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/ (https://brendenlake.github.io/CCM-site/)

This homework is due before midnight on Monday, Feb. 22, 2021.

```
Elena Georgieva
elena@nyu.edu
02/22/2021
```

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 (https://web.stanford.edu/group/pdplab/pdphandbook/) 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 (https://cims.nyu.edu/~brenden/software/IAC11final.app.zip) and here for Windows (https://cims.nyu.edu/~brenden/software/IACwindows64.zip).

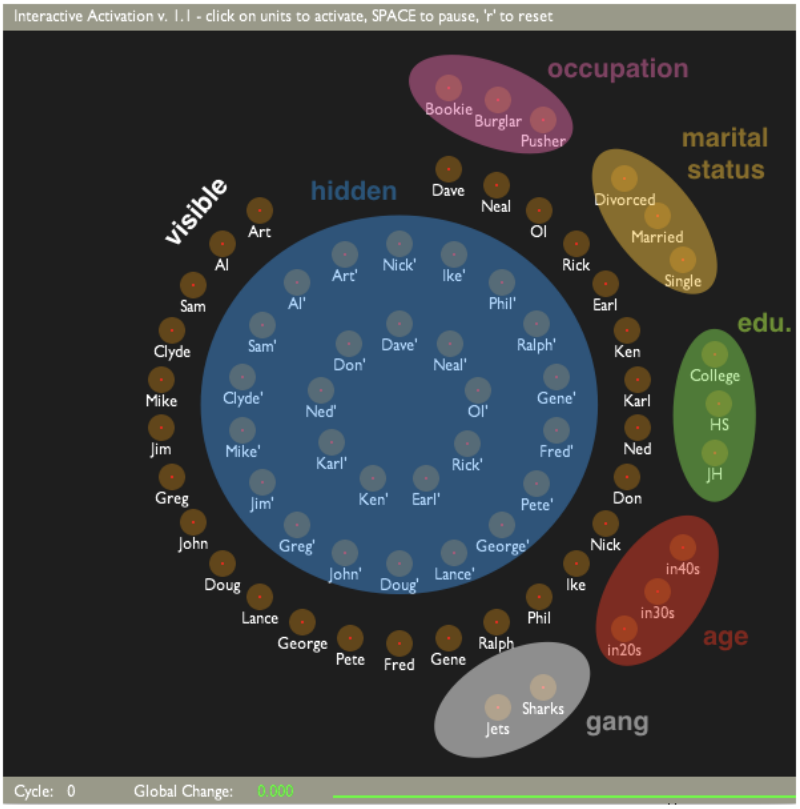


Figure 2: Screen shot from Cleeremans' IAC software. Units are organized into 7 groups. For illustration here, all groups have a different color background, while the group of visible name units have no background.

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.

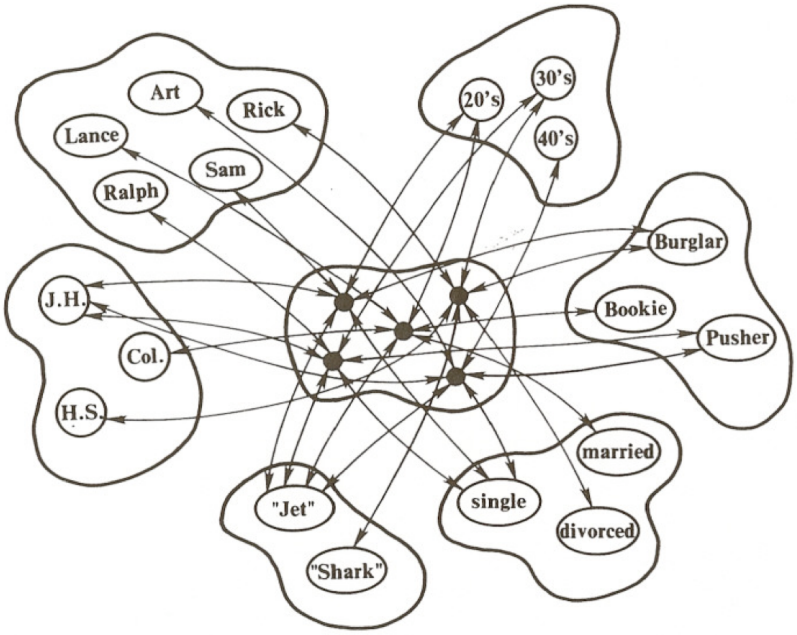


Figure 3: The units and connections for some of the individuals in Figure 1. The arrows represent excitatory connections. The outlined groups of units have mutually inhibitory connections (not shown). (From "Retrieving General and Specific Knowl- edge From Stored Knowledge of Specifics" by J. L. McClelland, 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).

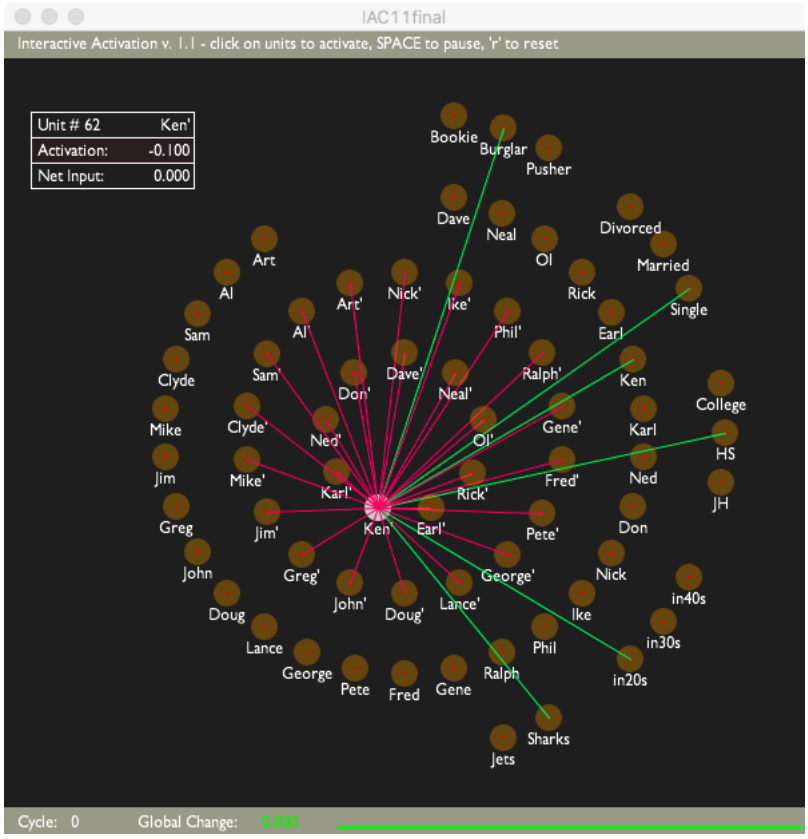


Figure 4: You can view the connections to a unit by placing your mouse over it. Green connections are excitatory and red connections are inhibitory.

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.

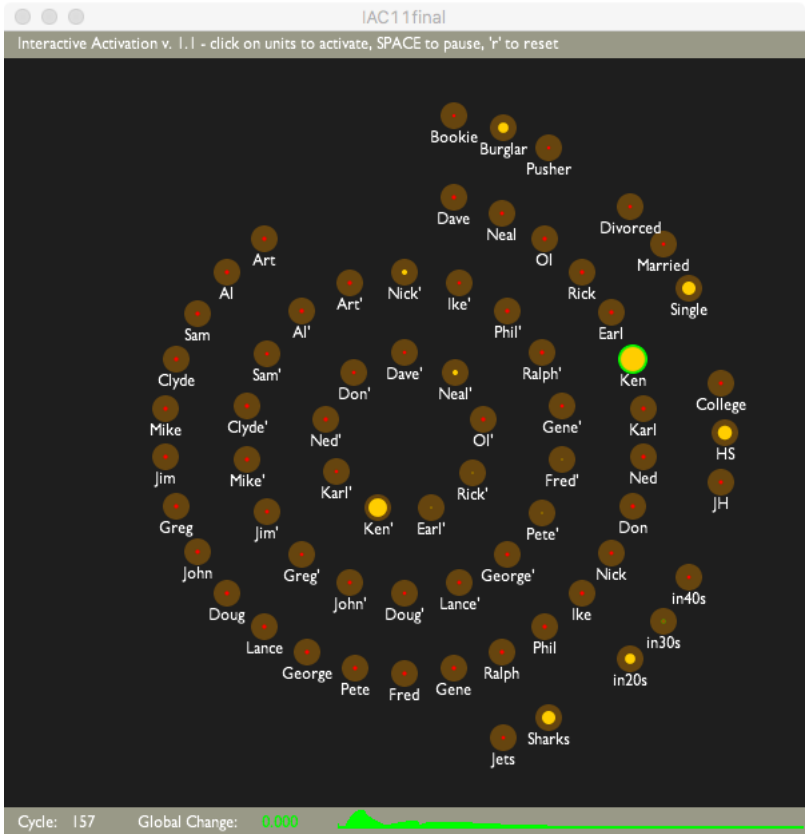


Figure 5: The display screen after about 150 cycles with external input to the name unit for Ken.

