Assignment

Part 1

1. [1 mark] Implement a model `NetLin` which computes a linear function of the pixels in the image, followed by log softmax. Run the code by typing: Copy the final accuracy and confusion matrix into your report. The final accuracy should be around 70%. Note that the **rows** of the confusion matrix indicate the target character, while the **columns** indicate the one chosen by the network. (0="o", 1="ki", 2="su", 3="tsu", 4="na", 5="ha", 6="ma", 7="ya", 8="re", 9="wo"). More examples of each character can be found here.

   ```
   python3 kuzu_main.py --net lin
   ```

   Final matrix:

   ```
   [[764.   5.   8.  15.  30.  62.   2.  63.  33.  18.]
    [  6. 672. 106.  18.  29.  25.  58.  11.  25.  50.]
    [  7.  61. 692.  27.  25.  21.  46.  37.  46.  38.]
    [  5.  36.  59. 759.  15.  56.  14.  18.  27.  11.]
    [ 61.  53.  81.  19. 619.  18.  33.  36.  21.  59.]
    [  8.  28. 126.  17.  19. 725.  27.   9.  33.   8.]
    [  4.  21. 148.  10.  25.  26. 719.  21.  12.  14.]
    [ 16.  29.  28.  12.  80.  17.  55. 623.  91.  49.]
    [ 12.  37.  97.  41.   5.  32.  44.   6. 705.  21.]
    [  8.  52.  89.   3.  52.  31.  21.  29.  39. 676.]]
   Test set: Average loss: 1.0102, Accuracy: 6954/10000 (70%)
   ```

2. [1 mark] Implement a fully connected 2-layer network `NetFull` (i.e. one hidden layer, plus the output layer), using tanh at the hidden nodes and log softmax at the output node. Run the code by typing: Try different values (multiples of 10) for the number of hidden nodes and try to determine a value that achieves high accuracy (at least 84%) on the test set. Copy the final accuracy and confusion matrix into your report, and include a calculation of the total number of independent parameters in the network.

```
python3 kuzu_main.py --net full
```

Final output:

```
<class 'numpy.ndarray'>
[[860.   2.   2.   5.  27.  26.   5.  37.  32.   4.]
 [  4. 815.  36.   4.  21.  10.  61.   4.  17.  28.]
 [  8.  20. 842.  36.  11.  17.  21.  14.  19.  12.]
 [  2.  10.  29. 919.   3.  15.   8.   3.   6.   5.]
 [ 41.  23.  21.   7. 812.  12.  33.  14.  20.  17.]
 [ 10.  15.  79.   8.  11. 832.  24.   3.  16.   2.]
 [  3.  13.  56.   7.  17.   8. 877.  11.   3.   5.]
 [ 17.   6.  16.   4.  34.  12.  35. 817.  23.  36.]
 [  8.  22.  24.  49.   7.  12.  28.   2. 844.   4.]
 [  5.  14.  53.   3.  25.  11.  30.  16.  10. 833.]]

Test set: Average loss: 0.5121, Accuracy: 8451/10000 (85%)
```

In this model, there are 28 inputs, 120 units in its hidden layer, and 10 outputs.

The number of weights Nw of an MLP is:

- Nw = (Ni+1) * Nh + (Nh + 1) * No = (28 + 1) * 120 + (120 + 1) * 10 = 95410

3. [2 marks] Implement a convolutional network called `NetConv`, with two convolutional layers plus one fully connected layer, all using the relu activation function, followed by the output layer, using log softmax.
   You are free to choose for yourself the number and size of the filters, meta parameter values (learning rate and momentum), and whether to use max pooling or a fully convolutional architecture. Run the code by typing:
   Your network should consistently achieve at least 93% accuracy on the test set after 10 training epochs.
   Copy the final accuracy and confusion matrix into your report, and include a calculation of the total number of independent parameters in the network.

   ```
   python3 kuzu_main.py --net conv
   ```

the final accuracy and confusion matrix:

```
[[948.   3.   1.   1.  25.   2.   3.   6.   5.   6.]
 [  1. 905.   6.   0.  14.   0.  39.   7.  10.  18.]
 [  8.  13. 898.  13.   9.   9.  18.   8.  13.  11.]
 [  3.   2.  18. 945.   4.  11.   6.   2.   4.   5.]
 [ 13.   9.   3.   4. 936.   3.   9.   5.  13.   5.]
 [  4.  12.  44.   4.   7. 877.  28.   7.   7.  10.]
 [  4.   2.  11.   1.   7.   3. 966.   3.   1.   2.]
 [  6.   4.   1.   0.   0.   1.  11. 955.   6.  16.]
 [  4.   9.  13.   3.   3.   3.   6.   0. 958.   1.]
```

```
  [  6.   4.  11.   1.  11.   0.   8.   8.  13. 938.]]

 Test set: Average loss: 0.3208, Accuracy: 9326/10000 (93%)
```

In this model:

For the conv1 layer:

- The number of in_channels = 1, out_channels = 8, kernel size = 5
  - 1 * 8 * 5 * 5 + 8 = 208

For the conv2 layer:

- The number of in_channels = 8, out_channels = 18, kernel size = 5
  - 8 * 18 * 5 * 5 + 18 = 3618

Fully connected layer 1:

- The number of in_channels = 3528, out_channels = 200
  - 3528 * 200 + 200 = 705,800

Fully connected layer 2:

- The number of in_channels = 200, out_channels = 10
  - 200 * 10 + 10 = 2010

Therefore, the sum of independent parameters

- 208 + 3618 + 705800 + 2010 = 711636

4. [4 marks] Briefly discuss the following points:

- the relative accuracy of the three models

According to the results prior, we can find that the linear network is not suitable for grayscale images. When we start to use the fully-connected network, the accuracy of the model is relatively high, such as the two-layer fully-connected network. The accuracy is greater than 15%.

Turn to the fully-connected network of convolutional layers. We use two convolutional layers to extract features from images. The accuracy is higher than linear models, up to 93%.

- the number of independent parameters in each of the three models
  - The number of independent parameters for the first model is 7850.

