Homework-NeuralNet-B

February 15, 2024

1 Homework - Neural networks - Part B (55 points)

1.1 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.

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.

[1]:
```
# 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.

[2]:
```
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 (**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.

[3]:
```
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.

[4]:
```
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.

[5]:
```
# Initialize parameters
#     Although you will implement gradient descent manually, let's set␣
 ↪requires_grad=True
#     anyway so PyTorch will track the gradient too, and we can compare your␣
 ↪gradient with PyTorch's.
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

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) * yhat * (1 - yhat) * x_pat)    # ([size␣
↪2] PyTorch tensor)
            b_grad = (2 * (yhat - y) * yhat * (1 - yhat))    # ([size 1]␣
↪PyTorch tensor)
            #  make sure to inclose your code in the "with torch.no_grad()"␣
↪wrapper,
            #   otherwise PyTorch will try to track the "gradient" of the␣
↪gradient computation, which we don't want.
        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()
```

```
Compute the gradient manually
 Input: [0. 1.]
 Output: 0.844
 Target: 1.0
  d_loss / d_w = [-0.         -0.04106285]
  d_loss / d_b = [-0.04106285]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [-0.         -0.04106285]
  d_loss / d_b = [-0.04106285]

Compute the gradient manually
 Input: [1. 1.]
 Output: 0.972
 Target: 1.0
  d_loss / d_w = [-0.00155091 -0.00155091]
  d_loss / d_b = [-0.00155091]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [-0.00155091 -0.00155091]
  d_loss / d_b = [-0.00155091]

Compute the gradient manually
 Input: [1. 0.]
 Output: 0.913
 Target: 1.0
  d_loss / d_w = [-0.01374633 -0.        ]
  d_loss / d_b = [-0.01374633]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [-0.01374634  0.        ]
  d_loss / d_b = [-0.01374634]

Compute the gradient manually
```

```
 Input: [0. 0.]
 Output: 0.625
 Target: 0.0
  d_loss / d_w = [0. 0.]
  d_loss / d_b = [0.29290763]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [0. 0.]
  d_loss / d_b = [0.29290763]

epoch 0; error=0.423
epoch 50; error=0.316
epoch 100; error=0.261
epoch 150; error=0.226
epoch 200; error=0.199
epoch 250; error=0.178
epoch 300; error=0.16
epoch 350; error=0.145
epoch 400; error=0.132
epoch 450; error=0.122
epoch 500; error=0.112
epoch 550; error=0.104
epoch 600; error=0.097
epoch 650; error=0.09
epoch 700; error=0.085
epoch 750; error=0.08
epoch 800; error=0.075
epoch 850; error=0.071
epoch 900; error=0.067
epoch 950; error=0.064
epoch 1000; error=0.061
epoch 1050; error=0.058
epoch 1100; error=0.055
epoch 1150; error=0.053
epoch 1200; error=0.051
epoch 1250; error=0.049
epoch 1300; error=0.047
epoch 1350; error=0.045
epoch 1400; error=0.044
epoch 1450; error=0.042
epoch 1500; error=0.041
epoch 1550; error=0.039
epoch 1600; error=0.038
epoch 1650; error=0.037
epoch 1700; error=0.036
epoch 1750; error=0.035
epoch 1800; error=0.034
epoch 1850; error=0.033
epoch 1900; error=0.032
```

