ECE 1513 Introduction to Machine Learning

Assignment 3 Part I

Zihao Jiao (1002213428)

Date: Feb25th, 2020

Part A

Logistic function:

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

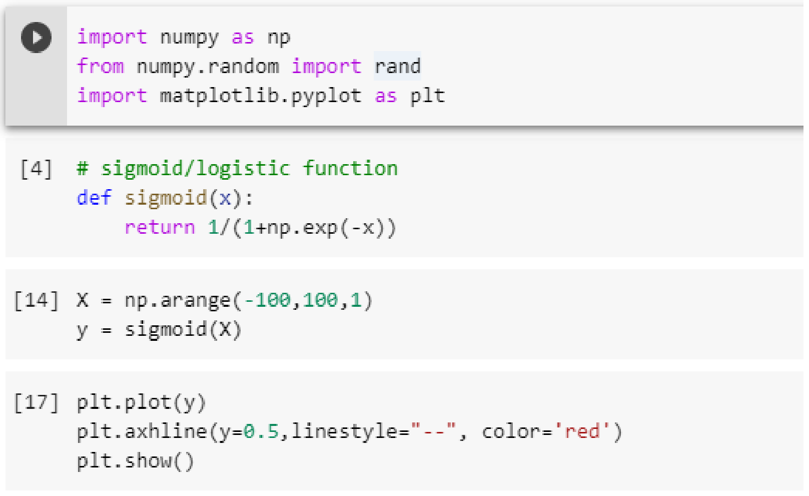
Analytically:

$$\sigma(z) = \frac{1}{1 + e^{-\infty}} \approx 1$$

$$\sigma(z) = \frac{1}{1 + e^{-(-\infty)}} \approx 0$$

Logistic function basically is a non-linear function, which squashes the results to be between 0 and 1. It is smooth everywhere, thus, it is possible to compute derivative. Thus, it is good idea of binary classifier function. And threshold is 0.5, any result greater than this threshold will be predicted as positive, any result less than this threshold will be predicted as negative.

Below is the python code to implement logistic function:

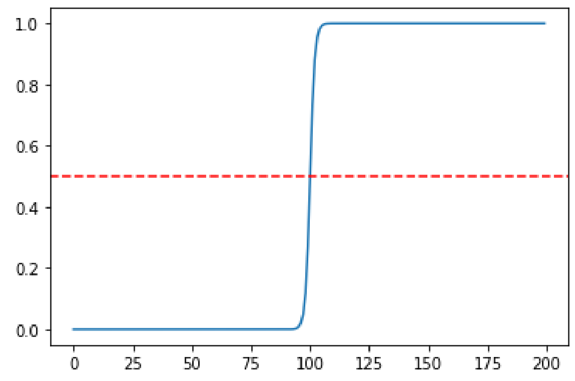


```
import numpy as np
from numpy.random import rand
import matplotlib.pyplot as plt
```

```
[4]  # sigmoid/logistic function
     def sigmoid(x):
         return 1/(1+np.exp(-x))
```

```
[14] X = np.arange(-100,100,1)
     y = sigmoid(X)
```

```
[17] plt.plot(y)
     plt.axhline(y=0.5,linestyle="--", color='red')
     plt.show()
```



As we can see from graph, logistic function is a S-shaped function, there are two asymptotes 0 and 1, which means any value goes in this function, the result will be squashed to 0 and 1. Besides, it increases monotonically and smooth everywhere.

Logistic function Derivative:

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

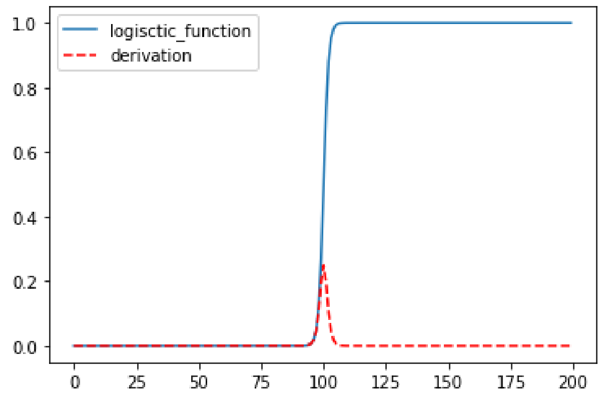
$$\sigma(z)' = \frac{e^{-z}}{(1 + e^{-z})^2} = \frac{1}{1 + e^{-z}} * (1 - \frac{1}{1 + e^{-z}}) = \sigma(z)(1 - \sigma(z))$$

Below is the python code to implement derivative of logistic function:

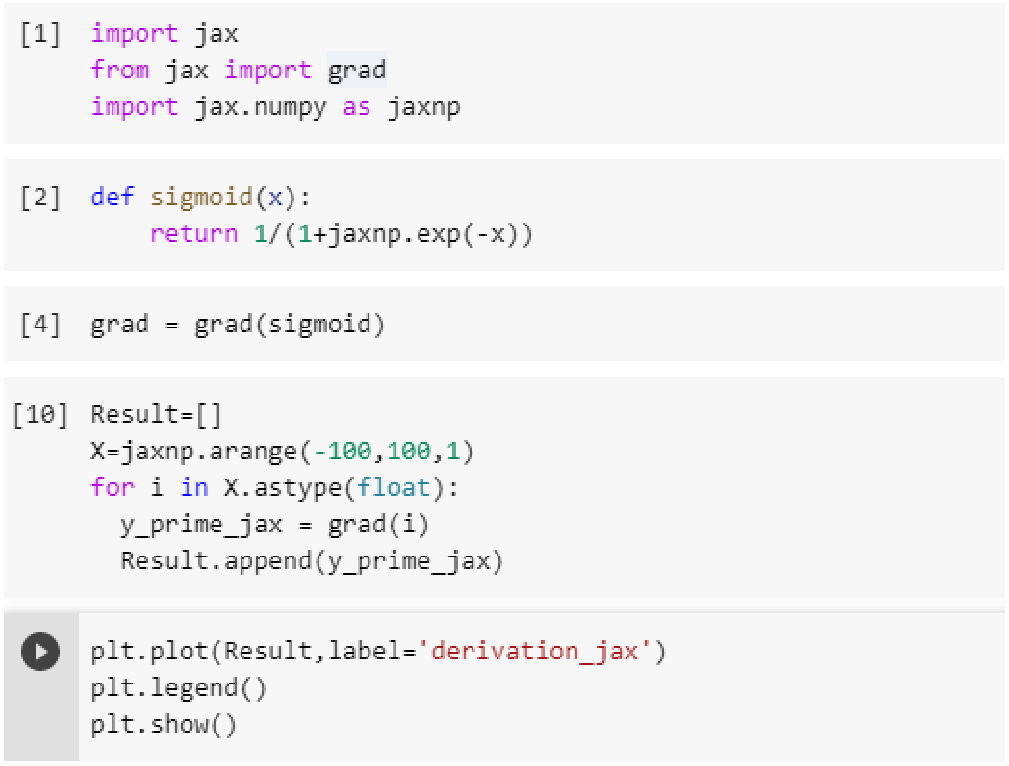
```
#derivative of logistic function
def logistic_derivative(x):
  f = 1/(1+np.exp(-x))#logistic function itself
  return f * (1 - f)
```

```
[6]  X = np.arange(-100,100,1)
     y_prime = logistic_derivative(X)
```

```
[7]  plt.plot(y,label='logisctic_function')
     plt.plot(y_prime,linestyle="--", color='red',label='derivation')
     plt.legend()
     plt.show()
```



By using Jax package:



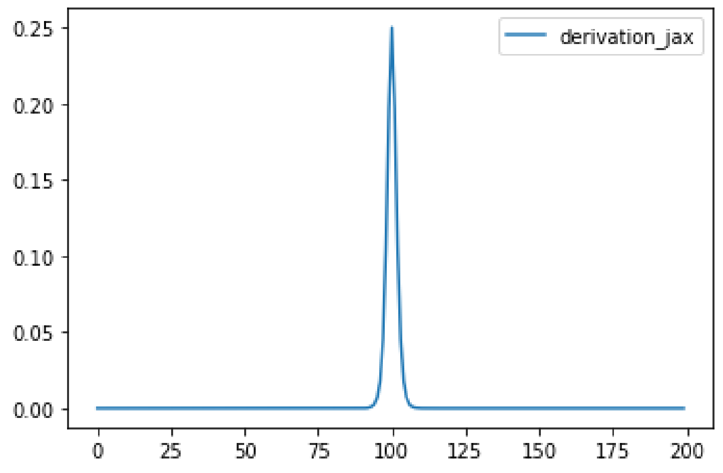
```
[1]  import jax
     from jax import grad
     import jax.numpy as jaxnp
```

```
[2]  def sigmoid(x):
         return 1/(1+jaxnp.exp(-x))
```

```
[4]  grad = grad(sigmoid)
```

```
[10] Result=[]
     X=jaxnp.arange(-100,100,1)
     for i in X.astype(float):
       y_prime_jax = grad(i)
       Result.append(y_prime_jax)
```

```
plt.plot(Result,label='derivation_jax')
plt.legend()
plt.show()
```



The results from Jax and NumPy are identical, verified.

SoftMax function:

$$\sigma(Z)_i = \frac{e^{Z_k}}{\sum_{j=1}^{K} e^{Z_k}}$$

Where i = 1,…,K; Z = $(Z_1, \dots, Z_K)$.

Analytically, we try to plug in K = 2:

$$\sigma(Z) = \frac{e^{Z_1}}{e^{Z_1} + e^{Z_2}} + \frac{e^{Z_2}}{e^{Z_1} + e^{Z_2}} = 1$$

Basically, SoftMax function's outputs are nonnegative and summation of all elements are 1, it is a normalized exponential function, which can be used to represent a categorical distribution – that is, a probability distribution over K different possible outcomes.

Below is the Softmax function implementation in python:

```
[1]  from jax import grad
     import jax.numpy as jaxnp
     import matplotlib.pyplot as plt
     import numpy as np
```

```
[2]  data = np.array([1, 2, 3, 4, 5, 1, 2, 5])
```

```
[3]  np.arange(4)
```

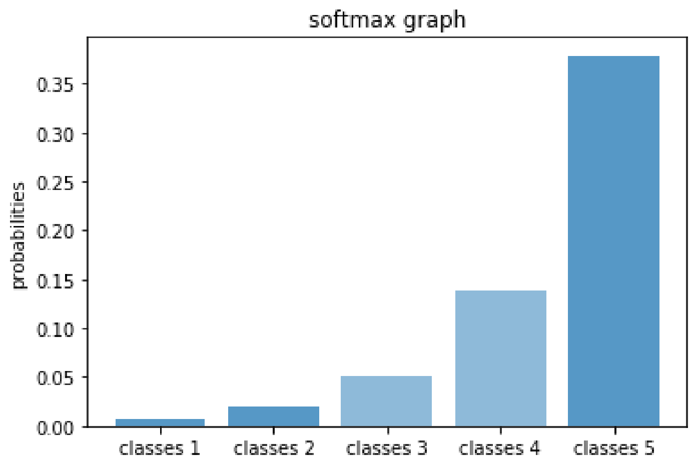
```
array([0, 1, 2, 3])
```

```
[4]  def softmax(x):
       return (np.exp(x)) / np.sum(np.exp(x))
```

```
[5]  res = softmax(data)
     res
```

```
array([0.00693927, 0.01886288, 0.05127463, 0.13937889, 0.3788711 ,
       0.00693927, 0.01886288, 0.3788711 ])
```

```
[6]  plt.bar(data,res,align='center',alpha=0.5)
     plt.xticks(data,('classes 1','classes 2','classes 3', 'classes 4', 'classes 5'))
     plt.ylabel('probabilities')
     plt.title('softmax graph')
     plt.show()
```



As we can see from the result, the array ‘data’ has 5 classes, the result shows probabilities of each class. The summation of them are equal to 1 at the end.

