Project 2 - Mini Deep Learning Framework

Axel Dinh Van Chi[1] **and Nino Herve**[1]

[1]EE-559 Deep Learning, Ecole Polytechnique Fedérale de Lausanne

The aim of this project is to design a mini 'deep learning framework' using only pytorch's tensor operations and the standard math library, hence in particular without autograd or the neural-network modules. The objective we fixed ourselves is to design a framework capable of:

- **Creating fully connected Networks with desired number of nodes and layers**
- **Apply different activation functions**
- **Apply different losses**
- **Apply different weight initializations**
- **Apply dropout**
- **Choose between different optimizers**
- **Use a scheduler**

All the code is available here.
This report will go through the results after training models with various functionalities, providing small coding examples along.

Introduction - First Network

All functionalities (except for optimizers and the scheduler) inherit from a class called 'Module'. Each module implements its own forward and backward pass. A network is simply a series of modules. Weights and gradients are stored in each module and can be accessed through module_name.param(). To define a network as one entity, modules can be regrouped in a 'Sequential' also inheriting from Module. To update the parameters of the model, an optimizer is required. The optimizer stores references towards the parameters (weights and gradients) of the model. When calling optimizer.step(), all weights get updated with the last computed gradients. Gradients accumulate as long as the model.zero_grad() function is not called.
Here is a small example of what was summarized.

```python
from modules import Linear, Sequential
from activations import ReLU
from optimizer import SGD

# Create model
model = Sequential([Linear(..., 25),
                    ReLU(),
                    Linear(25, 1)])
criterion = LossMSE()
optimizer = SGD(model.param())

# Forward pass
model.forward(X)
loss = criterion(X)

# Refresh gradient buffer if needed
model.zero_grad()

# Backward pass
dl = loss.backward()
du1 = model.backward(dl)

# Update weights
optimizer.step()
```

Listing 1. Python example

Using the previous components, it is possible to generate a simple network with two input units, one output unit, three hidden layers of 25 units, ReLu as activation function and train it with MSE and SGD for 200 epochs. To test this model (and all models to come in this project) dummy data are required. Samples are generated as 2D uniformly random points with elements between 0 and 1. Each point is labeled as 1 if it is contained in the circle of center (0.5, 0.5) and radius $\frac{1}{\sqrt{2\pi}}$. Figure 1 shows that training the model with this data successfully reduces loss over the epochs.

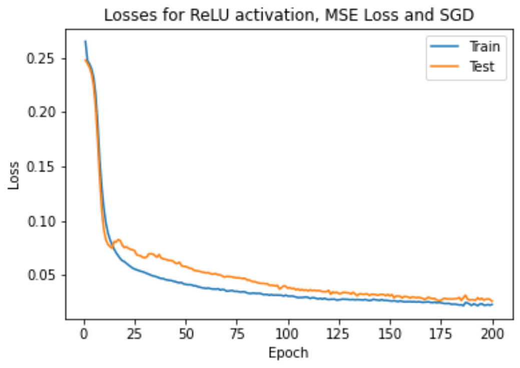

Fig. 1. *Training session of first model*: loss of train set and test set over 200 epochs

For the following sections, all networks will consist of two input units, three hidden layers of 25 units and 200 epochs. The rest might vary but will be clarified when needed.

Activation Functions and Weight Initialization

The framework contains two activation functions (Table 1) and three different weights initialization methods (Table 2).

Name	Function	Derivative
ReLU	$f(x) = max(0, x)$	$f'(x) = 1$ if $x > 0$ and 0 otherwise
Tanh	$f(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$	$f'(x) = 1 - tanh(x)^2$

Table 1. *Activation functions of the framework.* Functions are applied during the forward pass while derivatives are used during the backward pass.

Name	Distribution	Parameters
Default	$w_{ij} = Uni(-a, a)$	$a = 1/\sqrt{l}$
Xavier normal	$w_{ij} = N(0, std^2)$	$std = gain/\sqrt{l+m}$
He	$w_{ij} = N(0, std^2)$	$std = gain/\sqrt{l}$

Table 2. *Weight initialization* where l and m correspond to the number of inputs and number of outputs of the layer. The gain is a value in $\{1, \frac{5}{3}, \sqrt{2}\}$ depending on whether the activation function of the layer is None, Tanh or ReLU respectively.