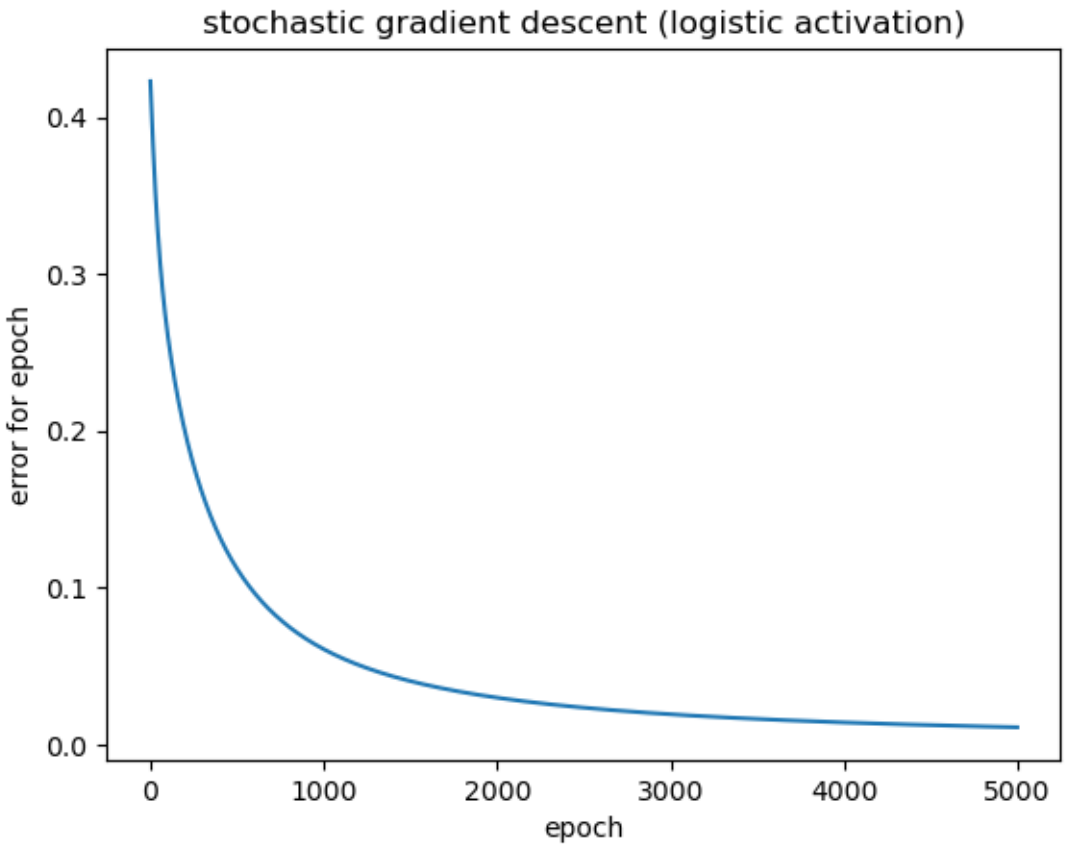
```
epoch 1950; error=0.031
epoch 2000; error=0.03
epoch 2050; error=0.029
epoch 2100; error=0.028
epoch 2150; error=0.028
epoch 2200; error=0.027
epoch 2250; error=0.026
epoch 2300; error=0.026
epoch 2350; error=0.025
epoch 2400; error=0.025
epoch 2450; error=0.024
epoch 2500; error=0.024
epoch 2550; error=0.023
epoch 2600; error=0.023
epoch 2650; error=0.022
epoch 2700; error=0.022
epoch 2750; error=0.021
epoch 2800; error=0.021
epoch 2850; error=0.021
epoch 2900; error=0.02
epoch 2950; error=0.02
epoch 3000; error=0.019
epoch 3050; error=0.019
epoch 3100; error=0.019
epoch 3150; error=0.018
epoch 3200; error=0.018
epoch 3250; error=0.018
epoch 3300; error=0.018
epoch 3350; error=0.017
epoch 3400; error=0.017
epoch 3450; error=0.017
epoch 3500; error=0.016
epoch 3550; error=0.016
epoch 3600; error=0.016
epoch 3650; error=0.016
epoch 3700; error=0.015
epoch 3750; error=0.015
epoch 3800; error=0.015
epoch 3850; error=0.015
epoch 3900; error=0.015
epoch 3950; error=0.014
epoch 4000; error=0.014
epoch 4050; error=0.014
epoch 4100; error=0.014
epoch 4150; error=0.014
epoch 4200; error=0.014
epoch 4250; error=0.013
epoch 4300; error=0.013
```

```
epoch 4350; error=0.013
epoch 4400; error=0.013
epoch 4450; error=0.013
epoch 4500; error=0.013
epoch 4550; error=0.012
epoch 4600; error=0.012
epoch 4650; error=0.012
epoch 4700; error=0.012
epoch 4750; error=0.012
epoch 4800; error=0.012
epoch 4850; error=0.012
epoch 4900; error=0.011
epoch 4950; error=0.011
Final result:
 Input: [0. 0.]
 Output: 0.079
 Target: 0.0

Final result:
 Input: [0. 1.]
 Output: 0.951
 Target: 1.0

Final result:
 Input: [1. 0.]
 Output: 0.951
 Target: 1.0

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



[6]:
```
print(w, b)
```

```
tensor([5.4095, 5.4083], requires_grad=True) tensor([-2.4499],
requires_grad=True)
```

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 $g(\cdot)$:

[7]:
```
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.

