Hotgrad: a deep learning framework

NICCOLÒ SACCHI, SUCCA RICCARDO & MARCO ZOVERALLI
École polytechnique fédérale de Lausanne

May 12, 2018

***Abstract*—This project aims at designing and implementing a simple deep learning framework based only on pytorch Tensors and standard math libraries. Such a framework consists in implementing backward compliant operations so that all the needed gradients can be effectively computed. We achieved this goal by implementing forward and backward functions for the basic operations, i.e. (1) operations between two tensors (e.g. matrix multiplication) and (2) operations on a single tensor input (e.g. ReLU). This approach lead to a straightforward implementation of all the needed operations while keeping the framework versatile and easy to understand. Finally, this framework had been tested by implementing a simple neural network trained on a two dimensional dataset and comparing the performance with some baselines.**

I. INTRODUCTION

A neural network consists in a series of layers, which compute a linear mapping of their input. In order to improve the capacity of the model, non-linear functions are usually introduced after each linear layer. These non-linear functions avoid that the output of the network can be reproduced from a linear combination of the input, which leads to poor accuracy when the input dataset is not linearly separable (classification) or the target is not linearly dependent on the input (regression). When a neural network is created its parameters are initialized randomly, therefore, in a 2-class classification problem, the accuracy of the network is initially around 50%. The latter is improved during the learning phase which is based on three main steps repeated for a certain number of iterations: (1) the *forward pass*, (2) the *backward pass* and (3) *parameters update*.

1) The *forward pass* can just be seen as a series of function applied in sequence to the input in order to compute the output of the network y:
$$y = f(g(h(...(x)))$$
2) Then a loss function is used to measure how much the output of the network differs from the expected output. The *backward pass* computes the gradients of the loss with respect of the network parameters by applying the chain rule.
3) Finally, during the *parameters update* step, all the computed gradients are used to update the respective network parameters with the final goal of minimizing the train loss. For example, the stochastic gradient descent uses the following formula to update a parameter:
$$p = p + \eta \nabla L(p)$$
Where p is a trainable parameters, η the step size and L the loss function.

In a fully connected layer the trainable parameters consist in the weights and the biases.

$$y = X * W where y....$$

y taken from R ...

These general steps can be applied to any differentiable structure. It is therefore important for a deep learning framework to be easy to use and highly versatile so that various operations can be freely combined in order to construct both simple and more complex neural networks. Our approach focused on these features and aimed to implement the more complex operations (e.g. linear layer, MSE loss) by breaking them into basic operations between tensors and making them backward compliant by simply implementing the gradient of their output with respect of their input.

II. IMPLEMENTATION

Since this project provided the structure of the final neural network, a simple approach to the problem would have been to directly implement the forward and the backward for the whole given network. This would lead to a simple but difficult-to-read implementation, without any possibility to expand or modify the network. We immediately discarded this option and we instead focused on a modular approach. The backward propagation can be easily computed by iteratively applying the chain rule and propagating the computed gradients starting from the very last operation, i.e. the loss. The following formula shows how the chain rule is applied to compute the gradient of a composite function: Particular attention has to be payd to the shapes of the tensors when applying this formula. With this in mind, we understand that to be able to propagate the gradients all the way back to the parameters we need to store all the acyclic graph of operations that lead from the input to the computation of the loss. We achieve this goal with two simple ideas:

add chain rule Vx(f(g(x))) = Vy(f(y))*Vx

- Each *operation* should keep track of its inputs other than computing the result of the operation. In this way it can compute the derivative of its output with respect of its input, multiply it with the current gradient and pass it to the corresponding input. This operation has to be repeated for each input of the operation.
- In turn, each *operand* should (1) be able to propagate the gradient to the operation that computed the operand itself and (2) store (and accumulate) the received gradient representing the gradient of the loss with respect of the operand. We achieved this by implementing a class called *Variable* to wrap all the necessary information: the operand itself (which consists in a torch FloatTensor), its gradient and the link to the operation that generated it.

This structure allows to perform both the forward and backward passes. In the **forward pass**, each operation takes its inputs Variables and produces a new output Variable which contains the resulting tensor, a gradient tensor (initialized to zeros) and the link to the operation itself. Then, this new Variable can be used as input for any subsequent operation, and the process is repeated. Starting from the input, this chain of operations continues until the computation of the loss. In the **backward pass**, each operation receives the gradient from its output Variable, i.e. the gradient of the loss with respect of it, and multiply it with the gradient of the output Variable with respect of each input Variable. The obtained gradients are finally passed to the relative input Variables. Whenever a Variable receives a gradient, it accumulates it in its gradient field and then passes it to the previous operation if any.

