Project 2 – Mini deep-learning framework

Maksim Kriukov
SCIPER:313810
SV Section

Fedor Moiseev
SCIPER: 309259
SC Section

Simona Leserri
SCIPER: 312970
SV Section

May 22, 2020

Abstract

We implemented a mini "deep learning framework" using only PyTorch's tensor operations with completely disabled autograd function. The built framework contains basic modules of typical neural network and is able to calculate both forward and backward passes similarly to PyTorch's way. To test functionality of the framework we generated a "toy" dataset of 1000 samples with 2 randomly distributed features that belong to either of 2 classes. Using the implemented framework we constructed a 3-layer perceptron which is optimized using SGD for MSE. As a result of a training over 200 epochs, we achieved final train error = 0.032 and final test error = 0.042 (both errors are rounded to 3 decimal places).

1 Framework description

1.1 General structure of the framework

Overall, our mini "deep learning framework" contains all fundamental and basic modules with additional sigmoid function, binary cross entropy loss and dropout layer. To be more precise we implemented:

- Sequential container - class `Sequential`
- Layers: fully-connected layer - class `Linear`, dropout layer - class `Dropout`
- Losses: mean squared error (MSE) loss - class `LossMSE`, binary cross entropy (BCE) loss - class `lossBCE`
- Activation functions: rectified linear unit (ReLU) - class `ReLU`, hyperbolic tangent function - class `Tanh`, sigmoid function - class `Sigmoid`
- Optimizers: stochastic gradient descent (SGD) - class `SGD`

We implemented every module of the neural network using oriented object programming. Every module class, except for optimizers, is inherited from base class called `Module` that contains 5 methods common for all child modules. Among them `forward()` method calculates the forward pass of the module - computes its output with given input, while `backward()` method calculates backward pass of the module - computes gradients of the module parameters with respect to the output and back-propagates the gradient. It also computes the gradients with respect to parameters of the module for the following optimizer step. `backward()` should be always called after `forward()`, because it can rely on some information that is stored from the forward pass. `param()` method returns the list of pairs corresponding to the network parameter values and gradients with respect to the network parameters. If the module does not have parameters `param()` returns empty list. The parameters of the module are stored in the attributes of the class (e.g. in `Linear` class weight matrix and bias are stored in `Linear.W` and `Linear.b` respectively). Finally, base class `Module` additionally contains `train()` and `eval()` methods that respectively turn on or turn off training mode of the module. This feature is implemented for integration of all modules with dropout module, which requires an explicit flag to disable random network units (neurons) during training or enable them during inference.

All optimizers are inherited from class `Optimizer`. This base class contains two methods - `step()` and `zero_grad()`. The former method computes optimizer step and updates the passed network parameters. When created, an `Optimizer` object explicitly receives the network parameters (list of pairs $(parameter, grad)$, e.g. output of the `Module.param()` method) so that after each epoch the parameters can be updated. `zero_grad()` method resets gradients with respect to parameters.

The class inheritance scheme of all modules is represented on Fig. 1.

1.2 Helper functions

We additionally implemented two helper functions `generate_data()` and `batchify()`. While the first generates data from the toy dataset, the second executes batch processing. More precisely, it randomly splits the generated dataset into batches to speed up the computation.

