Homework - Neural networks - Part B (55 points)

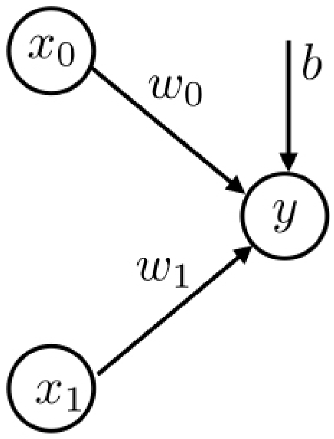
Gradient descent for simple two and three layer models

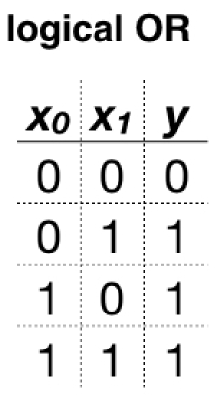
by *Brenden Lake* and *Todd Gureckis*
Computational Cognitive Modeling
NYU class webpage: https://brendenlake.github.io/CCM-site/

This homework is due before midnight on Feb. 15, 2024.

The first part of this assignment implements the gradient descent algorithm for a simple artificial neuron. The second part implements backpropagation for a simple network with one hidden unit.

In the first part, the neuron will learn to compute logical OR. The neuron model and logical OR are shown below, for inputs x_0 and x_1 and target output y.





logical OR

x_0	x_1	y
0	0	0
0	1	1
1	0	1
1	1	1

This assignment requires some basic PyTorch knowledge. You can review your notes from lab and this PyTorch tutorial. The "Introduction to PyTorch" section on the PyTorch website is also helpful.

In [ ]:
```
# Import libraries
from __future__ import print_function
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import torch
```

Let's create `torch.tensor` objects for representing the data matrix `X` with targets `Y_or` (for the logical OR function). Each row of `X` is a different input pattern.

In [ ]:
```
X_list = [[0.,0.], [0.,1.], [1.,0.], [1.,1.]]
X = torch.tensor(X_list)
Y_or = torch.tensor([0.,1.,1.,1.])
N = X.shape[0] # number of input patterns
print("Input tensor X:")
print('  has shape',X.shape)
print('  and contains',X)
print('Target tensor Y:')
print('  has shape',Y_or.shape)
print('  and contains',Y_or)
```

```
Input tensor X:
  has shape torch.Size([4, 2])
  and contains tensor([[0., 0.],
        [0., 1.],
        [1., 0.],
        [1., 1.]])
Target tensor Y:
  has shape torch.Size([4])
  and contains tensor([0., 1., 1., 1.])
```

The artificial neuron operates as follows. Given an input vector x (which is one row of input tensor X), the net input ($\mathbf{net}$) to the neuron is computed as follows

$$\mathbf{net} = \sum_i x_i w_i + b,$$

for weights w_i and bias b. The activation function $g(\mathbf{net})$ is the logistic function

$$g(\mathbf{net}) = \frac{1}{1 + e^{-\mathbf{net}}},$$

which is used to compute the predicted output $\hat{y} = g(\mathbf{net})$. Finally, the loss (squared error) for a particular pattern x is defined as

$$E(w, b) = (\hat{y} - y)^2,$$

where the target output is y. **Your main task is to manually compute the gradients of the loss E with respect to the neuron parameters:**

$$\frac{\partial E(w, b)}{\partial w}, \frac{\partial E(w, b)}{\partial b}.$$

By manually, we mean to program the gradient computation directly, using the formulas discussed in class. This is in contrast to using PyTorch's `autograd` (Automatric differentiation) that computes the gradient automatically, as discussed in class, lab, and in the PyTorch tutorial (e.g., `loss.backward()`). First, let's write the activation function and the loss in PyTorch.

In [ ]:
```
def g_logistic(net):
    return 1. / (1.+torch.exp(-net))

def loss(yhat,y):
    return (yhat-y)**2
```

Next, we'll also write two functions for examining the internal operations of the neuron, and the gradients of its parameters.

In [ ]:
```
def print_forward(x,yhat,y):
    # Examine network's prediction for input x
    print(' Input: ',end='')
    print(x.numpy())
    print(' Output: ' + str(round(yhat.item(),3)))
    print(' Target: ' + str(y.item()))

def print_grad(grad_w,grad_b):
    # Examine gradients
    print('  d_loss / d_w = ',end='')
    print(grad_w)
    print('  d_loss / d_b = ',end='')
    print(grad_b)
```

Now let's dive in and begin the implementation of stochastic gradient descent. We'll initialize our parameters w and b randomly, and proceed through a series of epochs of training. Each epoch involves visiting the four training patterns in random order, and updating the parameters after each presentation of an input pattern.

Problem 1 (10 points)

In the code below, fill in code to manually compute the gradient in closed form.

- See lecture slides for the equation for the gradient for the weights w.
- Derive (or reason) to get the equation for the gradient for bias b.

Problem 2 (5 points)

