Mini-Project 2 - Mini deep-learning framework without autograd

Timothee Jaubert (261816), Simon Roquette (246540), Anton Soldatenkov (314433)
Deep Learning EE-559, EPF Lausanne, Switzerland

***Abstract*—The implemented framework is based on PyTorch tensor operations with autograd (automatic differentiation) set globally off. It provides basic tools, layers and optimizers for building neural networks which can be used to solve various machine learning problems. The framework is tested on a simple binary classification task with accuracy being the target metric.**

I. Introduction

The goal of this project was to implement a neural network framework able to combine multiple layers, run the forward and backward passes and optimize its parameters using stochastic gradient descent and then test the model with a given set of layers on a small data set. The specific of the binary classification task influences our choice of layers which will be covered in subsequent sections. We will also explain the mathematical concepts behind our implementation. At the end we will analyze the performance of the model.

II. Architecture

All modules are classes inherited from the same base class `Module` with a constructor (`__init__`), `forward` and `backward` methods (to be used in forward and backward passes respectively) and `param` method which returns its parameters. `forward` takes an input tensor, saves it to compute the gradient of the loss in the backward pass later and computes an output. `backward` takes as input the gradient of the loss with respect to its outputs, computes the gradient with respect to its parameters (if applicable) and to the input, the latter value is returned. Each parameter is stored inside a list in a tuple with its gradient (computed and filled during the backward pass). Some modules (e.g. ReLU) are non-parametric, in this case their parameter list is empty.

`Model` class allows to create a neural network by combining layers. Its constructor accepts a list of modules, `forward` method passes an input tensor over these layers, calling the `forward` method of each layer and returns the result. The `backward` method takes the gradient of the loss and propagates it backwards using `backward` of the layers in the reversed order.

In order to optimize a continuous function and perform an iterative update of the parameters, a separate gradient descent optimizer is implemented in the `SGD` class. It takes model parameters and a learning rate in the constructor. This class has an only method `step` without arguments that performs the update:

$$\omega = \omega - \eta \cdot \frac{\partial L}{\partial w} \tag{1}$$

where η is the learning rate, $\frac{\partial L}{\partial w}$ is the derivative of the loss with respect to the parameter.

III. Layers

Our framework provides tools to combine linear layers with nonlinear activation functions. We implemented *ReLU* and *tanh* activations and *Sequential* to operate multiple layers as one.

A. Linear

In general, a linear layer takes as input a batch of samples $X \in \mathbb{R}^{N\times d}$ and outputs $Y = XW$, $Y \in \mathbb{R}^{N\times m}$ where W is a $d \times m$ weight matrix, N is the number of samples in the batch. Then a bias $b \in \mathbb{R}^m$ is added to each sample. During the backward pass, we assume that the matrix of partial derivatives of the loss with respect to the outputs $\frac{\partial L}{\partial Y}$ has already been computed. It has the same shape as Y, $n \times m$. The goal is to compute $\frac{\partial L}{\partial X}$, $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$. By applying the chain rule we obtain

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial X} = \frac{\partial L}{\partial Y} W^T \tag{2}$$

and

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial W} = X^T \frac{\partial L}{\partial Y}. \tag{3}$$

For the bias,

$$\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial Y_i}\frac{\partial Y_i}{\partial b_i} = \sum_{n=1}^{N} \left(\frac{\partial L}{\partial Y}\right)_{n,i}, \tag{4}$$

where Y_i is the output of i-th neuron.

The weights are initialized in the standard PyTorch way:

$$w \sim U\left(-\frac{1}{\sqrt{d}}, \frac{1}{\sqrt{d}}\right). \tag{5}$$

In our implementation, there is also an option (inspired by PyTorch) whether to include bias or not.

B. ReLU

ReLU is an activation function, therefore, it has no parameters. During the backward pass it keeps only those components where the input was positive:

$$\left(\frac{\partial L}{\partial X}\right)_{ij} = \mathbb{1}(x_{ij} > 0)\left(\frac{\partial L}{\partial Y}\right)_{ij}. \tag{6}$$

Here, as usual, X denotes the input, Y the output.

C. Tanh

$tanh$ is another activation implemented in our framework. By definition,

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{7}$$

Its derivative is $1 - tanh^2(x)$, therefore,

$$\frac{\partial L}{\partial X} = \left(1 - tanh^2(X)\right) \odot \frac{\partial L}{\partial Y}, \tag{8}$$

where $\odot$ stands for element-wise multiplication.

D. Sequential

The way this module is organised is quite similar to `Model` class described in section II. The only difference is that `Model` accepts one argument - a list of modules whereas `Sequential` accepts an arbitrary number of arguments (like in PyTorch).