[8]:
```
# Initialize parameters
#     Although you will implement gradient descent manually, let's set␣
 ↪requires_grad=True
```

```
#     anyway so PyTorch will track the gradient too, and we can compare your␣
 ↪gradient with PyTorch's.
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

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)    # ([size 2] PyTorch tensor)
            b_grad = (2 * (yhat - y))            # ([size 1] PyTorch tensor)
            #  make sure to inclose your code in the "with torch.no_grad()"␣
 ↪wrapper,
            #   otherwise PyTorch will try to track the "gradient" of the␣
 ↪gradient computation, which we don't want.
        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()
```

```
Compute the gradient manually
 Input: [1. 0.]
 Output: -1.769
 Target: 1.0
  d_loss / d_w = [-5.538352 -0.      ]
  d_loss / d_b = [-5.538352]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [-5.538352 -0.      ]
  d_loss / d_b = [-5.538352]

Compute the gradient manually
 Input: [0. 1.]
 Output: -0.56
 Target: 1.0
  d_loss / d_w = [-0.       -3.119256]
  d_loss / d_b = [-3.119256]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [ 0.       -3.119256]
```

```
  d_loss / d_b = [-3.119256]

Compute the gradient manually
 Input: [1. 1.]
 Output: -0.844
 Target: 1.0
  d_loss / d_w = [-3.6884487 -3.6884487]
  d_loss / d_b = [-3.6884487]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [-3.6884487 -3.6884487]
  d_loss / d_b = [-3.6884487]

Compute the gradient manually
 Input: [0. 0.]
 Output: -0.278
 Target: 0.0
  d_loss / d_w = [-0. -0.]
  d_loss / d_b = [-0.5568675]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [0. 0.]
  d_loss / d_b = [-0.5568675]

epoch 0; error=13.579
epoch 50; error=0.307
epoch 100; error=0.307
epoch 150; error=0.309
epoch 200; error=0.305
epoch 250; error=0.301
epoch 300; error=0.302
epoch 350; error=0.3
epoch 400; error=0.307
epoch 450; error=0.296
epoch 500; error=0.3
epoch 550; error=0.307
epoch 600; error=0.306
epoch 650; error=0.307
epoch 700; error=0.312
epoch 750; error=0.306
epoch 800; error=0.308
epoch 850; error=0.3
epoch 900; error=0.308
epoch 950; error=0.306
epoch 1000; error=0.3
epoch 1050; error=0.301
epoch 1100; error=0.306
epoch 1150; error=0.306
epoch 1200; error=0.306
epoch 1250; error=0.306
```

