NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

LAB FILE

Course: Artificial Intelligence in IoT

Course Code: EIECE02

Submitted By:
Vineet
2020UEI2808

INDEX

S. No	TITLE	SIGN.
1.	Design an artificial neuron which takes a three-dimensional data as input and uses sigmoidal function as its activation function.	
2.	To train a linear neuron regression model to map data of the given nature using Stochastic Gradient Descent Algorithm.	
3.		
4.		
5.		
6.		
7.		

Experiment-1

Aim: - Design an artificial neuron which takes a three-dimensional data as input and uses sigmoidal function as its activation function.

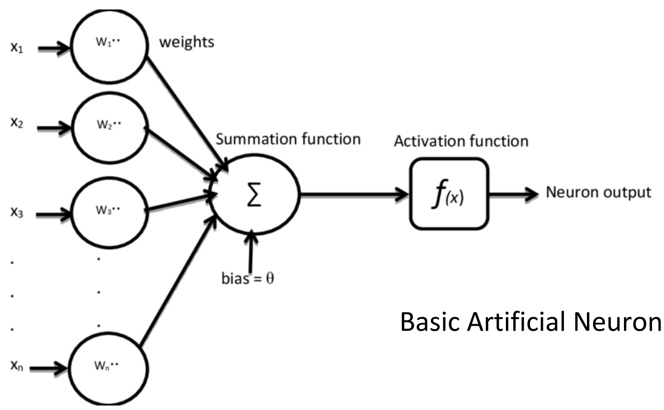
Apparatus: -

- Laptop Configuration
  - MacBook Air M1
  - 8GB RAM
  - 8-Core CPU
  - 7-Core GPU
- Libraries Used: -
  - TensorFlow 1.40
  - NumPy
  - PyLab
  - Matplotlib
- Coding Environment: - Python 3.7.

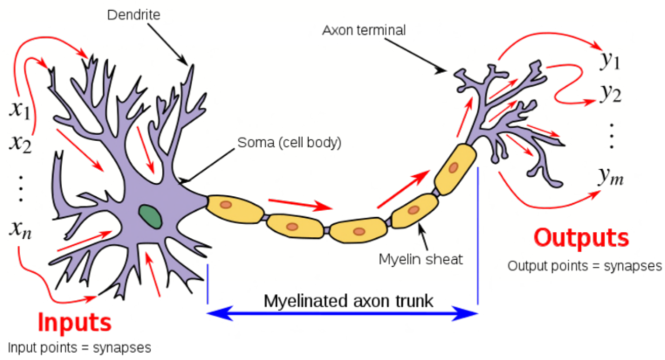
Theory: -

- **Artificial Neuron**

  An artificial neuron is the basic unit of neural network



Basic Artificial Neuron



The basic element of Artificial Neuron

i. A set of input signal. The input is a vector.

$$X = [x_1, x_2, x_3, \dots \dots \dots x_n]^T.$$

where n is the number or the dimension of input signals. Inputs are also referred to as features. input are connected to neuron by a synaptic connection whose strength are represented by their weight. the weight vector W or, where W_i is a synaptic weight connecting i^{th} input to the neuron.

$$W = [w_1, w_2, w_3, \dots \dots \dots w_n]^T.$$

The total synaptic input u to the neuron is given by the sum of the products of input and their corresponding connecting weight minus threshold The total synaptic input to the neuron

$$U = x_1w_1 + x_2w_2 + x_3w_3 + \dots \dots \dots + x_nw_n + (-1)\theta$$

$$U = \sum_{i=1}^{n} x_iw_i - \theta$$

$$U = [x_1 \quad x_2 \quad x_3 \cdots \quad \cdots \quad x_n]\begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ \vdots \\ w_n \end{bmatrix} - \theta$$

$$U = X^TW - \theta$$

The activation function f relates synaptic input to the activation function of neuron.

$f(u)$ denotes activation function of the neuron

So, the output will be

$$\boldsymbol{y} = \boldsymbol{f}\,(\boldsymbol{u})$$

Python CODE: -

```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
#building computational graph
W=tf.Variable([2.5,-0.2,1.0],tf.float32)
b=tf.Variable([-0.5],tf.float32)
x=tf.placeholder(tf.float32)
#placeholder makes x to expect some input
#dot product of
u=tf.tensordot(W,x,axes=1)+b
y=0.8/1+tf.exp(-1.2*u)
#Evaluating computational graph
sess=tf.Session()
init=tf.global_variables_initializer()
#as name suggests
sess.run(init)
u,y=sess.run([u,y],{x:[0.8,2.0,-0.5]})
print(u,y)
```

OUTPUT: -

```
[0.6] [1.2867522]
```

Precautions: -

- Design an artificial neur
- on which takes a three-dimensional
- data as input and uses sigmoidal function
- n as its activation function

Experiment-2

Aim: - To train a linear neuron regression model to map data of the given nature using Stochastic Gradient Descent Algorithm.

$x = (x_1, x_2)$	y
(0.54, -0.95)	1.33
(0.27, 0.50)	0.45
(0.00, -0.55)	0.56
(-0.60, 0.52)	-1.66
(-0.66, -0.82)	-1.07
(0.37, 0.91)	0.30

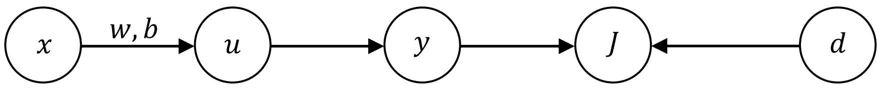
Use Learning rate $\alpha = 0.01$ and number of iterations = 200.
Plot the optimized best fit line and the graph of cost function vs epoch.

Apparatus: -

- Laptop Configuration
  - MacBook Air M1
  - 8GB RAM
  - 8-Core CPU
  - 7-Core GPU
- Libraries Used: -
  - TensorFlow 1.40
  - NumPy
  - PyLab
  - Matplotlib
- Coding Environment: - Python 3.7.

Theory: -

Linear Regression: -



Representing a dependent (output) variables as a linear combination of independent (input) variables is known as linear regression.