Derivative:

For i = j case:

$$\frac{\partial \sigma(Z)}{\partial x} = \frac{e^{Z_k} \sum_{j=1}^{K} e^{Z_{k_j}} - e^{Z_k} e^{Z_{k_j}}}{\left(\sum_{j=1}^{K} e^{Z_{k_j}}\right)^2}$$

$$= \frac{e^{Z_k}}{\sum_{j=1}^{K} e^{Z_{k_j}}} \frac{\sum_{j=1}^{K} e^{Z_{k_j}} - e^{Z_{k_j}}}{\sum_{j=1}^{K} e^{Z_{k_j}}}$$

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

For i ≠ j case:

$$\frac{\partial \sigma(Z)}{\partial x} = \frac{0 - e^{Z_k} e^{Z_{k_j}}}{\left(\sum_{j=1}^{K} e^{Z_{k_j}}\right)^2}$$

$$= -\frac{e^{Z_{k_j}}}{\sum_{j=1}^{K} e^{Z_{k_j}}} \frac{e^{Z_k}}{\sum_{j=1}^{K} e^{Z_{k_j}}}$$

$$= -\,\sigma(Z)_j \sigma(Z)$$

So overall,

$$\frac{\partial \sigma(Z)}{\partial x} = \begin{cases} \sigma(Z)\big(1 - \sigma(Z)\big), i = j \\ -\,\sigma(Z)_j \sigma(Z), i \neq j \end{cases}$$

Below is the code for implementation of derivative of softmax function in python:

```
[7] def softmax_derivative(x):
        softmax = (np.exp(x)) / np.sum(np.exp(x))
        result = softmax.reshape(-1,1)
        return np.diagflat(result) - np.dot(result, result.T)
```

```
[8] der_softmax = softmax_derivative(data)
```

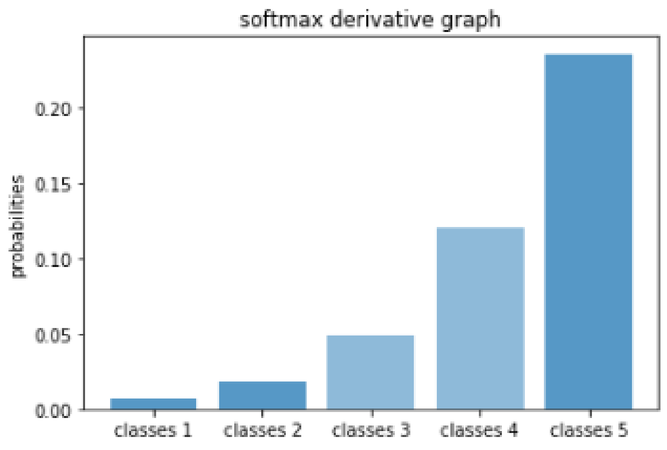
```
[9] print(der_softmax)
```

```
[[ 6.89111276e-03 -1.30894553e-04 -3.55808285e-04 -9.67187196e-04
  -2.62908738e-03 -4.81534150e-05 -1.30894553e-04 -2.62908738e-03]
 [-1.30894553e-04  1.85070729e-02 -9.67187196e-04 -2.62908738e-03
  -7.14660045e-03 -1.30894553e-04 -3.55808285e-04 -7.14660045e-03]
 [-3.55808285e-04 -9.67187196e-04  4.86455397e-02 -7.14660045e-03
  -1.94264741e-02 -3.55808285e-04 -9.67187196e-04 -1.94264741e-02]
 [-9.67187196e-04 -2.62908738e-03 -7.14660045e-03  1.19952413e-01
  -5.28066316e-02 -9.67187196e-04 -2.62908738e-03 -5.28066316e-02]
 [-2.62908738e-03 -7.14660045e-03 -1.94264741e-02 -5.28066316e-02
   2.35327789e-01 -2.62908738e-03 -7.14660045e-03 -1.43543307e-01]
 [-4.81534150e-05 -1.30894553e-04 -3.55808285e-04 -9.67187196e-04
  -2.62908738e-03  6.89111276e-03 -1.30894553e-04 -2.62908738e-03]
 [-1.30894553e-04 -3.55808285e-04 -9.67187196e-04 -2.62908738e-03
  -7.14660045e-03 -1.30894553e-04  1.85070729e-02 -7.14660045e-03]
 [-2.62908738e-03 -7.14660045e-03 -1.94264741e-02 -5.28066316e-02
  -1.43543307e-01 -2.62908738e-03 -7.14660045e-03  2.35327789e-01]]
```

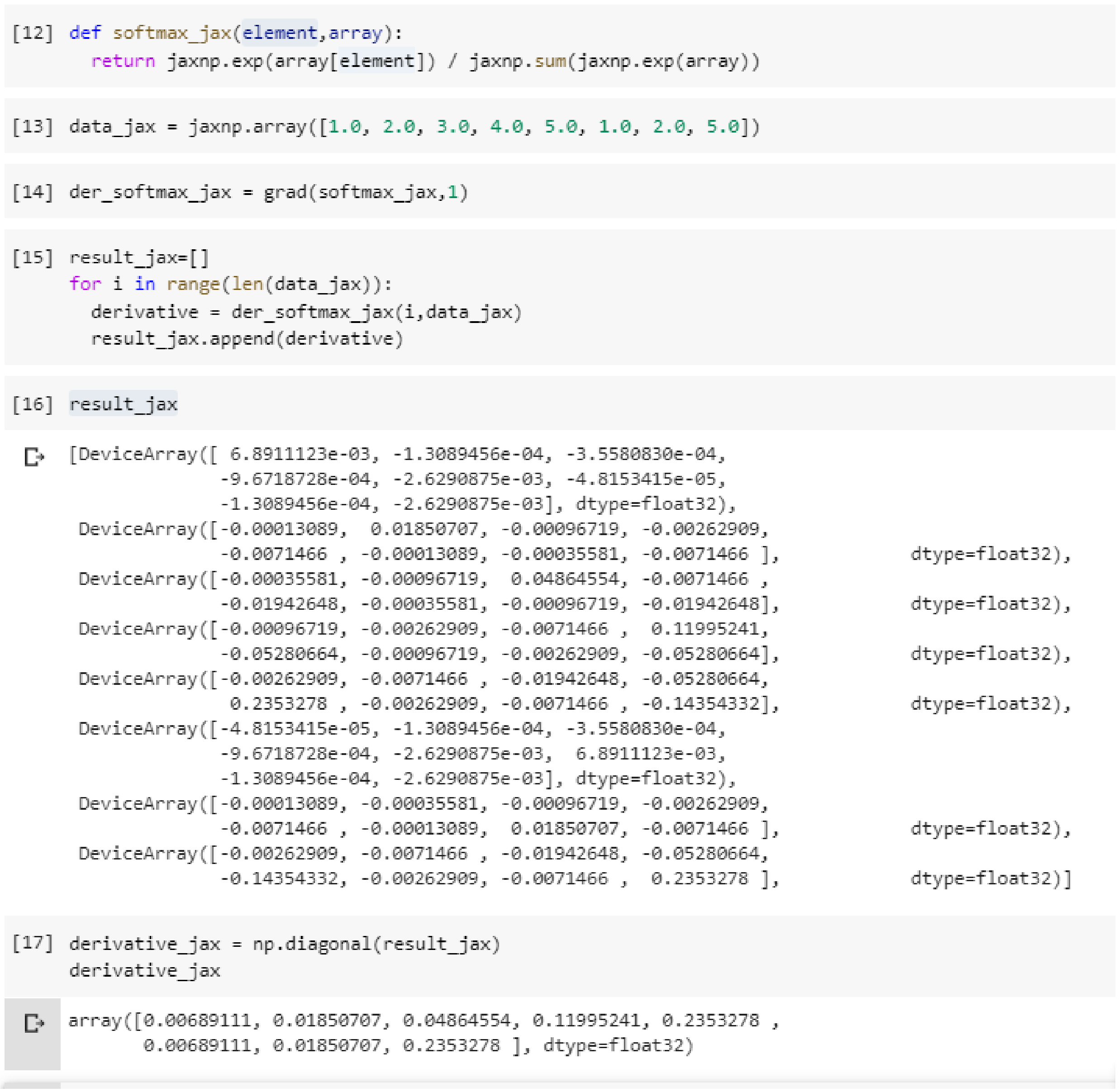
```
[10] derivative = np.diagonal(der_softmax)
     derivative
```

```
array([0.00689111, 0.01850707, 0.04864554, 0.11995241, 0.23532779,
       0.00689111, 0.01850707, 0.23532779])
```

```
[11] plt.bar(data,derivative,align='center',alpha=0.5)
     plt.xticks(data,('classes 1','classes 2','classes 3', 'classes 4', 'classes 5'))
     plt.ylabel('probabilities')
     plt.title('softmax derivative graph')
     plt.show()
```



Blow is the code for implementing jax to calculate derivative of softmax:



```
[12] def softmax_jax(element,array):
       return jaxnp.exp(array[element]) / jaxnp.sum(jaxnp.exp(array))
```

```
[13] data_jax = jaxnp.array([1.0, 2.0, 3.0, 4.0, 5.0, 1.0, 2.0, 5.0])
```

```
[14] der_softmax_jax = grad(softmax_jax,1)
```

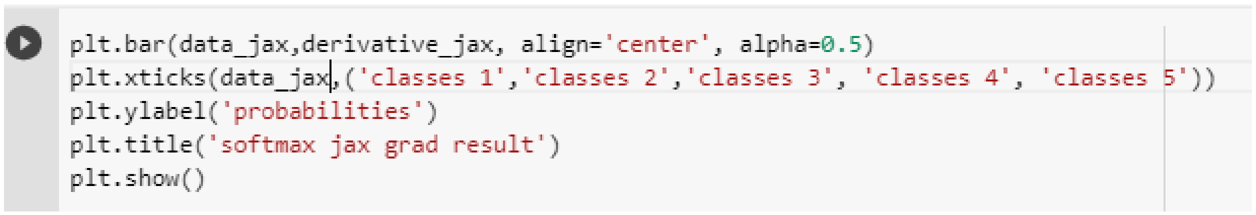
```
[15] result_jax=[]
     for i in range(len(data_jax)):
       derivative = der_softmax_jax(i,data_jax)
       result_jax.append(derivative)
```

```
[16] result_jax
```

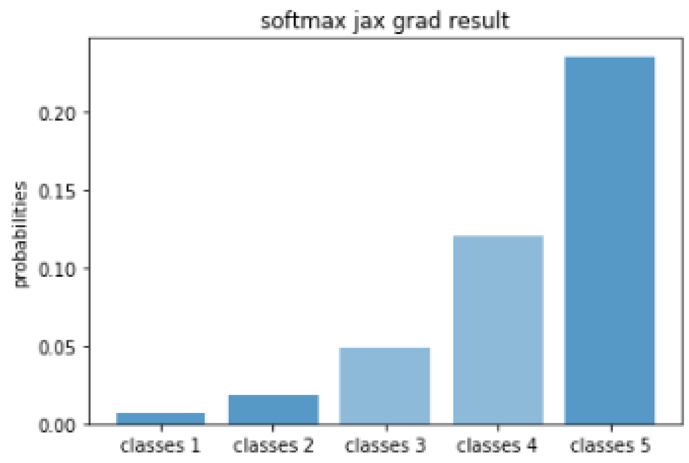
```
[DeviceArray([ 6.8911123e-03, -1.3089456e-04, -3.5580830e-04,
              -9.6718728e-04, -2.6290875e-03, -4.8153415e-05,
              -1.3089456e-04, -2.6290875e-03], dtype=float32),
 DeviceArray([-0.00013089,  0.01850707, -0.00096719, -0.00262909,
              -0.0071466 , -0.00013089, -0.00035581, -0.0071466 ],            dtype=float32),
 DeviceArray([-0.00035581, -0.00096719,  0.04864554, -0.0071466 ,
              -0.01942648, -0.00035581, -0.00096719, -0.01942648],            dtype=float32),
 DeviceArray([-0.00096719, -0.00262909, -0.0071466 ,  0.11995241,
              -0.05280664, -0.00096719, -0.00262909, -0.05280664],            dtype=float32),
 DeviceArray([-0.00262909, -0.0071466 , -0.01942648, -0.05280664,
               0.2353278 , -0.00262909, -0.0071466 , -0.14354332],            dtype=float32),
 DeviceArray([-4.8153415e-05, -1.3089456e-04, -3.5580830e-04,
              -9.6718728e-04, -2.6290875e-03,  6.8911123e-03,
              -1.3089456e-04, -2.6290875e-03], dtype=float32),
 DeviceArray([-0.00013089, -0.00035581, -0.00096719, -0.00262909,
              -0.0071466 , -0.00013089,  0.01850707, -0.0071466 ],            dtype=float32),
 DeviceArray([-0.00262909, -0.0071466 , -0.01942648, -0.05280664,
              -0.14354332, -0.00262909, -0.0071466 ,  0.2353278 ],            dtype=float32)]
```

```
[17] derivative_jax = np.diagonal(result_jax)
     derivative_jax
```

```
array([0.00689111, 0.01850707, 0.04864554, 0.11995241, 0.2353278 ,
       0.00689111, 0.01850707, 0.2353278 ], dtype=float32)
```



```
plt.bar(data_jax,derivative_jax, align='center', alpha=0.5)
plt.xticks(data_jax,('classes 1','classes 2','classes 3', 'classes 4', 'classes 5'))
plt.ylabel('probabilities')
plt.title('softmax jax grad result')
plt.show()
```



The results from Jax and NumPy are identical, verified. For derivative of softmax function, the output result is a matrix, we only interested in diagonal values (i=j).

