Deep Learning Mini-Project 2

Pierre-Antoine Desplaces, Raphaël Steinmann
Department of Computer Science, EPFL, Switzerland

***Abstract*—This report presents the design and implementation of a mini deep learning framework with Pytorch's Tensor operations.**

I. Introduction

The goal of this project is to implement a mini deep learning framework using only pytorch's tensor operations and the standard math library, hence in particular without using Pytorch's autograd or neural-network modules.

The framework must provide the necessary tools to

- build networks combining fully connected layers, Tanh and ReLU,
- run the forward and backward passes,
- optimize parameters with SGD for MSE.

II. Methodology

Our implementation of the mini deep learning framework is divided into several modules, whose purposes and implementation details are explained in the following sections.

A. Module

`Module` is the superclass that represents a module. Every component described below is a module and thus inherits from this class. `Module` contains three methods:

- `forward` defines how the module behaves during the forward pass of the neural network,
- `backward` defines how the module behaves during the backward pass of the neural network,
- `param` returns a list containing, for each parameter p of the module, a tuple containing p's parameter tensor and gradient tensor, i.e. p and the derivative of the loss with respect to p.

B. Linear

The `Linear` module represents a fully connected layer in the neural network.

- `__init__(self, in_dim, out_dim)` The constructor of a linear layer takes as input its number of input and output dimensions. It also defines three instance variables: `w`, `b` and `x_previous_layer`. Their purpose is to keep track of the weights, the biases and the output of the previous layer's activation function.
- `forward(self, input_)` The output of layer l's forward pass is computed as follows:

  $$z^{[l]} = w^{[l]} x^{[l-1]} + b^{[l]}$$

  The method receives as input the output of the previous layer's activation function, i.e. $x^{[l-1]}$ (or just the input data if $l = 1$). It outputs $z^{[l]}$, which will be received as input by layer l's activation function.
- `backward(self, gradwrtoutput)` The output of layer l's backward pass is computed as follows:

  $$dw^{[l]} = dz^{[l]} x^{[l-1]},$$
  $$db^{[l]} = dz^{[l]},$$
  $$dx^{[l-1]} = {w^{[l]}}^T dz^{[l]}$$

  The notation was simplified here: dw (resp. dz, db, dx) represents the derivative of the loss with respect to w (resp. z, b, x). The layer receives $dz^{[l]}$ as input from its activation function and $x^{[l-1]}$ was cached during the forward pass. The backward pass outputs $da^{[l-1]}$ which is sent to the previous layer's activation function to continue the backward pass.

  During the backward pass, dw and db also become the new values of layer l's gradients, which will be used by the SGD optimizer to perform a step in the direction of the gradient.
- `param(self)` Returns this layer's parameters (the weights and biases), and the derivative of the loss with respect to them.
- `zero_grad(self)` The purpose of this method is to set the gradients of the layer to zero before the next batch can go through the neural network. It is called every time before a batch performs its backward pass.

C. Activation functions

An activation function is usually located after each layer of the network. In our implementation, it performs the following computations:

- `forward(self, input_)` In the forward pass, the activation function receives as input the output of

layer l, i.e. $z^{[l]}$. This value is cached in an instance variable for future use during the the backward pass. g represents the activation function (Tanh, ReLU, ...).

$$x^{[l]} = g^{[l]}(z^{[l]})$$

- `backward(self, gradwrtoutput)` For the backward pass, the activation function receives as input $dx^{[l]}$ from layer $l+1$ and uses the value of $z[l]$ that was cached during the forward pass to compute and return

$$dz^{[l]} = dx^{[l]} \odot g^{[l]\prime}(z^{[l]}),$$

where g' is the derivative of the activation function.

1) ReLU: The ReLU activation function is given by

$$g(x) = max(0, x)$$

and its derivative is

$$g'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & otherwise \end{cases}$$

2) Tanh: The Tanh activation function is given by

$$g(x) = \frac{e^x}{1 + e^x}$$

and its derivative is

$$g'(x) = 1 - tanh^2(x)$$

However, note that we used the method `tanh` from torch.Tensor and thus did not have to re-implement this function as described above for the forward pass.