The output of a linear neuron can be written as

$$y = w_1x_1 + w_2x_2 + w_3x_3 + \ \dots\dots\dots + w_nx_n + b$$

Where $x_1, x_2, x_3 \dots\dots\dots x_n$ are input features.

i.e., a Linear neuron performs linear regression and the weights and biases acts as regression coefficients.

$$\# \; In \; a \; given \; data \; set \; \{x_p, d_p\}_{p=1}^{P}$$

$$where \; input \; x_p \in R^n$$

$$\& \; d_p \in R$$

Training a linear neuron finds a regression function

$$\varphi: R^n \to R$$

Given by Linear Mapping;

$$y = W^T X + b$$

\# The cost function $J(w, b)$, for regression is given as the square error b/w. neuron outputs, and the target.

Given a training pattern $\{\overline{\boldsymbol{x}}, \boldsymbol{d}\}$, half squared error Cost function is given by: - $-\frac{1}{2}(\boldsymbol{d}-\boldsymbol{y})^2$

Where y is the neuron for the input pattern $\bar{x}$ & $\boldsymbol{y} = \boldsymbol{W}^T\boldsymbol{X} + \boldsymbol{b}$

The ½ in the cost function is introduced to simplify the learning process and data does not affect the optimization parameters (weights and biases)

$\boldsymbol{J} = \frac{1}{2}(\boldsymbol{d}-\boldsymbol{y})^2$

$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{u}) = \boldsymbol{u} = \boldsymbol{W}^T\boldsymbol{X} + \boldsymbol{b}$

Linear Activation Function

$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial y} \times \frac{\partial y}{\partial u} \times \frac{\partial u}{\partial w}$

$\frac{\partial J}{\partial y} = -(\boldsymbol{d}-\boldsymbol{y})$

$\frac{\partial y}{\partial u} = \mathbf{1}$

$\frac{\partial u}{\partial w} = \boldsymbol{x}$

=> $\frac{\partial J}{\partial w} = -(\boldsymbol{d}-\boldsymbol{y})\vec{\boldsymbol{x}}$

Now, we will update bias-

$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial y} \times \frac{\partial y}{\partial u} \times \frac{\partial u}{\partial w}$

$= -(\boldsymbol{d}-\boldsymbol{y})$

#The Gradient Learning Equation

$\boldsymbol{w} = \boldsymbol{w} - \boldsymbol{\alpha}\nabla_{\boldsymbol{w}}\boldsymbol{J}$

$\boldsymbol{w} = \boldsymbol{w} + \boldsymbol{\alpha}(\boldsymbol{d}-\boldsymbol{y})\vec{\boldsymbol{x}}$

$\boldsymbol{b} = \boldsymbol{b} - \boldsymbol{\alpha}\nabla_{\boldsymbol{b}}\boldsymbol{J}$

$\boldsymbol{b} = \boldsymbol{b} + \boldsymbol{\alpha}(\mathbf{d}-\mathbf{y})$

Learning Algorithm: -

$\boldsymbol{for\ training\ patten}\ \{\boldsymbol{x_p}, \boldsymbol{d_p}\}_{\boldsymbol{p=1}}^{\boldsymbol{P}}$

1) $\boldsymbol{Set\ learning\ Rate}\ \ \boldsymbol{\alpha}$
2) $\boldsymbol{initialize}\ (\mathbf{w}, \boldsymbol{b})$
3) $\boldsymbol{Until\ the\ Convergence}$:

$\boldsymbol{for\ each\ pattern\ in}\ \{\boldsymbol{x_p}, \boldsymbol{d_p}\}_{\boldsymbol{p=1}}^{\boldsymbol{P}}$

$\boldsymbol{y} = \boldsymbol{w}^T\boldsymbol{x_p} + \boldsymbol{b}$

$\boldsymbol{w} = \boldsymbol{w} + \boldsymbol{\alpha}(\boldsymbol{d_p} - \boldsymbol{y_p})\vec{\boldsymbol{x}}$

$\boldsymbol{b} = \boldsymbol{b} + \boldsymbol{\alpha}(\boldsymbol{d_p} - \boldsymbol{y_p})$

Python CODE: -

```
import tensorflow as tf
import numpy as np
import pylab as plt
from mpl_toolkits.mplot3d import Axes3D

import os
if not os.path.isdir('figures'):
  os.makedirs('figures')

tf.logging.set_verbosity(tf.compat.v1.logging.ERROR)

no_iters = 200
lr = 0.01

SEED = 10
np.random.seed(SEED)

# generate training data
X = 2*np.random.rand(6, 2) - 1
Y = np.dot(X, [2.53, -0.47]) - 0.5 + np.random.rand(6)

print(X)
print(Y)
print(lr)

# Model parameters
w = tf.Variable(np.random.rand(2), dtype=tf.float32)
b = tf.Variable(0., dtype=tf.float32)

# Model input and output
x = tf.placeholder(tf.float32, [2])
d = tf.placeholder(tf.float32)

y = tf.tensordot(x, w, axes=1) + b
loss = tf.square(d - y) # sum of the squares
# optimizer
grad_w = -(d - y)*x
grad_b = -(d - y)
w_new = w.assign(w - lr*grad_w)
b_new = b.assign(b - lr*grad_b)

# initialize variables
sess = tf.Session()
sess.run(tf.global_variables_initializer())
# print initial weights and biases
w_, b_ = sess.run([w, b])
print('w: {}, b: {}'.format(w_, b_))
```

