Midterm Miniproject: Simple Recurrent Networks (SRNs)

Prerequisites

Be sure to follow the instructions for downloading the emergent software from piazza before beginning the midterm.

Introduction

This midterm mini-project has the dual purpose of exposing you to the emergent code -- you will get instructions to build and train a network from scratch -- and exploring the power of simple recurrent networks (SRNs) to learn sequences. We'll see how a network can learn to predict the next item in a sequence depending on its current input (e.g. predicting the next word in a sentence based on the current and previous words). As a reminder, this topic is not currently covered as such in the online text, but is covered in the lecture, so you may want to refer back to the slides on "Temporal learning and representation".

Why do we need a special type of network for this? After all, we have seen that networks can learn to associate specific inputs with specific outputs using error-driven and self-organized learning. So if I'm in the middle of an input sequence that I've seen before, then shouldn't a trained network be able to predict the next item easily? Well, it all depends on how complicated the sequence is. ABCDEF is easy -- there's never any ambiguity about what the next letter should be. ABCBDBEBF is not, on the other hand. It's not clear what item should come after a B, unless you look further back in the sequence for some disambiguating context. That's what SRNs use a **context layer** for.

Before getting started on building and modifying our network, you should use `go build` (run from the terminal in the directory with srn.go) to build the srn.go file. If you then run it, you should see that it consists of two layers and that it doesn't do anything. We're going to change that! To get oriented, let's first take a brief tour of the go code. It will be helpful to keep Emergent's basic hierarchy in mind. Recall that:

- The **network** is composed of **layers**
- **Layers** are groups of **units** (neurons), which are organized together so that **projections** (synaptic connections) can be specified between whole groups which roughly share some function (e.g. representing conjunctions of lines).
- When training the network, we *force* **Input** and **Output** layers to have the activity that we want the network to reproduce. Technically, layers we clamp on are actually called **Target** layers, but those are usually outputs.
- **Hidden** layers are layers that we generally consider inaccessible to the outside world, where "the outside world" could be other brain areas, an experimenter, etc.
- Time in emergent is organized into **cycles**, which make up **trials**. A trial is the time period during which each particular input or input/output pair is given. Groups of trials form **epochs**. These are useful because we may want to test a network's performance on some task after it's seen a

specified number of training examples, such as after one round of seeing all the different input/output pairs.

- **Runs** are groups of epochs. If we train a network and observe it's performance over time, this may or may not be representative of how most networks would behave. Thus we can run a set of epochs, record performance, initialize the network again, and repeat to get a better idea of the general case.

Tour of the project code

Please open the srn.go file in a text editor and search for "Landmark 1". You'll make two basic stops, then look around a bit. After you do that, come back here.

Section 1. Simple sequences without context layer

Next we'll create a basic network with a simple sequential input, and see whether the network can learn this sequence. To do so, find "Section 1a" in the project code, then follow the instructions from there.

Once you've completed the subsections 1a-1d, which set up the network, create input, and turn on a plot for monitoring performance, you should step through some trials (with init, then step trial). Verify that you understand the relationship between the data table you created and which units are on. You can also observe this by clicking the `TrainEnv` button on the side panel, then the `TrainPats` button in the window that opens. This should display the contents of the training file you created. Finally, verify that there is now another tab in the program plotting the training data.

> **Question 1:** Train the network. Does it learn to produce the correct outputs for the given inputs?

> **Question 2:** What do you think would happen during training if you had a sequence like ABCBA? Why?

Section 2. First order dependencies

Now we'll try making longer sequences with first order dependencies (e.g. ABCBD) and see if this standard hidden-layer model can solve these tasks. One thing that's good to do, so that you can compare variations of a model is to make multiple input data tables, rather than repeatedly editing the same one. To do so, create another data file, first-order.dat. Edit it to have an input sequence such as ABCBDE, where at least one input predicts different outputs depending on what came before it. The names of these might be A, BC, C, BD, D, and E, so that you can easily distinguish which "B" trial you're in in the sequence. It is *very important* that the *input* for the trial named BD be the *same* as the input for the trial named BC. This is what will force the network to hold onto context.

