with respect to the input is 1 if the component is positive and zero otherwise.

Tanh : tanh is the hyperbolic tangent function apply component-wise so the derivative of the output with respect to the input is $1 - \tanh(x)^2$ for each input x.

To compute the derivative when backwarding we need to store the value of the input.

2.3 Loss function

The loss function is the mean square error (MSE). It also should store its input to compute the partial derivative.

2.4 Sequential

Sequential is a module that given a list of module will chain them so that when an input is given he will send it to the first module then the output is given to the second module and so on.
When we do a backwardation, it will ask the gradients of the last module (given an output) and then will feed these gradients to the module just before and so one (until the first module is reached).

2.5 Module

Module is the big class that correspond to our framework, at initialisation it will create a Sequencial with each layer (Linears + activation functions). For forward and backward, it's gonna pass the request to the Sequential.

2.6 optimizer

The optimizer will ask the Module for the parameters and gradients of the framework and update each of them. It will also ask for the reset of each gradient.

2.7 pseudo-code

We get the following pseudo code :

```
class Linear(object):
    def __init__(self,in_features,out_features):
        initial weight of size (out_features,in_features) with uniform law
        initial bias set to 0
        initial gradient to 0
    def forward(self,X):
        return X@self.weight.T+self.bias
    def __call__(self,X):
        store input for backwardation
        return self.forward(X)
    def backward(self,*gradwrtoutput):
        for each gradient g_i with respect to the output of images i,
         update the following gradient :
        gradient bias += g_i
        gradient weight += outer product between g_i
         and input store for image i
```

```
        gradient input : gradwrtintput append self.weight.T times g
        return gradwrtintput
    def zero_grad(self):
        reset the gradients to 0.
    def param(self):
        return a tuple of parameters and corresponding gradients

class Tahn(object):
    def forward(self,X):
        return X.tanh()
    def __call__(self,X):
        store X
        return self.forward(X)
    def backward(self,*gradwrtoutput):
        gradient of input : gradwrtoutput times derivative of tanh at X
        return gradwrtintput
    def zero_grad(self):
        nothing
    def param(self):
        return []

class ReLU(object):
    def forward(self,X):
        return max(x,0)
    def __call__(self,X):
        store X
        return self.forward(X)
    def backward(self,*gradwrtoutput):
        gradient is same of the output if x>0, 0 otherwise
        return gradwrtintput
    def zero_grad(self):
        nothing
    def param(self):
        return []

class MSE(object):
    def forward(self):
        return MSE(traget,pred)
    def __call__(self,target,pred):
        store target and pred
        return self.forward()
    def backward(self):
        return a list of gradient with respect of each images
    def param(self):
        return []
```

```
class Sequential(object):
    def __init__(self,*chain):
        store the chain
    def forward(self,X):
                out=X
        for each module M in the chain
                out = M(out)
        return out
    def __call__(self,X):
        return self.forward(X)
    def backward(self,*gradwrtoutput):
        gradwrtintput=gradwrtoutput
        for each module M in the chain starting from the end
                gradwrtintput = M.backward(*gradwrtintput)
        return gradwrtintput
    def zero_grad(self):
        for each module in the chain: reset their gradients
    def param(self):
        out=[]
        for each module M in the chain
            out append parameters of M
        return out

class Module(object):
    def __init__(self,*hidden_layers,in_features=2,out_features=1,ReLU=False):
        create a Sequential with given neurons at each layers
    def forward(self,X):
        return the evaluation of the sequential for X
    def __call__(self,X):
        return self.forward(X)
    def backward(self,*gradwrtoutput):
        ask Sequential for backwardation
    def zero_grad(self):
        tell Sequential to reset gradient
    def param(self):
        return the parameters in the Sequential

class OptimSGD(object):
    def __init__(self,method,learning_rate):
        store the method and learning rate
    def step(self):
        for each tuple (parameter,gradient) in the model
                make the gradient descent step
    def zero_grad(self):
        ask method to reset gradient
```

3 Results

Now runing this Neural Network on 1000 training data and test on 1000 test data where the data are normalized, we get the following graph for Mean Square Error (MSE):

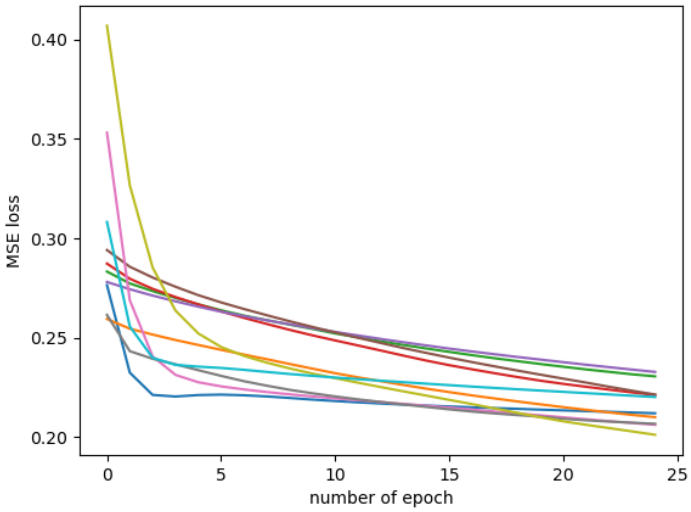

Fig : MSE loss for 5 with tanh and 5 with ReLU where lr=10^{-2}.

