Implementing from scratch a mini deep-learning framework

Sergii Shynkaruk, Roman Bachmann
EE-559 Deep Learning, Spring 2018, EPFL, Switzerland
{sergii.shynkaruk, roman.bachmann}@epfl.ch

***Abstract*—The objective of this project is to design, implement and evaluate a mini "deep learning framework" using only PyTorch's tensor operations and standard python math library. Thereby we developed a set of core modules (components) which enable users to construct a neural network combining fully connected hidden layers and activation functions (Tanh, ReLU, etc). Training procedure comprises both forward and backward passes, the latter implies model parameters optimization via stochastic gradient descent, RMSProp or Adam for mean-squared error.**

I. Introduction

As part of the Spring 2018 EPFL EE-559 Deep Learning course, we set out to construct a mini deep learning framework allowing users to flexibly build neural networks in a modular fashion.

Using the new and popular PyTorch framework [1] throughout the course, we were getting to know and like its pythonic way of setting up neural networks. Given the proposed structure of the mini-framework and our working knowledge of PyTorch, the overall API of our classes and their functions was mainly inspired by it.

Throughout this report, we will closer explain the overall framework architecture, the functionality of each Module and the math behind. We will talk about the way we implemented the back-propagation and methods like mini-batch gradient descent that speed up computation. Finally, we show experimental results on the framework and benchmark several components thereof.

II. Framework Architecture

As it was suggested in the description of this project [2], we defined a class Module as a base class for all core components. We can see in the class diagram 1, that the class Module serves as an interface class for its descendants. Its `forward()`, `backward()` and `param()` methods are redefined in every subclass w.r.t its unique behavior and math.

A. Containers

In our framework we implemented a sequential container which aggregates in a provided order a sequence of layers and activations. It takes a list of Module objects, and stores it. These modules are provided by a user from the outside. Roughly speaking, 'Sequential' serves as a neural network (NN) architecture keeper. During network training phase

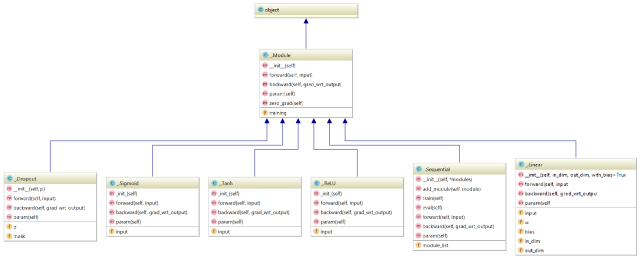


Figure 1. UML class diagram shows inheritance hierarchy from a common base class Module. Every sub-class provides its own implementation of an interface: `forward()`, `backward()` and `param()`.

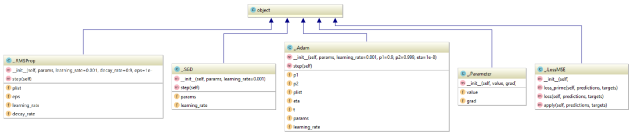


Figure 2. UML class diagram of parameter-less types.

it invokes appropriate forward/backward methods of every constituting module in the same order they were added by a user. On demand it returns a list of parameters of all modules it stores.

B. Activations

Our framework provides several non-linear activation functions - ReLU, Tanh and Sigmoid. Their workings and derivatives are described in Table I. The activation functions are stateless, meaning that they do not return any updatable Parameters in their `param()` function.

	Formula	Derivative
Tanh	$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$tanh'(x) = 1 - \tanh^2$
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	$\sigma'(x) = \frac{\sigma(x)-1}{\sigma(x)}$
ReLU	$f(x) = \max(0, x)$	$f'(x) = \begin{cases} 1; & \text{if } x > 0 \\ 0; & \text{otherwise} \end{cases}$

Table I
Activation functions

C. Initializers

1) Xavier: As weight initializers we chose to implement Xavier initializers. The Xavier Normal initializer draws samples from a truncated normal distribution centered on 0 with standard deviation:

$$\sigma(.) = \sqrt{\frac{2}{n_{in} + n_{out}}} \tag{1}$$

where n_{in} is the number of input units in the weight tensor and n_{out} is the number of output units in the weight tensor.

And Xavier uniform initializer, which draws samples from a uniform distribution within [-*limit*, *limit*], where *limit* equals to $\sqrt{3}\sigma(.)$.

2) Uniform: Another standard uniform initializer was implemented sampling from a uniform distribution within [-*limit*, *limit*], where *limit* equals to $\sqrt{\frac{3}{n_{in}}}$.

