Guarin Flück
Marc Glettig
Peter Oriane

Mini Project 2
Deep Learning

2020-06-03

1 Introduction

We created a deep learning framework that mirrors some of the functionality of PyTorch [1] without exploiting the automatic differentiation mechanism. Instead, all of the required functions to keep track of gradients are implemented from scratch and assembled in easy to use modules. Using the framework, we were successfully able to predict labels on a simple dataset consisting of points in $[0,1]^2$ that fall into class 1 if they are less than $\sqrt{2/\pi}$ units away from the origin and class 0 otherwise.

2 Framework

2.1 Structure

Our package follows the structure of PyTorch and all building blocks are accessible from the `nn` package. Each network element falls into one of two base classes: `Module` or `Optimizer`. A `Module` is either a layer, an activation function or a loss. `Loss` is a subclass of `Module` as it requires a `target` argument in the forward pass. A class implementing `Optimizer` is always an optimizer function. Figure 1 shows all elements implemented in the framework.

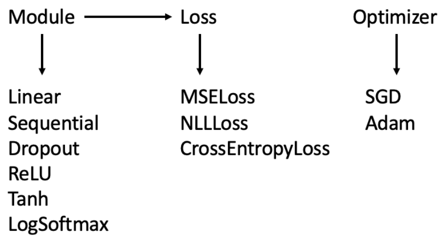

Figure 1: Class structure of the framework. Arrows point from super- to subclass.

2.2 Modules

Each module must implement a `forward` and a `backward` method. The `forward` method returns the output of a module from an input. Given the gradient of the loss with respect to the modules output, the `backward` method calculates the gradient with respect to the modules input. Unlike PyTorch, the `backward` method has to be explicitly defined for each module as the framework does not have access to the autograd functionality of tensors.

Parameters of the modules, such as weights, are implemented using the `Parameter` class. Each parameter has two `Tensor` attributes, the `value` attribute holds the ”value” of the parameter and the `gradient` holds the gradient of the loss with respect to the `value`. As we only consider scalar losses, the `gradient` of a parameter always has the same dimensions as the parameter itself. The `backward` method is also responsible for updating the gradients of its parameters. Table 1 lists all modules and their attributes. Each module holds only as few attributes as necessary.

2.2.1 Linear / Fully-connected Layer

The `Linear` module implements a fully connected layer with a fixed input and output size. The `weight` is created using either Xavier [2] or He [3] initialization to avoid vanishing gradients. The `bias` is initialized to all zero values. The `forward` method also keeps track of the last `input` it was called with, which is saved in the `_input` attribute. This is required to calculate the gradient of the `weight` parameter and the gradient of the module with respect to the input in the backward pass. The actual updating of the parameters is handled by the `Optimizer`.

2.2.2 Sequential

The `Sequential` module takes a list of modules as input, here called submodules. It represents a sequence of modules called one after the other, with the output of the first module used as the input of the second module and so on. This is implemented by the `forward` method. The `backward` method traverses the submodules in reversed order and lets the gradient backpropagate through each individual submodule.

2.2.3 Dropout

The `Dropout` module sets a user-defined percentage of the activations of the input to zero. This is achieved by random Bernoulli variables and a rescaling of the non-zero activations by a factor of $1/(1-p)$ where p is the probability of an activation being set to zero. The refactoring is necessary to maintain a constant sum over all activations. The module keeps track of the values sampled in the forward pass and reuses them in the backward pass. A special `training` attribute is set on all modules to indicate whether the network is in train or test mode. The dropout module acts as an identity function when `training` is set to `False`. Changing between train and test mode is done by calling `module.train()` and `module.eval()` respectively.

2.2.4 Activation Functions

Three activation functions are implemented: `ReLU`, `Tanh` and `LogSoftmax`. `ReLU` returns the input with negative values set to zero. `Tanh` applies an element-wise hyperbolic tangent to the input and `LogSoftmax` applies a numerically stable element-wise softmax function to the input and returns the log of it. `LogSoftmax`

Table 1: Module attributes and their types. Input and target refer to the arguments of the forward method.

