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.

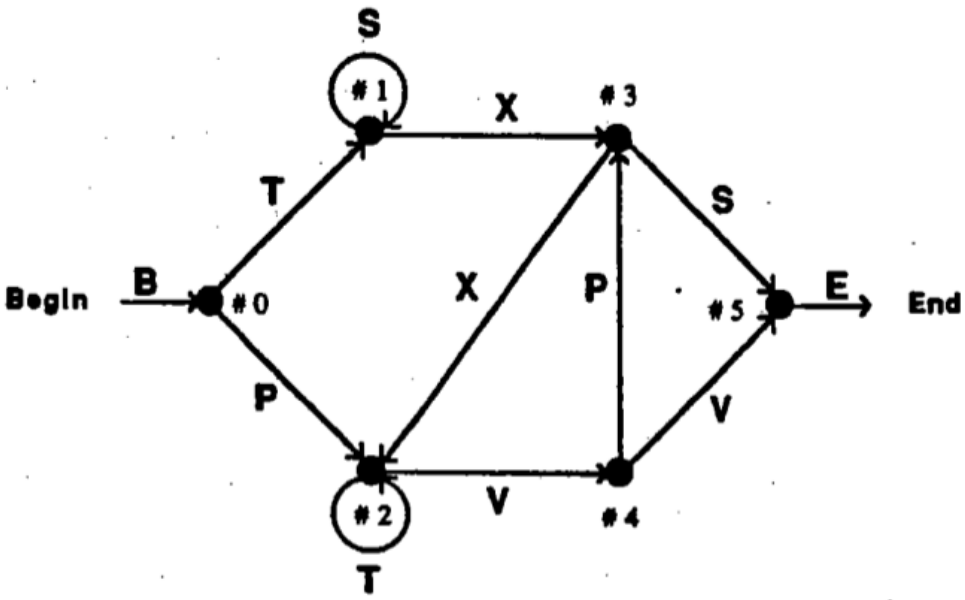
References:

- 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

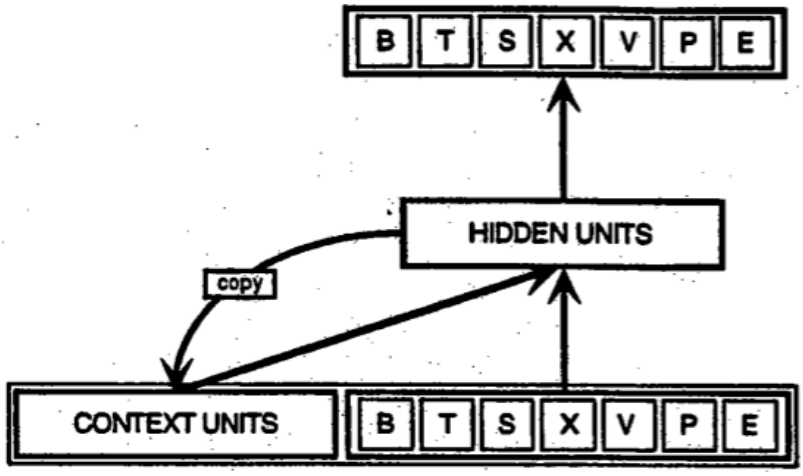
Now let's dive into some code. In our simulations, we are going to limit consideration to all the strings that are grammatical, with a limit on length of 10 symbols or less. There are 43 such strings, divided into a 21 strings training set (data/reber_train.txt) and a 22 strings test set (data/reber_test.txt)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