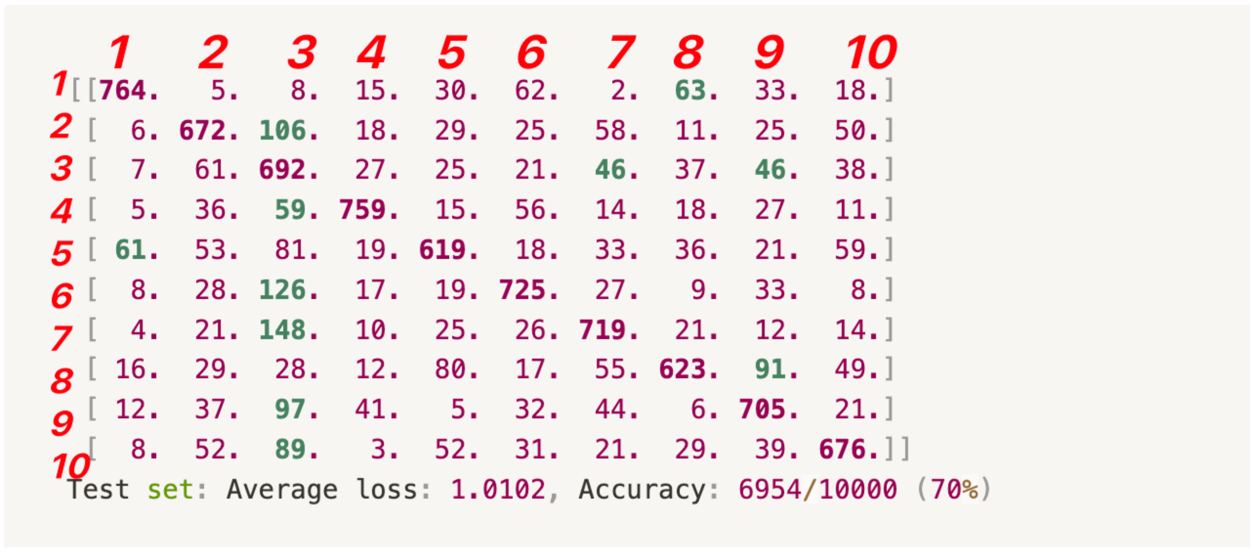
- The number of independent parameters for the second model is 95410.
- The number of independent parameters for the third model is 711636.

The confusion matrix for each model: which characters are most likely to be mistaken for which other characters, and why?

- According to the confusion matrix for the 1-layer NetLin, 2-layer NetLin, and NetConv, we can get the confusion table below:

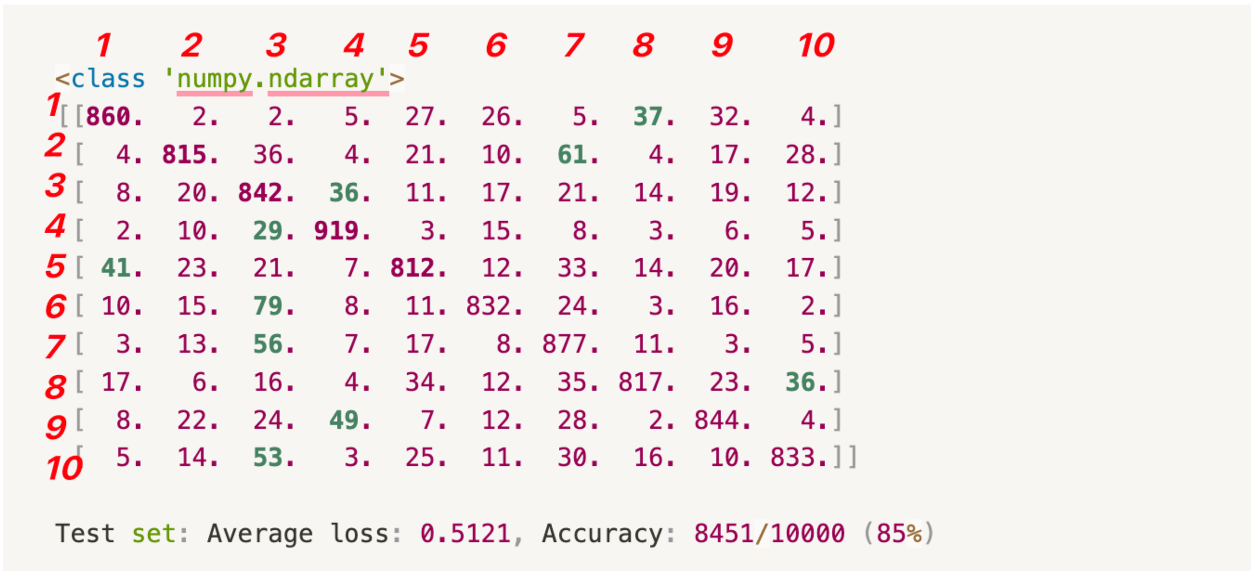
NetLinear	NetFull	NetConv
1-8	1-8	1-5
2-3	2-7	2-7
3-7, 3-9	3-4	3-7
4-3	4-3	4-3
5-1	5-1	5-1
6-3	6-3	6-3
7-3	7-3	7-3
8-8	8-10	8-10
9-3	9-4	9-3
10-3	10-3	10-9

NetLin:



```
[[764.   5.   8.  15.  30.  62.   2.  63.  33.  18.]
 [  6. 672. 106.  18.  29.  25.  58.  11.  25.  50.]
 [  7.  61. 692.  27.  25.  21.  46.  37.  46.  38.]
 [  5.  36.  59. 759.  15.  56.  14.  18.  27.  11.]
 [ 61.  53.  81.  19. 619.  18.  33.  36.  21.  59.]
 [  8.  28. 126.  17.  19. 725.  27.   9.  33.   8.]
 [  4.  21. 148.  10.  25.  26. 719.  21.  12.  14.]
 [ 16.  29.  28.  12.  80.  17.  55. 623.  91.  49.]
 [ 12.  37.  97.  41.   5.  32.  44.   6. 705.  21.]
 [  8.  52.  89.   3.  52.  31.  21.  29.  39. 676.]]
Test set: Average loss: 1.0102, Accuracy: 6954/10000 (70%)
```