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).

Problem 1: The hidden instance units of a few others are active, including 'Nick' and 'Neal.' Those units are excited because they are supported by property units including 'HS' and 'single.' They share those properties with Ken.

Despite this, Ken's unit is more strongly activated than these others because there is also inhibition coming from Ken's unit.

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

short (about 3 sentences).

Problem 2: Some of Ken's properties are activated more strongly than others, for example 'HS' is most activated. This is because several instance units other than 'Ken' are active, for example Nick and Neal, and those send activation to their property units. These other units reinforce the activation of the properties they share with Ken.

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).

Problem 3: In addition to 'Burglar,' 'Bookie' and 'Pusher' are partially activated. Activating 'Sharks' and 'in20s' like we did activated a few names other than Ken, including Neal, Fred, Pete, and Nick. Some of those individuals have the properties 'Bookie' and 'Pusher,' that is hwy those units are partially active.

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).

Problem 4: Highlighting 'Jets' activates cells including 'Single,' 'JH,' 'in20s,' and it slighlty activates all three of the occupations. It does an okay job generalizing. However, over time, it highlights the names of individuals who are the most typical Jets. Therefore it seems like it gives an idea of what a Jet is like, but that idea is biased and a generalization. Tools like this should be used with caution.

Homework - Neural networks - Part B (45 points)

Gradient descent for simple two and three layer models

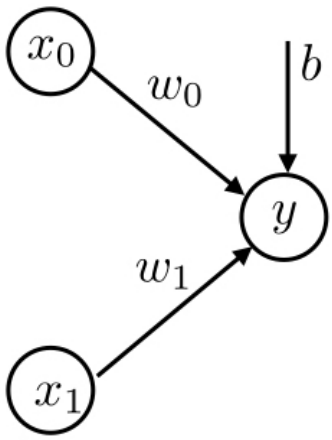
by *Brenden Lake* and *Todd Gureckis*
Computational Cognitive Modeling
NYU class webpage: https://brendenlake.github.io/CCM-site/ (https://brendenlake.github.io/CCM-site/)

This homework is due before midnight on Monday, Feb. 22, 2021.

```
Elena Georgieva
elena@nyu.edu
02/22/2021
```

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

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



logical OR

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

This assignment requires some basic PyTorch knowledge. You can review your notes from lab and also two basic PyTorch tutorials (https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html), "What is PyTorch?" and "Autograd", which should have the basics you need.

In [2]:
```
# Import libraries
from __future__ import print_function
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
```

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

In [3]:
```
# Data
D = np.zeros((4,2),dtype=float)
D[0,:] = [0.,0.]
D[1,:] = [0.,1.]
D[2,:] = [1.,0.]
D[3,:] = [1.,1.]
D = torch.tensor(D,dtype=torch.float)
Y_or = torch.tensor([0.,1.,1.,1.])
N = D.shape[0] # number of input patterns
```

The artificial neuron operates as follows. Given an input vector 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 [4]:
```
def g_logistic(net):
    return 1. / (1.+torch.exp(-net))

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

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

In [5]:
```
def print_forward(x,yhat,y):
    # Examine network's prediction for input x
    print(' Input: ',end='')
    print(x.numpy())
    print(' Output: ' + str(round(yhat.item(),3)))
    print(' Target: ' + str(y.item()))

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

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

Problem 1 (10 points)

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

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

Problem 2 (5 points)

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

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

In [6]:
```
# Initialize parameters
#     Although you will implement gradient descent manually, let's set requ
#     anyway so PyTorch will track the gradient too, and we can compare you
w = torch.randn(2) # [size 2] tensor
b = torch.randn(1) # [size 1] tensor
w.requires_grad = True
b.requires_grad = True

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

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

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

        # compute loss
        y = Y_or[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,yhat,y)
            print("")

        # Compute the gradient manually
        if verbose:
            print('Compute the gradient manually')
            print_forward(x,yhat,y)
        with torch.no_grad():
            # TODO : YOUR GRADIENT CODE GOES HERE
            #  two lines of the form
            w_grad = 2*(yhat - y)*yhat*(1-yhat)*x    #([size 2] PyTorch ten
            b_grad = 2*(yhat-y)*yhat*(1-yhat)    #([size 1] PyTorch tensor)
            #  make sure to inclose your code in the "with torch.no_grad()"
            #   otherwise PyTorch will try to track the "gradient" of the g
            # raise Exception('Replace with your code.')
        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():
            # TODO : YOUR PARAMETER UPDATE CODE GOES HERE
            #  two lines of the form:
            w -= alpha * w_grad
            b -= alpha * b_grad
            #raise Exception('Replace with your code.')

    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 (logistic activation)')
plt.ylabel('error for epoch')
plt.xlabel('epoch')
plt.show()
```

```
Compute the gradient manually
 Input: [0. 0.]
 Output: 0.63
 Target: 0.0
  d_loss / d_w = [0. 0.]
  d_loss / d_b = [0.29371554]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [0. 0.]
  d_loss / d_b = [0.29371554]

Compute the gradient manually
 Input: [1. 0.]
 Output: 0.754
 Target: 1.0
  d_loss / d_w = [-0.09147369 -0.        ]
  d_loss / d_b = [-0.09147369]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [-0.09147366  0.        ]
  d_loss / d_b = [-0.09147366]
```

Now let's change the activation function to "tanh" from the "logistic" function, such that $g(\mathbf{net}) = \tanh(\mathbf{net})$. Here is an implementation of tanh.

In [7]:
```
def g_tanh(x):
    return (torch.exp(x) - torch.exp(-x))/(torch.exp(x) + torch.exp(-x))
```

The derivative of the tanh function is as follows:

$$\frac{\partial g(\mathbf{net})}{\partial \mathbf{net}} = \frac{\partial \tanh(\mathbf{net})}{\partial \mathbf{net}} = 1.0 - (\tanh(\mathbf{net}))^2$$

Problem 3 (5 points)

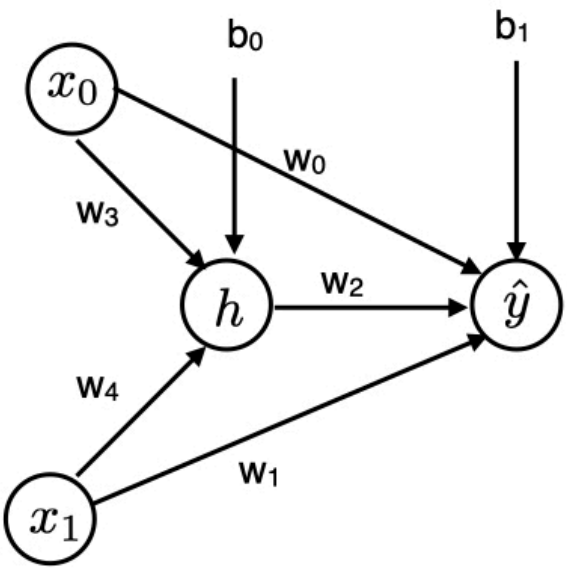
Just as before, fill in the missing code fragments for implementing gradient descent. This time we are using the tanh activation function. Be sure to change your gradient calculation to reflect the new activation function.

