Mini Project 2

Deep Learning (EE-559), Prof. François Fleuret

May 22^{nd}, 2020

Name: PILLONEL Ken SCIPER: 270852
Name: PLASSMANN Jérémy SCIPER: 273908

Contents

Introduction

In this project, the goal is to create a mini deep learning library. This task can prove itself to be a real challenge and is a great educational tool to get a better and more profound understanding of neural networks.

1 Functionalities

1.1 Layer

A fully connected linear layer can be created by using the Linear object and specifying the number of input and output : *Linear(nb_input, nb_output)*.

- When an object *Linear* is initialized, the weights and biases are randomly generated following a uniform distribution $U\left[-\frac{1}{\sqrt{N_{nb\_input}}}, \frac{1}{\sqrt{N_{nb\_input}}}\right]$. The parameters are created in a new class called *Parameters* which holds the value of the parameters with the gradient of the loss with respect to these parameter.
- *forward*: this function computes the forward pass of a linear layer given an input.
- *backward*: this function calculates the gradients with respect to the parameters and holds them into the object *Parameter* thanks to the gradient with respect to the output calculated in the previous layer. It also computes the gradient with respect to the output in order to transmit to the next layer in the backward pass of the backpropagation algorithm.
- *parameters* : this returns the tensor parameters of the linear layer.

1.2 Activation functions

Three activation functions have been implemented:

- ReLU : *ReLU()*
- Tanh : *Tanh()*
- Sigmoid : *Sigmoid()*

These classes have two functions:

- The *forward* function which takes an input tensor and applies to it the activation function.
- The *backward* function which transmits to the next layer in the backpropagation algorithm the product of the gradient with respect to the output of the previous layer with the derivative of the activation function applied to the input tensor.

1.3 Losses

Two different losses have been implemented:

- Mean Squared Error : *LossMSE()*
- Binary Cross Entropy : *LossBCE()*

The Binary Cross Entropy has been implemented in this library because it is more adapted when the Tanh or Sigmoid activation functions are used as it makes the loss function convex.

- The *forward* function of the losses object computes the loss of the model on a dataset.
- The *backward* function of the losses object computes the first step of the backpropagation algorithm which is the derivative of the loss with respect to the output of the model.

1.4 Optimizer

A separate optimizer has been created. To use it, it has to be initialized as follows : *SGD(model.parameters())*. The *learning rate* can be specified while initializing the optimizer. The optimizer decreases the learning rate during the training according to a *decay* coefficient that can be also specified to the optimizer. With trial and error, the hyperparameters *learning rate* and *decay* were fine-tuned with different values for our two models described in Section 2.

The optimizer has then two functions:

- *zero_grad()* : this function will set all the gradient with respect to the parameters to zero.
- *step()* : this function computes one step of the optimization. It updates the weights and biases according to the gradients saved during the backward pass.

This class is used to hold all potential optimizer.

1.5 Models

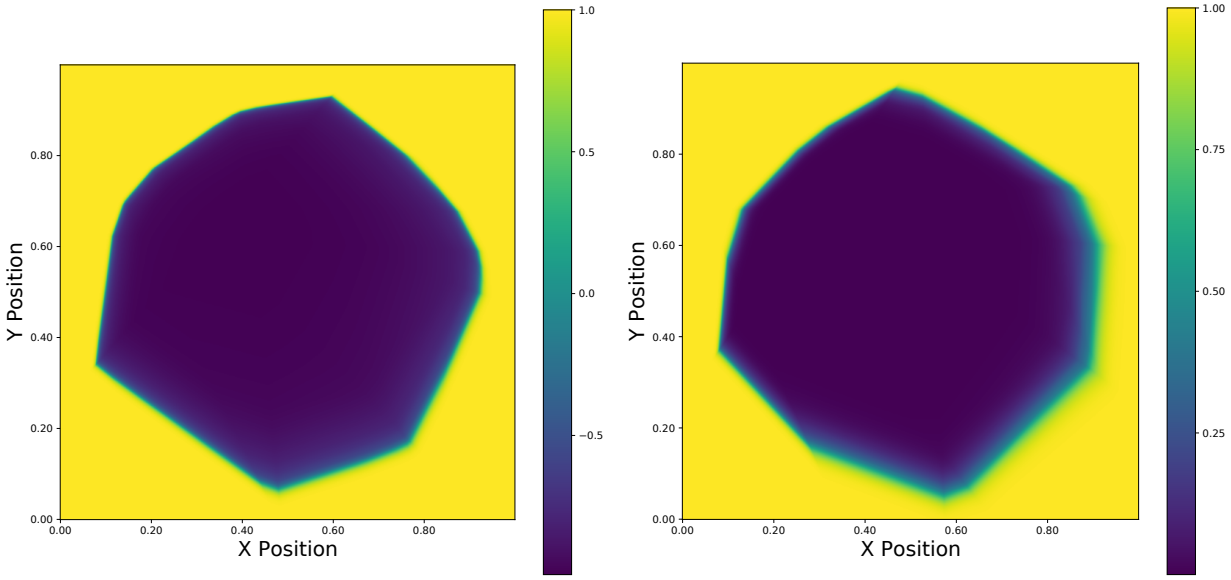
The class *Sequential* has been created in order to create a model by specifying sequentially the different layers and activation functions at the initialization. It can be used as follows: *Sequential(layer1, activationFunction1, ..., layerN, activationFunctionN)*. Its functions are:

- *forward* : this function computes the output of the model given an input tensor.
- The function *__call__* has been overwritten in order to compute the forward pass by calling *model(input)*.
- *backward* : this function triggers all the backward functions sequentially from the last layer of the model until the first one. All the gradients are computed and backpropagated.
- *parameters* : this returns a list of all the tensor parameters of the model.

2 Results

Using our framework, we created two simple models. Both of them have 5 *Linear* layers with 4 *ReLU* activation between them. The two models differ with their output activation function. The first model uses a Sigmoïd output activation function with the Binary Cross Entropy function (*LossBCE()*) to compute the losses. The second one uses the Tanh output activation function with the Mean Squared Error function (*LossMSE()*).

To demonstrate our framework, a dataset of 1000 points in a 2D space has been created. The goal of our neural net is to determine if the point belongs to the circle of center (0.5, 0.5) and radius $1/\sqrt{2\pi}$. You can see in figure 1 a color map of the neural network trained from that dataset.



(a) Sigmoïd Output Activation

(b) Tanh Output Activation

Figure 1: Model output when passed coordinates in the first quadrant over 10'000 epochs

We ran both models over 10'000 epochs. We first noticed that the Tanh model saturates in accuracy after roughly 3000 epochs to 97.7% accuracy. The Sigmoïd model saturates to 96% after roughly 6000 epochs. From our experience, we found that even very small changes when fine-tuning the hyperparameters (*learning rate* and *decay*) in the optimizer can lead to considerable changes in model accuracy.

3 Conclusion

In conclusion it was a great mini-project to help us get a better and more profound understanding of neural networks. By making our framework, we now have a better idea of what it takes to build such a tool and can show more appreciation and understanding towards the practicality and ease of use of libraries such as PyTorch.