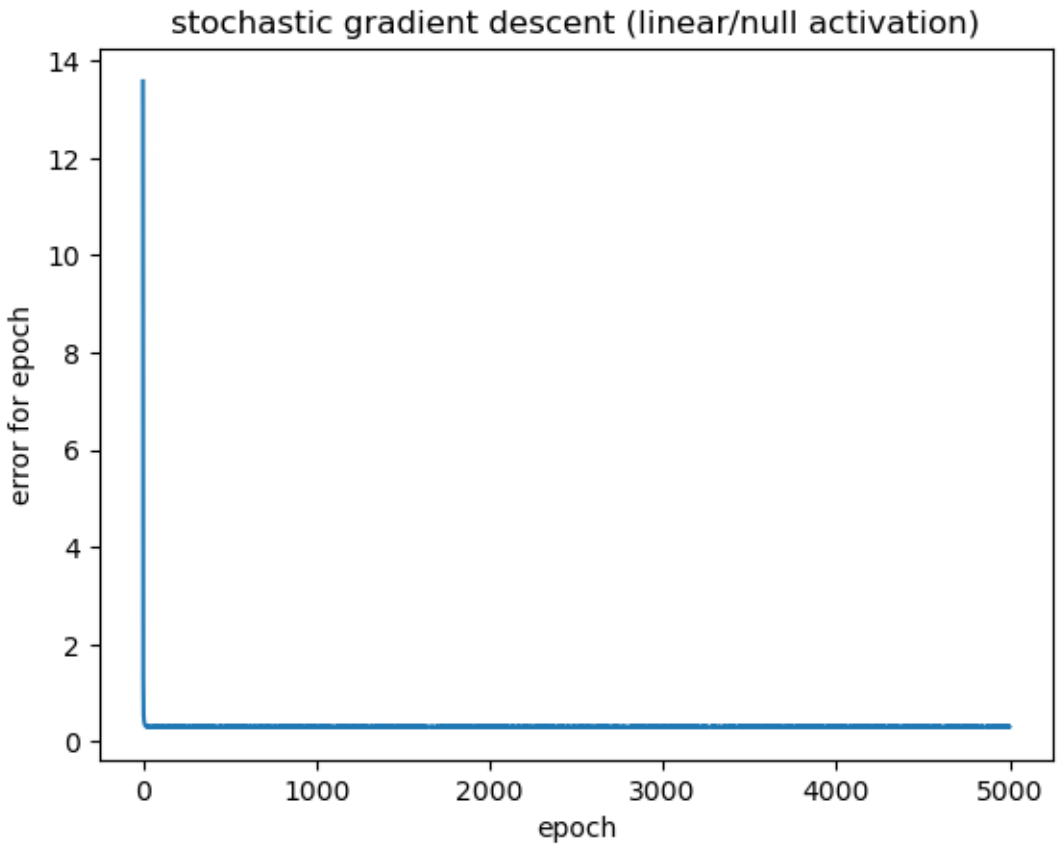
```
epoch 1300; error=0.308
epoch 1350; error=0.301
epoch 1400; error=0.306
epoch 1450; error=0.304
epoch 1500; error=0.308
epoch 1550; error=0.3
epoch 1600; error=0.306
epoch 1650; error=0.306
epoch 1700; error=0.3
epoch 1750; error=0.308
epoch 1800; error=0.301
epoch 1850; error=0.303
epoch 1900; error=0.311
epoch 1950; error=0.306
epoch 2000; error=0.308
epoch 2050; error=0.307
epoch 2100; error=0.304
epoch 2150; error=0.301
epoch 2200; error=0.296
epoch 2250; error=0.308
epoch 2300; error=0.308
epoch 2350; error=0.303
epoch 2400; error=0.306
epoch 2450; error=0.299
epoch 2500; error=0.304
epoch 2550; error=0.308
epoch 2600; error=0.304
epoch 2650; error=0.308
epoch 2700; error=0.305
epoch 2750; error=0.305
epoch 2800; error=0.3
epoch 2850; error=0.301
epoch 2900; error=0.303
epoch 2950; error=0.307
epoch 3000; error=0.3
epoch 3050; error=0.304
epoch 3100; error=0.303
epoch 3150; error=0.302
epoch 3200; error=0.307
epoch 3250; error=0.298
epoch 3300; error=0.306
epoch 3350; error=0.304
epoch 3400; error=0.307
epoch 3450; error=0.308
epoch 3500; error=0.305
epoch 3550; error=0.301
epoch 3600; error=0.3
epoch 3650; error=0.305
```

```
epoch 3700; error=0.302
epoch 3750; error=0.307
epoch 3800; error=0.304
epoch 3850; error=0.296
epoch 3900; error=0.305
epoch 3950; error=0.304
epoch 4000; error=0.306
epoch 4050; error=0.306
epoch 4100; error=0.307
epoch 4150; error=0.305
epoch 4200; error=0.3
epoch 4250; error=0.3
epoch 4300; error=0.304
epoch 4350; error=0.3
epoch 4400; error=0.303
epoch 4450; error=0.3
epoch 4500; error=0.307
epoch 4550; error=0.308
epoch 4600; error=0.298
epoch 4650; error=0.309
epoch 4700; error=0.308
epoch 4750; error=0.309
epoch 4800; error=0.309
epoch 4850; error=0.305
epoch 4900; error=0.3
epoch 4950; error=0.303
Final result:
 Input: [0. 0.]
 Output: 0.258
 Target: 0.0

Final result:
 Input: [0. 1.]
 Output: 0.742
 Target: 1.0

Final result:
 Input: [1. 0.]
 Output: 0.744
 Target: 1.0

Final result:
 Input: [1. 1.]
 Output: 1.227
 Target: 1.0
```



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.

[11]:
```
print(w, b)
```

```
tensor([0.4855, 0.4833], requires_grad=True) tensor([0.2583],
requires_grad=True)
```

1.1.1 Problem 4

The simple linear activation function does worse on the OR problem because it can output numbers outside of the range of [0,1], which is definitely not ideal in a boolean classification problem such as this. For (0,0): (0.4855 * 0) + (0.4833 * 0) + 0.2583 = 0.2583 For (0,1): (0.4855 * 0) + (0.4833 * 1) + 0.2583 = 0.742 For (1,0): (0.4855 * 1) + (0.4833 * 0) + 0.2583 = 0.744 For (1,1): (0.4855 * 1) + (0.4833 * 1) + 0.2583 = 1.227

Because the weights are so close to each other and the bias is positive, the input of (0,0) gives us an output of 0.26 instead of approaching 0, as it should. When only a single 1 is included as input, we get an output that is lower than it should be, around 0.74. And finally, when we use (1,1) as input,

we get a number that exceeds 1. Thus, when only one 1 is provided as input, the output will always be different than in the case of (1,1) being provided as input. This is different from the sigmoid function, which is able to adjust the outputs up or down to approach 0 or 1 more definitively.

