Homework - Neural networks - Part A (35 points)

Interactive activation and competition

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.

Note:
Please complete the responses to these questions as a markdown cell inserted beneath the question prompts.

In this assignment, you will get hands on experience with a classic neural network model of memory known as the Interactive Activation and Competition (IAC) model. We will go through a series of exercises that will stretch your understanding of the IAC model in various ways. The exercises below examine how the mechanisms of interactive activation and competition can be used to illustrate two key properties of human memory:

- Retrieval by name and by content.
- Spontaneous generalization over a set of familiar items.

These exercises are from Chapter 2 of the online PDP Handbook by James McClelland.

You should review the slides from lecture and read Section 2.1 of the PDP Handbook before continuing. This has important background and technical details on how the IAC model works. The IAC model instantiates knowledge that someone may have from watching the 1960s musical "West Side Story," where two gangs the "Jets" and "Sharks" struggle for neighborhood control in Manhattan. The "database" for this exercise is the Jets and Sharks data base shown in Figure 1, which has the central characters from the two gangs. You are to use the IAC model in conjunction with this data base to run illustrative simulations of these basic properties of memory.

The Jets and The Sharks

Name	Gang	Age	Edu	Mar	Occupation
Art	Jets	40's	J.H.	Sing.	Pusher
Al	Jets	30's	J.H.	Mar.	Burglar
Sam	Jets	20's	COL.	Sing.	Bookie
Clyde	Jets	40's	J.H.	Sing.	Bookie
Mike	Jets	30's	J.H.	Sing.	Bookie
Jim	Jets	20's	J.H.	Div.	Burglar
Greg	Jets	20's	H.S.	Mar.	Pusher
John	Jets	20's	J.H.	Mar.	Burglar
Doug	Jets	30's	H.S.	Sing.	Bookie
Lance	Jets	20's	J.H.	Mar.	Burglar
George	Jets	20's	J.H.	Div.	Burglar
Pete	Jets	20's	H.S.	Sing.	Bookie
Fred	Jets	20's	H.S.	Sing.	Pusher
Gene	Jets	20's	COL.	Sing.	Pusher
Ralph	Jets	30's	J.H.	Sing.	Pusher
Phil	Sharks	30's	COL.	Mar.	Pusher
Ike	Sharks	30's	J.H.	Sing.	Bookie
Nick	Sharks	30's	H.S.	Sing.	Pusher
Don	Sharks	30's	COL.	Mar.	Burglar
Ned	Sharks	30's	COL.	Mar.	Bookie
Karl	Sharks	40's	H.S.	Mar.	Bookie
Ken	Sharks	20's	H.S.	Sing.	Burglar
Earl	Sharks	40's	H.S.	Mar.	Burglar
Rick	Sharks	30's	H.S.	Div.	Burglar
Ol	Sharks	30's	COL.	Mar.	Pusher
Neal	Sharks	30's	H.S.	Sing.	Bookie
Dave	Sharks	30's	H.S.	Div.	Pusher

Figure 1: Characteristics of a number of individuals belonging to two gangs, the Jets and the Sharks. (From "Retrieving General and Specific Knowledge From Stored Knowledge of Specifics" by J. L. McClelland, 1981, Proceedings of the Third Annual Conference of the Cognitive Science Society.)

Software and architecture

We will be using IAC software from Axel Cleeremans which you can download here for Mac OS and here for Windows.
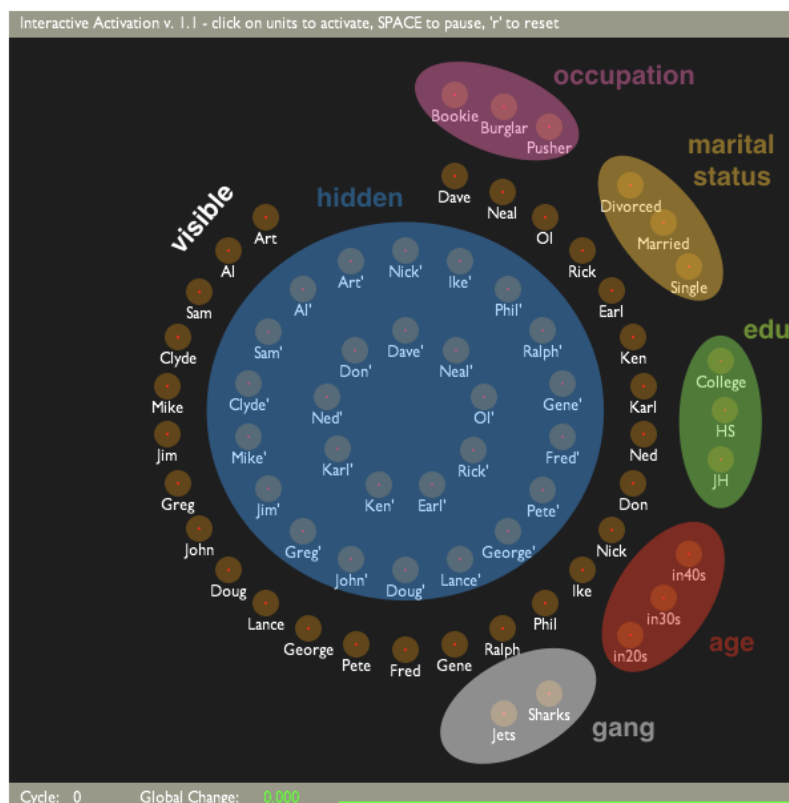
Figure 2: Screen shot from Cleeremans' IAC software. Units are organized into 7 groups. For illustration here, all groups have a different color background, while the group of visible name units have no background.

Upon downloading and loading the software, you will see a display that looks like Figure 2. The units are grouped into seven pools: a pool of *visible* name units, a pool of *gang* units, a pool of *age* units, a pool of *education* units, a pool of *marital status* units, a pool of *occupation* units, and a pool of *hidden* units. The name pool contains a unit for the name of each person; the gang pool contains a unit for each of the gangs the people are members of (Jets and Sharks); the age pool contains a unit for each age range; and so on. Finally, the *hidden* pool contains an instance unit for each individual in the set.

The units in the first six pools can be called visible units, since all are assumed to be accessible from outside the network. Those in the gang, age, education, marital status, and occupation pools can also be called property units. The instance units are assumed to be inaccessible, so they can be called hidden units.
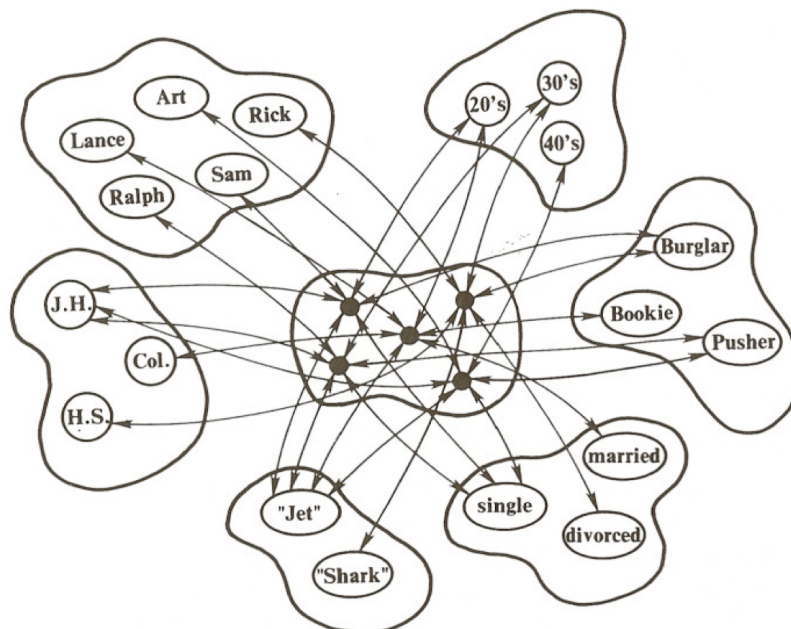
Figure 3: The units and connections for some of the individuals in Figure 1. The arrows represent excitatory connections. The outlined groups of units have mutually inhibitory connections (not shown). (From "Retrieving General and Specific Knowledge From Stored Knowledge of Specifics" by J. L. McClelland, 1981, Proceedings of the Third Annual Conference of the Cognitive Science Society.)

Each unit has an inhibitory connection to every other unit in the same pool. In addition, there are two-way excitatory connections between each instance unit and the units for its properties, as illustrated in Figure 3. Note that the figure is incomplete, in that only some of the name and instance units are shown. These names are given only for the convenience of the user, of course; all actual computation in the network occurs only by way of the connections. You can also view the different connections using the IAC software by hovering your mouse over a particular unit (Figure 4).
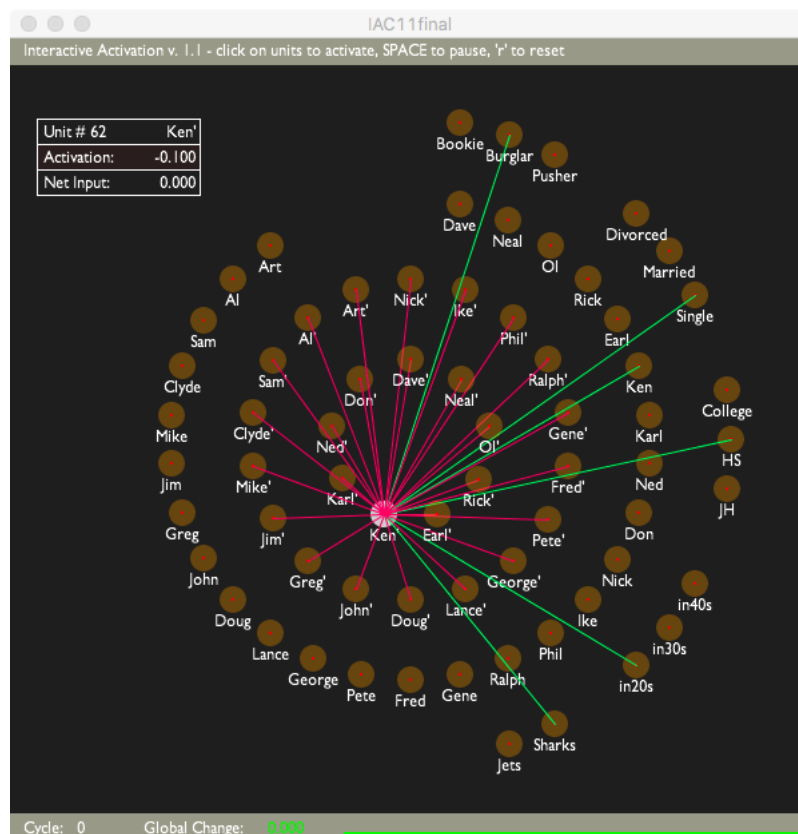
Figure 4: You can view the connections to a unit by placing your mouse over it.
Green connections are excitatory and red connections are inhibitory.

Since everything is set up for you, you are now ready to do each of the separate parts of the exercise. Each part is accomplished by using the interactive activation and competition process to do pattern completion, given some probe that is presented to the network. For example, to retrieve an individual's properties from his name, you simply provide external input to his name unit, then allow the IAC network to propagate activation first to the name unit, then from there to the instance units, and from there to the units for the properties of the instance.

Exercise: Retreiving an individual from his name

To illustrate retrieval of the properties of an individual from his name, we will use Ken as our example. Make sure the simulation is paused (press SPACE) and press 'r' to reset it. Set the external input of Ken's name unit to 1 by clicking on the name unit (not the hidden unit!). The circle's background should turn bright green to represent the external input.

