lab1

March 16, 2019

資科工碩 0756110 李東霖

1 Introduction

In this lab, we need to implement NN and back propagation
Some request:

- Write a simple neural networks without framework (e.g. Tensorflow, PyTorch)
- Only use Numpy and other standard lib
- NN with two hidden layers
- Plot your comparison figure that show the predict result and ground truth

1.1 Implementation

- X,$\hat{y}$: Data
- x_1,x_2 : NN inputs
- y : NN output
- $L(\theta)$: Lost function (MSE $E(|\hat{y} - y|^2)$)
- W : weight matrix
- σ : activation function (sigmoid $\frac{1}{1+e^{-x}}$)

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from matplotlib.colors import LinearSegmentedColormap
```

1.2 Dataset

We have two data generator

- Linear
- XOR

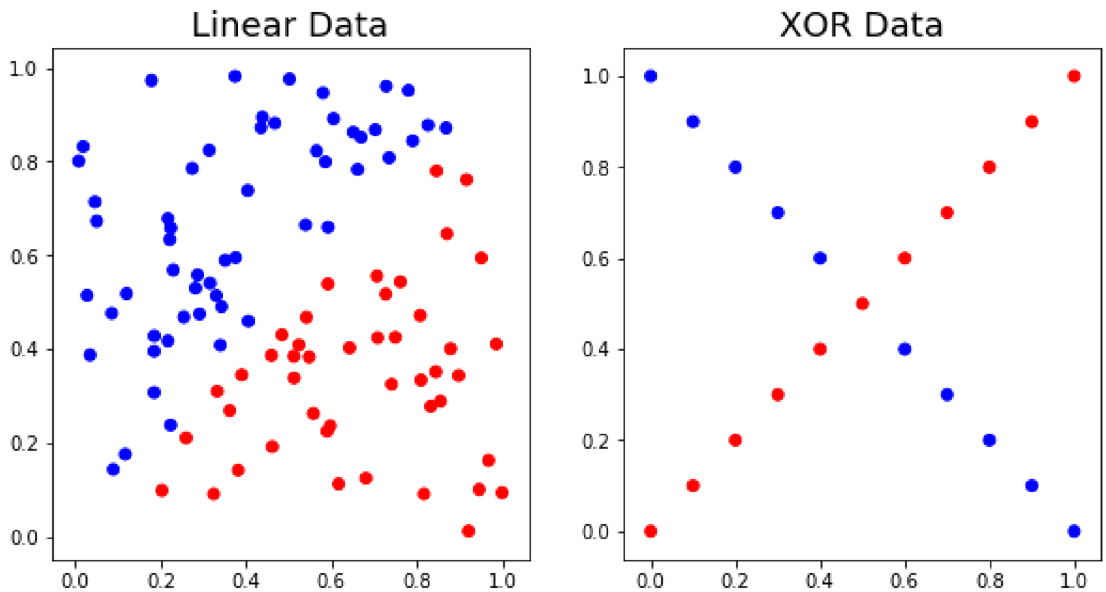
Target y is 0 or 1, just like one class classification.

```
In [2]: def show_result(x, y, pred_y):
            cm = LinearSegmentedColormap.from_list(
                'mymap', [(1, 0, 0), (0, 0, 1)], N=2)
            plt.figure(figsize=(10,5))
```

```
    plt.subplot(1,2,1)
    plt.title('Ground truth', fontsize=18)
    plt.scatter(x[:,0], x[:,1], c=y[:,0], cmap=cm)

    plt.subplot(1,2,2)
    plt.title('Predict result', fontsize=18)
    plt.scatter(x[:,0], x[:,1], c=pred_y[:,0], cmap=cm)

    plt.show()

def show_data(xs, ys, ts):
    cm = LinearSegmentedColormap.from_list(
        'mymap', [(1, 0, 0), (0, 0, 1)], N=2)
    n = len(xs)
    plt.figure(figsize=(5*n, 5))
    for i, x, y, t in zip(range(n), xs, ys, ts):
        plt.subplot(1,n, i+1)
        plt.title(t, fontsize=18)
        plt.scatter(x[:,0], x[:,1], c=y[:,0], cmap=cm)