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.

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.

[12]:
```
# 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_grad=True
#     anyway so PyTorch will track the gradient too, and we can compare your␣
 ↪gradient with PyTorch's.
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_1,w_2]
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(w_34, x_pat) + b_0
        h = g_logistic(net_h)
        input = torch.cat((x_pat, h), 0)

        # Compute the output of neuron yhat
        # e.g., two lines like the following
        net_y = torch.dot(w_012[:2], x_pat) + (w_012[2] * h) + 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("")

        # Compute the gradient manually
        if verbose:
            print('Compute the gradient manually')
            print_forward(x_pat,yhat,y)
        with torch.no_grad():
            # TODO : YOUR GRADIENT CODE GOES HERE
            #  should include at least these 4 lines (helper lines may be␣
↪useful)
            # Common terms for the derivatives of the error and logistic␣
↪function
            e_der = 2 * (yhat-y)
            log_der_y = yhat * (1 - yhat)

            # Common term for the hidden neuron
            log_der_h = (h * (1 - h))

            w_34_grad = (e_der * log_der_y * w_012[2] * log_der_h * x_pat)
            b_0_grad = (e_der * log_der_y * w_012[2] * log_der_h)

            w_012_grad = (e_der * log_der_y * input)
            b_1_grad = (e_der * log_der_y)

        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: 0.746
 Target: 0.0
 Grad for w_34 and b_0
  d_loss / d_w = [-0.0855889 -0.0855889]
  d_loss / d_b = [-0.0855889]
 Grad for w_012 and b_1
  d_loss / d_w = [0.2828363  0.2828363  0.05890123]
  d_loss / d_b = [0.2828363]
```

```
Compute the gradient using PyTorch .backward()
 Grad for w_34 and b_0
  d_loss / d_w = [-0.08558889 -0.08558889]
  d_loss / d_b = [-0.08558889]
 Grad for w_012 and b_1
  d_loss / d_w = [0.28283626 0.28283626 0.05890122]
  d_loss / d_b = [0.28283626]

Compute the gradient manually
 Input: [0. 1.]
 Output: 0.381
 Target: 1.0
 Grad for w_34 and b_0
  d_loss / d_w = [0.        0.1332427]
  d_loss / d_b = [0.1332427]
 Grad for w_012 and b_1
  d_loss / d_w = [-0.         -0.29203886 -0.13362642]
  d_loss / d_b = [-0.29203886]

Compute the gradient using PyTorch .backward()
 Grad for w_34 and b_0
  d_loss / d_w = [0.         0.13324271]
  d_loss / d_b = [0.13324271]
 Grad for w_012 and b_1
  d_loss / d_w = [ 0.         -0.29203886 -0.13362642]
  d_loss / d_b = [-0.29203886]

Compute the gradient manually
 Input: [0. 0.]
 Output: 0.231
 Target: 0.0
 Grad for w_34 and b_0
  d_loss / d_w = [-0. -0.]
  d_loss / d_b = [-0.03735571]
 Grad for w_012 and b_1
  d_loss / d_w = [0.         0.         0.03807506]
  d_loss / d_b = [0.08199956]

Compute the gradient using PyTorch .backward()
 Grad for w_34 and b_0
  d_loss / d_w = [0. 0.]
  d_loss / d_b = [-0.0373557]
 Grad for w_012 and b_1
  d_loss / d_w = [0.         0.         0.03807505]
  d_loss / d_b = [0.08199955]

Compute the gradient manually
```

```
 Input: [1. 0.]
 Output: 0.577
 Target: 1.0
 Grad for w_34 and b_0
  d_loss / d_w = [0.06399292 0.        ]
  d_loss / d_b = [0.06399292]
 Grad for w_012 and b_1
  d_loss / d_w = [-0.20662391 -0.         -0.04447644]