With this structure in mind, we implemented the base operations, i.e. the activation functions ReLU and Tan and addition, subtraction, element-wise multiplication, matrix multiplication, pow and mean. This can be seen as a bottom up approach, where we break the problem into very basic operations, and then use them to construct more complex modules. The graph shown in picture 1 displays how this modules and variables are used to compute the forward and backward pass.

add link to image

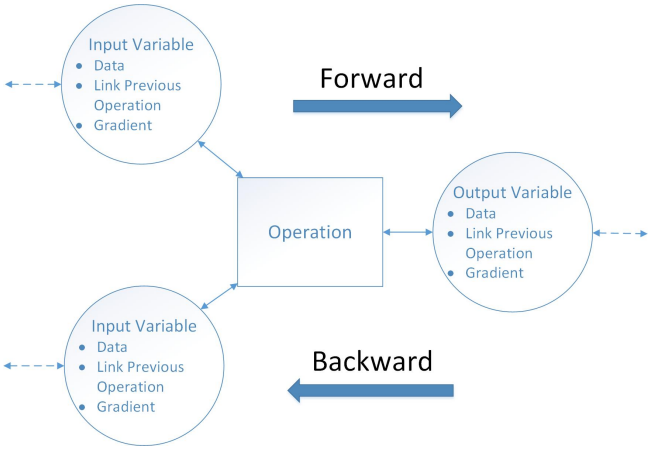


Figure 1: Forward: The Operation Module generates a new Variable, containing the result and the link to the operation itself. Backward: The Operation Module computes the new gradients and sends them to the respective Input Variables.

III. Code Structure

We now describe in details the structure of the code and the classes you can find in our library folder *hotgrad*:

- *module.py*: this file contains the abstract class Module, with the basic functions that each module will implement, in particular the forward and backward pass. This Module class is extended by two other classes, Module2Operands and Module1Operand, that distinguish between operations that respectively require one or two inputs.
- *exception.py*: contains the exceptions needed to handle the errors.
- *optimizers.py*: as optimizer, we implemented the Stochastic Gradient Descend (SGD), initialized with a learning rate parameter. The method *set_params* allows to register the list of all the trainable network parameters, whose update is performed by the *step* function.
- *sequential.py*: this module is used as a container of other modules. It receives as input the list of modules that compose the network, the loss criterion and the optimizer. Unlike the other modules, Sequential does not implement the backward pass. It has a *fit* method that is used to train the network on a given train data and for a specified number of epochs. Each epoch correspond on a pass on the whole training data, however, in order to speed up the performance by updating more often the parameters, the *batch_size* specifies on how many samples we estimate the gradient before the parameters update step. In this way, the parameters are optimized every batch size and not with the entire train set.
- *variable.py*: this class is the wrapper class for the torch FloatTensor. It stores:
  - *data*: a FloatTensor representing the value of the variable.
  - *grad*: a FloatTensor representing the accumulated gradient of the loss with respect of this variable. It's shape is equal to the *data* shape.
  - *requires_grad*: boolean used to indicate whether we have to use the *grad* to update the *data* during the parameter update step.
  - *previous_op*: is the link to the operation which generated this Variable.
- *functions/*: this folders contain different modules that implements the basic functions. Table 1 describes its content in details.

We also added support for the broadcasting functionality, which is the possibility of repeating a FloatTensor on one

add reference

dimension in order to allow for a certain matrix operation. This functionality is for example used in the Linear module, when the biases are added to the result obtained from the multiplication between the input and the weight matrix. In the forward pass, this operation is trivially handled by the broadcast support of pytorch Tensors. However, in the backward pass, the gradient received by the broadcasted Variable is not of the same size of its *data* field. We have to simply sum the gradient tensor on the replicated dimension before adding in with the Variable's grad. This operation derives from the mathematical property:

$$\frac{d}{dx}f(x,y)|_{x=y} = \frac{d}{dx}f(x,y) + \frac{d}{dy}f(x,y)$$

When a variable is broadcasted, the dimensions that are expanded contains replicated values, which refers to the constraint $x = y$ of the above formula. Consequently, the derivative of a replicated entry can be seen as the sum of the derivatives along the axes that were replicated.

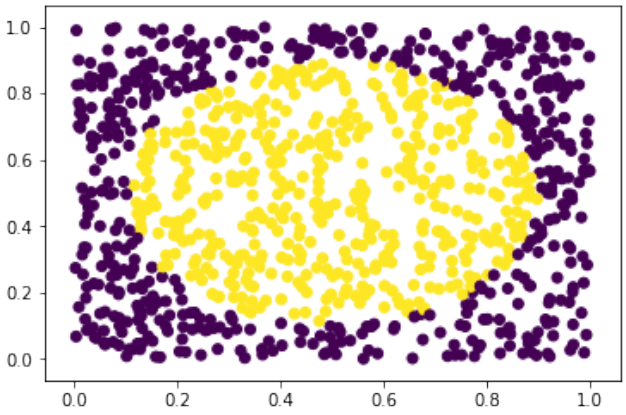
Table I summarizes the modules that are implemented in our framework, along with a short description and a specification of the forward pass.

