Lab 1: Simple Connectionist Models of Language

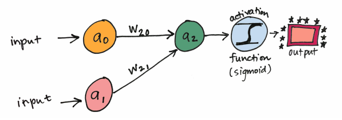
Jennifer Spenader and Jakob Dirk Top

February 15, 2019

Learning Goals for this lab:

- Learn to use the LENS software (the light, efficient network simulator)
- Build simple feed-forward connectionist networks
- Understand the usefulness of hidden layers
- Understand the features of simple recurrent networks (SRNs)
- Build a simple recurrent network in LENS that performs the same as Elman (1990)
- Carry out a follow-up experiment with the Elman (1990) network
- Answer questions related to Elman (1990)
- All lab reports should be handed in individually. However, all parts of question 6 can be done in pairs if desired. Please clearly identify who you worked with in your homework.
- Note: only questions next to capital letters need to be included in the report. Other questions are rhetorical.
- Due date: March 12th. Turn in 6 files (with your own studentnumber). 1) Lab1_s1234567.pdf, 2) OR_s1234567.in, 3) OR_s1234567.ex 4) XOR_s1234567.in, 5) XORSEQ_s1234567.in, 6) ELMAN_s1234567.in

Connectionist networks are fun! Let's learn more!



A simple neural network with two input nodes and one output node.

Practical Information:

Starting (and installing) lens

Short Lens reference

On Nestor, there should be a file called 'lens_additionalreference.pdf', which contains some extra notes on installation and usage of Lens which may be hard to find in the manual.

Lens manual

`http://tedlab.mit.edu/~dr/Lens/`

Creating a simple feed-forward network in LENS

AND in Lens

In the file `AND.in` you can see an example of a simple feed-forward network that learns logical conjunction. Let's look at the main features in the script by opening the file in a text editor. First, to add the network you need to use the command `addNet`:

```
addNet AND
```

This just means that there will be a network with this name, but no features of the network have been defined yet. We want to have a very simple network, so we'll have two input nodes and one output node. Each layer, including input and output layers, are called "Groups" in lens, so we need to use the command:
`addGroup`:

```
addGroup input 2 INPUT
addGroup output 1 OUTPUT SUM_SQUARED
```

Above we have also specified that the measure of error on the output should be calculated using the squared-error function. If you don't write anything, Lens will use cross-entropy by default. This will be a bit less strict. Now we have to define how the groups are connected. We will simply connect the input layer to the output layer using the command `connectGroups`.

Cross-entropy penalizes output units more severely if they are wrong, e.g. near 0 when they should be 1, creating very high error values e.g. approaching infinity. We don't want to be so extreme.).

```
connectGroups input output
```

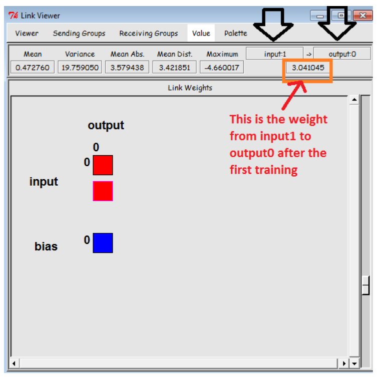
Then we also need a data file to train from. This includes the input, and what is the expected output, called in connectionism the 'target' or 'teacher' pattern:

```
loadExamples AND.ex
```

Here is the truth table for conjunction:

Input1	Input2	Output
1	1	1
1	0	0
0	1	0
0	0	0

Now if you choose `Run script` from the Control Panel and load `AND.in`. You should see that the Batch size is 1, and run each batch 100 times (so 100 epochs of 1 item each). In the Unit viewer you will be able to see the network drawn, but note, it only shows each node, not the connections between them. In the left panel of the Unit viewer you can see the training instances. There are four of course (see margin). Train the network with Batch=1 using the default parameters. Now let's look at the weights. Click on the link viewer. Here you see the link connections. (It's usually very small, you can make it bigger by clicking on Viewer (at the top), then Cell Size, then selecting e.g. 30). If you click on each block, you will see in the right-hand side the weights for that link.