  d_loss / d_b = [-0.20662391]

Compute the gradient using PyTorch .backward()
 Grad for w_34 and b_0
  d_loss / d_w = [0.06399291 0.        ]
  d_loss / d_b = [0.06399291]
 Grad for w_012 and b_1
  d_loss / d_w = [-0.20662388  0.         -0.04447644]
  d_loss / d_b = [-0.20662388]

epoch 0; error=1.172
epoch 50; error=1.112
epoch 100; error=1.084
epoch 150; error=1.06
epoch 200; error=1.04
epoch 250; error=1.023
epoch 300; error=1.011
epoch 350; error=1.001
epoch 400; error=0.994
epoch 450; error=0.987
epoch 500; error=0.981
epoch 550; error=0.974
epoch 600; error=0.968
epoch 650; error=0.96
epoch 700; error=0.952
epoch 750; error=0.944
epoch 800; error=0.934
epoch 850; error=0.922
epoch 900; error=0.91
epoch 950; error=0.896
epoch 1000; error=0.88
epoch 1050; error=0.862
epoch 1100; error=0.843
epoch 1150; error=0.823
epoch 1200; error=0.801
epoch 1250; error=0.777
epoch 1300; error=0.753
epoch 1350; error=0.727
epoch 1400; error=0.701
epoch 1450; error=0.675
```

```
epoch 1500; error=0.649
epoch 1550; error=0.622
epoch 1600; error=0.596
epoch 1650; error=0.571
epoch 1700; error=0.546
epoch 1750; error=0.522
epoch 1800; error=0.499
epoch 1850; error=0.477
epoch 1900; error=0.456
epoch 1950; error=0.436
epoch 2000; error=0.417
epoch 2050; error=0.399
epoch 2100; error=0.381
epoch 2150; error=0.365
epoch 2200; error=0.349
epoch 2250; error=0.335
epoch 2300; error=0.321
epoch 2350; error=0.308
epoch 2400; error=0.296
epoch 2450; error=0.284
epoch 2500; error=0.273
epoch 2550; error=0.263
epoch 2600; error=0.253
epoch 2650; error=0.244
epoch 2700; error=0.235
epoch 2750; error=0.227
epoch 2800; error=0.219
epoch 2850; error=0.212
epoch 2900; error=0.205
epoch 2950; error=0.198
epoch 3000; error=0.192
epoch 3050; error=0.186
epoch 3100; error=0.18
epoch 3150; error=0.175
epoch 3200; error=0.17
epoch 3250; error=0.165
epoch 3300; error=0.16
epoch 3350; error=0.156
epoch 3400; error=0.152
epoch 3450; error=0.148
epoch 3500; error=0.144
epoch 3550; error=0.14
epoch 3600; error=0.136
epoch 3650; error=0.133
epoch 3700; error=0.13
epoch 3750; error=0.127
epoch 3800; error=0.124
epoch 3850; error=0.121
```

```
epoch 3900; error=0.118
epoch 3950; error=0.115
epoch 4000; error=0.113
epoch 4050; error=0.11
epoch 4100; error=0.108
epoch 4150; error=0.106
epoch 4200; error=0.104
epoch 4250; error=0.101
epoch 4300; error=0.099
epoch 4350; error=0.097
epoch 4400; error=0.096
epoch 4450; error=0.094
epoch 4500; error=0.092
epoch 4550; error=0.09
epoch 4600; error=0.089
epoch 4650; error=0.087
epoch 4700; error=0.085
epoch 4750; error=0.084
epoch 4800; error=0.082
epoch 4850; error=0.081
epoch 4900; error=0.08
epoch 4950; error=0.078
epoch 5000; error=0.077
epoch 5050; error=0.076
epoch 5100; error=0.075
epoch 5150; error=0.073
epoch 5200; error=0.072
epoch 5250; error=0.071
epoch 5300; error=0.07
epoch 5350; error=0.069
epoch 5400; error=0.068
epoch 5450; error=0.067
epoch 5500; error=0.066
epoch 5550; error=0.065
epoch 5600; error=0.064
epoch 5650; error=0.063
epoch 5700; error=0.062
epoch 5750; error=0.062
epoch 5800; error=0.061
epoch 5850; error=0.06
epoch 5900; error=0.059
epoch 5950; error=0.058
epoch 6000; error=0.058
epoch 6050; error=0.057
epoch 6100; error=0.056
epoch 6150; error=0.055
epoch 6200; error=0.055
epoch 6250; error=0.054
```

