Deep Learning Project 2

Group 58: Xingce Bao, Xi Fan, Danjiao Ma
Date: May 15, 2018
Course: Deep Learning

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Contents

List of Figures

List of Tables

1 Project Description

The idea of this project is to design a deep-learning framework, while only using pytorch's tensor operations and standard math library. Without using NN modules in Pytorch, Chapter 2 introduces how we designed the whole framework, including fully connected layers, activation function, loss function, optimizers and so on. Once each component of the network is available, we designed three ANNs to fit the model for uniformly distributed dataset. Besides, we also tried to discuss the reliability and accuracy of these networks.

As for the train and test datasets, they are designed to be uniformly distributed in $[0, 1]^2$. And the classification is defined by a disk of radius $\frac{1}{\sqrt{2\pi}}$. If the point is located inside the disk, it is labeled as 1, otherwise, it will be labeled as 0.

2 Project Structure

The whole ANN consists of fully connected layers, activation function, loss function and optimizer. Each module of NN should provide: 'Forward propagation' to give the module output; 'Backward propagation' to obtain the gradient of the loss at module's output layer and reversely accumulate the gradient of parameters to reduce the loss; 'param' to give back the parameter and gradient tensor.

2.1 Linear Layer

As a fully connected layer, each unit of the linear layer should be linked to all the units in the previous layer. And linear layer also holds two parameters, weight and bias, which should also be updated in optimization process. In this project, the weight tensor is initialized by a Gaussian distribution $W \sim \mathcal{N}(0, 1)$, and bias tensor is initialized by zeros.

2.1.1 Forward Pass

During 'forward pass', the output is a linear combination of an input X of shape $N \times D$ from the previous layer with weight matrix W of shape $D \times M$ and bias b (Eq.1). The output of linear layer Y will influence others layers and finally contribute to a scalar loss L.

$$\begin{array}{ccccccc} Y & = & X & * & W & + & b \\ (N*M) & & (N*D) & & (D*M) & & (N*M, broadcast) \end{array} \tag{1}$$

where Y is the output matrix of shape $N \times M$, b is the bias matrix, generated by broadcast.

2.1.2 Backward Pass

During backward, the derivation of $\frac{\partial L}{\partial Y}$ can be computed as Eq.2.

$$\frac{\partial L}{\partial Y} = \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}} & \frac{\partial L}{\partial y_{1,2}} & \cdots & \frac{\partial L}{\partial y_{1,m}} \\ \frac{\partial L}{\partial y_{2,1}} & \frac{\partial L}{\partial y_{2,2}} & \cdots & \frac{\partial L}{\partial y_{2,m}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial L}{\partial y_{n,1}} & \frac{\partial L}{\partial y_{n,2}} & \cdots & \frac{\partial L}{\partial y_{n,m}} \end{pmatrix} \tag{2}$$

To train the model by minimizing a loss, the goal of backward pass is to compute $\frac{\partial L}{\partial X}$, $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$. Based on chain rule, they could be obtained by Eq.3 [1].

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial X} = \frac{\partial L}{\partial Y}W \qquad \frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial W} = \frac{\partial L}{\partial Y}X^T \qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial b} = \frac{\partial L}{\partial Y} \tag{3}$$

Given their gradient, both weight and bias can be updated as follows:

```
self.w = self.w-lr*self.w_gradient/list(self.prev_module.output.size())[1]
self.b = self.b-lr*self.b_gradient.mean(1,True)
```

2.2 Activation Function

In this project, we designed several commonly used activation functions, including Sigmoid, Tanh, ReLU and softmax. Their key properties are indicated in Tab.1 [2].

