Mini-project 2: Mini deep-learning framework

Fares Ahmed, Andrej Janchevski, Nicolò Macellari
Deep Learning EE-559

***Abstract*—Using a limited portion of PyTorch's tensors operations, the goal of this project is to develop a deep learning framework allowing users to simply define, optimize and test neural network models while leveraging the computational complexity.**

I. Introduction

The objective of this project is to develop a mini deep-learning framework, using only basic PyTorch tensors operations. In particular, this project doesn't use anything from the `nn` and `autograd` modules, thus leaving to us the implementation of the modules and the gradient computations.

Our framework provides in particular the tools to

- create multi-layered neural networks of fully connected layers, using a variety of activation functions;
- optimize the parameters using the well-known SGD and Adam optimizers;
- easily evaluate the performance of the models.

In the following sections, we will first explore and discuss the design choices of the different available modules in our framework as well as present some results of our framework on some dataset.

II. Methodology

In each following subsections, we present the different components that compose our framework.

A. Models

The models component contains all the modules used to build neural networks from scratch. They all inherit from the superclass **`Module`** and have the following methods:

- `forward` defines the forward pass of the module;
- `backward` defines the backward pass of the module;
- `param` returns a list of tuples, containing each parameter alongside its gradient as well as its first and second moment raw estimate;
- `update` update the module parameters using the gradients and a given learning rate;
- `zero_grad` puts the gradient back to zero;
- `init_params` initialize the parameters of the module.

1) Linear: This module represents a fully connected linear layer of given in and out dimensions and defines the **`Module`** methods as:

- `forward(self, input_)`
$$z^{[l]} = W^{[l]}x^{[l-1]} + b^{[l]}$$
where W and b are the weights and bias of the module, respectively;

- `backward(self, *gradwrtoutput)` computes
$$\nabla z^{[l]} = \text{gradwrtoutput}$$
$$\nabla w^{[l]} = \nabla z^{[l]} x^{[l-1]}$$
$$\nabla b^{[l]} = \nabla z^{[l]}$$
and outputs $(W^{[l]})^T \nabla z^{[l]}$;

- `param(self)` returns a list of tuples of the form $(p, \nabla p, p_m, p_v)$ containing the parameter, its gradient, its first moment estimate and its second moment estimate respectively;

- `update(self, lr)` update the module parameters using the rule $p = p - \text{lr} * \nabla p$;

- `zero_grad(self)` puts the gradients back to zero;

- `init_params(xavier_init, xavier_gain)` which indicates how to initialize the module's weights and bias:
  - if xavier_init is set to `False`, then the parameters are set using a normal distribution $\mathcal{N}(0, 1)$
  - otherwise, the parameters are set using the **Xavier initialization rule** [1] following the distribution $\mathcal{N}(0, \sigma)$ with
$$\sigma = \text{xavier\_gain} * \sqrt{\frac{2.0}{N_{in} + N_{out}}}$$
, N_{in} and N_{out} being the in and out dimensions of the module.

2) Sequential: This module allows to combine several modules sequentially to build a complete neural network. It defines the following **`Module`** methods as follows:

- `forward(self, *input_)` iteratively calls the forward methods of its modules, using the previous module's output as input;

- `backward(self, *gradwrtoutput)` iteratively calls the backward methods of its modules in reverse order, thus using the following module's output as input;
- `param(self)` returns a list containing the parameters of all its modules;
- `update(self, lr)` update the modules' parameters using the rule $p = p - lr * \nabla p$;
- `zero_grad(self)` puts the gradients of all modules' parameters to zero.

3) Activation functions: Our framework defines the following activation functions: **`ReLU`**, **`Tanh`**, **`LeakyReLU`** and **`Sigmoid`**. They all redefine the `forward` and `backward` methods from the class **`Module`** as:

- **`ReLU`** :

$$\texttt{forward(self, input\_)} = \max(0, \text{input\_})$$

$$\texttt{backward(self, x)} = \begin{cases} 1, & \text{if x} > 0 \\ 0, & \text{otherwise} \end{cases}$$

- **`Tanh`** :

$$\texttt{forward(self, input\_)} = \tanh(\text{input\_})$$

$$\texttt{backward(self, x)} = 1 - \tanh^2(\text{x})$$

- **`LeakyReLU`** :

$$\texttt{forward(self, input\_)} = \max(\alpha \text{input\_}, \text{input\_})$$

$$\texttt{backward(self, x)} = \begin{cases} 1, & \text{if x} > 0 \\ \alpha, & \text{otherwise} \end{cases}$$

- **`Sigmoid`** :

$$\texttt{forward(self, input\_)} = \frac{1}{1 + e^{-\text{input\_}}}$$

$$\texttt{backward(self, x)} = \frac{e^x}{(1 + e^x)^2}$$

B. Losses

The losses component contains all the objective functions to be minimized by the models. Our framework implements the following loss functions : **`LossMSE`** and **`LossCrossEntropy`**. They both inherit from the class **`Module`** and redefine the `forward` and `backward` methods as:

- **`LossMSE`** :

$$\texttt{forward(self, y, target)} = \frac{1}{\|\text{target}\|} \sum_i (y_i - \text{target}_i)^2$$

$$\texttt{backward(self)} = \frac{2}{\|\text{target}\|}(y - \text{target})$$

- **`LossCrossEntropy`** : For the forward pass, we first convert the predicted values of each sample into probabilities using a softmax,

$$\texttt{soft}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

we then use this soft parameter to compute the forward and backward pass

$$\texttt{forward(self, y, target)} = -\frac{1}{\|\text{target}\|} \sum_i y_i \text{soft}_i$$

$$\texttt{backward(self)} = \text{soft} - \text{target}$$

Note that the target vector is converted into a matrix of one-hot vectors, with each containing zeros and ones indicating the correct label.

