Project 2: Mini deep learning framework

Zahra Jaleh, Seyedvahid Mousavinezhad

Abstract

In this project, we aim 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 modules. We studied effects of 18 different conditions on a neural network architectures and it gives us an inside look at what goes on behind the scenes.

1 Introduction

The basic idea behind developing the PyTorch framework is to develop a neural network, train, and build the model. Pytorch provides a torch.nn base class which can be used to wrap parameters, functions, and layers in the torch.nn modules. In order to develop a deep learning model, the subclass of torch.nn module is used, which uses methods like forward(input) that returns the output. In simple neural networks, input is fed through multiple hidden layers and weights and bias are added. The output is then produced.
In this project we implement 18 whole neural networks without using torch.nn and of course without using autograd. We implement two criterions(MSE and BCE) to calculate training loss, two optimizers(SGD and Adam) and four activations(ReLU, Tanh, LeakyReLU).

2 Data

The training set and test set of this project are 1,000 points sampled uniformly in $[0, 1]^2$, each with a label 0 if outside the disk centered at (0.5, 0.5) of radius $1/\sqrt{2\pi}$, and 1 inside.

3 Network structure

We implement a base Module class, so other modules inherit from it.

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

```
Network:
   Sequential(
      Linear(2,25)
      Activation()
      Linear(25,25)
      Activation()
      Linear(25,25)
      Activation()
      Linear(25,1)
      Activation())
```

Our network structure is as follow:

- **Sequential:** It is a Container used to combine different layers to create a feed-forward network.
- **Linear:** This class applies the linear transformation of the input data, based on $f(x) = weight \times x + bias$
- **Activation:** We implement four activation:
  - **Tanh:** The hyperbolic tangent activation function is calculated as $\frac{(e^x - e^{-x})}{(e^x + e^{-x})}$. Its back-propagation is $1 - Tanh(x)^2 \times gradwrtoutput$
  - **ReLU:** It applies ReLU function and takes the max between 0 and element. Its back-propagation is $1 \times gradwrtoutput$ for input > 0 and otherwise is $0 \times gradwrtoutput$.
  - **LeakyReLU:** It has small negative slope and applies element wise on input x,in its forward function, it returns max(0.01×input, input). Its back-propagation returns one if $x > 0$ and otherwise is $0.01 \times gradwrtoutput$.
  - **Sigmoid:** This function is calculated as $\frac{1}{1+\exp(-x)}$. Its back-propagation is $Sigmoid(x) \times (1 - Sigmoid(x)) \times gradwrtoutput$

We implemented two loss functions:

- **MSE:** It is used to measure the mean squared area in squared L2 norm between each element in input and output. The forward function returns $1/batchSize * \Sigma(y_{pred} - y)^2$. Its back-propagation function returns $\Sigma(2 \times (y_{pred} - y))/batchSize$
- **BCE:** It measures the Binary Cross Entropy loss between input and prediction. The forward function returns $-\Sigma(y_{pred} \times log(y) + (1 - y_{pred}) \times log(1-y))$. Its back-propagation function returns $\Sigma(y_{pred} - y)/batchSize$.

For optimization we implement two class as follow:

- **SGD:** In our Stochastic gradient decent class we have implemented an update function which updating the weight and bias by these formulas: first updating the velocity of weight and bias: $momentum \times velocity((weight, bias)) + (1 - momentum) \times gradient(weight, bias)$ then update the weight and bias by: $(weight, bias) - lr \times gradient(weight, bias)$.
- **Adam:** We have implemented Adam as another optimizer, we have some parameters such beta1, beta2, lr and $epsilon = 1e^{-8}$ in its update function we update the weight and bias by these formulas: $firstMomentum(weight, bias) = beta1 \times firstMomentum(weight, bias) + (1 - beta1) \times gradient(weight, bias)$, then $secondMomentum(weight, bias) = beta2 \times secondMomentum(weight, bias) + (1 - beta2) \times gradient(weight, bias)^iteration$. Afterward we do a bias correction by: $corrections = bothMomentum/(1 - (beta1, beta2)^2)$ and finally update the weight and bias by: $(weight, bias) = (weight, bias) - lr \times corrections/(\sqrt{corrections} + epsilon)$

4 Result

We constructed a neural network with with 3 hidden layers and evaluated its performance with different activation functions, different loss functions and different optimizers. With each activations (Tanh, ReLU, LeakyReLU) we implemented six networks with different optimizers and different loss functions, the result of each networks are in the following tables.
We visualized the testing data with true labels in figure1, then we visualized the result of each networks in the next figures. We run all the networks with and without SGD with batch size = 5 and epochs = 300, and all the networks with Adam with batch size = 5 and epochs = 150.