Mean	Variance	Mean Abs.	Mean Dist.	Maximum	input:1	->	output:0
0.472760	19.759050	3.579438	3.421851	-4.660017	3.041045		

This is the weight from input1 to output0 after the first training

1. Simple Networks

A. Draw a picture of the network.

B. Is the bias weight going from the bias node to the output node negative or positive?

C. Are the weights going from the input nodes to the output node negative or positive? Write them down.

D. How much error do you have still? (Look at the graph viewer)

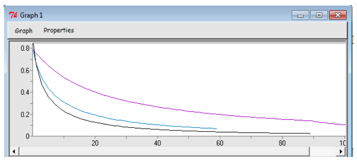
E. Train again. Now reexamine the weights. Have they changed? In what way? What does this mean? Try training again. (This is actually quite fun to do, as the link representations slowly turn color)

If you reset, and make the batch size 0 (write 0 in the control panel, it will turn pink, press enter, changes back to grey), that means the entire training set will be used for each epoch. This should lead to faster learning. Also it will lead to smoother learning (why?). Check for yourself that this is true. In the Lens Console you can also see the error. Do the same training again. Reset the network. Note that the errors change differently, even though the training is the same. What does this mean ? (Hint: how are the weights initialized?)

Batch size: How does batch size influence learning? If you have a batch size of 1, what does this mean? How does learning look with this batch size? If you have batch size = 0 you update the weights after seeing the entire training set. What does learning look like then?

Create inclusive OR and exclusive OR (XOR)

Now that you know how to build a simple network, try it yourself. The function for inclusive OR should be easy to create by modifying the `.in` and `.ex` files from AND. Use the same parameters for training that you used for AND. See if you can answer the following questions (for yourself):



The effect of three different learning rates on learning, 0.05, 0.15 and 0.25.

2. Inclusive OR

A. For inclusive OR, how many epochs of the entire training set do you need to train before the error gets close to zero? Is this more or less than for AND?

B. What is the final state of the network? Which weights are positive and which are negative? What is the weight of the bias?

C. What about exclusive OR (XOR). Can you see that the network cannot learn this function? (Do you recall from the lecture why this function is unlearnable without a hidden layer?)

Learning Rate: If the learning rate is set very high, the changes to the weights will be too extreme. If the learning rate is set very low, learning will take a long time. See for yourself what the effect of different learning rates are on the AND and inclusive OR problems. Set the learning rate to 0.1, 0.2 and 0.3 and train 100 epochs. After each time, reset the network and train again. Lens will create a nice comparative graph showing each learning rate. Different problems will react differently to learning rates. Does learning rate effect AND learning differently than inclusive OR?

Solving XOR: the magic of a hidden layer

Now create a new network to learn the exclusive or problem. But this time, create a hidden layer with four nodes.

You will need to use the `addGroup` command, and you will need to connect it to the other layers using `connectGroups`. For example:

```
addGroup hidden 2
```

It's possible to train XOR with one hidden layer with two nodes, but certain combinations of weight initializations will make the network end up in a local minimum, i.e. it will get stuck. Having four hidden nodes makes this less likely, and also makes training faster.

```
connectGroups input hidden
connectGroups hidden output
```

3. **Exclusive OR**

A. How quickly does the network learn? Does it take longer than for conjunction and inclusive OR? (Note: to make a true comparison you have to use the same parameters with each network, e.g. the same batch size, epochs, and learning rate). Write down what parameters you used in your comparison, as well as the answers to the above questions.

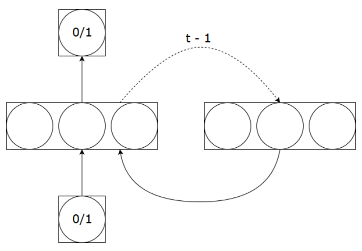
B. What are the connections for the network in the final form? Are they negative or positive? Draw a simple picture of the network.