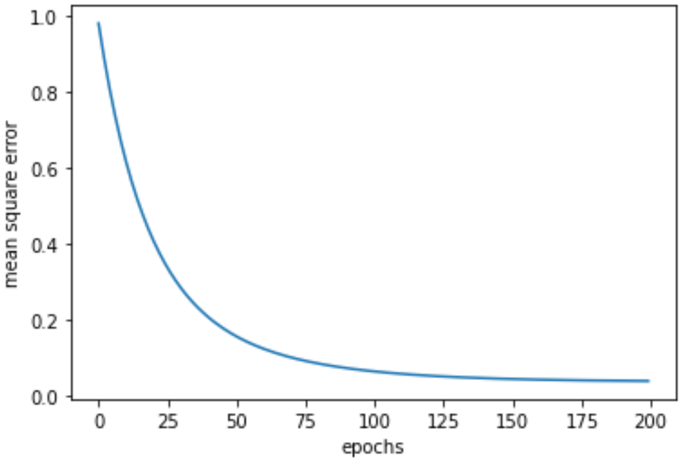
```
# training loop begins
err = []
idx = np.arange(len(X))
for i in range(no_iters):

  err_ = []
  np.random.shuffle(idx)
  X, Y = X[idx], Y[idx]
  for p in np.arange(len(X)):
    y_, loss_, w_, b_ = sess.run([y, loss, w_new, b_new], {x: X[p], d: Y[p]})

    if i == 0:
      print('iter: {}'.format(i+1))
      print('p: {}'.format(p+1))
      print('x:{}, d:{}'.format(X[p], Y[p]))
      print('y: {}'.format(y_))
      print('se: {}'.format(loss_))
      print('w: {}, b: {}'.format(w_, b_))

    err_.append(loss_)
  err.append(np.mean(err_))
  if i%10 == 0:
          print('iter: %d, mse: %g'%(i, err[i]))

# print final weights and error
w_, b_ = sess.run([w, b])
print('w: %s, b: %s'%(w_, b_))
print('mse: %.3f'%err[no_iters-1])

# plot learning curve
plt.figure(1)
plt.plot(range(no_iters), err)
plt.xlabel('epochs')
plt.ylabel('mean square error')
plt.savefig('./figures/2.1a_1.png')

# find the predicted values of inputs
pred = []
for p in np.arange(len(X)):
  pred.append(sess.run(y, {x:X[p]}))

# plot targets and predictions
fig = plt.figure(2)
ax = fig.gca(projection = '3d')
ax.scatter(X[:,0], X[:,1], Y, 'ro', label='targets')
ax.scatter(X[:,0], X[:,1], pred, 'b^', label='predicted')

X1 = np.arange(-1, 1, 0.1)
X2 = np.arange(-1, 1, 0.1)
X1,X2 = np.meshgrid(X1,X2)
```

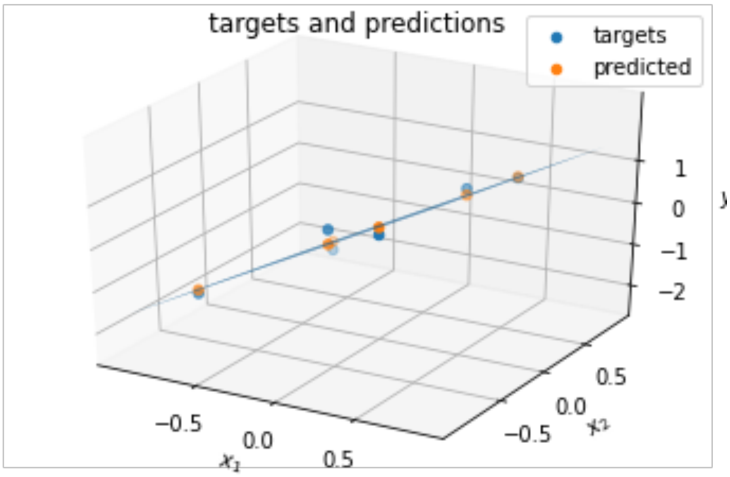
```
Z = w_[0]*X1 + w_[1]*X2 + b_
ax.plot_surface(X1, X2, Z)

ax.set_zticks([ -2, -1, 0, 1])
ax.set_xticks([-0.5, 0, 0.5])
ax.set_yticks([-0.5, 0, 0.5])
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_zlabel('$y$')
plt.title('targets and predictions')
plt.legend()
plt.savefig('./figures/2.1a_2.png')

plt.show()
```

OUTPUT: -

```
[[ 0.54264129 -0.9584961 ]
 [ 0.26729647  0.49760777]
 [-0.00298598 -0.55040671]
 [-0.60387427  0.52106142]
 [-0.66177833 -0.82332037]
 [ 0.37071964  0.90678669]]
[ 1.32732389  0.45457668  0.5637576  -1.66017471 -1.06558327  0.303607  ]
0.01
w: [0.91777414 0.71457577], b: 0.0
iter: 1
p: 1
x:[ 0.54264129 -0.9584961 ], d:1.3273238888598116
y: -0.1868959665298462
se: 2.2928619384765625
w: [0.92599094 0.70006204], b: 0.015142198652029037
iter: 1
p: 2
x:[-0.66177833 -0.82332037], d:-1.0655832748853409
y: -1.1740338802337646
se: 0.011761543340981007
w: [0.92527324 0.69916916], b: 0.01622670516371727
iter: 1
p: 3
x:[-0.00298598 -0.55040671], d:0.5637575971042242
y: -0.371363520622534
se: 0.8744515180587769
w: [0.92524534 0.6940222 ], b: 0.025577915832400322
iter: 1
p: 4
x:[0.26729647 0.49760777], d:0.454576682526435
y: 0.6182435750961304
se: 0.02678685635328293
w: [0.92480785 0.69320774], b: 0.023941246792674065
iter: 1
p: 5
x:[-0.60387427  0.52106142], d:-1.6601747069539354
y: -0.1733226180076599
se: 2.2107293605804443
w: [0.9337866 0.6854603], b: 0.009072725661098957
iter: 1
p: 6
```

```
x:[0.37071964 0.90678669], d:0.303607003688446484
y: 0.9768120646476746
se: 0.4532049894332886
w: [0.93129086 0.6793558 ], b: 0.0023406757973134518
iter: 0, mse: 0.978299
iter: 10, mse: 0.613169
  ⋮
  ⋮
  ⋮
iter: 180, mse: 0.0383732
iter: 190, mse: 0.0376284
w: [ 2.0039272  -0.43821055], b: -0.013066508
mse: 0.037
```





Precautions: -

- Design an artificial neur
- on which takes a three-dimensional
- data as input and uses sigmoidal function
- n as its activation function

Experiment-3

Aim: - To train a discrete perceptron classification model to classify the following two dimensional model.

