EE-559: Deep Learning

Mini Project 2: *"Mini deep-learning framework"*

Jelena Banjac, Pavlo Karalupov, Aslı Yörüsün

May 22, 2020

1 Introduction

In this project, our aim is to implement a mini "deep learning framework" using only PyTorch's tensor operations and the standard math library, hence in particular without using autograd or the neural-network modules available in PyTorch. Our framework mainly consists of the forward and backward propagation implementation of several modules such as: **Linear** as fully connected layer; **ReLU**, **Tanh**, and **Mish** as activation functions; **Sequential** as combination of layers; and **MSE** as a loss function with **SGD** to optimise the parameters. The accuracy and performance of the framework are checked on a two-class classification task.

2 Framework Implementation

This section gives information about different modules in our framework and a mathematical reasoning behind them. More detailed information about the framework implementation can be found in our github repository.

2.1 Module Class

The class *Module* stands for a base class for all neural network modules (i.e. Linear, ReLU, Tanh, Mish, Sequential, and MSE) defined in our framework. It consists of two abstract methods: *forward* and *backward* that raise an error if not implemented. Also, it includes *param* method which returns parameters (e.g. weight and bias for Linear), *update_params* method that updates parameters during optimization step and *zero_grad* method that is used to empty stored gradients that were calculated during backward pass.

2.2 Linear Module

Our *Linear* module takes four input parameters. Two of them are compulsory: *in_features* and *out_features* which are the dimensions of the input and output, respectively. And the others are not compulsory: *bias*, which is a Boolean input parameter to represent whether the use of a bias in the current layer, whose default value is "True" and *weight_init*, which is the last input parameter to control the use of initialisation of the weights in the current layer, whose default value is "None".

2.2.1 Initialization

We initialised the weights **W** and biases **b** at each layer uniformly distributed as $\mathcal{U}$(-a, a) where, $a = \frac{1}{\sqrt{n}}$, where n is the number of hidden units in the input layer.

2.2.2 Forward Propagation

The *Linear* module implements a fully-connected layer that applies a linear transformation to the incoming data as following: $y = xW^\top + b$.

2.2.3 Backward Propagation

The gradients are computed with respect to the input and parameters such as weight and bias.

2.3 Activation Functions

Here we applied the most commonly used activation functions such as *ReLU* and *Tanh*, and we decided to implement one new state-of-the-art solution which is *Mish*.

2.3.1 ReLU

Rectified Linear Unit (ReLU) which can be defined by $f(x) = max(0, x)$ is a non-linear activation function that is mostly used in hidden layers of deep neural networks. The graph of ReLU is shown in Fig.1. Forward propagation is applied based on the function ReLu (blue) and backward propagation is applied based on the function derivative of ReLU (orange) shown in the figure.

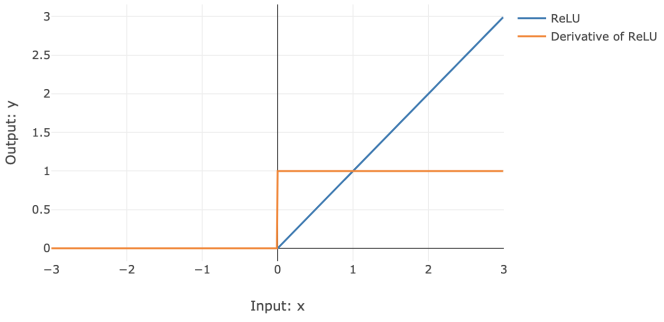

Figure 1: ReLU activation function

2.3.2 Tanh

Tanh or the Hyperbolic Tangent Activation Function, which is defined as $f(x) = 2 \cdot \sigma(2x) - 1$ where, $\sigma(x) = \frac{e^x}{1+e^x}$, is a sigmoidal ("s"shaped), but instead outputs values in the range (-1, 1). The graph of Tanh is shown in Fig.2. Forward propagation is applied based on the function tanh (blue) and backward propagation is applied based on the function derivative of tanh (orange) shown in the figure.

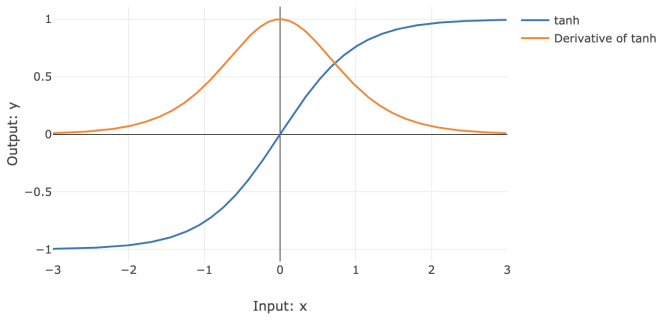

Figure 2: Tanh activation function

2.3.3 Mish

Mish is a novel smooth and non-monotonic neural activation function which can be defined as $f(x) = x \cdot tanh(\zeta(x))$ where, $\zeta(x) = ln(1+e^x)$ is the softplus activation function (for further information see the paper). The graph of Mish is shown in Fig.5. Forward propagation is applied based on the function Mish (blue) and backward propagation is applied based on the function derivative of Mish (orange) shown in the figure.

