EE 559 : Deep Learning
Mini-project II

Armand BOSCHIN Quentin REBJOCK Shengzhao LEI

***Abstract*—In this paper, we present a mini deep-learning framework using only Pytorch's tensor operations and the standard math library, hence in particular without using autograd or the neural-network modules. The aim is to be able to build and train a multi-layer perceptron capable of classification.**

The structure of the paper is the following : after a short introduction in section I, we present the data set in section II, the algorithms used in section III, the structure of the code in section IV, the improvements of the framework in section V and eventually we discuss the results in the last section.

I. INTRODUCTION

The goal is to implement a small deep-learning framework using only Pytorch's tensor operations and the standard math library. We want this framework to be able to build and train a multi-layer perceptron capable of classification. More precisely the framework must provide the necessary tools to:

- build networks combining fully connected layers and activation (Tanh, and ReLU) layers
- train this network using stochastic gradient descent

II. DATA SET

The data used is randomly generated. The function `build_data` of the framework (in `utils.py`) generates a set of 1000 points sampled uniformly in $[0,1]^2$, each with a label 1 if inside the disk of radius $\frac{1}{\sqrt{2\pi}}$and 0 otherwise. We can then generate a train, a validation and a test set easily. This data is used mainly to test the framework.

III. TRAINING THE NETWORK

In order to train our network, we use the back-propagation algorithm to perform a stochastic gradient descent. Those two algorithms are presented here.

Notations :

- $\hat{y_i}$ and t_i are the prediction and target vectors for sample i (using one-hot encoding).
- $\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} ||\hat{y_i} - t_i||^2 = \frac{1}{n}\sum_{i=1}^{n} \ell_i$ is the mean squared error
- $s^{(l)}$ is the summation of hidden layer l before activation.
- $x^{(l)}$ is the activated summation of hidden layer l. We have input $x_i = x^{(0)}$ and $\hat{y_i} = x^{(L)}$ where L is the number of layers.

A. SGD

In stochastic gradient descent, the true gradient of $\mathcal{L}(w)$ is approximated by a gradient at a single training sample in order to make the weight update.

$$w \leftarrow w - \eta \nabla \ell_i(w)$$

As the algorithm sweeps through the training set, it performs the above update for each training sample. Several passes (epochs) are made over the training set until the algorithm converges and at each epoch, the training set is shuffled to prevent cycles. Moreover, the learning rate η decreases at each epoch.

Algorithm 1 SGD

```
procedure SGD(w_0, η)                    ▷ initialized parameters
    while stopping criteria not met do
        Randomly shuffle samples in the training set
        for each sample x do
            w ← w − η∇ℓ_i(w)
```

In order to use SGD, we see that we need to be able to compute the gradient of ℓ_i with respect to any parameter of the network. To that prospect we use the back-propagation algorithm.

B. Back-propagation

The goal of back-propagation algorithm is to compute the gradient of the loss with respect to all parameters of the model and update these parameters (cf. SGD). The idea is to compute the gradient with respect to a layer's weights using the gradient with respect to the output of this layer and "pass up" the gradient with respect to the input. The algorithm can be decomposed in three steps.

Step 1 : Forward pass : Give a sample x_i as input to the network and send it forward in the network in order to compute the summations and activations all the way up to the output. For each layer $l \in [1, L]$, we have $s^{(l)} = w^{(l)}x^{(l-1)} + b^{(l)}$ and $x^{(l)} = \sigma(s^{(l)})$ where σ is the activation function. $w^{(l)}$ is a tensor of shape (output size of layer l $\times$ input size of layer l) and $b^{(l)}$ a tensor of shape (output size of layer l).

Step 2 : Backward pass : The goal of this step is to compute the derivative of the loss with respect to all the parameters. A clever use of the chain rule allows us to compute those gradients layer by layer starting by the last one and going backward. In this step, each layer l receives the tensor $\delta^{(l)} = \left[\frac{\partial \ell}{\partial x^{(l)}}\right]$, uses this $\delta^{(l)}$ to compute $\left[\frac{\partial \ell}{\partial w^{(l)}}\right]$ and $\left[\frac{\partial \ell}{\partial b^{(l)}}\right]$ and passes $\delta^{(l-1)}$ up to the previous layer after computing it. As shown in [2], we have the following equations (where $\odot$ stands for the Hadamard product of two

tensors). Note that shapes of tensors are indicated as indexes for consistency check (i and o mean respectively input size and output size of the layer l).