Table 1: List of Activation Functions

	Function	**Gradient**	**Output Range**	**Remark**
Sigmoid	$\Phi(x) = \frac{1}{1+e^{-x}}$	$\Phi'(x) = \Phi(x)(1-\Phi(x))$	[0,1]	low-speed convergence, gradient vanishing
Tanh	$\Phi(x) = \frac{e^x-e^{-x}}{e^x+e^{-x}}$	$\Phi'(x) = 1-\Phi^2(x)$	[-1,1]	zero-centered output, gradient vanishing
ReLU	$\Phi(x) = \max(0,x)$	$\Phi'(x) = \begin{cases} 1 & x>0 \\ 0 & x<0 \end{cases}$	[0,+∞]	efficient, alleviates over-fitting
Softmax	$\Phi_j(z) = \frac{e^{z_j-z_k}}{\sum_i e^{z_i-z_k}}$	$\Phi'(x) = \begin{cases} \Phi_i(1-\Phi_j) & i=j \\ -\Phi_i\Phi_j & i\neq j \end{cases}$	[0,1]	multiple classifications, sum of outputs = 1

During back-pass, the derivative of activation functions should also be calculated for passing it to the previous layer. For 'sigmoid','tanh' and 'ReLU', their gradient matrix only has elements in the diagonal, which transforms the matrix product to Hadamard product in code.

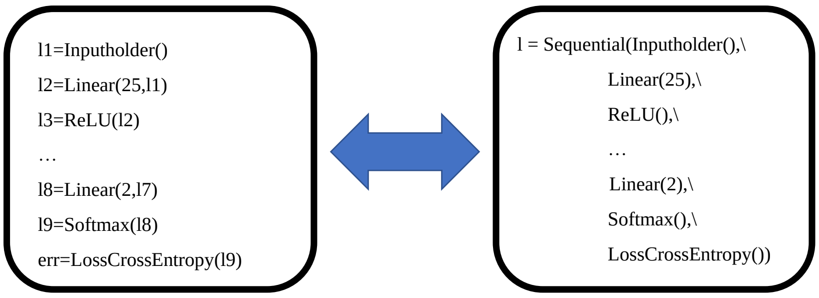
```
self.gradient = self.gradient * self.next_module.gradient
```

However, for code implementation, what should also be noticed is that the numerical range of floating point in Python is limited. Therefore, with exponential element, it is much easier for softmax to overshoot the limit. To avoid this overflow, we normalized the result with its maximum z_k (Tab.1). Besides, unlike the others, gradient matrix of 'softmax' has elements in both diagonal and non-diagonal parts. In this case, it is impossible to calculate the gradient by Hadamard product in Python. Instead, what we implemented is to generate the common element $\Phi_i \Phi_j$ for diagonal and non-diagonal parts first and then sum it with self output to get the overall matrix [2]. Additionally, we also applied the function **bmm** to avoid loop and accelerate the calculation.

```
self.gradient = self.output.t().clone()
s=list(self.gradient.size())[0]
self.gradient_1 = self.gradient.contiguous().view(s,-1,1).clone()
self.gradient_2 = self.gradient.contiguous().view(s,1,-1).clone()
# Compute the gradient matrix
self.gradient = self.gradient_1.bmm(self.gradient_2)
self.gradient = -1*self.gradient.bmm(self.next_module.gradient.t().contiguous().
    view(s,-1,1))
# Multiply the gradient afterwards and add the extra gradient in diagonal
self.gradient = self.gradient.contiguous().view(s,-1).t()+self.next_module.
    gradient*self.output
```

2.3 Sequential model

The whole ANN consists of 2 input units, 1 or 2 output units and three hidden layers with 25 points. To make the code more adaptable, we set a default previous module for each module as 'None'. In this way, whatever there is a previous module as input, the current module can be connected to network correctly. In other words, each layer of the whole sequential model could be defined either separately or together (Fig.1).



```
l1=Inputholder()
l2=Linear(25,l1)
l3=ReLU(l2)
…
l8=Linear(2,l7)
l9=Softmax(l8)
err=LossCrossEntropy(l9)
```

```
l = Sequential(Inputholder(),\
            Linear(25),\
            ReLU(),\
            …
            Linear(2),\
            Softmax(),\
            LossCrossEntropy())
```

Figure 1: Sequential Model

2.4 Loss Function

In this project, we designed MSE and cross entropy loss functions (Tab.2). MSE tries to minimize the difference between target and predication.

