Mini Deep Learning Framework

Ljupche Milosheski **Mladen Korunoski** **Dragoljub Djuric**
École Polytechnique Fédérale de Lausanne
{ljupche.milosheski, mladen.korunoski, dragoljub.djuric}@epfl.ch

***Abstract*—In this project, we present the implementation of a mini deep learning framework. The implementation of this framework was inspired by PyTorch and Autograd and contains all necessary components for executing a regression or classification task using deep neural networks. Additionally, we evaluate our implementation on an artificial classification task. We obtain an accuracy of 97.48% on the test set.**

I. Introduction

PyTorch[1] has proven itself as the industry standard when it comes to constructing, training, and utilizing deep neural networks for different machine learning tasks. At its core, it uses the Autograd library [1] for gradient-based optimization. This library supports reverse-mode differentiation (a.k.a. backpropagation), which means it can efficiently take gradients of scalar-valued functions with respect to array-valued arguments, as well as forward-mode differentiation, and the two can be composed arbitrarily. Our implementation was inspired by Autograd and follows its native API to execute the forward and backward passes when training deep neural networks.

II. Theory

Deep neural networks trained using gradient-based optimization methods use the backpropagation algorithm [2]. This algorithm completely relies on the chain rule from Calculus for differentiating the cost function w.r.t. to the models' parameters. There are four fundamental equations that define the backpropagation algorithm and are given bellow (inspired from [3]):

$$\delta^{(L)} = \nabla_a C \odot \sigma'(z^{(L)}) \tag{1}$$

$$\delta^{(l)} = ((W^{(l+1)})^T \cdot \delta^{(l+1)}) \odot \sigma'(z^{(l)}) \tag{2}$$

$$\frac{\partial C}{\partial W^{(l)}} = a^{(l-1)} \cdot (\delta^{(l)})^T \tag{3}$$

$$\frac{\partial C}{\partial b^{(l)}} = \delta^{(l)} \tag{4}$$

, where L is the index of the last layer, C is the loss function, $z^{(l)}$ is the output vector of the l-th layer before applying the activation function, σ is an activation function, $a^{(l)}$ is the output vector of the l-th activation function, $\nabla_a C$ is the gradient of the cost function w.r.t the last activation vector, $W^{(l)}$ is the weight matrix of the l-th layer, $b^{(l)}$ is the bias vector of the l-th layer, and $\delta^{(l)} = \frac{\partial C}{\partial z^{(l)}}$.

Lets define two vectors, $i_f^{(l)}$ - the input vector to the l-th layer during the forward pass and $i_b^{(l)}$ - the input vector to the l-th layer during the backward pass. We show how we compute these vectors for each module in the next section.

When executing the forward pass, we store the input vector $i_f^{(l)}$ into the context. When executing the backward pass, we retrieve $i_f^{(l)}$ from the context and use it with the input vector $i_b^{(l)}$ to compute the gradients.

III. Implementation

All components of our framework extend the base *Module* class that defines the main API calls required to train a deep neural network. These include the *forward* and *backward* methods that execute the forward and backward passes of each separate module. As noted before, some of the modules have a context where they store data necessary for computing the gradients. Furthermore, some of the modules accumulate gradients required for optimizing the parameters. Later in this section, we define how each of this methods operate for each module.

A. Modules

The main module supported by our framework is the *Sequential* module that defines a fully-connected deep neural network. We keep the layers and activation functions separate for more intuitive computation of gradients. This module computes the forward passes of each sub-module in sequential order. After completion, it computes the backward passes of each sub-module in reverse order.

B. Layers

The main layer supported by our framework is the *Linear* layer. The linear layer is defined by its weight matrix, W, and the bias vector, b. The forward pass of this layer is defined as:

[1]https://pytorch.org/

$$i_f^{(l+1)} = i_f^{(l)} \cdot W^{(l)} + b^{(l)}$$

The backward pass of this layer is defined as:

$$i_b^{(l-1)} = (W^{(l)})^T \cdot \delta^{(l+1)}$$

It accumulates the gradients using:

$$\frac{\partial C}{\partial W^{(l)}} = i_f^{(l)} \cdot (i_b^{(l)})^T$$
$$\frac{\partial C}{\partial b^{(l)}} = i_f^{(l)}$$

We initialize the weight matrix, W, and bias vector, b, by sampling uniformly from $[-\frac{1}{\sqrt{d_{in}}}, \frac{1}{\sqrt{d_{in}}}]$ [4]. This helps with the vanishing gradient problem.

C. Activation Functions

Our implementation supports three activation functions: *Sigmoid*, *Tanh*, and *ReLU*. The forward pass of the functions is defined as:

$$i_f^{(l+1)} = \sigma(i_f^{(l)})$$

The backward pass of the functions is defined as:

$$i_b^{(l-1)} = i_b^{(l)} \odot \sigma'(i_f^{(l)})$$

Bellow, we define each of the activation functions.

