1 Framework

Each module implements the following functions :

forward(self, *input) The function computes the output from the given input and the parameters of the module.

backward(self, *gradwrtoutput) The function computes the derivative of the loss with respect to the input. In the case of a linear layer, in addition to the derivative, it accumulates the gradient of the loss with respect to its parameters. To compute it we can use the fact that the function takes as parameter the gradient of the loss with respect to the output and thus we have :

$$\frac{\partial l}{\partial s} = \frac{\partial x}{\partial s}\frac{\partial l}{\partial x}$$

where $\frac{\partial x}{\partial s}$ is the derivative of the forward function and $\frac{\partial l}{\partial x}$ is the derivative of the loss with respect to the output.

param(self) Returns a tuple of tenors containing the parameters of the model and their corresponding derivatives.

reset(self) Resets the gradient accumulation of the linear layer to 0.

update(self,eta) Updates the parameters of the module according to the gradient. The size of the step is given by the parameter eta defined by the user.
In addition to the modules that define each layer, we added a module called `Sequential` that contains all the modules of the network. It also has a function `forward` and `backward` that calls the forward/backward function for all its modules. This way when we want to use our framework we can simply add modules to our network and make steps by simply calling `forward` once.

The complete list of modules is the following:

- Activation layers:
  - ReLU
  - Tanh
  - Sigmoid
- Linear Layer
- Criterion :
  - Mean Absolute Error
  - Mean Square Error
  - Cross Entropy Loss

1.1 Cross Entropy Loss

In addition to the requested modules we decided to implement the cross entropy loss. Its forward expression is given by the following equation :

$$\mathcal{L}(x) = -\log\left(\frac{e^{x_i}}{e^{x_i} + \Sigma_{n\neq i} e^{x_n}}\right) \text{where i is the target class} \tag{1}$$

From this expression we can compute the derivative with respect to x to compute the backward path. Before continuing we have to consider two cases, the one in which the input is part of the target class and the one in which the input isn't part of the target class.

$$
\left.\begin{aligned}
\frac{\partial}{\partial x_i}\mathcal{L}(x) &= \frac{\partial}{\partial x_i}\left(-\log\left(\frac{e^{x_i}}{e^{x_i}+\Sigma_{n\neq i}e^{x_n}}\right)\right)\\
&= \frac{\partial}{\partial x_i}\left(-\log(e^{x_i})+\log(e^{x_i}+a)\right)\\
&= \frac{\partial}{\partial x_i}\left(-x_i\log(e)+\log(e^{x_i}+a)\right)\\
&= -\frac{\partial}{\partial x_i}x_i+\frac{\partial}{\partial x_i}\log(e^{x_i}+a)\\
&= -1+\frac{1}{e^{x_i}+a}e^{x_i}\\
&= \frac{e^{x_i}}{\Sigma_n e^{x_n}}-1
\end{aligned}\right| \quad \Sigma_{n\neq i}e^{x_n}=a \tag{2}
$$

In a similar way we can calculate the second case :

$$
\begin{aligned}
\frac{\partial}{\partial x_j}\mathcal{L}(x) &= \frac{\partial}{\partial x_j}\left(-\log\left(\frac{e^{x_i}}{e^{x_j}+\Sigma_{n\neq j}e^{x_n}}\right)\right)\\
&= \frac{e^{x_j}}{\Sigma_n e^{x_n}} \text{ as } \frac{\partial}{\partial x_j}x_i=0
\end{aligned} \tag{3}
$$

We thus end up with an output that consists of the tensor :

$$
\frac{\partial l}{\partial x} = \begin{pmatrix} \frac{\partial}{\partial x_1}\mathcal{L}(x) \\ \frac{\partial}{\partial x_2}\mathcal{L}(x) \\ \vdots \\ \frac{\partial}{\partial x_n}\mathcal{L}(x) \end{pmatrix}
$$

2 Test of the framework

After building the requested network, we trained it on the random dataset. After several tests we got a test error of 7.71% with a standard deviation of 0.0513.

If we have a look at the graph of the test error versus the number of epochs we can observe that depending on the starting weights, the error drops only after 50 epochs or later or even sometimes it gets slowed down on a local flat spot.