C. How has the network solved the problem?

Simple Recurrent Network AXBY

In order to figure out how to build the Elman (1990) network, you first need to understand how to build a simple recurrent network in Lens. In this case, we already created a network capable of doing the AXBY task. In the AXBY task, a participant sees a sequence of symbols, which can either be X, A, or B. If he sees an X, he must press button 1. However, if he sees an X preceded by an A, he must do nothing. HOWEVER, if he sees an X, preceded by an A, preceded by a B, he must press button 2.

A neural network capable of performing this task can be found in axby_top.in. It is heavily commented so should be easy to understand. You can ignore the code preceded by two hashtags (##) for now. Note that Lens updates layers in the order they have been created: the context layer has to update BEFORE the hidden layer to make sure it contains the hidden layer's activation from the PREVIOUS timestep. axby16train.ex and axby16test.ex are the train and test sets.



A simple recurrent network in which activations are copied from hidden layer to context layer, on a one to one basis with fixed weight of 1.0. Non-dotted lines represent trainable connections.

4. **AXBY Questions**

A. Look at axby_top.in and draw a picture of the neural network it will create. Remember to include the bias node (by default, the bias node connects to every layer, except to any INPUT layer or ELMAN layer). You only have to draw one line for each connection from one layer to another - you don't have to draw a line for every single weight in the network. If you can't figure out the connections from the code, you can run the script and look at the link viewer.

B. Train the network until the error doesn't decrease anymore (look at the error graph). If the error graph contains a lot of hills, decrease the learning rate (for example to 0.001) and train again. What is (approximately) the lowest error you've reached? What does the Lens console report when you test the network?

C. Now look at the script again, and uncomment the lines preceded by two hashtags (##). What is the difference between the previous neural network and the neural network created by this new script?

D. Which of the two networks would you expect to do better on the AXBY task, and why?

E. Train this new neural network until the error doesn't decrease anymore. What is (approximately) the lowest error you reach this time? What does the Lens console report when you test the network?

F. Is there any difference between the results of both neural networks (especially compare the errors and unit costs reported by the console)? Why? Step through the example sets in the unit viewer (block 2 contains a BAX sequence and several AX sequences). Can the first of these networks solve BAX-sequences? If it can, how? If it can't, why not?

XOR in an SRN

Elman (1990) discusses how you can deal with the the XOR problem using an SRN. This is also a good way to understand how SRN's work. See pages 185-187 in the paper (Elman 1990). First, look at how you can represent the truth table for XOR as sequences:

1 1 0
1 0 1
0 1 1
0 0 0

Now wwe can just string these together, where the first two steps are the input, and each third step is the output:

1 1 0 1 0 1 0 1 1 0 0 0

But what would be better is to randomly put in one of the four different input patterns, e.g.

1 0 1 0 0 0 0 1 1 1 1 0 1 0 1

Interpreted as:

1 0 | 1 | 0 0 | 0 | 0 1 | 1 | 1 1 | 0 | 1 0 | 1

Elman then gave strings like this as input, and then the output to be predicted is the next step in the sequence, e.g. :

input: 1 0 1 0 0 0 0 1 1 1 1 0 1 0 1

output: 0 1 0 0 0 0 1 1 1 1 0 1 0 1 x

Note that we don't know the last item of the output sequence, because there is no next item in the input sequence. To solve this, we generated a slightly longer sequence and removed the last items. "Notice that, given the temporal structure of this sequence, it is only sometimes possible to predict the next item correctly" (Elman, 1990; p 185).

We've prepared a training and test file that you can use for this task. Have a look at both files, and make sure you understand what you are looking at. They are called `elmanxortrain.ex` and `elmanxortest.ex`.

5. An SRN for XOR

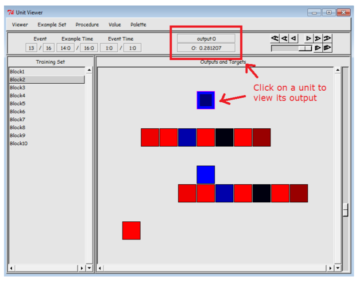
A. Create a script, `xorseq_s1234567.in` (using your own s-number) in a text editor that will solve the XOR problem with an SRN like the one described in the Elman paper. Begin first by drawing on paper what the network looks like. There should be one input node and one output node. What does the rest of the network look like? (Include this and all other network drawings in your report).

