# Deep Learning - Project 2

# Deepy

Bassel Belhadj, Charles Gallay, Gregoire Clement

## I. Abstract

In this report we present a simple but functional automatic differentiation library for Deep Learning called Deepy. It builds upon PyTorch [1] but without the use of its autograd module. This library allow the programmer to define basic neural networks, train them with back propagation [2] and use them for inference.

## II. Background

Nowadays, the landscape of deep learning framework is quite vast. To name only a few of them we have Tensorflow [3], PyTorch, Keras [4], Caffe [5]. All of them have their strengths and weaknesses, but they can be categorized into two big categories depending on how they construct their computational graph (See subsection IV-A for a definition).

- **Dynamic:** The computational graph is define on the fly by running the code and applying the operations on the tensors. The forward and backward pass are created for each batch, this gives a certain flexibility where the programmer can easily decide to change the froward pass any time during the training.
- **Static:** On the other hand the computational graph can be computed in advance and used for the forward and backward pass. Precomputing the graph allows for better optimization and a lower memory usage but gives less freedom to the user.

Deepy defines it's computational graph on the fly. Making it a define-by-run framework like PyTorch or Chainer [6] before it. Others libraries like Tensorflow (before the introduction of Eager mode) and Caffe define and use static graph for the computations.

## III. Deepy Architecture

### A. Modules

Here we present the different *Modules* that compose the library. For more details about the *Modules* please look for comment into the code directly.

- **Linear:** This *Module* define a fully connected layer and an optional bias. The forward pass is define as:

$$Y = XW + b$$

- **Tanh & ReLU:** Basics non linear function to serve as activation layer.
- **Mean Square Error (MSE):** A loss function for regression problems. Compute the

### B. Variable

A *Variable* is the most important class of the library. It contains two tensors which are implemented using PyTorch tensors, the data and its associated gradient, also it contains some meta-data. The main meta-data present in a *Variable* are listed in figure 1 and are used only during the backward pass.

One of the interesting meta-data is the 'grad_fn' (of type Function, see III-C). When the *Variable* is not a leaf of the computational graph, i.e it does depend on other variables, a reference to the operation that created the *Variable* is kept. Also, in order to compute the gradient during the backward pass, context information is encoded inside the 'gard_fn' as well.

In order to save up some memory we intentionally discard the gradient of non leaf *Variables*.

It implements a backward methods that permit the user to compute derivative of the *Variable* with respect to any *Parameters* that influenced that *Varaible*.

### C. Function

This class is invisible to the end user of the library. Nevertheless, it plays a central role during the backward pass. Its goal is to keep track of the *Variable* history, namely which previous operations (*Module*) created a *Variable* as well as enough context information to compute the gradient during the backward pass.

### D. Parameter

This class is used to represent the trainable parameters. In fact, it is a just a *Variable* which, when used inside of a *Module*, is automatically added to its parameter list. Therefore removing the needs for the user to implement a 'parameters()' method each time that a new subclass of *Module* is defined. This process eases the access to trainable parameters.

### E. Optimizer

The only optimizer implemented for now is the vanilla stochastic gradient descent known as SGD. An optimizer keep track of all the parameters that need to updated. Therefore, each time the optimizer 'step' method is called, the weights of all the *Parameters* with requires_grad set to True are updated. It is possible to set a weight_decay which adds L2 regularization to the update process. Once a step is perform, one might want to reset the gradient accumulator to zero, this can be done using the 'zero_grad()' methods (See VI-B for an example).

## IV. Computational Graph

### A. Definition

A computational graph defines in an ordered manner all the operations that you need to perform on the inputs in order to compute the output for a given model. It can be represented as a directed graph where nodes are either tensors or operations. Edges of graph goes from tensor to operation and from operation to tensor. This creates a history of all the basic operations used to compute the output *Variable*. This way it is easy to see which inputs has an influence on which output simply following the graph backward.

### B. Deepy Computation Graph Construction

As mentioned previously, Deepy follow the **define-by-run** paradigm. During the forward pass each time an operation is performed on a *Variable*, a context aware object is created and the output of the operation is stored in a new *Variable* (see Fig. 1).

### C. Debugging

In order to better understand a model the user has the possibility to get a string representation of the computational graph for any model. To do so one just need to provide a dummy input to the ‘graph_repr‘ function in the ‘utils‘ package.

## V. Extending the library

Adding a new *Module* is quite simple. A contributor only need to implement two classes, the first one is extending the *Module* class and describes the forward pass. The second one is extending *Function* which contains the derivative of the *Module* forward (i.e the backward pass) and context information, such as the input of the *Module*, in order to compute the gradient.

During the forward pass, the *Module* creates a new *Variable* that shares the data, but whose ‘grad_fn‘ is an instance of the *Function* corresponding to the *Module*.

