Project 2 – Worch: A minimalist deep learning framework

Thomas Berger, Pierre Bouquet, Eliot Walt

***Abstract*—In this report, we present a simple implementation of a deep learning micro-framework. Built on top of pytorch, this library supports simple multi-layer perceptrons with linear rectified unit, sigmoid and tanhgent hyperbolic activation functions and mean squared error loss using only tensor operations.**

I. Introduction

The recent development in deep neural networks heavily relies on deep learning frameworks which implement the necessary machinery and allow fast development and deployment in production. At the core of these frameworks is the concept of automatic differentiation which allows to compute derivatives with respect to each weights of a neural network. In this project, we implement some of the building blocks of a deep learning framework. Sections IV and V provide a comparison of the widely used library Pytorch with our implementation. The source code can be found at `https://github.com/eliotwalt/worch`.

II. Background

A. Forward pass

A neural network implements a parametric function $f(.;\mathbf{w})$ where $\mathbf{w}$ are the model parameters. This function is usually defined in terms of a graph of sub-functions, sometimes referred to as "layers". Formally, given a training set $\{(\mathbf{x}_i, \mathbf{y}_i) \in \mathbb{R}^{d_{in}} \times \mathbb{R}^{d_{out}}, i = 1, ..., N\}$, f is a composition of L functions $f^{(l)}$,

$$\begin{aligned} f &: \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}} \\ &\mathbf{x} \to \mathbf{y} = f(\mathbf{x};\mathbf{w}) = f^{(L)} \circ ... \circ f^{(1)}(\mathbf{x}) \end{aligned} \tag{1}$$

where

$$\begin{aligned} f^{(l)} &: \mathbb{R}^{d_{l-1}} \to \mathbb{R}^{d_l} \\ &\mathbf{x} \to \mathbf{y} = f^{(i)}(\mathbf{x};\mathbf{w}^{(l)}) \end{aligned} \tag{2}$$

Computing $f(\mathbf{x};\mathbf{w})$, where f is a neural network, is referred to as the *forward pass*.

B. Backward pass

Training a neural network requires to update its weights based on an error signal. Typically, this signal is expressed as a sum of loss functions evaluated at each data sample. Formally, given an individual loss function $\mathcal{L}_i : \mathbb{R}^{d_{out}} \to \mathbb{R}$, the error signal is given by the total loss

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_i(f(\mathbf{x}_i;\mathbf{w}); \mathbf{y}_i). \tag{3}$$

Ultimately, the training boils down to the optimization problem

$$\mathbf{w}^\star = \arg\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}). \tag{4}$$

The classical approach to solve this problem is to use an iterative method like *gradient descent* which updates the weights using the gradient of the loss with respect to the parameters. Given a learning rate η, the update rule is

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}_t) \tag{5}$$

As seen previously, a neural network can be expressed as a composition of multiple functions. Therefore, given the inputs of each sub-function during the forward pass, one can compute the gradient of the loss with respect to all the parameters of the network using the chain rule. Applying this process is referred to as *backpropagation* [2].

III. Methodology

The design of our framework is highly inspired by Pytorch. There are two main classes. First, `worch.nn.Module` is the base class for the neural networks components and loss functions. Then, `worch.optim.Optimizer` is the parent class of the different optimizers.

A. Linear

A *linear layer* consists of a parametric linear mapping between an input dimension, d_1 and an output dimension d_2. Given N input vectors in a matrix $X \in \mathbb{R}^{N\times d_1}$ (a batch), a weight matrix $W \in \mathbb{R}^{d_2\times d_1}$ and a bias vector $\mathbf{b} \in \mathbb{R}^{d_2}$, the linear layer produces

$$\begin{aligned} f : \mathbb{R}^{d} &\to \mathbb{R}^{d}_{+} \\ X \to Y &= XW^{\top} + \text{BC}(\mathbf{b}) \end{aligned} \tag{6}$$

where BC(.) is a broadcasting function which creates N copies of $\mathbf{b}$ to match the dimension of $WX^{\top}$. The gradient of a loss function with respect to the input of a linear layer is given by

$$\frac{\partial \mathcal{L}}{\partial X} = \frac{\partial \mathcal{L}}{\partial Y}\frac{\partial Y}{\partial X} = \frac{\partial \mathcal{L}}{\partial Y} W \in \mathbb{R}^{N\times d_1} \tag{7}$$

and the gradient of the loss with respect to the parameters W and $\mathbf{b}$

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial Y}\frac{\partial Y}{\partial W} = (\frac{\partial \mathcal{L}}{\partial Y})^{\top} X \in \mathbb{R}^{d_2\times d_1} \tag{8}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial Y}\frac{\partial Y}{\partial b} = (\frac{\partial \mathcal{L}}{\partial Y})^{\top} \mathbf{1}_n \in \mathbb{R}^{d_2} \tag{9}$$

where $\mathbf{1}_n$ is a n-dimensional vector containing only ones.

B. ReLU

The rectified linear unit, or *ReLU*, is non-linear activation function given by [2]

$$\begin{aligned} ReLU : \mathbb{R} &\to \mathbb{R}_{+} \\ x \to y &= \max(0, x), \end{aligned} \tag{10}$$

where the max function is applied component-wise. The gradient of the loss with respect to the input of a ReLU activation is given by

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial x} = \begin{cases} \frac{\partial \mathcal{L}}{\partial y} & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{11}$$

Note that ReLU can also be applied to tensors component-wise.