A unit's activity level can be visualized by the colored dot, where yellow dots are positive activation and red dots are negative activation. The larger the yellow dot, the stronger the activation. A unit's precise activity level can examined by rolling the mouse over the unit.
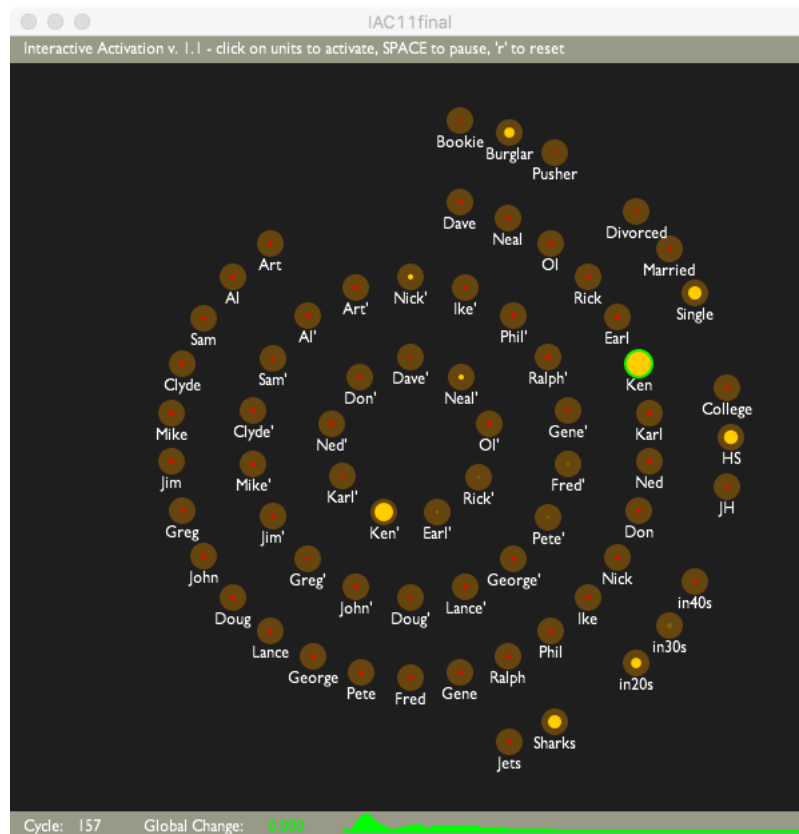
Figure 5: The display screen after about 150 cycles with external input to the name unit for Ken.

Press SPACE to unpause and allow the network to run for approximately 150 cycles (the cycle counter is in the bottom left of the panel). The simulation runs quickly, so be sure to pause at about 150 cycles exactly! A picture of the screen after 150 cycles is shown in Figure 5. At this point, you can check to see that the model has indeed retrieved the pattern for Ken correctly. There are also several other things going on that are worth understanding. Answer all of the following questions below regarding the network at this state (you'll have to refer to the properties of the individuals, as given in Figure 1).

> **Problem 1 (10 points)**
> None of the visible name units other than Ken were activated, yet a few other hidden instance units are active (i.e., their activation is greater than 0). Explain why these units are active. Keep your response short (about 3 sentences).

My Answer:

Among the other active hidden instance units, Neal' and Nick' are the most activated two units.

The reason that makes them active is that they both share three same properties (Single, HS, Sharks) with Ken', which makes them indirectly activated by Ken'.

> **Problem 2 (10 points)**
> Some of Ken's properties are activated more strongly than others. Why? Keep

your response short (about 3 sentences).

My Answer:

The properties with more entries will be activated not only by the source Ken' but by other entries it activated.

The three highest properties, Single, HS, and Sharks, all share the same entries: Neal' and Nick'.

HS has the highest activation among the three highest properties because it is suppressed by fewer entries than the other two properties.

Retrieval from a partial description

Next, we will use the IAC software to illustrate how it can retrieve an instance from a partial description of its properties. We will continue to use Ken, who, as it happens, can be uniquely described by two properties, Shark and in20s. Reset the network ('r') and make sure everything is paused and that all units have input of 0. Click to set the external input of the Sharks unit and the in20s unit to 1.00. Run a total of 150 cycles again, and take a look at the state of the network.

Of all of the visible name units, Ken's name should be the most active. Compare the state of the network's with the a screen shot of the previous network state when activating Ken's name directly, such as that in Figure 5.

> **Problem 3 (10 points)**
> Explain why the occupation units show partial activations of units other than Ken's occupation, which is Burglar. While being succinct, try to get to the bottom of this, and contrast the current case with the previous case. Keep your response short (about 3 sentences).

My Answer:

Sharks and in20s activate Ken' to a great extent, but then Ken' also activated HS and Single a lot, which leads to the activation of the entries Nick', Neal', Fred', and Pete'.

Then, Nick' and Fred' activate Pusher, while Neal' and Pete' activate Bookie.

In the end, since the other two occupations got activation sources as well, they are also activated although they do not reach the extent of Burglar.

Spontaneous generalization

Now we consider the network's ability to retrieve appropriate generalizations over sets of individuals—that is, its ability to answer questions like "What are Jets like?" or "What are people who are in their 20s and have only a junior high education like?"

Reset ('r') the network. Make sure all units have input of 0 and none are highlighted green.

Set the external input of Jets to 1.00 by clicking on it. Run the network for 150 cycles and observe what happens.

> **Problem 4 (5 points)**
> Given the network's state, what can you infer about a typical Jet? (1-2 sentences is plenty).

My Answer:

The typical Jet is Single, in their 20s, and have only a junior high education.

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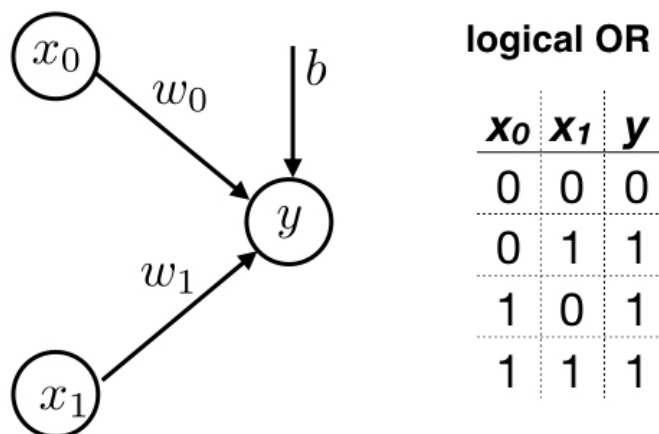
This assignment requires some basic PyTorch knowledge. You can review your notes from lab and this PyTorch tutorial. The "Introduction to PyTorch" section on the PyTorch website is also helpful.

```python
# 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.