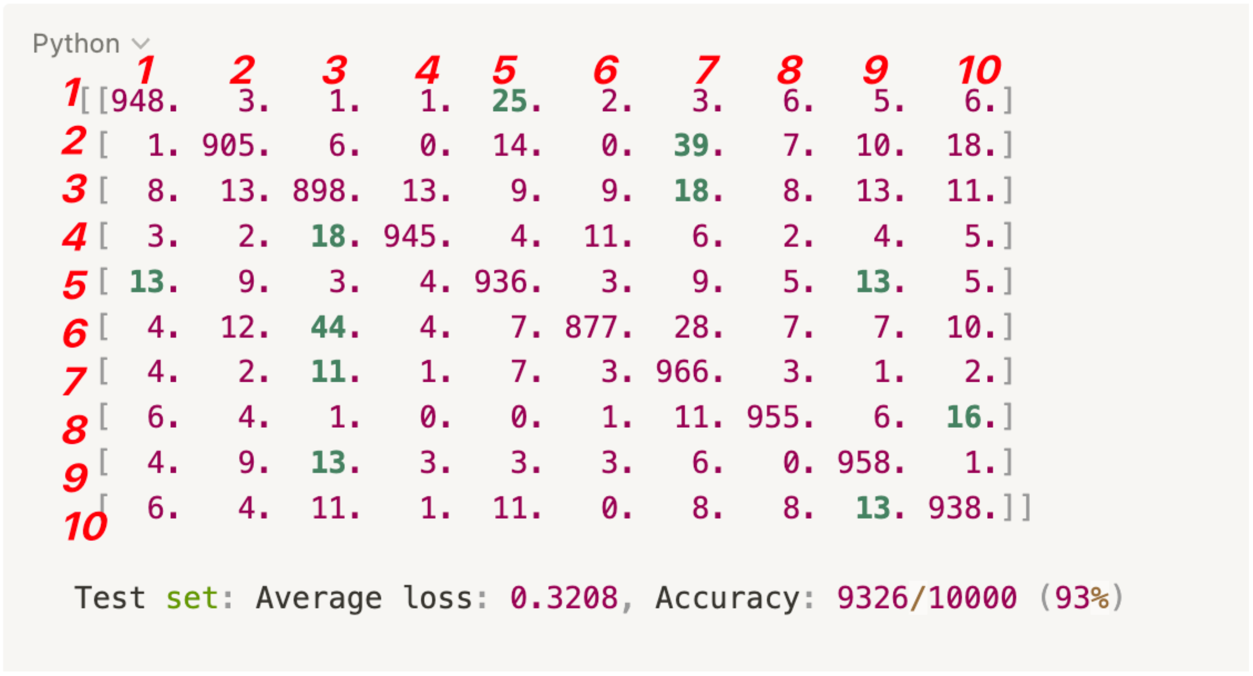
NetFull:



```
    1     2     3    4    5    6    7    8    9     10
 <class 'numpy.ndarray'>
1 [[860.   2.   2.   5.  27.  26.   5.  37.  32.   4.]
2  [  4. 815.  36.   4.  21.  10.  61.   4.  17.  28.]
3  [  8.  20. 842.  36.  11.  17.  21.  14.  19.  12.]
4  [  2.  10.  29. 919.   3.  15.   8.   3.   6.   5.]
5  [ 41.  23.  21.   7. 812.  12.  33.  14.  20.  17.]
6  [ 10.  15.  79.   8.  11. 832.  24.   3.  16.   2.]
7  [  3.  13.  56.   7.  17.   8. 877.  11.   3.   5.]
8  [ 17.   6.  16.   4.  34.  12.  35. 817.  23.  36.]
9  [  8.  22.  24.  49.   7.  12.  28.   2. 844.   4.]
10 [  5.  14.  53.   3.  25.  11.  30.  16.  10. 833.]]

 Test set: Average loss: 0.5121, Accuracy: 8451/10000 (85%)
```

NetConv:



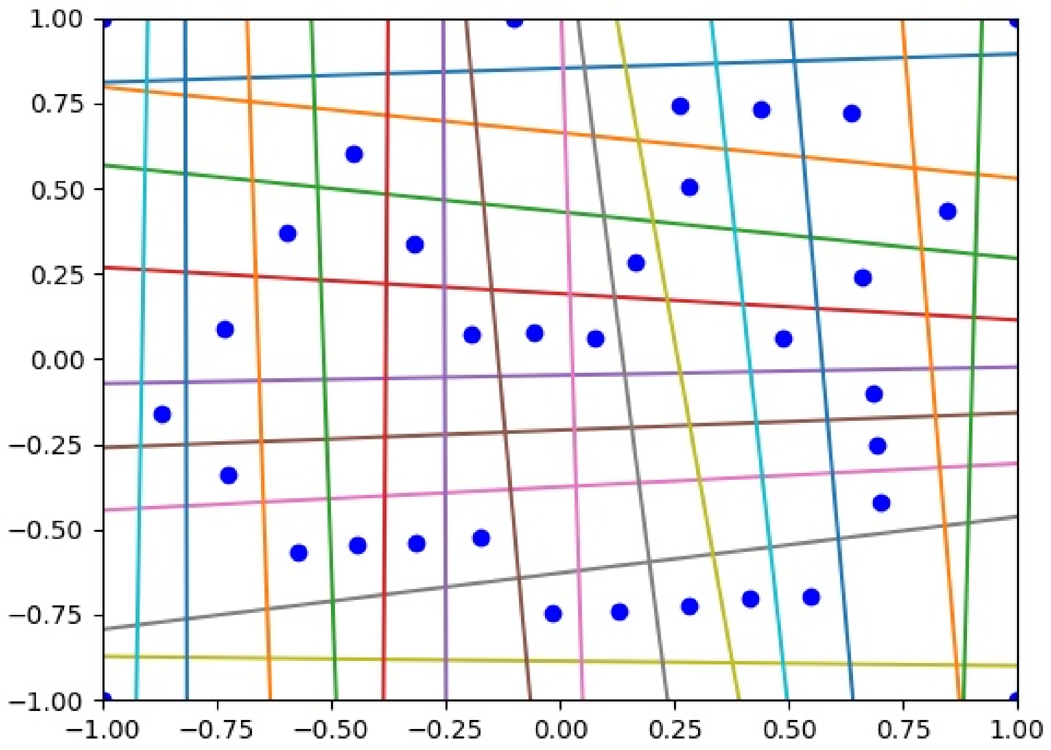
```
Python
     1     2     3     4     5     6     7     8     9     10
1 [[948.   3.    1.    1.   25.    2.    3.    6.    5.    6.]
2  [  1. 905.    6.    0.   14.    0.   39.    7.   10.   18.]
3  [  8.  13.  898.   13.    9.    9.   18.    8.   13.   11.]
4  [  3.   2.   18.  945.    4.   11.    6.    2.    4.    5.]
5  [ 13.   9.    3.    4.  936.    3.    9.    5.   13.    5.]
6  [  4.  12.   44.    4.    7.  877.   28.    7.    7.   10.]
7  [  4.   2.   11.    1.    7.    3.  966.    3.    1.    2.]
8  [  6.   4.    1.    0.    0.    1.   11.  955.    6.   16.]
9  [  4.   9.   13.    3.    3.    3.    6.    0.  958.    1.]
10 [  6.   4.   11.    1.   11.    0.    8.    8.   13.  938.]]

 Test set: Average loss: 0.3208, Accuracy: 9326/10000 (93%)
```

