Project 2 – Mini deep-learning framework

Justin Deschenaux, Guillaume Follonier
EE-559 Deep learning, EPFL Lausanne, Switzerland

May 22, 2021

1 Introduction

Tensor manipulation frameworks play an important role in the advance of machine learning in the recent years. Indeed, they allow the creation of models in a descriptive manner by abstracting away many technicalities, such as automatic differentiation. In this project, we implemented a simple but working toolbox for training neural networks.

2 Models and methods

First, note that we use the original torch tensors, allowing us to leverage the efficient implementations of common linear algebra operations as well as broadcasting. This made the project much easier than having to code them ourselves. When creating tensors in the framework's code, we always started with empty ones (calling `torch.empty`). On the other hand, when generating samples or writing experiments, we used other utilities (e.g. `torch.random`), as it was not the goal to reproduce the whole `torch` package. Only `autograd` was concerned. The building-blocks of the framework are separated in three categories: `Module`s, `Loss`es and `Optimizer`s.

2.1 Main API

```
class Module(object):
    def forward(self, input):
        pass

    def backward(self,
            gradwrtoutput, x):
        pass

    def zero_grad(self):
        pass

    def params(self):
        pass
```

```
class Loss(object):
    def forward(self, pred,
                target):
        pass

    def backward(self, pred,
                 target):
        pass
```

```
class Optimizer:
    def __init__(self,
                 model,
                 **kwargs):

    def zero_grad(self):
        pass

    def step(self):
        pass
```

As the name suggests, `forward` and `backward` implement the two passes of backpropagation. Unlike `torch`, we do not have a graph keeping track of transformations. Therefore, modules need the gradient with respect to their output to compute the backward pass. We decided that each module and loss would return the gradient with respect to its input when `backward` is called. The result is passed to the previous layer as `gradwrtoutput` afterwards. The `x` parameter of `bakward` must be the input used during the forward pass. It is required as the gradient typically depends on it. An alternative design choice would be to store `input` during the forward pass. In simple experiments, both approach works similarly, but more advanced models could work by sending multiple samples though `forward` before calling `backward`. For example, our model from project 1, which starts by sending each digit sequentially through a feature extraction network. In this case, the model would use an incorrect `input` during one of the backward passes. Consequently, we chose to explicitly provide the input `x` again.

The API for `Loss` is different because it is used to compare the prediction (`pred`) and ground-truth (`target`). We also expect `Loss.backward` to return the gradient with respect to `pred`.

The other methods are simple: `params` returns a list of the trainable parameters of the model. It is empty for layers such as `ReLU` or `Tanh`. Similar to `torch`, we chose to accumulate gradients when calling `backward`. Therefore, `zero_grad` resets them. Finally, `Optimizer`s are initialized with a `Module` whose parameters must be tuned with a specific algorithm such as SGD or Adam. To this end, it exploits the weights returned by `Module.params` and their gradients stored in `Module.grad`.

2.2 Implementation

We do not want to spend much time on every little detail since some where taken from the course "as-is" (e.g. formula of backprop). We believe our code is self-explanatory and contains enough comments to be easily understood. We only present an overview of the functionalities.

Linear layer The core module of the project. Implements a fully-connected layer using weights $W \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}^n$. Compute an affine transformation $y = W \cdot x + b$.

Sequential layer Required for larger models. With this module, it is easy to train multi-layer networks as it takes care of forwarding the input as well as performing the backward pass through all elements. `Sequential` does not contain any parameter itself, but a call `Sequential.params` returns a flattened list of the parameters of its layers. `Sequential.grad` works similarly.

Activation functions `ReLU`, `Tanh` and `Sigmoid` were implemented. We did not include others such as softmax, SELU since we are dealing with a simple problem and we believe they would not significantly increase the performance.

MSE loss Computes $||\hat{y} - y||_2^2$ where $\hat{y} \in \mathbb{R}$ is a prediction and $y \in \mathbb{R}$ the target/ground-truth.

SGD optimizer After `Model.backward` is called, the gradients are stored in the model and we can leverage `SGD.step()` to perform the update. The optimizer computes the following values at each iteration t:

$$(\boldsymbol{v}_{t+1})_i = \rho(\boldsymbol{v}_t)_i + (1-\rho)(\boldsymbol{g}_t)_i \qquad \text{(Momentum of the gradients)} \tag{1}$$

$$(\boldsymbol{w}_{t+1})_i = (\boldsymbol{w}_t)_i - \eta(\boldsymbol{v}_{t+1})_i \qquad \text{(Standard update rule)} \tag{2}$$

where $\rho \geq 0$ roughly controls how fast the gradients are "forgotten" (the larger ρ the slower). Therefore $\boldsymbol{v_t}$ is the moving average of gradients $\boldsymbol{g_t}$ with respect to the parameters $\boldsymbol{w}$. Note that with $\rho = 0$, we recover SGD without momentum. Finally, η is the learning rate.