D. Parameter

For ease of access throughout the collection of parameters for the updating of the weights, we chose to compose weight and their respective gradients into a Parameter object. If a module contains Parameters that need to be updated during gradient descent, their `param()` function needs to return a list of their Parameters.

E. Layers

1) Linear: To setup a NN our framework contains an implementation of the Linear Layer module, which applies a linear transformation to the incoming data: $y = Wx + b$, where W is the weight tensor and b is the bias. When instantiated it should be initialized with input and output dimensions. The bias term is optional and the user can choose to use it or not. The linear layer holds the weights and biases, as well as the gradient of the loss with respect to the weights or biases as Parameter objects. The standard initialization for the weights uses the Xavier Normal 1, while the biases are all initialized to zero.

In the forward pass, the linear layer computes the output $x^{(l)}$ by applying the transform specified by the weights and biases on the layer input $x^{(l-1)}$ as seen in Table II. For the backward pass, the unit takes the gradient of the loss with respect to its output and computes the gradient of the loss with respect to the module input, the weights and the bias separately. Those calculations can be seen in Table II as well.

2) Dropout: To reduce overfitting, adding Dropout layers to a neural network architecture is a tried and proven approach. We added the corresponding module to our framework accordingly.

During training, for each output element of the previous layer, the Dropout module samples from a Bernoulli distribution with probability p to set the element to zero. Since the total strength of the input to the next layer is weaker when using Dropout, we scale the elements that have not been zeroed by $\frac{1}{1-p}$. In the backward pass, we apply the same transformation to the gradient. During evaluation, the Dropout layer will be disabled. To achieve this, the user has to set the Module container to the training or evaluation mode manually.

	SGD	Mini-batch
$x^{(l)}$	$w^{(l)}x^{(l-1)} + b$	$x^{(l-1)}(w^{(l)})^T + b$
$[\![\frac{\partial \ell}{\partial w^{(l)}}]\!]$	$[\frac{\partial \ell}{\partial x^{(l)}}](x^{(l-1)})^T$	$[\![\frac{\partial \ell}{\partial x^{(l)}}]\!]^T x^{(l-1)}$
$[\frac{\partial \ell}{\partial b^{(l)}}]$	$[\frac{\partial \ell}{\partial s^{(l)}}]$	$\sum_n [\![\frac{\partial \ell}{\partial s^{(l)}}]\!]_n$
$[\![\frac{\partial \ell}{\partial x^{(l)}}]\!]$	$(w^{(l+1)})^T[\frac{\partial \ell}{\partial x^{(l+1)}}]$	$[\![\frac{\partial \ell}{\partial x^{(l+1)}}]\!]w^{(l+1)}$

Table II
FORWARD AND BACKWARD PASS FORMULAS FOR LINEAR LAYER

F. Losses

For our framework we implemented the Mean Squared Error (MSE) loss function. This function will be minimized during the training of the network. The MSE loss and derivative thereof are defined in the following ways, where $\hat{y}$ is the network prediction and y is the true output vector:

$$MSE(\hat{y}, y) = \sum_{i=1}^{N_{out}} (\hat{y}_i - y_i)^2 \tag{2}$$

$$MSE'(\hat{y}, y) = -2(\hat{y} - x) \tag{3}$$

G. Optimizers

We implemented the standard Stochastic Gradient Descent algorithm, as well as the RMSProp and Adam optimizers to speed up training and improve convergence.

1) Stochastic Gradient Descent (SGD): SGD is a standard approach at updating the gradient weights. The update rule is the following, where ℓ_B is the loss of the mini-batch with respect to the weights:

$$w_{t+1} = w_t - \eta \nabla \ell_B(w_t) \tag{4}$$

2) Root Mean Squared Propagation (RMSProp): RMSProp is an adaptive learning rate optimizer utilizing a second moment estimate proposed by Hinton [3] in 2012. We based our implementation on the pseudo code by Goodfellow et al. 2016 [4] in chapter 8.

The optimizer is initialized with a decay rate ρ in $[0, 1)$ and keeps a squared gradient term r (initialized to zero) for each weight throughout the optimization process. Given the latest calculated gradient g, the squared gradient term is updated in the following way, where $\odot$ is element-wise multiplication:

$$r_t = \rho r_{t-1} + (1 - \rho) g_t \odot g_t \tag{5}$$

The new r_t is then used with the learning rate η to calculate the weight update. A small constant δ is added to prevent division by zero.

$$\Delta w_t = -\frac{\eta}{\sqrt{\delta + r_t}} \odot g_t \tag{6}$$

The update is then added to the weight:

$$w_{t+1} = w_t + \Delta w_t \tag{7}$$

The recommended standard values are $\rho = 0.9$, $\eta = 0.001$ and $\delta = 10^{-6}$.