From the table, we can know they most do the same mistake.

Part 2

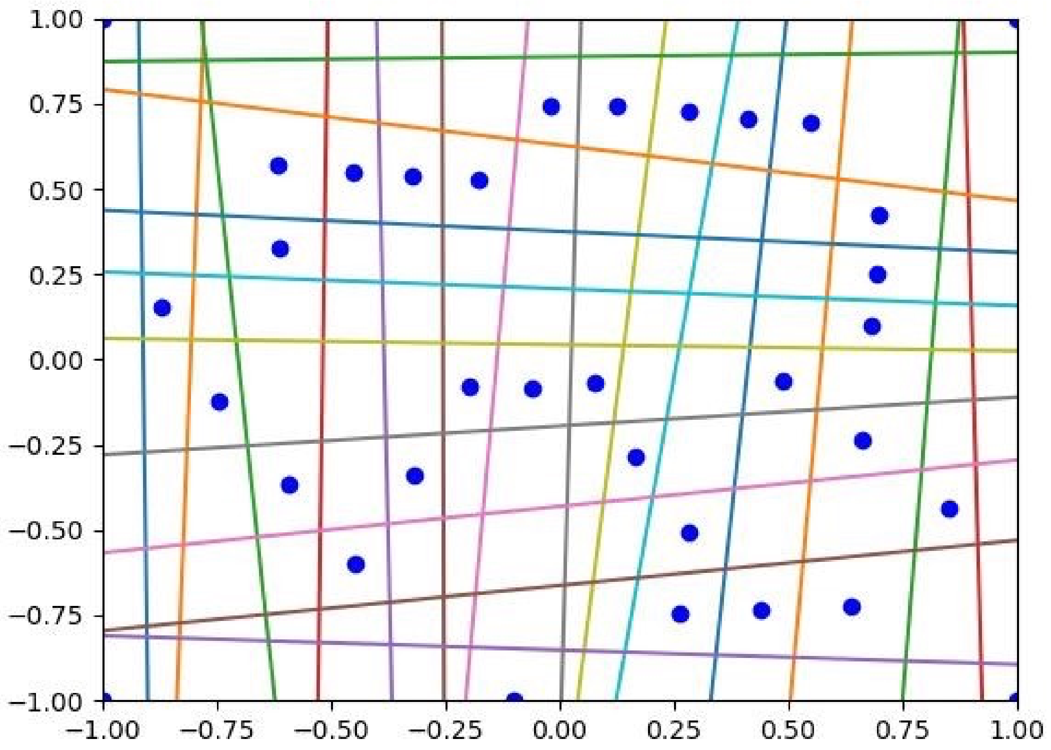
You will be editing the file `encoder.py` to create a dataset which, when run in combination with `encoder_main.py`, produces the following image (which is intended to be a stylized map of mainland China).



1. [2 marks] Create by hand a dataset in the form of a tensor called `ch34` in the file `encoder.py` which, when run with the following command, will produce an image essentially the same as the one shown above (but possibly rotated or reflected).The pattern of dots and lines must be topologically identical. But, it is fine for it to be rotated or reflected, compared to the image above. Note in particular the five "anchor points" in the corners and on the edge of the figure.

```
python3 encoder_main.py --target ch34
```

Your tensor should have 34 rows and 23 columns. Include the final image in your report, and include the tensor `ch34` in your file `encoder.py`



Part 3

1. [2 marks] Train a Simple Recurrent Network (SRN) on the Reber Grammar prediction task by typing

```
python3 seq_train.py --lang reber
```