Module	Attribute	Type
Linear	weight	Parameter[out_size, in_size]
	bias	Parameter[out_size]
	_input	Last input
Sequential	modules	List[Module]
Dropout	_bernoulli	torch.distributions.Bernoulli
	_sample	Last sampled values from distribution
ReLU	_input	Last input
Tanh	_input	Last input
LogSoftmax	_softmax	Softmax of last input
MSELoss	_diff	Difference between last input and target
NLLLoss	_target	Last target
	_shape	Shape of last input
CrossEntropyLoss	_log_softmax	Module[LogSoftmax]
	_nllloss	Module[NLLLoss]

keeps track of the softmax of the input while `ReLU` and `Tanh` store the input without any manipulation to be used for the calculation of the gradient.

2.2.5 Losses

`Loss` modules are a subclass of `Module` as their `forward` method does not only require an `input` but also a `target` argument. All losses return mean values over the batch. The losses implemented are mean squared error (`MSELoss`), negative log-likelihood (`NLLLoss`) and cross entropy loss (`CrossEntropyLoss`). The cross entropy loss is simply a `LogSoftmax` layer followed by a `NLLLoss`.

2.3 Optimizers

Optimizers are responsible for updating module parameters. They have a `parameters` attribute that holds all parameters of the network. Only the `step` method has to be implemented. Calling the `step` method triggers the optimizer to go through all the parameters and update them according to the algorithm implemented by the optimizer.

The `SGD` optimizer is a simple stochastic gradient descent which requires a `learning_rate` and an optional `momentum` argument on initialization. If the `momentum` is above zero the optimizer will also create a momentum tensor for each parameter. The momentum adds an exponentially decaying weighted average of past gradients to the current gradient.

The `Adam` [4] optimizer utilizes the moments of the gradient and squared gradient for faster convergence. It follows the classical formula and requires two extra tensors for each model parameter, thus doubling the total memory requirements to train the network.

3 Results

Different network architectures and optimization strategies were tested on artificially generated training and test data. The base network contains three fully-connected layers with 25 hidden nodes each, initialized using He initialization and ReLU activation functions. An alternative network with a dropout layer after each activation function was also explored. The networks were trained using mean squared error and cross entropy loss. Both optimizers implemented in the framework, SGD and Adam, were used for training.

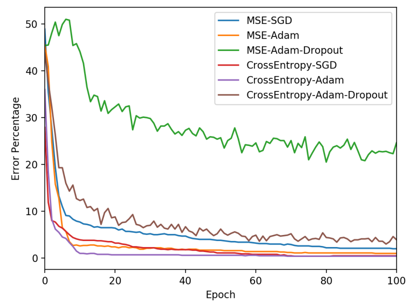

Figure 2: Train errors of different network architectures and optimizers for the first 100 epochs.

Figure 2 shows the training errors for the different architectures. The results are as expected, Adam reaches faster convergence than SGD and the models with dropout have higher error rates. But overall the task is too simple to compare the various architectures and we did not explore hyperparameters. The test results for the base network are 1.7% train and 2.4% test error after 250 epochs. The network using cross entropy loss and Adam performs the best and reaches 0.2% train and 1.2% test error.

4 Validation

The framework contains a `test_nn.py` file which is used to validate the implemented modules. The testing code creates an instance of each module and compares the outputs of the `forward` and `backward` passes with the outputs of the corresponding PyTorch module on some random input. Furthermore, it also compares all module parameters and their gradients. This is achieved by creating a base test class `ForwardBackwardTest` that contains the logic to run a `forward` and `backward` pass on any module. For each `<Module>` class a `Test<Module>` class was created as a subclass of `ForwardBackwardTest`. Each class just contains the code to create the specific module and its PyTorch counterpart. From there on the `ForwardBackwardTest` takes over and raises an error if any differences between the modules are found. The modules implemented in our framework successfully pass this test suite.

5 Conclusion

The project allowed us to learn how PyTorch works in depth and how to implement all the basic building blocks required for deep learning. We found that there is only little code necessary to cover a lot of functionality and adding further modules is straight forward. The validation suite helped a lot to make sure that each part works as expected and revealed many hidden bugs.

References

[1] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[2] Xavier Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research - Proceedings Track*, 9:249–256, 01 2010.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015.

[4] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.