1) Sigmoid: The Sigmoid activation function is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

and its derivative as:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

2) Hyperbolic Tangent: The Tanh activation function is defined as:

$$\sigma(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

and its derivative as:

$$\sigma'(x) = (1 - \sigma(x))^2$$

3) Rectified Linear Unit: The ReLU activation function [5] is defined as:

$$\sigma(x) = \max(0, x)$$

and its derivative as:

$$\sigma'(x) = \begin{cases} 1, x > 0 \\ 0, x < 0 \end{cases}$$

D. Loss Functions

Our implementation supports two loss functions: *Mean Squared Error* and *Binary Cross Entropy*.

1) Mean Squared Error: The MSE loss function for one sample is defined as:

$$C_a = \frac{1}{2}(y - a^{(L)})^2$$

and its derivative as:

$$\nabla C_a = y - a^{(L)}$$

2) Binary Cross Entropy: The BCE loss function for one sample is defined as:

$$C_a = y \cdot \log(a^{(L)}) + (1 - y) \cdot (1 - \log(a^{(L)}))$$

and its derivative as:

$$\nabla C_a = -(\frac{y}{a^{(L)}} - \frac{1 - y}{1 - a^{(L)}})$$

E. Optimizers

Our implementation uses the *Stochastic Gradient Descent with Momentum* [6]. The optimizer stores reference to all model parameters. After accumulating the gradients in the backward pass, it updates their values by:

$$v^{(t+1)} = \mu \cdot v^{(t)} + \alpha \cdot \frac{\partial C}{\partial (W^{(l)})^{(t)}}$$
$$(W^{(l)})^{(t+1)} = (W^{(l)})^{(t)} + v^{(t+1)}$$

, where t denotes the timestamp, μ is the momentum factor and α is the learning rate.

IV. Dataset

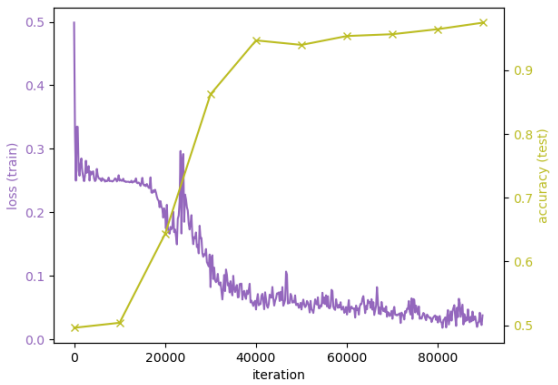

Fig. 1. Loss (train) vs. Accuracy (test)

For evaluating our implementation, we generate two datasets with 1000 samples each. The points are uniformly sampled from $[0, 1]^2$ and labelled with 0 if outside a sphere with center $(0.5, 0.5)$ and radius $\frac{1}{\sqrt{2\pi}}$ and 1 if inside.

V. Results

We define our deep neural network as follows:

$$\begin{gathered}\text{Linear}(d_{in} = 2, d_{out} = 25) \\ \downarrow \\ \text{Relu}() \\ \downarrow \\ \text{Linear}(d_{in} = 25, d_{out} = 25) \\ \downarrow \\ \text{Relu}() \\ \downarrow \\ \text{Linear}(d_{in} = 25, d_{out} = 25) \\ \downarrow \\ \text{Relu}() \\ \downarrow \\ \text{Linear}(d_{in} = 25, d_{out} = 2)\end{gathered}$$

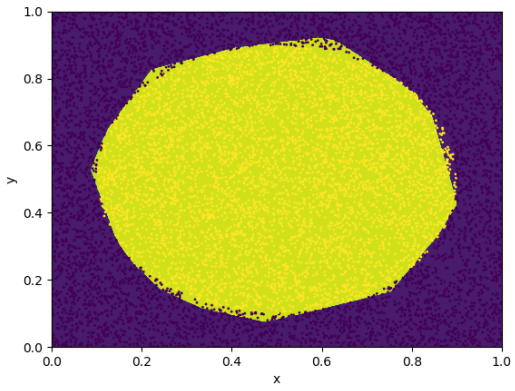

Fig. 2. Decision boundary (test)

We train with batch size of 200 for 10 epochs. The optimizer's hyperparameters are set to $\alpha = 0.1$ and $\mu = 0.9$. After each epoch, we plot the accuracy on the test set. After each iteration, we plot the loss on the train set. Results are given on Fig. 1.

We can see that the neural network has high capacity at modeling such a simple function on a bounded domain. With the accuracy increasing drastically even after 2^{nd} epoch we can observe its power. One problem that we experienced was that the network tends to get stuck in a local minimum. To overcome this problem, we simply re-execute the training procedure.

Besides the loss vs. accuracy plot, we show the decision boundary obtained on the test set of this deep classifier on Fig. 2.

VI. Conclusion

In this project we presented a mini deep learning framework that implements several modules, activation functions, loss functions, and optimizers. We evaluated our implementation on artificially generated data and obtained an accuracy of 97.48%.

Future steps include implementing *Convolutional*, *Pooling*, *Recurrent*, and *Dropout* layers, and also *Adam* and *RMSprop* optimizers.

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] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[3] M. A. Nielsen, *Neural networks and deep learning*. Determination press San Francisco, CA, 2015, vol. 25.

[4] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

[5] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Icml*, 2010.

[6] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*. PMLR, 2013, pp. 1139–1147.