Mini-Project 2 · Report

Deep Learning · EE-559

Tomas Larraz Giro ◇ Daniele Rocchi ◇ Giulio Trichilo

May 17, 2019

1 Introduction

The aim of this project is to implement and train from scratches a (sufficiently general) MLP in order to solve the following classification problem: predicting if a given test point (sampled uniformly in the space $[0,1]^2$) is inside or outside a given disk of radius $1/\sqrt{2\pi}$. Particularly, in order to do so we could not use pre-existing neural-network toolboxes as well as `pytorch` autograd functionality, which was globally turned off by setting `torch.set_grad_enabled(False)`. Hence, we first generated the training and testing dataset of 1000 samples each. This is implemented in the function `sample_generator`, which takes as input the number `dim` of samples and the `center` of the disk's radius, which is $(x, y) = (0.5, 0.5)$ in this case. The function returns two tensors: a 2×1000 tensor with the coordinates of the 1000 samples and another 2×1000 tensor with the label of each sample. Specifically, each row of this last tensor is either `[1,0]` or `[0,1]`, depending if the datapoint is respectively inside or outside the radius. The testing dataset is visualized in Figure 1b.

2 Network structure

Activation functions In order to construct a sufficiently general MLP, we created several modules that inherit from the suggested class `Module(object)`. Hence, we first coded the activation function class. Each class has 3 main components: `__init__(self)`, `forward(self, input)` and `backward(self, input)`. Specifically, the `forward` and `backward` methods are necessary, respectively, in the forward and backward pass of the back-propagation algorithm. In total, we programmed and tested 4 different activation functions:

- In the class `Linear(Module)` we implemented the fully connected layer. Hence, the `forward` method returns directly the tensor `input` without modifications and the `backward` method returns a tensor of same size as `input` but filled with 1s.
- In the class `ReLU(Module)` we implemented the ReLU activation. Hence, the `forward` method returns a tensor with components $\max\{0, x\}$ and the `backward` method returns a tensor with components equal to 1 if $x > 0$, 0 otherwise.
- In the class `Tanh(Module)` we implemented the tanh activation. Hence, the `forward` method returns a tensor with components $\tanh(x)$ and the `backward` method returns a tensor with components $1/\cosh^2(x)$.
- In the class `Sigmoid(Module)` we implemented the sigmoid activation. Hence, the `forward` method returns a tensor with components $1/(1+e^{-x})$ and the `backward` method returns a tensor with components $e^{-x}/(1+e^{-x})^2$.

Layers The class `Layer(Module)` is initialized with 3 external parameters: `in_nodes`, which defines the number of nodes on the left-hand side, `out_nodes`, which defines the number of nodes on the right-hand side, and `activ_fun`, which specifies the activation function to use. For instance, if we want to defined a layer with 5 nodes as input, 10 as output and ReLU activation function we would write `Layer(5, 10, ReLU())`. Hence, each time a layer is initialized, the class automatically instances a `in_nodes` $\times$ `out_nodes` dimensional tensor for the weights `w` and `out_nodes` dimensional tensor for the biases `b`. Specifically, the biases are initialized at 0, while the weights are initialized according to the Xavier initialization rule, i.e. by drawing a random number from a Normal distribution with mean 0 and variance $2/($`in_nodes` + `out_nodes`$)$. In addition, in `Layer(Module)`, two methods are implemented: `forward(self, input)`, which multiplies the `input` for the weights and adds the biases in the forward pass, and `backward(self, input)`, which directly returns `input`.

Loss function The class `MSELoss(Module)` defines the Mean Square Error loss criterion, necessary for training the network. In addition to the initialization, we defined two methods for this specific class: `loss(self, v, t)` and `dloss(self, v, t)`. The first one simply implements $\ell = \|\mathtt{v} - \mathtt{t}\|_2^2 = \sum_i (\mathtt{v}_i - \mathtt{t}_i)^2$, where `v` and `t` are the input tensor and the target tensor (using the same notation as in the practicals). The second one, returns the derivative of ℓ, i.e $2(\mathtt{v} - \mathtt{t})$. In addition, we implemented the Binary Cross-Entropy loss criterion in `CELoss(Module)`. The structure is almost the same as `MSELoss(Module)` with the difference that it returns $\ell = -t\log(y) - (1-t)\log(1-\sigma(v))$ in the `loss(self, v, t)` and $\frac{d}{dv}\ell = -\frac{t}{v} + (1-t)\frac{1}{1-\sigma(v)}\sigma'(v)$ in the `dloss(self, v, t)` method.

Sequential The class `Sequential(Module)` combines different layers into one unique network. Hence, it initializes for each layer a list with the derivatives with respect to weights and biases, namely `dl_dw` and `dl_db`, as well as two lists to store the data before and after applying the activation function (if specified). Hence, we implemented three methods: `forward(self, input)`, for the forward step, `backward(self, loss, target)`, for the backward step, and `print_structure(self)`, which prints on screen the network structure.