IV. Loss functions

Loss function is a function that is optimized during training. We implemented one commonly used loss function: Mean Squared Error. For matrix outputs we can define it as

$$MSE(Y, \hat{Y}) = \frac{\|Y - \hat{Y}\|_2^2}{N}, \tag{9}$$

where $\hat{Y}$ are true labels, Y is the output of a network, $\|\|_2$ denotes L^2 (Frobenius) norm. Its gradient can be written as

$$\frac{\partial L}{\partial Y} = \frac{2 \cdot (Y - \hat{Y})}{N}. \tag{10}$$

V. Dataset

We generate a training and a test sample of 1000 uniformly distributed in $[0, 1]^2$ points each. The target label is 1 inside the circle of radius $\frac{1}{\sqrt{2\pi}}$ centered at $\left(\frac{1}{2}, \frac{1}{2}\right)$ and 0 outside. To compute the target output, we first compute binary (-1 or 1) label as

$$sgn\left[(x - 0.5)^2 + (y - 0.5)^2 - \frac{1}{2\pi}\right] \tag{11}$$

Here the label is -1 if the point is inside the circle and 1 otherwise. Then we concatenate $1 \times N$ tensor of binary labels with itself multiplied by -1, add 1 to all values and divide them by 2 in order to create $2 \times N$ resulting tensor with one-hot encoded target output.

The area of the circle is $\frac{1}{2}$, therefore, the probability of a random point to be inside it is also $\frac{1}{2}$. Hence, the classes are balanced and accuracy could be a good metric to evaluate the performance of our model.

VI. Model structure and training

The model, as it was stated in the task, the model must have 2 input and 2 output units and, three hidden layers of 25 units. For the activation functions we have chosen *Tanh* for the last layer so that all outputs were in range (0, 1) and *ReLU* for other layers.

To facilitate the training process we created a function `train_model` that trains a given model using mini-batch stochastic gradient descent. One training cycle consists of (for each mini-batch):

1) Zeroing the gradients
2) Computing outputs (forward pass)
3) Computing and accumulating loss
4) Computing the gradient of the loss with respect to the outputs
5) Computing gradients of the loss with respect to the parameters (backward pass)
6) Updating the parameters (optimizer step)

Batch size, learning rate, loss, number of epochs are function parameters and can be varied. We also set default values for them to be able to call the function with only 3 arguments: model, training set and target outputs. During training the sum of the losses over the mini-batches is printed for each epoch (can be disabled by setting the parameter `print_loss` to False). To test model accuracy, the function `accuracy` accepting true and predicted values is available.

VII. Results

Training the model described in VI with 250 epochs and batches of size 100 takes less that one second. It is worth noticing that due to the fact that we used mini-batch stochastic gradient descent, the loss is not always monotonically decreasing. The model achieves 97.8% accuracy on the training set and 95.9% accuracy on the test set which means the model does not overfit. All misclassified points in both sets lie near the border of the circle, as we can see on the figures.

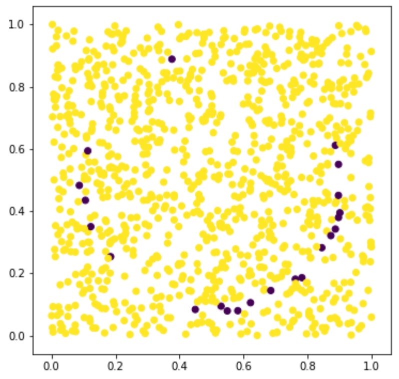

Figure 1. Misclassified points in the training set

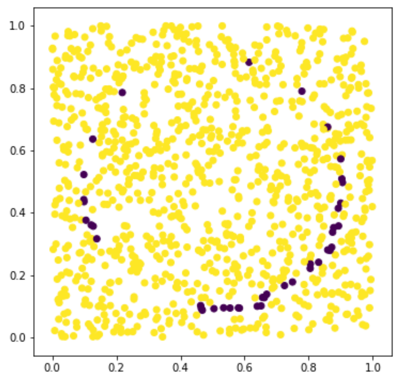

Figure 2. Misclassified points in the test set

VIII. Conclusion

Overall, the presented framework suffices to solve the given task. One of the possible drawbacks caused by disabled autograd is the fact that each layer has to store a copy of input data to do back-propagation which can, in some cases (e. g. deep networks), be an enormous amount of memory. However, for this particular task this is not a problem. To improve our framework, it would be good to implement some other layers and loss functions, especially for classification tasks (e. g. softmax, cross-entropy).