Relu function:

The main idea behind the ReLu activation function is to perform a threshold operation to each input element where values less than zero are set to zero and any value greater than zero is the value itself.

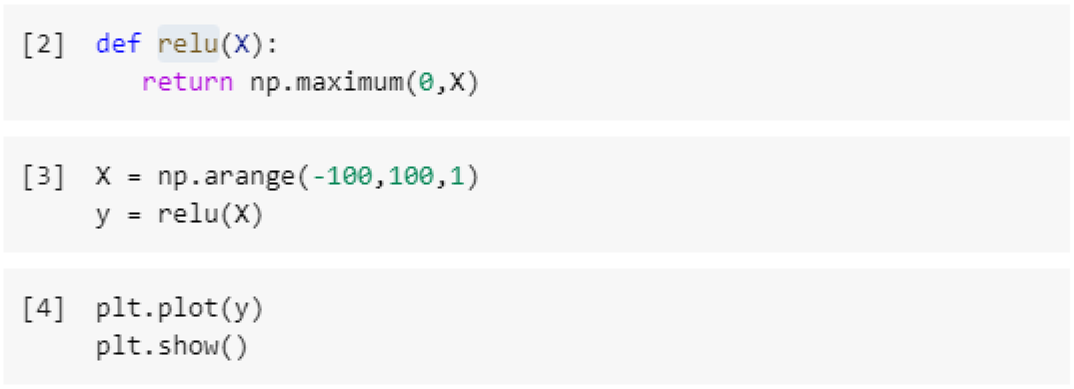
$$f(x) = \max(0, x) = \begin{cases} x_i \; if \; x_i > 0 \\ 0 \; if \; x_i < 0 \end{cases}$$

Analytically:

$$f(\infty) = \infty$$

$$f(-\infty) = 0$$

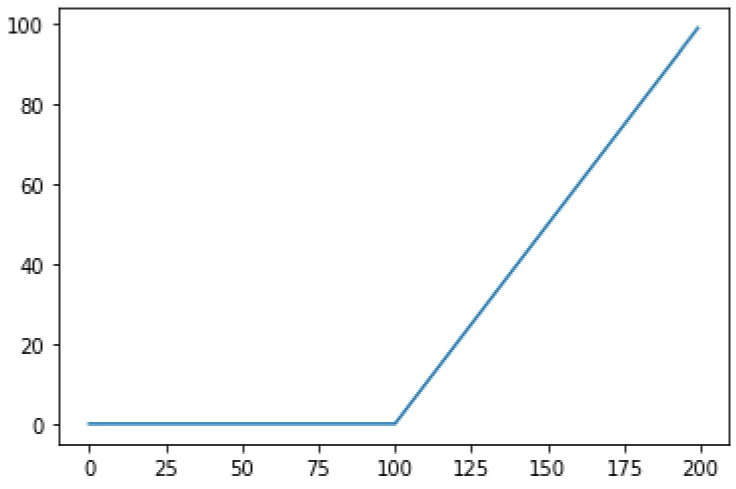
Below is the code of Relu function implementation in python:



```
[2]  def relu(X):
        return np.maximum(0,X)
```

```
[3]  X = np.arange(-100,100,1)
     y = relu(X)
```

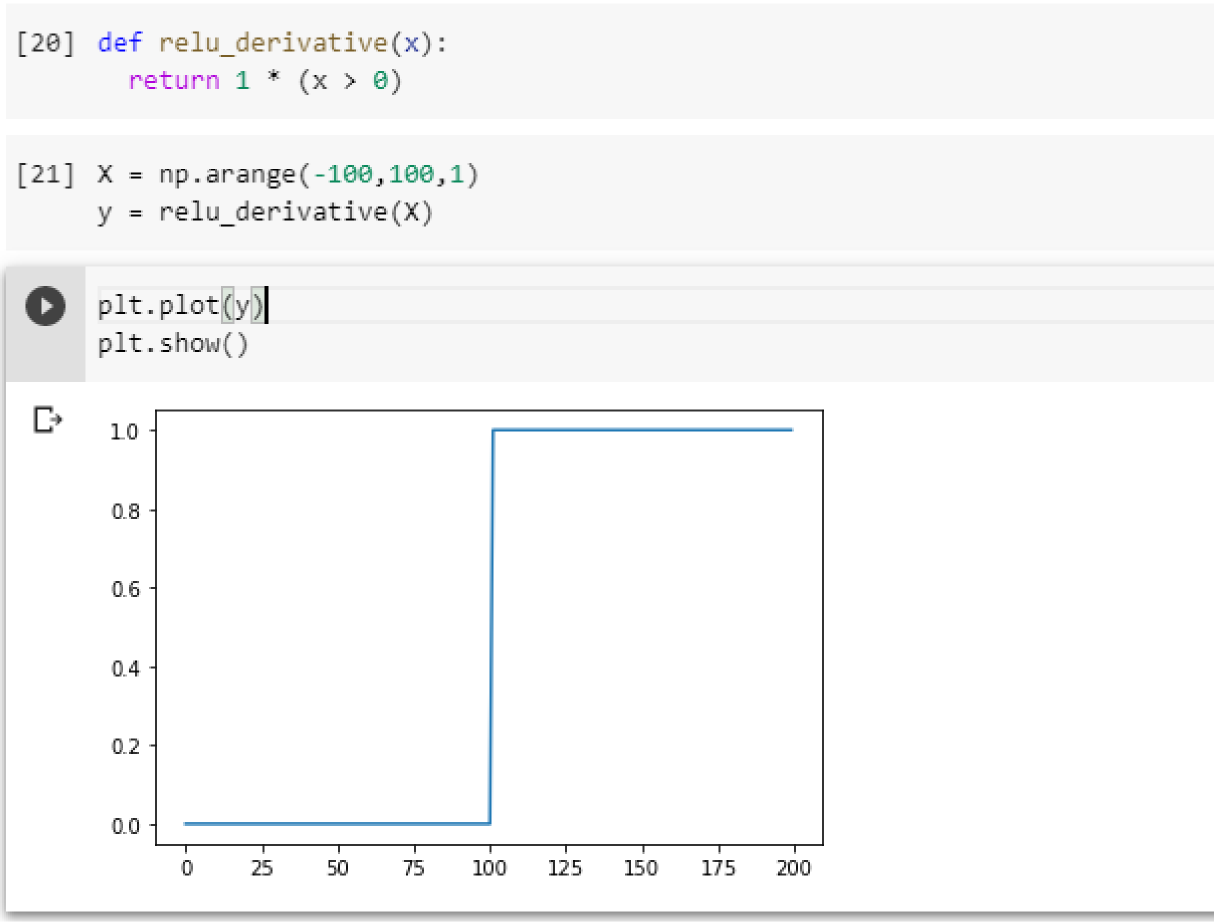
```
[4]  plt.plot(y)
     plt.show()
```



Derivative:

$$\frac{\partial Relu(x)}{\partial x} = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

Below is the code for implementation of derivative of Relu in python:

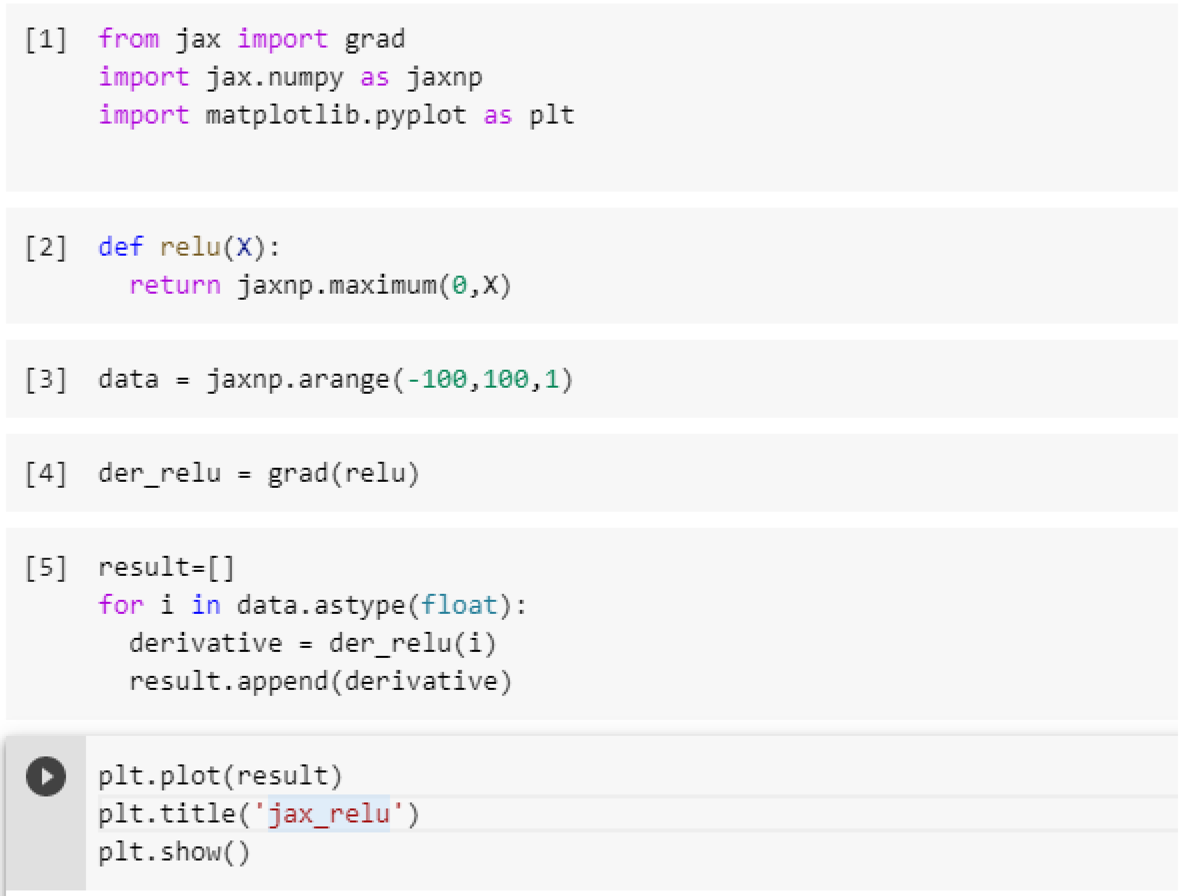


```
[20] def relu_derivative(x):
       return 1 * (x > 0)
```

```
[21] X = np.arange(-100,100,1)
     y = relu_derivative(X)
```

```
plt.plot(y)
plt.show()
```

Below is the code for implementing jax on Relu function:



```
[1]  from jax import grad
     import jax.numpy as jaxnp
     import matplotlib.pyplot as plt
```

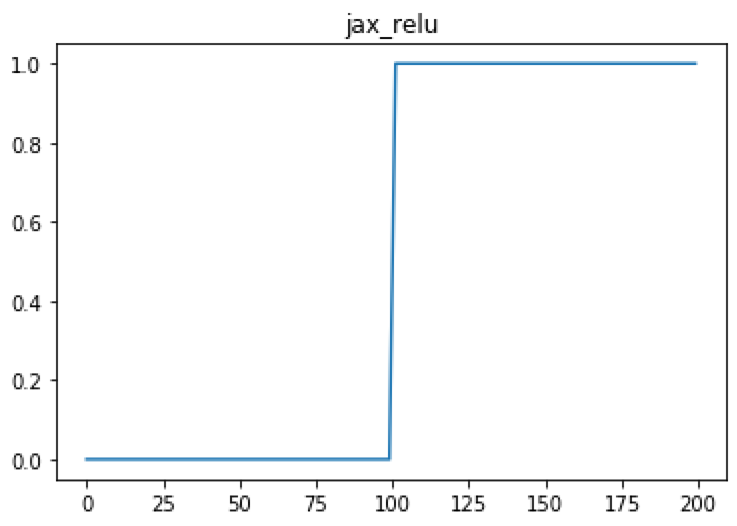
```
[2]  def relu(X):
       return jaxnp.maximum(0,X)
```

```
[3]  data = jaxnp.arange(-100,100,1)
```

```
[4]  der_relu = grad(relu)
```

```
[5]  result=[]
     for i in data.astype(float):
       derivative = der_relu(i)
       result.append(derivative)
```

```
plt.plot(result)
plt.title('jax_relu')
plt.show()
```