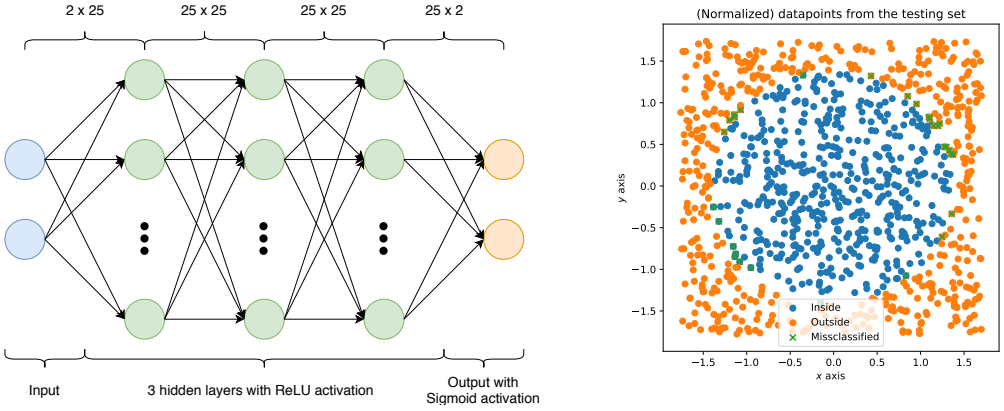
Optimizer The class `SGD(Optimizer)`, with `Optimizer(object)`, implements the gradient descent algorithm. Hence, after the initialization, necessary to set the learning rate η, we defined three methods: `optimize(self, model)`, which updates the weights and biases of each layer using the standard update step $w_{t+1} = w_t - \eta g_t$, where $g_t = \sum_{b=1}^{B} \nabla \ell_{n(t,b)}(w_t)$, `zero_grad (self, model)`, which initializes the gradients at zero (for a new iteration epoch), and `update_parameter(self,new_learning_rate)`, which updates the learning rate (in order to use a bigger rate at the beginning and a smaller at the end of the training).

Additional functions In order to train and test the MLP we defined the following functions: `training`, to train the model without mini-batches, `training_with_batches`, to train the model with mini-batches and `compute_accuracy`, to compute the accuracy during the training and testing.

3 Conclusions

First, we defined the following MLP, as shown in figure 1a:

```
model = Sequential( Layer(2,  25, ReLU()),
                    Layer(25, 25, ReLU()),
                    Layer(25, 25, ReLU()),
                    Layer(25, 2,  Sigmoid())
                )
```



(a) Network structure.

(b) Standardized dataset.

Figure 1: Visualization of the network structure (on the left) and standardized dataset (on the right). On the image on the right are shown in blue the datapoints inside the disk, in orange outside the disk and with a green cross the miss-classified.

Hence, after training the network with 2000 `epochs`, using `MSELoss()` as loss function, setting a `learning_rate` for the gradient descent equal to $5e^{-4}$, and applying a `mini_batch_size` of 100 on standardized data (i.e. subtracting the mean from the data and dividing by the standard deviation), we obtained the following results:

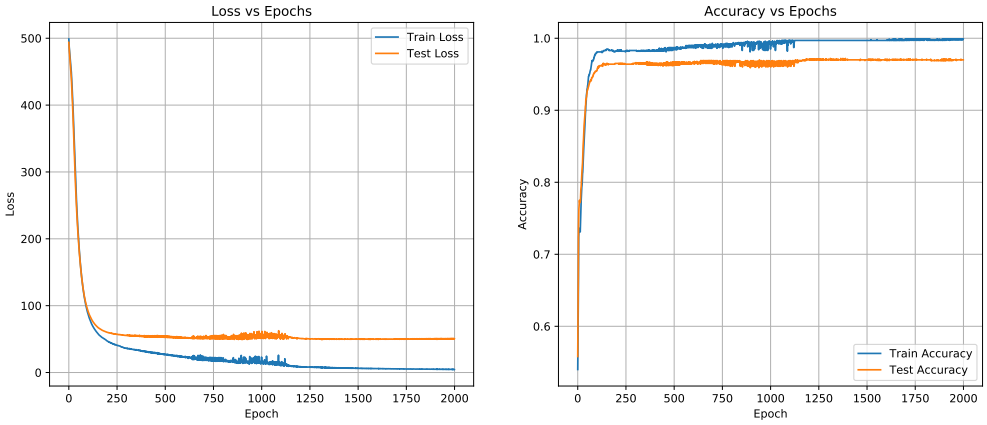


Figure 2: Loss and accuracy during training and testing as a function of the `epochs` using MSE loss.

As we can observe, with this specific structure of the network, it is possible to achieve up to 99.9% of accuracy during training and 97% during testing (which corresponds to 1 and 30 miss-classified datapoints respectively). In addition, the only miss-classified datapoints are the ones close to boundaries of the disk, as shown in Figure 1b. Similar results are achieved using the same setting (i.e same learning rate, number of epochs etc.) but Binary Cross-Entropy as loss function. Particularly, from Figure 2 we notice that between 0 and 250 epochs, the accuracy of the model increases significantly, while from 250 till 2000, the accuracy grows more slowly. Hence, with only 250 epochs our network is capable of reaching almost 95% accuracy during training and 90% during testing.