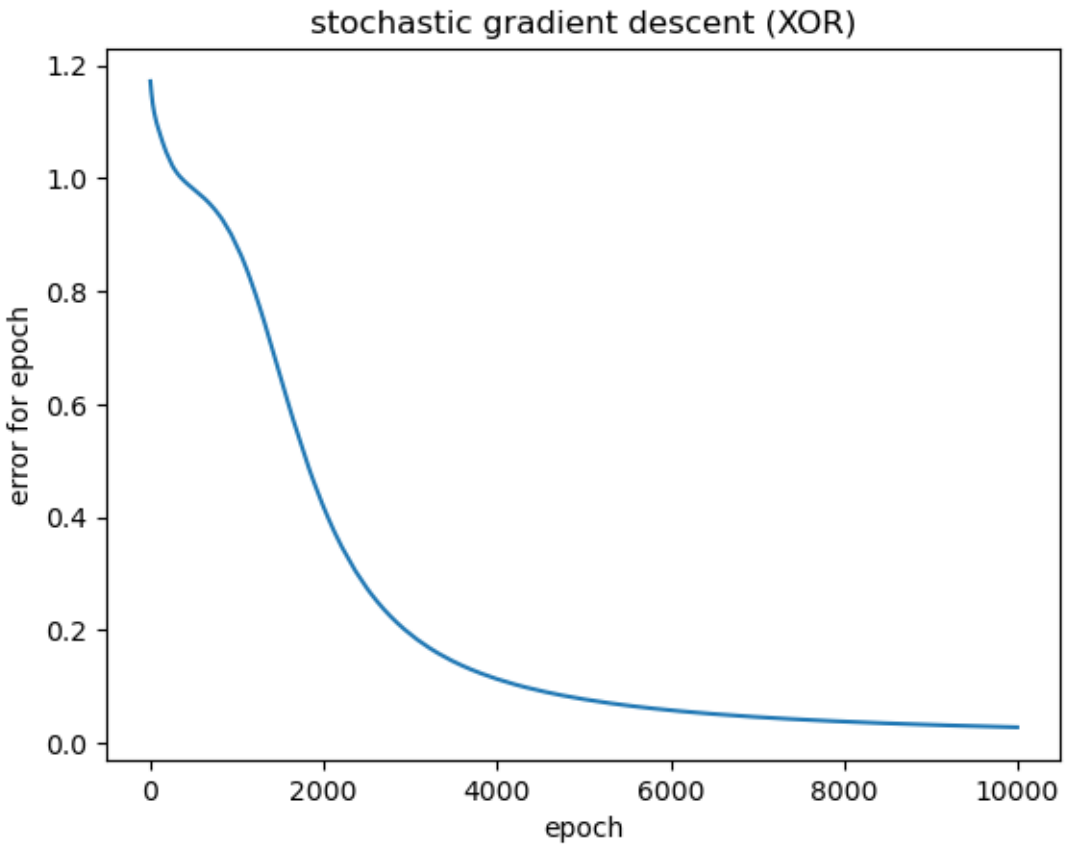
```
epoch 6300; error=0.053
epoch 6350; error=0.053
epoch 6400; error=0.052
epoch 6450; error=0.051
epoch 6500; error=0.051
epoch 6550; error=0.05
epoch 6600; error=0.05
epoch 6650; error=0.049
epoch 6700; error=0.049
epoch 6750; error=0.048
epoch 6800; error=0.048
epoch 6850; error=0.047
epoch 6900; error=0.046
epoch 6950; error=0.046
epoch 7000; error=0.046
epoch 7050; error=0.045
epoch 7100; error=0.045
epoch 7150; error=0.044
epoch 7200; error=0.044
epoch 7250; error=0.043
epoch 7300; error=0.043
epoch 7350; error=0.042
epoch 7400; error=0.042
epoch 7450; error=0.042
epoch 7500; error=0.041
epoch 7550; error=0.041
epoch 7600; error=0.04
epoch 7650; error=0.04
epoch 7700; error=0.04
epoch 7750; error=0.039
epoch 7800; error=0.039
epoch 7850; error=0.038
epoch 7900; error=0.038
epoch 7950; error=0.038
epoch 8000; error=0.037
epoch 8050; error=0.037
epoch 8100; error=0.037
epoch 8150; error=0.036
epoch 8200; error=0.036
epoch 8250; error=0.036
epoch 8300; error=0.036
epoch 8350; error=0.035
epoch 8400; error=0.035
epoch 8450; error=0.035
epoch 8500; error=0.034
epoch 8550; error=0.034
epoch 8600; error=0.034
epoch 8650; error=0.034
```

```
epoch 8700; error=0.033
epoch 8750; error=0.033
epoch 8800; error=0.033
epoch 8850; error=0.032
epoch 8900; error=0.032
epoch 8950; error=0.032
epoch 9000; error=0.032
epoch 9050; error=0.031
epoch 9100; error=0.031
epoch 9150; error=0.031
epoch 9200; error=0.031
epoch 9250; error=0.031
epoch 9300; error=0.03
epoch 9350; error=0.03
epoch 9400; error=0.03
epoch 9450; error=0.03
epoch 9500; error=0.029
epoch 9550; error=0.029
epoch 9600; error=0.029
epoch 9650; error=0.029
epoch 9700; error=0.029
epoch 9750; error=0.028
epoch 9800; error=0.028
epoch 9850; error=0.028
epoch 9900; error=0.028
epoch 9950; error=0.028
Final result:
 Input: [0. 1.]
 Output: 0.914
 Target: 1.0

Final result:
 Input: [1. 1.]
 Output: 0.096
 Target: 0.0

Final result:
 Input: [0. 0.]
 Output: 0.058
 Target: 0.0

Final result:
 Input: [1. 0.]
 Output: 0.914
 Target: 1.0
```