```python
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.

```python
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
```
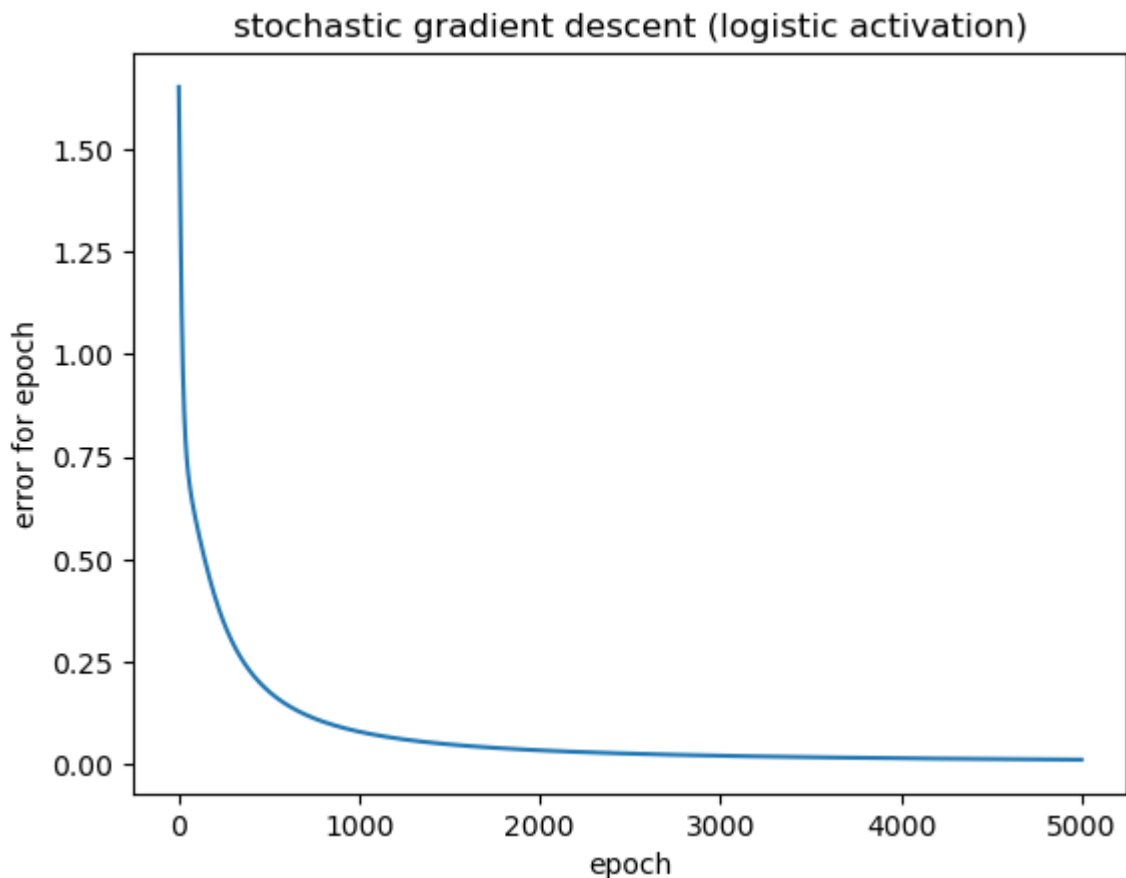
stochastic gradient descent (logistic activation)


```
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)')
```

```python
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
```
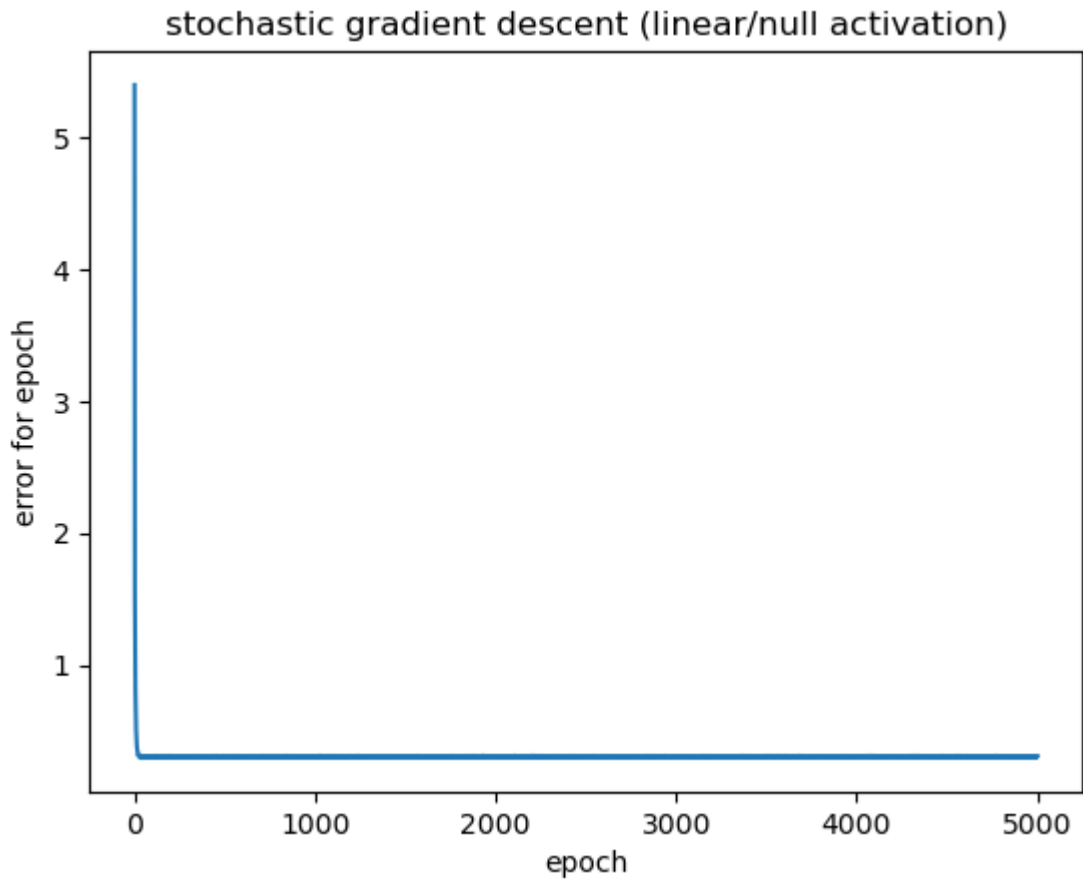
stochastic gradient descent (linear/null activation)


```
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
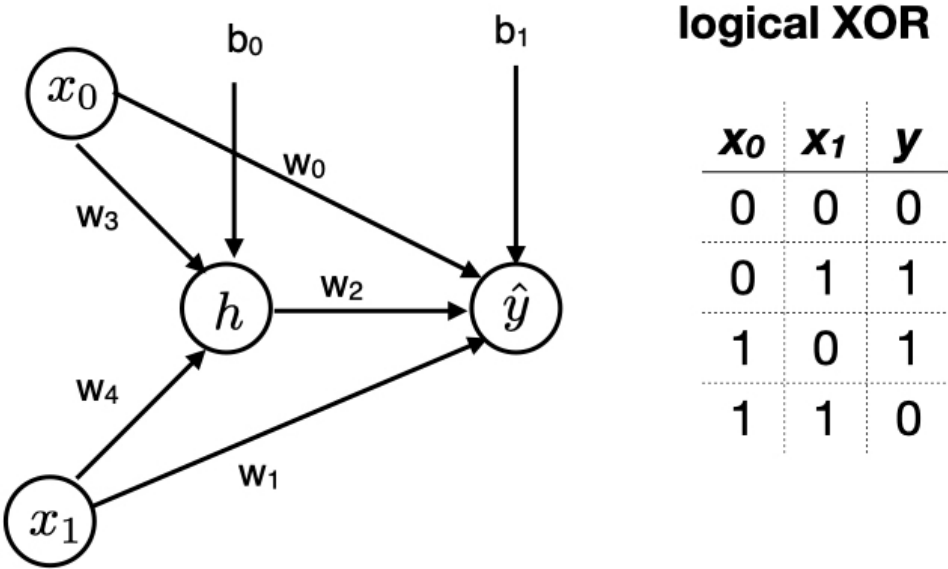
Problem 5 (15 points)

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")
```

```python
            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
```
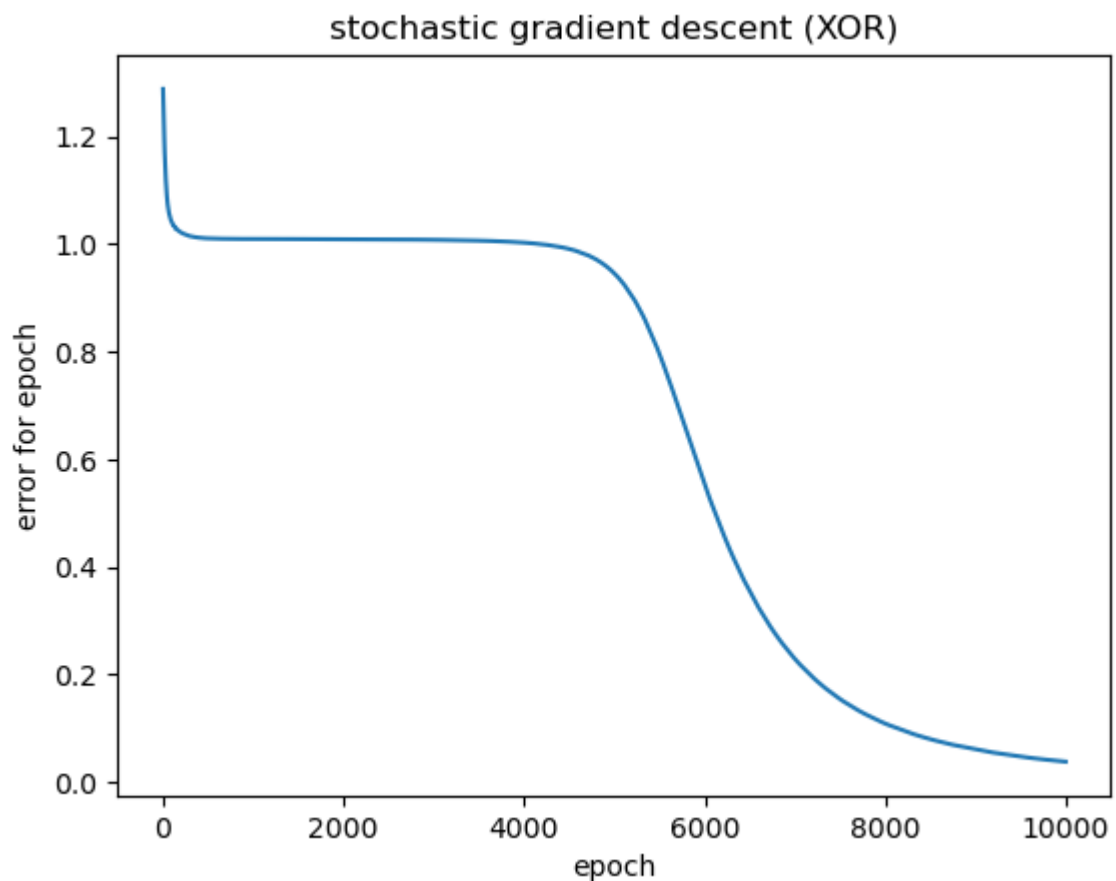
```
net_y: -2.29
yhat :  0.09

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 [ ]:  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	$\hat{y} = \text{output}$
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.

Homework - Neural networks - Part C (25 points)

A neural network model of semantic cognition

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.

In this assignment, you will help implement and analyze a neural network model of semantic cognition. Semantic cognition is our intuitive understanding of objects and their properties. Semantic knowledge includes observations of which objects have which properties, and storage of these facts in long term memory. It also includes the ability to generalize, or predict which properties apply to which objects although they have not been directly observed.

This notebook explores a neural network model of semantic cognition developed by Rogers and McClelland (R&M). R&M sought to model aspects of semantic cognition with a multi-layer neural network, which contrasts with classic symbolic approaches for organizing semantic knowledge. They model the cognitive development of semantic representation as gradient descent (the backpropgation algorithm), using a neural network trained to map objects to their corresponding properties. R&M also modeled the deterioration of semantic knowledge in dementia by adding noise to the learned representations.

The network architecture is illustrated below.
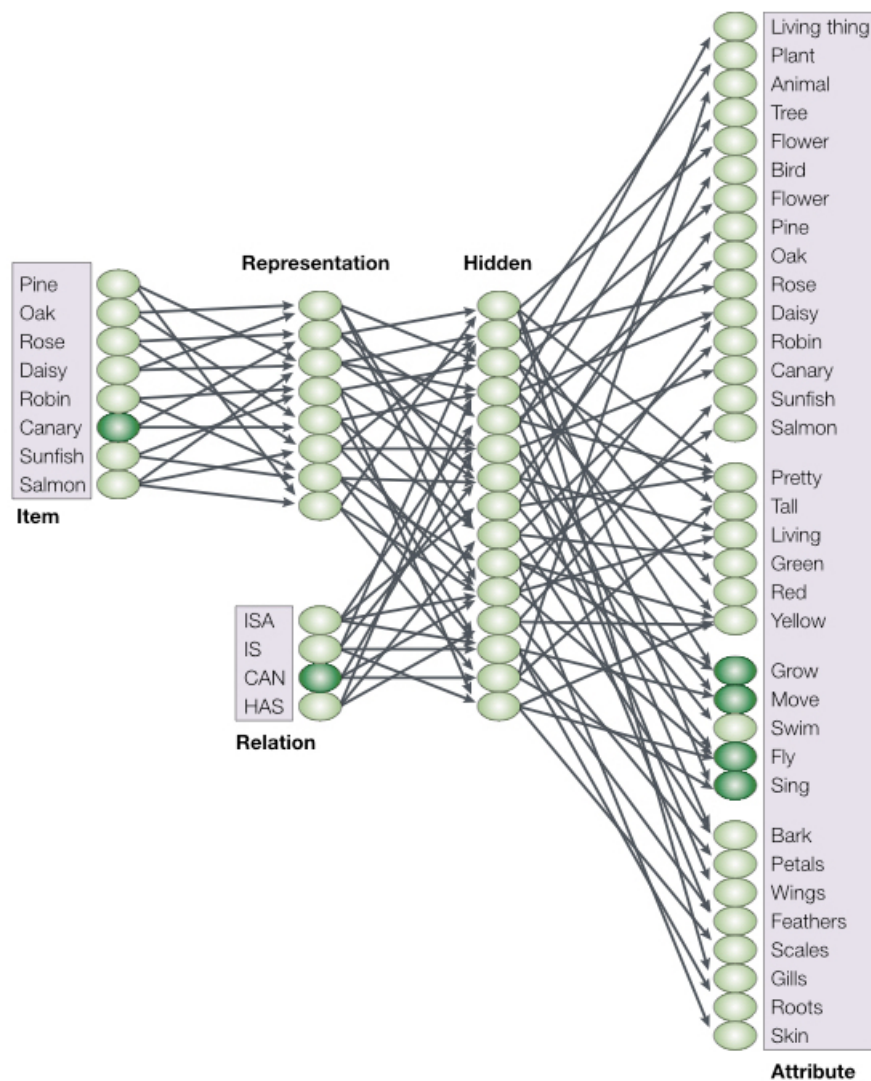
There are two input layers ("Item Layer" and "Relation Layer"), which pass through intermediate layers to produce an output pattern on the "Attribute Layer." In this example, dark green is used to indicate active nodes (activation 1) and light green for inactive nodes (activation 0). The network is trained to answer queries involving an item (e.g., "Canary") and a relation (e.g., "CAN"), outputting all attributes that are true of the item/relation pair (e.g., "grow, move, fly, sing").

For this assignment, you will set up the network architecture in PyTorch and train it. The dataset and code for training has been provided. You will then analyze how its semantic knowledge develops of the course of training. While the original model used logistic (sigmoid) activation functions for all of the intermediate and output layers, we will use the ReLu activation for the Representation and Hidden Layers, with a sigmoid activation for the Attribute Layer.

Completing this assignment requires knowledge of setting up a neural network architecture in PyTorch. Please review your notes from lab and the following PyTorch tutorial.

Reference (available for download on Brightspace):

> McClelland, J. L., & Rogers, T. T. (2003). The parallel distributed processing approach to semantic cognition. Nature Reviews Neuroscience, 4(4), 310.

```python
# Import libraries
from __future__ import print_function
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
from torch.nn.functional import sigmoid, relu
from scipy.cluster.hierarchy import dendrogram, linkage
```

Let's first load in the names of all the items, attributes, and relations into Python lists.

```python
with open('data/sem_items.txt','r') as fid:
    names_items = np.array([l.strip() for l in fid.readlines()])
with open('data/sem_relations.txt','r') as fid:
    names_relations = np.array([l.strip() for l in fid.readlines()])
with open('data/sem_attributes.txt','r') as fid:
    names_attributes = np.array([l.strip() for l in fid.readlines()])

nobj = len(names_items)
nrel = len(names_relations)
nattributes = len(names_attributes)
print('List of items:')
print(names_items)
print("List of relations:")
print(names_relations)
print("List of attributes:")
print(names_attributes)
```

```
List of items:
['Pine' 'Oak' 'Rose' 'Daisy' 'Robin' 'Canary' 'Sunfish' 'Salmon']
List of relations:
['ISA' 'Is' 'Can' 'Has']
List of attributes:
['Living thing' 'Plant' 'Animal' 'Tree' 'Flower' 'Bird' 'Fish' 'Pine'
 'Oak' 'Rose' 'Daisy' 'Robin' 'Canary' 'Sunfish' 'Salmon' 'Pretty' 'Big'
 'Living' 'Green' 'Red' 'Yellow' 'Grow' 'Move' 'Swim' 'Fly' 'Sing' 'Skin'
 'Roots' 'Leaves' 'Bark' 'Branch' 'Petals' 'Wings' 'Feathers' 'Gills'
 'Scales']
```

Next, let's load in the data matrix from a text file too. The matrix `D` has a row for each training pattern. It is split into a matrix of input patterns `input_pats` (item and relation) and their corresponding output patterns `output_pats` (attributes). The are `N` patterns total in the set.

For each input pattern, the first 8 elements indicate which item is being presented, and the next 4 indicate which relation is being queried. Each element of the output pattern corresponds to a different attribute. All patterns use 1-hot encoding.