Cross entropy loss indicates the difference between predicated output distribution and its real distribution. Cross entropy is usually combined with softmax activation as the last layer of the network, since the output of softmax represents the probability. As for code implementation, the

Table 2: List of Activation Functions

	Function	**Gradient**	**Remark**
MSE	$\mathcal{L}(w) = \frac{1}{N}\sum_{n=1}^{N}(y_n - f(x_n;w))^2$	$\frac{\partial\mathcal{L}}{\partial f} = \frac{2}{N}\sum_{n=1}^{N}(f(x_n;w) - y_n)$	y_n: real output, $f(x_n;w))$: prediction
Cross Entropy	$\mathcal{L}(y,p) = -\sum_i y_i \log(p_i)$	$\frac{\partial\mathcal{L}}{\partial p_i} = -\sum y_i \frac{1}{p_i}$	y_i is real output,p_i is predicted output

gradient of cross-entropy can be written in a compact way[2]. Based on the formula in Tab.2, the code can be designed for backward as follows:

```
self.gradient=-1*(1/(self.prev_module.output+self.eps)*self.target)
```

Within the code above, ‘self.eps’ is a defined constant ‘$1e-20$’. Since cross entropy is usually applied with ‘softmax’, whose output range is [0,1], the introduction of this small constant can avoid the denominator to be 0.

2.5 Optimizer–SGD

To minimize the loss, ‘gradient descent’ algorithm uses local information to iteratively move to a (local) minimum, by following the steepest descent (Eq.5).

$$w_{t+1} = w_t - \eta\nabla\mathcal{L}(w_t) \tag{4}$$

However, practically it is computational-intensive to calculate the gradient exactly and we usually train our set in batch size. In this case, mini-batch stochastic gradient decent(SGD) is designed in our project to update parameters after each batch (Eq.5).

$$w_{t+1} = w_t - \eta\sum_{b=1}^{B}\nabla l_{n(t,b)}(w_t) \tag{5}$$

where η is learning rate, B is the batch size, $n(t,b)$ is the order to visit samples.

As for learning rate, we designed four different ways for its continuous update, including ‘constant’, ‘step’,‘exponential’ and ‘frac’:

Table 3: List of Learning Rate Option

lr_option	**Function**
constant	$lr_{i+1} = lr_i$
step_decay	$lr_{i+1} = lr_i \times \text{decay\_rate}^2 \times (i\ mod\ \text{step\_decay\_length})$
exp_decay	$lr_{i+1} = lr_i \times \exp(-i \times decay\_rate)$
t_frac_decay	$lr_{i+1} = lr_i/(1 + i \times decay\_rate)$

3 Result Analysis

Based on the generated train data, we designed the network and trained the parameters by the optimizer. In this project, we tested with three different neural networks (Tab.4) and discuss their performance, in terms of error rate, variation and time consumption.

Table 4: Structure of Three ANN Models

ANN	**Activation Function**	**Loss Function**	**Learning Rate**	**Decay Rate**	**lr Option**	**Other Common Parameters**
1	ReLU, softmax	cross entropy	0.1	0.01	t_frac_decay	optimizer: SGD, batch size:500
2	ReLU, sigmoid	MSE	0.8	0.0002	exponential	
3	Tanh, softmax	cross entropy	0.03	0.001	t_frac_decay	

Fig.2 - Fig.4 show the performance of these NNs separately. With tunned parameters, we run experiments 10 times for each model and record the average value in Tab.5. Generally, the combination of 'softmax' and 'cross entropy' loss function is preferred to achieve a more stable model with a low error rate (ANN1 and ANN3). And both ReLU and Tanh could be used as activation layers efficiently. As for time consumption, due to exponential element in 'Tanh', ANN3 consumes more time than ANN1 and ANN2.