$x = (x_1, x_2)$	y
(1.0, 2.5)	B
(2.0, -1.0)	A
(1.5, 3.0)	B
(0.0, -1.5)	A
(-3.5, 1.0)	B
(2.5, 0.0)	A
(0.5, 1.5)	A
(0.0, -2.0)	A

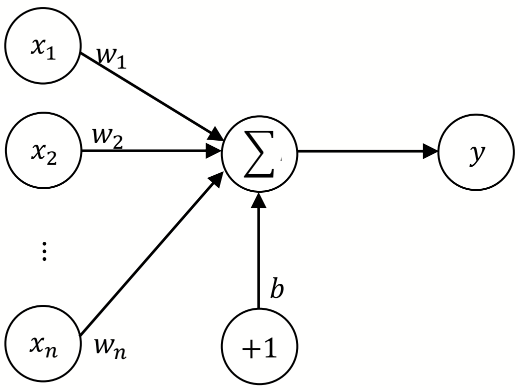
Plot the decision boundary and the graph of cost vs epoch.

Apparatus: -

- Laptop Configuration
    - MacBook Air M1
    - 8GB RAM
    - 8-Core CPU
    - 7-Core GPU
- Libraries Used: -
    - TensorFlow 1.40
    - NumPy
    - PyLab
    - Matplotlib
- Coding Environment: - Python 3.7.

Theory: -

 # Discrete Perceptron: - It's a neuron that has a threshold of unit step activation function.



$$y = f(u) = \begin{cases} 1, & u > 0 \\ 0, & u \leq 0 \end{cases}$$

 # Discrete Perceptron Classifier: - Classifies input patterns into two or more classes with a linear discriminant function.

 # Indicator Function I( ): - It takes a value 1 when the condition given is true and value 0 when the condition is false.

$$I(x) = \begin{cases} 1, & x \text{ is true} \\ 0, & x \text{ is false} \end{cases}$$

Using an indicator function, the output of a discrete perceptron can be written as , $y = 1(u > 0)\, where\ u = W^T X + b$

Learning Algorithm: -

$$\boldsymbol{Given\ \vec{p}\ training\ pairs\ \{x_p, d_p\}_{p=1}^{P}}$$
$$\boldsymbol{where, x_p \epsilon R^n}$$
$$\boldsymbol{is\ the\ n - dimensional\ input\ \&\ d_p \epsilon\ (0, 1)}$$

 # Discrete Perceptron Learning Algorithm: - is a supervised scheme. It was proposed by Ministry in 1950 and it's convergence cab be proved.

However, because of non-differentiable character states of the activation, the discrete perceptron learning algorithm can't be derived from cost function.

Discrete perceptron learning algorithm finds a linear decision boundary in the feature space.

The change of weight is proportional to the difference (error) between the desired output 'd' & the perceptron output 'y'.

$\boldsymbol{for\ training\ patten\ (X, d)}$:

$$\boldsymbol{u = W^T X + b}$$
$$\boldsymbol{y = 1(u > 0)}$$
$$\boldsymbol{\delta = d - y}$$

$\boldsymbol{untill\ the\ convergence}$:

$$\boldsymbol{w \longleftarrow w + \alpha\delta\vec{x}}$$
$$\boldsymbol{b \longleftarrow b + \alpha\delta}$$

$\boldsymbol{Note\ that\ \delta = \{-1, 0, 1\}}$

$$\boldsymbol{\alpha \epsilon (0, 1)}$$

$\boldsymbol{Where\ \alpha = 0.4\ , Learning\ equation\ are\ reffered\ to\ as\ simple\ perceptron\ rule}$

Python CODE: -

```
import tensorflow as tf
import numpy as np
import pylab as plt
from mpl_toolkits.mplot3d import Axes3D

import os
if not os.path.isdir('figures'):
    os.makedirs('figures')

tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)

no_iters = 30
lr = 0.4
SEED = 10
np.random.seed(SEED)

# training data
x_train = np.array([[1.0, 2.5], [2.0, -1.0], [1.5, 3.0],
  [0.0, -1.5], [-3.5, 1.0], [2.5, 0.0], [0.5, 1.5], [0.0, -2.0]])
y_train = np.array([1, 0, 1, 0, 1, 0, 0, 0])

print(x_train)
print(y_train)
print(lr)
# Model parameters
```

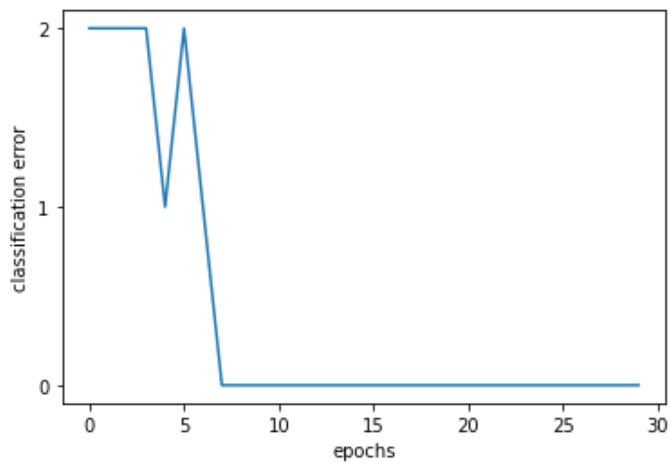
```python
w = tf.Variable(np.random.rand(2), dtype=tf.float32)
b = tf.Variable(0., dtype=tf.float32)

# Model input and output
x = tf.placeholder(tf.float32)
d = tf.placeholder(tf.int32)

u = tf.tensordot(x,w, axes=1) + b
y = tf.where(tf.greater(u, 0), 1, 0)

delta = d - y
delta = tf.cast(delta, tf.float32)

w_new = w.assign(w + lr*delta*x)
b_new = b.assign(b + lr*delta)


# initialize the variables
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

# print initial weights
w_, b_ = sess.run([w, b])
print('w: {}, b: {}'.format(w_, b_))

# training loop
err = []
idx = np.arange(len(x_train))
for i in range(no_iters):
    np.random.shuffle(idx)
    x_train, y_train = x_train[idx], y_train[idx]

    err_ = 0
    for p in np.arange(len(x_train)):
        u_, y_, w_, b_ = sess.run([u, y, w_new, b_new], {x: x_train[p], d: y_train[p]})
        err_ += y_ != y_train[p]

        if (i == 0):
            print('p: {}'.format(p+1))
            print('x: {}'.format(x_train[p]))
            print('d: {}'.format(y_train[p]))
            print('u: {}'.format(u_))
            print('y: {}'.format(y_))
            print('w: {}, b: {}'.format(w_, b_))

    err.append(err_)

    print('iter: {}, error: {}'.format(i+1, err[i]))
# print final weights
```