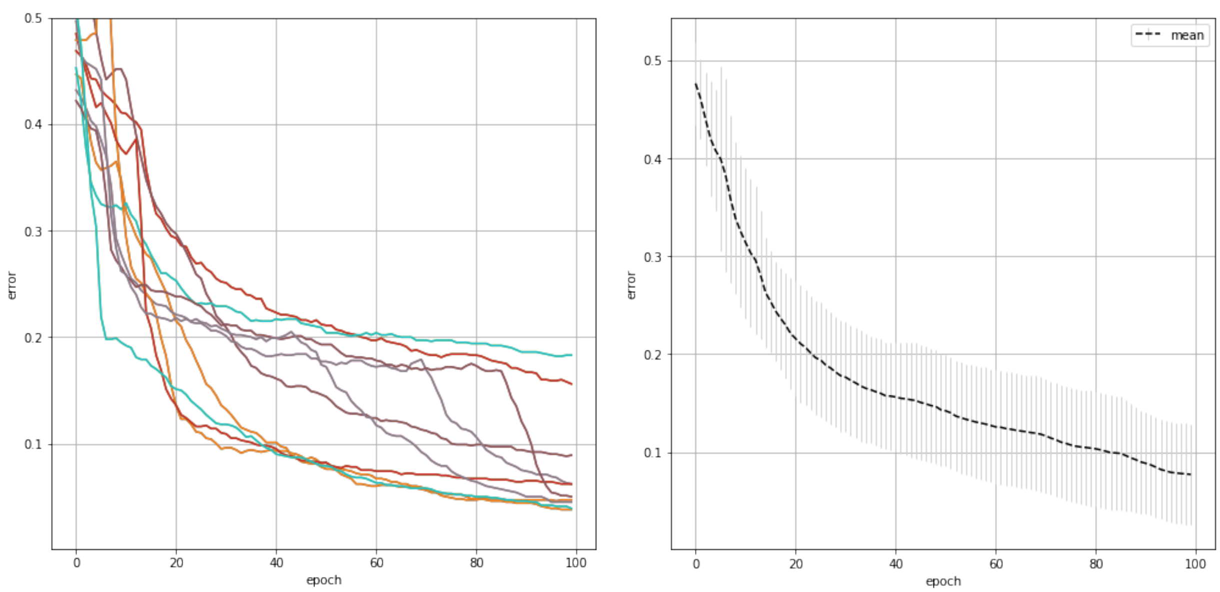


Figure 1: Test error versus number of epochs

If we have a look at the classification of the output we get can see that the classified circle isn't exactly round but it works pretty well.

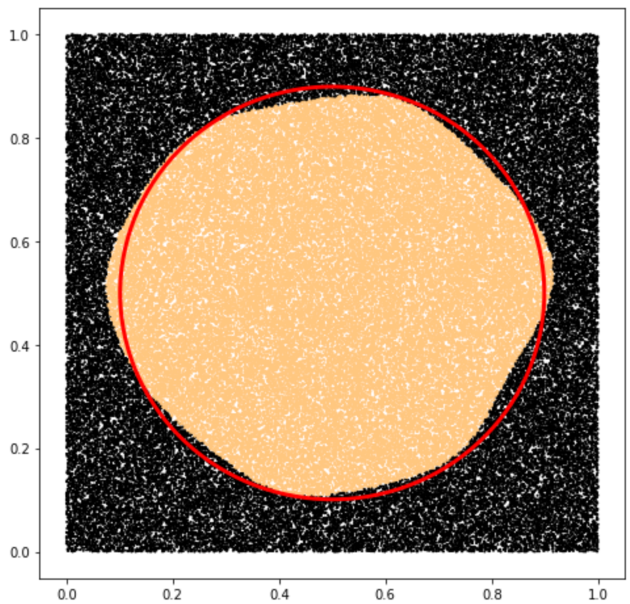
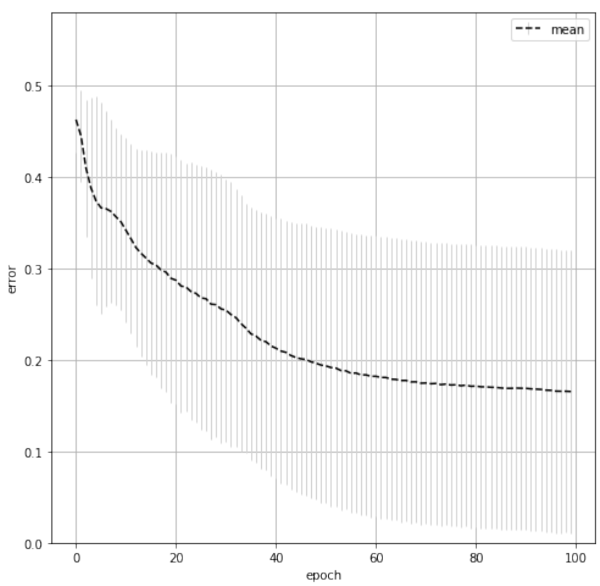


Figure 2: Output plot using MSE

We also tried with the Mean Absolute Error and the Cross Entropy Loss. We observed that MAE didn't work as well because the gradient descent got often stuck on a flat spot. With CEL (Cross Entropy Loss) we also achieved the best and most consistent results (minimal standard deviation of the error).



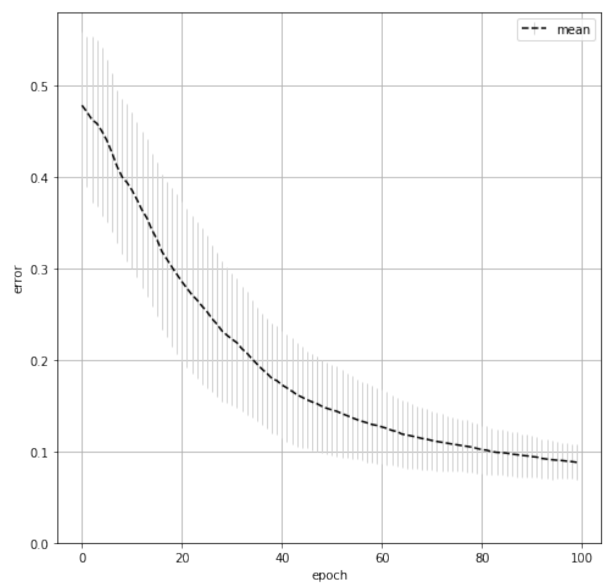


Figure 3: Test error versus number of epochs for MAE and CEL respectively