In [8]:
```
# Initialize parameters
#     Although you will implement gradient descent manually, let's set requ
#     anyway so PyTorch will track the gradient too, and we can compare you
w = torch.randn(2) # [size 2] tensor
b = torch.randn(1) # [size 1] tensor
w.requires_grad = True
b.requires_grad = True

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

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

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

        # compute loss
        y = Y_or[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,yhat,y)
            print("")

        # Compute the gradient manually
        if verbose:
            print('Compute the gradient manually')
            print_forward(x,yhat,y)
        with torch.no_grad():
            # TODO : YOUR GRADIENT CODE GOES HERE
            #  two lines of the form
            w_grad = 2*(yhat - y)*(1-(yhat)**2)*x    # ([size 2] PyTorch te
            b_grad = 2*(yhat - y)*(1-(yhat)**2)  # ([size 1] PyTorch tensor
            #  make sure to inclose your code in the "with torch.no_grad()"
            #   otherwise PyTorch will try to track the "gradient" of the g
            #raise Exception('Replace with your code.')
        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():
            # TODO : YOUR PARAMETER UPDATE CODE GOES HERE
            #  two lines of the form:
            w -= w_grad*alpha
            b -= b_grad*alpha
            # raise Exception('Replace with your code.')

    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 (tanh activation)')
plt.ylabel('error for epoch')
plt.xlabel('epoch')
plt.show()
```

```
Compute the gradient manually
 Input: [0. 0.]
 Output: -0.146
 Target: 0.0
  d_loss / d_w = [-0. -0.]
  d_loss / d_b = [-0.28637198]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [-0. -0.]
  d_loss / d_b = [-0.286372]

Compute the gradient manually
 Input: [0. 1.]
 Output: 0.987
 Target: 1.0
  d_loss / d_w = [-0.        -0.0007168]
  d_loss / d_b = [-0.0007168]
Compute the gradient using PyTorch .backward()
  d_loss / d_w = [ 0.        -0.0007168]
  d_loss / d_b = [-0.0007168]
```

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 4 (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 [29]:
```
# New data D and labels y for xor
D = np.zeros((4,2),dtype=float)
D[0,:] = [0.,0.]
D[1,:] = [0.,1.]
D[2,:] = [1.,0.]
D[3,:] = [1.,1.]
D = torch.tensor(D,dtype=torch.float)
Y_xor = torch.tensor([0.,1.,1.,0.])
N = D.shape[0] # number of input patterns

# Initialize parameters
#     Although you will implement gradient descent manually, let's set requ
#     anyway so PyTorch will track the gradient too, and we can compare you
w_34 = torch.randn(2) # [size 2] tensor representing [w_3,w_4]
w_012 = torch.randn(3) # [size 3] tensor representing [w_0,w_1,w_2]
b_0 = torch.randn(1) # [size 1] tensor
b_1 = torch.randn(1) # [size 1] tensor
w_34.requires_grad=True
w_012.requires_grad=True
b_0.requires_grad=True
b_1.requires_grad=True

alpha = 0.05 # learning rate
nepochs = 8000 # 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 = D[p,:] # input pattern
        print("x", x)
        print(w_34)
        print(w_012)
        # Compute the output of hidden neuron h
        # e.g., two lines like the following
        #  net_h = ...
        #  h = ...
        net_h = torch.dot(x,w_34)+b_0
        h = g_tanh(net_h)
        # TODO : YOUR CODE GOES HERE
        # raise Exception('Replace with your code.')
        # Compute the output of neuron yhat
        # e.g., two lines like the following
        #  net_y = ...
        #  yhat = ...
        xh = torch.torch.cat((x,h),0)
        #print("both", both)
        net_y = torch.dot(xh,w_012)+b_1
        yhat = g_tanh(net_y)
        # TODO : YOUR CODE GOES HERE
        # raise Exception('Replace with your code.')

        # 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,yhat,y)
            print("")

        # Compute the gradient manually
        if verbose:
            print('Compute the gradient manually')
            print_forward(x,yhat,y)
        with torch.no_grad():
            # TODO : YOUR GRADIENT CODE GOES HERE
            #  should include at least these 4 lines (helper lines may be u
            w_34_grad = 2*(yhat-y)*yhat*(1-yhat)*w_012[2]*(h*(1-h)*x)
            b_0_grad = 2*(yhat-y)*yhat*(1-yhat)*w_012[2]*(h*(1-h))
            w_012_grad = 2*(yhat-y)*yhat*(1-yhat)*torch.cat((x,h),0)
            b_1_grad = (2*(yhat-y)*yhat*(1-yhat))
            #  make sure to inclose your code in the "with torch.no_grad()"
            #   otherwise PyTorch will try to track the "gradient" of the g
            # raise Exception('Replace with your code.')
        if verbose:
            print(" Grad for w_34 and b_0")
            print_grad(w_34_grad.numpy(),b_0_grad.numpy())
            print(" Grad for w_012 and b_1")
            print_grad(w_012_grad.numpy(),b_1_grad.numpy())
            print("")

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

        # Parameter update with gradient descent
        with torch.no_grad():
            # TODO : YOUR PARAMETER UPDATE CODE GOES HERE
            # Four lines of the form
            w_34 -= w_34.grad * alpha
            b_0 -= b_0.grad * alpha
            w_012 -= w_012.grad * alpha
            b_1 -= b_1.grad * alpha
            #raise Exception('Replace with your code.')

    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()
```

```
tensor([0.0119, 0.4256], requires_grad=True)
tensor([ 1.0653,  1.4222, -0.2842], requires_grad=True)
Compute the gradient manually
 Input: [1. 1.]
 Output: 0.99
 Target: 0.0
 Grad for w_34 and b_0
  d_loss / d_w = [0.00852944 0.00852944]
  d_loss / d_b = [0.00852944]
 Grad for w_012 and b_1
  d_loss / d_w = [ 0.01886818  0.01886818 -0.01616435]
  d_loss / d_b = [0.01886818]

Compute the gradient using PyTorch .backward()
 Grad for w_34 and b_0
  d_loss / d_w = [-0.00286732 -0.00286732]
  d_loss / d_b = [-0.00286732]
 Grad for w_012 and b_1
  d_loss / d_w = [ 0.03791964  0.03791964 -0.03248571]
  d_loss / d_b = [0.03791964]
```

Problem 5 (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.

The XOR network uses two hidden nodes and one output node. The error remains the same throughout the 8000 epochs. If the input is either {0,0} or {1,1} the XOR should output 0. If the input is {0,1} or {1,0}, the output should be 1. It computes three of the combinations correctly, but tends to mis-compute the tensor {1,1}. It suggests an output of 0.99, when the target output is 0.0.

Homework - Neural networks - Part C (20 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/ (https://brendenlake.github.io/CCM-site/)

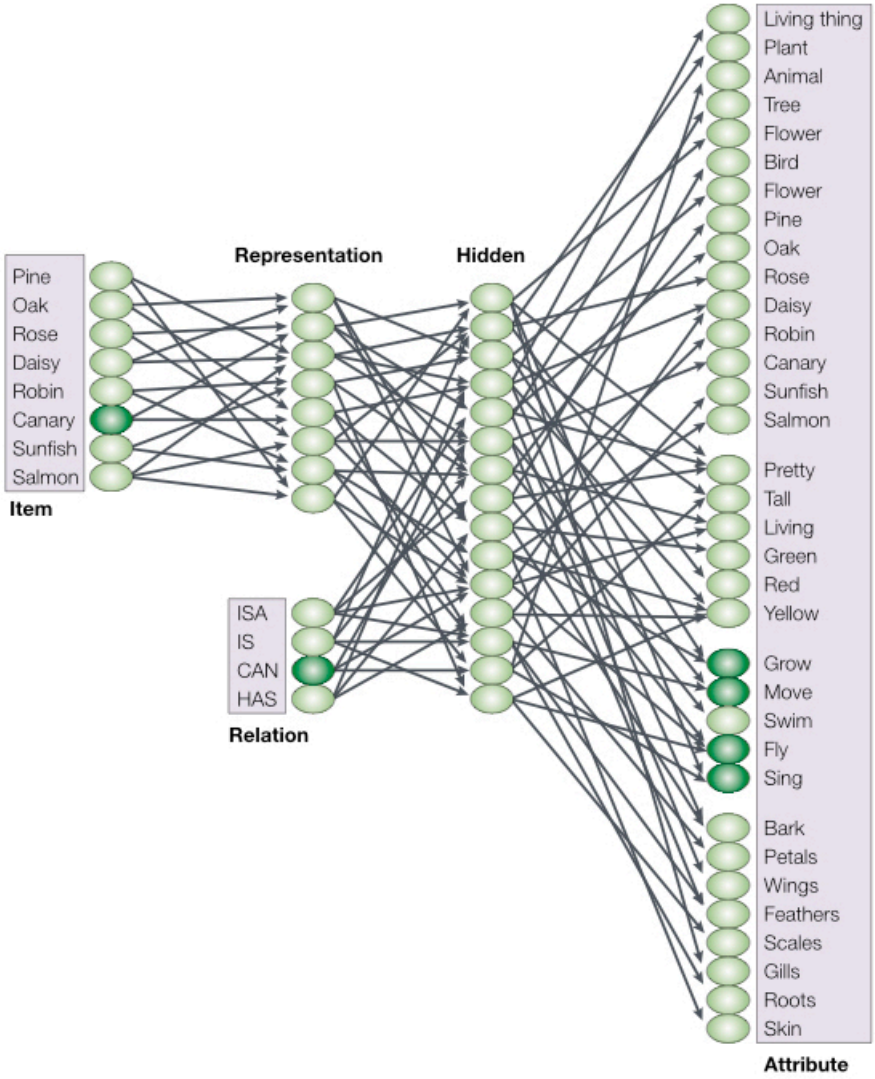
This homework is due before midnight on Monday, Feb. 22, 2021.