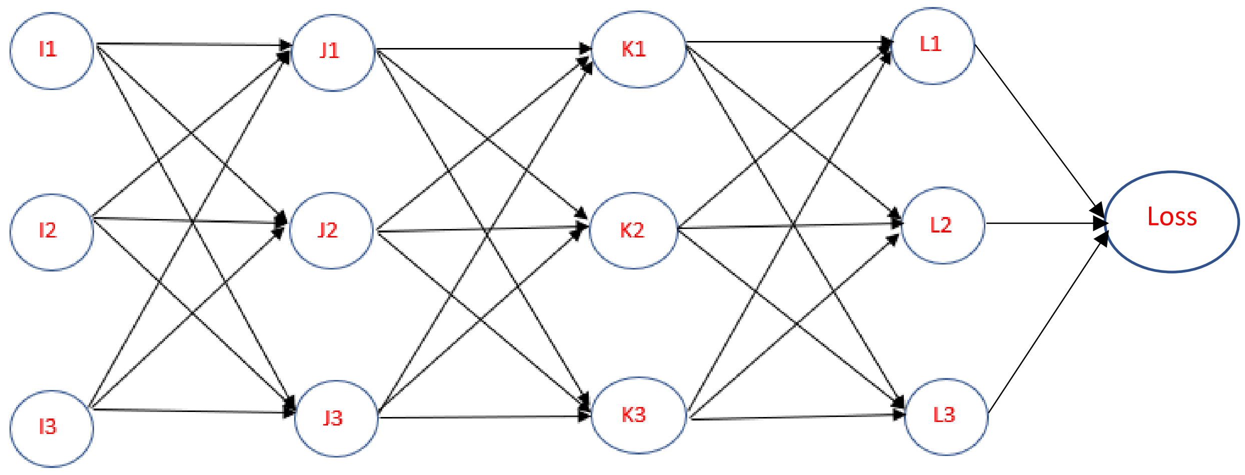


The results from Jax and NumPy are identical, verified.

Part B

Network graph:

(Input) (hidden layer 1) (hidden layer 2) (Output)



Computation graph(vectorized):

$$\begin{tikzcd}
 & W\_1 \arrow[d] & & W\_2 \arrow[d] & & W\_3 \arrow[d] & & t \arrow[d] \\
I \arrow[r] & J_{in} \arrow[r, "Relu"'] & J_{out} \arrow[r] & K_{in} \arrow[r, "Logistic"'] & K_{out} \arrow[r] & L_{in} \arrow[r, "Softmax"'] & L_{out} \arrow[r] & Loss \\
 & b\_1 \arrow[u] & & b\_2 \arrow[u] & & b\_3 \arrow[u] & &
\end{tikzcd}$$

Equations (forward):

$J_{in} = I \cdot {W_1}^T + b\_1$ $J_{out} = Relu(J_{in})$

$K_{in} = J_{out} \cdot {W_2}^T + b\_2$ $K_{out} = Logistic(K_{in})$

$L_{in} = K_{out} \cdot {W_3}^T + b\_3$ $L_{out} = Softmax(L_{in})$

Loss $= \frac{1}{N} \cdot \sum_{i=1}^{3} -t_i log\left(L_{out_i}\right) - (1 - t_i) log(1 - L_{out_i})$

Backpropagation:

$$\overline{Loss} = 1$$

$$\overline{L_{out}} = \overline{Loss}\ \frac{\partial Loss}{\partial L_{out}} = (1)(\frac{-t}{L_{out}} + \frac{1-t}{1-L_{out}})$$

$$\overline{L_{in}} = \overline{L_{out}}\ \frac{\partial \sigma_{Softmax}}{\partial L_{in}} = \overline{L_{out}}\ \sigma(L_{in})(1-\sigma(L_{in}))$$

$$\overline{W\_3} = \overline{L_{in}} \frac{\partial L_{in}}{\partial W\_3} = K_{out}^{T}\overline{L_{in}}$$

$$\overline{b\_3} = \overline{L_{in}} \frac{\partial L_{in}}{\partial b\_3} = \overline{L_{in}}^{T}$$

$$\overline{K_{out}} = \overline{L_{in}}\ \frac{\partial L_{in}}{\partial K_{out}} = \overline{L_{in}}\ \ W\_3^{T}$$

$$\overline{K_{in}} = \overline{K_{out}} \frac{\partial \sigma_{Logistic}}{\partial \overline{K_{out}}} = \overline{K_{out}}\sigma(K_{in})(1-\sigma(K_{in}))$$

$$\overline{W\_2} = \overline{K_{in}} \frac{\partial K_{in}}{\partial W\_2} = J_{out}^{T}\overline{K_{in}}$$

$$\overline{b\_2} = \overline{K_{in}} \frac{\partial K_{in}}{\partial b\_2} = \overline{K_{in}}^{T}$$

$$\overline{J_{out}} = \overline{K_{in}}\ \frac{\partial K_{in}}{\partial J_{out}} = \overline{K_{in}}\ \ W\_2^{T}$$

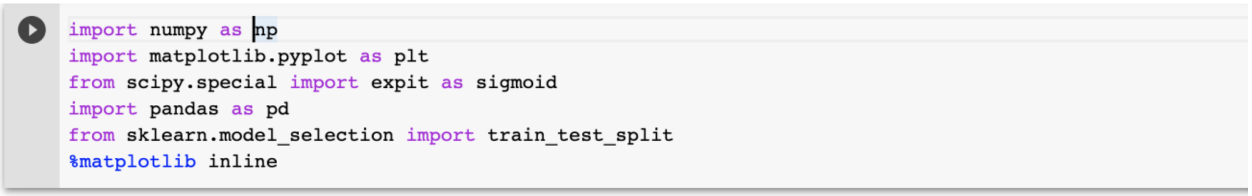
$$\overline{J_{in}} = \overline{J_{out}} \frac{\partial \sigma_{Relu}}{\partial \overline{J_{in}}} = \begin{cases} \overline{J_{out}} & , if\ \overline{J_{in}} > 0 \\ 0 & , if\ \ \overline{J_{in}} \leq 0 \end{cases}$$

$$\overline{W\_1} = \overline{J_{in}} \frac{\partial J_{in}}{\partial W\_1} = \overline{J_{in}}^{T}\ I$$

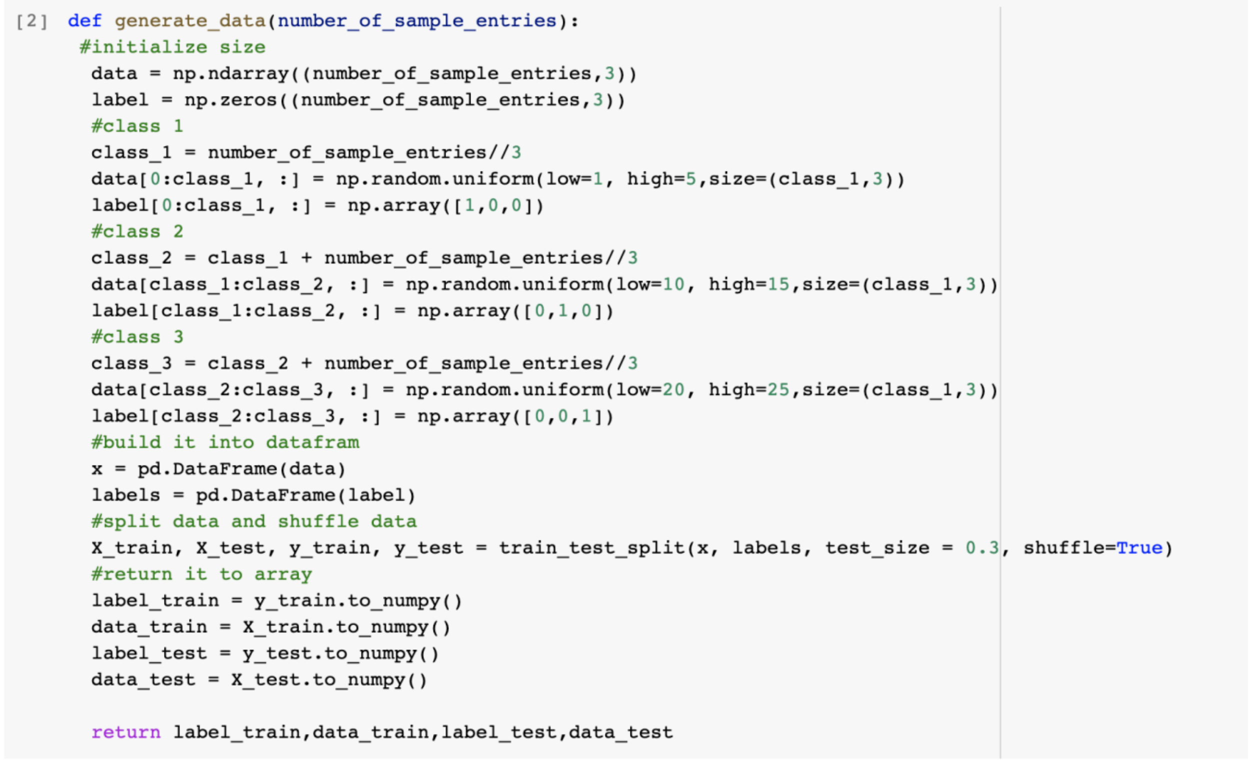
$$\overline{b\_1} = \overline{J_{in}}^{T}$$

Code for Multi-layer perceptron Part B:

(Python version)



```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import expit as sigmoid
import pandas as pd
from sklearn.model_selection import train_test_split
%matplotlib inline
```

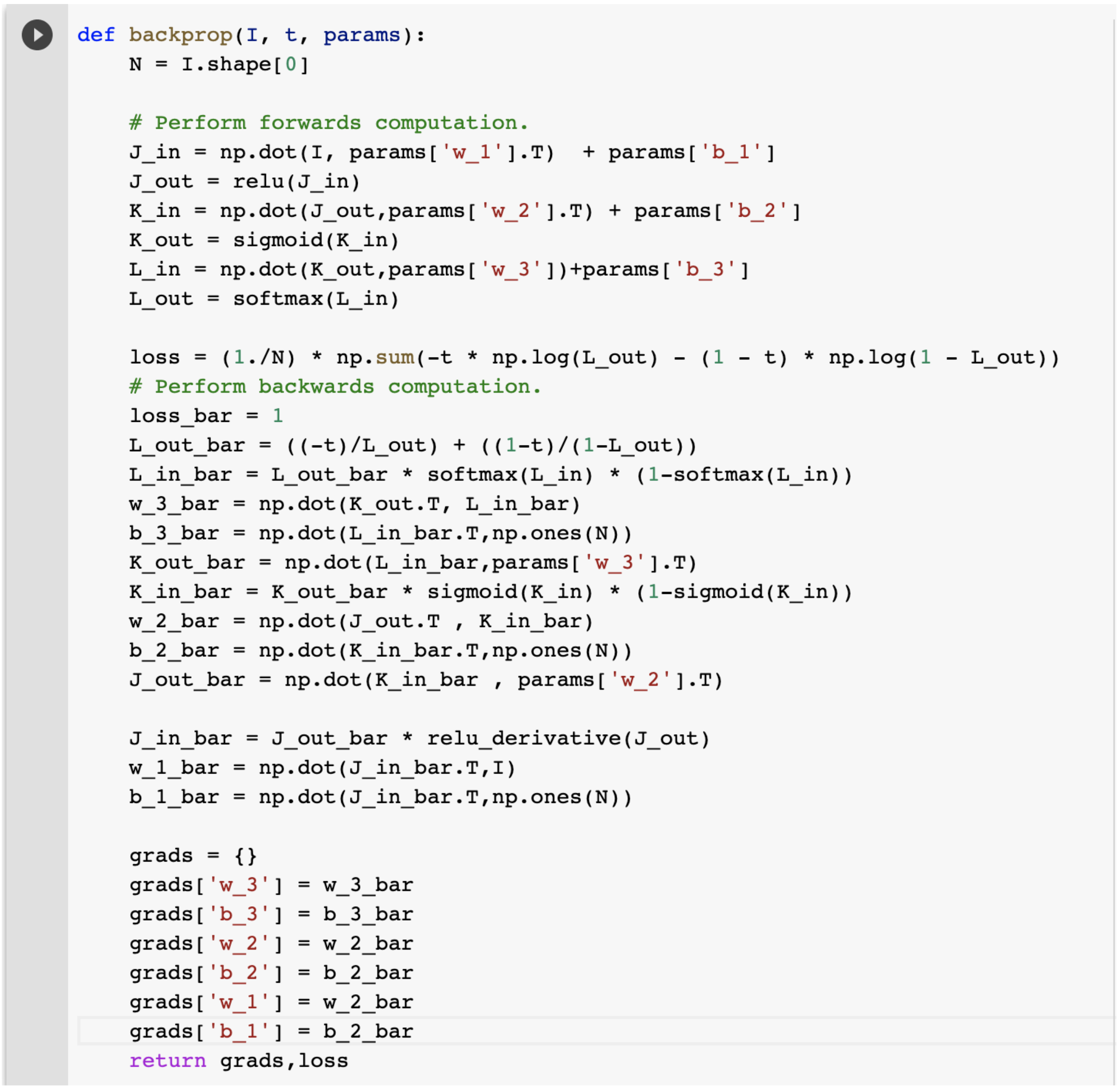


```python
def generate_data(number_of_sample_entries):
 #initialize size
  data = np.ndarray((number_of_sample_entries,3))
  label = np.zeros((number_of_sample_entries,3))
  #class 1
  class_1 = number_of_sample_entries//3
  data[0:class_1, :] = np.random.uniform(low=1, high=5,size=(class_1,3))
  label[0:class_1, :] = np.array([1,0,0])
  #class 2
  class_2 = class_1 + number_of_sample_entries//3
  data[class_1:class_2, :] = np.random.uniform(low=10, high=15,size=(class_1,3))
  label[class_1:class_2, :] = np.array([0,1,0])
  #class 3
  class_3 = class_2 + number_of_sample_entries//3
  data[class_2:class_3, :] = np.random.uniform(low=20, high=25,size=(class_1,3))
  label[class_2:class_3, :] = np.array([0,0,1])
  #build it into datafram
  x = pd.DataFrame(data)
  labels = pd.DataFrame(label)
  #split data and shuffle data
  X_train, X_test, y_train, y_test = train_test_split(x, labels, test_size = 0.3, shuffle=True)
  #return it to array
  label_train = y_train.to_numpy()
  data_train = X_train.to_numpy()
  label_test = y_test.to_numpy()
  data_test = X_test.to_numpy()

  return label_train,data_train,label_test,data_test
```

```
[3] label_train,data_train,label_test,data_test = generate_data(300)
```