B. Run the network. Note that any output > 0.5 can be interpreted as 1, and any output less than 0.5 can be interpreted as 0. This is common practice in neural networks. What is the average error per tick? Why?

C. The error never gets close to zero. Why not?

D. How can you see if the network has learned to predict XOR? Explain where you see how well the network is doing. (Hint read the bottom of page 186-187 in Elman 1990)

E. Compare how your network solves the problem compared to your feedforward solution to XOR. Explain (hint: look at the Elman paper) how the two networks arrive at very different solutions.



Important: Put the parameters you use for training in the .in file so we can see them. Also, include any network drawings in your lab report.

Elman Lexical Classes (1990)

In the following exercise, you will recreate the Elman (1990) neural network, used to predict words in sentences generated from an artificial grammar. In order to understand what you are doing you should carefully read the section in the Elman paper that describes his network and the results.

Here's what you'll need:

- Localist representations of the 31 words used, which then get made into:

- Input-output pairs to feed into the system based on the grammar (following the sentence patterns given in Elman.).

Create a network like Elman's. Read what the architecture of Elman's network was on pages 195-197. Name the file elman_s1234567.in.

6. Lexical Class Learning

We've prepared training materials based on Elman (1990) (page 196). It's called `elmanwords.ex`. Have a look at it.

Due to the size of the training set and the number of nodes and weights in the network that Elman specifies, training may take a while, depending on your network and the computer you're using. You may need up to 40 seconds or more for each pass.

A. Create the network according to Elman's specifications. Explain in your report how you did this, and make sure you comment everything well in the .in file, also including all relevant parameters.

B. Elman states: "Recall that the prediction task is non-deterministic". Explain why this is so, and also why using the network to solve a non-deterministic problem is more interesting than using it to solve a deterministic one.

Comparing Networks: To really compare one run to another, you need to use the same parameters and use the same error function to calculate error. .

C. Train the network. Use the SUM_SQUARED error function, to calculate error. Explain in your report what the results are. How do you get the best results?

D. For other networks, we looked at how long it took for error to decrease and how low we could get the error. Are these useful evaluation parameters for this problem? Why or why not? Explain your answer.

If your network selects the same output on each input, it may be in a local minimum or it may be over- or under-trained.

Extending Elman's Lexical Classes

Ideally what we would now do is include several homonyms, e.g. let the same localist representation be used as both a noun and a verbs.

7. Extending Elman

A. How can you modify the input/training files to let there be several homonyms? Explain what you would do.

B. We actually created such a modified example file, elmanhomonyms.ex, by allowing on verb to also be a noun and one noun to also be a verb. Train with this file and compare the results to the previous model. How do homonyms affect training and results?

C. Read Elman carefully. How did Elman evaluate the network to get the diagram he shows on page ? that showed that the network was able to learn the differences between nouns and verbs? Explain in your own words (not Elman's) making reference to the output you get from Lens how you could do a similar evaluation, pointing out the data and interpretation steps you would have to make.

Discussion Questions

8. **Literature Questions**

A. In the first lecture 5 advantages for computational modelling were presented (1. Explicitness,2. Study complex predictions or interactions,3. Inspiration,4. Practicality and 5. Control and Explanation). Explain how the SRN model of nouns and verbs relates to each one of the advantages. Your answers should not be longer than 400 Words.

B. Elman (1990) at the end of the paper states " Some problems change their nature when expressed as temporal events", and in the paper the main example of this is the modelling of XOR in a simple feed forward network compared to an SRN. In the lab we also modelled word sequences and saw that parts-of-speech like nouns and verbs could *emerge* from modelling sequences. But this problem was presented as a serial one. Can you speculate on how noun and verb detection could be learned with using a simple feedforward network? What would the input look like and do you think it could solve the problem? What would that then mean?