Deep Learning

Mini Project 2: Mini Deep Learning Framework

EE-559

Loïc Ferrot
Florian Richter

École Polytechnique Fédérale de Lausanne
May 28th, 2021

1. Our framework

We designed this framework with the mindset that we wanted to understand the flow of information of the training procedure, and not reinvent the wheel, since anything that we would produce would likely not be used in the future.

This is why our framework is oriented towards functionality rather than user friendliness, and needs a bit more work from the user than pytorch's autograd.

As suggested in the instructions, all of our modules inherit from one `Module` class, with, in addition to the suggested `forward`, `backward` and `param` methods, a `sgd` (for stochastic gradient descent) and a `zero_grad` method, that are optionally overwritten. Our framework supports the batch processing functionality.

A `StructFoo` empty class is used inside the modules to semantically separate the variables into a "parameter" `param` namespace and a "backward pass and gradient related" `grad` namespace.

For example, among others, the `Linear` module contains the member variables `param.weights` and `param.biases` that are the trainable parameters as well as `grad.weights`, `grad.biases` that contain their corresponding gradient and `grad.data_in` that stores the input data during the forward pass.

2. Implemented modules

2.1 Linear module

This module implements the basic fully connected layer.

The weights (`param.weights`) are initialized using Xavier's rule, with a non-linearity-specific gain, i.e. the weights follow a zero mean Gaussian distribution with a standard deviation depending on the input and output dimensions and on the type of the non-linearity to follow. Hence, when instantiating the `Linear` module, it is necessary to specify to which non-linearity the output of the module is connected to. At first when this initialization was not implemented we had problems of vanishing gradient, and using it helped a bit. The biases (`param.biases`) are just initialized with a normal Gaussian distribution (zero mean, standard deviation equal one).

During the forward pass, the input data is stored in the `grad.data_in` variable. During the backward pass, using the following module's gradient taken as input and the `grad.data_in` variable, the parameters' gradient is accumulated in the `grad.weights` and `grad.biases` variables. The gradient of this module is returned, so the backward propagation can continue with the previous module.

The training update itself is done using stochastic gradient descent with momentum, by calling the `sgd` method with an `eta` step size parameter and an optional `momentum` parameter. If the optional `momentum` is not specified, its value is equal to zero and the optimization is equivalent to the basic sgd. The "velocity" of the parameters are themselves stored in the `grad.weights_speed` and `grad.biases_speed` variables at each update. The contribution of the velocities to the parameters' update is weighted by the `momentum` parameter.

2.2 Non-linear functions

In addition to the requested `ReLu` and `Tanh` non-linearities, we implemented the `Leaky_ReLu`, `eLU`, `SeLu` and `Sigmoid` functions. All these functions do not have trainable parameters, and the only member variable they

contain is the `grad.data_in` saved during the forward pass and used in the backward pass. `ELu`, `SeLu`, and `Leaky-ReLu` have parameters that can be set when instanciating the activation function. These parameters are : *slope* of `Leaky-ReLu` (default 0.2), *alpha* for `ELu` (default 0.2), *alpha* and *lambda* for `SeLu` (default 1.6733 and 1.0507 respectively)

2.3 Losses

In addition to the requested `LossMSE` loss function, we implemented the `LossBCE` loss function. For clarity's sake, we made the choice of not storing any internal member variable. Therefore, when calling the backward method, the output of the network and the target have to be given as input. Indeed, the `forward` and `backward` methods are usually called close to each other in the code, so the `output` and `target` input variables are still available when calling `backward`.

2.4 Sequential

This module simply stores all the member modules and chains their output to the corresponding input during the forward and backward passes, and calls individually the corresponding methods in each module (e.g. `zero_grad`).

Since the loss function requires a `target` parameter, we thought it would be clearer and simpler not to put it in the `Sequential` module. This is why, during training, we need an intermediate variable to temporary store the gradient of the loss to pass it to the rest of the modules.

3. Using our framework

The script `test.py` contains a function `run_miniproj2` that trains and tests the network described in the instructions, with an input of size two, three hidden layers of 25 units and an output unit of size one. Each layer is followed by different activation functions which are, in order, ELU, ReLU, SeLU and Sigmoid. We chose these activation functions more "because we can", rather than following a specific logic. As additionally specified by the instructions, the loss function is a Mean Square Error. The optimization is done with stochastic gradient descent with a momentum of 0.2. Using the default parameters of the activation functions, 1000 train samples, 1000 test samples, 100 epochs and a batch size of 100, the network reaches a mean train error on the last epoch of 0.92% with 0.36% standard deviation, and test error of 2.38% with 1.04% of standard deviation on a 100-fold cross-validation.

We had to face a little challenge with the network built thanks to our framework. Sometimes the stochastic gradient descent gets stuck in a local optimum across the epochs, with a relatively high loss and around 50% of test error, doing no better than a random predictor. Since the network struggles to get out of the optimum by itself, we add a condition to reset the parameters if the loss difference between two epochs is close to zero and if the performance is below a threshold. This little added condition successfully allows the network to converge towards a decent result, even with a reduced number of epochs left for training.

Finally, we think that it was great to have such an assignment for a graded report, since we dug deeper in the mechanisms of gradient back propagation than we would have had the courage to if the assignment was not graded. Thus, we now understand better the core mechanism behind pytorch's autograd, and deep learning in general.