Implementing activation functions is shown in Listing 1. To apply a different weight initialization than the default version, a gain is needed, it is defined as a value in $\{1, \frac{5}{3}, \sqrt{2}\}$ depending if the activation is, respectively, None Tanh or ReLU. Then to initialize the parameters of the model simply use init_parameters on any Module instance with 'default', 'xavier' or 'he' as method argument and the proper gain (see Listing 2).

```
model.init_parameters(method='xavier', gain = 1.)
```

Listing 2. Example of weight initialization

Using different initialization methods and different activation functions changes the convergence speed of training and can also impacts the final error rate of the model. Figure 2 summarizes the latter for various combinations after training with MSE and SGD. It is for example possible to see that, for this project's problematic, ReLU seems to have better scores on average than Tanh.

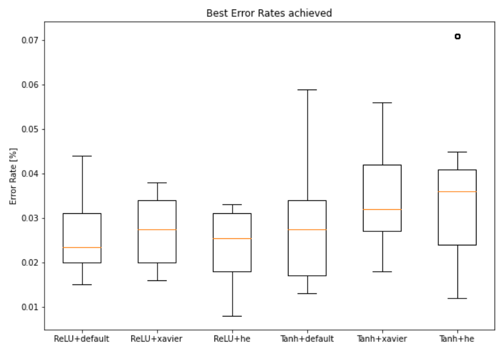

Fig. 2. *Testing error rates for different activation functions and weight initialization.* Training was done with MSE and SGD, no dropout and no scheduler. 10 trials were run for each model to obtain mean and variance

Losses and Optimizers

In this framework there are two available losses: the Mean Squared Error (MSE) and the Cross Entropy (CE).

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

where Y_i and $\hat{Y}_i$ are respectively the target value and the estimated value of the network.

$$CE = \frac{1}{n}\sum_{i=1}^{n} -\log\frac{e^{A_{class(i)}}}{\sum_{j=0}^{c} e^{A_j}}$$

where $A_{class(i)}$ is the network's output for the correct class of sample i.

Moving on to optimizers, it is possible to optimize parameters with either Stochastic Gradient Descent (SGD) or Adam. SGD uses the momentum,

$$v_t = \gamma v_{t-1} + \eta * \nabla_w gt$$

where $g_t = \frac{1}{N}\sum_{n=1}^{B}\nabla_w L$. γ and η correspond to hyper-parameters (η being the learning rate), L is the loss and B the batch size. In order to update weights as such :

$$w^{new} = w^{old} - v_t$$

Similarly, Adam implements its own moments :

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2$$

where g_t is as before and β_1 and β_2 are new hyper-parameters.
An additional step in Adam is required to correct the bias before updating the weights.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_1^t}$$

And the update is slightly different:

$$w^{new} = w^{old} - \frac{\eta}{\sqrt{v_t}+\epsilon} m_t$$

ϵ is a smoothing term to avoid division by 0.
Lastly, it is also possible to add an L2-penalty during the training, to do so a further step is necessary:

$$w^{penalized} = w^{new} - \eta\lambda w^{old}$$

where λ is the L2-penalty coefficient. This has the effect to regularize the training, hence avoiding overfitting. Given the predictable behaviour of the data, we decided not to finetune this parameter as it is very unlikely to observe overfitting.

Different error rates were obtained for the various losses and optimizers which can be compared in figure 3. This figure tells us that Adam leads to better accuracies than SGD on average. However, no evident differences can be seen between MSE and CE.

Scheduler

A reduce-on-plateau-like scheduler was added to optimize the learning rate during training. The function reduces the learning rate of the optimizer each time the loss stops improving for a specific number of epochs. This allows for the loss, when bouncing around the minimum, to take smaller

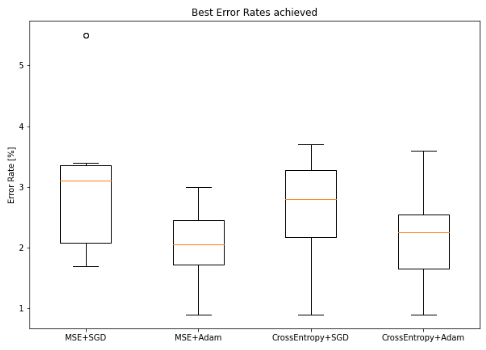

Fig. 3. *Testing error rates for different losses and optimizers.*: Training was done with ReLu and default weight initialization, no dropout and no scheduler. 10 trials were run for each model to obtain mean and variance