    plt.show()
```

```
In [3]: def generate_linear(n=100):
    pts = np.random.uniform(0, 1, (n, 2))
    inputs = []
    labels = []
    for pt in pts:
        inputs.append([pt[0], pt[1]])
        distance = (pt[0] - pt[1]) / 1.414
        if pt[0] > pt[1]:
            labels.append(0)
        else:
            labels.append(1)
    return np.array(inputs), np.array(labels).reshape(n, 1)

def generate_XOR_easy(n=11):
    inputs = []
    labels = []
    step = 1/(n-1)
    for i in range(n):
        inputs.append([step*i, step*i])
        labels.append(0)

        if i == int((n-1)/2):
            continue

        inputs.append([step*i, 1 - step*i])
        labels.append(1)
```

```
    return np.array(inputs), np.array(labels).reshape(n*2 - 1,1)

x1, y1 = generate_linear()
x2, y2 = generate_XOR_easy()
show_data([x1,x2], [y1,y2], ['Linear Data', 'XOR Data'])
```



2 Experiment setups

2.1 Activate function σ (Sigmoid)

In this lab, I use Sigmoid function as my activate function

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\sigma'(x) = \frac{d(1+e^{-x})^{-1}}{dx}$$

$$= -(1+e^{-x})^2 \frac{d}{dx}(1+e^{-x})$$

$$= -(1+e^{-x})(1+e^{-x})(-e^{-x})$$

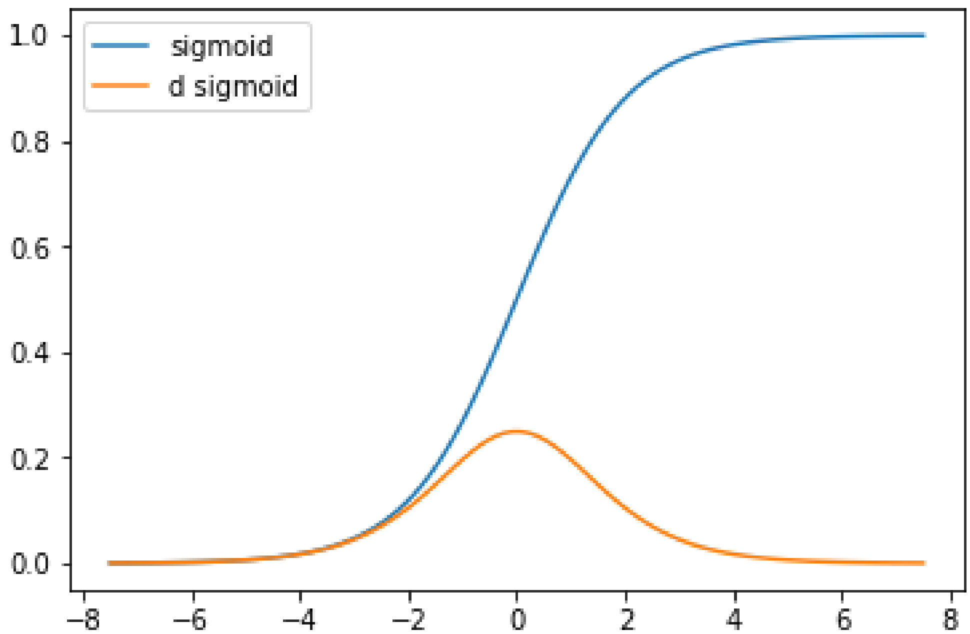
$$= \sigma(x)(1-\sigma(x))$$

implement reference from TAs.

```
In [4]: def sigmoid(x):
            return 1.0 / (1.0 + np.exp(-x))

        def derivative_sigmoid(x):
            return np.multiply(x, 1.0 - x)
```

```
In [5]: x = np.linspace(-7.5, 7.5, 100)
        plt.plot(x, sigmoid(x), label='sigmoid')
        plt.plot(x, derivative_sigmoid(sigmoid(x)), label='d sigmoid')
        plt.legend()
```

