Mini Deep-Learning Framework

Project 2 in EPFL Course EE-559

Doga Tekin Jonathan Doenz

I. Introduction

The aim of this project is to implement a mini deep-learning framework without making use of PyTorch's autograd functionality. In particular, we are tasked with implementing the framework using only PyTorch's tensor operations and the standard math library.

In this report, we first explain the general structure of our framework, then go into detail about how each module was implemented and how the optimization works, and finally show the results of our framework on the simulated dataset.

II. General Structure

We mostly follow the suggested structure in the project briefing. All our modules inherit from a Module class that has the following functions:

- `forward(self, *input)`: computes the forward pass using the input tensors and returns the output tensors.
- `backward(self, *grad_wrt_output)`: using the gradients with respect to its outputs, computes and accumulates the gradients for its parameters (if any), and returns tensors with the gradients with respect to its inputs.
- `parameters(self)`: returns a list of parameters of the module.
- `grads(self)`: returns a list of currently accumulated gradients of each parameter, in the same order as *parameters(self)*.
- `zero_grad(self)`: zeros the gradients of all parameters in the module.

In addition to these, we also make the `__call__(self, *input)` method of the module call its own forward method, so Module instances (and models) work as they do in PyTorch, e.g. `output = model(input)`.

The optimization is done by an external SGD instance like in *torch.optim*, as explained in Section IV.

Overall, the API is similar to PyTorch's. One notable difference is that in PyTorch, module parameters are objects that accumulate gradients in their `grad` attributes, whereas in our framework module parameters and their gradients are kept as separate lists of tensors.

III. Module Implementations

A. Linear

The fully connected layer is initialized with the dimensionality of its input (A), dimensionality of its output (B), and an optional argument to disable the bias term. The weights of the layer are kept in a tensor $W \in \mathbb{R}^{B\times A}$, and the biases are kept in a tensor $b \in \mathbb{R}^B$.

Both parameters are initialized using the suggestion from [1], same as PyTorch. Specifically,

$$W_{ij} \sim U\left[-\frac{1}{\sqrt{A}}, \frac{1}{\sqrt{A}}\right], \quad b_i \sim U\left[-\frac{1}{\sqrt{A}}, \frac{1}{\sqrt{A}}\right],$$

where $U[-a, a]$ is the uniform distribution in the range $(-a, a)$ and A is the input dimension of the layer as defined above. The gradients are initialized to zero.

The module takes a single tensor as input, $X \in \mathbb{R}^{N\times A}$, where N is the batch size. The forward pass saves the input to use in the backward pass, and computes the output tensor $Y \in \mathbb{R}^{N\times B}$ as

$$Y = XW^T + b.$$

For the backward pass, the module takes a single tensor with the gradient with respect to its output, $\nabla Y \in \mathbb{R}^{N\times B}$. The gradients of the parameters are computed (and accumulated) as

$$\nabla W = (\nabla Y)^T X, \quad \nabla b = \mathbf{1}(\nabla Y),$$

where $\mathbf{1} \in \mathbb{R}^N$ is a row vector of ones used to sum the gradient over the samples in the batch. The method returns a tensor with the gradient with respective to its input $\nabla X \in \mathbb{R}^{N\times A}$,

$$\nabla X = (\nabla Y)W.$$

B. ReLU

This layer does not have trainable parameters and does not take arguments during initialization, and returns empty lists for parameters and gradients.

For the forward pass, it takes a single tensor $X \in \mathbb{R}^{N \times D}$ as input, saves it to use for the backward pass, and computes the output tensor $Y \in \mathbb{R}^{N \times D}$ by making all negative elements of X equal to 0.

For the backward pass, it takes a single tensor with the gradient with respect to its output, $\nabla Y \in \mathbb{R}^{N \times D}$. It returns the gradient with respect to its input, $\nabla X \in \mathbb{R}^{N \times D}$, by cloning ∇Y and setting it to 0 wherever the input X is negative.

C. Tanh

This layer also does not have trainable parameters and also does not take any arguments. It returns empty lists for parameters and gradients.

For the forward pass, it takes a single tensor $X \in \mathbb{R}^{N \times D}$ as input, computes the output tensor $Y \in \mathbb{R}^{N \times D}$ by element-wise applying the hyperbolic tangent operation, and saves the *output* to use in the backward pass.

For the backward pass, it takes a single tensor with the gradient with respect to its output, $\nabla Y \in \mathbb{R}^{N \times D}$. Using the derivative of the hyperbolic tangent function, it computes the gradient with respect to its input $\nabla X \in \mathbb{R}^{N \times D}$ as $\nabla X = \nabla Y \odot (\mathbf{1} - Y^2)$, where $\odot$ is element-wise multiplication, $\mathbf{1} \in \mathbb{R}^{N \times D}$ is a tensor of ones, and the exponentiation is done element-wise.

D. Sequential

This module, like its PyTorch counterpart, takes several other modules when being initialized so that it can *sequentially* run the forward and backward passes through them.

For the forward pass, it takes an input tensor and passes it through all of its modules one by one, returning the output of the last one.

For the backward pass, it takes a tensor with the gradient with respect to the output of its last module. Then, it uses the backward methods of all of its modules to propagate the gradient backward from the last module all the way to the first, accumulating gradients in the meantime. It finally returns the gradient with respect to the input of its first module.

In theory, the framework allows stacking and/or nesting sequential modules even though it's not used in this project.

E. MSE

The MSE loss is treated like any other module with no trainable parameters. It returns empty lists for parameters and gradients.