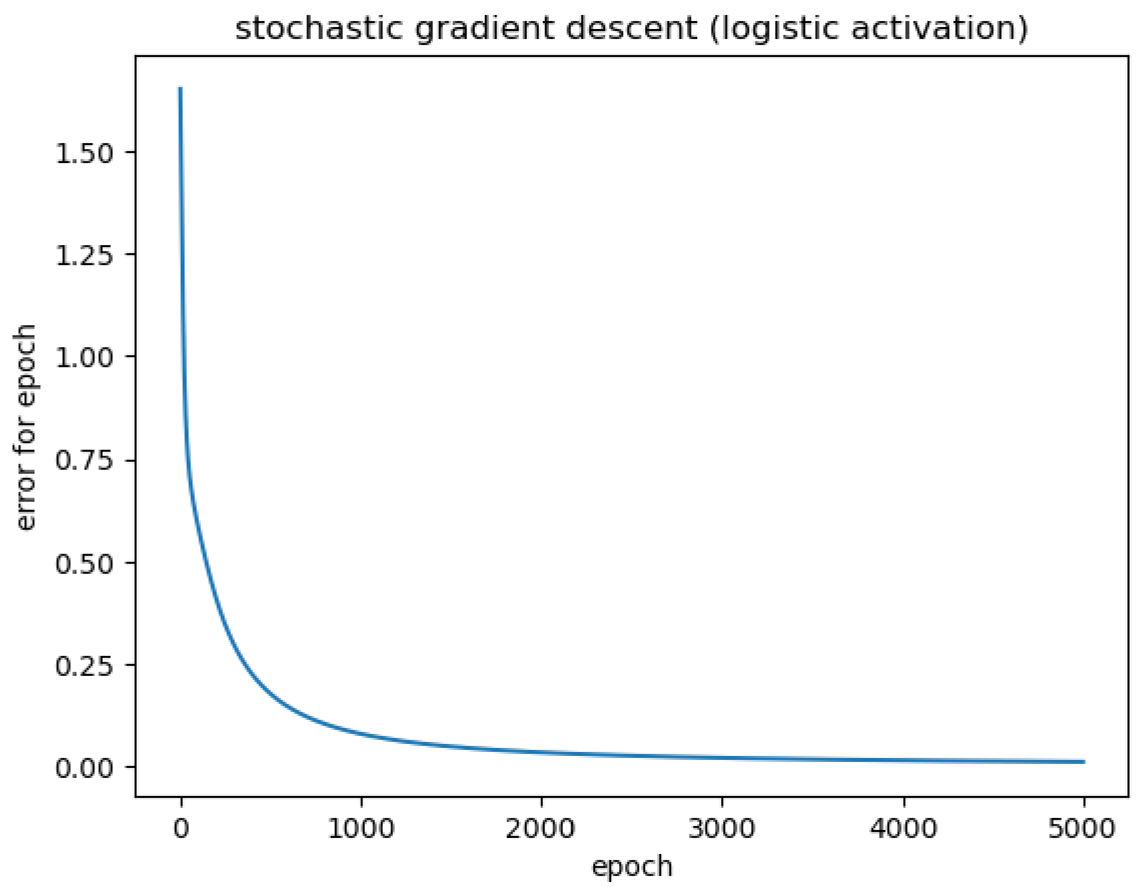
In the code below, fill in code for the weight and bias update rule for gradient descent.

After completing the code, run it to compare **your gradients** with the **ground-truth computed by PyTorch.** (There may be small differences that you shouldn't worry about, e.g. within 1e-6). Also, you can check the neuron's performance at the end of training.

In [ ]:
```
# Initialize parameters
#     Although you will implement gradient descent manually, let's set requires_
#     anyway so PyTorch will track the gradient too, and we can compare your gra
w = torch.randn(2, requires_grad=True) # [size 2] tensor
b = torch.randn(1, requires_grad=True) # [size 1] tensor

alpha = 0.05 # learning rate
nepochs = 5000 # number of epochs
```

```python
track_error = []
verbose = True
for e in range(nepochs): # for each epoch
    error_epoch = 0. # sum loss across the epoch
    perm = np.random.permutation(N)
    for p in perm: # visit data points in random order
        x_pat = X[p,:] # get one input pattern

        # compute output of neuron
        net = torch.dot(x_pat,w)+b
        yhat = g_logistic(net)

        # compute loss
        y = Y_or[p]
        myloss = loss(yhat,y)
        error_epoch += myloss.item()

        # Compute the gradient manually
        if verbose:
            print('Compute the gradient manually')
            print_forward(x_pat,yhat,y)
        with torch.no_grad():
            w_grad = 2 * (yhat - y) * g_logistic(net) * (1 - g_logistic(net)) *
            b_grad = 2 * (yhat - y) * g_logistic(net) * (1 - g_logistic(net))
        if verbose: print_grad(w_grad.numpy(),b_grad.numpy())

        # Compute the gradient with PyTorch and compre with manual values
        if verbose: print('Compute the gradient using PyTorch .backward()')
        myloss.backward()
        if verbose:
            print_grad(w.grad.numpy(),b.grad.numpy())
            print("")
        w.grad.zero_() # clear PyTorch's gradient
        b.grad.zero_()

        # Parameter update with gradient descent
        with torch.no_grad():
            w -= alpha * w_grad
            b -= alpha * b_grad

    if verbose==True: verbose=False
    track_error.append(error_epoch)
    if e % 50 == 0:
        print("epoch " + str(e) + "; error=" +str(round(error_epoch,3)))

# print a final pass through patterns
for p in range(X.shape[0]):
    x_pat = X[p]
    net = torch.dot(x_pat,w)+b
    yhat = g_logistic(net)
    y = Y_or[p]
    print("Final result:")
    print_forward(x_pat,yhat,y)
    print("")

# track output of gradient descent
plt.figure()
plt.clf()
plt.plot(track_error)
plt.title('stochastic gradient descent (logistic activation)')
plt.ylabel('error for epoch')
plt.xlabel('epoch')
plt.show()

# Print out the learned weights and bias
print(w, b)
```

```
Compute the gradient manually
 Input: [0. 1.]
 Output: 0.601
 Target: 1.0
  d_loss / d_w = [-0.         -0.19137797]
  d_loss / d_b = [-0.19137797]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [-0.         -0.19137798]
  d_loss / d_b = [-0.19137798]

Compute the gradient manually
 Input: [1. 1.]
 Output: 0.25
 Target: 1.0
  d_loss / d_w = [-0.2810817 -0.2810817]
  d_loss / d_b = [-0.2810817]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [-0.2810817 -0.2810817]
  d_loss / d_b = [-0.2810817]

Compute the gradient manually
 Input: [1. 0.]
 Output: 0.22
 Target: 1.0
  d_loss / d_w = [-0.26780987 -0.        ]
  d_loss / d_b = [-0.26780987]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [-0.2678099  0.       ]
  d_loss / d_b = [-0.2678099]

Compute the gradient manually
 Input: [0. 0.]
 Output: 0.566
 Target: 0.0
  d_loss / d_w = [0. 0.]
  d_loss / d_b = [0.27798703]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [0. 0.]
  d_loss / d_b = [0.27798706]

epoch 0; error=1.651
epoch 50; error=0.718
epoch 100; error=0.583
epoch 150; error=0.488
epoch 200; error=0.409
epoch 250; error=0.345
epoch 300; error=0.295
epoch 350; error=0.256
epoch 400; error=0.224
epoch 450; error=0.199
epoch 500; error=0.178
epoch 550; error=0.16
epoch 600; error=0.145
epoch 650; error=0.133
epoch 700; error=0.122
epoch 750; error=0.112
epoch 800; error=0.104
epoch 850; error=0.097
epoch 900; error=0.091
epoch 950; error=0.085
epoch 1000; error=0.08
epoch 1050; error=0.075
epoch 1100; error=0.071
epoch 1150; error=0.068
epoch 1200; error=0.064
epoch 1250; error=0.061
epoch 1300; error=0.058
epoch 1350; error=0.056
epoch 1400; error=0.053
epoch 1450; error=0.051
epoch 1500; error=0.049
epoch 1550; error=0.047
epoch 1600; error=0.045
epoch 1650; error=0.044
epoch 1700; error=0.042
epoch 1750; error=0.041
epoch 1800; error=0.039
epoch 1850; error=0.038
epoch 1900; error=0.037
epoch 1950; error=0.036
epoch 2000; error=0.035
epoch 2050; error=0.034
epoch 2100; error=0.033
epoch 2150; error=0.032
epoch 2200; error=0.031
epoch 2250; error=0.03
epoch 2300; error=0.029
epoch 2350; error=0.029
epoch 2400; error=0.028
epoch 2450; error=0.027
epoch 2500; error=0.027
epoch 2550; error=0.026
epoch 2600; error=0.025
epoch 2650; error=0.025
epoch 2700; error=0.024
epoch 2750; error=0.024
epoch 2800; error=0.023
epoch 2850; error=0.023
epoch 2900; error=0.022
epoch 2950; error=0.022
epoch 3000; error=0.021
epoch 3050; error=0.021
epoch 3100; error=0.021
epoch 3150; error=0.02
epoch 3200; error=0.02
epoch 3250; error=0.019
epoch 3300; error=0.019
epoch 3350; error=0.019
epoch 3400; error=0.018
epoch 3450; error=0.018
epoch 3500; error=0.018
epoch 3550; error=0.018
epoch 3600; error=0.017
epoch 3650; error=0.017
epoch 3700; error=0.017
epoch 3750; error=0.016
epoch 3800; error=0.016
epoch 3850; error=0.016
epoch 3900; error=0.016
epoch 3950; error=0.016
epoch 4000; error=0.015
epoch 4050; error=0.015
epoch 4100; error=0.015
epoch 4150; error=0.015
epoch 4200; error=0.014
epoch 4250; error=0.014
epoch 4300; error=0.014
epoch 4350; error=0.014
epoch 4400; error=0.014
epoch 4450; error=0.014
epoch 4500; error=0.013
epoch 4550; error=0.013
epoch 4600; error=0.013
epoch 4650; error=0.013
epoch 4700; error=0.013
epoch 4750; error=0.013
epoch 4800; error=0.012
epoch 4850; error=0.012
epoch 4900; error=0.012
epoch 4950; error=0.012
Final result:
 Input: [0. 0.]
 Output: 0.082
 Target: 0.0

Final result:
 Input: [0. 1.]
 Output: 0.949
 Target: 1.0

Final result:
 Input: [1. 0.]
 Output: 0.949
 Target: 1.0

Final result:
 Input: [1. 1.]
 Output: 1.0
 Target: 1.0
```



```
tensor([5.3485, 5.3493], requires_grad=True) tensor([-2.4191], requires_grad=Tru
e)
```