```
Out[5]: <matplotlib.legend.Legend at 0x7fa002340ba8>
```



2.2 Loss function $L(\theta)$ (MSE)

In this lab, I use MSE (Mean Square Error) as my loss function.

$$L(y, \hat{y}) = MSE(y, \hat{(y)}) = E((y - \hat{y})^2) = \frac{\sum(y - \hat{y})^2}{N}$$

$$L'(y, \hat{y}) = \frac{\partial E((y - \hat{y})^2)}{\partial y}$$

$$= \frac{1}{N}(\frac{\partial (y - \hat{y})^2}{\partial y})$$

$$= \frac{1}{N}(2(y - \hat{y})\frac{\partial (y - \hat{y})}{\partial y})$$

$$= \frac{2}{N}(y - \hat{y})$$

```
In [6]: def loss(y, y_hat):
            return np.mean((y - y_hat)**2)

        def derivative_loss(y, y_hat):
            return (y - y_hat)*(2/y.shape[0])
```

2.3 Neural network

2.3.1 Neural Unit

Our input x vector get output y scalar through neural unit
$z = w^T x + b, y = \sigma(z)$
Now extend neural unit as neural layer

2.3.2 Neural Layer

One neural unit can output one scalar. So if we want to output N scalar in this layer, we just put N units in layer.
explain some parameter in layer:
w : weight matrix

- size is (input_size + 1, output_size)
- initialize w in layer's `__init__`
- combine bias in w

x : input vector

- size is (data_size, input_size)
- x' automatically extend one columns for bias when `forward`

$z : z = x'w$

- size is (data_size, output_size)

$y : y = \sigma(z)$

- network output when output layer
- next layer input when hidden layer

$\frac{\partial C}{\partial w}, \frac{\partial z}{\partial w}, \frac{\partial C}{\partial z}$: gradient matrix

- there are stored into layer parameter
- use to update w when call `update`

Now, we see how to compute gradient from cost by using backpropagation

2.4 Backpropagation

In the begining, all weight parameters in network are randomly initial. And we want to minimize cost C from loss function $L(\theta)$.

So we use gradient descent to update network's weights. But $\frac{\partial C}{\partial w}$ is hard to compute.

Because of that, we use chain rules.

$$\frac{\partial C}{\partial w} = \frac{\partial z}{\partial w}\frac{\partial C}{\partial z}$$

2.4.1 Forward

$$\frac{\partial z}{\partial w} = \frac{\partial x'w}{\partial w} = x'$$

So we can record $\frac{\partial z}{\partial w}$ as `forward_gradient` when call `forward`

And matrix size = (data_size, input_size+1)

2.4.2 Backward

$$\frac{\partial C}{\partial z} = \frac{\partial y}{\partial z}\frac{\partial C}{\partial y}$$

we can get $\frac{\partial y}{\partial z}$ by:

$$y = \sigma(z), \frac{\partial y}{\partial z} = \sigma'(z)$$

We need to consider two case

- output layer:

we know C is come from $L(\theta)$ y is network output and $\hat{y}$ is groundtruth

$$C = L(y, \hat{y})\frac{\partial C}{\partial y} = L'(y, \hat{y})$$

we need to compute derivative loss function and then use it as backward input.

- hidden layer:

$\frac{\partial C}{\partial y}$ is more diffcult than other.

we know that this layer output y will be input for next layer. and we assume that $\frac{\partial C}{\partial z_{next}}$ already know.

$$\frac{\partial C}{\partial y_{this}} = \frac{\partial z_{next}}{\partial y_{this}}\frac{\partial C}{\partial z_{next}}$$

$$\frac{\partial z_{next}}{\partial y_{this}} = w_{next}^T, z_{next} = y_{this}w_{next}$$

Finally, we first compute output layer and then send parameters to previous layer. Thus we can compute $\frac{\partial C}{\partial z}$ every layer.

2.5 Gradient Descent

Now we have $\frac{\partial C}{\partial w}$ and use it to update our network weights w.
we can put a new hyperparameter called learning rate η to decide how fast

$$w = w - \eta \Delta w$$

2.6 implementation

I design a python class called `layer`. `layer` will initialize all weights when create python class. Every `layer` need two parameter `input_size` and `output_size`.

- `forward` function input x and get output y.
- `backward` function input $\frac{\partial C}{\partial y}$ and get output $\frac{\partial C}{\partial x}$
- `update` function use gradient to update layer's weights

In [7]:
```
class layer():
    def __init__(self, input_size, output_size):
        self.w = np.random.normal(0, 1, (input_size+1, output_size))