For the forward pass, it takes two tensors: the prediction $Y \in \mathbb{R}^{N \times D}$ and the target $T \in \mathbb{R}^{N \times D}$. It saves both to use later in the backward pass, and computes the loss as

$$\mathcal{L} = \frac{1}{ND} ||Y - T||_{\mathrm{F}}^2,$$

where $||A||_{\mathrm{F}}$ is the Frobenius norm of tensor A.

The backward pass does not take any tensors with respect to its output, as the loss is supposed to be the last in the pipeline. It computes and returns the gradient of the loss with respect to the prediction:

$$\frac{\partial \mathcal{L}}{\partial Y} = \nabla Y = \frac{2}{ND}(Y - T).$$

IV. Optimization

After a model is created using the Sequential module and the implemented layers, the actual optimization is done by an external optimizer implemented in the SGD class.

To instantiate an optimizer of this class, it needs to be given two lists of tensors, one for the parameters to be optimized and the other for the gradients of those parameters, in the same order. It also takes a learning rate argument.

After that, the API is the same as PyTorch's, the optimizer instance can call `step` to take one gradient step using the currently accumulated gradients and call `zero_grad` to zero all of the tracked gradients.

The `step` method simply performs $p \leftarrow p - \alpha \nabla p$ for each tracked parameter p and learning rate α.

Algorithm 1 Model Optimization

```
model = Sequential(...)
criterion = MSE()
optimizer = SGD(params, grads, lr)
for each epoch do
    for each input, target do
        output = model(input)
        loss = criterion(output, target)
        grad = criterion.backward()
        model.backward(grad)
        optimizer.step()
        optimizer.zero_grad()
    end for
end for
```

The optimization procedure in Algorithm 1 is almost the same as in PyTorch. The one difference is at lines 8–9: loss functions in PyTorch return a tensor that the user can directly call `backward` on, which automatically propagates the gradients back through the model. In our framework, the user calls `backward` on the loss function itself, not the returned loss tensor. This gives the gradient with respect to the model output, which can then be supplied to the `backward` method of the model to propagate the gradients. Once the gradients are accumulated, the optimizer can be used as explained above to optimize the parameters.

V. Results and Comparisons

On the simulated dataset of this project, our framework performs identically to PyTorch up to some floating point precision. To show this, we wrote the same training code for both our framework and PyTorch—the codes are remarkably similar as well—and created the same networks in each framework.

We create two networks with the architecture given in the briefing, one of them using ReLU and the other using Tanh. We show that our framework works with both activation functions and we get to compare the performances of the two networks as well.

Figure 1 shows the training loss of both networks over 500 epochs in our framework and in PyTorch for a single run. The same data is used in both cases and the same seed is used to initialize the networks in both frameworks. This results in almost identical performance, as indicated by the overlapping lines.

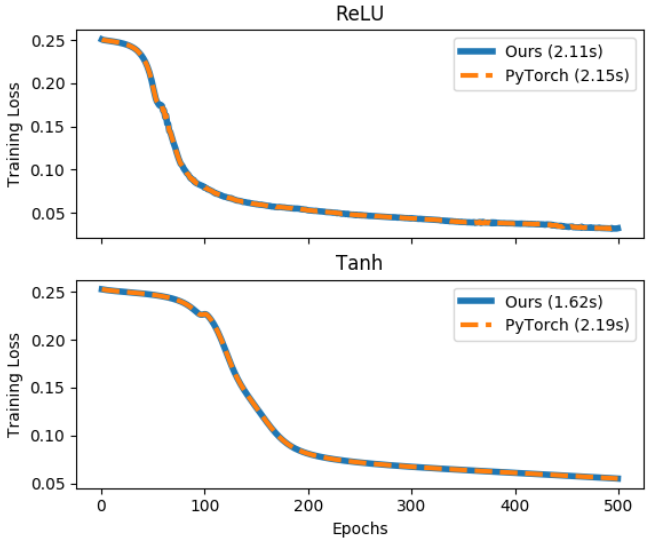

Fig. 1: Training loss for both networks in both frameworks.

Although hard to see at this scale, there are slight differences in the loss values, which we assume is related to an implementation detail we are not aware of. The small differences can build up and lead to slightly different error rates in some seeds, but the results are generally very similar. Table I shows the error rates at the end of the training run shown in Figure 1.

Table I: Comparison of error rates between frameworks after 500 epochs. Same seed to generate data and initialize networks.

		Ours	PyTorch
ReLU	Training	0.021	0.021
	Test	0.013	0.013
Tanh	Training	0.031	0.031
	Test	0.023	0.023

We also compare the training times of the frameworks in Table II by running them multiple times and looking at the average. We see that our framework is marginally faster in the ReLU case and considerably faster in the Tanh case. This might be due to some overhead in PyTorch that is necessary to perform functionality not found in our framework.

Table II: Comparison of 500 epoch CPU training times between frameworks (in seconds). Mean (±std) over 10 runs.

	Ours	PyTorch
ReLU	2.001 (±0.006)	2.049 (±0.012)
Tanh	1.503 (±0.008)	2.191 (±0.073)

VI. Conclusion

We have succeeded in implementing a mini deep-learning framework that provides a small portion of PyTorch's functionality without using automatic differentiation. It allows building simple neural networks and can train them faster than PyTorch on CPU, while performing at the exact same level.

References

[1] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, May 2010, pp. 249–256. URL: http://proceedings.mlr.press/v9/glorot10a.html.