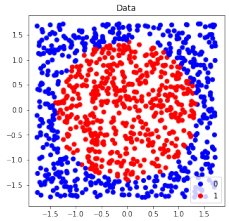

Figure 1: Test dataset with true labels

Table 1: Activation: Tanh

Criterion + Optimizer	Train accuracy	Test accuracy
MSE, SGD	98.6	94.0
MSE, Adam	97.8	93.1
MES, None	98.5	94.8
BCE, SGD	98.1	93.2
BCE, Adam	95.9	92.4
BCE, None	98.5	94.7

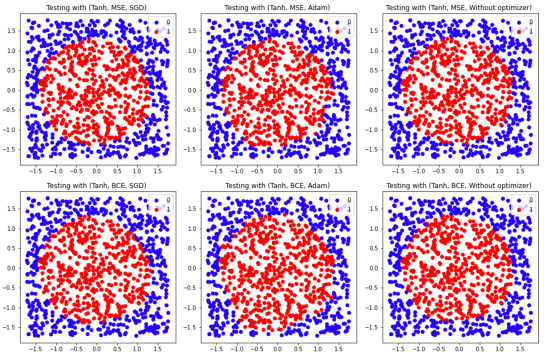

Figure 2: Visual results of testing data with 6 different networks with activation Tanh

Table 2: Activation: ReLU

Criterion + Optimizer	Train accuracy	Test accuracy
MSE, SGD	97.8	94.3
MSE, Adam	98.6	94.2
MES, None	97.4	94.1
BCE, SGD	97.2	93.8
BCE, Adam	98.1	94.0
BCE, None	97.8	93.1

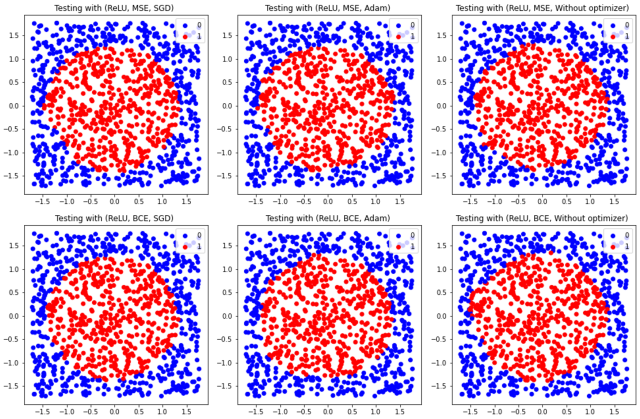

Figure 3: Visual results of testing data with 6 different networks with activation ReLU

Table 3: Activation: LeakyReLU

Criterion + Optimizer	Train accuracy	Test accuracy
MSE, SGD	98.5	94.8
MSE, Adam	97.6	94.7
MES, None	97.9	93.8
BCE, SGD	97.7	93.6
BCE, Adam	98.3	94.3
BCE, None	97.2	92.5

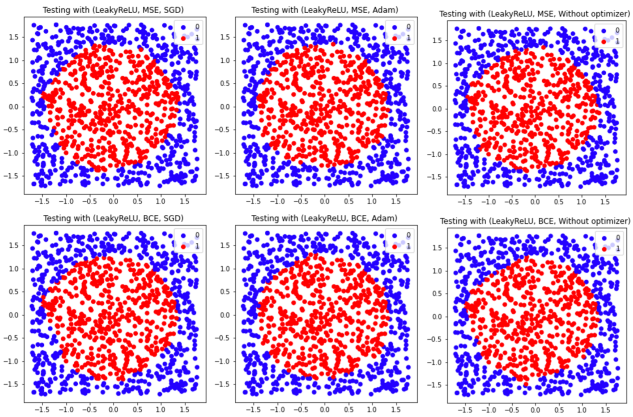

Figure 4: Visual results of testing data with 6 different networks with activation LeakyReLU

By considering the results we can conclude that using an optimizer helps us to obtain better accuracy, but Adam optimizer helps to converge to better results in a lesser number of epochs than SGD. However, our networks without using optimizer performed good enough. Also, based on our observation the accuracy of the models which use MSE as a loss function is better than the models using BCE.

5 Conclusion

In this project, we implemented our own neural networks with different activation functions and different optimizers. It truly helped us to have an inside look at what goes on behind the scenes.