    def forward(self, x):
        x = np.append(x, np.ones((x.shape[0],1)), axis=1)
        self.forward_gradient = x
        self.y = sigmoid(np.matmul(x, self.w))
        return self.y

    def backward(self, derivative_C):
        self.backward_gradient = np.multiply(
            derivative_sigmoid(self.y),
            derivative_C
        )
        return np.matmul(self.backward_gradient, self.w[:-1].T)

    def update(self, learning_rate):
        self.gradient = np.matmul(
            self.forward_gradient.T,
            self.backward_gradient
        )
        self.w -= learning_rate*self.gradient
        return self.gradient
```

Now I can combine multi layers become Neural Network
I design a python class called `NN`. `NN` will create layers by `size` when create it.

- `forward` function positive sequence call all layer's `forward`, return final result
- `backward` function reverse call all layer's `backward`, return final result
- `update` function call all layer's `update`

```
In [8]: class NN():
            def __init__(self, sizes, learning_rate = 0.1):
                self.learning_rate = learning_rate
                sizes2 = sizes[1:] + [0]
                self.l = []
                for a,b in zip(sizes, sizes2):
                    if (a+1)*b == 0:
                        continue
                    self.l += [layer(a,b)]

            def forward(self, x):
                _x = x
                for l in self.l:
                    _x = l.forward(_x)
                return _x

            def backward(self, dC):
                _dC = dC
                for l in self.l[::-1]:
                    _dC = l.backward(_dC)

            def update(self):
                gradients = []
                for l in self.l:
                    gradients += [l.update(self.learning_rate)]
                return gradients
```

3 Results of your testing

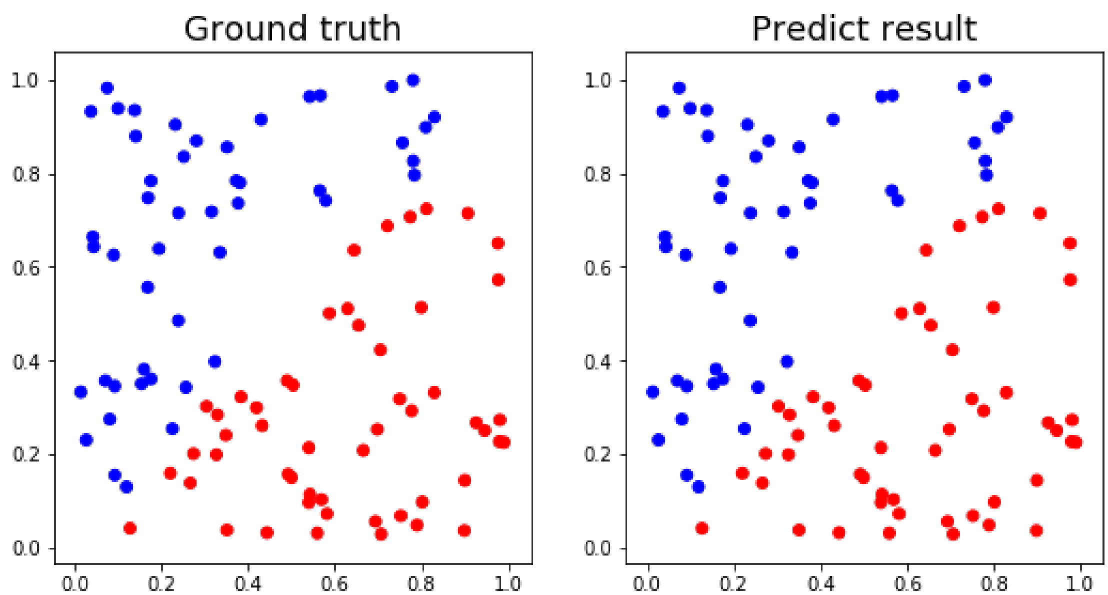
```
In [13]: nn_linear = NN([2,4,4,1], 1)
         nn_XOR = NN([2,4,4,1], 1)
         epoch_count = 10000
         loss_threshold = 0.005
         linear_stop = False
         XOR_stop = False
         x_linear, y_linear = generate_linear()
         x_XOR, y_XOR = generate_XOR_easy()
         for i in range(epoch_count):
             if not linear_stop:
                 y = nn_linear.forward(x_linear)
                 loss_linear = loss(y, y_linear)
                 nn_linear.backward(derivative_loss(y, y_linear))
                 nn_linear.update()

                 if loss_linear < loss_threshold:
                     print('linear is covergence')
                     linear_stop = True
```