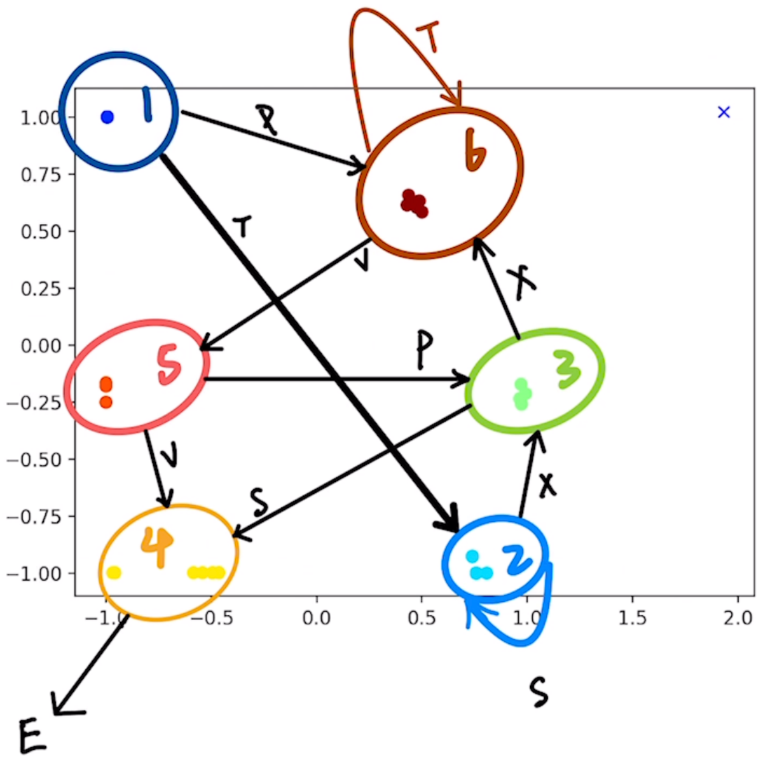
This SRN has 7 inputs, 2 hidden units and 7 outputs. The trained networks are stored every 10000 epochs, in the net subdirectory. After the training finishes, plot the hidden unit activations at epoch 50000 by typing

```
python3 seq_plot.py --lang reber --epoch 50
```

The dots should be arranged in discernable clusters by colour. If they are not, run the code again until the training is successful. The hidden unit activations are printed according to their "state", using the colourmap "jet":

Based on this colour map, annotate your figure (either electronically, or with a pen on a printout) by drawing a circle around the cluster of points corresponding to each state in the state machine, and drawing arrows between the states, with each arrow labelled with its corresponding symbol. Include the annotated figure in your report.

Answer:



2. [1 mark] Train an SRN on the $a^n b^n$ language prediction task by typing

```
python3 seq_train.py --lang anbn
```

The *anbn* language is a concatenation of a random number of A's followed by an equal number of B's. The SRN has 2 inputs, 2 hidden units and 2 outputs.

Look at the predicted probabilities of A and B as the training progresses. The first B in each sequence and all A's after the first A are not deterministic and can only be predicted in a probabilistic sense. But, if the training is successful, all other symbols should be correctly predicted. In particular, the network should predict the last B in each sequence as well as the subsequent A. The error should be consistently below 0.01. If the network appears to have learned the task successfully, you can stop it at any time using ⟨cntrl⟩-c. If it appears to be stuck in a local minimum, you can stop it and run the code again until it is successful.

After the training finishes, plot the hidden unit activations by typing

```
python3 seq_plot.py --lang anbn --epoch 100
```

Include the resulting figure in your report. The states are again printed according to the colourmap "jet". Note, however, that these "states" are not unique but are instead used to count either the number of A's we have seen or the number of B's we are still expecting to see.

Answer:

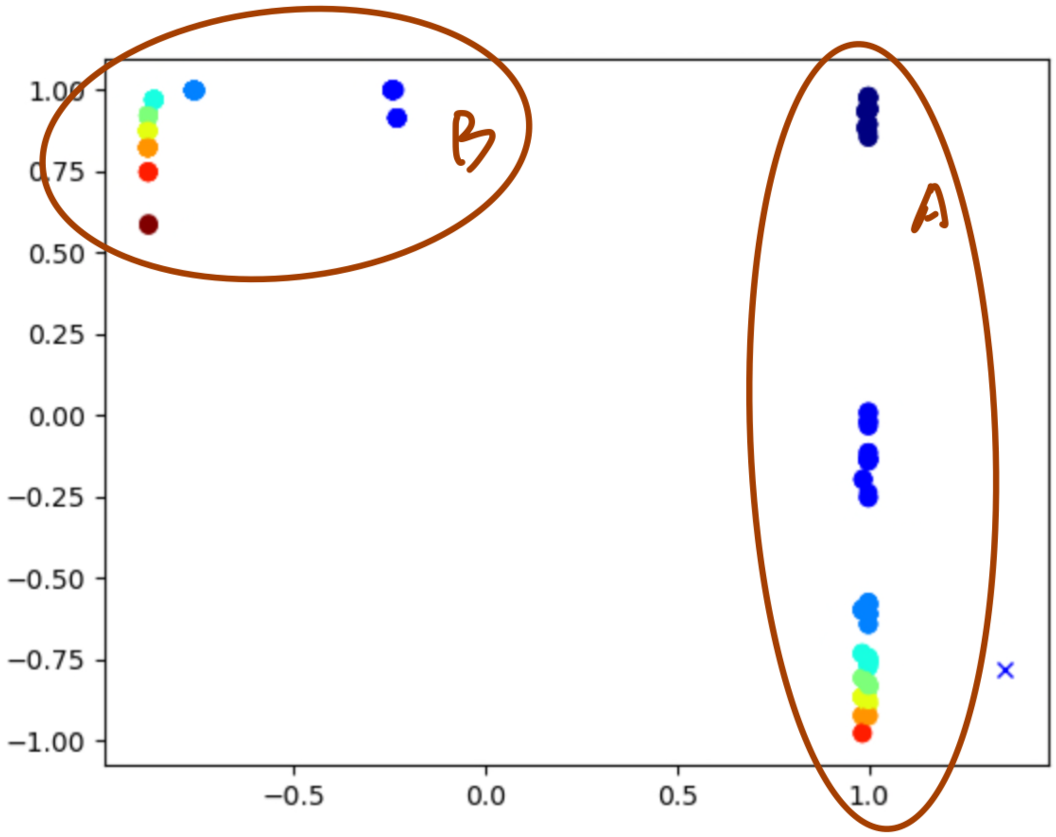
Here is the result I get from this case:

```
color = 01012101210123456543210121 0
symbol= ABAABBAABBAAAAAABBBBBBAABBA
label = 010011001100000011111100110
hidden activations and output probabilities:
B [-0.23  0.91] [ 0.86  0.14]
A [ 0.99  0.88] [ 0.95  0.05]
A [-0.24  1.  ] [ 0.92  0.08]
B [-0.76  1.  ] [ 0.87  0.13]
B [ 0.98 -0.2 ] [ 0.  1.]

A [ 1.   0.9] [ 0.96  0.04]
A [-0.24  1.  ] [ 0.92  0.08]
B [-0.76  1.  ] [ 0.87  0.13]
B [ 0.98 -0.19] [ 0.  1.]
A [ 1.   0.9] [ 0.96  0.04]
A [-0.24  1.  ] [ 0.92  0.08]
A [-0.76  1.  ] [ 0.87  0.13]
A [-0.87  0.97] [ 0.82  0.18]
A [-0.88  0.92] [ 0.75  0.25]
A [-0.88  0.88] [ 0.68  0.32]
B [-0.88  0.82] [ 0.58  0.42]
B [ 0.98 -0.87] [ 0.  1.]
B [ 1.   -0.83] [ 0.  1.]
B [ 1.   -0.76] [ 0.  1.]
B [ 1.   -0.61] [ 0.  1.]
B [ 1.   -0.13] [ 0.01  0.99]
A [ 1.    0.94] [ 0.97  0.03]
A [-0.24  1.  ] [ 0.92  0.08]

B [-0.76  1.  ] [ 0.87  0.13]
B [ 0.98 -0.2 ] [ 0.  1.]
A [ 1.   0.9] [ 0.96  0.04]
epoch: 100000
error: 0.0060
```