We see that with batch of size 100 the MSE loss drop at each iteration over all the data, it's still slow at the end of 25 epoch we're still around 30% error on the test set. For tanh we have a really bad MSE loss but it will quickly match the models with ReLU.

3.1 Tanh

We see that with smaller learning rate, the algorithm evolve smoother but takes more time to train. So we need to pick a learning rate that evolves quick but don't break.

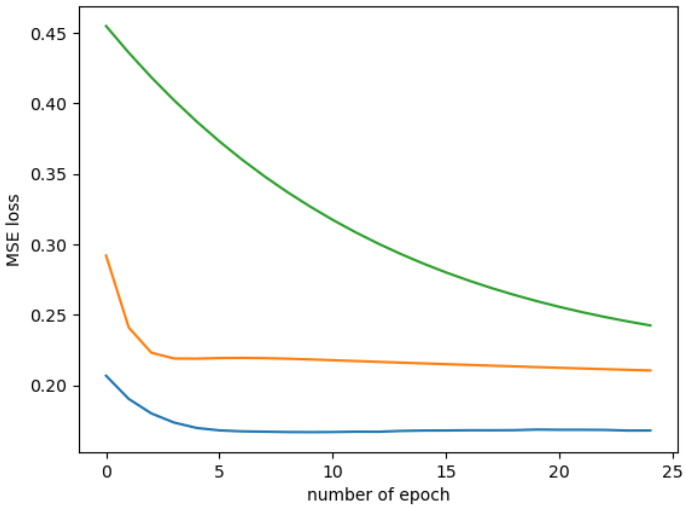

Fig : MSE loss using Tanh activation function.
bleu : $lr = 10^{-1}$*, orange :* $lr = 10^{-2}$ *and green :* $lr = 10^{-3}$

In the following graph, we run 10 times the model with tanh activation function and a learning rate of 10^{-1}, we see that the starting point as a lot of importance in the ending MSE loss but we still get bad step that destroy the model. The worst case has an error of 36% and the best has an error of 19.5% with a mean of error is 25% and a standard deviation of error is 4%.

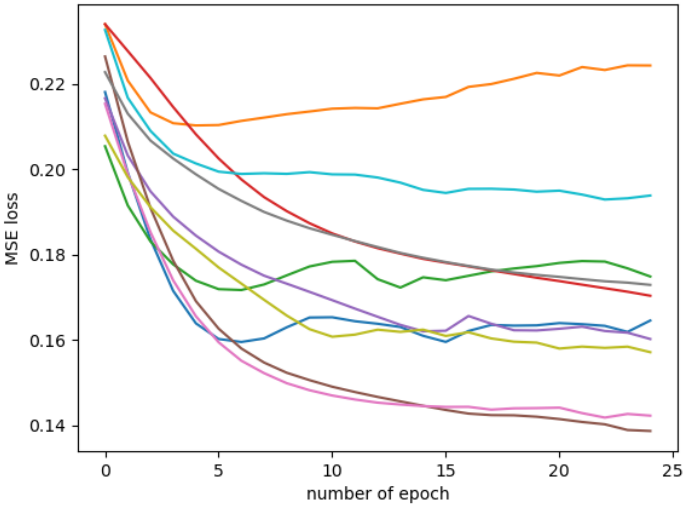

Fig : MSE of 10 run with tanh and lr=10^{-1}

In the following graph, we run 10 times the model with relu activation function and a learning rate of 10^{-2}. The worst case has an error of 32% and the best has an error of 24% with a mean of error is 27% and a standard deviation of error is 2%.

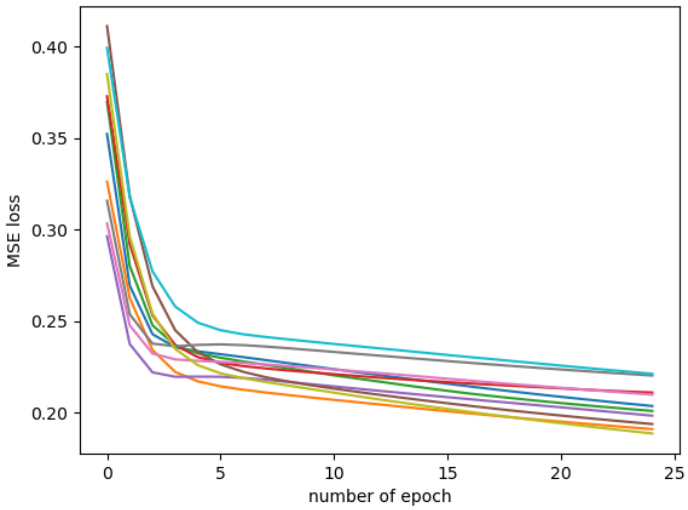

Fig : MSE of 10 run with tanh and lr=10^{-1}

3.2 ReLU

We get the same thing as tanh, we need a good learning rate that evolve quickly but stay stable.

