EE-559 Deep Learning

Spring Semester 2019

Development of Lamp : a small Deep Learning Framework

September 22, 2019

Project n°2

Sacha HAIDINGER *sacha.haidinger@epfl.ch*
Nils OLSEN *nils.olsen@epfl.ch*
Antoine SPAHR *antoine.spahr@epfl.ch*

Contents

1 Introduction

Lamp is a package written in python providing a simple deep learning framework. It gives the basic elements to build a multilayer perceptron (MLP). It implements three different activation functions (hyperbolic tangent, ReLU and SELU), three different loss functions (means square error (MSE), mean absolute error (MAE) and cross-entropy) and two gradient descent algorithms (stochastic gradient descent (SGD) and Adam). The package's operations are implemented with *pytorch* tensor object and the python built-in math library. However the deep learning machinery of pytorch is not used at any point. This report presents in the first instance the lamp framework and its features, as well as an example of use in the second part.

2 Lamp Structure

The lamp framework is composed of three main types of object : *Module*, *Sequential* and *Optimizer*. A visual overview of the framework is presented on figure 1.

2.1 Class Modules

Similarly as in pytorch, the modules are elements able to process an input forward and backward (*i.e.* provide the gradient of output with respect to the input of the module). The abstract mother class *Modules* forces any module object to implement a forward and a backward method. To build basic MLPs, three objects are build by inheriting from *Module* : *Linear*, *ActivationFunction* and *Loss*.

The *Linear* object defines a fully connected linear layer characterized by the number of input and output neurons. The *Linear* parameters are initialized with Xavier initialization.

The *ActivationFunction* object is an abstract class defining module representing an activation function. It requires the definition of two methods in the daughter classes : *sigma* and *dsigma* which represent the activation function and its derivative. Three activation functions are implemented : *Tanh*, *ReLU* and *SELU*.

The *Loss* object is also an abstract class defining a module which are loss functions. Similarly, it requires the definition of two methods on the daughter classes : *loss* and *dloss* that stand for the loss function and its derivative with respect to the output. Three losses are then defined : *LossMSE*, *LossMAE* and *CrossEntropyLoss*.

2.2 Class Sequential

The *Sequential* object defines the MLP. Its initialization requires a list of *Modules* and/or other *Sequential* objects. Since the modules are called sequentially, the order matters. Once created, the *Sequential* object orchestrates the forward and backward pass through all the modules: the output of one is the input of the next one. The *Sequential* object provides a method to recover all the model's parameters as a list of pairs (parameters , gradient of parameters).

2.3 Class Optimizers

The *Optimizers* object is an abstract class that defines how the gradient descent should be performed, hence how the learning should be optimized. This class enforces any daughter class to define a method *gradient_step* defining the behavior of the gradient descent. To be initialized, the *Optimizers* needs the list of parameters pairs (parameters and gradient) of the model to optimize (it is obtained by the *Sequential*'s method *parameters*). Two optimizers have been implemented : *SGD* and *Adam*.

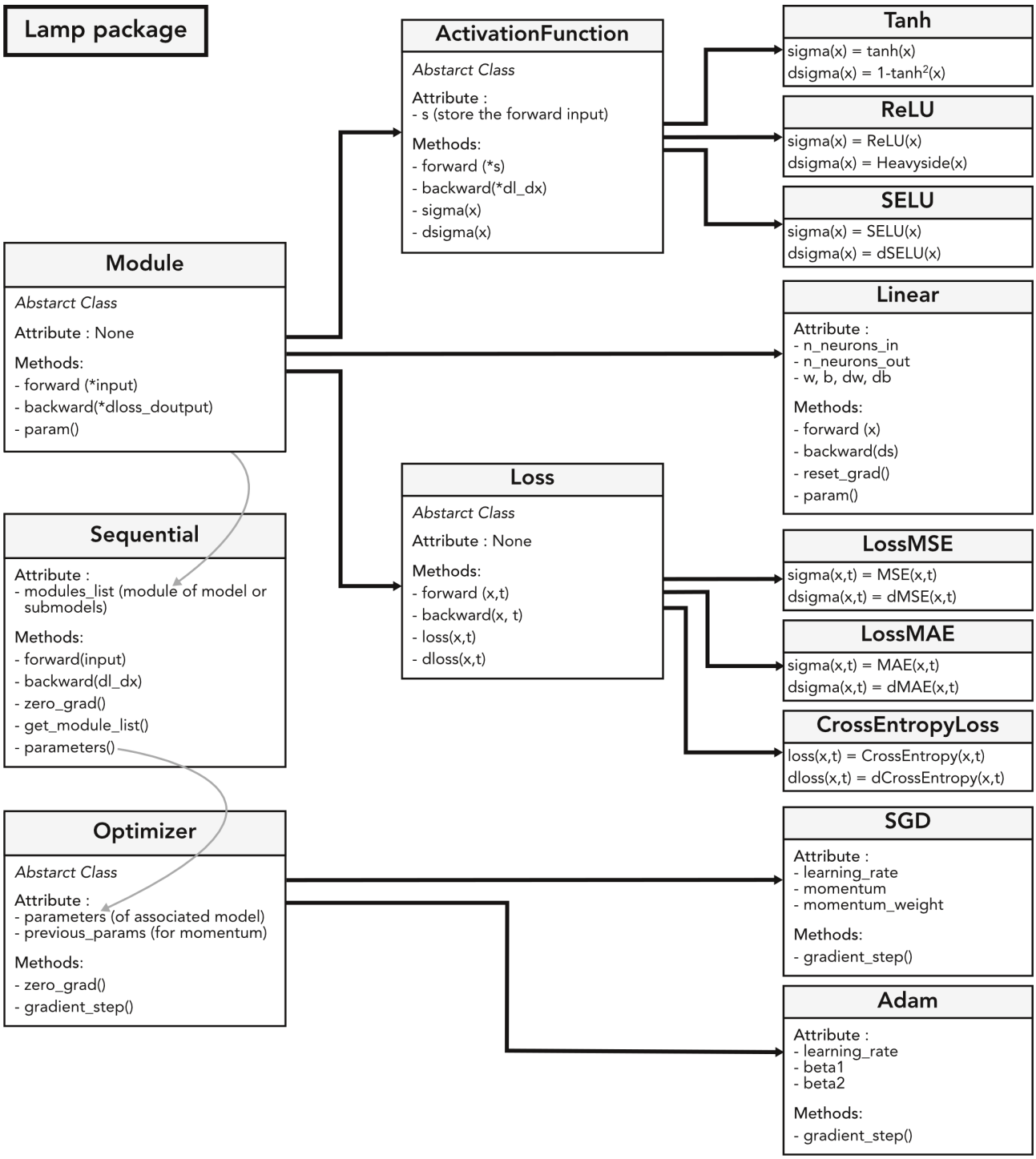

Figure 1: Lamp package structure. Each box represents a class. In the box, it is stated whether the class is an abstract class. The box also contains the methods and attributes defining the object. Black arrows highlight the inheritance of classes (the class at the end of the arrow inherits from the one at the beginning). The two gray arrows highlight the information transferred between the three main classes (*Modules*, *Sequential* and *Optimizers*).

The *SGD* performs the stochastic gradient descent (the momentum can be added optionally). *Adam* performs the optimization based on the Adam algorithm that take into account both the momentum and the velocity.

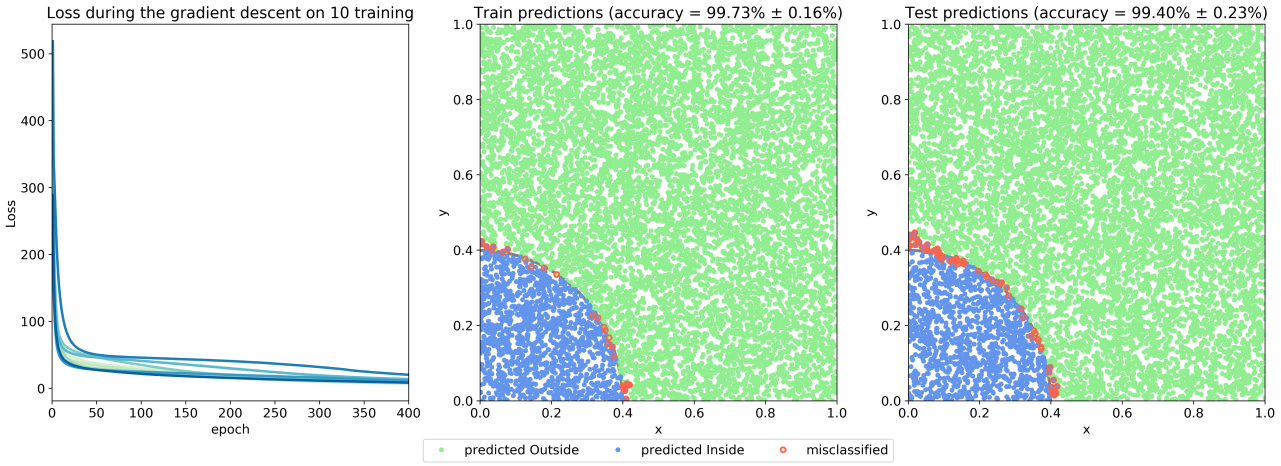

Figure 2: Example output. The left plot presents the loss evolution for 10 training procedure. The center plot presents the superimposition of the training prediction and mis-classifications over the 10 training procedure. The right plot presents the superimposition of the test predictions over the 10 training procedures. On the center and right plot, the prediction accuracy as mean ± std is displayed on the plot-title.

3 Example of use

3.1 Task

To illustrate the use of *Lamp* a simple learning task is implemented. The goal is to train a MLP in order to predict if a point uniformly sampled in $[0,1]^2$ is situated inside the disk of radius $1/\sqrt{2\pi}$ or not.

3.2 Model

To solve this task, the following model is proposed: a MLP with three fully connected hidden layers of 25 neurons each, SELU as activation function, the cross-entropy loss and an Adam optimizer. 1000 training and 1000 testing units are used with a minibatch size of 25. The training is performed over 400 epochs.

3.3 Implementation with lamp

The proposed model is implemented and trained as presented in the python code below.

```
# import lamp
import lamp.Modules as mod
import lamp.Sequential as seq
import lamp.Optimizers as optim

# get data with the proper function
train_input = get_data()

# build model
model = seq.Sequential(mod.Linear(2,25), mod.SELU(), \
                       mod.Linear(25,25), mod.SELU(), \
                       mod.Linear(25,25), mod.SELU(), \
                       mod.Linear(25,2))

# define the loss
criterion = mod.CrossEntropyLoss()
# define the optimizer
optimizer = optim.Adam(model.parameters())
```

```
# train
N_epochs = 400
mbs = 100 # mini_batch_size

for e in range(N_epochs):
    for b in range(0,train_input.size(0),mbs):
        # forward pass
        output = model.forward(train_input[b:b+mbs])
        # get the loss
        loss = criterion.forward(output, target[b:b+mbs])
        # set gradient to zero
        optimizer.zero_grad()
        # backward pass
        dl = criterion.backward(output, target[b:b+mbs])
        model.backward(dl)
        # perform the gradient descent
        optimizer.gradient_step()
```

3.4 Discussion

The outcome of 10 models training (loss evolution, and train/test performance) is presented on figure 2. The 10 times, the loss decreases in an expected fashion. The performances obtained in train and test also meet the expectancies, and the model clearly grasp the pattern of the disk.