Now let's change the activation function to "linear" (identity function) from the "logistic" function, such that $g(\mathbf{net}) = \mathbf{net}$. With a linear rather than logistic activation, the output will no longer be constrained between 0 and 1. The artificial neuron will still try to solve the problem with 0/1 targets. Here is the simple implementation of $g(\cdot)$:

In [ ]:
```
def g_linear(x):
    return x
```

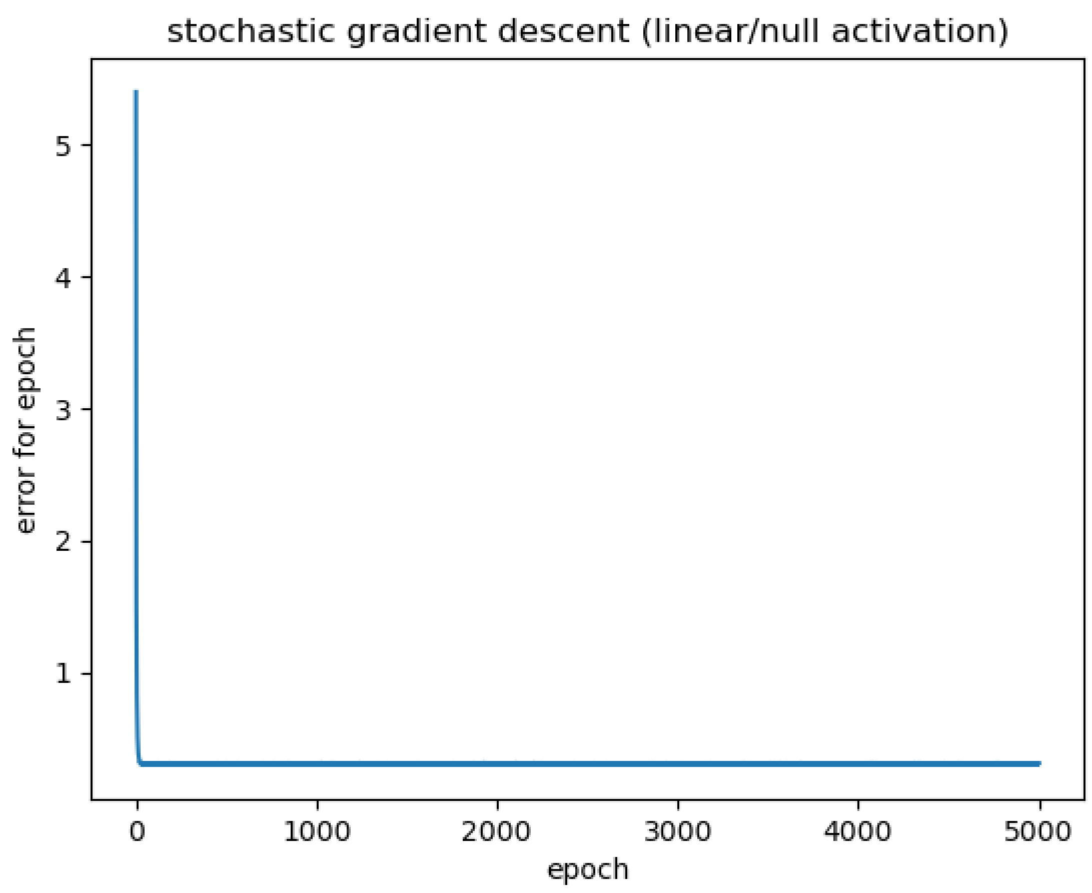
Problem 3 (5 points)