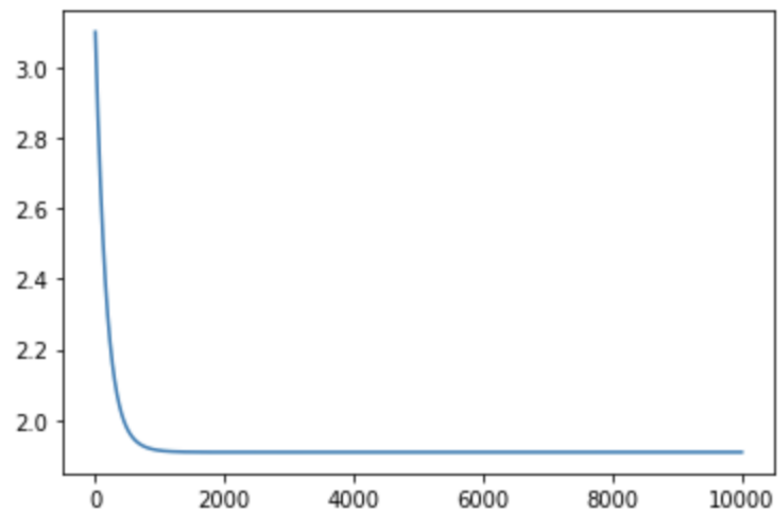
```
[4] def relu(X):
       return np.maximum(X,0)



    def relu_derivative(x):
      return np.where(x>0, 1, 0)



    def softmax(x):
      list_res=[]
      for i in range(len(x)):
        res = np.exp(x[i])/np.sum(np.exp(x[i]))
        list_res.append(res)
      return np.array(list_res)
```

```
[5] # Initialize our neural network parameters.
    params = {}
    params['w_1'] = np.random.randn(3, 3)
    params['b_1'] = np.zeros(3)

    params['w_2'] = np.random.randn(3, 3)
    params['b_2'] = np.zeros(3)

    params['w_3'] = np.random.randn(3, 3)
    params['b_3'] = np.zeros(3)
```


```
def backprop(I, t, params):
    N = I.shape[0]

    # Perform forwards computation.
    J_in = np.dot(I, params['w_1'].T)  + params['b_1']
    J_out = relu(J_in)
    K_in = np.dot(J_out,params['w_2'].T) + params['b_2']
    K_out = sigmoid(K_in)
    L_in = np.dot(K_out,params['w_3'])+params['b_3']
    L_out = softmax(L_in)

    loss = (1./N) * np.sum(-t * np.log(L_out) - (1 - t) * np.log(1 - L_out))
    # Perform backwards computation.
    loss_bar = 1
    L_out_bar = ((-t)/L_out) + ((1-t)/(1-L_out))
    L_in_bar = L_out_bar * softmax(L_in) * (1-softmax(L_in))
    w_3_bar = np.dot(K_out.T, L_in_bar)
    b_3_bar = np.dot(L_in_bar.T,np.ones(N))
    K_out_bar = np.dot(L_in_bar,params['w_3'].T)
    K_in_bar = K_out_bar * sigmoid(K_in) * (1-sigmoid(K_in))
    w_2_bar = np.dot(J_out.T , K_in_bar)
    b_2_bar = np.dot(K_in_bar.T,np.ones(N))
    J_out_bar = np.dot(K_in_bar , params['w_2'].T)

    J_in_bar = J_out_bar * relu_derivative(J_out)
    w_1_bar = np.dot(J_in_bar.T,I)
    b_1_bar = np.dot(J_in_bar.T,np.ones(N))

    grads = {}
    grads['w_3'] = w_3_bar
    grads['b_3'] = b_3_bar
    grads['w_2'] = w_2_bar
    grads['b_2'] = b_2_bar
    grads['w_1'] = w_2_bar
    grads['b_1'] = b_2_bar
    return grads,loss
```

```
[9] alpha = 0.00002
    cost_list=[]
    number_steps = 10000
    for step in range(number_steps):
        grads, loss = backprop(data_train, label_train, params)
        for k in params:
            params[k] -= alpha * grads[k]
        cost_list.append(loss)
```

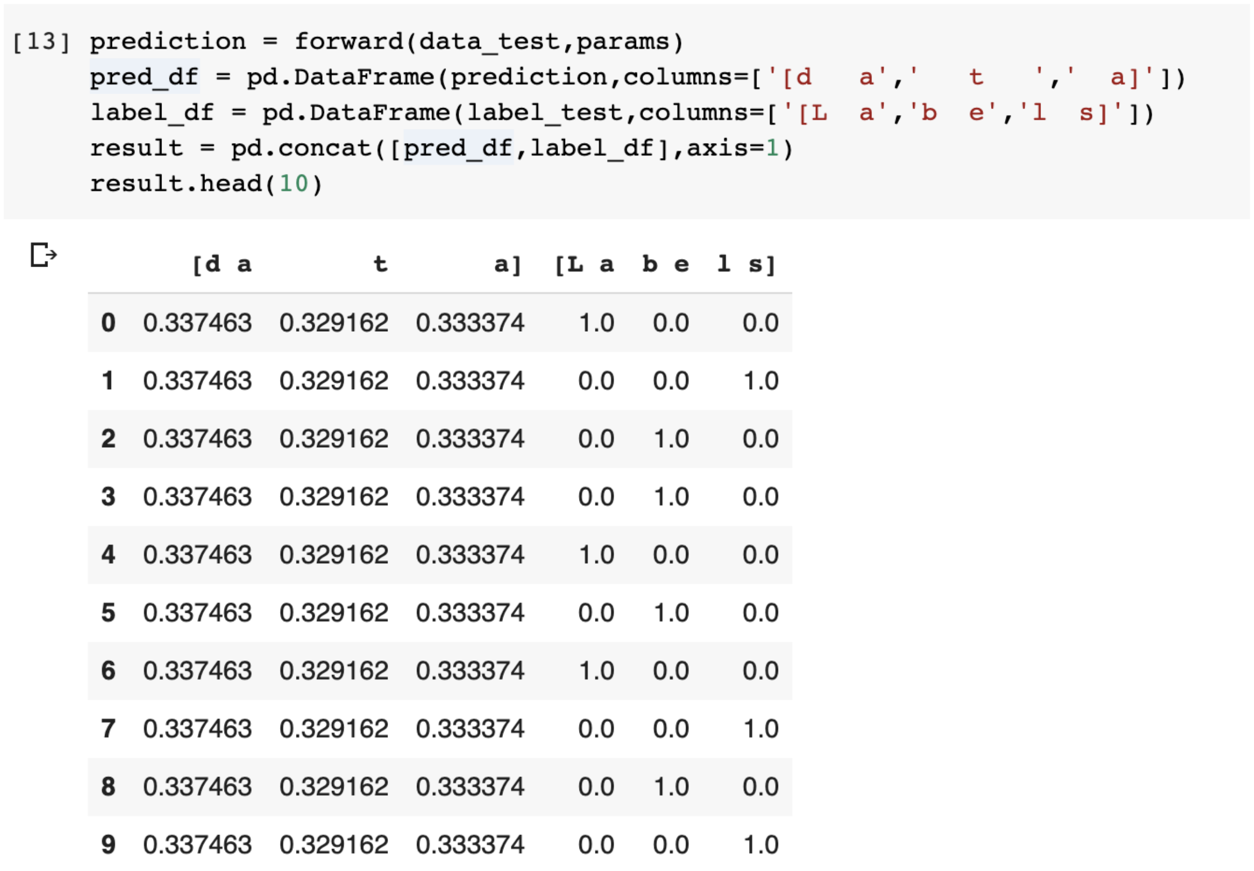
```
[10] plt.plot(cost_list)
     plt.show()
```



```
[11] params#updated result
```

```
{'b_1': array([-0.21559943, -0.23552499, -0.35982087]),
 'b_2': array([-0.21559943, -0.23552499, -0.35982087]),
 'b_3': array([ 0.45218024, -0.86313228,  0.41095204]),
 'w_1': array([[-0.63282052, -0.95398609, -0.04290488],
        [ 0.82282193, -0.53411616, -1.65824656],
        [-1.46243582, -0.28651117, -0.46044691]]),
 'w_2': array([[-0.42638379, -0.76065386, -0.73199923],
        [-1.042936  , -0.82764766,  0.94791957],
        [-1.65171804, -0.27136179,  1.72624633]]),
 'w_3': array([[-0.12064187,  0.71925989,  0.09653366],
        [-0.42923501,  0.38809017, -0.45781506],
        [-1.2144331 ,  0.13541762, -1.34892259]])}
```

```
[12] def forward(I, params):
         N = I.shape[0]

         # Perform forwards computation.
         J_in = np.dot(I, params['w_1'].T)  + params['b_1']
         J_out = relu(J_in)
         K_in = np.dot(J_out,params['w_2'].T) + params['b_2']
         K_out = sigmoid(K_in)
         L_in = np.dot(K_out,params['w_3'])+params['b_3']
         L_out = softmax(L_in)
         return L_out
```



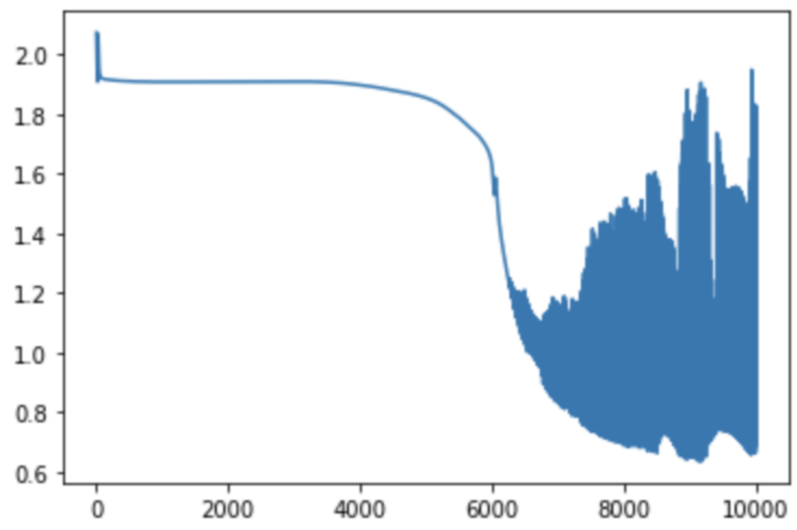
```
[13] prediction = forward(data_test,params)
     pred_df = pd.DataFrame(prediction,columns=['[d   a','   t   ','  a]'])
     label_df = pd.DataFrame(label_test,columns=['[L  a','b  e','l  s]'])
     result = pd.concat([pred_df,label_df],axis=1)
     result.head(10)
```

```
        [d a          t         a]  [L a   b e   l s]
0   0.337463   0.329162   0.333374    1.0   0.0    0.0
1   0.337463   0.329162   0.333374    0.0   0.0    1.0
2   0.337463   0.329162   0.333374    0.0   1.0    0.0
3   0.337463   0.329162   0.333374    0.0   1.0    0.0
4   0.337463   0.329162   0.333374    1.0   0.0    0.0
5   0.337463   0.329162   0.333374    0.0   1.0    0.0
6   0.337463   0.329162   0.333374    1.0   0.0    0.0
7   0.337463   0.329162   0.333374    0.0   0.0    1.0
8   0.337463   0.329162   0.333374    0.0   1.0    0.0
9   0.337463   0.329162   0.333374    0.0   0.0    1.0
```

I have tried to regenerate dataset multiple times. The results above appears most often. However, as we can see the comparison between predicting results and labels, our predictions from neural network are not really good.

