Project 2
Mini deep-learning framework

Nicolas Baldwin[1], Ana Lucia Carrizo[2], Christopher Julien Stocker[3]
EE-559 Deep Learning, Swiss Federal Institute of Technology Lausanne

Abstract— **The goal of Project 2 is to design a mini "deep learning framework" using only PyTorch's tensor operations and the standard math library; in particular, without using autograd or the neural-network module.**

I. Objective

THE objective of this project is to create our own framework using only PyTorch's tensor operations and the standard math library. The framework must provide the tools necessary to build networks combining fully connected layers, Tanh, and ReLU. It implements forward and backward passes, and parameter optimization with stochastic gradient descent (SGD) for mean square error (MSE).

II. Neural Network Architecture

In this section, the framework of our modules and the mathematical formulas used to build it are explained. Firstly, N is defined as the number of data with n and m the dimensions of the input and output layers. N_l is the number of units in layer l and σ is the activation function where w are the weights, and $s^{(l)}$ are summations before the activation functions.

A. Module

In order to build a neural network from scratch we built a superclass that possess the main methods most for modules use. We also built the architecture for a fully connected linear layer, and a module that will come in handy when combining the different elements that make up our network. Our file *module.py* contains the following classes:

1. **Module**

The class Module is the superclass that all other model architectures inherit. It possesses the following methods, that are later redefined:

- *__init__(self)*
- *forward(self, *arguments)*: defines the forward pass of the module.
- *backward(self, *gradwrtoutput)*: defines the backward pass of the module.
- *param(self)*: returns the list of parameters from all layers.
- *step(self,lr)*: updates the weights and biases given the learning rate.
- *zero_grad(self)*: re-initializes the gradient to zero.
- *reset_weight(self,*parameters)*: it re-initializes the weights using the Xavier method seen in class and it sets the gradient weight and bias to zero.

2. **Linear**

This class represents the fully connected linear layer.

- *__init__(self, in_size, out_size, gain = 1)*: raises an error if the input or output sizes are not positive integers. It initializes the weights and biases using Xavier. Initializes the gradient's weights and biases with zeroes.

$$V\left(w^{(l)}\right) = \frac{1}{\frac{N_{I-1}+N_I}{2}} = \frac{2}{N_{I-1}+N_l}$$

- *forward(self, x)*: linear fully connected forward pass. It takes the output of the previous layer.

$$x^{(0)} = x, \quad \forall I = 1, \ldots, L, \begin{cases} s^{(l)} = w^{(l)} x^{(I-1)} + b^{(l)} \\ x^{(l)} = \sigma\left(s^{(I)}\right) \end{cases}$$

- *backward(self, *gradwrtoutput)*: linear fully connected backward pass. Takes as argument the gradient of the next layer.

$$\left[\frac{\partial \ell}{\partial x^{(I)}}\right] = \left(w^{(l+1)}\right)^\top \left[\frac{\partial \ell}{\partial s^{(l+1)}}\right] \quad \textit{and}$$

$$\left[\frac{\partial \ell}{\partial w^{(I)}}\right] = \left[\frac{\partial \ell}{\partial s^{(l)}}\right] \left(x^{(I-1)}\right)^\top$$

3. **Sequential**

This class is used to build a multi-layer neural network. It combines all the elements that compose our network in a sequential manner. This class applies the basic pipeline: *init, forward propagation, back propagation, reset weights, zero grad ...*

- *forward(self, x)*: recursively calls *forward* on each layer of the network.
- *backward(self, *gradwrtoutput)*: recursively calls *backward* on each layer of the network, starting from the end.
- *param(self)*: returns a list with the parameters for every layer.
- *zero_grad(self)*: calls *zero_grad* recursively; thus, setting all gradients to zero.
- *reset_weights(self,*parameters)*: calls *re-*

[1]Msc. Data Science, Swiss Federal Institute of Technology Lausanne, e-mail: nicolas.baldwin@epfl.ch.
[2]Msc. Data Science, Swiss Federal Institute of Technology Lausanne, e-mail: ana.carrizodelgado@epfl.ch.
[3]Msc. Robotics, Swiss Federal Institute of Technology Lausanne, e-mail: christopher.stockersalas@epfl.ch.

set_weights recursively; thus, resetting all weights and biases.

B. Activation Functions

Our next module contains all the activation functions we will use in our model. As specified in the the project's description, we built a ReLU activation function. Furthermore, we took the initiative to explore other activation functions in the hope of improving performance. The definition of these classes can be found in the *activation_functions.py* file.

For every activation function, we redefine the forward and backward methods. For the *Tanh* and *Sigmoid* activation functions we also define some auxiliary methods to calculate their derivative.

1. **ReLU**
   - *forward(self, x)*: returns
   $$max(0, x)$$
   - *backward(self , * gradwrtoutput)*: returns 1 if $x > 0$, and 0 otherwise.

2. **Tanh**
   - *derivative(self, x)*: we used the following relation:
   $$\frac{d}{dx}tanh(x) = 1 - tanh^2(x)$$
   - *forward(self, x)*: returns $tanh(x)$
   - *backward(self, *gradwrtoutput)*: calls the *derivative* function defined above.

3. **Sigmoid**
   - *sigmoid(x)*: the sigmoid function is defined as follows:
   $$S(x) = \frac{1}{1 + e^{-x}}$$
   - *derivative(x)*: the derivative of the sigmoid function is:
   $$\frac{d}{dx}S(x) = S(x) \cdot (1 - S(x))$$
   - *forward(self, x)*: calls the *sigmoid* function defined above.
   - *backward(self , * gradwrtoutput)*: calls the *derivative* function defined above.