Just as before, fill in the missing code fragments for implementing gradient descent. This time we are using the linear activation function. Be sure to change your gradient calculation to reflect the new activation function.

In [ ]:
```
# Initialize parameters
#     Although you will implement gradient descent manually, let's set requires_
#     anyway so PyTorch will track the gradient too, and we can compare your gra
w = torch.randn(2, requires_grad=True) # [size 2] tensor
b = torch.randn(1, requires_grad=True) # [size 1] tensor

alpha = 0.05 # learning rate
nepochs = 5000 # number of epochs
```

```python
track_error = []
verbose = True
for e in range(nepochs): # for each epoch
    error_epoch = 0. # sum loss across the epoch
    perm = np.random.permutation(N)
    for p in perm: # visit data points in random order
        x_pat = X[p,:] # get one input pattern

        # compute output of neuron
        net = torch.dot(x_pat,w)+b
        yhat = g_linear(net)

        # compute loss
        y = Y_or[p]
        myloss = loss(yhat,y)
        error_epoch += myloss.item()

        # Compute the gradient manually
        if verbose:
            print('Compute the gradient manually')
            print_forward(x_pat,yhat,y)
        with torch.no_grad():
            w_grad = 2 * (yhat - y) * x_pat
            b_grad = 2 * (yhat - y)
        if verbose: print_grad(w_grad.numpy(),b_grad.numpy())

        # Compute the gradient with PyTorch and compre with manual values
        if verbose: print('Compute the gradient using PyTorch .backward()')
        myloss.backward()
        if verbose:
            print_grad(w.grad.numpy(),b.grad.numpy())
            print("")
        w.grad.zero_() # clear PyTorch's gradient
        b.grad.zero_()

        # Parameter update with gradient descent
        with torch.no_grad():
            w -= alpha * w_grad
            b -= alpha * b_grad

    if verbose==True: verbose=False
    track_error.append(error_epoch)
    if e % 50 == 0:
        print("epoch " + str(e) + "; error=" +str(round(error_epoch,3)))

# print a final pass through patterns
for p in range(X.shape[0]):
    x_pat = X[p]
    net = torch.dot(x_pat,w)+b
    yhat = g_linear(net)
    y = Y_or[p]
    print("Final result:")
    print_forward(x_pat,yhat,y)
    print("")

# track output of gradient descent
plt.figure()
plt.clf()
plt.plot(track_error)
plt.title('stochastic gradient descent (linear/null activation)')
plt.ylabel('error for epoch')
plt.xlabel('epoch')
plt.show()

# Print out the learned weights and bias
print(w, b)
```

```
Compute the gradient manually
 Input: [0. 1.]
 Output: -1.176
 Target: 1.0
  d_loss / d_w = [-0.        -4.3526096]
  d_loss / d_b = [-4.3526096]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [-0.        -4.3526096]
  d_loss / d_b = [-4.3526096]

Compute the gradient manually
 Input: [0. 0.]
 Output: -0.206
 Target: 0.0
  d_loss / d_w = [-0. -0.]
  d_loss / d_b = [-0.41225305]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [0. 0.]
  d_loss / d_b = [-0.41225305]

Compute the gradient manually
 Input: [1. 1.]
 Output: 0.219
 Target: 1.0
  d_loss / d_w = [-1.561613 -1.561613]
  d_loss / d_b = [-1.561613]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [-1.561613 -1.561613]
  d_loss / d_b = [-1.561613]

Compute the gradient manually
 Input: [1. 0.]
 Output: 0.91
 Target: 1.0
  d_loss / d_w = [-0.179456 -0.      ]
  d_loss / d_b = [-0.179456]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [-0.179456  0.      ]
  d_loss / d_b = [-0.179456]

epoch 0; error=5.397
epoch 50; error=0.307
epoch 100; error=0.305
epoch 150; error=0.304
epoch 200; error=0.308
epoch 250; error=0.306
epoch 300; error=0.3
epoch 350; error=0.297
epoch 400; error=0.3
epoch 450; error=0.305
epoch 500; error=0.302
epoch 550; error=0.305
epoch 600; error=0.297
epoch 650; error=0.307
epoch 700; error=0.31
epoch 750; error=0.303
epoch 800; error=0.306
epoch 850; error=0.303
epoch 900; error=0.308
epoch 950; error=0.306
epoch 1000; error=0.3
epoch 1050; error=0.31
epoch 1100; error=0.302
epoch 1150; error=0.303
epoch 1200; error=0.303
epoch 1250; error=0.308
epoch 1300; error=0.307
epoch 1350; error=0.302
epoch 1400; error=0.308
epoch 1450; error=0.308
epoch 1500; error=0.309
epoch 1550; error=0.297
epoch 1600; error=0.31
epoch 1650; error=0.307
epoch 1700; error=0.301
epoch 1750; error=0.301
epoch 1800; error=0.3
epoch 1850; error=0.309
epoch 1900; error=0.306
epoch 1950; error=0.309
epoch 2000; error=0.31
epoch 2050; error=0.3
epoch 2100; error=0.306
epoch 2150; error=0.307
epoch 2200; error=0.307
epoch 2250; error=0.311
epoch 2300; error=0.303
epoch 2350; error=0.307
epoch 2400; error=0.307
epoch 2450; error=0.3
epoch 2500; error=0.299
epoch 2550; error=0.308
epoch 2600; error=0.309
epoch 2650; error=0.306
epoch 2700; error=0.308
epoch 2750; error=0.305
epoch 2800; error=0.307
epoch 2850; error=0.308
epoch 2900; error=0.307
epoch 2950; error=0.304
epoch 3000; error=0.306
epoch 3050; error=0.304
epoch 3100; error=0.308
epoch 3150; error=0.308
epoch 3200; error=0.305
epoch 3250; error=0.3
epoch 3300; error=0.302
epoch 3350; error=0.308
epoch 3400; error=0.307
epoch 3450; error=0.3
epoch 3500; error=0.307
epoch 3550; error=0.3
epoch 3600; error=0.308
epoch 3650; error=0.304
epoch 3700; error=0.305
epoch 3750; error=0.303
epoch 3800; error=0.305
epoch 3850; error=0.296
epoch 3900; error=0.299
epoch 3950; error=0.307
epoch 4000; error=0.305
epoch 4050; error=0.307
epoch 4100; error=0.311
epoch 4150; error=0.308
epoch 4200; error=0.3
epoch 4250; error=0.296
epoch 4300; error=0.306
epoch 4350; error=0.307
epoch 4400; error=0.307
epoch 4450; error=0.308
epoch 4500; error=0.306
epoch 4550; error=0.311
epoch 4600; error=0.305
epoch 4650; error=0.3
epoch 4700; error=0.308
epoch 4750; error=0.308
epoch 4800; error=0.307
epoch 4850; error=0.307
epoch 4900; error=0.305
epoch 4950; error=0.307
Final result:
 Input: [0. 0.]
 Output: 0.247
 Target: 0.0

Final result:
 Input: [0. 1.]
 Output: 0.718
 Target: 1.0

Final result:
 Input: [1. 0.]
 Output: 0.732
 Target: 1.0

Final result:
 Input: [1. 1.]
 Output: 1.203
 Target: 1.0
```



```
tensor([0.4851, 0.4713], requires_grad=True) tensor([0.2466], requires_grad=True)
```