Notations : If l is an activation layer, its input is $s^{(l)}$ and its output is $x^{(l)}$. If l is a linear layer, its input is $x^{(l-1)}$ and its output is $s^{(l)}$.

Derivatives with respect to summations and activations:

$$\left[\frac{\partial \ell}{\partial x^{(L)}}\right]_{(2,1)} = \nabla \ell(x^{(L)}) \tag{1}$$

$$\forall l > 0, \left[\frac{\partial \ell}{\partial x^{(l-1)}}\right]_{(i,1)} = (w^{(l)})^T|_{(i,o)} \left[\frac{\partial \ell}{\partial s^{(l)}}\right]_{(o,1)} \tag{2}$$

$$\left[\frac{\partial \ell}{\partial s^{(l)}}\right]_{(o,1)} = \left[\frac{\partial \ell}{\partial x^{(l)}}\right]_{(o,1)} \odot \sigma'(s^{(l)})|_{(o,1)} \tag{3}$$

Derivatives with respect to parameters

$$\left[\!\left[\frac{\partial \ell}{\partial w^{(l)}}\right]\!\right]_{(o,i)} = \left[\frac{\partial \ell}{\partial s^{(l)}}\right]_{(o,1)} (x^{(l-1)})^T|_{(1,i)} \tag{4}$$

$$\left[\frac{\partial \ell}{\partial b^{(l)}}\right]_{(o,1)} = \left[\frac{\partial \ell}{\partial s^{(l)}}\right]_{(o,1)} \tag{5}$$

Step 3 : Gradient Step : Update all the parameters of the model using the gradients computed in the backward pass.

$$w^{(l)} \leftarrow w^{(l)} - \eta \left[\!\left[\frac{\partial \ell}{\partial w^{(l)}}\right]\!\right] \tag{6}$$

$$b^{(l)} \leftarrow b^{(l)} - \eta \left[\frac{\partial \ell}{\partial b^{(l)}}\right] \tag{7}$$

IV. Structure of the code

The framework's code is organized in various classes. We first present `Module` which is an abstract class implemented in layer classes `Linear` (fully connected layer), `Tanh` (tanh activation layer) and `ReLU` (ReLU activation layer). Here are the methods of this interface.

```
class Module(object):
    def forward(self, input_tensor):
        return output_tensor

    def backward(self, grad_wrt_output):
        return grad_wrt_input

    def gradient_step(self, step_size):

    def summary(self):
```

For activation layers:

- `forward()` : activates input tensor ($x^{(l)} = \sigma(s^{(l)})$)
- `backward()` : computes the gradient with respect to $s^{(l)}$ (input of the layer), using the gradient with respect to $x^{(l)}$ (output of the layer) as in equation (3).
- `gradient_step()` does nothing as there are no parameters to update

For linear layers:

- `forward()` : computes the linear transformation of the input $s^{(l)} = w^{(l)}x^{(l-1)} + b^{(l)}$.
- `backward()` : computes
  - and stores gradient with respect to parameters as in equation (4) and (5)
  - and returns gradient with respect to $x^{(l-1)}$ (input of the layer), using the gradient with respect to $s^{(l)}$ (output of the layer) as in equation (2).
- `gradient_step()` : updates the parameters of the layer as in equation (6) and (7).

The fully connected multi-layer perceptron models are then implemented using the following `Sequential` class.

```
class Sequential:
    def add_layer(self, layer):

    def forward(self, model_input):

    def backward(self, grad_wrt_output):

    def gradient_step(self, step_size):

    def fit(self, x_train, y_train,
            x_validation, y_validation,
            epochs=100, step_size=0.1):

    def predict(self, x, y=None):

    def summary(self):
```

- `add_layer()` : appends a new layer (e.g. `Linear`, `ReLU` or `Tanh`) to the model (after checking the compatibility of sizes)
- `forward()` : sends the input tensor through the network by calling successively the `forward` methods of the layers.
- `backward()` : backward pass on the network (compute and store gradients with respect to parameters) by calling successively the `backward` methods of the layers in reverse order.
- `gradient_step()` : update parameters of the network calling `gradient_step` methods of all the layers.
- `fit()` : train the network using SGD (for each epoch and each sample in random order call `forward`, `backward` and `gradient_step` methods).