resulting figure:



---

3. [1 mark] Briefly explain how the *anbn* prediction task is achieved by the network, based on the figure you generated in Question 2. Specifically, you should describe how the hidden unit activations change as the string is processed, and how it is able to correctly predict the last B in each sequence as well as the following A.

- **Overview analysis:**

At the beginning of this result, the symbol is `ABAABBAABBAAAAAABBBBBBAABBA`, and every character `B` will occur below `A` with the same number of appearances.

According to the result, we can turn our attention to the lines below:

```
hidden activations and output probabilities:
...
1 A [ 1.   0.9] [ 0.96  0.04]
2 A [-0.24  1.  ] [ 0.92  0.08]
3 B [-0.76  1.  ] [ 0.87  0.13]
```

```
4 B [ 0.98 -0.19] [ 0.  1.]
5 A [ 1.   0.9] [ 0.96  0.04]
6 A [-0.24  1.  ] [ 0.92  0.08]
7 A [-0.76  1.  ] [ 0.87  0.13]
8 A [-0.87  0.97] [ 0.82  0.18]
9 A [-0.88  0.92] [ 0.75  0.25]
10 A [-0.88  0.88] [ 0.68  0.32]
11 B [-0.88  0.82] [ 0.58  0.42]
12 B [ 0.98 -0.87] [ 0.  1.]
13 B [ 1.   -0.83] [ 0.  1.]
14 B [ 1.   -0.76] [ 0.  1.]
15 B [ 1.   -0.61] [ 0.  1.]
16 B [ 1.   -0.13] [ 0.01  0.99]
17 A [ 1.    0.94] [ 0.97  0.03]
18 A [-0.24  1.  ] [ 0.92  0.08]
...
```

- **Analysis of the probabilities**

Over the time from the fourth to the fifth row:

```
hidden activations and output probabilities:
B [ 0.98 -0.19] [ 0.  1.]
A [ 1.   0.9] [ 0.96  0.04]
```

we can find -**0.19** up to **0.9** dramatically. And the content of the second square brackets of **B** is `[0, 1.],` which means the probability of **B** is up to 100%. As we can see, the next result that comes up is **A.** Furthermore, in rows 16 and 17, the expected **A** is `0.99`. After B appears six times(11 - 16), the appearance turns to **A.**

- **Analysis of the hidden activations**

Continue to look at the fifth raw:

```
hidden activations and output probabilities:
...
5 A [ 1.   0.9] [ 0.96  0.04]
6 A [-0.24  1.  ] [ 0.92  0.08]
7 A [-0.76  1.  ] [ 0.87  0.13]
8 A [-0.87  0.97] [ 0.82  0.18]
9 A [-0.88  0.92] [ 0.75  0.25]
10 A [-0.88  0.88] [ 0.68  0.32]
...
```

The dimensional of the first dimensional is 1, and the second dimensional is 0.9. As time go by, the first-dimensional decrease continues. When the value decreases to a certain extent, the value of the second-dimensional start to decrease. This scenario is shown in the rows of **5-8.**

Then we look at line **11.**

```
hidden activations and output probabilities:
...
11 B [-0.88  0.82] [ 0.58  0.42]
12 B [ 0.98 -0.87] [ 0.  1.]
13 B [ 1.   -0.83] [ 0.  1.]
14 B [ 1.   -0.76] [ 0.  1.]
15 B [ 1.   -0.61] [ 0.  1.]
16 B [ 1.   -0.13] [ 0.01  0.99]
17 A [ 1.    0.94] [ 0.97  0.03]
...
```

The hidden activations of line 11 in the first square bracket are similar to line 11 because the current layer of line 11 is inputted by the prior layer. After line 11, for the second **B,** we can find the second hidden activation is `-0.87,` this number changed to a great extent compared with the former one. This is because the activation function of **A and B** is different.

The second dimensional of **B** record the appearance of **B.** The value of this dimensional decrease with time goes by. When the value is about to rise to a positive value, it means the model expects the next appearance to be **A.**

Overall, this SRN model can predict the transition from **B** to **A** accurately.