Problem 4 (10 points)

You'll see above that the artificial neuron, with the simple linear (identity) activation, does worse on the OR problem. Examine the learned weights and bias, and explain why the network does not arrive at a perfect solution.

My Answer:

In the sigmoid section, the learned weights are both about 5.35 and the bias is -2.42.

This leads to the cases:

x_1	x_2	net	$g(net) =$ output
1	1	8.28	~ 1.0
1	0	5.35	0.95
0	1	5.35	0.95
0	0	-2.42	0.08

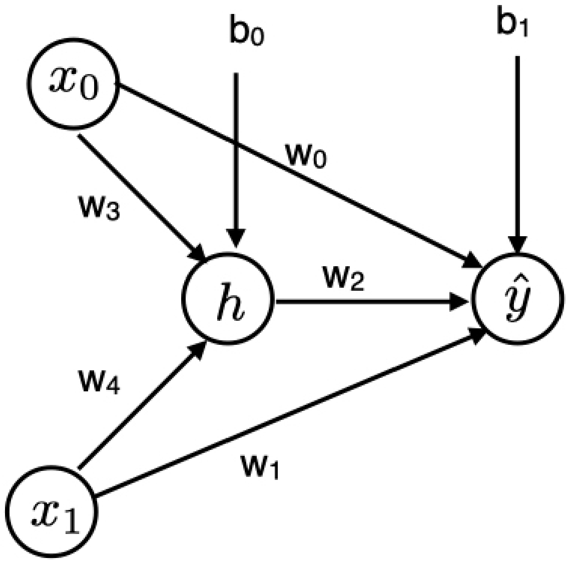
Meanwhile, in the linear section, the learned weights are both about 0.48 and the bias is 0.25.

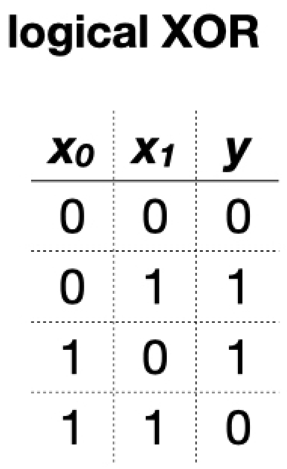
This leads to the cases:

x_1	x_2		$net =$ output
	1	1	1.2
	1	0	0.73
	0	1	0.73
	0	0	0.25

Without the non-linear activation function, sigmoid, it is difficult for the model to spread out the output to the different ends of 1 and 0.

In the next part, we have a simple multi-layer network with two input neurons, one hidden neuron, and one output neuron. Both the hidden and output unit should use the logistic activation function. We will learn to compute logical XOR. The network and logical XOR are shown below, for inputs x_0 and x_1 and target output y.





logical XOR

x_0	x_1	y
0	0	0
0	1	1
1	0	1
1	1	0

Problem 5 (15 points)