Elena Georgieva
elena@nyu.edu
02/22/2021

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 these three basic PyTorch tutorials (https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html), "What is PyTorch?", "Autograd", and "Neural Networks" which should have the basics you need.

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

In [1]:
```
# 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.

In [2]:
```
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.

In [3]:
```
D = np.loadtxt('data/sem_data.txt')
input_pats = D[:,:nobj+nrel]
input_pats = torch.tensor(input_pats,dtype=torch.float)
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]

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

Problem 1 (15 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" (https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#sphx-glr-beginner-blitz-neural-networks-tutorial-py).

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.

In [6]:
```
class Net(nn.Module):
    def __init__(self, rep_size, hidden_size):
        super(Net, self).__init__()
        # Input
        #  rep_size : number of hidden units in "Representation Layer"
        #  hidden_Size : number of hidden units in "Hidden Layer"
        #
        # TODO : YOUR CODE GOES HERE
        item_size = 8
        self.representation = nn.Linear(item_size, rep_size)
        rel_size = 4
        self.hidden = nn.Linear(12,hidden_size)
        self.output = nn.Linear(hidden_size, 36)
        # raise Exception('Replace with your code.')

    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
        #      (containing both object and relation 1-hot identifier) (batc
        #   OR
        #   x : [B x (nobj+nrel) Tensor], which is a batch of B input patte
        #
        # Output
        #   output [B x nattribute Tensor], which is the output pattern for
        #   hidden [B x hidden_size Tensor], which are activations in the H
        #   rep [B x rep_size Tensor], which are the activations in the Rep
        x = x.view(-1,nobj+nrel) # reshape as size [B x (nobj+nrel) Tensor]
        x_item = x[:,:nobj] # input to Item Layer [B x nobj Tensor]
        x_rel = x[:,nobj:] # input to Relation Layer [B x nrel Tensor]
        # TODO : YOUR CODE GOES HERE
        # ----
        x = self.representation(x_item)
        rep = relu(x)
        x = self.hidden(torch.cat((rep,x_rel),1))
        hidden = relu(x)
        x = self.output(hidden)
        output = sigmoid(x)
        #raise Exception('Replace with your code.')
        # -----
        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 [7]:
```
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(erro
    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 [16]:
```
def get_rep(net):
    # Extract the hidden activations on the Representation Layer for each i
    #
    # Input
    #  net : 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):
        input_clean[idx,idx] = 1. # 1-hot encoding of each object (while Re
    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 differen
    #  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 differen
    #  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) #this line causes
    dendrogram(linked1, 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) #this line causes
    dendrogram(linked2, 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) #this line causes
    dendrogram(linked3, color_threshold=0)
    plt.ylim([0,mx])
    plt.show()
```

The next script initializes the neural network and trains it for 4000 epochs total. It trains in three stages, and the item representations (on the Representation Layer) are extracted after 500 epochs, 1500 epochs, and then at the end of training (4000 epochs).

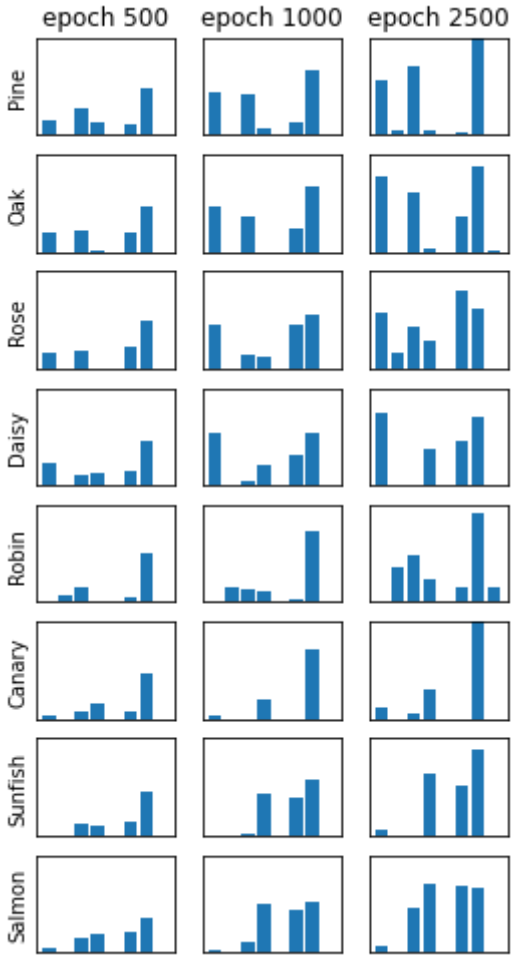
In [17]:
```
learning_rate = 0.1
criterion = nn.MSELoss() # mean squared error loss function
mynet = Net(rep_size=8,hidden_size=15)
optimizer = torch.optim.SGD(mynet.parameters(), lr=learning_rate) # stochas

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-nep
rep2 = get_rep(mynet)
epoch_count = train(mynet,epoch_count,nepochs_additional=nepochs_phase3-nep
rep3 = get_rep(mynet)
```

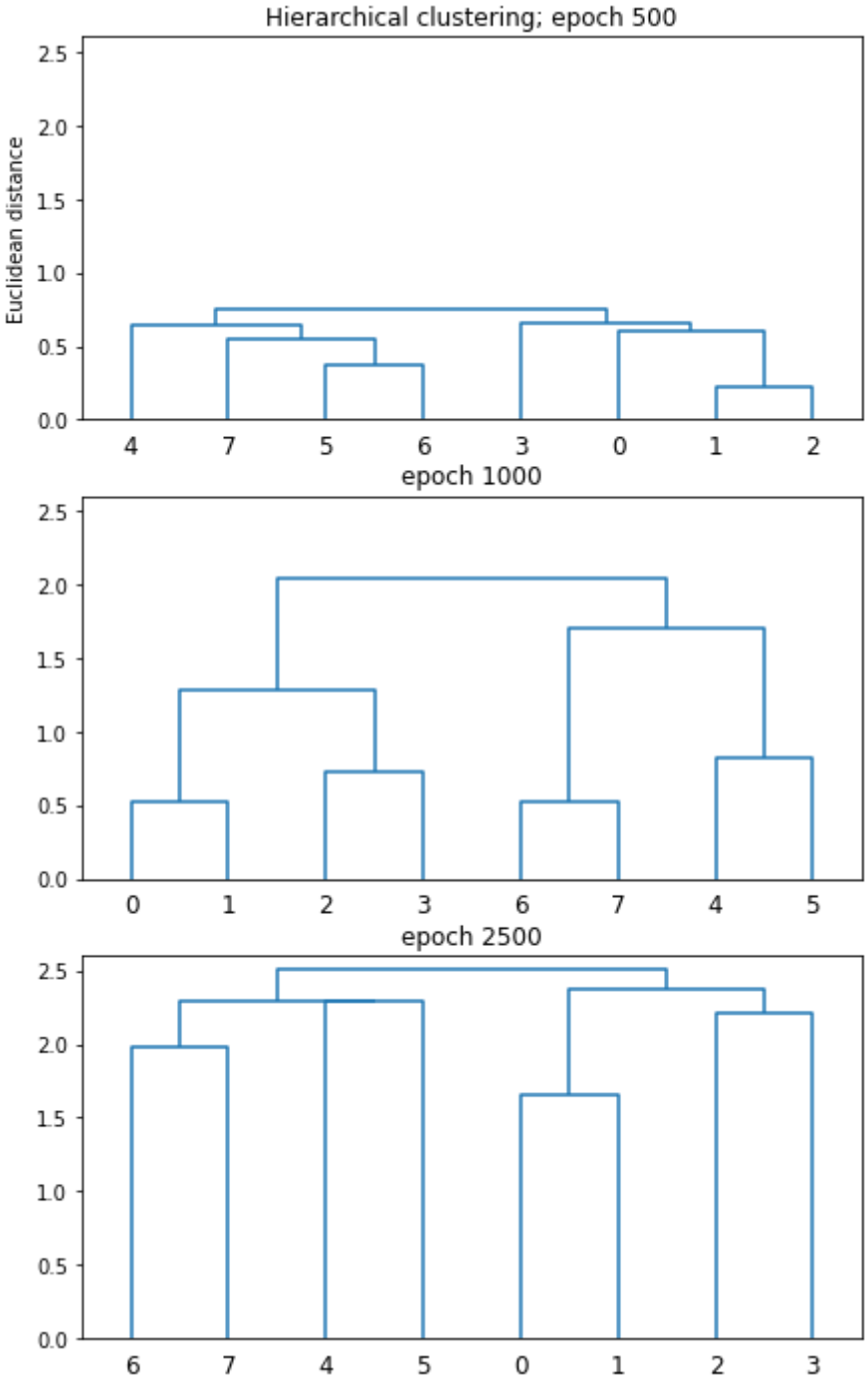
```
epoch 0 loss 0.247
epoch 50 loss 0.069
epoch 100 loss 0.066
epoch 150 loss 0.06
epoch 200 loss 0.056
epoch 250 loss 0.052
epoch 300 loss 0.049
epoch 350 loss 0.047
epoch 400 loss 0.045
epoch 450 loss 0.044
epoch 500 loss 0.042
epoch 550 loss 0.04
epoch 600 loss 0.038
epoch 650 loss 0.033
epoch 700 loss 0.029
epoch 750 loss 0.026
epoch 800 loss 0.024
epoch 850 loss 0.022
epoch 900 loss 0.019
epoch 950 loss 0.017
epoch 1000 loss 0.014
epoch 1050 loss 0.012
epoch 1100 loss 0.011
epoch 1150 loss 0.01
epoch 1200 loss 0.009
epoch 1250 loss 0.008
epoch 1300 loss 0.007
epoch 1350 loss 0.007
epoch 1400 loss 0.006
epoch 1450 loss 0.005
epoch 1500 loss 0.004
epoch 1550 loss 0.004
epoch 1600 loss 0.003
epoch 1650 loss 0.003
epoch 1700 loss 0.002
epoch 1750 loss 0.002
epoch 1800 loss 0.002
epoch 1850 loss 0.002
epoch 1900 loss 0.001
epoch 1950 loss 0.001
```