---

4. [1 mark] Train an SRN on the *anbncn* language prediction task by typingThe SRN now has 3 inputs, 3 hidden units and 3 outputs. Again, the "state" is used to count up the A's and count down the B's and C's. Continue training (re-starting, if necessary) for 200k epochs, or until the network is able to reliably predict all the C's as well as the subsequent A, and the error is consistently in the range of 0.01 or 0.02. Rotate the figure in 3 dimensions to get one or more good view(s) of the points in hidden unit space.

```
python3 seq_train.py --lang anbncn
```

After the training finishes, plot the hidden unit activations by typing

```
python3 seq_plot.py --lang anbncn --epoch 200
```

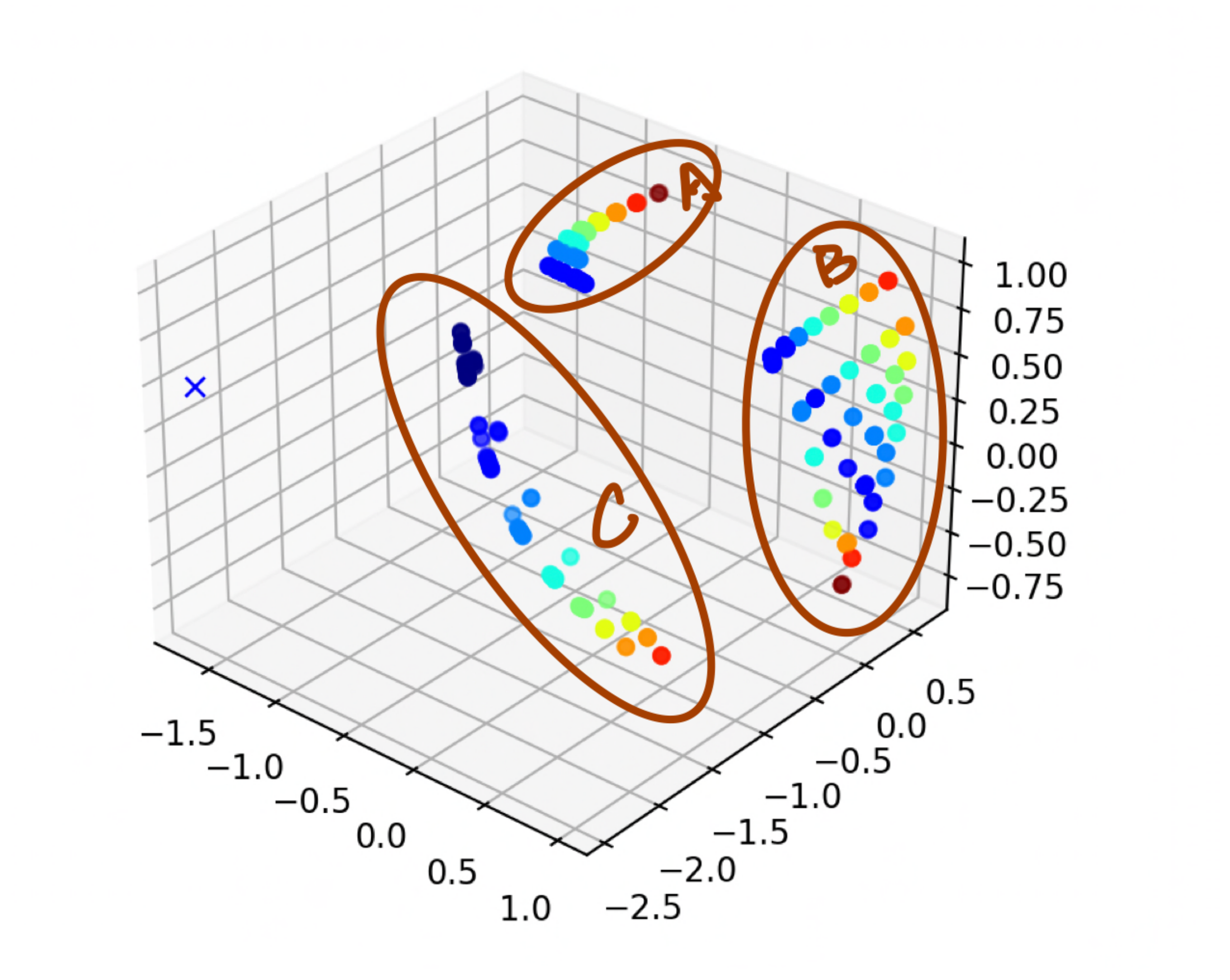
Result:

```
color = 0123456543216543210123432143210123432143210123213210123456543216543210
symbol= AAAAAABBBBBBCCCCCCAAAABBBBCCCCAAAABBBBCCCCAAABBBCCCAAAAAABBBBBBCCCCCCA
label = 00000011111122222200001111222200001111222200011122200000011111122222220
hidden activations and output probabilities:
```

```
A [-0.36 -0.81  1.  ] [ 0.85  0.15  0.  ]
A [-0.5  -0.61  1.  ] [ 0.86  0.14  0.  ]
A [-0.56 -0.46  1.  ] [ 0.85  0.15  0.  ]
A [-0.59 -0.32  1.  ] [ 0.8  0.2  0. ]
A [-0.59 -0.17  1.  ] [ 0.73  0.27  0.  ]
B [-0.59 -0.01  1.  ] [ 0.61  0.39  0.  ]
B [ 0.95  0.08  0.95] [ 0.  1.  0.]
B [ 1.    0.24  0.64] [ 0.  1.  0.]
B [ 1.    0.31  0.4 ] [ 0.  1.  0.]
B [ 1.    0.31  0.17] [ 0.  1.  0.]
B [ 1.    0.23 -0.08] [ 0.  1.  0.]
C [ 1.    0.06 -0.35] [ 0.    0.01  0.99]
C [ 0.24 -0.96 -0.68] [ 0.  0.  1.]
C [-0.06 -1.   -0.71] [ 0.  0.  1.]
C [-0.28 -1.   -0.61] [ 0.  0.  1.]
C [-0.51 -1.   -0.44] [ 0.  0.  1.]
C [-0.74 -1.   -0.14] [ 0.  0.  1.]
A [-0.9  -1.    0.32] [ 0.89  0.    0.1 ]
A [-0.2  -0.83  1.  ] [ 0.71  0.29  0.  ]
A [-0.39 -0.62  1.  ] [ 0.78  0.22  0.  ]
A [-0.5  -0.46  1.  ] [ 0.8  0.2  0. ]
B [-0.56 -0.32  1.  ] [ 0.77  0.23  0.  ]
B [ 0.95 -0.26  0.94] [ 0.  1.  0.]
B [ 1.   -0.14  0.6 ] [ 0.  1.  0.]
B [ 1.   -0.12  0.31] [ 0.    0.91  0.09]
C [ 1.   -0.19 -0.  ] [ 0.  0.  1.]
C [-0.18 -0.97 -0.46] [ 0.  0.  1.]
C [-0.58 -1.   -0.35] [ 0.  0.  1.]
C [-0.8 -1.  -0. ] [ 0.  0.  1.]
A [-0.93 -1.    0.48] [ 0.99  0.    0.  ]
A [-0.39 -0.81  1.  ] [ 0.87  0.13  0.  ]
A [-0.51 -0.61  1.  ] [ 0.87  0.13  0.  ]
A [-0.57 -0.46  1.  ] [ 0.85  0.15  0.  ]
B [-0.59 -0.32  1.  ] [ 0.81  0.19  0.  ]
B [ 0.94 -0.26  0.94] [ 0.  1.  0.]
B [ 1.   -0.14  0.6 ] [ 0.  1.  0.]
B [ 1.   -0.12  0.31] [ 0.    0.91  0.09]
C [ 1.  -0.2 -0. ] [ 0.  0.  1.]
C [-0.19 -0.97 -0.46] [ 0.  0.  1.]
C [-0.58 -1.   -0.35] [ 0.  0.  1.]
C [-0.8 -1.   0. ] [ 0.  0.  1.]
A [-0.94 -1.    0.49] [ 0.99  0.    0.  ]
A [-0.4  -0.81  1.  ] [ 0.87  0.13  0.  ]
A [-0.52 -0.61  1.  ] [ 0.87  0.13  0.  ]
B [-0.57 -0.46  1.  ] [ 0.85  0.15  0.  ]
B [ 0.94 -0.4   0.94] [ 0.  1.  0.]
B [ 1.   -0.29  0.58] [ 0.    0.92  0.08]
C [ 1.   -0.29  0.27] [ 0.    0.02  0.98]
C [-0.47 -0.97 -0.22] [ 0.  0.  1.]
C [-0.82 -1.    0.08] [ 0.02  0.    0.98]
A [-0.95 -1.    0.56] [ 1.  0.  0.]
A [-0.47 -0.8   1.  ] [ 0.91  0.09  0.  ]
A [-0.56 -0.61  1.  ] [ 0.9  0.1  0. ]
A [-0.59 -0.46  1.  ] [ 0.87  0.13  0.  ]
A [-0.6  -0.32  1.  ] [ 0.82  0.18  0.  ]
A [-0.6  -0.17  1.  ] [ 0.74  0.26  0.  ]
B [-0.59 -0.01  1.  ] [ 0.62  0.38  0.  ]
B [ 0.95  0.08  0.95] [ 0.  1.  0.]
```

```
B [ 1.    0.24  0.64] [ 0.  1.  0.]
B [ 1.    0.31  0.4 ] [ 0.  1.  0.]
B [ 1.    0.3   0.17] [ 0.  1.  0.]
B [ 1.    0.23 -0.08] [ 0.  1.  0.]
C [ 1.    0.06 -0.35] [ 0.  0.  1.]
C [ 0.24 -0.96 -0.69] [ 0.  0.  1.]
C [-0.06 -1.   -0.71] [ 0.  0.  1.]
C [-0.27 -1.   -0.62] [ 0.  0.  1.]
C [-0.5  -1.   -0.45] [ 0.  0.  1.]
C [-0.73 -1.   -0.15] [ 0.  0.  1.]
A [-0.9  -1.    0.31] [ 0.86  0.    0.13]
epoch: 200000
error: 0.0020
```