IV. Test

In this section we provide the description of the dataset and of the network built with our framework. Then, the performance of our model is compared it with well-known baselines.

A. Dataset

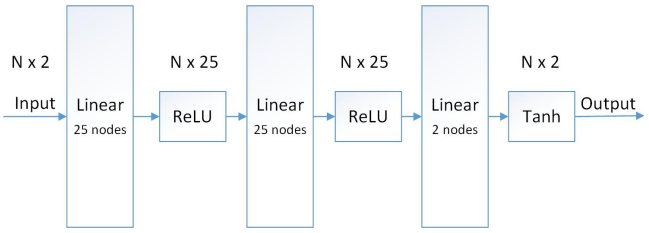
The dataset consists of a training set and a test set of 1000 points each. These points are sampled uniformly in [0, 1], and they are labeled with -1 if they end up out of the disk of radius $\frac{1}{\sqrt{2\pi}}$ and center (0.5, 0.5), with label 1 otherwise. The following exhibit - which represents the train set used during the testing phase - contains a set of points that satisfy these properties:



For the sake of reproducibility, we used a manual seed to ensure that all the runs of the test would be the same.

B. Network

The network consists of two input units, two output units and three hidden layers of 25 units.



A Tanh activation function is used as the last component of our network. The reason behind this choice is that having an unbounded output in a classification problem where MSE is used, would highly penalize both very positives and very negatives outcomes. In fact, the true labels can only have two values and if the network output is unbounded any value that is either too high or too small is interpreted, by the MSE function, as something very different from the labels. By using the Tanh this problem is mitigated as the output of the network is bounded to [-1, 1]. The predicted target values are obtained by is approximating the output to the closest label.

C. Model Comparison

Our model had been tested on the aforementioned dataset. The results are quite satisfying, since the test accuracy is around 98%. A comparison of our model with some common baselines and the same model built by using the pytorch library is shown below:

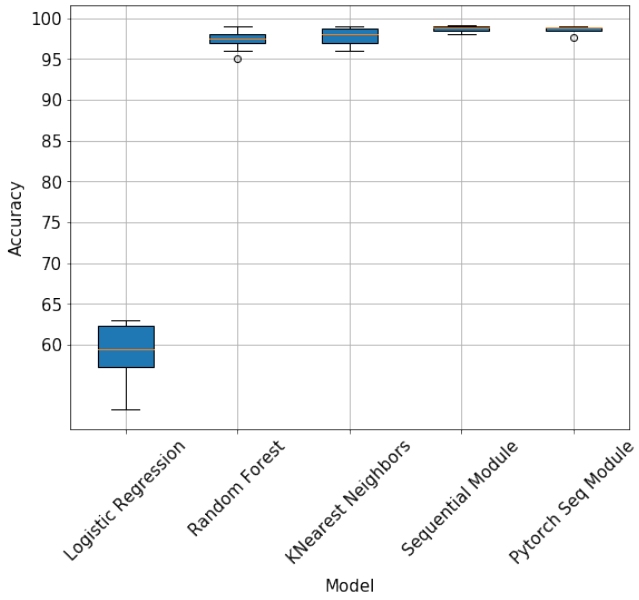


Table I: Modules

Module	Classes	Description	Forward
Activations	ReLU	Rectified linear unit, inherits Module1Operands.	$f(x) = max(0, x)$
	Tanh	Hyperbolic tangent function, inherits Module1Operands.	$f(x) = tanh(x)$
Operands	Add	All these classes extend Module2Operands or Module1Operands depending on the number of inputs. They implement the forward and backward passes.	$f(x, y) = x + y$
	Sub		$f(x, y) = x - y$
	Mul		$f(x, y) = x \odot y$
	MatMul		$f(x, y) = x * y$
	Pow		$f(x, exp) = x^{exp}$
	Mean		$\frac{1}{N} \sum_{i=1}^{N} x_i$
Layers	Linear	The Linear module is a wrapper of two operation: matrix multiplication of the input with the weights and sum of the biases.	$f(x) = x * w + b$
Losses	MSE	The mean square error extend the Module2Operands	$f(x, y) = \frac{1}{N} \sum_{i=1}^{N} (x_i - y_i)^2$

Intuitively, as expected, non-linear classifiers, e.g. K-NN, random forest, perform reasonably well, while linear ones, e.g. logistic regression, give the worst result because the points in the dataset are not linearly separable. With the latter models, higher performance can be achieved only through feature extraction. It can also be seen that our model performs slightly better than any of the tested baselines.

Finally, it is we can notice that the the sequential module built by using our framework gives very similar accuracies as the one built by using the pytorch primitives.

V. Future developments and Conclusions

We designed and implemented a functioning deep learning framework which allows to easily develop additional features such as loss functions, optimizers, layers by providing their respective backward pass. Some improvements could be brought, e.g. the backward propagation stops only when a Variable with no previous operation receives the gradient and stops propagating it, this could potentially correspond to a waste of computation as these Variables usually do not require any gradients.