```
epoch 2000 loss 0.001
epoch 2050 loss 0.001
epoch 2100 loss 0.001
epoch 2150 loss 0.001
epoch 2200 loss 0.001
epoch 2250 loss 0.001
epoch 2300 loss 0.001
epoch 2350 loss 0.001
epoch 2400 loss 0.001
epoch 2450 loss 0.001
```

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

In [18]:
```
plot_rep(rep1,rep2,rep3,names_items)
plot_dendo(rep1,rep2,rep3,names_items)
```





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.

Based on the plots, the internal representation of the network improves over the course of learning. The model starts broad and gradually gets better at specific differentiation. For example, it first separates plants and animals before separating different classes of animals.

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/ (https://brendenlake.github.io/CCM-site/)

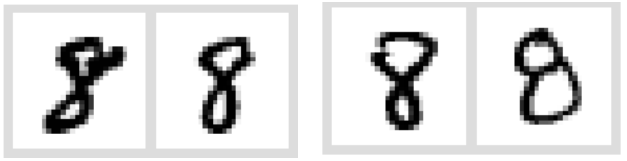
This homework is due before midnight on Monday, Feb. 22, 2021.

```
Elena Georgieva
elena@nyu.edu
02/22/2021
```

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!*

High similarity **Low similarity**



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.

In [1]:
```
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('data',
                                                         train=False,
                                                         download=True,
                                                         transform=trans
                                                         transforms.ToTe
                                                         transforms.Norm
                                                       ),batch_size=1000,
print('MNIST has been loaded.')
```

```
Loading MNIST
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.

In [2]:
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5) # 10 channels in first
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5) # 20 channels in seco
        self.fc1 = nn.Linear(320, 50) # 50 hidden units in first fully-conn
        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 (https://github.com/pytorch/examples/tree/master/mnist) 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.

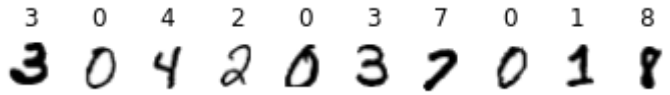
The function `test_viz` picks a few images from the test set at random and displays them along with the network's predicted class labels.

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.

In [4]:
```
# 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-pr
        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
def test_viz(nshow=10):

    # grab a random subset of the data
    testiter = iter(test_loader)
    images, target = testiter.next()
    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
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.

In [5]:
```
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 = testiter.next()
    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

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.

In [6]:
```
images1_digit8,images2_digit8 = get_random_subset(digit_select=8, npairs =
```

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 (https://en.wikipedia.org/wiki/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.

In [7]:
```
# 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 ('conv
    #
    # 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')

    # 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:
[0.80407757 0.70533806 0.80146641 0.84901017 0.86681259 0.89934206
 0.72382498 0.83646357 0.81410724 0.84798265 0.81999856 0.73276353
 0.92761302 0.83448356 0.95749485 0.79342932 0.84225667 0.76638401
 0.75170499 0.8744083 ]
```

Problem 1 (10 points)

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

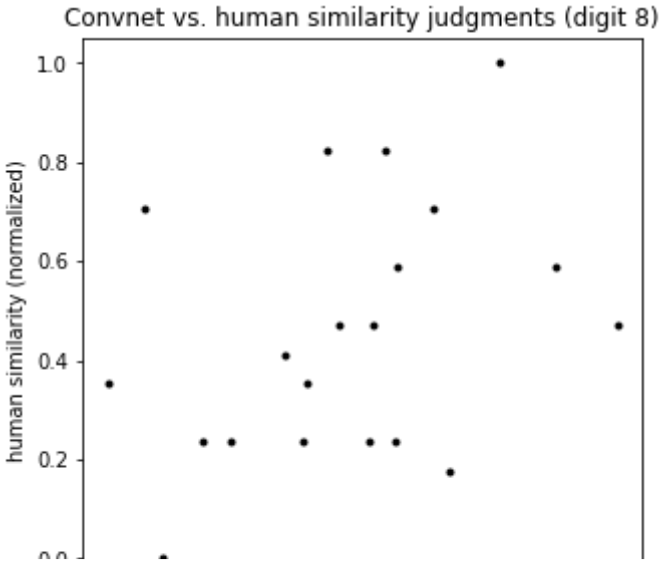
1. *Get judgments from the network.* Run all of the code above to sample a list of 20 pairs of images for that digit. For each pair, get a measure of the network's similarity between the images as measured on the fully-connected `fc` layer. (You may have already done all of this.)

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.

In [9]:
```
# YOUR PARTICIPANT RATINGS GO HERE
ratings_human_1_digit8 = np.array([4,5,2,7,8,10,7,5,8,4,5,2,4,3,5,5,9,4,5,3
ratings_human_2_digit8 = np.array([5,4,5,6,7,10,8,6,9,3,6,1,9,4,6,5,8,3,2,3

# Analysis code
ratings_human_mean_digit8 = (ratings_human_1_digit8 + ratings_human_2_digit

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_n
plt.show()
```

```
Correlation between net and human similarity ratings: r =
0.361
```



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.

The convolutional neural network does a moderately good job at predicting similarity ratings. The network does a good job at capturing shape, but not a good job at capturing handwriting style charactaristics including thickness, legibility, stroke, etc. The convnet ratings and the human ratings were similiar when those values were in the mid-range, for example for the first pair the net output a value of .39, and the humans did .35. For extreme values, the human an convnet varied greatly, for example the convnet scores 0.0 on the second pair, and humans chose .35. It seems, if digits are moderately similar, we reasonably agree. However if they are either very similar or very different, the human and computer evaluations differ.

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

This homework is due before midnight on Monday, Feb. 22, 2021.

Elena Georgieva
elena@nyu.edu
02/22/2021

In this notebook, you will train a recurrent neural network in PyTorch to learn an artificial grammar known as the "Reber grammar" (Reber, 1976). The Reber grammar is a classic learning task in psychology, and it is also a great domain for getting your hands dirty with recurrent neural networks.

The goal of this assignment is to build intuition about the training and operation of recurrent neural networks. We will use a Simple Recurrent Network (SRN) as proposed in Elman (1990). Conceptually, SRNs are similar to the very popular LSTM and GRU networks in contemporary machine learning. We certainly could use more these complex types of recurrent networks instead, but we don't need the additional power here.

This assignment is adapated from the online Parallel Distributed Processing (PDP) Handbook (https://web.stanford.edu/group/pdplab/pdphandbook/handbookch8.html#x24-1060007) by James McClelland, which has excellent background material on SRNs and their application to the Reber grammar task. We strongly suggest that you read this background information now (Section 7.1 of the PDP Handbook).

The original research was reported in Servan-Schreiber, Cleeremans, and McClelland (1991) as an investigation into the properties of recurrent neural networks. The SRN was also used been studied as a model of implicit sequence learning in humans, which was also investigated with the Reber grammar (Cleeremans and McClelland, 19991). We refer you to these references for more details if you are interested, especially in how RNNs can be used as a cognitive model of sequence learning.