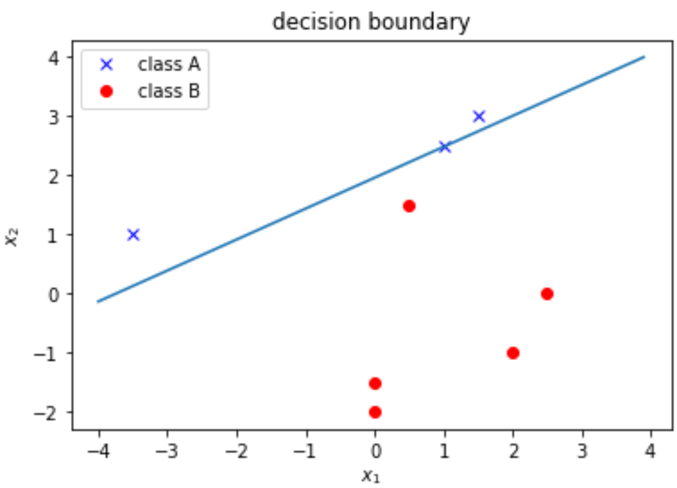
```python
print('w: {}, b: {}'.format(w_, b_))

# plot the learning curves
plt.figure(2)
plt.plot(range(no_iters), err)
plt.xlabel('epochs')
plt.ylabel('classification error')
plt.yticks([0, 1, 2])
plt.savefig('./figures/3.1a_2.png')

# find predicctions
pred = []
for p in np.arange(len(x_train)):
  pred.append(sess.run(y, {x: x_train[p]}))
print(y_train, pred)

# plot the training data
plt.figure(1)
plt.plot(x_train[y_train==1,0], x_train[y_train==1,1],'bx', label ='class A')
plt.plot(x_train[y_train==0,0],x_train[y_train==0,1],'ro', label='class B')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.title('training data')
plt.legend()
plt.savefig('./figures/3.1a_1.png')

# plot the decision boundary
x1 = np.arange(-4, 4, 0.1)
x2 = -(x1*w_[0] + b_)/w_[1]
plt.figure(3)
plt.plot(x_train[y_train==1,0], x_train[y_train==1,1],'bx', label ='class A')
plt.plot(x_train[y_train==0,0],x_train[y_train==0,1],'ro', label='class B')
plt.plot(x1, x2, '-')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.title('decision boundary')
plt.legend()
plt.savefig('./figures/3.1a_3.png')
plt.show()
```

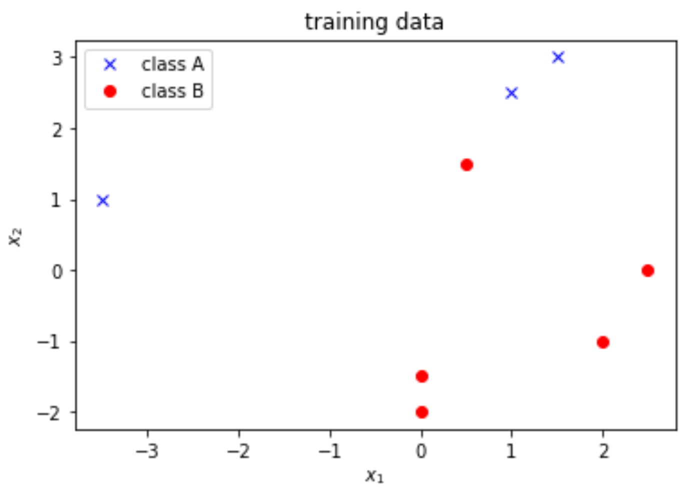
OUTPUT: -

```
[[ 1.   2.5]
 [ 2.  -1. ]
 [ 1.5  3. ]
 [ 0.  -1.5]
 [-3.5  1. ]
 [ 2.5  0. ]
 [ 0.5  1.5]
 [ 0.  -2. ]]
[1 0 1 0 1 0 0 0]
```

```
0.4

w: [0.77132064 0.02075195], b: 0.0
p: 1
x: [1.5 3. ]
d: 1
u: 1.2192368507385254
y: 1
w: [0.77132064 0.02075195], b: 0.0
p: 2
x: [ 0. -2.]
d: 0
u: -0.0415038987994194
  ⋮
  ⋮
  ⋮
w: [-0.22867936  0.02075195], b: -0.4000000059604645
p: 7
x: [ 2. -1.]
d: 0
u: -0.8781106472015381
y: 0
w: [-0.22867936  0.02075195], b: -0.4000000059604645
p: 8
x: [1.  2.5]
d: 1
u: -0.5767995119094849
y: 0
w: [0.17132065 1.020752  ], b: 0.0
iter: 1, error: 2
iter: 2, error: 2
  ⋮
  ⋮
  ⋮

iter: 29, error: 0
iter: 30, error: 0
w: [-0.4286793  0.8207519],
b: -1.600000023841858
[1 1 0 1 0 0 0 0]
[1, 1, 0, 1, 0, 0, 0, 0]
```







Experiment-4

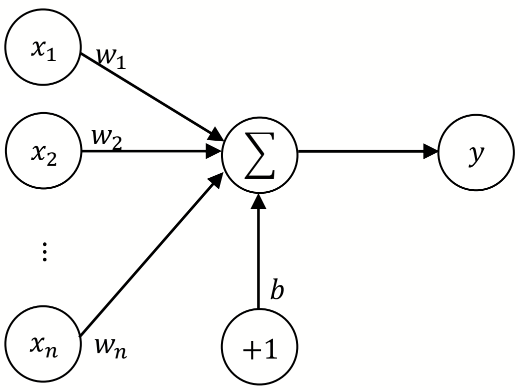
Aim: - To train a discr

Apparatus: -

- Laptop Configuration CPU, GPU, Core, clock etc.
- Laptop Configuration
    - MacBook Air M1
    - 8GB RAM
    - 8-Core CPU
    - 7-Core GPU
- Libraries Used: -
    - TensorFlow 1.40
    - NumPy
    - PyLab
    - Matplotlib
- Coding Environment: - Python 3.7.