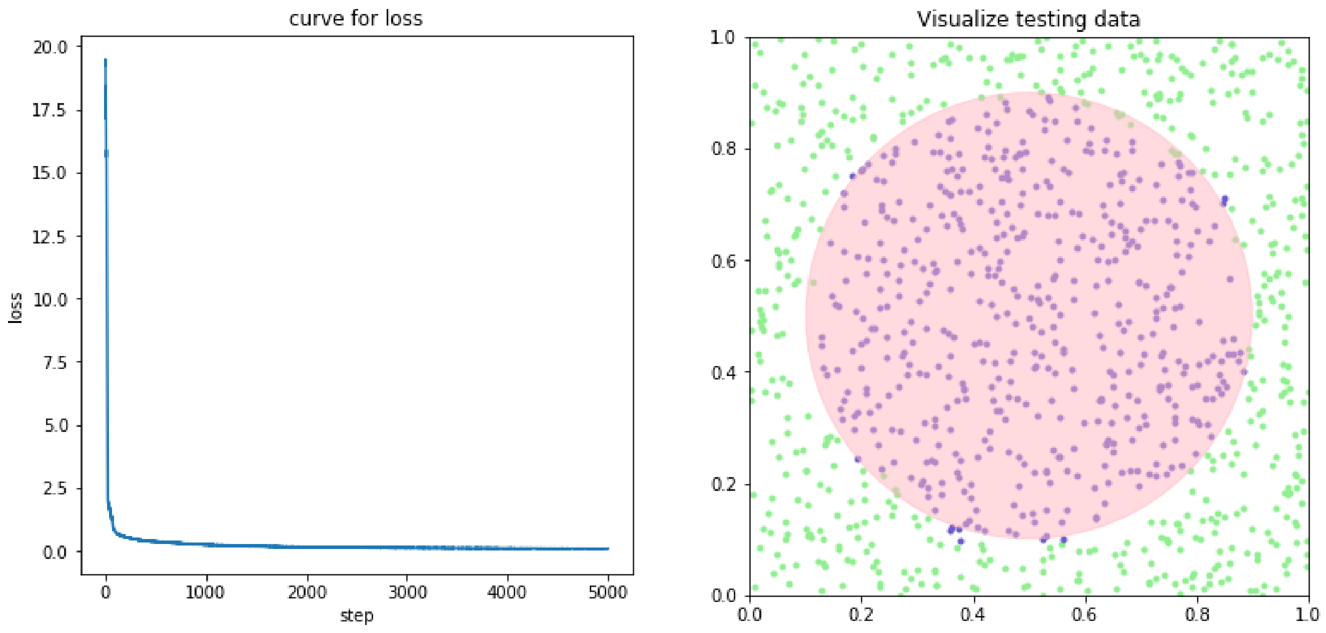


Figure 2: Performance for ANN1

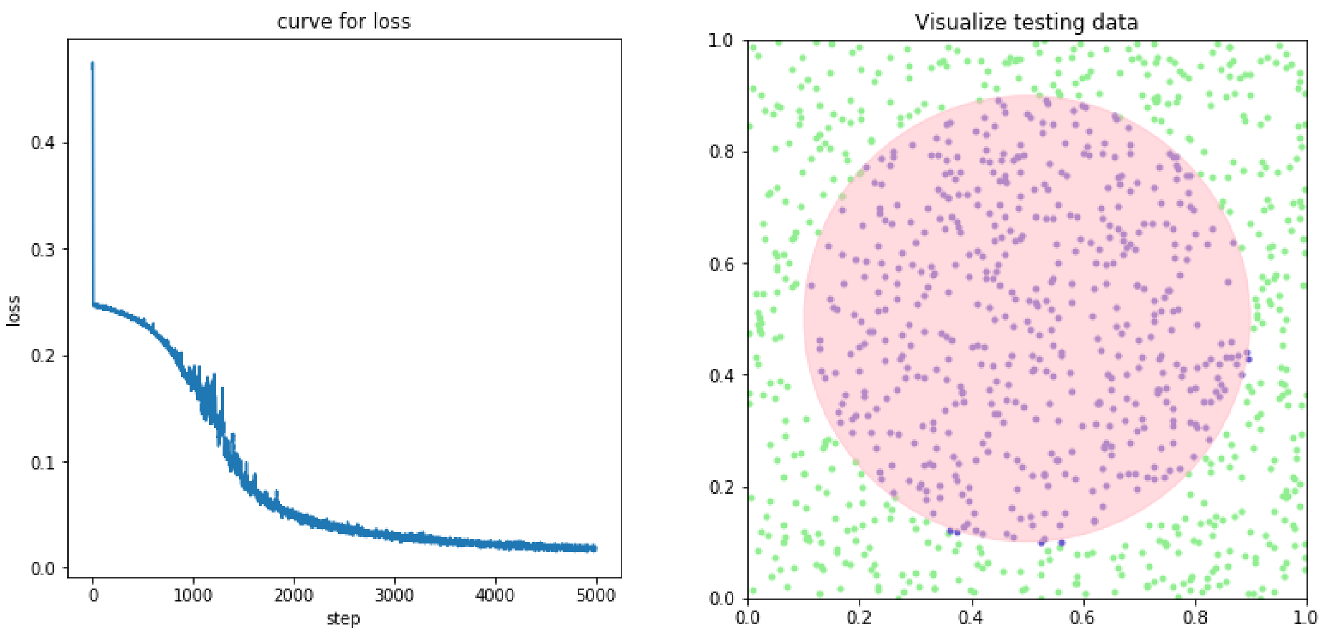


Figure 3: Performance for ANN2

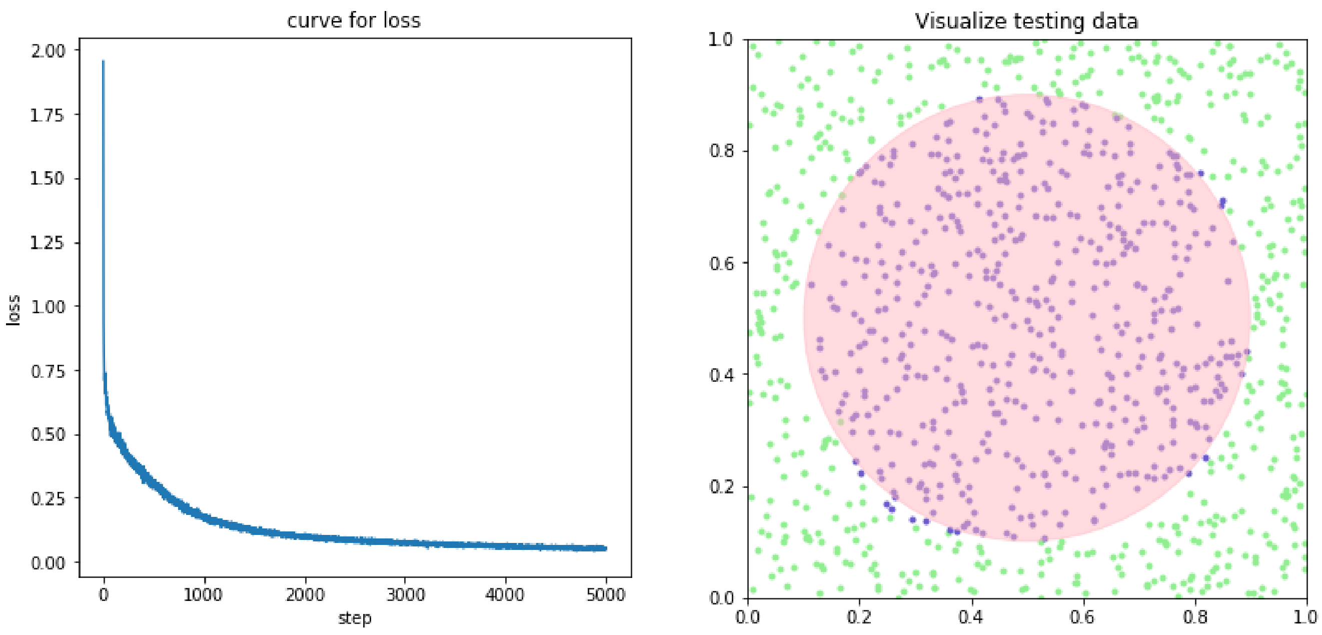


Figure 4: Performance for ANN3

Table 5: Performance of Three ANN Models

ANN	**Error Rate %**		**Variation**		**Time(s)**
	Train	Test	Train	Test	
1	1.22	2.14	0.07	0.31	21.74
2	1.80	1.84	1.92	1.38	20.42
3	0.72	1.32	0.14	0.16	40.5

Bibliography

[1] "Back propagation for a Linear Layer"[Online]. Available: http://cs231n.stanford.edu/handouts/linear-backprop.pdf. [Accessed: 25-Apr-2018].
[2] "Classification and Loss Evaluation - Softmax and Cross Entropy Loss" [Online]. Available: https://deepnotes.io/softmax-crossentropy. [Accessed: 25-Apr-2018].