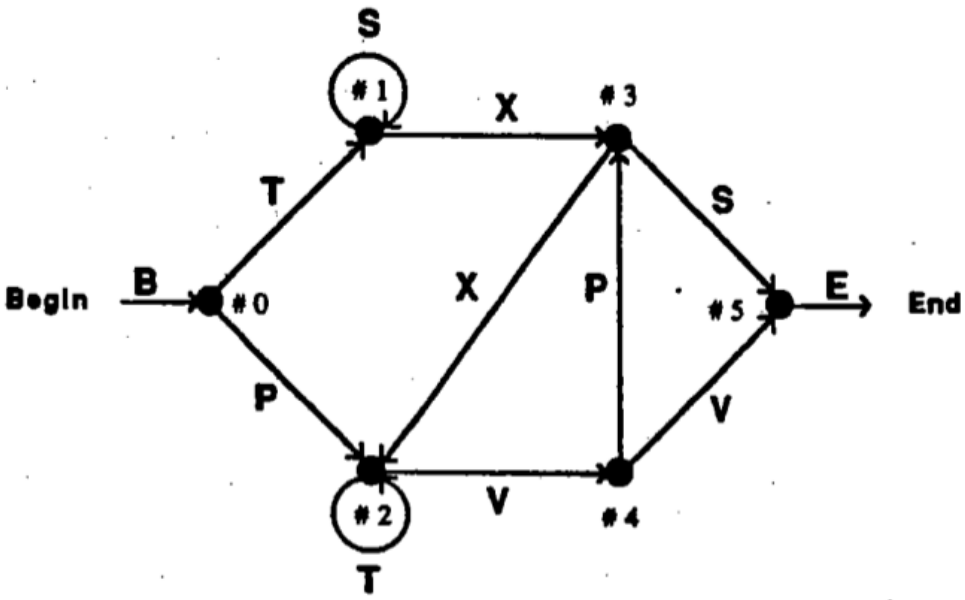
References:

- Cleeremans, A. and McClelland, J. L. (1991). Learning the structure of event sequences. J Exp Psychol Gen, 120:235–253.
- Elman, J. L. (1990). Finding structure in time. Cognitive Science, 14:179–211.
- Servan-Schreiber, D., Cleeremans, A., and McClelland, J. L. (1991). Graded state machines: The representation of temporal contingencies in simple recurrent networks. Machine Learning, 7:161–193.

Additional resources:

Some of this code was adapted from a PyTorch recurrent network tutorial (http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html). Please use this as an additional resource.

Reber grammar



This is the Reber grammar that we will be attempting to master with the SRN. The Reber grammar defines a set of allowable or "grammatical" sequences. The SRN will learn a sequence prediction task: what is the next element/symbol in a sequence, given the past elements. If the SRN predictions are aligned with the grammar, we can say that the SRN has implicitly learned to behave like the grammar, although without explicitly learning rules!

The Reber grammar is diagrammed above as a finite state machine (FSM) with six nodes (states) indicated by #0,#1,...,#5. The FSM creates a string by traversing a path through the graph and visiting various nodes. It can transition between nodes that are connected by a directed edge, and when traversing an edge the FSM emits the output symbol associated with that edge. In the Reber grammar, each string begins with 'B' and ends with 'E'.

Let's go over an example. Starting at 'Begin', let's trace a path through the following nodes, #0, #1, #3, and #5, now ending at 'End.' This would create the string 'BTXSE' (please confirm for yourself). Other paths are possible, and self-connections and loops allow the FSM to create an infinite set of strings. For example, we can retrace the same path as before, but following the self-connection at node #1, to create the path #0, #1, #1, #3, and #5. This creates the string 'BTSXSE.' Likewise, we could keep adding loops, creating 'BTSSXSE', 'BTSSSXSE', ..., 'BTSSSSSSSXSE' and so on. All of these strings are considered "grammatical" according to the Reber grammar. A string that cannot be produced by the grammar is called "ungrammatical."

Here is are some initial exercises to check your understanding of the Reber grammar.

Problem 1 (5 points)

Which of the following strings are grammatical? For each string below, please write 'YES' (for grammatical) or 'NO' (for ungrammatical)

- BTSSXSSE
- BTXXVPSE
- BTXXVPXVPSE
- BTXXXVPXVPSE
- BTXXTVPXVPSE

Write your answers in a new Markdown cell below this one.

BTSSXSSE - NO BTXXVPSE - YES BTXXVPXVPSE - YES BTXXXVPXVPSE - NO BTXXTVPXVPSE - YES

Problem 2 (5 points)

The Reber grammar was carefully designed to display some interesting qualities to make it a difficult learning task. In general, the best a learner can do is predict one of two possible successors for a given node (except at the end of the sequence). For instance, if the grammar is at state #1, both 'S' and 'X' are valid next symbols.

Note that, except for the special beginning and end symbols 'B' and 'E', *each symbol appears in two different places (on different edges) in the grammar.* Therefore, for a learner aiming to master the grammar and make the best possible predictions, she cannot *just* pay attention to the previous symbol. This strategy does not uniquely identify the right set of possible next symbols. Instead, she must track the history further back, in order to make optimal predictions.

In this problem, your job is to simulate a "first-order" memory predictor, meaning you can only remember one symbol back (we also call this the "first-order statistics" of the domain). For each of these symbols, {'B', 'T', 'S', 'X', 'P', 'V'}, what is the set of possible successors given just a first-order memory?

Hint: 'B' has two possible successors. Three of the other symbols have the same exact set of successors.

Write your answers in a new Markdown cell below this one.

For each symbol, possible successors:

- B: T, P
- T: S, X, T, V
- S: S, X, E
- X: T, S, X, V

- P: T, V, S, X
- V: V, P, E

Loading the data

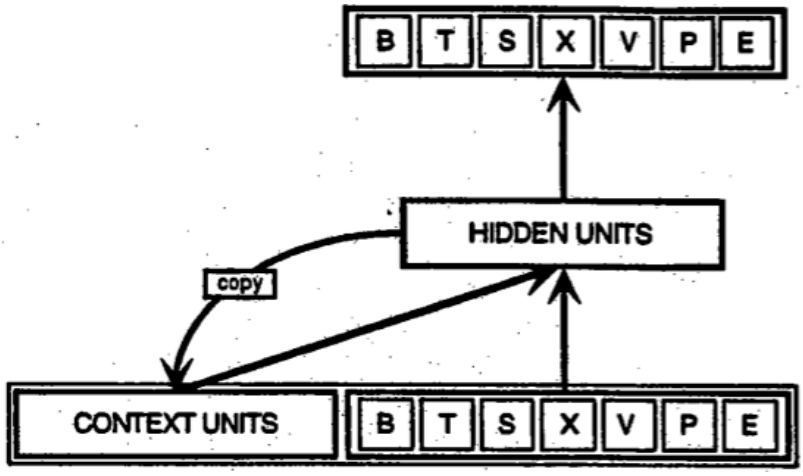
Now let's dive into some code. In our simulations, we are going to limit consideration to all the strings that are grammatical, with a limit on length of 10 symbols or less. There are 43 such strings, divided into a 21 strings training set (data/reber_train.txt) and a 22 strings test set (data/reber_test.txt) (Servan-Schreiber et al., 1991). Follow the links to see the training and test sets.

The following code first loads the training and test sets. We also define a function `seqToTensor` that takes a sequence of symbols and converts them to a 3D PyTorch tensor of size (seq_length x 1 x n_letters). The first dimension iterates over each symbol in the sequence, and each symbol is encoded as a one-hot vector (a 1 x n_letters tensor) indicating which symbol is present.

Execute the code below to see an example tensor for the sequence `BTSXSE` .