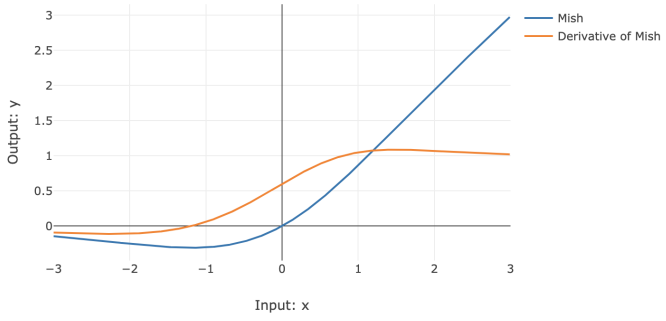

Figure 3: Mish activation function

2.4 Sequential Module

The *Sequential* module works as a sequential container that allows the modules such as Linear, ReLu, etc. to be added to it so as to build a neural network structure. The modules are added in the order they are passed in the constructor in order to perform forward and backward operations for this sequence of modules. Alternatively, an ordered dict of modules can also be passed in.

2.5 MSE Module

We used the Mean Squared Error (squared L2 norm), defined as $\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$, as a loss criterion function for our neural network in order to measure the averaged squared error between each element in the target value y and the predicted value $\hat{y}$.

2.6 SGD

Stochastic gradient descent (SGD) is an iterative method for optimizing an objective function with suitable smoothness properties. The use of SGD in the neural network setting is motivated by the high cost of running back propagation over the full training set as GPU is far smaller than dataset size. Hence we split dataset into batches and used SGD as an optimizer to overcome this cost and still lead to fast and stable convergence.

3 Framework Testing

3.1 Data Generation

For a two-class classification task, we generated a training and a test data set of 1,000 points sampled uniformly in $[0, 1]^2$, each with a label 0 if it is outside the disk centered at (0.5, 0.5) of radius $\frac{1}{\sqrt{2\pi}}$ and 1 if it is inside.

We normalised the training and test data set by subtracting the mean and dividing by the standard deviation of the data in order to be able to guarantee a stable convergence of the weights and biases. The graph for normalised train data set can be seen in Fig.4.

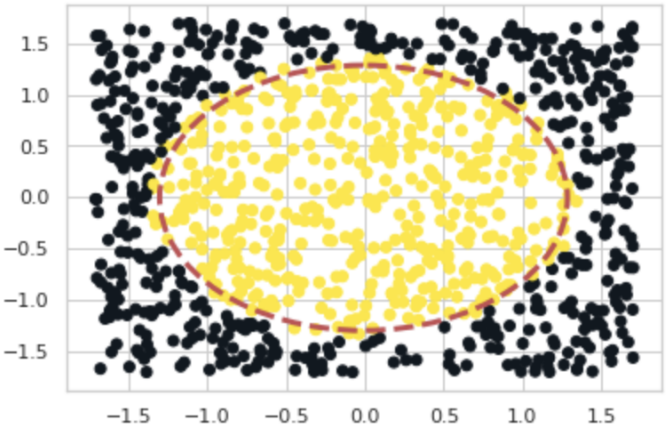

Figure 4: Normalised data

3.2 Weight Initialisation

We initialised the weights by using Xavier Uniform which fills the input tensor with values using a uniform distribution. The resulting tensor will have values sampled from $\mathcal{U}$(-a, a) where, $a = \sqrt{\frac{6}{\text{fan\_in}+\text{fan\_out}}}$ (*fan_in* is the number of hidden units of the input layer, and *fan_out* is the number of hidden units of the output layer).

3.3 Neural Network Structure

The neural network structure used to train the framework is composed of an input layer of two input units, an output layer of two output units, and in between three hidden layers of 25 units. We used ReLU and Mish as activation functions for all except the last layer in which we used Tanh as the activation function. The reason why we used two activation functions in the hidden layers is that we wanted to compare the performance of ReLU and Mish.

3.4 Model Training and Testing

One of the first experiments we ran was comparison of sequential model containing ReLU activation function and the sequential model containing Mish activation functions. The performance we observed can be seen in Fig.5.

