PyChrot: A Pytorch like Deep Learning Framework
Project of EE559 Deep Learning

Yuxuan Wang, Saibo Geng, Shuo Wen
EPFL, Switzerland

***Abstract*—In this article, we introduce a toy Deep Learning Framework developed for the project2 of EE559 Deep Learning: PyChrot. Following the instructions of the project, PyChrot has been built upon Pytorch's basic Data Structure Tensor without the Auto Differentiation function activated. We implemented all required modules in PyChrot including dense layer, MSE loss , but also a couple of additional modules to handle more complex situations in Deep Learning such as gradient explosions or gradient vanishing. PyChrot is developed with Pytorch as reference, thus we take Pytorch as the benchmark objective in testing.**

I. Interface

A Module is an abstraction over the implementation of some function or algorithm, possibly associated with some persistent data. In PyChrot, this abstraction is realized by the base class `Module`. A Module can be distinguished to Parametric Module or Non-Parametric Module depends on whether the module contains parameters(weight and bias) to optimize. In PyChrot, Parametric Modules include `Linear`,`Conv1d` and `BatchNorm` while Modules like `Flatten`, `Tanh` are Non-parametric. A Module may contain further Modules ("submodules"), each with their own implementation, persistent data and further submodules. This type of Module is known as containers and in PyChrot we have implemented `Sequential` as a container Module.

```
class Module(object):
    def forward(self, *input):
        raise NotImplementedError
    def backward(self, *gradwrtoutput):
        raise NotImplementedError
    def param(self):
        return []
```

Each specific Module will inherit from it and implement `forward` to calculate the forward pass, `backward` to do backpropagation and `params` to get parameters of a module.

- `forward` is defined to run the Forward Propagation (or forward pass) of the Module. This involves calculation and storage of intermediate variables and final output from the input layer to the output layer. This method returns the output value and a dictionary `cache` in which we store the intermediate variables useful in the backward pass
- `backward` is defined to run the Back Propagation (or backward pass) of the Module. The goal of backpropagation is to calculate the gradient of parameters in the Neural Network Module. The method takes the gradient with regard to output variables and intermediate variables saved in forward pass as input, calculate the gradient of parameters(weight and bias) and also the gradient with regard to input variables of the Module. This method return the gradient with regard to input variables of this Module to the precedent Module in the computational graph.
  In a container Module such as `Sequential`, this method traverses the network in reverse order, from the output to the input layer.
- `params` is defined to return a list of pairs, each composed of a parameter tensor, and a gradient tensor of same size. This list is empty for parameterless modules (e.g. ReLU).

II. Implementation

`Linear`
: Fully connected linear layer with bias option available. Weight and bias parameters are initialized with Xavier initialization.

`Conv1d`
: 1D convolution over an input signal composed of several input planes(channels_in). This is a commonly used Neural Net module in computer vision. Depending on channels_out, multiple convolution kernel will be applied. Sliding window size can be specified via stride parameter. Padding is fixed to False and no padding option available.

`ReLU`
: Implementation of ReLU (Rectified Linear Unit) activation fucntion. compared to $Sigmoid$ or $Tanh$, it could avoid gradient vanishing as the gradient is always 1 for positive input. After comparing several different implementation of the forward pass, we choose `y = x * (x > 0)` because it's the most fast.

`Sigmoid`
: Implementation of Sigmoid activation function. As a commonly used activation function, $Sigmoid$ map the input to between 0 and 1, which could

help to avoid gradient explosion. We choose to implement $\theta(x) = \frac{1}{1+e^{-}x}$ instead of $\theta(x) = \frac{e^x}{1+e^x}$ to avoid 32-bit float overflow when x is large.

`Tanh`

Implementation of Tanh (hyperbolic tangent) activation fucntion. Among several equivalent mathematical expressions, We choose to implement $tanh(x) = 1 - \frac{2}{e^{2x}+1}$

`BatchNorm`

Applies Batch Normalization over an input tensor.Batch Normalization helps to accelerate the training and avoid gradient explosion. When affine option is set True , the learned affine transform applied to these normalized activations allows the BN transform to represent the identity transformation and preserves the network capacity.