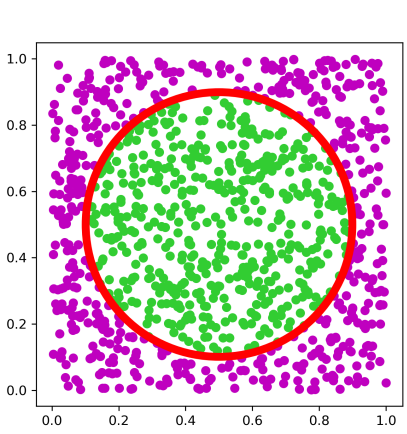
Adam optimizer Once the gradients are available, `Optim.Adam` tunes the parameters as follows: at each iteration t, it computes an estimate of the first and second order statistics of the derivatives and uses them to scale the update rule:

$$(\boldsymbol{m}_{t+1})_i = \beta_1(\boldsymbol{m}_t)_i + (1-\beta_1)(\boldsymbol{g}_t)_i \qquad \text{(Momentum of the gradients)} \tag{3}$$

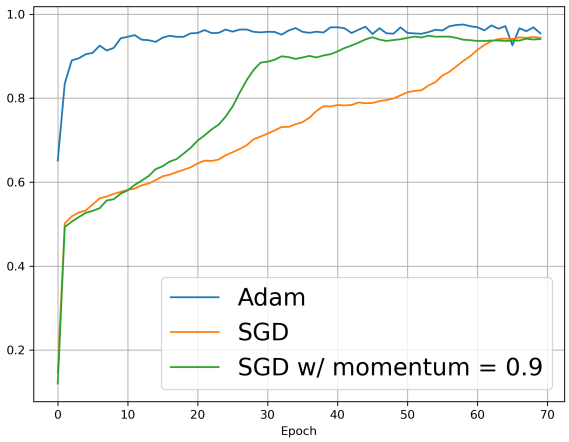
$$(\boldsymbol{v}_{t+1})_i = \beta_2(\boldsymbol{v}_t)_i + (1-\beta_2)[(\boldsymbol{g}_t)_i]^2 \qquad (2^{\text{nd}} \text{ order statistic of gradients}) \tag{4}$$

$$(\boldsymbol{w}_{t+1})_i = (\boldsymbol{w}_t)_i - \eta \frac{(\boldsymbol{m}_{t+1})_i}{\sqrt{(\boldsymbol{v}_{t+1})_i} + \epsilon} \qquad \text{(Rescaling before update)} \tag{5}$$

$\beta_1, \beta_2 \geq 0$ controls the running averages of moments (similar to ρ for SGD). The $\epsilon > 0$ in the denominator of eq. (5) is required to avoid a division by zero that would issue in NaN problems. Both $(\boldsymbol{v}_0)_i$ and $(\boldsymbol{v}_0)_i$ are initialized to zero. Finally, as for SGD, η is the learning rate and $\boldsymbol{g}_t$ is the gradient with respect to the parameters at iteration t.



(a) Classification problem



(b) F1-score with best learning rate

Figure 1

3 Results

Data distribution Both the training and test set are generated in a similar fashion and contain 1,000 examples. Each point x is sampled uniformly on $[0, 1]^2$ and is assigned label $t_x = 0$ if it lies outside of the red circle centered in $(0.5, 0.5)$ and of radius $\frac{1}{\sqrt{2\pi}}$ and $t_x = 1$ otherwise. Fig. 1a represents a typical training set. The high sample density makes the problem easy.

Comparing optimizers In order to test our code, we trained three layer fully-connected networks with 25 hidden units each. Since we implemented multiple optimizers, we also wanted to compare them. All models where trained for 70 epochs and with learning rates 0.1, 0.01, 0.02, 0.05, 0.001, 0.002 and 0.005. The run with the best F1-score was kept. Using Adam requires less training epochs for a better performance. However, by a careful choice of hyperparameters and enough time, we can reach close performance with all algorithms. This is satisfactory and shows that our implementation is working as intended. The following table summarizes the results of this experiment. $\beta_1 = 0.9, \beta_2 = 0.999$ and $\rho = 0.9$ are PyTorch defaults for the momentum parameters. Fig 1b shows the F1-score across all epochs.

Optimizer	Best learning rate	Best F1-score
SGD ($\rho = 0$)	0.002	0.946
SGD ($\rho = 0.9$)	0.1	0.949
Adam ($\beta_1 = 0.9, \beta_2 = 0.999$)	0.005	**0.975**

4 Discussion

Working on this project was very fulfilling and allowed us to understand better how deep-learning toolboxes such as PyTorch works under the hood. There are a few things we would like to try in the future. Aside of implementing other modules and losses, we first tried to create a `Tensor` class that would keep track of operations and create a graph for the backward pass. It turns out it is much harder than we expected, and not only due to the fact that we have to perform a topological sort on the nodes before backprop. Nonetheless, we believe that understanding precisely how tools work grants us with more flexibility than treating them as blackboxes. Therefore, we plan to revisit this project in the near future and are grateful to the teaching team for it.