C. Sigmoid

The sigmoidal unit, or *sigmoid*, is a very popular activation function given by [2]

$$\begin{aligned} \sigma : \mathbb{R} &\to [0, 1] \\ x \to y &= \frac{1}{1+\exp(-x)}, \end{aligned} \tag{12}$$

where the exponential function is applied component-wise. The gradient of the loss with respect to the input of a sigmoid is

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial \mathcal{L}}{\partial y}\sigma(x)(1-\sigma(x)) \tag{13}$$

Note that sigmoid can also be applied to tensors component-wise.

D. Tanh

The tangent hyperbolic, or *tanh*, is a very popular activation function given by [2]

$$\begin{aligned} \tanh : \mathbb{R} &\to [-1, 1] \\ x \to y &= \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}, \end{aligned} \tag{14}$$

where the exponential functions are applied component-wise. The gradient of the loss with respect to the input of a hyperbolic tangent is

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial \mathcal{L}}{\partial y}(1-\tanh^2(x)) \tag{15}$$

Note that tanh can also be applied to tensors component-wise.

E. MSE

The mean squared error, or *MSE*, is a widely used machine-learning loss function. It measures the distance between a vector predictions $\hat{\mathbf{y}} \in \mathbb{R}^D$ and a vector of D ground truth values $\mathbf{y} \in \mathbb{R}^D$ by [2]

$$\mathcal{L}_{MSE}(\hat{\mathbf{y}}; \mathbf{y}) = \frac{1}{D}\sum_{i=1}^{D}(\hat{y}_i - y_i)^2 \tag{16}$$

The gradient of the MSE loss with respect to the input is

$$\frac{\partial \mathcal{L}_{MSE}}{\partial \hat{y}_i} = \frac{2}{D}(\hat{y}_i - y_i) \tag{17}$$

F. Sequential

A Sequential object stacks multiple modules on top of another. Its backward pass consists in calling successively the backward pass of its sub-modules. Each time, it feeds the gradient with respect to the outputs and collects the gradient with respect to the inputs, therefore completing the backpropagation chain.

G. SGD

Stochastic gradient descent, or *SGD*, is an optimization algorithm used in many machine learning applications. Despite its simplicity, it can most of the time converge in the non-convex deep learning optimization landscape. The update rule is very similar to gradient descent (equation 5). However, SGD works on a subset of the dataset. Let $\mathcal{B}$ be a mini-batch of training data sampled uniformly at random, then [1]

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \sum_{i=1}^{|\mathcal{B}|} \nabla \mathcal{L}_i(\mathbf{w}_t), \tag{18}$$

where $\mathcal{L}_i$ is the loss function evaluated at a single point $(f(\mathbf{x}_i; \mathbf{w}_t), \mathbf{y}_i)$.

IV. Experiments

We tested our implementation on a toy dataset and compared it with pytorch. The training and validation data consist of 1000 samples uniformly distributed in $[0,1]^2$ and associated to the label 0 if outside of the circle centered at (0.5, 0.5) of radius $1/\sqrt{2\pi}$. A multi-layer perceptron with two input units, two sigmoidal output units and three hidden layers of 25 units with ReLU activations was trained with both frameworks for 100 epochs with a batch size of 32. We used the mean squared error loss and SGD with $\eta = 0.1$.

V. Results

Both frameworks achieve very good performance on this simple task. Figure 2 shows the evolution of the loss on the training and validation sets for both models. Even though our implementation seems to converge slightly faster, the overall performance is comparable and shows the correctness of our framework.

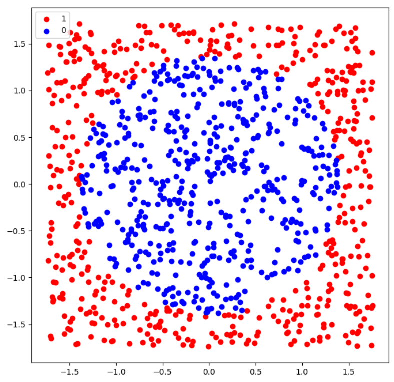

Fig. 1: Experiments dataset

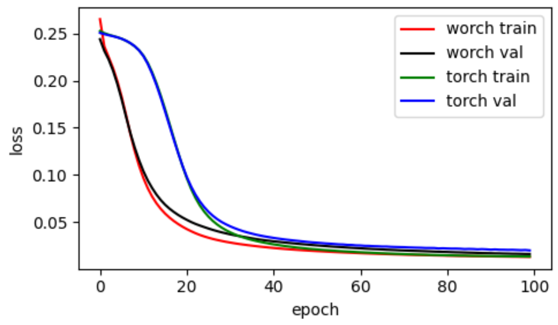

Fig. 2: Evolution of the loss as a function of the number of epochs

VI. Conclusion

The experiment shows that our framework successfully implements a multi-layer perceptron. Even though this kind of model is very basic, we have observed that quite a lot of complexity already arises when it comes to implementation. In conclusion, building this minimalist framework was a great experience but we are still very thankful to the different teams developing reliable, open source libraries allowing the community to focus their attention on the models design.

REFERENCES

[1] Léon Bottou and Olivier Bousquet. “The Tradeoffs of Large Scale Learning”. In: *Advances in Neural Information Processing Systems 20 (NIPS 2007)*. Ed. by J.C. Platt et al. NIPS Foundation (http://books.nips.cc), 2008, pp. 161–168. URL: http://leon.bottou.org/papers/bottou-bousquet-2008.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. “Deep Learning”. In: http://www.deeplearningbook.org. MIT Press, 2016, pp. 164–223.