Once you have this file, you can use the TrainPats button in the GUI to inspect the input, and the `OpenCSV` button in that window to open your new file.

> **Question 3:** Was it able to learn this sequence? Why or why not? Explain any discrepancy between what you expected and what you observed.

Section 3. Adding a context layer

As you should have observed, our current model has trouble learning the higher order sequence (if it did learn your sequence, you must have done something wrong setting up the Network, or the sequence). As you proceed through this section we will explore how adding a context layer allows the network to discern which of several outputs is required given a single input, based on the context. We will learn that second and third order dependencies are harder for the network to learn but that adjusting the rate at which the network takes activity from the hidden layer can help.

Go back to Section 1a in the code, and create another layer just like the hidden layer, but named "Hidden Context". Write another `net.ConnectLayers(...)` statement that's just like the one connecting the input to the hidden layer, but which goes from the hidden context layer you just created to the hidden layer. Then find the `LayStatNms` list and add "Hidden Context".

Once you've done this, insert the following code at the top of the `AlphaCyc()` function. It's somewhat specific to this project. Can you see what it does just by reading it?

```
context_in := ss.Net.LayerByName("Hidden").(*leabra.Layer)
context_out := ss.Net.LayerByName("Hidden Context").(*leabra.Layer)

context_in.UnitVals(&ss.TmpVals1, "Act")
context_out.UnitVals(&ss.TmpVals2, "Act")
for i,_ := range ss.TmpVals1 {
   ss.TmpVals1[i] = ss.TmpVals1[i] * 1.0 + ss.TmpVals2[i] * 0.0
}
context_out.ApplyExt1D32(ss.TmpVals1)
```

Re-build the network and verify that now there's an additional hidden layer. Then train the network again. You should see activity in the new context layer, and it should be a copy of the hidden layer (of the last trial)! If you look at sending and receiving weights, these should accord with your modifications to the code.

It is a good idea to get a sense of the variability of learning, because some networks will learn faster than others due to randomness in the initial weights. You should allow the first network to run for a few hundred epochs and then also train a series of networks rather than just a single one, by modifying the number `MaxRuns` in the control panel. You can then see how much variability there is by looking at the overlaid plots in the `TrnEpcPlot` tab, the plot in `RunPlot`, or by inspecting the various logs in the control panel. For example, you can see summary information for the different runs by looking at the `RunLog`. For reference, the `RunPlot` was created in the same way as the `TrnEpcPlot` you created earlier.

> **Question 4:** Is the network able to learn the sequence this time? Explain how the context layer allows learning of sequences with higher orders.

Section 4. Second-order dependencies.

The example sequence A B C B D E only has first order dependencies because you would only have to remember the very last item before the repeated item to answer correctly. A sequence with second order dependencies could look like this: A B C D B C E F. To predict E when given C it is not sufficient to keep remember the previous B trial, because that is the same as earlier in the sequence; You need to know

that the BC sequence was preceded by D, two trials back. Likewise, a third order dependency would be something like A B C D E B C D F.

Create a second-order (and third-order if you are brave) sequence. See whether the current network can learn this task (i.e. whether it can get to zero errors, and also how long it takes to do so).

If you are having trouble getting your network to learn a higher order contingency, besides trying to manipulate parameters you should make sure that you didn't just get unlucky with the initial weights. (Same thing if your network seems to have learned higher order contingencies very easily, you might have just gotten lucky!). You should train a sequence of networks as specified above (before Question 4).

> **Question 5:** Is the network able to learn the sequence this time? Why or why not? And if so, in how many epochs?

In order for the network to do better at remembering second and third order dependencies we can change the rate at which the context layer takes information from the hidden layer. Although we said that it "copies" the previous hidden layer activity, this is actually a 'special case' of the SRN, and this function can easily be altered so that the context layer activities reflect part of the previous hidden layer activity but also part of the previous context activity itself. This makes the network more likely to hold onto information from the past.