C. Losses

In this module, we construct a class that contains the necessary tools to compute our neural network's losses. Particularly, we compute the Mean Square Error, and redefine the *backward* function. This class can be found in the *losses.py* file.

1. **MSE**

   - *__init__(self)*: we set the initial difference between our prediction and actual value to zero.
   - *compute_loss(self, pred, actual)*: computed as
   $$Loss = \frac{1}{n}\sum_i (pred_i - actual_i)^2$$
   - *backward(self)*: returns
   $$2 \cdot (pred - actual) \cdot \frac{1}{n}$$

D. Cross Validation

In this module, found in the file *cross_validation.py*, we obtain the hyperparameters that optimize our network. To do this, it resorts to certain methods such as:

- *parseModel*
- *createData*
- *train_model*
- *computeAccuracy*

which will be explained in more depth in the next section.

When running *crossValidation* we obtain the following best parameters for each activation function:

Table I: Best Parameters for each Activation Function

	ReLU	**Tanh**	**Sigmoid**
Learning Rate	0.01	0.1	0.1
Batch Size	16	50	100
Momentum	0.8	0.9	0.9
Xavier Gain	1.0	1.0	1.0
Epochs	100	100	100

III. Implementation

In this section we discuss how we trained and tested our neural network.

A. Training

This module contains all methods and classes needed to train our data, given the specifications found in the project's description.

First, we build a stochastic gradient descent optimizer: *SGD()*; it initializes the layers, learning rate, momentum and velocity.

Besides the *SGD* class, this module contains the following functions:

- createData: method that creates our training and testing data in a uniform manner. It produces 1000 samples. It returns the following: *train_input*, *one_hot_train_target*, *test_input*, and *one_hot_test_target*.
- train_model: this is our main function where our model is trained. It trains the model on a specified number of epochs and can log the loss and accuracy of the test and train data after every epochs.
- computeAccuracy: returns the ratio of predictions that corresponded to the its true value.

- parseModel: creates a more user friendly way to define models. Given a python dictionary containing a field 'model' which contains a list of strings describing the model, the parser will return the actual model. For example, if given a dictionary such that dictionary['model'] = ["Linear(2,25)",ReLU,"Linear(25,2)"], the parser will return this model.

IV. Framework Results

Our results for each model are summarized in the following table:

Table II: Performance of Models

	ReLU	**Tanh**	**Sigmoid**
Av. Train Accuracy	96.29 %	78.47 %	94.91 %
Std. Train Accuracy	0.0238	4.146	0.0359
Av. Test Accuracy	0.9578	0.7902	0.9461
Std. Test Accuracy	0.000181	0.03806	0.000128
Av. Train Loss	0.034730	0.121696	0.056277
Std. Train Loss	8.579e-05	0.009421	0.000226
Av. Train Loss	0.0367749	0.121489	0.058873
Std. Test Loss	7.821 e-05	0.009395	0.000189

Fig. 1: Train Accuracy Plot

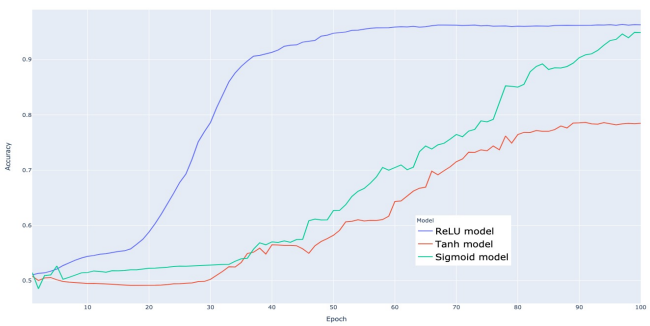

Fig. 2: Test Accuracy Plot

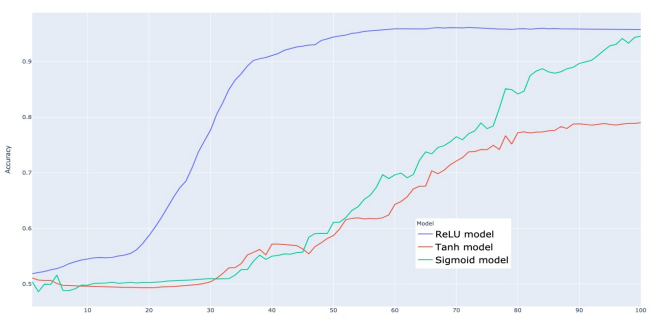

Fig. 3: Test Loss Plot

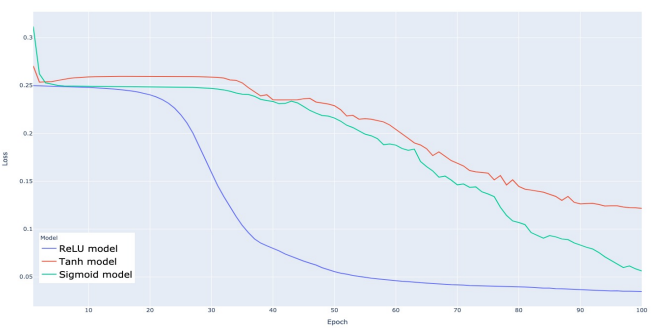

Fig. 4: Train Loss Plot