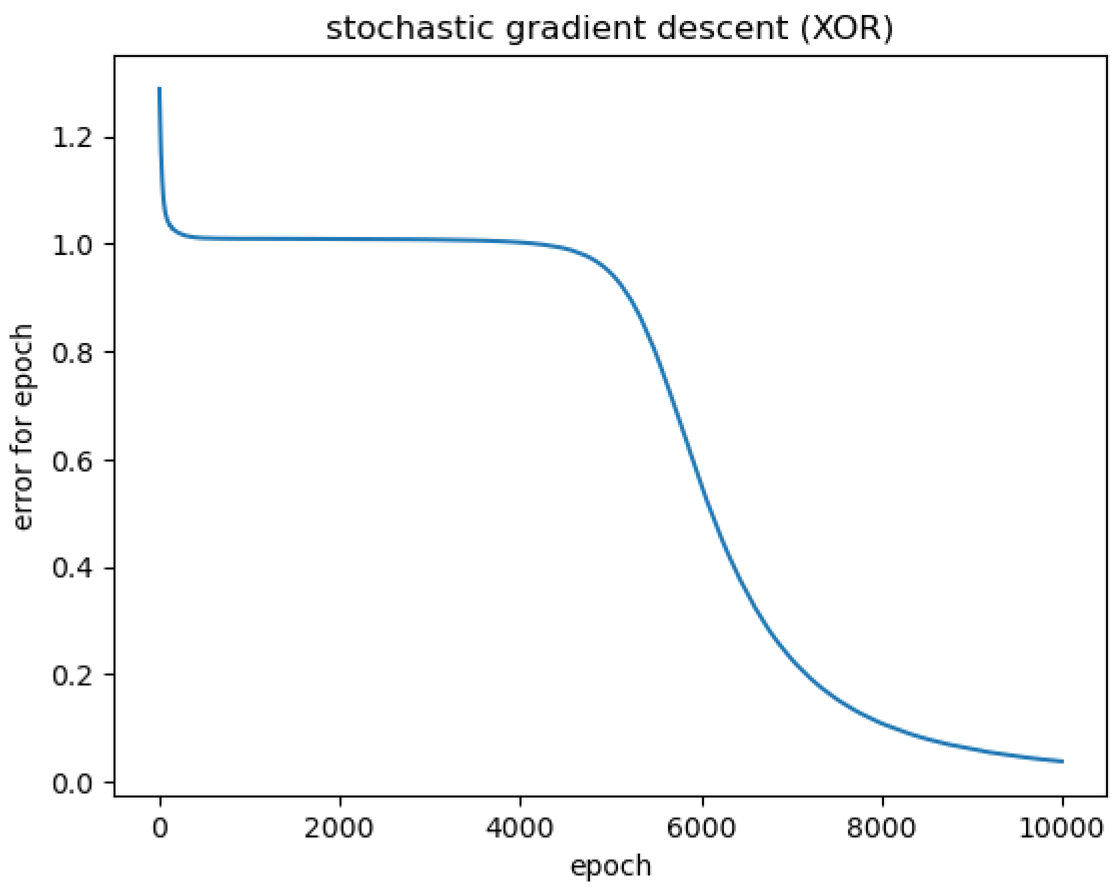
You will implement backpropagation for this simple network. In the code below, you have several parts to fill in. First, define the forward pass to compute the output \`yhat\` from the input \`x\`. Second, fill in code to manually compute the gradients for all five weights w and two biases b in closed form. Third, fill in the code for updating the biases and weights.

After completing the code, run it to compare **your gradients** with the **ground-truth computed by PyTorch.** (There may be small differences that you shouldn't worry about, e.g. within 1e-6). Also, you can check the network's performance at the end of training.