Theory: -

 # Discrete Perceptron: - It's a neuron that has a threshold of unit step activation function.



$$y = f(u) = \begin{cases} \mathbf{1}, & u > 0 \\ \mathbf{0}, & u \leq 0 \end{cases}$$

 # Discrete Perceptron Classifier: - Classifies input patterns into two or more classes with a linear discriminant function.

 # Indicator Function I( ): - It takes a value 1 when the condition given is true and value 0 when the condition is false.

$$I(x) = \begin{cases} 1, & x \text{ is true} \\ 0, & x \text{ is false} \end{cases}$$

Using an indicator function, the output of a discrete perceptron can be written as , $y = 1(u > 0) where\ u = W^T X + b$

Learning Algorithm: -

$$\textit{Given } \vec{p} \textit{ training pairs } \{x_p, d_p\}_{p=1}^{P}$$
$$\textit{where}, x_p \epsilon R^n$$
$$\textit{is the } n - \textit{dimensional input } \& \ d_p \epsilon\ (0, 1)$$

 # Discrete Perceptron Learning Algorithm: - is a supervised scheme. It was proposed by Ministry in 1950 and it's convergence cab be proved.

However, because of non-differentiable character states of the activation, the discrete perceptron learning algorithm can't be derived from cost function.

Discrete perceptron learning algorithm finds a linear decision boundary in the feature space.

The change of weight is proportional to the difference (error) between the desired output ‘d’ & the perceptron output ‘y’.

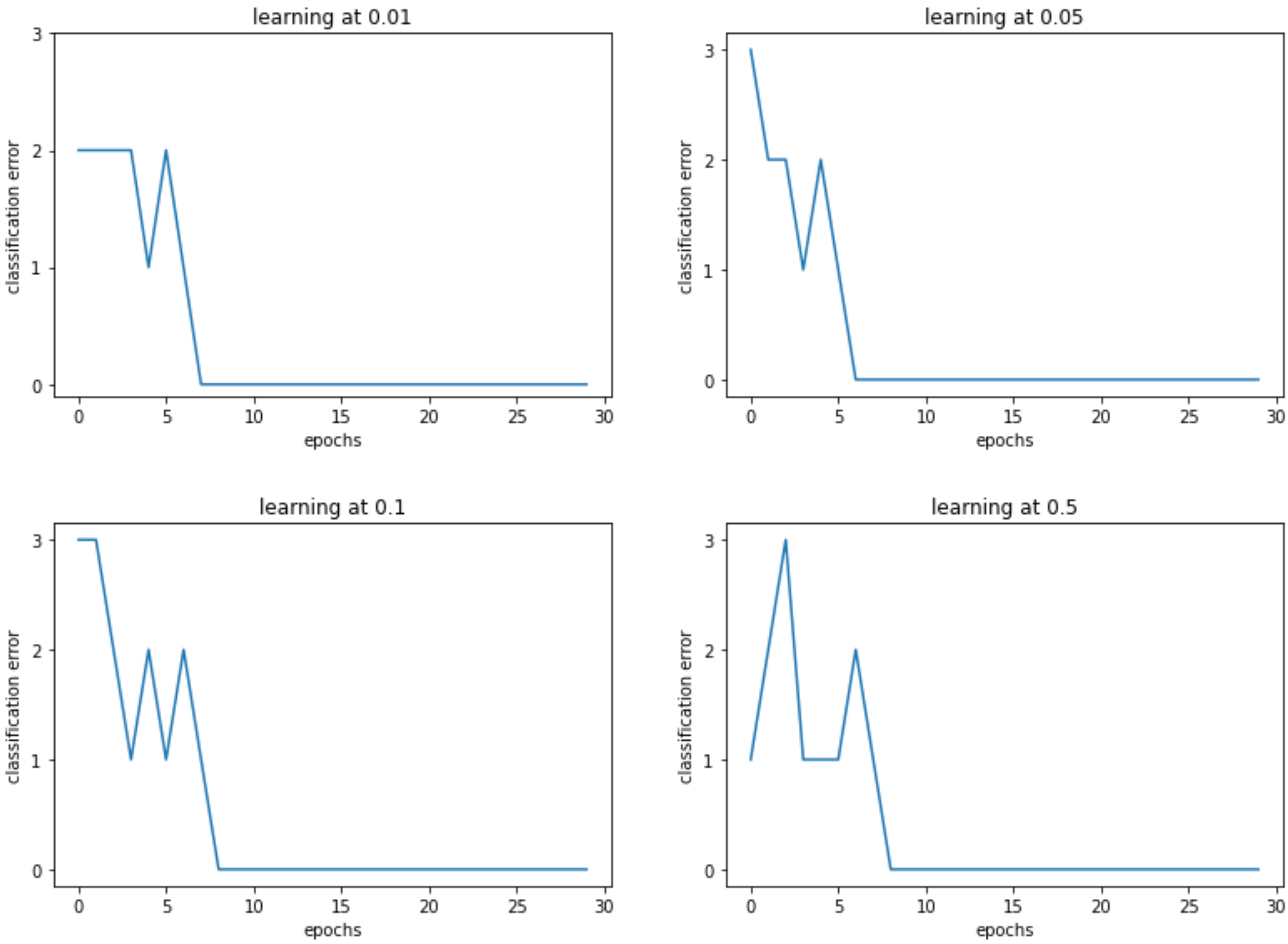
$$\boldsymbol{for\ training\ patten}\ (\boldsymbol{X}, \boldsymbol{d}):$$
$$\boldsymbol{u} = \boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{X} + \boldsymbol{b}$$
$$\boldsymbol{y} = \mathbf{1}(\boldsymbol{u} > \mathbf{0})$$
$$\boldsymbol{\delta} = \boldsymbol{d} - \boldsymbol{y}$$
$$\boldsymbol{untill\ the\ convergence}:$$
$$\boldsymbol{w} \longleftarrow \boldsymbol{w} + \boldsymbol{\alpha}\boldsymbol{\delta}\vec{\boldsymbol{x}}$$
$$\boldsymbol{b} \longleftarrow \boldsymbol{b} + \boldsymbol{\alpha}\boldsymbol{\delta}$$
$$\boldsymbol{Note\ that\ \delta} = \{-\mathbf{1}, \mathbf{0}, \mathbf{1}\}$$
$$\boldsymbol{\alpha}\boldsymbol{\epsilon}(\mathbf{0}, \mathbf{1})$$
$$\boldsymbol{Where\ \alpha} = \mathbf{1.0}, \boldsymbol{Learning\ equation\ are\ reffered\ to\ as\ simple\ perceptron\ rule}$$