3) Adaptive Moment Estimation (Adam): Adam is an adaptive learning rate optimizer utilizing first and second moment estimators proposed by Kingma and Ba [5] in 2014. We based our implementation on the pseudo code by Goodfellow et al. 2016 [4] in chapter 8.

The Adam optimizer is initialized with decay rates ρ_1 and ρ_2 in $[0, 1)$. Over all training epochs, Adam keeps and updates a first moment estimate s and a second moment estimate r, both initialized with zero. Given the latest calculated gradient g, we calculate the biased first and second order estimates:

$$s_t = \rho_1 s_{t-1} + (1 - \rho_1) g_t, \tag{8}$$

$$r_t = \rho_2 r_{t-1} + (1 - \rho_2) g_t \odot g_t \tag{9}$$

We then correct for the bias:

$$\hat{s} = \frac{s}{1 - \rho_1^t}, \quad \hat{r} = \frac{r}{1 - \rho_2^t} \tag{10}$$

The new $\hat{s}$ and $\hat{r}$ are then used with the learning rate η to calculate the weight update. A small constant δ is added to prevent division by zero.

$$\Delta w = -\eta \frac{\hat{s}}{\sqrt{\hat{r}} + \delta} \tag{11}$$

The update is then added to the weight:

$$w_{t+1} = w_t + \Delta w_t \tag{12}$$

The recommended standard values are $\rho_1 = 0.9$, $\rho_2 = 0.999$, $\eta = 0.001$ and $\delta = 10^{-8}$.

III. Back-propagation

The goal of the back-propagation algorithm is to calculate the gradient of the loss with respect to every network parameter to then update each parameter in a way that decreases the loss function. This algorithm works in two phases, the forward and backward passes.

A. Forward pass

During the forward pass, the input Tensor is propagated through the network in a forward fashion. At each step the input to the module is saved for the backward pass and the output of the module passed as input to the next modules `forward()` function. The flow of the forward pass can either be handled by a container (like the Sequential Module) or has to be user implemented.

B. Backward pass

The output of the forward pass is fed into a loss function returning the loss and the gradient of the loss with respect to the neural network output. This gradient is then fed through the network in the inverse direction of the forward pass. Using the chain rule at each step as in equation 13, a module receives as input to the `backward()` function the gradient of the loss with respect to its output, computes the gradient of the loss with respect to all its parameters (if it has any) and sends the gradient with respect to the module input to the next modules `backward()` function. The flow of the backward pass can also be handled by either a container or needs to be user implemented.

$$\frac{\partial \ell}{\partial x_i^{(l)}} = \frac{\partial \ell}{\partial x_i^{(l+1)}} \frac{\partial x_i^{(l+1)}}{\partial x_i^{(l)}} \tag{13}$$

IV. Mini-batch Gradient Descent

Instead of training the network using all data points in one pass (standard gradient descent) or taking only one sample at a time (stochastic gradient descent), we can take a middle ground and process the whole data set in mini-batches. This yields advantages in the calculation speed thanks to tensor operations and overall convergence is more stable than in SGD.

For mini-batches, the loss term is now the average of the loss of each data point in the batch. Another difference lies in the calculation of the forward and backward pass in the Linear layer. In Table II we display the differences assuming an activation function is not part of the Linear layer.

V. Experimental Results

A. Testing the framework on generated data

For the experimental part we generated a training and a test set of 1000 points sampled uniformly in $[0, 1]^2$, each with a label 0 outside the disk of radius $\frac{1}{\sqrt{2\pi}}$ and 1 inside. Following the project test requirements we created a model with input layer of 2 units, 3 hidden layers of 25 units each and output layer of 2 units. This NN was trained using standard SGD with epoch number = 300, learning rate = 0.001 and batch size = 100. We obtained values of training and test errors of 0.2% and 0.3% accordingly proving the high efficiency of deep learning application in the current context.

B. Optimizer benchmark

We ran the implemented standard SGD, RMSProp and Adam optimizers on the dataset from section V-A and recorded the training loss over 300 epochs. All optimizers used learning rate 0.001 and the standard parameters recommended in Goodfellow et. al. 2016 [4], namely a decay rate of 0.9 for RMSProp and decay rates of 0.9 and 0.999 for the first and second moment estimates in the Adam optimizer.