```python
D = np.loadtxt('data/sem_data.txt')
input_pats = D[:,:nobj+nrel]
input_pats = torch.tensor(input_pats,dtype=torch.float)
```

```python
output_pats = D[:,nobj+nrel:]
output_pats = torch.tensor(output_pats,dtype=torch.float)
N = input_pats.shape[0] # number of training patterns
input_v = input_pats[0,:].numpy().astype('bool')
output_v = output_pats[0,:].numpy().astype('bool')
print('Example input pattern:')
print(input_v.astype('int'))
print('Example output pattern:')
print(output_v.astype('int'))
print("")
print("Which encodes...")
print('Item ',end='')
print(names_items[input_v[:8]])
print('Relation ',end='')
print(names_relations[input_v[8:]])
print('Attributes ',end='')
print(names_attributes[output_v])
```

```
Example input pattern:
[1 0 0 0 0 0 0 0 1 0 0 0]
Example output pattern:
[1 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

Which encodes...
Item ['Pine']
Relation ['ISA']
Attributes ['Living thing' 'Plant' 'Tree' 'Pine']
```

Problem 1 (20 points)

Your assignment is to create the neural network architecture shown in the figure above. Fill in the missing pieces of the "Net" class in the code below. For an example, refer to the PyTorch tutorial on "Neural Networks".

Use the ReLu activation function ("relu") for the Representation and Hidden Layers, with a Logistic/Sigmoid activation function for the Attribute Layer ("sigmoid").

You will need PyTorch's "nn.Linear" function for constructing the layers, and the "relu" and "sigmoid" activation functions.

```python
In [ ]: class Net(nn.Module):
    def __init__(self, rep_size, hidden_size):
        super(Net, self).__init__()
        self.rep_layer = nn.Linear(nobj, rep_size)
        self.hid_layer = nn.Linear(rep_size + nrel, hidden_size)
        self.out_layer = nn.Linear(hidden_size, nattributes)

    def forward(self, x):
        # Defines forward pass for the network on input patterns x
        #
        # Input can take these two forms:
        #
        #   x: [nobj+nrel 1D Tensor], which is a single input pattern as a 1D te
```

```
        #     (containing both object and relation 1-hot identifier) (batch siz
        #   OR
        #   x : [B x (nobj+nrel) Tensor], which is a batch of B input patterns (
        #
        # Output
        #   output [B x nattribute Tensor], which is the output pattern for each
        #   hidden [B x hidden_size Tensor], which are activations in the Hidden
        #   rep [B x rep_size Tensor], which are the activations in the Represen
        x = x.view(-1, nobj + nrel) # reshape as size [B x (nobj+nrel) Tensor] i
        x_pat_item = x[:,:nobj] # input to Item Layer [B x nobj Tensor]
        x_pat_rel = x[:,nobj:] # input to Relation Layer [B x nrel Tensor]
        x_pat_rep = nn.ReLU()(self.rep_layer(x_pat_item))
        x_pat_hid = nn.ReLU()(self.hid_layer(torch.cat([x_pat_rep, x_pat_rel], d
        x_pat_out = nn.Sigmoid()(self.out_layer(x_pat_hid))
        output, hidden, rep = x_pat_out, x_pat_hid, x_pat_rep
        return output, hidden, rep
```

We provide a completed function `train` for stochastic gradient descent. The network makes online (rather than batch) updates, adjusting its weights after the presentation of each input pattern.

```
In [ ]:  def train(mynet, epoch_count, nepochs_additional=5000):
            # Input
            #  mynet : Net class object
            #  epoch_count : (scalar) how many epochs have been completed so far
            #  nepochs_additional : (scalar) how many more epochs we want to run
            mynet.train()
            for e in range(nepochs_additional): # for each epoch
                error_epoch = 0.
                perm = np.random.permutation(N)
                for p in perm: # iterate through input patterns in random order
                    mynet.zero_grad() # reset gradient
                    output, hidden, rep = mynet(input_pats[p,:]) # forward pass
                    target = output_pats[p,:]
                    loss = criterion(output, target) # compute loss
                    loss.backward() # compute gradient
                    optimizer.step() # update network parameters
                    error_epoch += loss.item()
                error_epoch = error_epoch / float(N)
                if e % 50 == 0:
                    print('epoch ' + str(epoch_count+e) + ' loss ' + str(round(error_epo
            return epoch_count + nepochs_additional
```

We provide some useful functions for extracting the activation pattern on the Representation Layer for each possible item. We provide two functions `plot_rep` and `plot_dendo` for visualizing these activation patterns.

```
In [ ]:  def get_rep(mynet):
            # Extract the hidden activations on the Representation Layer for each item
            #
            # Input
            #  mynet : Net class object
            #
            # Output
            #  rep : [nitem x rep_size numpy array], where each row is an item
            input_clean = torch.zeros(nobj, nobj+nrel)
            for idx, name in enumerate(names_items):
```

```python
        input_clean[idx, idx] = 1. # 1-hot encoding of each object (while Relati
    output, hidden, rep = mynet(input_clean)
    return rep.detach().numpy()

def plot_rep(rep1, rep2, rep3, names):
    # Compares Representation Layer activations of Items at three different tim
    # using bar graphs
    #
    # Each rep1, rep2, rep3 is a [nitem x rep_size numpy array]
    # names : [nitem list] of item names
    #
    nepochs_list = [ nepochs_phase1, nepochs_phase2, nepochs_phase3 ]
    nrows = nobj
    R = np.dstack((rep1, rep2, rep3))
    mx = R.max()
    mn = R.min()
    depth = R.shape[2]
    count = 1
    plt.figure(1,figsize=(4.2, 8.4))
    for i in range(nrows):
        for d in range(R.shape[2]):
            plt.subplot(nrows, depth, count)
            rep = R[i, :, d]
            plt.bar(range(rep.size), rep)
            plt.ylim([mn, mx])
            plt.xticks([])
            plt.yticks([])
            if d==0:
                plt.ylabel(names[i])
            if i==0:
                plt.title("epoch " + str(nepochs_list[d]))
            count += 1
    plt.show()

def plot_dendo(rep1, rep2, rep3, names):
    # Compares Representation Layer activations of Items at three different tim
    # using hierarchical clustering
    #
    # Each rep1, rep2, rep3 is a [nitem x rep_size numpy array]
    # names : [nitem list] of item names
    #
    nepochs_list = [ nepochs_phase1, nepochs_phase2, nepochs_phase3 ]
    linked1 = linkage(rep1, "single")
    linked2 = linkage(rep2, "single")
    linked3 = linkage(rep3, "single")
    mx = np.dstack((linked1[:,2], linked2[:,2], linked3[:,2])).max() + 0.1
    plt.figure(2, figsize=(7, 12))
    plt.subplot(3, 1, 1)
    dendrogram(linked1, labels=names, color_threshold=0)
    plt.ylim([0, mx])
    plt.title("Hierarchical clustering; " + "epoch " + str(nepochs_list[0]))
    plt.ylabel("Euclidean distance")
    plt.subplot(3, 1, 2)
    plt.title("epoch " + str(nepochs_list[1]))
    dendrogram(linked2, labels=names, color_threshold=0)
    plt.ylim([0, mx])
    plt.subplot(3, 1, 3)
    plt.title("epoch " + str(nepochs_list[2]))
    dendrogram(linked3, labels=names, color_threshold=0)
```

```
        plt.ylim([0, mx])
        plt.show()
```

The next script initializes the neural network and trains it for 2500 epochs total. It trains in three stages, and the item representations (on the Representation Layer) are extracted after 500 epochs, 1000 epochs, and then at the end of training (2500 epochs).

In [ ]:
```
learning_rate = 0.1
criterion = nn.MSELoss() # mean squared error loss function
mynet = Net(rep_size=8, hidden_size=15)
optimizer = torch.optim.SGD(mynet.parameters(), lr=learning_rate) # stochastic g

nepochs_phase1 = 500
nepochs_phase2 = 1000
nepochs_phase3 = 2500
epoch_count = 0
epoch_count = train(mynet, epoch_count, nepochs_additional=nepochs_phase1)
rep1 = get_rep(mynet)
epoch_count = train(mynet, epoch_count, nepochs_additional=nepochs_phase2-nepoch
rep2 = get_rep(mynet)
epoch_count = train(mynet, epoch_count, nepochs_additional=nepochs_phase3-nepoch
rep3 = get_rep(mynet)
```

```
epoch 0 loss 0.241
epoch 50 loss 0.07
epoch 100 loss 0.067
epoch 150 loss 0.064
epoch 200 loss 0.059
epoch 250 loss 0.055
epoch 300 loss 0.052
epoch 350 loss 0.049
epoch 400 loss 0.047
epoch 450 loss 0.045
epoch 500 loss 0.043
epoch 550 loss 0.041
epoch 600 loss 0.039
epoch 650 loss 0.036
epoch 700 loss 0.033
epoch 750 loss 0.03
epoch 800 loss 0.028
epoch 850 loss 0.026
epoch 900 loss 0.025
epoch 950 loss 0.024
epoch 1000 loss 0.023
epoch 1050 loss 0.022
epoch 1100 loss 0.02
epoch 1150 loss 0.018
epoch 1200 loss 0.016
epoch 1250 loss 0.015
epoch 1300 loss 0.014
epoch 1350 loss 0.013
epoch 1400 loss 0.012
epoch 1450 loss 0.011
epoch 1500 loss 0.01
epoch 1550 loss 0.01
epoch 1600 loss 0.009
epoch 1650 loss 0.009
epoch 1700 loss 0.009
epoch 1750 loss 0.008
epoch 1800 loss 0.008
epoch 1850 loss 0.008
epoch 1900 loss 0.007
epoch 1950 loss 0.007
epoch 2000 loss 0.007
epoch 2050 loss 0.007
epoch 2100 loss 0.006
epoch 2150 loss 0.006
epoch 2200 loss 0.005
epoch 2250 loss 0.005
epoch 2300 loss 0.005
epoch 2350 loss 0.004
epoch 2400 loss 0.004
epoch 2450 loss 0.004
```

Finally, let's visualize the Representation Layer at the different stages of learning.