In [ ]:
```
# Same input tensor X and new labels y for xor
Y_xor = torch.tensor([0.,1.,1.,0.])
N = X.shape[0] # number of input patterns

# Initialize parameters
#     Although you will implement gradient descent manually, let's set requires_
#     anyway so PyTorch will track the gradient too, and we can compare your gra
w_34 = torch.randn(2,requires_grad=True) # [size 2] tensor representing [w_3,w_4
w_012 = torch.randn(3,requires_grad=True) # [size 3] tensor representing [w_0,w_
b_0 = torch.randn(1,requires_grad=True) # [size 1] tensor
b_1 = torch.randn(1,requires_grad=True) # [size 1] tensor

alpha = 0.05 # learning rate
nepochs = 10000 # number of epochs

track_error = []
verbose = True
for e in range(nepochs): # for each epoch
    error_epoch = 0. # sum loss across the epoch
    perm = np.random.permutation(N)
    for p in perm: # visit data points in random order
        x_pat = X[p,:] # input pattern

        # Compute the output of hidden neuron h
        # e.g., two lines like the following
        net_h = torch.dot(x_pat, w_34) + b_0
        h = g_logistic(net_h)

        # Compute the output of neuron yhat
        # e.g., two lines like the following
        net_y = torch.dot(torch.cat([x_pat, h]), w_012) + b_1
        yhat = g_logistic(net_y)

        # compute loss
        y = Y_xor[p]
        myloss = loss(yhat,y)
        error_epoch += myloss.item()

        # print output if this is the last epoch
        if (e == nepochs-1):
            print("Final result:")
            print_forward(x_pat,yhat,y)
            print(f" net_h: {round(net_h.detach().numpy()[0], 2):5.2f}")
            print(f" h    : {round(    h.detach().numpy()[0], 2):5.2f}")
            print(f" net_y: {round(net_y.detach().numpy()[0], 2):5.2f}")
            print(f" yhat : {round( yhat.detach().numpy()[0], 2):5.2f}")
            print("")

        # Compute the gradient manually
        if verbose:
            print('Compute the gradient manually')
            print_forward(x_pat,yhat,y)
        with torch.no_grad():
            w_34_grad  = 2 * (yhat - y) * yhat * (1 - yhat) * w_012[-1] * h * (1
            b_0_grad   = 2 * (yhat - y) * yhat * (1 - yhat) * w_012[-1] * h * (1
            w_012_grad = 2 * (yhat - y) * g_logistic(net) * (1 - g_logistic(net)
            b_1_grad   = 2 * (yhat - y) * g_logistic(net) * (1 - g_logistic(net)
        if verbose:
            print(" Grad for w_34 and b_0")
            print_grad(w_34_grad.numpy(),b_0_grad.numpy())
            print(" Grad for w_012 and b_1")
            print_grad(w_012_grad.numpy(),b_1_grad.numpy())
            print("")

        # Compute the gradient with PyTorch and compre with manual values
        if verbose: print('Compute the gradient using PyTorch .backward()')
        myloss.backward()
        if verbose:
            print(" Grad for w_34 and b_0")
            print_grad(w_34.grad.numpy(),b_0.grad.numpy())
            print(" Grad for w_012 and b_1")
            print_grad(w_012.grad.numpy(),b_1.grad.numpy())
            print("")
        w_34.grad.zero_() # clear PyTorch's gradient
        b_0.grad.zero_()
        w_012.grad.zero_()
        b_1.grad.zero_()

        # Parameter update with gradient descent
        with torch.no_grad():
            w_34  -= alpha * w_34_grad
            b_0   -= alpha * b_0_grad
            w_012 -= alpha * w_012_grad
            b_1   -= alpha * b_1_grad

    if verbose==True: verbose=False
    track_error.append(error_epoch)
    if e % 50 == 0:
        print("epoch " + str(e) + "; error=" +str(round(error_epoch,3)))

# track output of gradient descent
plt.figure()
plt.clf()
plt.plot(track_error)
plt.title('stochastic gradient descent (XOR)')
plt.ylabel('error for epoch')
plt.xlabel('epoch')
plt.show()
```

```
Compute the gradient manually
 Input: [1. 1.]
 Output: 0.862
 Target: 0.0
 Grad for w_34 and b_0
  d_loss / d_w = [-0.00505043 -0.00505043]
  d_loss / d_b = [-0.00505043]
 Grad for w_012 and b_1
  d_loss / d_w = [0.30632773 0.30632773 0.11987449]
  d_loss / d_b = [0.30632773]

Compute the gradient using PyTorch .backward()
 Grad for w_34 and b_0
  d_loss / d_w = [-0.00505043 -0.00505043]
  d_loss / d_b = [-0.00505043]
 Grad for w_012 and b_1
  d_loss / d_w = [0.20472315 0.20472315 0.08011381]
  d_loss / d_b = [0.20472315]

Compute the gradient manually
 Input: [0. 0.]
 Output: 0.617
 Target: 0.0
 Grad for w_34 and b_0
  d_loss / d_w = [-0. -0.]
  d_loss / d_b = [-0.007598]
 Grad for w_012 and b_1
  d_loss / d_w = [0.         0.         0.08538812]
  d_loss / d_b = [0.21916655]

Compute the gradient using PyTorch .backward()
 Grad for w_34 and b_0
  d_loss / d_w = [0. 0.]
  d_loss / d_b = [-0.007598]
 Grad for w_012 and b_1
  d_loss / d_w = [0.         0.         0.11361026]
  d_loss / d_b = [0.29160458]

Compute the gradient manually
 Input: [1. 0.]
 Output: 0.623
 Target: 1.0
 Grad for w_34 and b_0
  d_loss / d_w = [0.0049874 0.       ]
  d_loss / d_b = [0.0049874]
 Grad for w_012 and b_1
  d_loss / d_w = [-0.1339767  -0.         -0.07401012]
  d_loss / d_b = [-0.1339767]

Compute the gradient using PyTorch .backward()
 Grad for w_34 and b_0
  d_loss / d_w = [0.0049874 0.       ]
  d_loss / d_b = [0.0049874]
 Grad for w_012 and b_1
  d_loss / d_w = [-0.1771989   0.         -0.09788651]
  d_loss / d_b = [-0.1771989]

Compute the gradient manually
 Input: [0. 1.]
 Output: 0.851
 Target: 1.0
 Grad for w_34 and b_0
  d_loss / d_w = [0.         0.00078063]
  d_loss / d_b = [0.00078063]
 Grad for w_012 and b_1
  d_loss / d_w = [-0.         -0.05295912 -0.0132274 ]
  d_loss / d_b = [-0.05295912]

Compute the gradient using PyTorch .backward()
 Grad for w_34 and b_0
  d_loss / d_w = [0.         0.00078063]
  d_loss / d_b = [0.00078063]
 Grad for w_012 and b_1
  d_loss / d_w = [ 0.         -0.03782633 -0.00944774]
  d_loss / d_b = [-0.03782633]

epoch 0; error=1.289
epoch 50; error=1.076
epoch 100; error=1.039
epoch 150; error=1.028
epoch 200; error=1.022
epoch 250; error=1.018
epoch 300; error=1.015
epoch 350; error=1.013
epoch 400; error=1.012
epoch 450; error=1.011
epoch 500; error=1.011
epoch 550; error=1.011
epoch 600; error=1.01
epoch 650; error=1.01
epoch 700; error=1.01
epoch 750; error=1.01
epoch 800; error=1.01
epoch 850; error=1.01
epoch 900; error=1.01
epoch 950; error=1.01
epoch 1000; error=1.01
epoch 1050; error=1.01
epoch 1100; error=1.01
epoch 1150; error=1.01
epoch 1200; error=1.01
epoch 1250; error=1.01
epoch 1300; error=1.01
epoch 1350; error=1.01
epoch 1400; error=1.01
epoch 1450; error=1.01
epoch 1500; error=1.01
epoch 1550; error=1.01
epoch 1600; error=1.01
epoch 1650; error=1.01
epoch 1700; error=1.01
epoch 1750; error=1.009
epoch 1800; error=1.009
epoch 1850; error=1.009
epoch 1900; error=1.009
epoch 1950; error=1.009
epoch 2000; error=1.009
epoch 2050; error=1.009
epoch 2100; error=1.009
epoch 2150; error=1.009
epoch 2200; error=1.009
epoch 2250; error=1.009
epoch 2300; error=1.009
epoch 2350; error=1.009
epoch 2400; error=1.009
epoch 2450; error=1.009
epoch 2500; error=1.009
epoch 2550; error=1.009
epoch 2600; error=1.009
epoch 2650; error=1.009
epoch 2700; error=1.009
epoch 2750; error=1.009
epoch 2800; error=1.009
epoch 2850; error=1.008
epoch 2900; error=1.008
epoch 2950; error=1.008
epoch 3000; error=1.008
epoch 3050; error=1.008
epoch 3100; error=1.008
epoch 3150; error=1.008
epoch 3200; error=1.008
epoch 3250; error=1.008
epoch 3300; error=1.007
epoch 3350; error=1.007
epoch 3400; error=1.007
epoch 3450; error=1.007
epoch 3500; error=1.007
epoch 3550; error=1.006
epoch 3600; error=1.006
epoch 3650; error=1.006
epoch 3700; error=1.006
epoch 3750; error=1.005
epoch 3800; error=1.005
epoch 3850; error=1.004
epoch 3900; error=1.004
epoch 3950; error=1.004
epoch 4000; error=1.003
epoch 4050; error=1.002
epoch 4100; error=1.002
epoch 4150; error=1.001
epoch 4200; error=1.0
epoch 4250; error=0.999
epoch 4300; error=0.998
epoch 4350; error=0.996
epoch 4400; error=0.995
epoch 4450; error=0.993
epoch 4500; error=0.991
epoch 4550; error=0.989
epoch 4600; error=0.986
epoch 4650; error=0.983
epoch 4700; error=0.98
epoch 4750; error=0.976
epoch 4800; error=0.971
epoch 4850; error=0.966
epoch 4900; error=0.96
epoch 4950; error=0.954
epoch 5000; error=0.946
epoch 5050; error=0.937
epoch 5100; error=0.927
epoch 5150; error=0.915
epoch 5200; error=0.903
epoch 5250; error=0.888
epoch 5300; error=0.873
epoch 5350; error=0.855
epoch 5400; error=0.837
epoch 5450; error=0.817
epoch 5500; error=0.796
epoch 5550; error=0.773
epoch 5600; error=0.75
epoch 5650; error=0.726
epoch 5700; error=0.701
epoch 5750; error=0.677
epoch 5800; error=0.651
epoch 5850; error=0.627
epoch 5900; error=0.602
epoch 5950; error=0.577
epoch 6000; error=0.553
epoch 6050; error=0.53
epoch 6100; error=0.507
epoch 6150; error=0.485
epoch 6200; error=0.464
epoch 6250; error=0.444
epoch 6300; error=0.424
epoch 6350; error=0.406
epoch 6400; error=0.387
epoch 6450; error=0.37
epoch 6500; error=0.354
epoch 6550; error=0.338
epoch 6600; error=0.324
epoch 6650; error=0.309
epoch 6700; error=0.296
epoch 6750; error=0.284
epoch 6800; error=0.271
epoch 6850; error=0.26
epoch 6900; error=0.249
epoch 6950; error=0.239
epoch 7000; error=0.229
epoch 7050; error=0.219
epoch 7100; error=0.211
epoch 7150; error=0.202
epoch 7200; error=0.194
epoch 7250; error=0.187
epoch 7300; error=0.18
epoch 7350; error=0.173
epoch 7400; error=0.166
epoch 7450; error=0.16
epoch 7500; error=0.154
epoch 7550; error=0.149
epoch 7600; error=0.143
epoch 7650; error=0.138
epoch 7700; error=0.133
epoch 7750; error=0.129
epoch 7800; error=0.124
epoch 7850; error=0.12
epoch 7900; error=0.116
epoch 7950; error=0.112
epoch 8000; error=0.109
epoch 8050; error=0.105
epoch 8100; error=0.102
epoch 8150; error=0.099
epoch 8200; error=0.095
epoch 8250; error=0.093
epoch 8300; error=0.09
epoch 8350; error=0.087
epoch 8400; error=0.084
epoch 8450; error=0.082
epoch 8500; error=0.08
epoch 8550; error=0.077
epoch 8600; error=0.075
epoch 8650; error=0.073
epoch 8700; error=0.071
epoch 8750; error=0.069
epoch 8800; error=0.067
epoch 8850; error=0.065
epoch 8900; error=0.064
epoch 8950; error=0.062
epoch 9000; error=0.06
epoch 9050; error=0.059
epoch 9100; error=0.057
epoch 9150; error=0.056
epoch 9200; error=0.055
epoch 9250; error=0.053
epoch 9300; error=0.052
epoch 9350; error=0.051
epoch 9400; error=0.049
epoch 9450; error=0.048
epoch 9500; error=0.047
epoch 9550; error=0.046
epoch 9600; error=0.045
epoch 9650; error=0.044
epoch 9700; error=0.043
epoch 9750; error=0.042
epoch 9800; error=0.041
epoch 9850; error=0.04
epoch 9900; error=0.039
epoch 9950; error=0.039
Final result:
 Input: [0. 1.]
 Output: 0.882
 Target: 1.0
 net_h: -9.85
 h    :  0.00
 net_y:  2.01
 yhat :  0.88

Final result:
 Input: [1. 0.]
 Output: 0.936
 Target: 1.0
 net_h:  2.34
 h    :  0.91
 net_y:  2.68
 yhat :  0.94

Final result:
 Input: [1. 1.]
 Output: 0.092
 Target: 0.0
 net_h: -3.88
 h    :  0.02
```

```
     net_y: -2.29
     yhat :  0.09

    Final result:
     Input: [0. 0.]
     Output: 0.105
     Target: 0.0
     net_h: -3.63
     h    :  0.03
     net_y: -2.14
     yhat :  0.11
```