Note: By accidently, the random generator gave me a dataset perform really well. (nothing changed but regenerate dataset):

```
[ ]  plt.plot(cost_list)
     plt.show()
```



```
[ ]  params#updated results
```

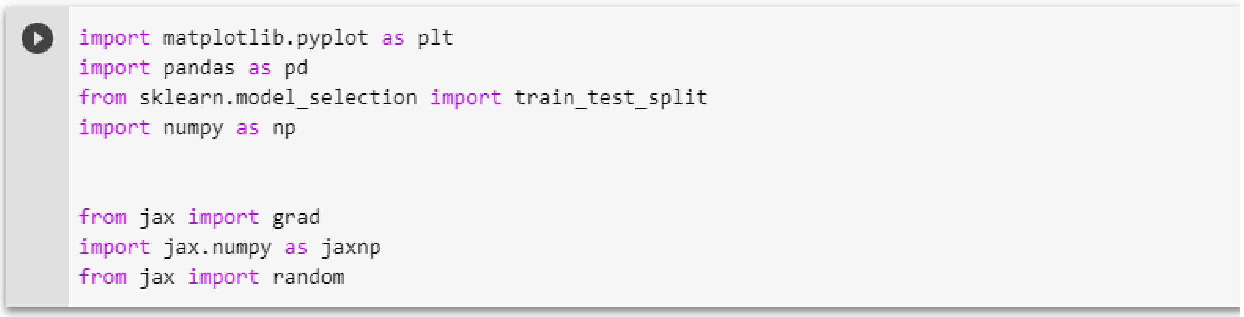
```
{'b_1': array([11.94692475,  0.44241461,  0.35109688]),
 'b_2': array([11.94692475,  0.44241461,  0.35109688]),
 'b_3': array([-2.69551367,  0.97924416,  1.71626951]),
 'w_1': array([[-0.15835763,  0.6173231 ,  0.48571954],
        [-1.84938517, -1.90766899,  3.20121433],
        [-0.99722143, -0.20116152, -0.27684717]]),
 'w_2': array([[-0.55595624,  0.57160051, -1.26515345],
        [-1.61654094, -0.9921629 ,  3.15037534],
        [-1.14159903,  0.10028088,  1.69827081]]),
 'w_3': array([[ 6.90761415,  0.49501025, -7.29705402],
        [ 0.12152897,  0.07805653,  0.62909393],
        [ 1.50893798,  0.29002243, -1.17686828]])}
[ ] def forward(I, params):
        N = I.shape[0]

        # Perform forwards computation.
        J_in = np.dot(I, params['w_1'].T)  + params['b_1']
        J_out = relu(J_in)
        K_in = np.dot(J_out,params['w_2'].T) + params['b_2']
        K_out = sigmoid(K_in)
        L_in = np.dot(K_out,params['w_3'])+params['b_3']
        L_out = softmax(L_in)
        return L_out
```

```
[ ] prediction = forward(data_test,params)
    pred_df = pd.DataFrame(prediction,columns=['[d   a','   t   ','  a]'])
    label_df = pd.DataFrame(label_test,columns=['[L  a','b  e','l  s]'])
    result = pd.concat([pred_df,label_df],axis=1)
    result.head(10)
```

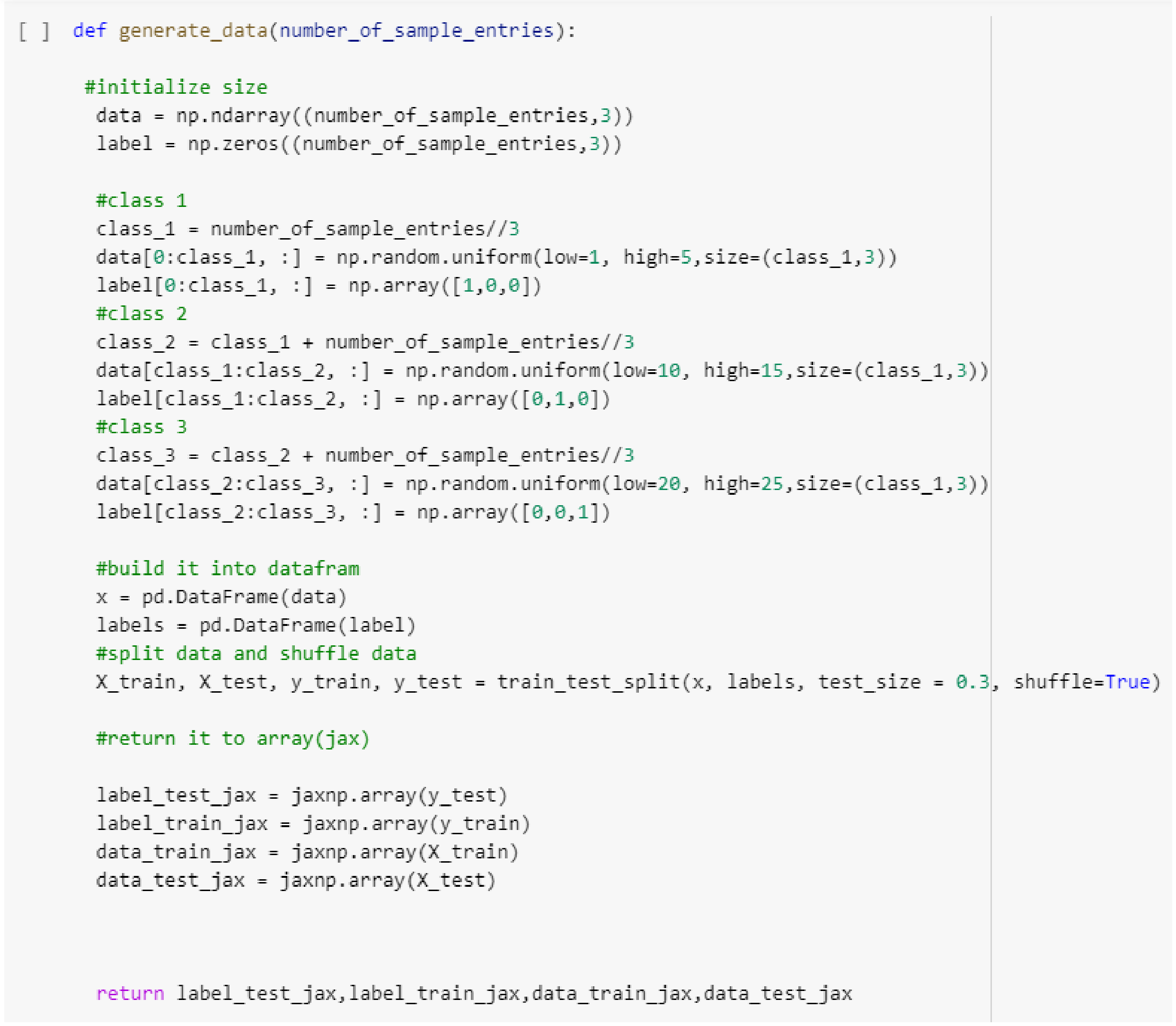
	[d a	t	a]	[L a	b e	l s]
0	0.289326	0.666513	0.044161	0.0	1.0	0.0
1	0.033265	0.556520	0.410214	0.0	1.0	0.0
2	0.008425	0.325983	0.665592	0.0	0.0	1.0
3	0.114386	0.722537	0.163077	0.0	1.0	0.0
4	0.008279	0.323463	0.668258	0.0	0.0	1.0
5	0.034964	0.565297	0.399739	0.0	1.0	0.0
6	0.932174	0.067759	0.000067	1.0	0.0	0.0
7	0.008214	0.322319	0.669467	0.0	0.0	1.0
8	0.931431	0.068499	0.000069	1.0	0.0	0.0
9	0.026399	0.515363	0.458238	0.0	1.0	0.0

As we can see this dataset gives good predictions, but with a weird cost plot.

(Jax version)



```python
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split
import numpy as np


from jax import grad
import jax.numpy as jaxnp
from jax import random
```



```python
[ ] def generate_data(number_of_sample_entries):

     #initialize size
      data = np.ndarray((number_of_sample_entries,3))
      label = np.zeros((number_of_sample_entries,3))

      #class 1
      class_1 = number_of_sample_entries//3
      data[0:class_1, :] = np.random.uniform(low=1, high=5,size=(class_1,3))
      label[0:class_1, :] = np.array([1,0,0])
      #class 2
      class_2 = class_1 + number_of_sample_entries//3
      data[class_1:class_2, :] = np.random.uniform(low=10, high=15,size=(class_1,3))
      label[class_1:class_2, :] = np.array([0,1,0])
      #class 3
      class_3 = class_2 + number_of_sample_entries//3
      data[class_2:class_3, :] = np.random.uniform(low=20, high=25,size=(class_1,3))
      label[class_2:class_3, :] = np.array([0,0,1])

      #build it into datafram
      x = pd.DataFrame(data)
      labels = pd.DataFrame(label)
      #split data and shuffle data
      X_train, X_test, y_train, y_test = train_test_split(x, labels, test_size = 0.3, shuffle=True)

      #return it to array(jax)

      label_test_jax = jaxnp.array(y_test)
      label_train_jax = jaxnp.array(y_train)
      data_train_jax = jaxnp.array(X_train)
      data_test_jax = jaxnp.array(X_test)



      return label_test_jax,label_train_jax,data_train_jax,data_test_jax
```

```
[ ]  label_test,label_train,data_train,data_test = generate_data(300)
```

```
[ ]  def relu(X):
        return jaxnp.maximum(X,0)



     def softmax(x):
       list_res=[]
       for i in range(len(x)):
         res = jaxnp.exp(x[i])/jaxnp.sum(jaxnp.exp(x[i]))
         list_res.append(res)
       return jaxnp.array(list_res)

     # def softmax(x):
     #   return (jaxnp.exp(x)) / jaxnp.sum(jaxnp.exp(x))

     def sigmoid(x):
       return 1/(1 + jaxnp.exp(-x))
```

```
[ ]  # Initialize our neural network parameters.
     key = random.PRNGKey(0)
     key, W_key, b_key = random.split(key, 3)

     params = {}
     params['w_1'] = random.normal(W_key,(3,3))
     params['b_1'] = random.normal(b_key,(3,))
     params['w_2'] = random.normal(W_key,(3,3))
     params['b_2'] = random.normal(b_key,(3,))
     params['w_3'] = random.normal(W_key,(3,3))
     params['b_3'] = random.normal(b_key,(3,))
```