```
In [ ]:  plot_rep(  rep1, rep2, rep3, names_items)
         plot_dendo(rep1, rep2, rep3, names_items)
```

Hierarchical clustering; epoch 500
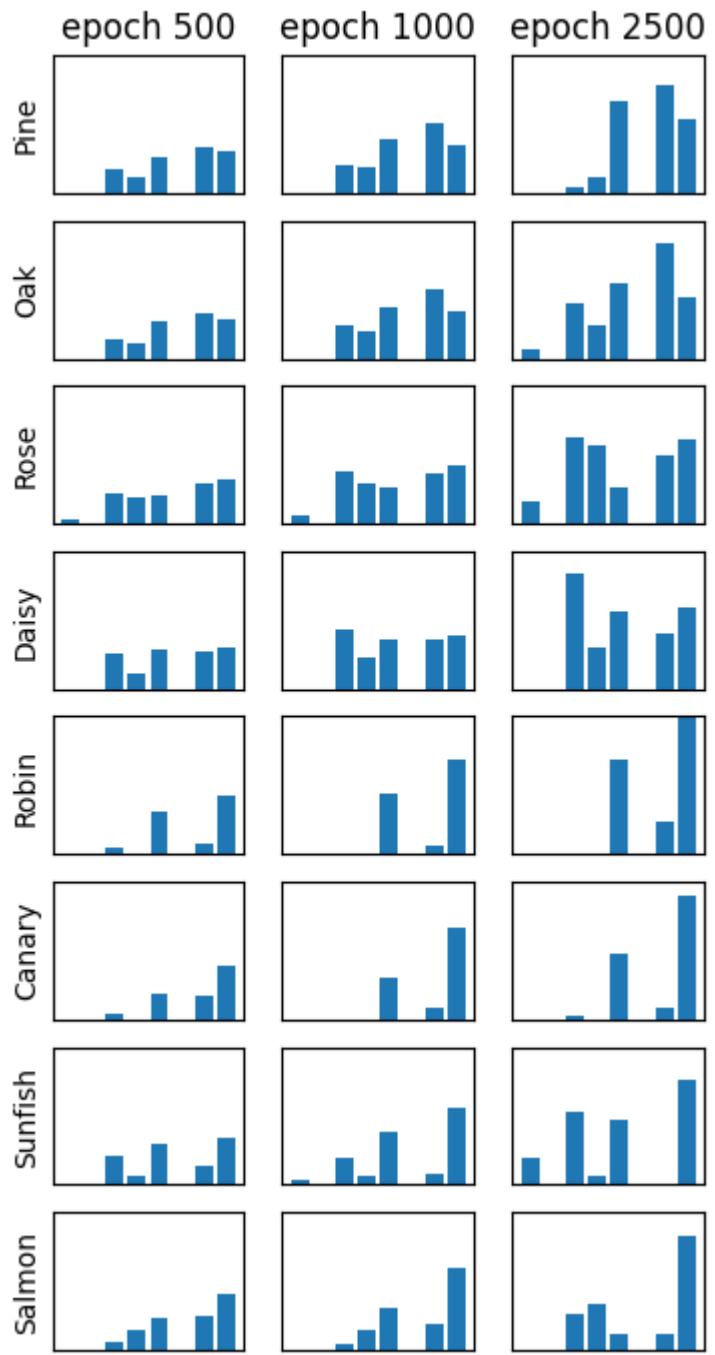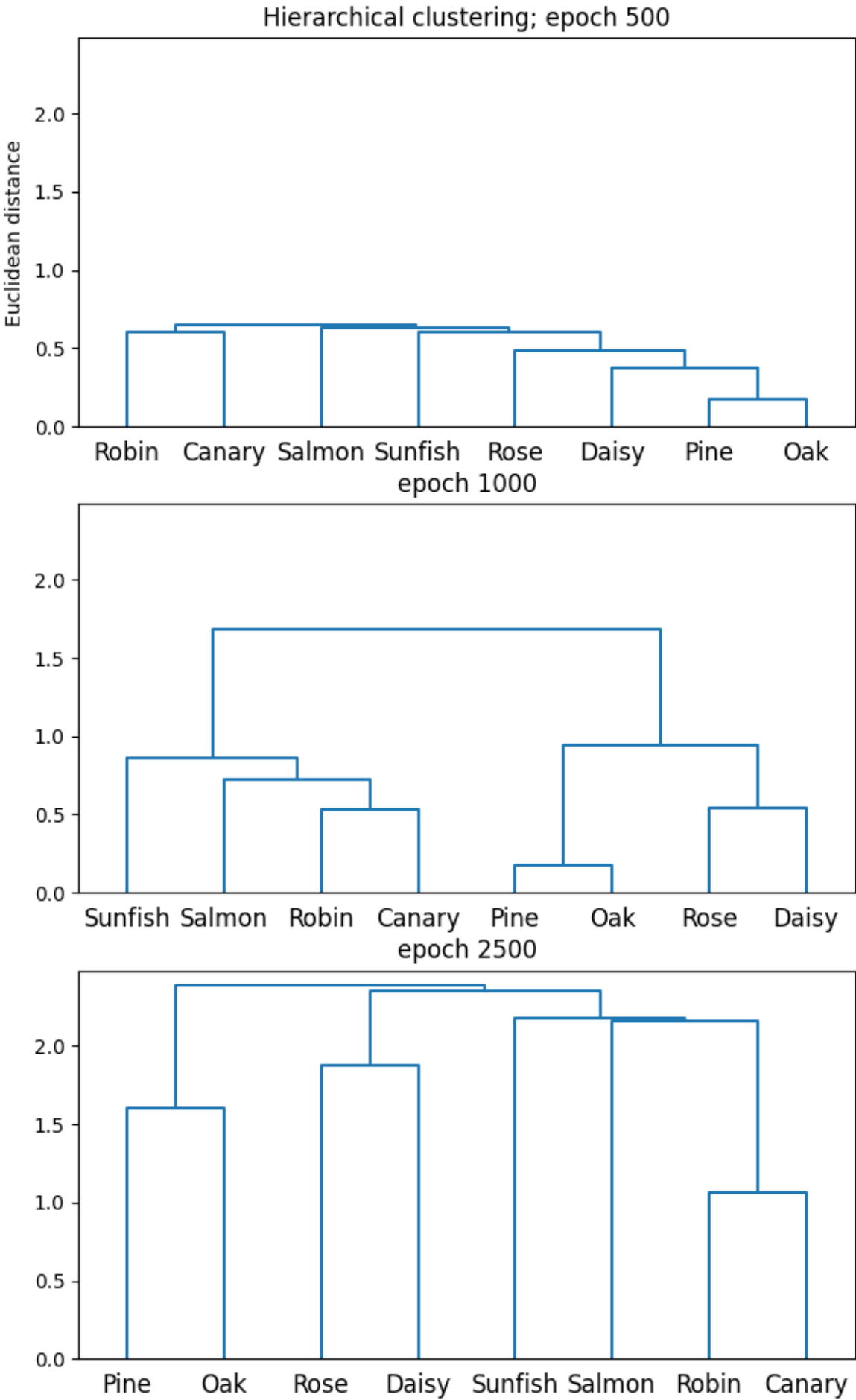
Problem 2 (5 points)

Based on your plots, write a short analysis (4-5 sentences) of how the internal representations of the network develop over the course of learning. How does learning progress? Does the network start by differentiating certain classes of patterns from each other, and then differentiate others in later stages?

Hint: You can refer to your lecture slides and notes for the R&M model for help with your analysis. Your network should broadly replicate their findings, but since the training patterns and activation function aren't identical, don't expect the exact same results.

My Answer:

From the representation graph, we can observe that at epoch 500, the net have divided items into two parts, plants and animals.

I.e., we can see the distinction in the patterns of plants and animals.

Then at epoch 1000, the net is able to classify trees, flowers, birds, and fishes.

Last, at epoch 2500, except for Canary and Robin, the representations of each item have become very unlike from each other.

The reason that the representations of Canary and Robin are quite similar could be that they don't have too many differences. (They are only different at name, color, and that Canary can sing.)

Homework - Neural networks - Part D (20 points)

Predicting similarity ratings with a deep convolutional network

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.

Summary: We will examine whether deep convolutional neural networks (convnets) trained for image classification can predict human similarity ratings for handwritten digits. The images below are all clear examples of the digit '8'. Nonetheless, the left two '8's look more "similar" to me than the right two '8's. Can a neural network trained for classification help to explain judgments such as this one?

In this notebook, we will use a pre-trained convnet for digit recognition to predict similarity judgments. We will then test the network's predictions by collecting similarity ratings from a couple of friends. *This assignment requires collecting a small amount of behavioral data, so grab a friend and don't wait to the last minute to get started!*
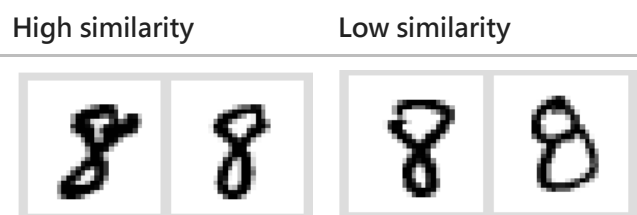
Background

The goal of this assignment is to give you some hands-on experience with a powerful toolkit for computer vision (deep convnets), which has also recently been applied for studying human perception and categorization.

In 2012, Krizhevsky, Sutskever, and Hinton trained a deep convnet (now called 'AlexNet') for object recognition and achieved very impressive results, reducing the number of errors on ImageNet by almost half compared to the next best approach. This paper ignited the recent deep learning revolution in computer vision, although convnets were being used for this purpose long before (LeCun et al. (1989) was the first to train deep convnets for digit recognition).

Recently, the success of convnets has led researchers in cognitive science and neuroscience to explore their potential applications for understanding human visual perception and categorization. Here are some examples: Yamins et al. (2014) showed that deep convnets can predict neural response in high-level visual areas; Lake, Zaremba, Fergus, and Gureckis (2015) showed that deep convnets can explain human typicality ratings for some classes of natural images; Peterson, Abbott, and Griffiths (2016) explored convents for predicting similarity ratings between images.

In this assignment, like Peterson et al., we will explore convnets for predicting human similarity judgments between images. We use a relatively small-scale network trained for digit recognition, but the same principles can be used to study much more complex deep convnets trained for object recognition. Although "similarity" can be a tricky construct in cognitive science (Medin, Goldstone, & Gentner, 1993), similarity judgments are still a useful tool in a cognitive scientist's toolbox for understanding representation.

References:

- Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems.
- Lake, B. M., Zaremba, W., Fergus, R., & Gureckis, T. M. (2015). Deep Neural Networks Predict Category Typicality Ratings for Images. In Proceedings of the Cognitive Science Society.
- LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. Neural Computation, 1(4), 541-551.
- Medin, D. L., Goldstone, R. L., & Gentner, D. (1993). Respects for similarity. Psychological Review, 100(2), 254.
- Peterson, J. C., Abbott, J. T., & Griffiths, T. L. (2016). Adapting deep network features to capture psychological representations. arXiv preprint arXiv:1608.02164.
- Yamins, D. L., Hong, H., Cadieu, C. F., Solomon, E. A., Seibert, D., & DiCarlo, J. J. (2014). Performance-optimized hierarchical models predict neural responses in higher visual cortex. Proceedings of the National Academy of Sciences, 111(23), 8619-8624.

Downloading MNIST

MNIST is one of the most famous data sets in machine learning. It is a digit recognition task, where the goal is classify images of handwritten digits with right label, e.g, either '0','1','2', ... '9'. The training set consists of 60,000 images (6,000 per digit), and the test set of has 10,000 additional images.

Execute the code below to load some libraries and the MNIST "test set." We will not need the training set, since we will use a pre-trained network for this assignment.

```python
In [ ]:  from __future__ import print_function
         %matplotlib inline
         import matplotlib.pyplot as plt
         import torch
         import torch.nn as nn
         import torch.nn.functional as F
         from torchvision import datasets, transforms, utils
         from torch.autograd import Variable
         import os
         import numpy as np
         from scipy.spatial.distance import cosine
         import random

         print('Loading MNIST')
         test_loader = torch.utils.data.DataLoader(datasets.MNIST('~/shared',
                                                          train=False,
                                                          download=True,
                                                          transform=transforms
                                                          transforms.ToTensor(
                                                          transforms.Normalize
                                                   ),batch_size=1000, shuff
         print('MNIST has been loaded.')
```

```
Loading MNIST
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
16.9%
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to C:\Use
rs\Aisu/shared\MNIST\raw\train-images-idx3-ubyte.gz
100.0%
Extracting C:\Users\Aisu/shared\MNIST\raw\train-images-idx3-ubyte.gz to C:\Users
\Aisu/shared\MNIST\raw
100.0%
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to C:\Use
rs\Aisu/shared\MNIST\raw\train-labels-idx1-ubyte.gz
Extracting C:\Users\Aisu/shared\MNIST\raw\train-labels-idx1-ubyte.gz to C:\Users
\Aisu/shared\MNIST\raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
95.4%
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to C:\User
s\Aisu/shared\MNIST\raw\t10k-images-idx3-ubyte.gz
100.0%
100.0%
Extracting C:\Users\Aisu/shared\MNIST\raw\t10k-images-idx3-ubyte.gz to C:\Users\A
isu/shared\MNIST\raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to C:\User
s\Aisu/shared\MNIST\raw\t10k-labels-idx1-ubyte.gz
Extracting C:\Users\Aisu/shared\MNIST\raw\t10k-labels-idx1-ubyte.gz to C:\Users\A
isu/shared\MNIST\raw

MNIST has been loaded.
```

Convnet architecture for digit recognition

Here is the `Net` class defining the network we are using for digit recognition.

First convolutional layer:

This layer takes an image (28x28 for MNIST) and applies a bank of learnable 5x5 filters to produce 10 new images (also known as channels or feature maps). Each feature map is passed through a non-linearity (a rectified linear unit or "ReLU"). Last, there is a max pooling operation that reduces the size of each feature map by half.

Second convolutional layer:

This layer takes the feature maps from the previous layer, applies a bank of learnable 5x5 filters, and produces 20 new feature maps. Again, ReLU's are applied as well as max pooling.

Fully-connected layer:

Next is a standard fully-connected layer of 50 units. At this stage, the entire image is summarized with a vector of size 50. ReLu's are used again.

Output layer: The output layer has 10 units with one to represent each of the 10 digits. It uses a softmax activation function, to ensure the network's predictions are a valid probability distribution of the 10 possibilities.

Execute the code to define the `Net` class.