Problem 6 (10 points)

After running your XOR network, print the values of the learned weights and biases. Your job now is to describe the solution that the network has learned. How does it work? Walk through each input pattern to describe how the network computes the right answer (if it does). See discussion in lecture for an example.

In [ ]:
```python
print(f"w_012: {w_012.detach().numpy()}")
print(f"w_34 : { w_34.detach().numpy()}")
print(f"b_0  :  { b_0.detach().numpy()[0]}")
print(f"b_1  :  { b_1.detach().numpy()[0]}")
```

```
w_012: [-4.51507    4.423488  10.5274935]
w_34 : [ 5.9690337 -6.224827 ]
b_0  :  -3.626152753829956
b_1  :  -2.4166808128356934
```

My Answer:

The cases are:

x_0	x_1	net_h	$h = g(net_h)$	net_y	$\hat{y} =$ output
1	1	-3.88	0.02	-2.29	0.09
1	0	2.34	0.91	2.68	0.94
0	1	-9.85	0.00	2.01	0.88
0	0	-3.63	0.03	-2.14	0.11

The learned mechanism is:

The hidden neuron h is only activated when (x_0, x_1) is (1, 0).

First, When (x_0, x_1) is (1, 1) or (0, 0), the sum of w_0x_0, w_1x_1, and h will be zero, which leads to negative net_y.

Second, When (x_0, x_1) is (1, 0), h is activated so the net_y can be positive.

Last, When (x_0, x_1) is (0, 1), net_y is positive because w_1x_1 is positive and way greater than the bias.