- `predict()` : make a prediction (forward pass for each sample in data set).

Eventually another class is required : the `MSE` class with two explicit methods. Equation (1) is implemented in `compute_grad` method.

```
class MSE:
    @staticmethod
    def compute_loss(predictions, targets):

    @staticmethod
    def compute_grad(predictions, targets):
```

Details of the implementation :

- weights and biases of linear layers are initialized using uniform distribution in $[-1, 1]$. Usually uniform initialization is used but the range is adapted to the number of parameters of the layer (e.g. Xavier initialization) [3]. This is a possible improvement of the framework.
- update step size in the SGD algorithm are multiplied by 0.9 at each epoch. Therefore the step size becomes smaller and smaller and that helps convergence.

V. Adding features to the framework

Two features were added to the framework : the possibility to switch from SGD to Batch Gradient Descent and the possibility to use SGD with momentum.

A. Batch gradient descent

Batch gradient descent is a variant of SGD that computes the gradient of the loss on a batch of samples rather than simply on one random sample. In order to use Batch GD, simply pass `batch_size` argument to the fit method of the model.

B. SGD with momentum

This is a variant of SGD that is supposed to create stability during the training of the model. To do so, the usual gradient steps equations are modified as follows [1]:

$$V_t^{(l)} = \beta \times V_{t-1}^{(l)} + (1-\beta) \times \frac{\partial \ell}{\partial w^{(l)}}$$
$$w^{(l)} \leftarrow w^{(l)} - \eta V_t^{(l)}$$

In order to use this algorithm, just pass `momentum` parameter to the fit method of the model.

VI. Results

A. Basic model trained with SGD

In order to test the framework, we built a network with two input units, two output units, three hidden layers of 25 units and trained it with MSE on data generated uniformly at random (c.f. section II). Activations of the three layers are successively ReLU, ReLU and Tanh. Though we usually do not mix the two, we did it here to prove the framework works.

	MSE	Accuracy
Training	0.04256 ± 0.00469	0.9852 ± 0.0057
Validation	0.04013 ± 0.00455	0.9858 ± 0.0068

Fig. 1. Performance of the model (100 epochs, lr = 0.01)

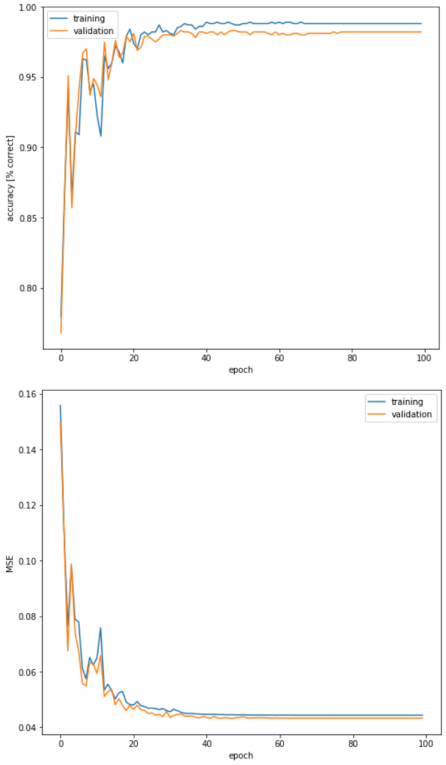

Fig. 2. Training and Validation curves (SGD training)

The model was trained 30 times independently (with new data each time). A summary of the performance is presented in array 1. Learning curves are also presented in figure 2. We note that those curves are not particularly smooth (especially the accuracy one).

Eventually we present in figure 3 a example of misclassified samples along with the circle boundary that the model is trying to find. We see that in this case (which is representative), the misclassified samples are located really close to the boundary. That seems reasonable.