Python CODE: -

```
import tensorflow as tf
import numpy as np
import pylab as plt
import multiprocessing as mp

import os
if not os.path.isdir('figures'):
    print('creating the figures folder')
    os.makedirs('figures')

tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)

no_iters = 30
SEED = 10
np.random.seed(SEED)

# training data
x_train = np.array([[1.0, 2.5], [2.0, -1.0], [1.5, 3.0],
  [0.0, -1.5], [-3.5, 1.0], [2.5, 0.0], [0.5, 1.5], [0.0, -2.0]])
y_train = np.array([1, 0, 1, 0, 1, 0, 0, 0])

# Model parameters
w = tf.Variable(np.random.rand(2), dtype=tf.float32)
b = tf.Variable(0., dtype=tf.float32)
lr = tf.Variable(0.4, dtype=tf.float32)

# Model input and output
x = tf.placeholder(tf.float32)
d = tf.placeholder(tf.int32)


u = tf.tensordot(x,w, axes=1) + b
y = tf.where(tf.greater(u, 0), 1, 0)
```

```
delta = d - y
delta = tf.cast(delta, tf.float32)

w_new = w.assign(w + lr*delta*x)
b_new = b.assign(b + lr*delta)


# training loop
def my_train(rate):
  init = tf.global_variables_initializer()
  sess = tf.Session()
  sess.run(init) # reset values to wrong

  X, Y = x_train, y_train
  err = []
  idx = np.arange(len(X))
  for i in range(no_iters):
    np.random.shuffle(idx)
    X, Y = X[idx], Y[idx]
    err_ = 0
    for p in np.arange(len(X)):
        y_, w_, b_ = sess.run([y, w_new, b_new], {x: X[p], d: Y[p]})
        err_ += y_ != Y[p]

    err.append(err_)

  return err


rates = [0.01, 0.05, 0.1, 0.5]


for i in range(len(rates)):
    cost = my_train(rates[i])
    plt.figure()
    plt.plot(range(no_iters), cost)
    plt.xlabel('epochs')
    plt.ylabel('classification error')
    plt.yticks([0, 1, 2, 3])
    plt.title('learning at {}'.format(rates[i]))
    plt.savefig('./figures/3.1b_{}.png'.format(rates[i]))


plt.show()
```

OUTPUT: -


learning at 0.01
learning at 0.05
learning at 0.1
learning at 0.5
classification error
epochs
0
1
2
3
5
10
15
20
25
30

Experiment-5

Aim: - 3.2

Apparatus: -

- Laptop Configuration CPU, GPU, Core, clock etc.
- Laptop Configuration
  - MacBook Air M1
  - 8GB RAM
  - 8-Core CPU
  - 7-Core GPU
- Libraries Used: -
  - TensorFlow 1.40
  - NumPy
  - PyLab
  - Matplotlib
- Coding Environment: - Python 3.7.

Theory: -

 # <u>Logistic Regression Neuron</u>: - A Logistic regression neuron performs a binary classification of input, i.e., it classifies inputs into two classes with levels 0 & 1.

The activation of logistic regression neuron gives the probability of the neuron, belongs to class one.

Given an input $\vec{X}$, the activation of neuron is-

$$\begin{aligned} f(u) &= P(y = 1/\vec{x}) \\ &= \frac{1}{1+e^{-u}} \end{aligned}$$

Activation function of neuron is given by sigmoidal or logistic function.

The output y of the neuron is not equal to the activation: -

Now,

$$\begin{aligned} P\left(y = \frac{0}{\vec{x}}\right) &= 1 - P(y = 1/\vec{x}) \\ &= 1 - f(u) \end{aligned}$$

And,

$$y = 1(f(u) > 0.5)$$

Given a training pattern $\vec{X}, \vec{b}$ where $\vec{X} \epsilon R^n$ and $\vec{d} \epsilon (0, 1)$.

The cost function of classification is given by cross entropy.

$$J = -d.\log\big(f(u)\big) - (1-d).\log\,(1-f(u))$$

The cost function J, is minimized using the gradient descent procedure.

 # <u>Computing the gradient</u>$\left(\frac{\partial J}{\partial u}\right)$: -

$$\frac{\partial J}{\partial u} = -\frac{\partial}{\partial u}\left[d.\log\big(f(u)\big) - (1-d).\log\big(1-f(u)\big)\right] \times \frac{\partial f(u)}{\partial u}$$

$$\frac{\partial J}{\partial u} = -\left[\frac{d}{f(u)} - \frac{(1-d)}{\big(1-f(u)\big)}\right] \times f'(u)$$

$$where, f'(u) = f(u).\,[1-f(u)]$$

$$thus, \quad \frac{\partial J}{\partial u} = \left[\frac{d-f(u)}{f(u)[1-f(u)]}\right] \times f(u)[1-f(u)]$$

$$\frac{\partial J}{\partial u} = -[d-f(u)]$$

Now,

$$\nabla_w J = \frac{\partial J}{\partial u} \times \frac{\partial u}{\partial w} = -[d-f(u)]\vec{x}$$

$$\nabla_b J = \frac{\partial J}{\partial u} \times \frac{\partial u}{\partial b} = -[d-f(u)]$$