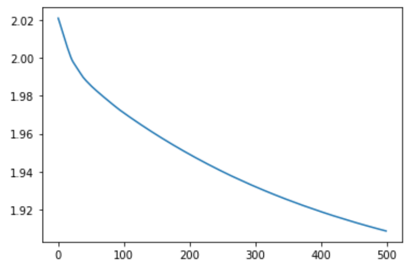
```
[ ]  def loss_function (data, t, w_1, b_1, w_2, b_2, w_3, b_3):
       J_in = jaxnp.dot(data, jaxnp.transpose(w_1))  + b_1
       J_out = relu(J_in)
       K_in = jaxnp.dot(J_out,jaxnp.transpose(w_2)) + b_2
       K_out = sigmoid(K_in)
       L_in = jaxnp.dot(K_out,jaxnp.transpose(w_3)) + b_3
       L_out = softmax(L_in)

       loss = (1./data.shape[0]) * jaxnp.sum(-t * jaxnp.log(L_out) - (1 - t) * jaxnp.log(1 - L_out))
       return loss
```

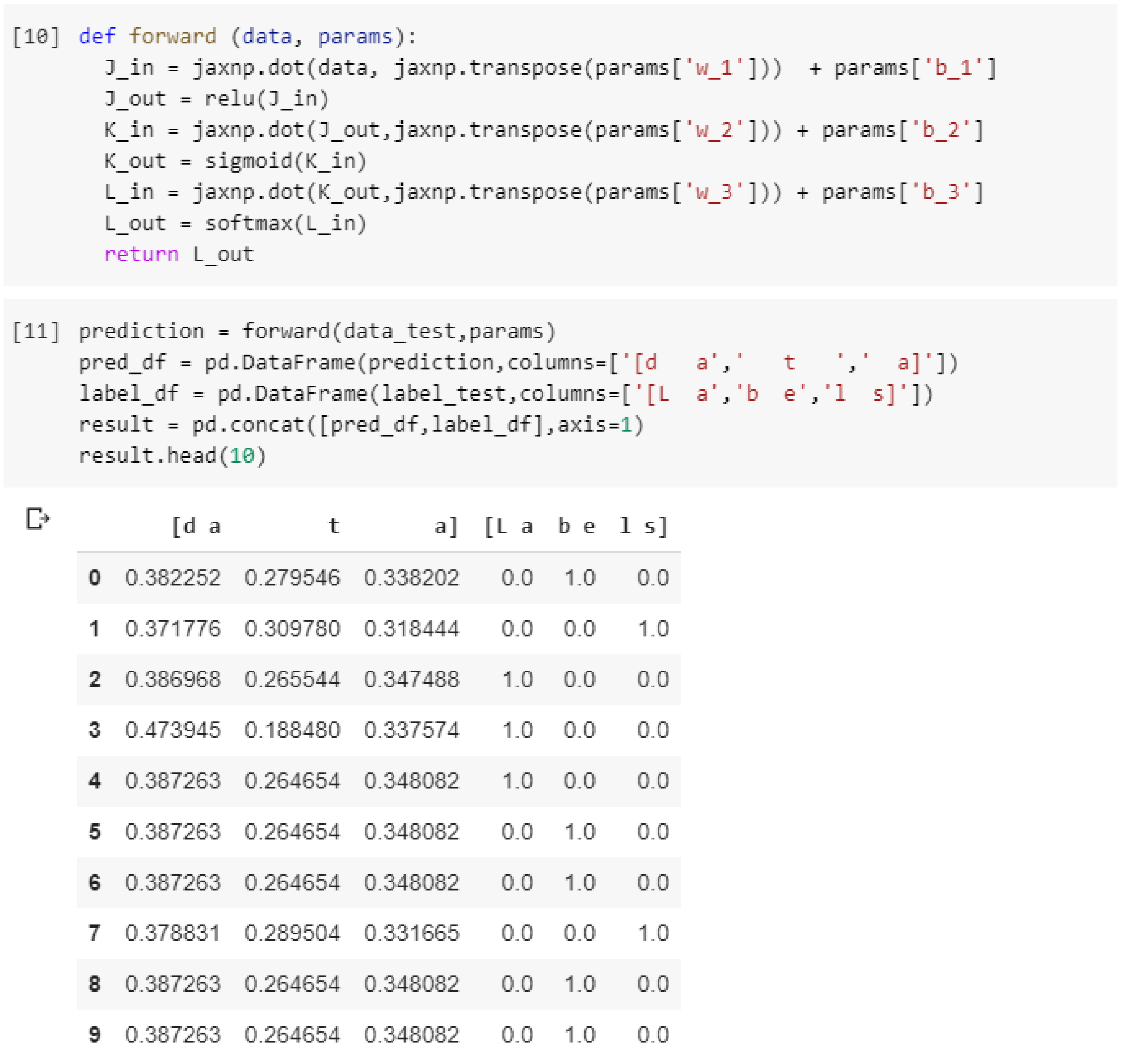
```
[ ] dw_1 = grad(loss_function,2)
    db_1 = grad(loss_function,3)

    dw_2 = grad(loss_function,4)
    db_2 = grad(loss_function,5)

    dw_3 = grad(loss_function,6)
    db_3 = grad(loss_function,7)
```

```
[8] alpha = 0.002
    iterations = 500
    cost_list = []
    for i in range(iterations):
        loss = loss_function(data_train, label_train, params['w_1'],params['b_1'],params['w_2'],params['b_2'],params['w_3'],params['b_3'])

        #weights update
        params['w_1'] -= alpha * dw_1(data_train, label_train, params['w_1'],params['b_1'],params['w_2'],params['b_2'],params['w_3'],params['b_3'])
        params['w_2'] -= alpha * dw_2(data_train, label_train, params['w_1'],params['b_1'],params['w_2'],params['b_2'],params['w_3'],params['b_3'])
        params['w_3'] -= alpha * dw_3(data_train, label_train, params['w_1'],params['b_1'],params['w_2'],params['b_2'],params['w_3'],params['b_3'])
        #bias update
        params['b_1'] -= alpha * db_1(data_train, label_train, params['w_1'],params['b_1'],params['w_2'],params['b_2'],params['w_3'],params['b_3'])
        params['b_2'] -= alpha * db_2(data_train, label_train, params['w_1'],params['b_1'],params['w_2'],params['b_2'],params['w_3'],params['b_3'])
        params['b_3'] -= alpha * db_3(data_train, label_train, params['w_1'],params['b_1'],params['w_2'],params['b_2'],params['w_3'],params['b_3'])

        cost_list.append(loss)
```

```
'loss: 1.9086766 iterations: 499'
```

```
[9] plt.plot(cost_list)
    plt.show()
```





```
[10] def forward (data, params):
       J_in = jaxnp.dot(data, jaxnp.transpose(params['w_1']))  + params['b_1']
       J_out = relu(J_in)
       K_in = jaxnp.dot(J_out,jaxnp.transpose(params['w_2'])) + params['b_2']
       K_out = sigmoid(K_in)
       L_in = jaxnp.dot(K_out,jaxnp.transpose(params['w_3'])) + params['b_3']
       L_out = softmax(L_in)
       return L_out
```

```
[11] prediction = forward(data_test,params)
     pred_df = pd.DataFrame(prediction,columns=['[d   a','   t   ','   a]'])
     label_df = pd.DataFrame(label_test,columns=['[L  a','b  e','l  s]'])
     result = pd.concat([pred_df,label_df],axis=1)
     result.head(10)
```

```
       [d a         t          a]  [L a  b e  l s]
0  0.382252  0.279546  0.338202   0.0  1.0   0.0
1  0.371776  0.309780  0.318444   0.0  0.0   1.0
2  0.386968  0.265544  0.347488   1.0  0.0   0.0
3  0.473945  0.188480  0.337574   1.0  0.0   0.0
4  0.387263  0.264654  0.348082   1.0  0.0   0.0
5  0.387263  0.264654  0.348082   0.0  1.0   0.0
6  0.387263  0.264654  0.348082   0.0  1.0   0.0
7  0.378831  0.289504  0.331665   0.0  0.0   1.0
8  0.387263  0.264654  0.348082   0.0  1.0   0.0
9  0.387263  0.264654  0.348082   0.0  1.0   0.0
```

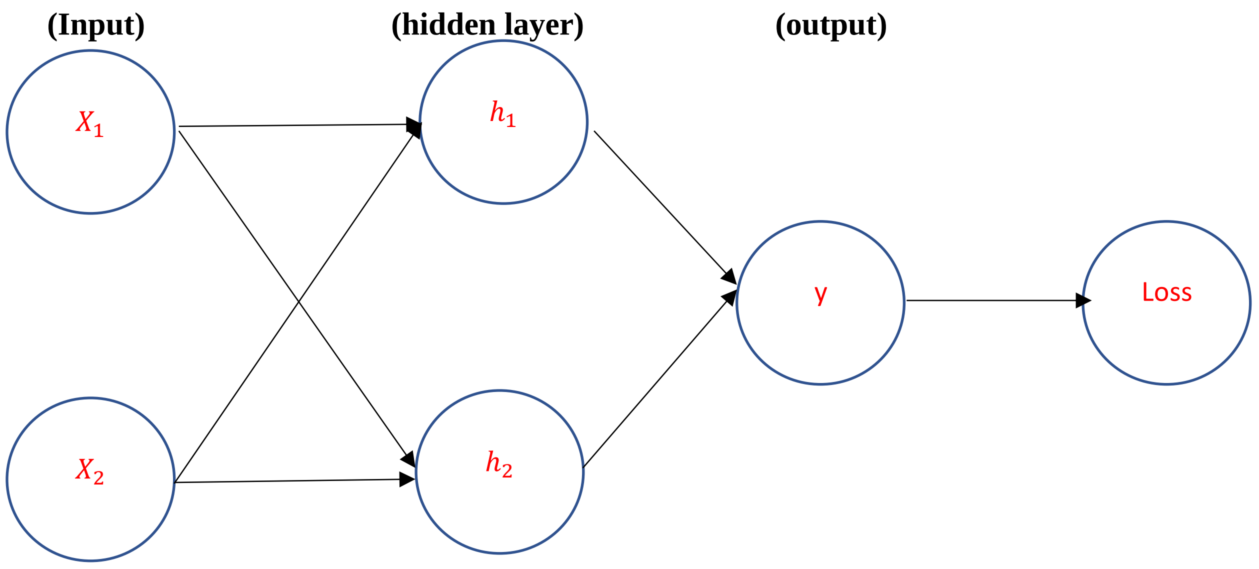
<u>Conclusion for this question:</u>

The results from Jax are basically same compare to NumPy version:

1. **Dataset and all parameters are randomly generated. Hard to get exactly same results. In majority time, both methods give the prediction for three classes equally result. Around [0.33,0.33,0.33].**
2. **Jax code's running time significantly longer than NumPy version, I assume if I leave it for enough long training time until the loss reaches to global min. In that case, the prediction will be robust and similar with that best NumPy result.**

Part C: XOR NN

<u>Network graph:</u>



Computation Graph:

$$\begin{tikzcd}
& \mathbf{W\_1} \arrow[d] & & \mathbf{W\_2} \arrow[d] & & \mathbf{t} \arrow[d] \\
\mathbf{X} \arrow[r] & \mathbf{h_{in}} \arrow[r, "\text{Logistic}"'] & \mathbf{h_{out}} \arrow[r] & \boldsymbol{y}_{\mathbf{in}} \arrow[r, "\text{Logistic}"'] & \mathbf{y_{out}} \arrow[r] & \textbf{Cross Entropy Loss} \\
& \mathbf{b\_1} \arrow[u] & & \mathbf{b\_2} \arrow[u] & &
\end{tikzcd}$$

Equations (forward):

$h_{in} = X \cdot {W_1}^T + b\_1$ $\qquad$ $h_{out} = Logistic(h_{in})$

$y_{in} = h_{out} \cdot {W_2}^T + b\_2$ $\qquad$ $y_{out} = Logistic(y_{in})$

Loss $= \frac{1}{N} \cdot \sum_{i=1}^{2} -t_i log(y_i) - (1 - t_i) log(1 - y_i)$

Backpropagation:

$$\overline{Loss} = 1$$

$$\overline{y_{out}} = \overline{Loss}\ \frac{\partial Loss}{\partial y_{out}} = (1)(\frac{-t}{y_{out}} + \frac{1-t}{1-y_{out}})$$

$$\overline{y_{in}} = \overline{y_{out}}\ \frac{\partial \sigma_{logistic}}{\partial y_{in}} = \overline{y_{out}}\ \sigma(y_{in})(1-\sigma(y_{in}))$$

$$\overline{W\_2} = \overline{y_{in}} \frac{\partial y_{in}}{\partial W\_2} = \overline{y_{in}} h\_out^{T}$$

$$\overline{b\_2} = \overline{y_{in}} \frac{\partial y_{in}}{\partial b\_2} = \overline{y_{in}}$$