This is controlled by the parameters named `FmHid` and `FmPrv` The `FmHid` parameter controls the percentage of the context layer activity that will be taken directly from the hidden layer. The `FmPrv` parameter controls the percentage of the context layer activity that will be taken from the previous context activity (on the last trial, but again remember that this last trial will in turn be incorporating activity from the trial before that, and so on).

Go into the srn.go file, and find the `FmHid` and `FmPrv` parameters. They're in the definition of the `Sim` structure, and the `New()` method of `Sim`. Why? Uncomment the parameters and their default settings. Go to the code snippet you inserted earlier that starts with `context_in` and replace the `1.0` and `0.0` with `ss.FmHid` and `ss.FmPrv` respectively. Rebuild the network (using `go build`) and you should see them appear in the control panel.

> **Question 6:** What values of `FmHid` and `FmPrv` allow your network to learn the second order dependency of your previous sequence from Section 2? If it learned already, what values allow it to learn with reduced training time (number of epochs)? Optional: if you tried a third order dependency, what values are needed to more reliably learn this or to optimize the speed at which it learns?

> **Question 7:** Why do these parameters change the network's ability to respond appropriately given different sequential dependencies?

Now we will take a look at what the Hidden layer units are representing. First we'll create a button for the analysis. If you navigate back to the section of the code dealing with the GUI, you'll see a bunch of `tbar.AddAction(...)` statements. As you may have noticed earlier, `tbar` is short for toolbar, so this is where the buttons at the top of the GUI are coming from. Read over the code below, and then copy it into the comment we've left in that section for you. The code for generating the cluster plot is a little complicated, which is why you're not creating it yourself, but you should read it over and give it some thought, as usual.

Note: If you're reading this in the pdf, you may have to open the README.md to actually get the full first line of the code below.

```
   tbar.AddAction(gi.ActOpts{Label: "Reps Analysis", Icon: "fast-fwd",
Tooltip: "Does an All Test All and analyzes the resulting Hidden and
AgentCode activations.", UpdateFunc: func(act *gi.Action) {
     act.SetActiveStateUpdt(!ss.IsRunning)
   }}, win.This(), func(recv, send ki.Ki, sig int64, data interface{}) {
     if !ss.IsRunning {
        ss.IsRunning = true
        tbar.UpdateActions()
        go ss.RepsAnalysis()
     }
   })
```

Re-build the network and train it, making sure that the network is able to get a number of trials correct in a row, then hit the new RepsAnalysis button to generate the cluster plot. The RepsAnalysis button should become available in the control panel. Compare what the cluster plot suggests about the hidden layer's representations with what you see stepping through trials.

Remember: If the input data is still set to zeroth-order.dat in the code, which would be the case unless you changed it, you should use the gui to select the proper higher order sequence input using `TrainPats`.

Remember: As per "VERY IMPORTANT NOTE" #2 in srn.go, if your higher-order sequence is longer than the default .dat file in OpenPats(), change the default .dat in OpenCSV() to your sequence before you re-build. Else, your RepsAnalysis will only consider the first few trials in testing.

> **Question 8:** Which patterns are clustered together and why? How strongly similar are the clustered patterns?

Remember: The total horizontal distance that you have to traverse from two distinct pattern-names to their common branching point in the cluster plot is equal to the similarity of these patterns. If you're not sure about the cluster plot, think about it while actually stepping through and watching the hidden layer!

> **Optional Question 9:** What is the greatest order of dependency that you can still train the network on? This is your chance to show off! What values of `FmHid` and `FmPrv` did you need to use, and why? Feel free to make any other customizations you think may help as well, but be sure to document them here.

Take a moment to consider what SRNs do and how they function.

> **Question 10:** What are some examples of what, behaviorally, they might capture?