```python
In [ ]: class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5) # 10 channels in first conv
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5) # 20 channels in second co
        self.fc1 = nn.Linear(320, 50) # 50 hidden units in first fully-connected
        self.fc2 = nn.Linear(50, 10) # 10 output units

    def forward(self, x):

        # first convolutional layer
        h_conv1 = self.conv1(x)
        h_conv1 = F.relu(h_conv1)
        h_conv1_pool = F.max_pool2d(h_conv1, 2)

        # second convolutional layer
        h_conv2 = self.conv2(h_conv1_pool)
        h_conv2 = F.relu(h_conv2)
        h_conv2_pool = F.max_pool2d(h_conv2, 2)

        # fully-connected layer
        h_fc1 = h_conv2_pool.view(-1, 320)
        h_fc1 = self.fc1(h_fc1)
        h_fc1 = F.relu(h_fc1)

        # classifier output
        output = self.fc2(h_fc1)
        output = F.log_softmax(output,dim=1)
        return output, h_fc1, h_conv2, h_conv1
```

Evaluating classification performance of the network

Here we will execute our first bit of non-trivial code. To save time, we already trained a the MNIST network for you. The code below loads the pre-trained model. If you are curious, it only takes a couple minutes to train a MNIST model from scratch on a standard laptop, and we used a version of the PyTorch code from here to train this network.

After loading the network, the function `test_all` iterates through the whole test set and computes the network's predicted class for each image. It outputs the percent correct.

The function `test_viz` picks a few images from the test set at random and displays them along with the network's predicted class labels.

Execute the code below. If everything is working right, **you should get test performance of 98.71**% correct which isn't state-of-the-art, but is reasonable. You can also see some of the network's specific predictions on some test images, which should also look good. You can re-run to see a few different sets of test images.

```python
In [ ]: # Evaluate classification accuracy on the entire MNIST test set
def test_all():
    correct = 0
    for data, target in test_loader:
        # run the data through the network
        output, h_fc1, h_conv2, h_conv1 = model(data)
        # compare prediction to ground truth
        pred = torch.max(output,dim=1)[1] # get the index of the max log-probabi
        correct += torch.eq(pred,target.view_as(pred)).cpu().sum().item()
    perc_correct = 100. * correct / len(test_loader.dataset)
    return perc_correct

# Show the network's predicted class for an arbitrary set of `nshow` images from
def test_viz(nshow=10):

    # grab a random subset of the data
    testiter = iter(test_loader)
    images, target = next(testiter)
    perm = np.random.permutation(images.shape[0])
    data = images[perm[:nshow]]

    # get predictions from the network
    output, h_fc1, h_conv2, h_conv1 = model(data)
    pred = torch.max(output,dim=1)[1]
    pred = pred.numpy().flatten()

    # plot predictions along with the images
    for i in range(nshow):
        ax = plt.subplot(1, nshow, i+1)
        imshow(utils.make_grid(data[i]))
        plt.title(str(pred[i]))

# Display an image from the MNIST data set
```

```python
def imshow(img):
    img = 1 - (img * 0.3081 + 0.1307) # invert image pre-processing
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.axis('off')

checkpoint = torch.load('models/convnet_mnist_pretrained.pt')
model = Net()
model.load_state_dict(checkpoint['state_dict'])
model.eval()
print('Convnet has been loaded successfully.')
print('Running through the test set...')
test_acc = test_all()
print(' Accuracy on the test set is %.2f percent correct!' % test_acc)
print("")
print("Here are some predictions from the network.")
print("The images are shown below their predicted class labels.")
test_viz()
```

```
Convnet has been loaded successfully.
Running through the test set...
 Accuracy on the test set is 98.71 percent correct!

Here are some predictions from the network.
The images are shown below their predicted class labels.
```


Selecting digits for similarity analysis

Here are two more useful functions: `get_random_subset` and `plot_image_pairs.`

As mentioned, we are looking at within-class similarity comparisons (e.g., comparing two different images of the digit '8'). The function `get_random_subset` generates a set of random images pairs from a particular image class `digit_select` from the MNIST test set. The number of random pairs can be set by `npairs` .

The other function is `plot_image_pairs` which visualizes each of the random image pairs. The pairs are input as two list of image tensors, where images with the same index are paired.

```python
In [ ]: def get_random_subset(digit_select, npairs=20):
            # digit_select: which digit do we want to get images for?
            testiter = iter(test_loader)
            images, target = next(testiter)
            indices = np.flatnonzero(target.numpy() == digit_select)
            np.random.shuffle(indices)
            indx1 = torch.tensor(indices[:npairs],dtype=torch.long)
            indx2 = torch.tensor(indices[npairs:npairs*2],dtype=torch.long)
            images1 = images[indx1]
            images2 = images[indx2]
            plt.figure(1,figsize=(4,40))
            plot_image_pairs(images1,images2)
            return images1, images2
```

```python
def plot_image_pairs(images1,images2,scores_net=[],scores_people=[]):
    # images1 : list of images (each image is a tensor)
    # images2 : list of images (each image is a tensor)
    # scores_net (optional) : network similarity score for each pair
    # scores_people (optional) : human similarity score for each pair
    npairs = images1.size()[0]
    assert images2.size()[0] == npairs
    for i in range(npairs):
        ax = plt.subplot(npairs, 1, i+1)
        imshow(utils.make_grid([images1[i], images2[i]]))
        mytitle = ''
        if len(scores_net)>0:
            mytitle += 'net %.2f, ' % scores_net[i]
        if len(scores_people)>0:
            mytitle += 'human. %.2f' % scores_people[i]
        if mytitle:
            plt.title(mytitle)
```

Here is code for sampling 20 random pairings of images of the digit '8' (or whichever digit you set `digit_select` to). Run to sample the pairings and then visualize them (you may have to scroll).

> To save or print out the image pairs, which will be helpful later in the assignment, you can right click and save the output as an image.

```python
In [ ]: images1_digit8,images2_digit8 = get_random_subset(digit_select=8, npairs = 20)
```

Computing similarity judgments with the network

To get predictions out of the network, we take the hidden representation at a particular layer as a representation of an image. The hidden representation computes high-level features of the images, which we will use to compare the images. To compute the similarity between two images, we get the two hidden vectors and compare them with the cosine similarity metric. Cosine similarity measures the angle between two vectors, which can be an effective way of measuring similarity between patterns of activation on the hidden layer.

The function `get_sim_judgments` takes two lists of images and computes the similarity between each pair. Using the `layer` parameter, you can choose between the fully connected layer (`fc`), first convolutional layer (`conv1`), or second convolutional layer (`conv2`).

The function `normalize` rescales a vector to have a minimum value of 0 and maximum value of 1.

```python
# re-scale a vector to have a minimum value of 0, maximum of 1
def normalize(v):
    # v : numpy vector
    v = v - v.min()
    v = v / v.max()
    return v

# Compute convnet similarity for each pair of images
def get_sim_judgments(images1,images2,layer='fc'):
    # images1 : list of N images (each image is a tensor)
    # images2 : list of N images (each image is a tensor)
    # layer : which layer do we want to use? fully connected ('fc'),
    #         first convolutional ('conv1'), or second convolutional ('conv2')
    #
    # Return
    #   v_sim : N-dimensional vector
    N = images1.size()[0] # number of pairs
    assert N == images2.size()[0]
    output_1, h_fc1_1, h_conv2_1, h_conv1_1 = model(images1)
    output_2, h_fc1_2, h_conv2_2, h_conv1_2 = model(images2)

    # grab the tensors from the appropriate layer
    if layer=='fc':
        T1 = h_fc1_1
        T2 = h_fc1_2
    elif layer=='conv1':
        T1 = h_conv1_1
        T2 = h_conv1_2
    elif layer=='conv2':
        T1 = h_conv2_1
        T2 = h_conv2_2
    else:
        raise Exception('Layer parameter has unrecognized value')
```

```python
    # flatten the tensors for each image
    T1 = T1.detach().numpy().reshape(N,-1)
    T2 = T2.detach().numpy().reshape(N,-1)

    v_sim = np.zeros(N)
    for i in range(N): # for each pair
        v1 = T1[i,:]
        v2 = T2[i,:]
        v_sim[i] = 1-cosine(v1,v2) # using cosine distance
    return v_sim

# Get similarity bsaed on the fully conneted layer at the end
v_sim_net_digit8 = get_sim_judgments(images1_digit8,images2_digit8,'fc')
print("Similarity ratings from the neural network the for digit 8:")
print(v_sim_net_digit8)
```

```
Similarity ratings from the neural network the for digit 8:
[0.89012032 0.85038942 0.79451427 0.91285908 0.75113269 0.85352826
 0.8943915  0.91537622 0.89762527 0.77289525 0.75942052 0.81272061
 0.80574378 0.79841236 0.95246642 0.94559149 0.87369754 0.88077555
 0.84055772 0.86749835]
```

Problem 1 (10 points)

Go through the following steps for the digit '8':

1. *Get judgments from the network.* Run all of the code above to sample a list of 20 pairs of images for that digit. For each pair, get a measure of the network's similarity between the images as measured on the fully-connected `fc` layer. (You may have already done all of this.)

2. *Get judgments from people.* Ask two people to rate the same image pairs on a 1 (least similar) to 10 (most similar) scale, using the question "How similar do these two images look to you?" (You can right-click and save the jupyter notebook figures as images). Ask your participant (in a socially distanced way) to rate each pair. Enter the ratings in the arrays `ratings_human_1_digit8` and `ratings_human_2_digit8` below. One participant can be yourself, but the other should be someone else.

3. *Compare.* Run the code below to compute the correlation coefficient and scatter plot.

```python
In [ ]: # YOUR PARTICIPANT RATINGS GO HERE
ratings_human_1_digit8 = np.array([5,8,8,8,5,6,9,6,6,7,5,7,10,8,4,7,7,8,9,7], dt
ratings_human_2_digit8 = np.array([5,7,8,9,4,8,9,6,6,8,7,8,10,8,2,8,6,9,8,6], dt

# Analysis code
ratings_human_mean_digit8 = (ratings_human_1_digit8 + ratings_human_2_digit8)/2

v_sim_net_norm = normalize(v_sim_net_digit8)
```

```
v_sim_human_norm = normalize(ratings_human_mean_digit8)

print("Correlation between net and human similarity ratings: r ="),
print(round(np.corrcoef(v_sim_net_norm,v_sim_human_norm)[0][1],3))

# Scatter plot
plt.figure(1,figsize=(5,5))
plt.plot(v_sim_net_norm,v_sim_human_norm,'k.')
plt.xlabel('network similarity (normalized)')
plt.ylabel('human similarity (normalized)')
plt.title('Convnet vs. human similarity judgments (digit 8)')
plt.show()