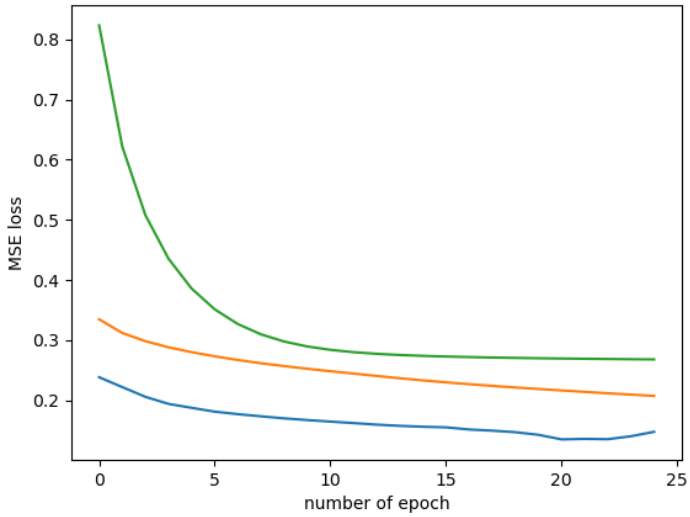

Fig : MSE loss using relu activation function.
bleu : $lr = 10^{-1}$*, orange :* $lr = 10^{-2}$ *and green :* $lr = 10^{-3}$

In the following graph, we run 10 times the model with relu activation function and a learning rate of 10^{-1}. The worst case has an error of 49% and the best has an error of 20% with a mean of error is 28% and a standard deviation of error is 7%.

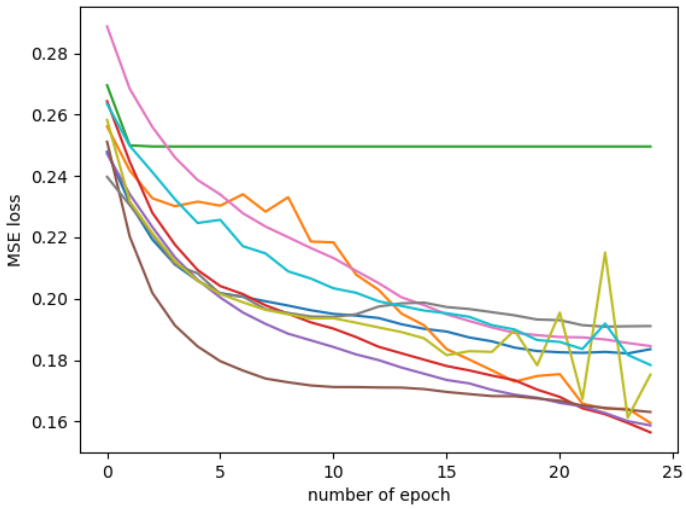

Fig : MSE of 10 run with relu and lr=10^{-1}

In the following graph, we run 10 times the model with relu activation function and a learning rate of 10^{-2}. The worst case has an error of 45% and the best has an error of 30% with a mean of error is 36% and a standard deviation of error is 6%.

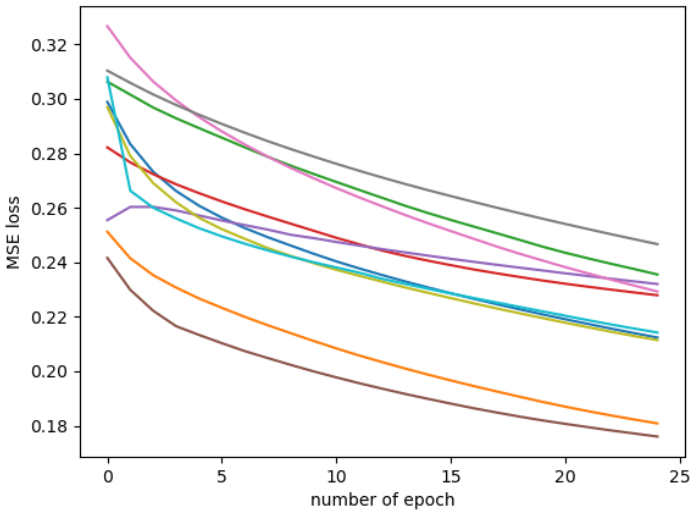

Fig : MSE of 10 run with relu and lr=10^{-2}

3.3 Conclusion

For both models, having a learning rate of 10^{-2} seems good, every run evolve the same way (smooth) and quick enough in comparison with a smallet learning rate.
Tanh has slightly better result than ReLU but ReLU is a much more simpler and faster activation function (for forward and backward) so when choosing a framework, we should have a execution time by performance ratio to take into account.