In [1]:
```
# Let's start with some packages we need
from __future__ import print_function
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

# Get the index of 'letter' in 'all_letters'
def letterToIndex(letter):
    return all_letters.find(letter)

# Turn a sequence of symbols into a tensor <seq_length x 1 x n_letters>,
# where each symbol is a one-shot vector
def seqToTensor(line):
    tensor = torch.zeros(len(line), 1, n_letters)
    for li, letter in enumerate(line):
        tensor[li][0][letterToIndex(letter)] = 1
    return tensor

# load the training and test data
with open('data/reber_train.txt','r') as fid:
    lines = fid.readlines()
    strings_training = [l.strip() for l in lines]

with open('data/reber_test.txt','r') as fid:
    lines = fid.readlines()
    strings_test = [l.strip() for l in lines]

#
strings_all = strings_training + strings_test
all_letters = ''.join(set(''.join(strings_training))) # string containing a
n_letters = len(all_letters) # total number of possible symbols
training_pats = [seqToTensor(s) for s in strings_training]
ntrain = len(training_pats)
print('all possible letters: %s' % all_letters)
print('first training pattern: %s' % strings_training[0])
print('tensor representation of %s:' % strings_training[0])
print(training_pats[0])
```

```
all possible letters: EVTPBSX
first training pattern: BTSXSE
tensor representation of BTSXSE:
tensor([[[0., 0., 0., 0., 1., 0., 0.]],

        [[0., 0., 1., 0., 0., 0., 0.]],

        [[0., 0., 0., 0., 0., 1., 0.]],

        [[0., 0., 0., 0., 0., 0., 1.]],

        [[0., 0., 0., 0., 0., 1., 0.]],

        [[1., 0., 0., 0., 0., 0., 0.]]])
```



SRN architecture

This is the SRN architecture and its implementation in PyTorch. The goal of the SRN is to predict the next symbol in the sequence, based on previous sequence of symbols it is provided. For instance, it may read 'B', then it may read 'T', and then try to predict the next element which could be either 'S' or 'X', as defined by the Reber grammar.

The SRN has an input, hidden, and output layer. Each unit in the input and output layer corresponds to a different symbols (see Figure). We use a softmax output layer to ensure that the network's prediction is a valid probability distribution over the set of possible symbols which could occur next.

The `SRN` class processes a sequence (and makes predictions) one symbol at a time. Since the hidden layer is recurrent, the hidden activations from the previous symbol are passed to the `forward` method as the tensor `hidden`, along with the next symbol to be processed as the tensor `input` (The previous hidden state is shown as the "Context units" in the above diagram). The output is the predicted probability of the next symbol (in the code below, `output` is the log-probability vector).

In [2]:
```
class SRN(nn.Module):

    def __init__(self, nsymbols, hidden_size):
        #  nsymbols: number of possible input/output symbols
        super(SRN, self).__init__()
        self.hidden_size = hidden_size
        self.i2h = nn.Linear(nsymbols + hidden_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, nsymbols)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        # input: [1 x nsymbol tensor] with one-hot encoding of a single sym
        # hidden: [1 x hidden_size tensor] which is the previous hidden sta
        combined = torch.cat((input, hidden), 1) # tensor size 1 x (nsymbol
        hidden = self.i2h(combined) # 1 x hidden size
        hidden = torch.sigmoid(hidden)
        output = self.h2o(hidden) # 1 x nsymbol
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size, requires_grad=True)
```

Function for training the network

This code provides a `train` function for processing a sequence during training. First, the hidden layer is initialized as zeros `hidden = rnn.initHidden()` and the gradient is cleared with `rnn.zero_grad()`.

In the loop `for i in range(seq_length-1):`, the SRN processes each element in the sequence. After the network makes a prediction at each step, the backpropagation algorithm is used to compute the gradients `loss.backward()` and then the `optimizer` (defined later) updates the weights through `optimizer.step()`.

The final `loss` is the negative log-likelihood of the predicted symbol, averaged across each step of prediction.

In [3]:
```
# Turn a one-hot Variable of <1 x n_letters> into a Tensor of size <1>
def variableToIndex(tensor):
    idx = np.nonzero(tensor.numpy().flatten())[0]
    assert len(idx) == 1
    return torch.tensor(idx, dtype=torch.long)

def train(seq_tensor, rnn):
    # seq_tensor: [seq_length x 1 x nsymbols tensor]
    # rnn : instance of SRN class
    hidden = rnn.initHidden()
    rnn.train()
    rnn.zero_grad()
    loss = 0
    seq_length = seq_tensor.shape[0]
    for i in range(seq_length-1):
        output, hidden = rnn(seq_tensor[i], hidden)
        loss += criterion(output, variableToIndex(seq_tensor[i+1]))
    loss.backward()
    optimizer.step()
    return loss.item() / float(seq_length-1)
```

Functions for evaluating the network

The `eval_viz` function can be used to visualize the sequences of internal states of the network as it processes a string. `eval_viz` is similar to the `train` function above, except that it takes a string `seq` as input and displays the internal state of the network at every step of processing.

Importantly, unlike `train` there are no weight updates. There is no need to study how this function works, it's just to visualize the network state. We'll explain how to use it later.

In [4]:
```
def eval_viz(seq, rnn):
    # seq: string of symbols
    if not all(c in all_letters for c in seq):
        raise Exception('Input sequence contains an invalid letter')
    seq_tensor = seqToTensor(seq)
    rnn.eval()
    hidden = rnn.initHidden()
    seq_length = seq_tensor.shape[0]
    for i in range(seq_length):
        output, hidden = rnn(seq_tensor[i], hidden)
        print('Current input symbol: %s' % seq[i])
        if seq[i] != 'E':
            mydisplay(seq_tensor[i], output, hidden)

def mydisplay(input,output,hidden):
    v_extract = lambda x : x.data.numpy().flatten()
    output_prob = np.exp(v_extract(output))
    print('  Input:', end=' ')
    display_io_v(v_extract(input))
    print('Predicted next symbol:')
    print('  Hidden:', end=' ')
    display_v(v_extract(hidden))
    print('  Output:', end=' ')
    display_io_v(output_prob)
    print(" ")

def display_io_v(v):
    assert len(v)==len(all_letters)
    for idx,c in enumerate(all_letters):
        print('%s:%2.2f' % (c, v[idx]), end=' ')
    print('')

def display_v(v):
    for vi in v:
        print('%2.2f' % (vi), end=' ')
    print('')
```

Similarly, the `eval_set` function takes a list of strings `list_seq` as input and computes the average accuracy of the network's predictions across the list of strings.

How is performance defined? At each step of a string, the SRN's output is considered "correct" only if the highest scoring output symbol is valid according to the grammar. The `eval_set` function returns the average accuracy across the list of strings and elements of each string.

As above, the details of how this function works are not that important. It just defines a reasonable measure of accuracy.

In [5]:
```
def eval_set(list_seq, rnn, strings_possible):
    # list_seq : list of strings to process
    # rnn : instance of SRN class
    # strings_possible : list of all possible strings
    n = len(list_seq)
    acc = np.zeros(n)
    for i in range(n):
        acc[i] = eval_pat(list_seq[i],rnn, strings_possible)
    return np.mean(acc)

def eval_pat(seq, rnn, strings_possible):
    # seq: string of symbols to process
    if not all(c in all_letters for c in seq):
        raise Exception('Input sequence contains an invalid letter')
    seq_tensor = seqToTensor(seq)
    rnn.eval()
    hidden = rnn.initHidden()
    seq_length = seq_tensor.shape[0]
    correct = 0.0
    history = seq[0]
    for i in range(seq_length-1):
        output, hidden = rnn(seq_tensor[i], hidden)
        widx = output.detach().topk(1)[1].item() # index of most likely suc
        myhistory = history + all_letters[widx] # history with next letter
        context = [s[:len(myhistory)] for s in strings_possible] # all poss
        if myhistory in context: # was this a valid continuation?
            correct += 1.
        history += seq[i+1]
    correct = 100. * correct / (seq_length-1)
    return correct
```

Training the simple recurrent network

Finally, we are ready to train the network!

We create a SRN with 8 hidden units and train it for 500 epochs (sweeps through the training data).

Please run the network now!

After the 500 epochs, the accuracy on the training and test set should be pretty good (above 95% correct). These numbers are computed every 20 epochs using our `eval_set` function. The training loss should be around 0.60.

These numbers show that the network has mastered the structure of the grammar, predicting a reasonable successor symbol at each step of the sequence. It, however, can never be a perfect predictor. It is impossible to exactly predict each successor symbol, since there are almost always two possibilities.