`Sequential`

A sequential container. Modules will be added to it in the order they are passed in the constructor. Calling forward or backward on this Module will trigger the sequential execution of the underlying modules.

`MSE`

Module to measure the mean square error (MSE) between each element in the input.

`L2loss`

Module to measure the L2 error between each element in the input

`L1loss`

Module to measure the L1 error between each element in the input

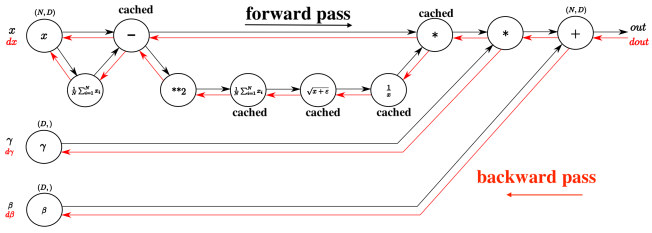

Figure 1. Batch Normalization Computational Graph: From left to right, following the black arrows flows the forward pass. From right to left, following the red arrows flows the backward pass which distributes the gradient from above layer to gamma and beta and all the way back to the input.

III. Performance Benchmark

To validate the performance of PyChrot, we follow instructions to benchmark it with the classification task: classify whether a 2D point taken from interval

$$[0, 1] \times [0, 1)$$

is inside the unit circle with $(0.5, 0.5)$ as origin or not. From mathematics, this classification task can be solved perfectly

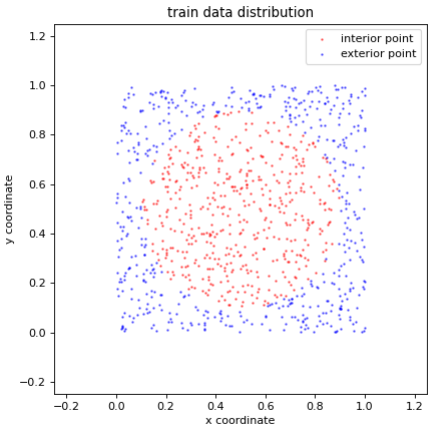

Figure 2. Data scatter plot

by the formula 1 where $y = 1$ denotes inside the circle and $y = 0$ denotes outside the circle.

$$\begin{aligned} y &= \text{sign}\{1 - (x_1 - 0.5)^2 - (x_2 - 0.5)^2\} \\ &= \text{sign}\{-(x_1^2 + x_2^2) + x1 + x2 + 0.5\} \end{aligned} \tag{1}$$

As shown from the formula, the problem is not linearly separable as there is quadratic terms and also a sign function. To deal with non-linearity, activation functions must be introduced into the model. In addition, as the formula only contains quadratic terms, polynomial data augmentation of degree 2 should be able to tackle this non-linearity.

A. Data

Data is generated randomly with fixed seed. We use 1000 data points for training and another 1000 for testing. The distribution of data is visualized as in the figure 2.

B. Model Structure

We start with a sequential architecture composed of 5 repetitive blocks as shown in the 3. Each block contains a Linear Layer with hidden dimension $dim_h = 20$, an activation function and a batch normalization layer. As the input of this classification task is only two floats, it doesn't make sense to include a Convolution Layer into the model. But for some other sequential modeling task, Conv1D should be useful.

C. Experimental Setup

1) batch size = 10
2) number of epochs = 200
3) step size = 1 for Tanh and Sigmoid; 0.001 for ReLU
4) use torch.manual_seed(0) to fix the randomness
5) optimizer is fixed to SGD

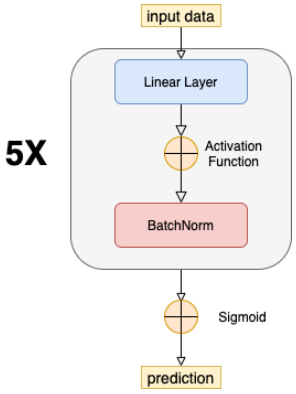

Figure 3. Model Architecture.

D. Experiment Results and Ablation study

The losses of models using various activation model is given in the following figure. As showed in Figure 5, the training loss decrease exponentially at the first 10 epochs, with the quick increment of accurancy.