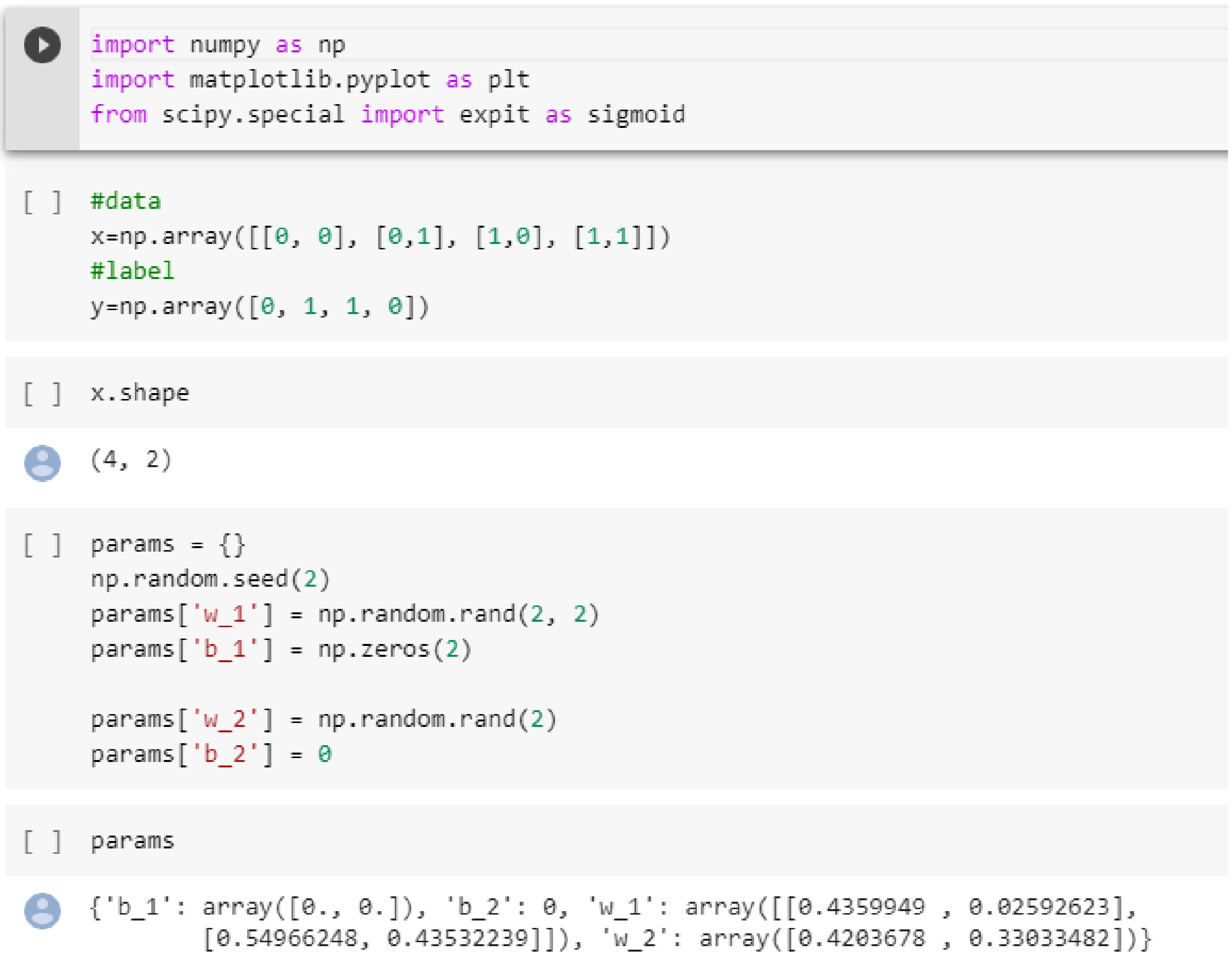
$$\overline{h_{out}} = \overline{y_{in}}\ \frac{\partial y_{in}}{\partial h_{out}} = \overline{y_{in}}\ \ W\_2^{T}$$

$$\overline{h_{in}} = \overline{h_{out}} \frac{\partial \sigma_{Logistic}}{\partial \overline{h_{out}}} = \overline{h_{out}} \sigma(h_{in})(1-\sigma(h_{in}))$$

$$\overline{W\_1} = \overline{h_{in}} \frac{\partial h_{in}}{\partial W\_1} = (h\_in\ )^{\ T} X$$

$$\overline{b\_1} = \overline{h_{in}}^{T}$$

Code for XOR:



```
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import expit as sigmoid
```

```
[ ] #data
    x=np.array([[0, 0], [0,1], [1,0], [1,1]])
    #label
    y=np.array([0, 1, 1, 0])
```

```
[ ] x.shape
```

```
(4, 2)
```

```
[ ] params = {}
    np.random.seed(2)
    params['w_1'] = np.random.rand(2, 2)
    params['b_1'] = np.zeros(2)

    params['w_2'] = np.random.rand(2)
    params['b_2'] = 0
```

```
[ ] params
```

```
{'b_1': array([0., 0.]), 'b_2': 0, 'w_1': array([[0.4359949 , 0.02592623],
        [0.54966248, 0.43532239]]), 'w_2': array([0.4203678 , 0.33033482])}
```

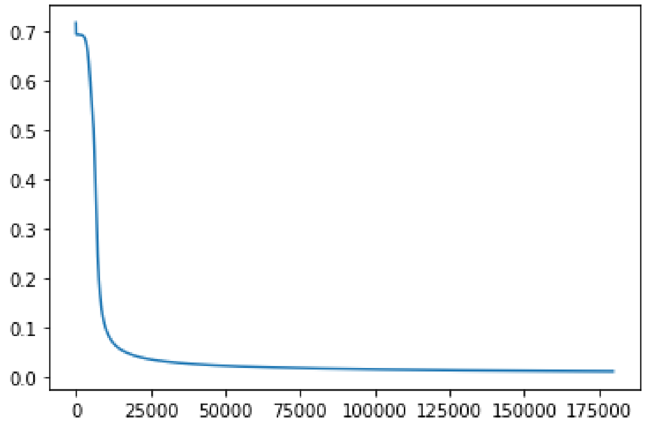
```
[ ]  def backprop(x, t, p):

         N = x.shape[0]

         # forward pass
         h_in = np.dot(x, p['w_1'].T) + p['b_1']
         h_out = sigmoid(h_in)
         y_in = np.dot(h_out, p['w_2'].T) + p['b_2']
         y_out = sigmoid(y_in)

         # loss
         loss = (1./N) * np.sum(-t * np.log(y_out) - (1 - t) * np.log(1 - y_out))

         # backprop
         l_bar = 1
         yout_bar = (1./N) * (y_out - t)
         yin_bar = yout_bar * sigmoid(y_in) * (1 - sigmoid(y_in))
         b2_bar = np.dot(yin_bar.T, np.ones(N))
         w2_bar = np.dot(yin_bar.T, h_out)
         hout_bar = np.outer(yin_bar, p['w_2'])
         hin_bar = hout_bar * sigmoid(h_in) * (1 - sigmoid(h_in))
         b1_bar = np.dot(hin_bar.T, np.ones(N))
         w1_bar = np.dot(hin_bar.T, x)

         # Wrap our gradients in a dictionary.
         grads = {}
         grads['w_1'] = w1_bar
         grads['w_2'] = w2_bar
         grads['b_1'] = b1_bar
         grads['b_2'] = b2_bar

         return grads, loss
```

updating parameters

```
iterations = 0
loss = 1
cost_list=[]
alpha = 0.3
while loss > 0.01:
    iterations+=1
    grads, loss = backprop(x, y, params)
    for k in params:
        params[k] -= alpha * grads[k]
    cost_list.append(loss)
```

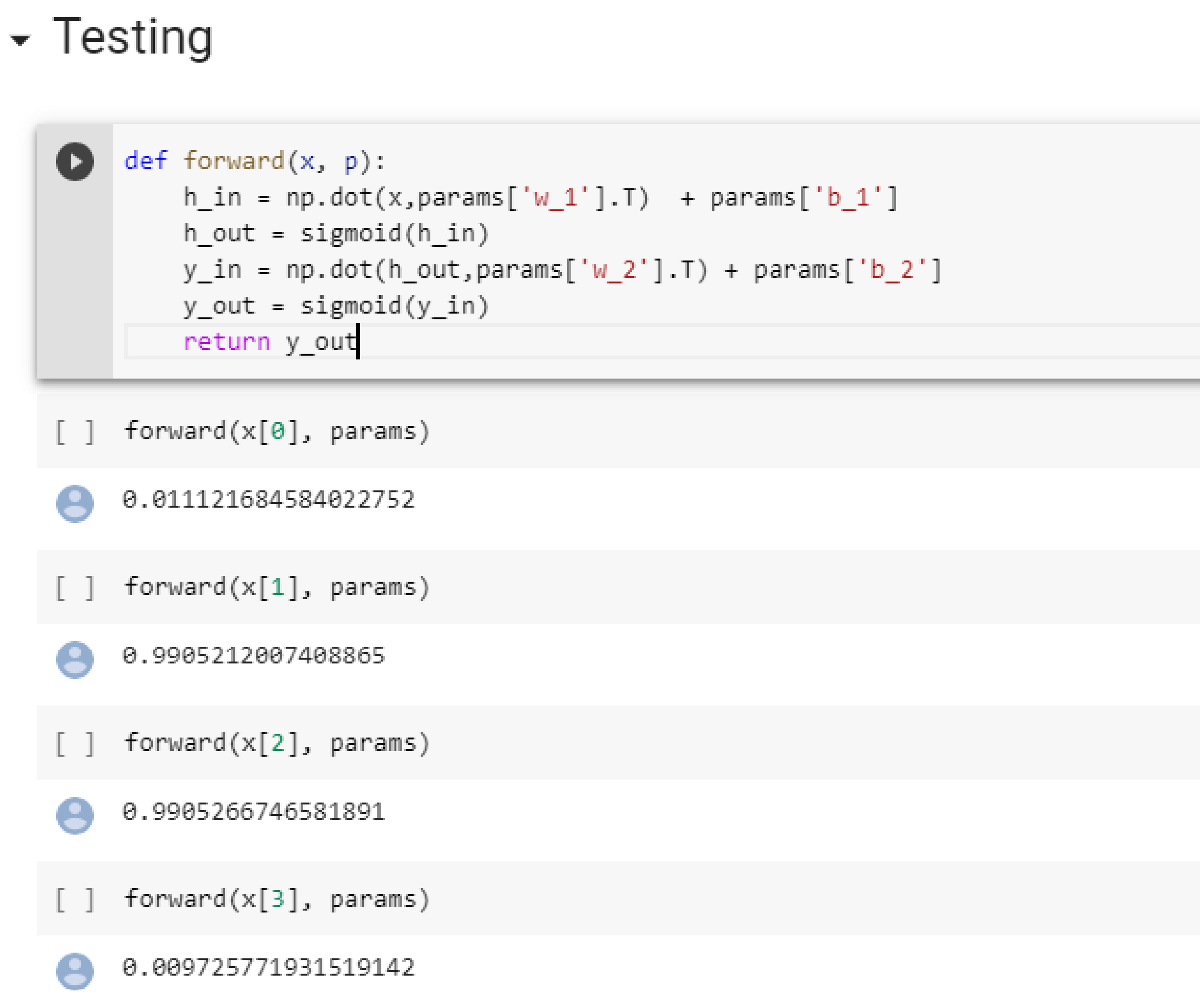
```
[ ] plt.plot(cost_list)
```

```
[<matplotlib.lines.Line2D at 0x7f7124ad5240>]
```



```
[ ] params
```

```
{'b_1': array([-7.5905812 , -3.05395555]),
 'b_2': -4.960001920624812,
 'w_1': array([[4.95096294, 4.94874687],
        [6.78073753, 6.77199414]]),
 'w_2': array([-11.29550268,  10.61182277])}
```



Testing

```
def forward(x, p):
    h_in = np.dot(x,params['w_1'].T)  + params['b_1']
    h_out = sigmoid(h_in)
    y_in = np.dot(h_out,params['w_2'].T) + params['b_2']
    y_out = sigmoid(y_in)
    return y_out
```

```
forward(x[0], params)
```

```
0.011121684584022752
```

```
forward(x[1], params)
```

```
0.9905212007408865
```

```
forward(x[2], params)
```

```
0.9905266746581891
```

```
forward(x[3], params)
```

```
0.009725771931519142
```

The prediction results are really good.