D. Sequential

The purpose of the `Sequential` module is to combine several modules in a basic sequential structure. For example, `Sequential(Linear(2,25), Tanh(), Linear(25,25), Tanh(), Linear(25,2), Tanh())` creates a network with two input units, two output units and two hidden layers of 25 units each.

- `forward(self, input_)` The forward pass basically iterates over each module, calls its forward method, feeds the output to the forward method of the next module, etc.

- `backward(self, gradwrtoutput)` The same concept is applied for backward method. The only difference is that we iterate over the modules in reversed order (backward pass).

- `param` returns a flatened list of each module's parameters. Each parameter in the list is represented as a tuple containing the parameter tensor (e.g. w) and the gradient tensor (e.g. $d\mathcal{L}/dw$).

- `zero_grad` calls the `zero_grad` method of each module. This method does not apply to some modules (i.e. activation functions), in that case, it does nothing.

E. LossMSE

The purpose of the `LossMSE` module is to compute the mean square error given some predictions $\hat{y}$ and their corresponding labels y.

- `forward(self, preds, labels)` In the forward pass, the MSE is computed as follows:

$$\mathcal{L} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Note that during the forward pass, $(y_i - \hat{y}_i)$ is stored in an instance variable. This allows us to call the `backward` method with no parameters, which is quite convenient.

- `backward` During the backward pass, we compute the derivative of the loss function, which in the case of the MSE is given by

$$2 \sum_{i=1}^{n} (y_i - \hat{y}_i)$$

F. LossCrossEntropy

In addition to what was asked, we also implemented a module for the cross-entropy loss function.

- `forward(self, preds, labels)` The cross-entropy for a single sample is given by

$$\mathcal{L} = y_1 log(p_1) + y_2 log(1 - p_1)$$

where $y = (y_1, y_2)$ is a one hot vector indicating the correct classification label of the sample and p_1 (resp. $1 - p_1$) is the predicted probabilty that the sample belongs to class c_1 (resp. c_2). The forward pass calculates the cross-entropy loss by summing the cross-entropy for each sample in the minibatch.

To turn our predicted values into predicted probabilities, we use the softmax function:

$$stable\_softmax(p) = \frac{e^{p - max(p)}}{\sum_{i=1}^{2} e^{p - max(p)}}$$

with $p = (pred_1, pred_2)$ the predicted values. Note that this is not exactly the softmax function, but a stabilized version that prevents overflows due to exponentials of big values.

- `backward` Again, we compute the derivative of the loss function during the backward pass. The derivative of the cross-entropy loss function is simply given by

$$p - y$$

G. optim_SGD

The purpose of this class is to perform a gradient step to optimize the parameters of the model. The constructor of this module takes as input the parameters of the model (i.e. the parameters of each Module in the model) and the learning rate lr (also called step size).

- `step` This method iterates over every parameter of every module, gets its parameter tensor p and gradient tensor dp and updates the former with the latter as follows:

$$p = p - lr * dp$$

III. Results

To test our mini deep learning framework, we generated a training and a test set of 1000 points sampled uniformly in $[0, 1]^2$, each with a label 0 if outside the disk of radius $\frac{1}{\sqrt{2\pi}}$ centered in $(\frac{1}{2}, \frac{1}{2})$ and 1 if inside. In other words, we built a neural network and trained it to predict if, given its (x, y) coordinates, a point lies inside or outside the disk.

We used our deep learning framework to build a network with two input units, two output units and three hidden layers of 25 units each. We also added a $tanh$ activation function after each layer, except the very last one. With this configuration, and using SGD with the MSE loss function we obtained (on an average of 50 runs) a training accuracy of 97.3% and a testing accuracy of 96.2%.