In [19]:
```
nepochs = 500 # number of passes through the entire training set
nhidden = 8 # number of hidden units
learning_rate = 0.01

rnn = SRN(n_letters,nhidden) # create the network
optimizer = torch.optim.SGD(rnn.parameters(), lr=learning_rate) # stochasti
criterion = nn.NLLLoss() #log-likelihood loss function

for myiter in range(1,nepochs+1): # for each epoch
    permute = np.random.permutation(ntrain)
    loss = np.zeros(ntrain)
    for p in permute:
        pat = training_pats[p]
        loss[p] = train(pat, rnn)
    if myiter % 20 == 0 or myiter == 1 or myiter == nepochs:
        train_acc = eval_set(strings_training, rnn, strings_all)
        test_acc = eval_set(strings_test, rnn, strings_all)
        print("epoch  %s: train loss %2.2f; train accuracy %2.2f; test accu
```

```
epoch  1: train loss 1.90; train accuracy 38.88; test accuracy 45.77
epoch  20: train loss 1.75; train accuracy 38.88; test accuracy 45.77
epoch  40: train loss 1.61; train accuracy 53.27; test accuracy 59.02
epoch  60: train loss 1.44; train accuracy 66.86; test accuracy 67.73
epoch  80: train loss 1.28; train accuracy 83.21; test accuracy 81.80
epoch  100: train loss 1.13; train accuracy 86.87; test accuracy 83.78
epoch  120: train loss 0.99; train accuracy 86.87; test accuracy 83.78
epoch  140: train loss 0.89; train accuracy 90.41; test accuracy 88.77
epoch  160: train loss 0.80; train accuracy 95.63; test accuracy 93.16
epoch  180: train loss 0.74; train accuracy 95.63; test accuracy 95.83
epoch  200: train loss 0.70; train accuracy 95.63; test accuracy 93.16
epoch  220: train loss 0.68; train accuracy 95.63; test accuracy 95.83
epoch  240: train loss 0.66; train accuracy 95.63; test accuracy 95.83
epoch  260: train loss 0.64; train accuracy 95.63; test accuracy 95.83
epoch  280: train loss 0.63; train accuracy 98.41; test accuracy 97.92
epoch  300: train loss 0.62; train accuracy 96.76; test accuracy 96.91
epoch  320: train loss 0.62; train accuracy 98.41; test accuracy 97.92
epoch  340: train loss 0.61; train accuracy 98.41; test accuracy 97.92
epoch  360: train loss 0.61; train accuracy 98.41; test accuracy 97.92
epoch  380: train loss 0.60; train accuracy 98.41; test accuracy 97.92
epoch  400: train loss 0.60; train accuracy 98.41; test accuracy 97.41
epoch  420: train loss 0.60; train accuracy 98.41; test accuracy 97.92
epoch  440: train loss 0.59; train accuracy 98.41; test accuracy 97.92
epoch  460: train loss 0.59; train accuracy 98.41; test accuracy 97.92
epoch  480: train loss 0.59; train accuracy 99.47; test accuracy 98.42
epoch  500: train loss 0.59; train accuracy 100.00; test accuracy 98.42
```

Visualizing how a sequence is processed

Now let's understand what the network has learned!

Let's process the test string `BPTVVE` with the `eval_viz` function (see code in next cell). The function will walk you through each step of the sequence, and show you the network's internal state and predicted output. For instance, for the first symbol 'B', you will see a display roughly like:

```
Current input symbol: B
  Input : B:1.00 E:0.00 P:0.00 S:0.00 T:0.00 V:0.00 X:0.00
Predicted next symbol:
  Hidden: 0.21 0.13 0.06 0.65 0.93 0.60 0.15 0.95
  Output: B:0.00 E:0.00 P:0.42 S:0.00 T:0.57 V:0.01 X:0.00
```

The 'Input' pattern shows the one-hot encoding of 'B'.
The 'Hidden' pattern shows the activation of the hidden layer.
The 'Output' pattern shows the prediction, in terms of probabilities of each of the successor symbols.

In this case, the network is splitting its bet between 'P' and 'T' when predicting the next symbol, which is exactly right. Note that the network will not always choose exactly 50/50 when predicting successors, since the training patterns are not perfectly balanced in this way.

Problem 3 (5 points)

Execute the code below to consider each step of the test sequence `'BPTVVE'`. Does the network make good predictions at each step? Write two or three sentences about the performance and your reaction.

The model makes probabilistic predictions, so don't just analyze the "most probable" prediction at each step. Look at how the network is distributing its probability across multiple successors.

The network makes good predictions at each step. I notice it predicts two successors each time: It starts with around .57 and .42 likelyhoods of the two, and then starts to get more certain about one prediction as it goes on. In the last iteration, it is most certain and predicts .97 likelyhood of the symbol E.

In [21]:
```
eval_viz('BPTVVE',rnn)

## For Problem 4
#print(strings_test[4])
#eval_viz(str(strings_test[4]),rnn)
```

```
BPVPSE
Current input symbol: B
  Input: E:0.00 V:0.00 T:0.00 P:0.00 B:1.00 S:0.00 X:0.00
Predicted next symbol:
  Hidden: 0.03 0.03 0.77 0.90 0.94 0.31 0.86 0.39
  Output: E:0.00 V:0.01 T:0.57 P:0.42 B:0.00 S:0.00 X:0.00

Current input symbol: P
  Input: E:0.00 V:0.00 T:0.00 P:1.00 B:0.00 S:0.00 X:0.00
Predicted next symbol:
  Hidden: 0.31 0.60 0.08 0.87 0.10 0.16 0.87 0.36
  Output: E:0.00 V:0.43 T:0.54 P:0.00 B:0.00 S:0.01 X:0.01

Current input symbol: V
  Input: E:0.00 V:1.00 T:0.00 P:0.00 B:0.00 S:0.00 X:0.00
Predicted next symbol:
  Hidden: 0.78 0.02 0.99 0.98 0.77 0.40 0.14 0.57
  Output: E:0.01 V:0.24 T:0.01 P:0.74 B:0.00 S:0.00 X:0.00

Current input symbol: P
  Input: E:0.00 V:0.00 T:0.00 P:1.00 B:0.00 S:0.00 X:0.00
Predicted next symbol:
  Hidden: 0.84 0.92 0.07 0.14 0.03 0.07 0.87 0.18
  Output: E:0.00 V:0.01 T:0.00 P:0.00 B:0.00 S:0.56 X:0.43

Current input symbol: S
  Input: E:0.00 V:0.00 T:0.00 P:0.00 B:0.00 S:1.00 X:0.00
Predicted next symbol:
  Hidden: 0.92 0.31 0.86 0.08 0.59 0.86 0.30 0.65
  Output: E:0.97 V:0.00 T:0.00 P:0.01 B:0.00 S:0.01 X:0.01

Current input symbol: E
```

```
## Analyzing the timecourse of learning

Cleeremans and McClelland (1991) ran a psychology experiment that studied
how people implicitly learn the structure of the Reber grammar. They found
that in the initial stages of learning, people's behavior was largely
consistent with the first-order statistics, as computed by the first-order
memory that you worked out in Problem 2. Thus, if the previous symbol was
'V', their behavior would be largely consistent with four possible
successors (rather than just 2, if they had mastered the grammar),
regardless of what had happened previously in the sequence.

In the problem below, you will examine the timecourse of learning in
detail.
```

Problem 4: (15 points)

Retrain the network for only **5 epochs** (set `nepochs = 5` a few cells up). Using the `eval_viz` function and by examining AT LEAST TWO different sequences from the **test set**, how would you describe what the network has learned? Please write your answer in a few sentences, using specific examples. Mention first-order statistics (or higher-order statistics) if appropriate.

Hint: It is helpful to examine a sequence that has the same symbol appearing twice, but where the Reber grammar is visiting different nodes (e.g., 'P' on the way to visiting node #2, and also 'P' on the way to visiting node #3). Does the network make the same prediction in both places? Does looking at the pattern of hidden activations help?

Write your answers in a new Markdown cell below this one.

With nepochs = 5, the test accuracy comes out to around 46%. I tested the model on two sequences from the test set. First, BPTTTVVE. It settled on output weights in the first iteration, and there were almost no changes to the likelyhood of each letter after that. It only predicted two of the eight characters correctly. Second, I tested it on BPVPSE. Again, it maintained almost identical output weights throughout and only predicted one letter correctly. There were some changes to the hidden layer each time, but those did not reflect on the output.

Retrain the network again. This time train for **50 epochs**. Answer the same questions as above.

Write your answers in a new Markdown cell below this one.

With nepochs = 50, the test accuracy came out to 52%. This was a decrease from the 40th iteration, when the test accuracy was 59%. I tested it on the same two sequences. It only predicted two letters of the first sequence correctly. However, it was different from the nepochs = 5 because the output layers varied throughout. The output got more accurate later in the sequence. For the second sequence, it predicted only one of the characters correctly. The second sequence passes the letter 'P' twice, once on the way to 'V' and once on the way to 'S.' The outputs were almost identical, predicting 'V' both times.

Retrain the network again, this time train for the full **500 epochs**. Answer the same questions as above.

You may want to go through this whole process twice, in case there is some variability in your network runs.

This time, it computes five characters correctly. Both of the two that were computed incorrectly falsely predicted 'V.' It seems the model is biased towards 'V.' The model is very successful by the

end, predicting the final E with a value of .98. It is easy to see how the hidden layer contributes to the output.

For the second sequence, it computes three letters successfully -- the last three. This is another example of the model improving over its iteration. The second sequence passes the letter 'P' twice, once on the way to 'V' and once on the way to 'S.' It predicts 'T' the first time (incorreclty) and 'S' the second time (correctly).