C. Optimizers

The optimizer component contains the different optimizers used to train the models. Our framework implements the following optimizer functions: **`SGD`** and **`Adam`** [2]. They both inherit from a superclass **`Optimizer`** with the following methods:

- `train` train the optimizer's model;
- `step` defines the update rule for the parameters of the model.

Both optimizers redefine the `step` function accordingly:

- for **`SGD`**, the parameters are updated as

$$p_t = p_{t-1} - \text{lr} * \nabla p$$

where lr is the learning rate,

- for **`Adam`**, the parameters are updated as

$$p_t = p_{t-1} - \text{lr} * \frac{m_t}{\sqrt{v_t} + \epsilon}$$

where `lr` is the learning rate, m_t and v_t are unbiaised estimates of the first and second moments (see [2] for more details on the computations of those estimates) and ϵ is a parameter of the optimizer.

D. Cross Validation

The cross validation module contains the classes **`SGDCV`** and **`AdamCV`**, which are equivalent to their counterpart from the optimizer component, but add an additional method `cross_validate` that can be used to select among provided inputs the best set of hyperparameters for the optimizer on a given model.

E. Evaluation

The evaluation module is a simple class **`Evaluator`** that can be used to test the performance of a model. It provides a single method `compute_accuracy` that returns the accuracy of the evaluator's model on the provided input and target datasets.

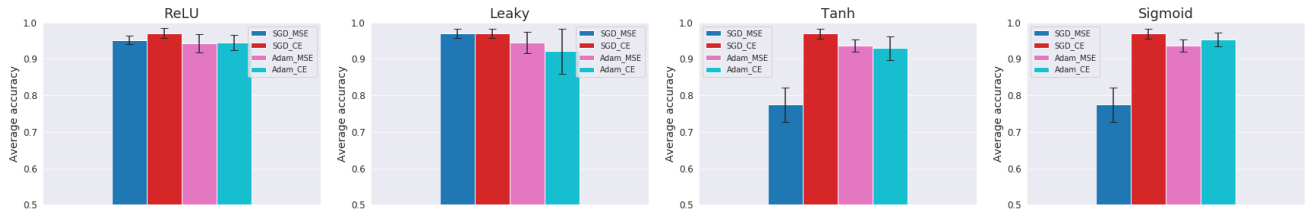

Figure 1: Results of a 2-input, 2-output network with three hidden layers of 25 nodes with the different activation layers available in our framework using the best parameters for each model, a batch size of 10 and 50 epochs. SGD stands for the SGD optimizer, Adam for the Adam optimizer, MSE for the Mean-Square Error Loss and CE for the Cross-Entropy Loss.

III. Results

To test our framework, we generate a training and a train set of a 1000 points uniformly sampled around $[0, 1]^2$.To each of them, we associate the label 0 if the point is outside the disk of radius $\frac{1}{\sqrt{2\pi}}$, and 1 otherwise. Thus, the objective of the model is to predict if a given point is outside or inside the disk.

For this task, we will construct a model using our framework with two input and output units, and three hidden layers of 25 units. We will also add an activation layer between each linear module. An example on how to construct such a network can be seen in Listing 1 with a ReLU activation layer.

Listing 1: Model used for the classification task

```
model = Sequential(Linear(2, 25), ReLU(),
                   Linear(25, 25), ReLU(),
                   Linear(25, 25), ReLU(),
                   Linear(25, 2))
```

In Listing 2 we present an example of how the model training with SGD, cross-validation of the SGD learning rate and finally the accuracy result can be efficiently computed using our framework with only a few lines of code.

Listing 2: Standard model training, cross-validation and evaluation

```
train_input, train_target = generate_disc_set(1000)
test_input, test_target = generate_disc_set(1000)
values_cv = {"lr": [1e-6, 1e-5, 1e-4, 1e-3, 1e-2]}
optimizer = SGDCV(model, nb_epochs=50,
mini_batch_size=1, criterion=LossMSE())
optimizer.cross_validate(k=5, values=values_cv)
optimizer.set_params()
optimizer.train(train_input, train_target)
evaluator = Evaluator(model)

print("Test accuracy: ",
evaluator.compute_accuracy(test_input, test_target))
```

The final results, averaged over 10 runs, are presented in Figure 1. We test this model using all four activation layers (`ReLU`, `LeakyReLU`, `Tanh` and `Sigmoid`) with both optimizers (`SGD` and `Adam`) and both our losses (`LossCrossEntropy` and `LossMSE`). Note that, because of PyTorch's lack of full reproducibility [3], results might diverge between subsequent runs. To reduce this divergence as much as possible and get similar results as the one presented, one should use the code snippet in Listing 3 before anything.

Listing 3: Code snippet for reproducibility

```
seed = 1
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed_all(seed)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

IV. Summary

During this project, we built a mini-deep learning framework, reimplementing some of the modules available in PyTorch. This includes modules such as `Sequential` to build multi-layer networks and `Linear` to create a fully-connected linear layer, activation function such as `ReLU` and `Tanh` and the losses `LossMSE` and `LossCrossEntropy`.

On top of that, we also added tools to simplify the training and testing process: a training method available in all optimizers, cross-validation classes for the selection of the hyperparameters and an `Evaluator` class that provides testing facilities.

References

[1] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256.

[2] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[3] Pytorch. (2019) Pytorch reproducibility. [Online]. Available: https://pytorch.org/docs/stable/notes/randomness.html