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.

[22]:
```
print('weight for w3 :', w_34[0], '\nweight for w4 :', w_34[1], '\nbias 0 :',␣
  ↪b_0, '\n\nweight for w0 :', w_012[0], '\nweight for w1 :', w_012[1],␣
  ↪'\nweight for w2 :', w_012[2], '\nbias 1 :', b_1)
```

```
weight for w3 : tensor(-6.8323, grad_fn=<SelectBackward0>)
weight for w4 : tensor(-6.8301, grad_fn=<SelectBackward0>)
bias 0 : tensor([2.5026], requires_grad=True)

weight for w0 : tensor(-4.7487, grad_fn=<SelectBackward0>)
weight for w1 : tensor(-4.7484, grad_fn=<SelectBackward0>)
weight for w2 : tensor(-10.8628, grad_fn=<SelectBackward0>)
bias 1 : tensor([7.2501], requires_grad=True)
```

1.1.2 Problem 6

When examining the learned weights and biases of our network, we see that all of the weights are negative, whereas the biases are both positive, unlike the example shown in lecture. The weights connected to the hidden layer (w3 and w4) are nearly identical, as are the weights connecting the initial inputs to the output layer (w0 and w1). After performing the calculations below, we can see that the hidden layer is acting as a sort of "AND" unit, inhibiting y_hat when both inputs to the hidden unit are equal. Let's walk through each input pattern to see if we are correct:

For [1,1]: Hidden layer calculation: 1 / (1 + e^-((-6.8323 * 1) + (-6.8301 * 1) + 2.5)) = 0.00001 Direct Connections calculation: 1 / (1 + e^-((-4.7487 * 1) + (-4.7484 * 1) + (0.00001 * -10.8628) + 7.25)) = 0.096 Here we see that the hidden layer outputs a very low value since both inputs are 1 and the connections to the hidden layer are negatively weighted. The negative connections to y_hat (w0, w1, w2) and the positive bias still result in a low net input to y_hat, leading to an output close to 0.

For [1,0]: Hidden layer calculation: 1/(1 + e^-((-6.8323 * 1) + (-6.8301 * 0)+ 2.5)) = 0.012 Direct Connections calculation: 1 / (1 + e^-((-4.7487 * 1) + (-4.7484 * 0) + (0.012 * -10.8628) + 7.25)) = 0.914

For [0,1]: Hidden layer calculation: 1/(1 + e^-((-6.8323 * 0) + (-6.8301 * 1)+ 2.5)) = 0.012 Direct Connections calculation: 1 / (1 + e^-((-4.7487 * 0) + (-4.7484 * 1) + (0.012 * -10.8628) + 7.25)) = 0.914 In the previous two examples, the hidden layer outputs a low value because of the negative weights w3 and w4, but the direct connections to y_hat and the bias lead to activation of y_hat and we get results very close to 1.

For [0,0]: Hidden layer calculation: 1/(1 + e^-((-6.8323 * 0) + (-6.8301 * 0)+ 2.5)) = 0.924 Direct Connections calculation: 1 / (1 + e^-((-4.7487 * 0) + (-4.7484 * 0) + (0.924 * -10.8628) + 7.25)) = 0.058 Here we see that the hidden layer outputs a higher value than the other three examples since both weights are negative and the bias is positive. Though, since weights w0 and w1 are negative, the input to y_hat is lower and the positive bias is not enough to engage y_hat.

In these examples, we see that the hidden layer is indeed acting as an "AND" unit.

[ ]: