MINI DEEP-LEARNING FRAMEWORK

ShiLin WANG shi-lin.wang@epfl.ch SCIPER: 294925
SiCheng XIE sicheng.xie@epfl.ch SCIPER: 319874
FuTong LIU futong.liu@epfl.ch SCIPER: 298089

Introduction

Being a powerful tool in the field of classification and regression for supervised machine learning, Deep-Learning has already a plenty of packages ready for implementation. These packages are common such as Pytorch, tensorflow etc. In this project, we aim to device a python-based Deep-Learning toolbox from scratch. It consists of some ordinary modules like Linear, ReLU, Tanh and so on. In the following sections, we will discuss the basic idea and test the designed structure. All the relevant files are available on github,

https://github.com/tim010007/mini-project-2020-DL-EPFL

1 Model Description

1.1 Basic Idea and Usage

A schematic representation of our toolbox design is shown as Figure-1. As one can notice, in our design, the toolbox can be divided into two parts, constructor and tool modules. A constructor can combine all the user defined operations into an automatic process and it records corresponding input, output and parameters at the same time. Tool modules are those operations common for Deep-Learning. Here we provide modules for **Linear, ReLU, Sigmoid, MSELoss, SGD**. They access constructor by taking it as argument, then do the calculation and update the attributes of it.

To put the toolbox into practice, first, a constructor and all the desired tools will be defined. After defining constructor, user can train the model manually by applying all the tool modules onto the constructor. Alternatively, user can also wraps desired operations into constructor and let it do the training automatically. This can be done by putting tool modules with desired arguments into the method **train** provided by constructor. Below, we show a simple example of two ways to train the model by our toolbox. After training, user can see the output, input and parameters by examining the attributes of constructor as shown in the last line.

```
from mynn_module import *
mynn = model()
```

1. Manual way:
```
lin = linear(2,25,mynn)
sig = sigmoid(mynn)
lin.forward(train_input)
sig.forward(train_input)
sig.backward(train_input)
lin.backward(train_input)
```

2. Automatic way:
```
seq = [ linear(2,25,mynn),
sigmoid(mynn)]
mynn.train(seq, input)
```

```
mynn.x
```

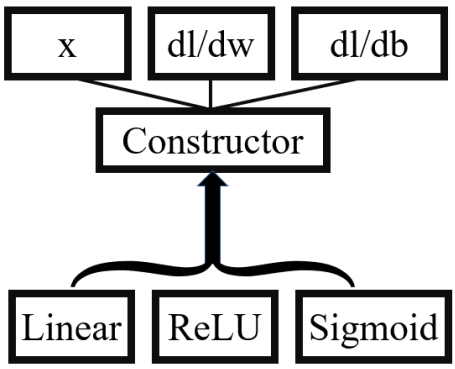

Figure 1: Graphical representation of toolbox

1.2 Modules Description

After a general description from section-1.1, we will briefly show how we implement modules in practice in this section. Detailed comments can be found in source code. Again, it should be noted that except from constructor module, all modules have to take constructor as argument to access parameters.

- Constructor Module **model** : **model** has several attributes including **x,w,b,dl_dw,dl_db** and etc. It also has methods for automatic processing. Method **train** takes arguments of sequence of user defined modules (operations) and do the forward / backward pass automatically based on current layer **ln**. First, for first time **train** been called, it will run **ini** method for each modules provided by user for initialization. Second, **train** will sequentially run the **forward** methods of these modules and **ln** will be updated by modules. Third, **backward** methods will be done with the same manner. If keyword **forward_only** is **True**, only the forward pass will be done. Finally, **train** will do forward pass until **ln**=**lt** and will do backward until **ln**=0. (To help automatic process, method **l_counter** is to denote total number of **linear** (**lt**) and will only be called during initialization.)
- Linear Module **linear** : **linear** takes **constructor**, **criterion**, input units and output units as arguments. It contains three methods. First is **ini**, which is automatically called when **train** is first time called. **ini** will create and append parameters to the corresponding attributes of constructor. For the rest, depending on current layer (**ln**), they will do the linear operations as below and update attributes of constructor.
  - For **forward** :
    1. x[ln+1] = w[ln+1] · x[ln] + b[ln+1]
    2. ln+=1
  - For **backward** :
    1. if (ln = lt) { dl_dx = **criterion.dloss**(x[ln]) } else { w[ln+1].T( ) · dl_ds[ln+1] }
    2. dl_ds[ln] = dl_ds[ln] $\otimes$ dl_dx[ln]
    3. dl_dw[ln] = dl_ds[ln] · x[ln-1]
    4. dl_db[ln] = dl_ds[ln]
    5. ln-=1
- Activation Module **sigmoid & relu** : **sigmoid** and **relu** take **constructor** as argument. Similar as **linear**, they have **forward**, **backward** and **ini** methods and work in the same principle that parameters and **ln** will be updated. The only difference are their operations.
- Criterion Module **criterion_mse** : **criterion_mse** has two methods, **loss** and **dloss** and it takes **constructor** as argument. Two methods calculate loss and gradient of Mean-Square-Error.
- Optimizer Module **optimizer_sgd** : **optimizer_sgd** implements Stochastic-Gradient-Descent method. This module should be called after each training. It updates parameters **w** and **b** in constructor depending on input learning rate (**lr**).

1.3 Input and Target

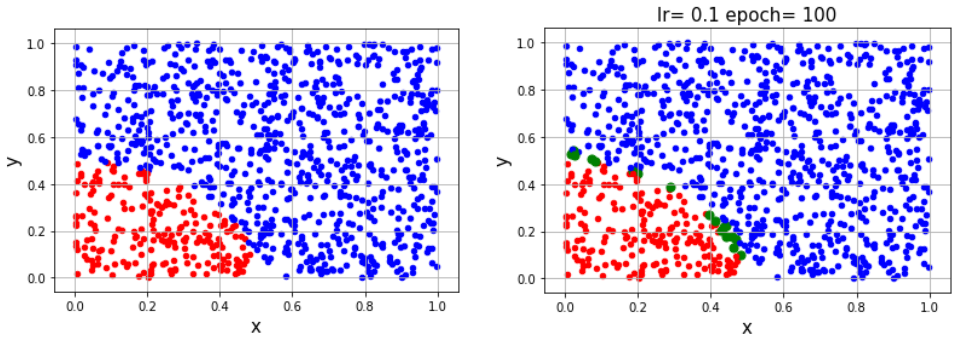

Figure 2: (left) Plotting of randomly generated data set. For color of points in the dist with raius of 0.5, they are red. Otherwise, they are blue. In the calculation, we present them as $[0, 1]$ and $[1, 0]$ instead. (right) Result of testing data set after 100 epoch with learning rate lr=0.1. Here the green points are points with error prediction.

In this project, we randomly generate two sets of 2D coordinates $\{x_i, y_i\}^N \in R$, each has $N = 1000$ points and $\forall(x_i, y_i)$, $x_i \in [0, 1] \wedge y_i \in [0, 1]$. They are taken as input for training and testing respectively. As for target, we use the one-hot encoded style. For those points reside outside or just on edge of the disk of radius $1/2$, the target will be $[0, 1]$. Contrarily, for those points inside the disk of radius $1/2$, the target will be $[1, 0]$. A simple representation will be as Figure-2-(left).

2 Result and Discussion

We test our designed toolbox on the standard model as indicated by handout. The model is made of 3 fully connected (FC) layer. Each layer has 25 hidden units and each is activated by sigmoid function. First we run a simple test with learning rate lr=0.1 and lr=0.001. The training results are shown as Figure-3 and the plot is shown as Figure-2-(right). We do a series of run based on various learning rates. Results are listed in Table-1. As one can notice, there is a optimum learning rate based on given structure, lr=0.1 for this case.

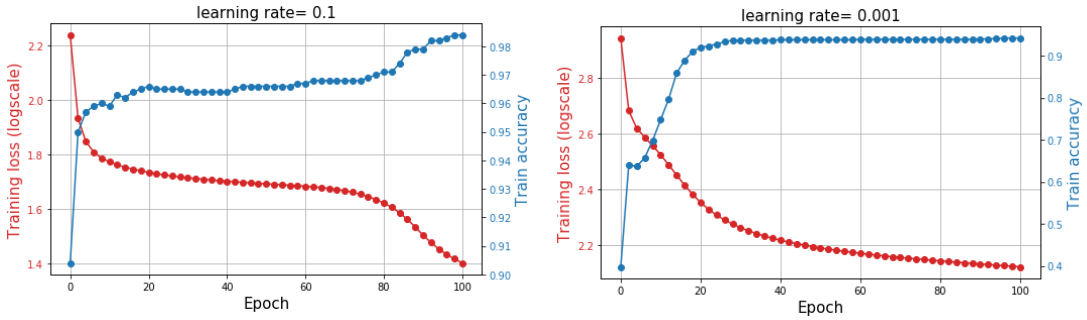

Figure 3: Training result (left) w/ lr=0.1 (right) w/ lr=0.001

	lr=10	lr=1	lr=0.1	lr=0.01
Total loss ($\log_{10}$)	2.4 ($\pm$ 0.18)	1.4 ($\pm$ 0.1)	1.386 ($\pm$ 0.17)	1.724 ($\pm$ 0.05)
Test accuracy (%)	87 ($\pm$ 6)	98 ($\pm$ 0.8)	99 ($\pm$ 0.8)	96 ($\pm$ 0.4)
Total wall time (sec)	77.71 ($\pm$ 4.5)	80.09 ($\pm$ 1.5)	84.43 ($\pm$ 2.9)	79.91 ($\pm$ 5.5)

Table 1: Training results after 100 epoch with various learning rates w/ 10 samples each

We also test a different structure which composes of 4 layers and 1 with **relu** shown as Figure-4. The testing accuracy is pushed further toward 100% and it is not an over-fitting. In conclusion, our designed toolbox is flexible to create arbitrary structure and can be fully automatic. More functionalities (such as the weight initialization, convolution and etc) can be improved.

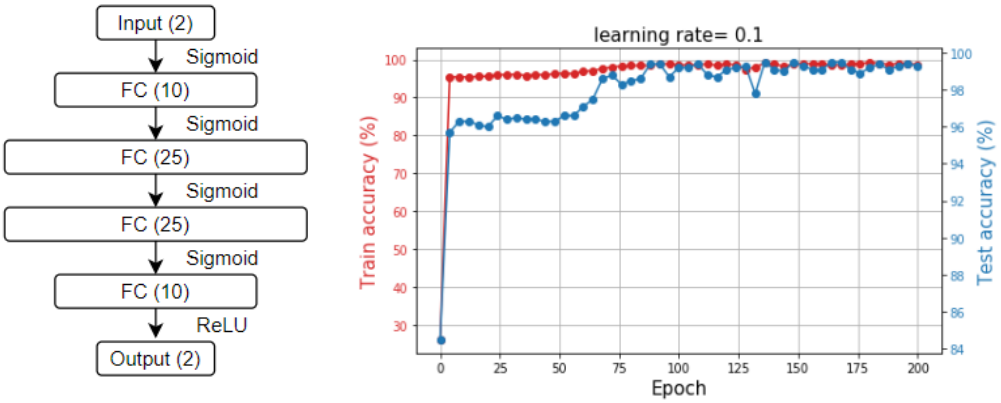

Figure 4: (right) Training Result w/ (left) different structure and lr=0.1