B. Comparison with a torch model

Out of curiosity, we implemented the exact same model using Pytorch and trained it on the same data set. The results are quite interesting. On one hand the training curves seem to be smoother than the ones of our framework (cf. figure 4) but on the other hand, training seem to be longer. It seems longer because it takes approximately 150 epochs to achieve comparable accuracy (vs. 70 epochs) and also because our framework takes 0.24 seconds per epoch whereas the Pytorch model takes 0.65 seconds per epoch (those benchmarks were

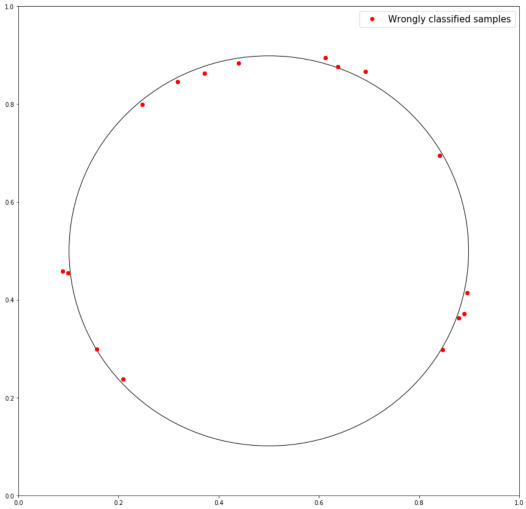

Fig. 3. Representation of misclassified samples (red) in a prediction of the trained network.

performed on the same machine, same configuration).

It is not clear what causes these differences. We checked that torch uses biases along with weights and that both models have the same global implementation. The only visible difference is that the learning rate is constant in Pytorch. Adaptive learning rate seemed to improve the performance of our home made model however and we don't see how this could be linked to longer epochs.

C. Training with SGD with momentum

This training method should increase the regularity of the learning curves and this is indeed what we observed. In figure 6 is an example of a usual loss curve with SGD with momentum training. It is clearly smoother than the one presented in figure 2. It also seems however that momentum slightly decreases the quality of training as the final validation accuracy is around 0.96 (vs. 0.98 earlier). This could probably be tackled by further tuning the hyper-parameters of the model (learning rate and momentum values) but as this was not the point of this project, it has been let to do.

VII. Conclusion

In this paper, we presented an implementation of a framework capable of building a multi-layer perceptron (MLP) and training it using stochastic gradient descent and back-propagation algorithm. We noted that a clean tensor formulation of the steps make it easy to implement and that the MLP performs very well on such toy data.

References

[1] Vitaly Bushaev. Stochastic gradient descent with momentum, Dec 2017.
[2] Franois Fleuret. Electrical-engineering - 559 : Deep learning, handout 3b. 2018.
[3] Franois Fleuret. Electrical-engineering - 559 : Deep learning, handout 5. 2018.

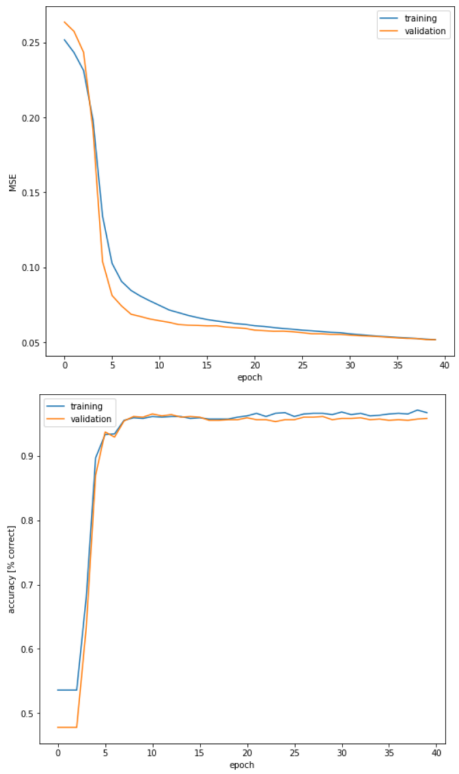

Fig. 4. Training and Validation curves of the torch model

	MSE	Accuracy
Training	0.02514 ± 0.00295	0.9751 ± 0.0061
Validation	0.03010 ± 0.00349	0.9611 ± 0.00613

Fig. 5. Performance of the torch model (150 epochs, lr = 0.01)

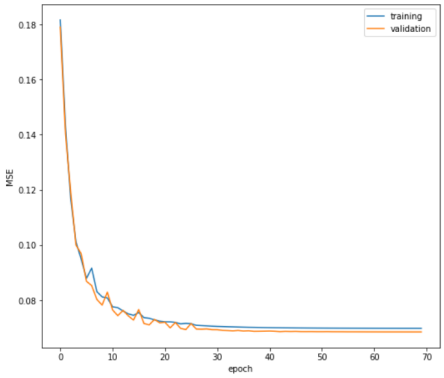

Fig. 6. Training and validation curves using SGD with momentum (momentum = 0.5)