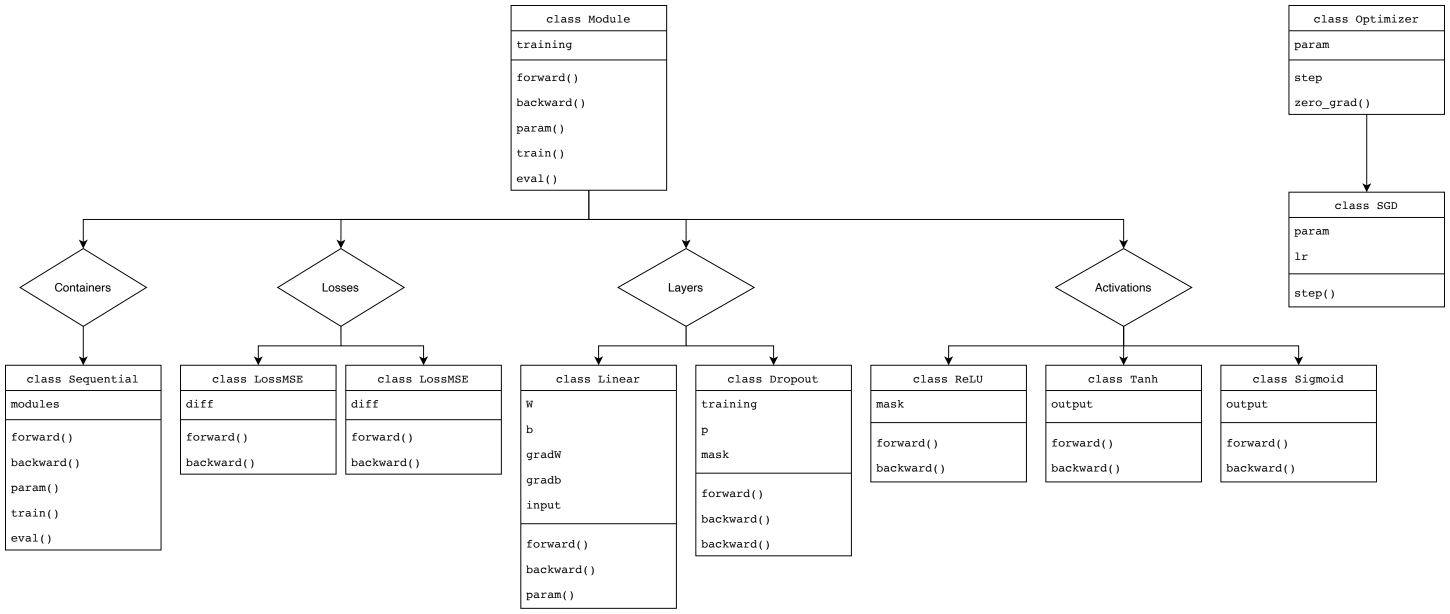

Figure 1: The class inheritance scheme of all implemented modules. Below the class name class parameters/attributes and methods are listed for each class.

1.3 Activation functions

All implemented activation functions are element-wise. We are not going to present all their calculations in details, despite them being straightforward. Notably, ReLU module saves a mask that represents elements of the input that are positive during forward pass, while Tanh and Sigmoid directly compute and save the output of the layer during forward pass. Stored attributes are used to effectively perform backward pass.

1.4 Linear layer

The Linear layer takes as input a tensor with [Batch_size, Dim_in] size and applies a linear transformation of the features, resulting in an output tensor of size [Batch_size, Dim_out]. During forward pass the following function is applied:

$$T_{\text{out}} = T_{\text{in}} \cdot W^{\text{T}} + b, \tag{1}$$

where T_{out}, T_{in} are output and input tensors respectively, W - weight matrix, b - bias.

Backward pass, in turn, computes gradients with respect to parameters and back-propagates input gradients:

$$\begin{aligned} \nabla W &= grad_{out}^{\text{T}} \cdot T_{in} \\ \nabla b &= \sum_{n=1}^{\text{Batch\_size}} (\text{grad}_{\text{out}})_n \\ \text{grad}_{\text{in}} &= \text{grad}_{\text{out}} \cdot W, \end{aligned} \tag{2}$$

where T_{in}, T_{out} are again input and output tensors respectively, grad_{out} is the gradient with respect to the output of the layer, grad_{in}, is the gradient with respect to the input of the layer, W - weight matrix, b - bias, $\cdot$ represents matrix multiplication.

Parameter weights are initialized by sampling from the uniform distribution $\mathcal{U}[-1/\sqrt{\text{Dim\_in}}, 1/\sqrt{\text{Dim\_in}}]$ (it is some sort of Xavier initialization).

1.5 Dropout layer

During training, a dropout layer randomly sets to zero some of the elements of the input tensor with probability p (by default $p = 0.5$) sampling from a Bernoulli distribution. To be more precise, it creates a mask with a size of input tensor that has randomly distributed zeros and ones. In the training mode the output of the module is the result of element-wise mask multiplication with the input tensor divided by (1 - p) - this latter trick preserves expectation of the sum. The mask is stored to be used during the following backward pass. During backward pass we apply saved mask to the gradient with respect to the output and divide by (1-p). In the evaluation mode dropout does not change the input.