# pairs with similarity ratings
plt.figure(2,figsize=(4,40))
plot_image_pairs(images1_digit8,images2_digit8,v_sim_net_norm,v_sim_human_norm)
plt.show()
```
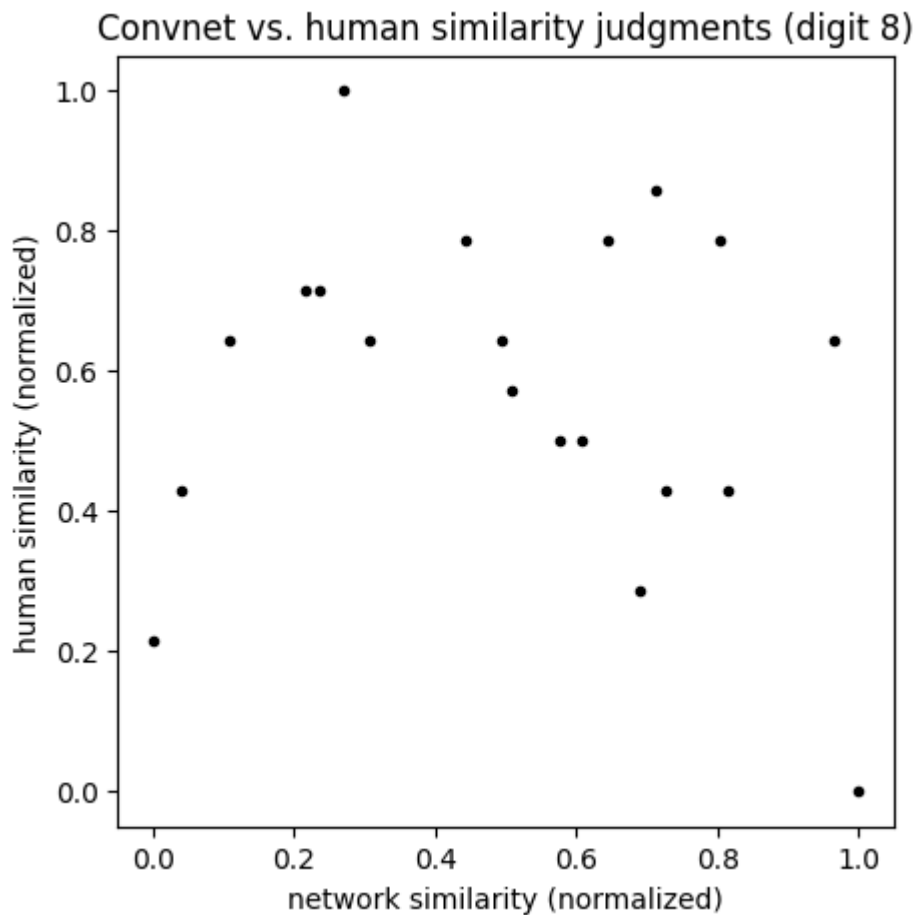
Correlation between net and human similarity ratings: r =
-0.171

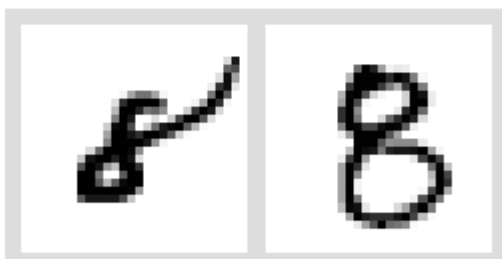net 0.69, human. 0.29

net 0.49, human. 0.64

net 0.22, human. 0.71

net 0.80, human. 0.79

net 0.00, human. 0.21

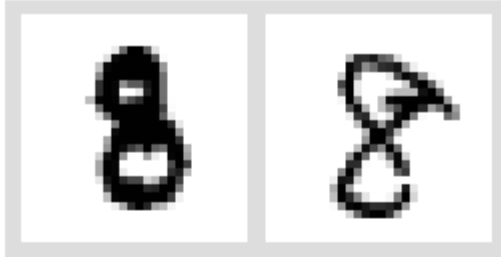net 0.51, human. 0.57

net 0.71, human. 0.86

net 0.82, human. 0.43

net 0.73, human. 0.43

net 0.11, human. 0.64

net 0.04, human. 0.43

net 0.31, human. 0.64

net 0.27, human. 1.00

net 0.23, human. 0.71

net 1.00, human. 0.00

net 0.97, human. 0.64

net 0.61, human. 0.50

net 0.64, human. 0.79

net 0.44, human. 0.79

net 0.58, human. 0.50

Problem 2 (10 points)

How well does the convolutional neural network do at predicting the similarity ratings? What is it capturing? What is it not capturing? (Keep your answer brief. About 5 sentences is good.)

Don't worry if your fit isn't very good! A convnet is very far from a perfect account of human similarity ratings. I had reasonable success with the `8` digit but I wouldn't except a correlation higher than r=0.6. This assignment is not graded on how good the fit is.

My Answer:

The convnet did terrible on the similarity predicting.

The most impressive case is the 13th and the 15th pairs. The result is far from the human's intuition.

It is so difficult to figure out what does the model actually trying to capture.

I can barely tell what the model is doing from the most confident two pairs, the 15th and the 16th, which the model gives almost 100% similarity ratings.

I would guess that the model is giving the ratings by comparing the angles of the numbers in the image pairs.

Homework - Neural networks - Part E (50 points)

Discovering lexical classes from simple sentences

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.

In this assignment, you will follow in Elman's (1990) footsteps by coding and training a Simple Recurrent Network (SRN) on a set of simple sentences.

- **Before training**, the SRN can process sequences but otherwise knows nothing about language. Initially, it represents each word as an arbitrary continuous vector (input embedding) without knowledge of their roles or how they relate to each other.
- **During training**, the SRN aims to predict the next word in a sentence given the previous words. The optimizer takes a step after each sentence.
- **After training**, you will analyze the SRN's internal representations (input embeddings) for evidence that it has discovered something about lexical classes (e.g., nouns and verbs).

> Reference (available for download on Brightspace):
> Elman, J. L. (1990). Finding Structure in Time. Cognitive Science, 14:179–211.

```python
In [ ]:  # Let's start with some packages we need
         from __future__ import print_function
         import torch
         import torch.nn as nn
         import numpy as np
         import matplotlib
         %matplotlib inline
         import matplotlib.pyplot as plt
         from scipy.cluster.hierarchy import dendrogram, linkage

         import random
         from tqdm import tqdm
```

Elman's set of simple sentences

The training set consists of 10,000 sentences each with 2 or 3 words. Elman generated each sentence as follows:

1. Choose one of 16 templates specificying a sequence of lexical classes (see below).
2. Each lexical class is replaced by a word sampled from that class (see below, only a subset of words shown).

The vocabulary contained 29 words. For example, the template `NOUN-AGRESS VERB-EAT NOUN-FOOD` can lead to the sentence `dragon eat cookie` along with other possibilities. We generated 10,000 sentences using our best guess of Elman's procedure (the full set of lexical classes isn't listed). You can see these sentences in the external text file `data/elman_sentences.txt`

TABLE 3
Categories of Lexical Items Used in Sentence Simulation

Category	Examples
NOUN-HUM	man, woman
NOUN-ANIM	cat, mouse
NOUN-INANIM	book, rock
NOUN-AGRESS	dragon, monster
NOUN-FRAG	glass, plate
NOUN-FOOD	cookie, break
VERB-INTRAN	think, sleep
VERB-TRAN	see, chase
VERB-AGPAT	move, break
VERB-PERCEPT	smell, see
VERB-DESTROY	break, smash
VERB-EAT	eat

TABLE 4
Templates for Sentence Generator

WORD 1	WORD 2	WORD 3
NOUN-HUM	VERB-EAT	NOUN-FOOD
NOUN-HUM	VERB-PERCEPT	NOUN-INANIM
NOUN-HUM	VERB-DESTROY	NOUN-FRAG
NOUN-HUM	VERB-INTRAN	
NOUN-HUM	VERB-TRAN	NOUN-HUM
NOUN-HUM	VERB-AGPAT	NOUN-INANIM
NOUN-HUM	VERB-AGPAT	
NOUN-ANIM	VERB-EAT	NOUN-FOOD
NOUN-ANIM	VERB-TRAN	NOUN-ANIM
NOUN-ANIM	VERB-AGPAT	NOUN-INANIM
NOUN-ANIM	VERB-AGPAT	
NOUN-INANIM	VERB-AGPAT	
NOUN-AGRESS	VERB-DESTROY	NOUN-FRAG
NOUN-AGRESS	VERB-EAT	NOUN-HUM
NOUN-AGRESS	VERB-EAT	NOUN-ANIM
NOUN-AGRESS	VERB-EAT	NOUN-FOOD

Loading the data

The following code will load and process the set of simple sentences. As is common in neural networks for text and natural language processing, the sentence strings are first "tokenized" into a list of discrete elements (words in this case). Additionally, special tokens indicating the start-of-sentence `<SOS>` and end-of-sentence `<EOS>` are added at the beginning and end of the sentence, respectively. The SRN requires an input at every step and thus we use `<SOS>` as the first input when the SRN is predicting the first word as output. The SRN can self-terminate a sentence by producing `<EOS>` as an output. The dict `token_to_index` maps each token to a unique integer, which is the format that the SRN actually uses as input.

Running the code below will show you the dict `token_to_index` and how the first sentence `dragon break plate` is tokenized into integers. Make sure you understand how this works and how to map back and forth between the formats!

```python
In [ ]: def sentenceToTensor(tokens_list):
            # Convert list of strings to tensor of token indices (integers)
            #
            # Input
            #  tokens_list : list of strings, e.g. ['<SOS>','lion','eat','man','<EOS>']
            # Output
            #  1D tensor of the same length (integers), e.g., tensor([ 1, 17, 12, 18,  0
            assert(isinstance(tokens_list,list))
            tokens_index = [token_to_index[token] for token in tokens_list]
            return torch.tensor(tokens_index)

        # load and process the set of simple sentences
        with open('data/elman_sentences.txt','r') as fid:
            lines = fid.readlines()
        sentences_str = [l.strip() for l in lines]
        sentences_tokens = [s.split() for s in sentences_str]
        sentences_tokens = [['<SOS>']+s+['<EOS>'] for s in sentences_tokens]
        unique_tokens = sorted(set(sum(sentences_tokens, [])))
        n_tokens = len(unique_tokens) # all words and special tokens
        token_to_index = {t : i for i,t in enumerate(unique_tokens)}
        index_to_token = {i : t for i,t in enumerate(unique_tokens)}
        training_pats = [sentenceToTensor(s) for s in sentences_tokens] # python list of
        ntrain = len(training_pats)
        print('mapping unique tokens to integers: %s \n' % token_to_index)
        print('example sentence as string: %s \n' % ' '.join(sentences_tokens[0]))
        print('example sentence as tensor: %s \n' % training_pats[0])
```

```
mapping unique tokens to integers: {'<EOS>': 0, '<SOS>': 1, 'book': 2, 'boy': 3,
'bread': 4, 'break': 5, 'car': 6, 'cat': 7, 'chase': 8, 'cookie': 9, 'dog': 10,
'dragon': 11, 'eat': 12, 'exist': 13, 'girl': 14, 'glass': 15, 'like': 16, 'lio
n': 17, 'man': 18, 'monster': 19, 'mouse': 20, 'move': 21, 'plate': 22, 'rock': 2
3, 'sandwich': 24, 'see': 25, 'sleep': 26, 'smash': 27, 'smell': 28, 'think': 29,
'woman': 30}

example sentence as string: <SOS> dragon break plate <EOS>

example sentence as tensor: tensor([ 1, 11,  5, 22,  0])
```
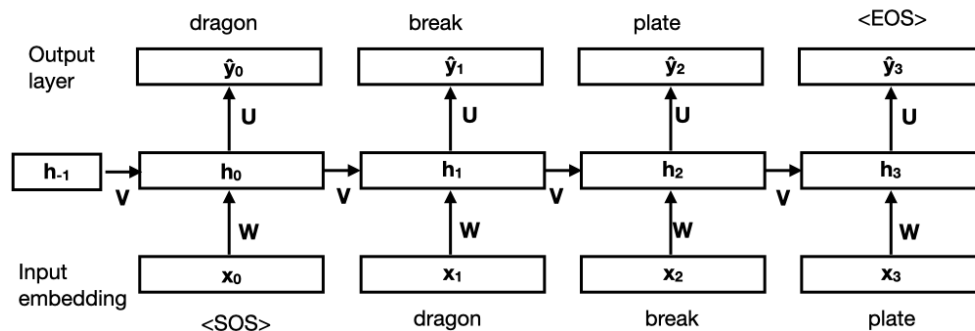
Simple Recurrent Network

The diagram below shows the unrolled SRN that you will develop here. As is always true for recurrent networks, notice the tied weights U, W, V, etc.

We will deviate from Elman's exact model in a few ways to make it more modern. Here is the specification we will use.

- **Input embedding**. In Elman's original model, each word was represented by a fixed one-hot input vector. Instead, here we will learn a continuous embedding vector (size `hidden_size=20`) to represent each input word. These vectors are learnable parameters. When a word is provided as input to the SRN, it is converted to the corresponding input embedding. This layer is setup for you already in the started class, `self.embed = nn.Embedding(vocab_size,hidden_size)`
- **Hidden layer**. This layer has length `hidden_size` and uses the **logistic** activation function. The initial vector h_{-1} should be all zeros.
- **Output layer**. This layer has length `vocab_size` and uses the **softmax** activation function. Thus, the SRN will represent an explicit probability distribution over the next token w_j given the past tokens $w_1, \dots, w_{j-1}$, through the equation $P(w_j | w_1, \dots, w_{j-1})$
- **Loss**. The SRN will train to maximize the log-likelihood of the target output words, e.g., we use the negative log-likelihood loss `nn.NLLLoss`. If passed a tensor representing multiple target predictions, this loss takes the mean across predictions.
- **Optimizer**. We found reasonable results with the `AdamW` optimizer with weight decay of 0.04. Adam is like stochastic gradient descent but adapts the learning rate for each parameter based on the variance of the gradient. Weight decay encourages the parameters to be close to zero leading to more stable input embeddings.
- **Batching**. We suggest *no batching* for this simple code. Thus, the optimizer takes a step after each individual sentence. The `forward` method should process only one input word at a time. Batching produces much faster code and is recommended in practice, but it's not required here. If you want to rewrite the

code to process multiple timesteps and sentences simultaneously, that's fine too.

Problem 1 (20 points)

Write code to complete the SRN class.