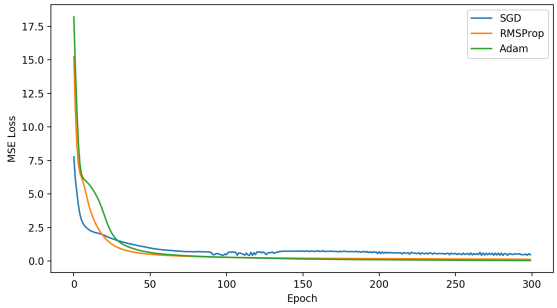


Figure 3. SGD, RMSProp and Adam optimizers run on the above generated dataset

In figure 3 we can clearly see that the SGD optimizer is faster than the other two in the beginning, but after ca. 10 epochs has trouble converging. In fact, the Adam and RMSProp optimizers do not oscillate much and are both converging to an optimum, while the SGD curve is having trouble achieving the same stability and performance.

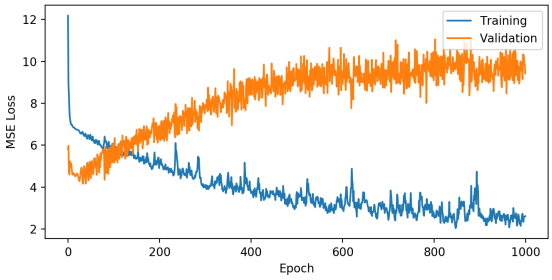
C. Training speed comparison with PyTorch

We decided to build the exact same model from section V-A in PyTorch and measure the time it takes to run the 300 training epochs. Training the network with our own implementation was done in 0.9678 seconds, while the PyTorch equivalent took 1.4866 seconds. This large time difference might stem from the fact that PyTorch is doing a lot more in the background, like building the computation graph in every iteration or needing to handle the dataset as Variables instead of using a very lightweight structure like we do.

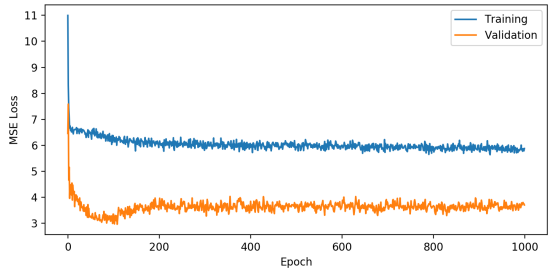
D. Dropout benchmark

To test the Dropout module, we modified the data generation from section V-A to include some Gaussian noise in the circle. The goal of this was to create a model with a high degree of freedom that would easily overfit on the noise and to evaluate a test set without the noise on it. For this purpose, we created a neural network with four hidden layers, each with 500 units and using the ReLU activation. The last layer uses again the Tanh activation. For the model with Dropout, we added a Dropout layer using probability 0.5 after each ReLU activation. Both models were trained using the same properties as the models before, but for 1000 epochs to allow for sufficient overfitting time.

As we see in figure 4(a), the validation loss of the model without Dropout layers started to increase steadily after few epochs. This with the fact that the training loss was decreasing is a clear indication that overfitting is happening. After 1000 epochs, this model achieved a train error of 4.20% and a test error of 13.50%. As the test set does not contain noise, we would expect a model that learned to generalize perform better, but it clearly overfit.



(a) Without Dropout



(b) With Dropout

Figure 4. Training and validation losses with and without Dropout layers

In figure 4(b), we see the validation loss increasing shortly at epoch 100, but not for long. Afterwards both the validation and training losses stay very stable. The Dropout layers are indeed helping the model to stop overfit. The final train error was 10.10% and the test error 4.00%. We see that the model with the Dropout layers learned to generalize and did not overfit on the noise.

VI. Conclusion

For this mini deep learning framework, we built the Layer Modules Linear and Dropout, the activation functions ReLU, Tanh and Sigmoid, a Sequential Module container, the Xavier and uniform initializers, the MSE loss and the SGD, RMSProp and Adam optimizers. All these components can be modularly put together to build any fully connected neural network. Experimental results show that the mini-framework is working well and thanks to its low overhead is running faster than PyTorch.

REFERENCES

[1] A. Paszke, S. Gross, S. Chintala, and G. Chanan, "PyTorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration." [Online]. Available: https://pytorch.org/

[2] F. Fleuret, "EPFL EE-559 Deep Learning Course," 2018. [Online]. Available: https://fleuret.org/dlc

[3] G. Hinton, "Lecture 6a Overview of mini-batch gradient descent," 2012. [Online]. Available: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[5] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980