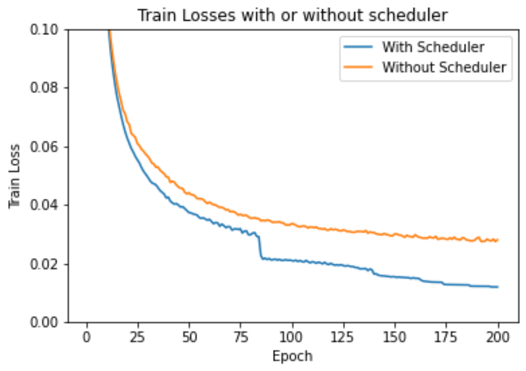

Fig. 4. *Training loss for model with and without a scheduler.*: Training was done with ReLU, default weight initialization, Cross Entropy and Adam.

steps during updates and make finer optimization. Small improvements, looking like steps, can be observed in Figure 4, which correspond to epochs were the learning rate got reduced. Thanks to the scheduler, most models were able to increase their final scores.
To implement the scheduler, instantiate it by assigning the optimizer of your model. This way the scheduler can access the learning rate of the optimizer and reduce it. The functionality has to be called after each epoch.

```
from scheduler import ReduceOnPlateau

# Instantiate
scheduler = ReduceOnPlateau(optimizer, patience
    =10, reduce_coef=0.5)

# Scheduler step after each epoch
scheduler.step(loss)
```

Listing 3. Scheduler example

Dropout

The Dropout module filters a layer by randomly setting nodes to 0. To decide whether a node should be set to 0 or not, a Bernouilli distribution with probability $1-p$ is used, the node is eliminated if the output is 0. Hence each new forward pass, the nullified nodes are different. Furthermore, the outputs are multiplied by $\frac{1}{1-p}$ to preserve the expectation of the outputs.

Two functions, model.train() and model.eval(), have been implemented so that the user can activate dropout when training and deactivate it during evaluation.

```
from dropout import Dropout

p=0.5 # Probability of skipping a node

model = Sequential([Linear(2,25),
                    activation(),
                    Dropout(p),
                    Linear(25,out_features)])

model.train() # If training
model.forward()

model.eval() # If evaluating
model.forward()
```

Listing 4. Dropout example

The main purpose of dropout is to avoid overfitting. Since no overfitting was observed for any model, there was thus no significant differences between final error rates when using dropout or not. However it is still possible to assess the implementation of dropout due to higher fluctuation during training (refer to Figure 5). This is because, when computing the loss, the model.train() option was activated, skipping 50% of the nodes. At each epoch, different nodes were selected resulting in slightly different models and therefore inducing the higher fluctuation.

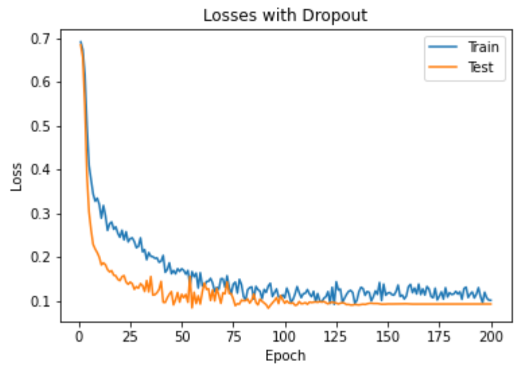

Fig. 5. *Testing loss for model with dropout.* Training was done with ReLu, default weight initialization, a scheduler, Cross Entropy and Adam. Dropout was activated during loss computation with a probability of 10% to skip a node.

Conclusion

In this project, we successfully implemented a small deep-learning framework able to make simple neural-networks, and showed that it could get good performance on simple datasets.
To go further, more modules could have been implemented, such as batch-normalization or convolutional layers. Sadly, the latter would have asked us to revise our implementation of the backward pass and we could not implement it in time.