```python
            if not XOR_stop:
                y = nn_XOR.forward(x_XOR)
                loss_XOR = loss(y, y_XOR)
                nn_XOR.backward(derivative_loss(y, y_XOR))
                nn_XOR.update()

                if loss_XOR < loss_threshold:
                    print('XOR is covergence')
                    XOR_stop = True

            if i%200 == 0 or (linear_stop and XOR_stop):
                print(
                    '[{:4d}] linear loss : {:.4f} \t XOR loss : {:.4f}'.format(
                        i, loss_linear, loss_XOR))

            if linear_stop and XOR_stop:
                break
```

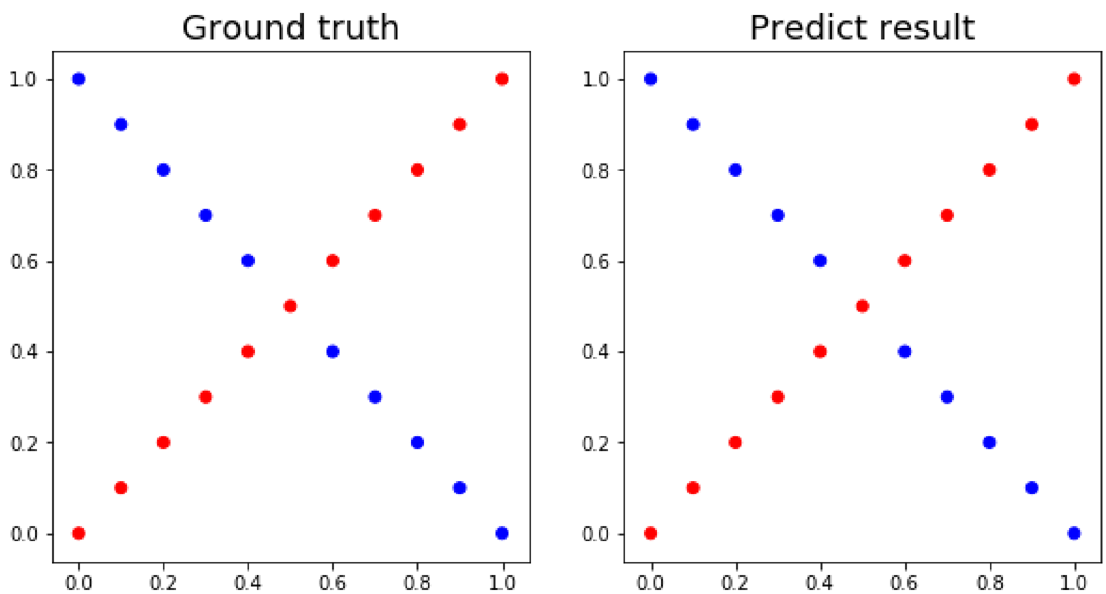
```
[   0] linear loss : 0.3835          XOR loss : 0.2512
[ 200] linear loss : 0.1824          XOR loss : 0.2461
[ 400] linear loss : 0.0497          XOR loss : 0.2360
[ 600] linear loss : 0.0287          XOR loss : 0.2179
[ 800] linear loss : 0.0210          XOR loss : 0.2035
[1000] linear loss : 0.0169          XOR loss : 0.1885
[1200] linear loss : 0.0144          XOR loss : 0.1131
[1400] linear loss : 0.0127          XOR loss : 0.0459
[1600] linear loss : 0.0115          XOR loss : 0.0226
[1800] linear loss : 0.0105          XOR loss : 0.0124
[2000] linear loss : 0.0098          XOR loss : 0.0077
[2200] linear loss : 0.0092          XOR loss : 0.0053
XOR is covergence
[2400] linear loss : 0.0087          XOR loss : 0.0050
[2600] linear loss : 0.0083          XOR loss : 0.0050
[2800] linear loss : 0.0079          XOR loss : 0.0050
[3000] linear loss : 0.0076          XOR loss : 0.0050
[3200] linear loss : 0.0073          XOR loss : 0.0050
[3400] linear loss : 0.0070          XOR loss : 0.0050
[3600] linear loss : 0.0068          XOR loss : 0.0050
[3800] linear loss : 0.0065          XOR loss : 0.0050
[4000] linear loss : 0.0063          XOR loss : 0.0050
[4200] linear loss : 0.0061          XOR loss : 0.0050
[4400] linear loss : 0.0059          XOR loss : 0.0050
[4600] linear loss : 0.0057          XOR loss : 0.0050
[4800] linear loss : 0.0054          XOR loss : 0.0050
[5000] linear loss : 0.0052          XOR loss : 0.0050
[5200] linear loss : 0.0051          XOR loss : 0.0050
linear is covergence
```