1.6 Sequential container

During forward pass Sequential container iteratively applies `forward()` method of all children from the first to the last. During backward pass it iteratively applies `backward()` method to all children from the last to the first. Input of the current call works as the output of the previous one in both cases. The container can also iteratively set either training or inference mode and returns all parameters of the network.

1.7 Loss functions

Loss functions differ from other modules, because their `backward()` method has no parameters. This method `backward()` computes gradient of loss function with respect to input using information stored during forward pass. We save difference between input and target for MSE loss and just input and target for BCE loss. The `forward()` method of loss functions takes two arguments: prediction and true targets. It also averages the output by batch dimension.

1.8 Training procedure

The training procedure of the implemented framework is very similar to PyTorch's way. Firstly, we define network architecture by using `Sequential` class and explicitly passing layers to the container. Then we define the loss function that we aim to minimize. Next, we define the optimizer function by passing directly the model parameters using `model.param()` method. By doing so, during training the optimizer is able to update the model parameters as both optimizer and model are linked to the same object. Then we put the model into training mode. The whole training is processed in batches. For each batch we reset the gradients, calculate the output of the model using forward pass, calculate the loss, compute the backward pass starting from the loss and update the parameters using optimization step function.

2 Framework testing

To check if the implemented model worked we generated a "toy" dataset of 1000 samples that are distributed randomly in $[0,1]^2$ with label 0 if outside the disk centered at $(0.5, 0.5)$ or radius $1/\sqrt{2\pi}$, and 1 inside. To make a model that could classify samples we implemented a multi-layer perceptron (MLP) with 3 hidden layers of 25 units, ReLU activation and MSE loss function that predicts the sample class. The network parameters were optimized with SGD (learning rate 0.1) by batches of size 100 for 200 epochs. Both generated data and the training curve are presented on Fig. 2. To reproduce these results use `test.py`[1] file. The final metrics of the model performance are: train error $= 0.032$ and test error $= 0.042$ (both errors are rounded to 3 decimal places).

Moreover, we also created another network to test other elements of out framework. It has the following structure: `Linear(2, 25)` $\rightarrow$ `ReLU` $\rightarrow$ `Linear(25, 25)` $\rightarrow$ `ReLU` $\rightarrow$ `Dropout(0.3)` $\rightarrow$ `Linear(25, 25)` $\rightarrow$ `Dropout(0.5)` $\rightarrow$ `Tanh` $\rightarrow$ `Linear(25, 1)` $\rightarrow$ `Sigmoid` and the optimised loss is binary cross entropy. Trained with the same setup as the first network, it achieved train error $= 0.035$ and test error $= 0.035$. You can reproduce this using `test_advanced.py` file.

3 Conclusion

In this project 2 we implemented a mini "deep learning framework" that comprises of basic modules used in a typical neural network. We tested its functionality in a binary classification task and showed that it is able to converge and make predictions with high accuracy.

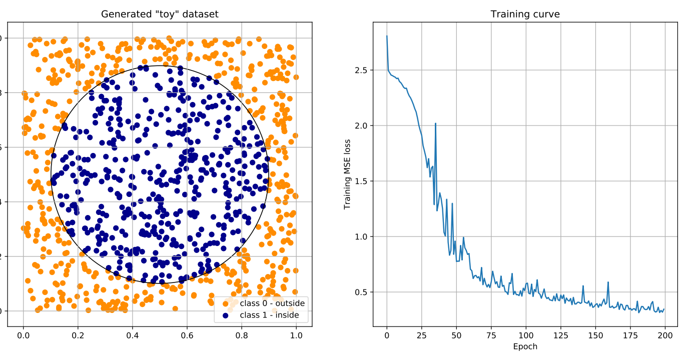

Figure 2: The generated dataset and training curve of the model.

[1]In some files of our framework we import `torch`, but we never use `nn` or autograd features. The only functions from `torch` that we use are `manual_seed, set_grad_enabled(False), rand, exp, clamp, zeros_like, ones_like`