Thus, our weight updating equation will become-

$$w \longleftarrow w + \alpha[d-f(u)]\vec{x}$$

$$b \longleftarrow b + \alpha[d-f(u)]$$

Learning Algorithm: -

$for\ a\ given\ input\ patten\ \vec{X}, \vec{d}$

1) $Set\ learning\ Rate\ \ \alpha$
2) $initialize\ (w, b)$
3) $Repet\ until\ the\ Convergence$

$$w \longleftarrow w + \alpha[d-f(u)]\vec{x}$$
$$b \longleftarrow b + \alpha[d-f(u)]$$

Python CODE: -

```
import tensorflow as tf
import numpy as np
import pylab as plt
from mpl_toolkits.mplot3d import Axes3D

import os
if not os.path.isdir('figures'):
    print('creating the figures folder')
    os.makedirs('figures')

tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)

no_iters = 300
lr = 0.4
```

```
SEED = 10
np.random.seed(SEED)


# training data
x_train = np.array([[1.33, 0.72], [-1.55, -0.01], [0.62, -0.72],
    [0.27, 0.11], [0.0, -0.17], [0.43, 1.2], [-0.97, 1.03], [0.23, 0.45]])
y_train = np.array([0, 1, 1, 1, 1, 0, 0, 0]).reshape(8,1)

print(x_train)
print(y_train)
print(lr)

# Model parameters
w = tf.Variable(np.random.rand(2,1), dtype=tf.float32)
b = tf.Variable(0., dtype=tf.float32)

# Model input and output
x = tf.placeholder(tf.float32, x_train.shape)
d = tf.placeholder(tf.int32, y_train.shape)


u = tf.matmul(x, w) + b
f_u = tf.sigmoid(u)
d_float = tf.cast(d, tf.float32)

loss = -tf.reduce_sum(d_float*tf.log(f_u) + (1-d_float)*tf.log(1-f_u))
class_err = tf.reduce_sum(tf.cast(tf.not_equal(f_u > 0.5, y_train), tf.int32))

grad_u = -(d_float - f_u)
grad_w = tf.matmul(tf.transpose(x), grad_u)
grad_b = tf.reduce_sum(grad_u)

w_new = w.assign(w - lr*grad_w)
b_new = b.assign(b - lr*grad_b)


# training loop
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init) # reset values to wrong
w_, b_ = sess.run([w, b])
print('w: {}, b: {}'.format(w_, b_))

err = []
c_err = []
for i in range(no_iters):
    u_, f_u_, loss_, c_err_, w_, b_ = sess.run([u, f_u, loss, class_err, w_new, b_new],
{x: x_train, d: y_train})
```

```python
    if (i == 0):
        print('u:{}'.format(u_))
        print('f_u:{}'.format(f_u_))
        print('y:{}'.format(f_u_ > 0.5))
        print('loss:{}'.format(loss_))
        print('error:{}'.format(c_err_))
        print('w: {}, b: {}'.format(w_, b_))

    err.append(loss_)
    c_err.append(c_err_)

    if (i%10 == 0):
        print('iter: {}, err: {}, cost: {}'.format(i, c_err[i], err[i]))


# evaluate training accuracy
print('w: {}, b: {}'.format(w_, b_))

print(f_u_ > 0.5)

plt.figure(1)
plt.plot(x_train[y_train[:,0]==1,0], x_train[y_train[:,0]==1,1],'bx', label ='class A')
plt.plot(x_train[y_train[:,0]==0,0],x_train[y_train[:,0]==0,1],'ro', label='class B')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.title('training data')
plt.legend()
plt.savefig('./figures/3.2_1.png')

plt.figure(2)
plt.plot(range(no_iters), err)
plt.xlabel('epochs')
plt.ylabel('cross-entropy')
plt.savefig('./figures/3.2_2.png')


plt.figure(3)
plt.plot(range(25), np.array(c_err)[:25])
plt.xlabel('epochs')
plt.ylabel('classification error')
plt.savefig('./figures/3.2_3.png')

x1 = np.arange(-2, 2, 0.1)
x2 = -(x1*w_[0] + b_)/w_[1]

plt.figure(4)
plt.plot(x_train[y_train[:,0]==1,0], x_train[y_train[:,0]==1,1],'bx', label ='class A')
plt.plot(x_train[y_train[:,0]==0,0],x_train[y_train[:,0]==0,1],'ro', label='class B')
plt.plot(x1, x2, '-')
plt.xlabel('$x_1$')
```

```
plt.ylabel('$x_2$')
plt.title('decision boundary')
plt.legend()
plt.savefig('./figures/3.2_4.png')

plt.show()
```

OUTPUT: -

```
[[ 1.33  0.72]
 [-1.55 -0.01]
 [ 0.62 -0.72]
 [ 0.27  0.11]
 [ 0.   -0.17]
 [ 0.43  1.2 ]
 [-0.97  1.03]
 [ 0.23  0.45]]
[[0]
 [1]
 [1]
 [1]
 [1]
 [0]
 [0]
 [0]]
0.4
w: [[0.77132064]
 [0.02075195]], b: 0.0
u:[[ 1.040798  ]
 [-1.1957545 ]
 [ 0.4632774 ]
 [ 0.2105393 ]
 [-0.00352783]
 [ 0.3565702 ]
 [-0.7268065 ]
 [ 0.18674213]]
f_u:[[0.73900396]
 [0.2322313 ]
 [0.6137914 ]
 [0.5524413 ]
 [0.49911806]
 [0.5882099 ]
 [0.3258959 ]
 [0.54655033]]
y:[[ True]
 [False]
 [ True]
 [ True]
 [False]
 [ True]
 [False]
 [ True]]
loss:6.6521759033203125
error:5
w: [[ 0.021263 ]
 [-0.8357367]], b: -0.038896895945072174
iter: 0, err: 5, cost: 6.6521759033203125
iter: 10, err: 0, cost: 1.4870713949203491
  ⋮
  ⋮
  ⋮
```

```
iter: 280, err: 0, cost: 0.2432633638381958
iter: 290, err: 0, cost: 0.23725102841854095
w: [[ -1.2952098]
 [-12.666678 ]], b: 3.8386905193328857
[[False]
 [ True]
 [ True]
 [ True]
 [ True]
 [False]
 [False]
 [False]]
```