```
[5254] linear loss : 0.0050          XOR loss : 0.0050
```

```
In [14]: y1 = nn_linear.forward(x_linear)
         show_result(x_linear, y_linear, y1)
         print('linear test loss : ', loss(y1, y_linear))
         y2 = nn_XOR.forward(x_XOR)
         show_result(x_XOR, y_XOR, y2)
         print('XOR test loss : ', loss(y2, y_XOR))
         print('\n linear test result : \n',y1)
         print('\n XOR test result : \n',y2)
```




```
linear test loss :  0.004998301926023007
```




```
XOR test loss :  0.004984722899437262

 linear test result :
 [[0.00127438]
 [0.99963048]
 [0.00110397]
 [0.99967442]
 [0.00244083]
 [0.99963929]
 [0.9987874 ]
 [0.00140453]
 [0.00131287]
 [0.00150334]
 [0.99956104]
 [0.99895211]
 [0.67410639]
 [0.99927398]
 [0.00112912]
 [0.98738375]
 [0.97859111]
 [0.99943916]
 [0.99844841]
 [0.99937749]
 [0.99902285]
 [0.00121178]
 [0.99881502]
 [0.00108736]
```

```
[0.00163561]
[0.9346675 ]
[0.00543449]
[0.01082715]
[0.99968211]
[0.00730359]
[0.99942062]
[0.99877801]
[0.0011435 ]
[0.00111643]
[0.99923101]
[0.99773533]
[0.99365109]
[0.00146336]
[0.99960882]
[0.00135928]
[0.2934638 ]
[0.00129483]
[0.99946953]
[0.09278763]
[0.82878393]
[0.00112694]
[0.00127975]
[0.00120702]
[0.00291165]
[0.99968023]
[0.99961166]
[0.99935175]
[0.99955449]
[0.00110964]
[0.00126752]
[0.9991323 ]
[0.00111803]
[0.99939436]
[0.99962201]
[0.00122198]
[0.02690406]
[0.00112542]
[0.39744108]
[0.99961734]
[0.99966773]
[0.05587581]
[0.00154267]
[0.00150998]
[0.00317554]
[0.99947727]
[0.00494795]
[0.00215899]
```

```
[0.99964369]
[0.68546723]
[0.02156635]
[0.99842914]
[0.00559345]
[0.00526649]
[0.00125945]
[0.02088807]
[0.98882897]
[0.99862005]
[0.00110082]
[0.00158562]
[0.00122363]
[0.00126444]
[0.01860277]
[0.99954892]
[0.00110318]
[0.01020721]
[0.99950236]
[0.99964959]
[0.00424223]
[0.98935269]
[0.03530341]
[0.99945057]
[0.99966074]
[0.99952167]
[0.97406166]
[0.00111279]]

XOR test result :
[[0.06823995]
[0.99178587]
[0.06777908]
[0.99120513]
[0.06743711]
[0.98941854]
[0.06720555]
[0.98084773]
[0.06707543]
[0.82957466]
[0.06703769]
[0.06708346]
[0.85767423]
[0.06720427]
[0.96107665]
[0.06739225]
[0.96658707]
[0.06764024]
```

```
 [0.96769416]
 [0.06794189]
 [0.96804362]]
```

```
In [ ]:
```