```
In [ ]:  class SRN(nn.Module):

             def __init__(self, vocab_size, hidden_size):
                 # vocab_size : number of tokens in vocabulary including special tokens <
                 # hidden_size : dim of input embeddings and hidden layer
                 super().__init__()
                 self.vocab_size = vocab_size
                 self.hidden_size = hidden_size
                 self.embed = nn.Embedding(vocab_size, hidden_size)
                 self.hidden_layer = nn.Linear(hidden_size*2, hidden_size)
                 self.output_layer = nn.Linear(hidden_size, vocab_size)

             def forward(self, input_token_index: int,
                         hidden_prev: torch.Tensor) -> tuple[torch.Tensor, torch.Tensor]:
                 # Input
                 #    input_token_index: [integer] index of current input token
                 #    hidden_prev: [length hidden_size 1D tensor] hidden state from previ
                 # Outpuut
                 #    output: [length vocab_size 1D tensor] log-probability of emitting e
                 #    hidden_curr : [length hidden_size 1D tensor] hidden state for curre
                 input_embed = self.embed(input_token_index) # hidden_size 1D tensor
                 hidden_curr = nn.Sigmoid()(self.hidden_layer(torch.cat([input_embed, hid
                 output = nn.LogSoftmax(dim=0)(self.output_layer(hidden_curr))
                 return output, hidden_curr

             def initHidden(self) -> torch.Tensor:
                 # Returns length hidden_size 1D tensor of zeros
                 return torch.zeros(self.hidden_size)

             def get_embeddings(self):
                 # Returns [vocab_size x hidden_size] numpy array of input embeddings
                 return self.embed(torch.arange(self.vocab_size)).detach().numpy()
```

Problem 2 (20 points)

Write code to complete the `train` function and the main training loop. In the training loop, for each epoch, print out the mean loss over all training patterns. An epoch should visit each sentence in random order, taking an optimizer step after each sentence.

Hint: In my implementation, after 10 epochs, I found that the mean loss to reach about 1.57. In other words, the SRN predicts the right word with roughly

$e^{-1.57} = 0.208$ probability of getting it right. (Of course, perfect prediction is impossible in even this simple language).

In [ ]:
```python
def train(seq_tensor: torch.Tensor, rnn: SRN) -> float:
    # Process a sentence and update the SRN weights. With <SOS> as the input at
    # predict every subsequent word given the past words.
    # Return the mean loss across each symbol prediction.
    #
    # Input
    #   seq_tensor: [1D tensor] sentence as token indices
    #   rnn : instance of SRN class
    # Output
    #   loss : [scalar] average NLL loss across prediction steps
    output_tensors = []
    hidden_prev = rnn.initHidden()
    rnn.zero_grad()
    for input_token_index in seq_tensor[:-1]:
        output, hidden_prev = rnn(input_token_index, hidden_prev)
        output_tensors.append(output)
    output_tensors = torch.stack(output_tensors)
    loss = criterion(output_tensors, seq_tensor[1:])
    loss.backward()
    optimizer.step()
    return loss.item()
```

In [ ]:
```python
# Main training loop
nepochs = 10 # number of passes through the entire training set
nhidden = 20 # number of hidden units in the SRN
rnn = SRN(n_tokens, nhidden)
optimizer = torch.optim.AdamW(rnn.parameters(), weight_decay=0.04) # w/ default
criterion = nn.NLLLoss()
for epcoh in range(1, nepochs+1):
    loss = 0
    random.shuffle(training_pats)
    for x_pat in tqdm(training_pats):
        loss += train(x_pat, rnn)
    print(f"Epoch {epcoh}/{nepochs} | Avg Loss: {loss/len(training_pats)}")
```

```
100%|██████████| 10000/10000 [00:24<00:00, 414.59it/s]
Epoch 1/10 | Avg Loss: 1.7663621869325639
100%|██████████| 10000/10000 [00:23<00:00, 431.72it/s]
Epoch 2/10 | Avg Loss: 1.587000988471508
100%|██████████| 10000/10000 [00:43<00:00, 227.44it/s]
Epoch 3/10 | Avg Loss: 1.5797672462940215
100%|██████████| 10000/10000 [00:48<00:00, 206.01it/s]
Epoch 4/10 | Avg Loss: 1.5778505248904229
100%|██████████| 10000/10000 [00:47<00:00, 208.54it/s]
Epoch 5/10 | Avg Loss: 1.577480607676506
100%|██████████| 10000/10000 [00:47<00:00, 209.09it/s]
Epoch 6/10 | Avg Loss: 1.5766222214341163
100%|██████████| 10000/10000 [00:48<00:00, 205.17it/s]
Epoch 7/10 | Avg Loss: 1.5750796798348428
100%|██████████| 10000/10000 [00:47<00:00, 209.02it/s]
Epoch 8/10 | Avg Loss: 1.5749413407564163
100%|██████████| 10000/10000 [00:45<00:00, 217.73it/s]
Epoch 9/10 | Avg Loss: 1.5746542495250702
```

```
100%|████████████| 10000/10000 [00:42<00:00, 232.79it/s]
Epoch 10/10 | Avg Loss: 1.5743791903853417
```

Analyze the SRN internal representations

Once training is done, we want to examine the internal representations to see what the network has learned about the lexical items. Elman ran a hierarchical clustering analysis using the mean hidden representation of each word when presented across the corpus.
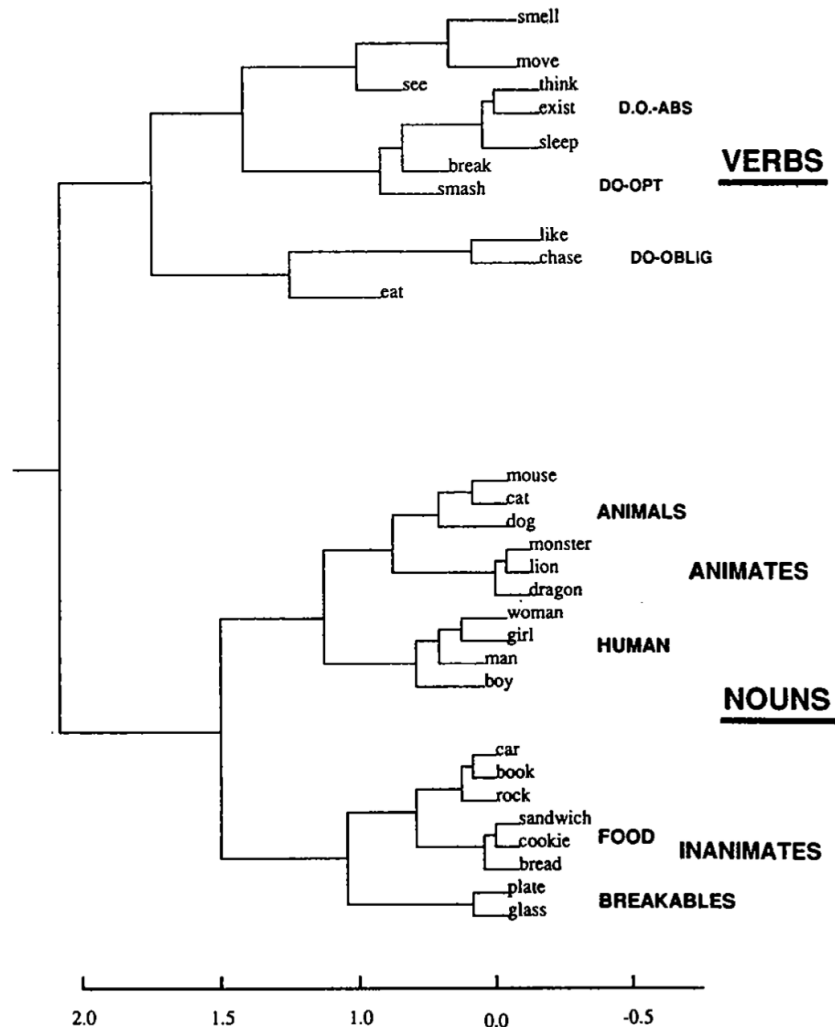
Figure 7. Hierarchical cluster diagram of hidden unit activation vectors in simple sentence prediction task. Labels indicate the inputs which produced the hidden unit vectors; inputs were presented in context, and the hidden unit vectors averaged across multiple contexts.

Unlike Elman we have an **explicit input embedding** for each word, and thus we can more simply look at these embedding vectors. Run the code to compare with Elman's results. *You shouldn't expect a close match.* There are differences in network architecture, training, and the dataset. Still, it's interesting to see what your SRN has learned.

```
In [ ]: def plot_dendo(X, names, exclude=['<SOS>','<EOS>']):
            #  Show hierarchical clustering of vectors
            #
            # Input
```

```python
    #  X : numpy tensor [nitem x dim] such that each row is a vector to be clust
    #  names : [length nitem] list of item names
    #  exclude: list of names we want to exclude
    nitem = len(names)
    names  = np.array(names)
    include = np.array([myname not in exclude for myname in names], dtype=bool)
    linked = linkage(X[include],'single', optimal_ordering=True)
    plt.figure(1, figsize=(20,6))
    dendrogram(linked, labels=names[include], color_threshold=0, leaf_font_size=
    plt.show()

plot_dendo(rnn.get_embeddings(), unique_tokens)
```
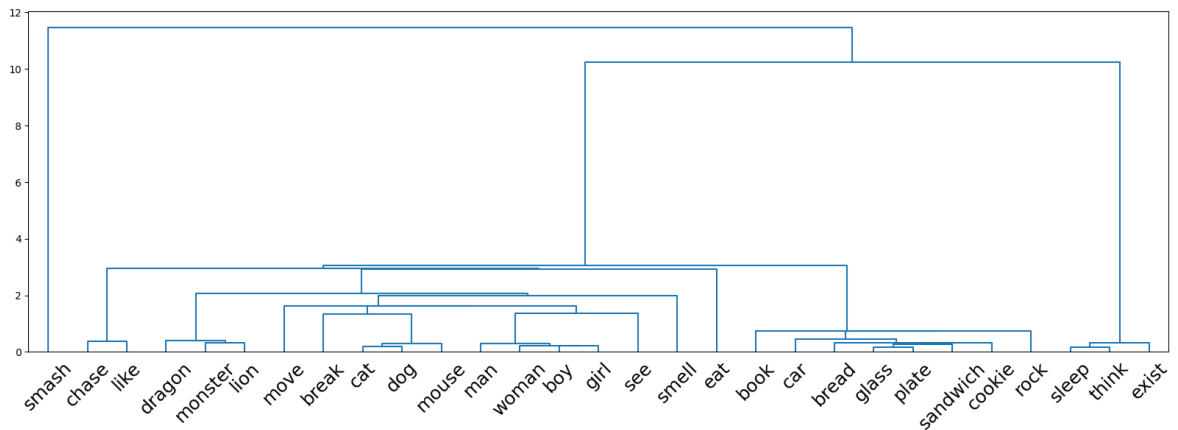

Problem 3 (10 points)

Write a function `generate` to probabilistically sample sentences from your network. Generate 10 sample sentences in this manner. For each, convert the sequence of token indices back to string form. When printing the sentence, you can either include the SOS and EOS or ignore them. It's fine to assume a maximum length.

Hint: You will find `torch.distributions.categorical.Categorical` useful.

```python
In [ ]: def generate(rnn: SRN, maxlen=4):
    sentence_tkns = []
    hidden_prev = rnn.initHidden()
    for _ in range(maxlen-1):
        output, hidden_prev = rnn.forward(torch.tensor(1), hidden_prev)
        categorical = torch.distributions.categorical.Categorical(nn.Softmax()(o
        random_sampled_tkn = categorical.sample()
        sentence_tkns.append(int(random_sampled_tkn.detach().numpy()))
        if random_sampled_tkn == 0:
            break
    sentence = ' '.join([index_to_token[tkn] for tkn in sentence_tkns])
    print(sentence)

for i in range(10):
    generate(rnn)
```

```
girl eat lion
car eat man
woman eat mouse
dragon smash lion
dragon exist man
boy eat lion
monster sleep boy
lion eat woman
boy break cat
girl move dragon
```