After 200 epochs, the training accurancy can approch 0.97 with a test accurancy equals 0.94. The results of the deep learning network to solve the non-linear classification problem are satiesfied.

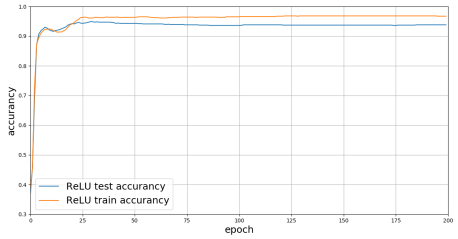

Figure 4. train precision vs test precision under ReLU.

Table I
PERFORMANCE OF DIFFERENT ACTIVATION FUNCTION

model	Precision	Recall	F1	F1 (train)	Loss
model(sigmoid)	0.935	0.934	0.935	0.959	0.265
model(Tanh)	0.922	0.922	0.922	0.933	0.268
model(ReLU)	0.927	0.927	0.927	0.945	0.284

We conduct experiments with various activation functions to explore their performance. Table I gives some evaluation indexes of the model, we can see $Sigmoid$ gives a slight high precision.

Comparison of their train loss is given in Figure 5. Using $Sigmoid$ as activation function causes a higher decreasing speed at the first epoch, while can easily falls in a local optima. The decrement of $ReLU$ is faster than $Tanh$, and the precision of $ReLU$ is also a little bit higher. As this task is aiming to benchmark PyChrot's performance, we didn't do much parameter tuning because we think it's unnecessary. From experiments, longer training(500 epochs) yields **higher performance** close to **98%** on both test and train F1 score.

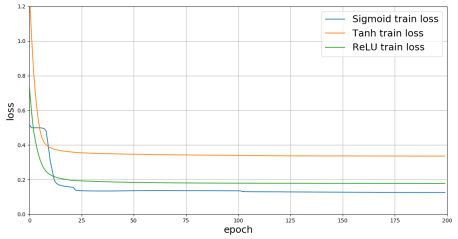

Figure 5. train loss of various activation functions.

Table II
IMPACT OF BATCH NORMALIZATION

model	Prec	Recall	F1	F1 (train)	Loss	time(s)
sigmoid+BN	0.935	0.934	0.935	0.959	0.265	33.13
sigmoid	0.756	0.744	0.742	0.760	0.338	26.05
Tanh+BN	0.922	0.922	0.922	0.933	0.268	32.01
Tanh	**0.979**	**0.978**	**0.979**	**0.986**	**0.037**	26.39
ReLU+BN	0.927	0.927	0.927	0.945	0.284	32.61
ReLU	0.736	0.723	0.720	0.762	0.452	26.29

In order to validate our Mini framework's correctness and performance, we build a same model using Pytorch to serve as reference. From Table III, it can be seen that our network PyChort achieves almost the same performance with PyTorch both on the measurements of precision and runtime. The reason PyTorch has slightly lower performance than PyChrot could be due to the difference on internal implementation of MSE loss. PyTorch takes the mean of squared loss with regards to all elements of input tensor while we only considers averaging across the batch dimension. This makes gradient in PyChrot twice as large as in PyTorch.

Table III
COMPARISON PYCHROT VS PYTORCH

model	framework	F1 test	F1 train	Loss	runtime(s)
model(Tanh)	PyChrot	0.922	0.933	0.269	32.01
-	PyTorch	0.912	0.943	0.293	30.39
model(ReLU)	PyChrot	0.927	0.945	0.284	32.11
-	PyTorch	0.886	0.926	0.297	29.84
model(Sigmoid)	PyChrot	0.935	0.959	0.265	33.13
-	PyTorch	0.896	0.937	0.292	28.09

IV. SUMMARY

In this project, we built a miniframework for Deep Learning based on PyTorch's Tensor and Tensorial Operations. The Framework yields similar performance compared to PyTorch in the benchmark, achieving 98.7% F1 score on test set. In the future, higher dimensional CNN layer and other classical modules including RNN layer should be included. As Deep Learning is evolving fast, new modules such as Attention Layer, Contrastive Loss are also important composites for a modern framework.