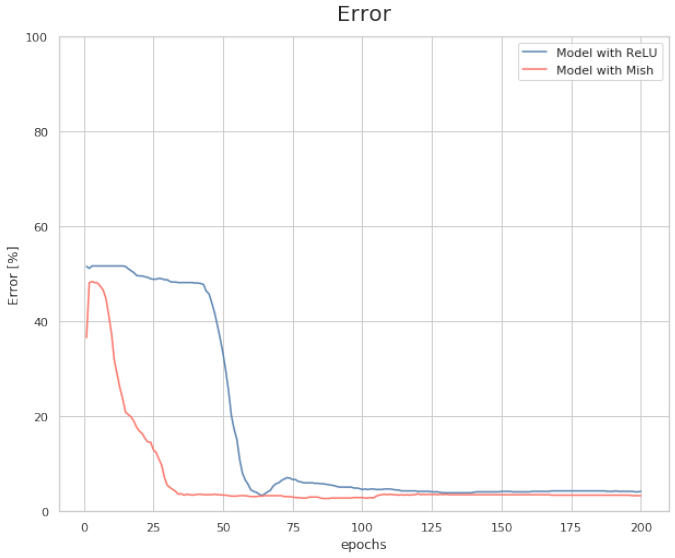

Figure 5: Error of model containing ReLU versus model containing Mish activation functions

From this figure we can observe that sequential model with Mish activation function performs better than the model implemented with ReLU activation functions. Therefore, we decided to include this activation function into our mini deep learning framework.

For the same network architecture and training settings the performance of these two frameworks can be observed in the Fig.6 displaying the error w.r.t. training and testing data depending on the framework. We observe that the error is lower by 0.7% in PyTorch (lower by 0.7% in training dataset and lower by 0.8% in testing dataset). In Fig.7 we observe the loss w.r.t. training and testing data depending on the framework. We notice the loss is lower in the PyTorch by 9 (lower by 9.4 in training dataset and lower by 9.3 in the testing dataset). However, the implemented deep learning framework finished 15 rounds of training with learning rate equals to 0.001 and 300 epochs in 65.5 seconds whereas the Pytorch finished in 79.6 seconds.

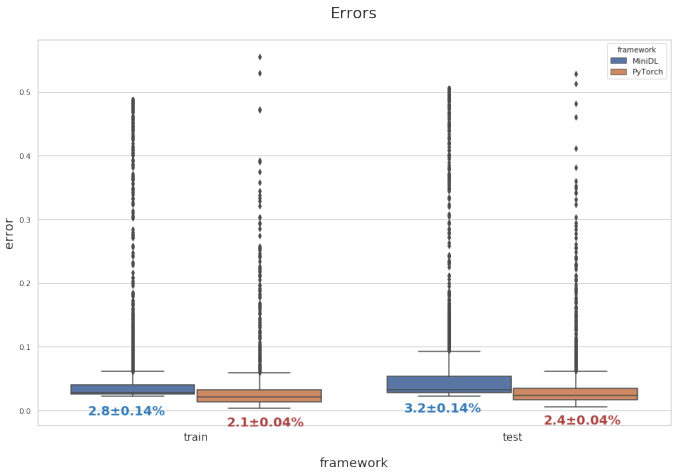

Figure 6: Error comparison of MiniDL (implemented) and PyTorch frameworks. Plot shows the output of 15 rounds of training and evaluation with 300 epochs, 0.05 learning rate, and batch size of 100.

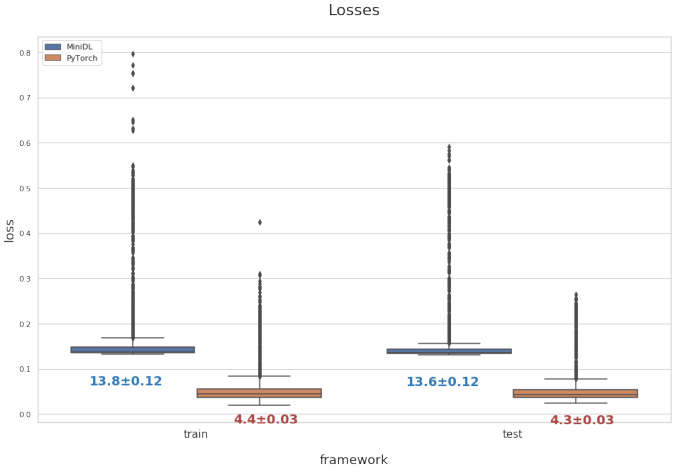

Figure 7: Losses comparison of MiniDL (implemented) and PyTorch frameworks. Plot shows the output of 15 rounds of training and evaluation with 300 epochs, 0.05 learning rate, and batch size of 100.

4 Conclusion

In this project we successfully built custom deep learning framework. In order to test and compare the implementation of new framework and PyTorch, we used dataset of 1000 uniformly in $[0, 1]^2$ positioned points with the label 0 if in the circle and 1 otherwise. We built the test network with two input units and two output units with three hidden layers. We trained and evaluated these same models implemented in two different frameworks using MSE loss and SGD optimizer. We compared the performance of these frameworks and conclude that their performance is close regarding the accuracy (difference around 0.7%) of the models as well as losses (difference around 9 datapoints). In addition, the implemented framework is slightly faster than the PyTorch implementation.