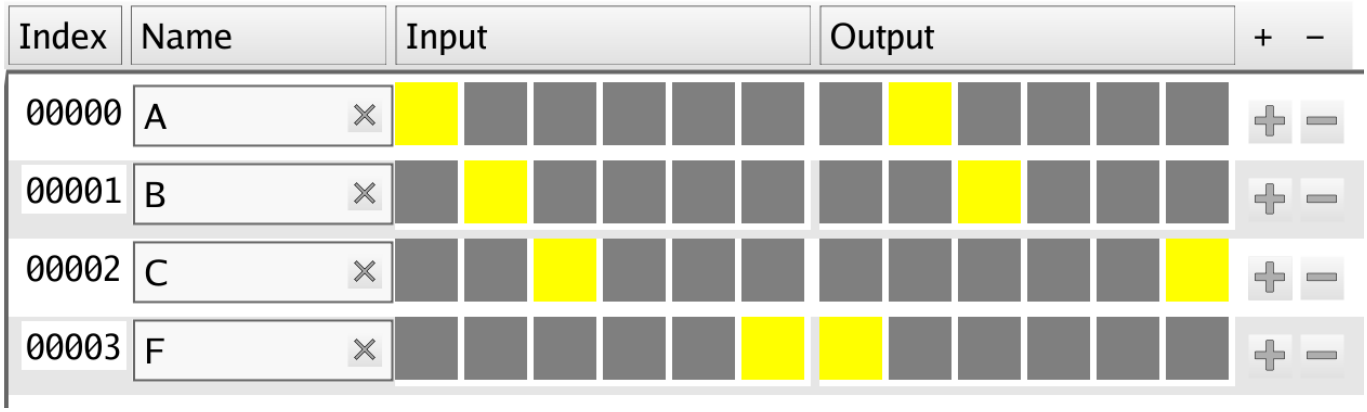
APPENDIX.

Clarification for .dat files

Let's pretend we want the network to learn a different (zeroth order) sequence: "ABCF." We would want to have a .dat file that looks like the below. The highlighted text corresponds to the columns with "Input" as a header. **Make sure that you are using tabs (evident by the long dashes below vs. dots/spaces).**

```
_H: $Name    %Input[2:0,0]<2:1,6>    %Input[2:0,1]
%Input[2:0,2]    %Input[2:0,3]    %Input[2:0,4]
%Input[2:0,5]    %Output[2:0,0]<2:1,6>    %Output[2:0,1]
%Output[2:0,2]    %Output[2:0,3]    %Output[2:0,4]
%Output[2:0,5]
_D: A    1    0    0    0    0    0    0    1    0    0    0    0
_D: B    0    1    0    0    0    0    0    0    1    0    0    0
_D: C    0    0    1    0    0    0    0    0    0    0    0    1
_D: F    0    0    0    0    0    1    1    0    0    0    0    0
```

When we open this file in the simulations we should see:



To verbally interpret what this does, when the network is presented with "A" (i.e. first input is activated), it is learning to respond with "B" (i.e. second output unit is activated). When it is presented "B", it should respond with "C" (I.e. third unit is activated). etc....What is important for the network is that the INPUT AND OUTPUT patterns (I.e. the highlighted units) are correct. The first column in the .dat file is just the "Name" for your own clarity and interpretation.

For higher-order sequences, you are trying to get the network to learn a sequence (e.g. "ABCBDE"). So when it is presented with "A" (i.e. first input is activated), it should learn to respond with "B" (i.e. second output unit is activated). When it is presented "B" following an "A" (i.e. second unit is activated), it should respond with "C" (I.e. third unit is activated). We can name this case a "BC" trial (or alternatively name it "B1") since when "B" is presented in this part of the sequence, "C" should be the output. Labeling the name "BC" or "B1" is just a way for YOU to better understand how the network treats "BC"/"B1" (the B after an A and before a C) differently than "BD"/"B2" (the B after the C and before the D).