## VI. Example of a run

### A. Network definition

In order to create a network the user needs to define a class that extends ‘deepy.nn.Module‘.

```
from deepy.nn import (Module, Relu,
                      Tanh, Linear)
class MyNet(Module):
    def __init__(self):
        self.l1 = Linear(784, 50)
        self.a1 = Relu()
        self.l2 = Linear(50, 10)
        self.a2 = Tanh()

    def forward(self, x):
        out = self.l1(x)
        out = self.a1(out)
        out = self.l2(out)
        out = self.a2(out)
        return out

net = MyNet()
```

A special *Module* named Sequential allow the user to define simple network in a way such that the output of the first layer is the input to the second one and so on. For example the same network as before could simple be define as below.

```
net = Sequential([
        Linear(784, 50), Relu(),
        Linear(50, 10), Tanh()])
```

Note for example that for Sequential networks one could easily implement a static computation graph where the backward pass of each *Module* is called in the inverse order.

### B. Training Loop

This illustrate a simple training loop. Each *Module* are callable, they expect as input of type *Variable* and shape [batch_size, in_features] where the batch can be of any size.

```
from deepy.loss import MSE
from deepy.optim import SGD
from deepy.tensor import Variable

optimizer = SGD(net.parameters(),
    lr=0.1, weight_decay=0.01)
criterion = MSE()

for i in range(nb_epochs):
    optimizer.zero_grad()
    x = Variable(train_input)
    out = net(x)
    loss = criterion(out, train_target)
    loss.backward()
    optimizer.step()
```

## VII. Conclusion

With this paper we presented our way of implementing the back-propagation algorithm that allow the user to compute the derivative with respect to any *Variable* computed from a combination of *Variables* and *Modules*. With that we are able to define simple but fully functional neural network and train them to perform deep learning task.

## References

[1] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” 2017.

[2] R. J. W. David E. Rumelhart, Geoffrey E. Hinton, “Learning representations by back-propagating errors,” 1986.

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[4] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[5] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[6] S. Tokui, K. Oono, S. Hido, and J. Clayton, "Chainer: a next-generation open source framework for deep learning," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015. [Online]. Available: http://learningsys.org/papers/LearningSys_2015_paper_33.pdf

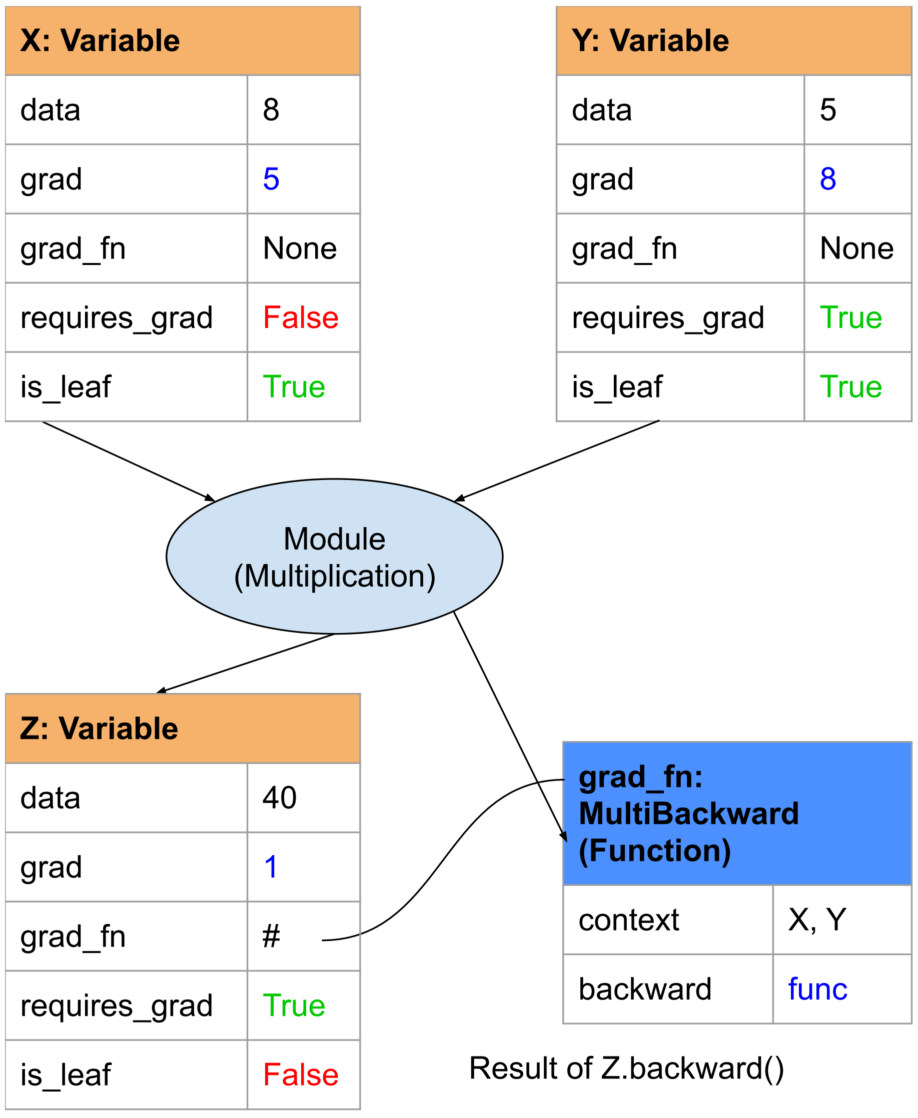

Fig. 1. Example of a simple Computational graph for the multiplication of two *Variable*