Result Figure:



5. [1 mark] Briefly explain how the *anbncn* prediction task is achieved by the network, based on the figure you generated in Question 4. Specifically, you should describe how the hidden

unit activations change as the string is processed, and how it is able to correctly predict the last B in each sequence as well as all of the C's and the following A.

Answer:

- **Overview analysis:**

At the beginning of this result, the symbol is `AAAAAABBBBBBCCCCCCAAAABBBBCCCCAAAABBBBCCCCAAABBBCCCAAAAAABBBBBBCCCCCCA`, every character `B` will occur below `A` with the same number of appearances, which is as same as `C`

- **hidden activations analysis:**

According to the result, we can turn our attention to the lines below:

```
hidden activations and output probabilities:
1 C [-0.74 -1.   -0.14] [ 0.  0.  1.]
2 A [-0.9  -1.    0.32] [ 0.89  0.    0.1 ]
3 A [-0.2  -0.83  1.  ] [ 0.71  0.29  0.  ]
4 A [-0.39 -0.62  1.  ] [ 0.78  0.22  0.  ]
5 A [-0.5  -0.46  1.  ] [ 0.8  0.2  0. ]
6 B [-0.56 -0.32  1.  ] [ 0.77  0.23  0.  ]
7 B [ 0.95 -0.26  0.94] [ 0.  1.  0.]
8 B [ 1.   -0.14  0.6 ] [ 0.  1.  0.]
9 B [ 1.   -0.12  0.31] [ 0.    0.91  0.09]
10 C [ 1.   -0.19 -0.  ] [ 0.  0.  1.]
11 C [-0.18 -0.97 -0.46] [ 0.  0.  1.]
12 C [-0.58 -1.   -0.35] [ 0.  0.  1.]
13 C [-0.8 -1.  -0. ] [ 0.  0.  1.]
14 A [-0.93 -1.    0.48] [ 0.99  0.    0.  ]
```

For the first raw, we can know the first dimension is -0.74, the second dimension is -1, and the third dimension is -0.14. When the C turn to the A, the hidden activations are similar to the prior one, which means that hidden activations in row 2 are using the hidden activations in row 1.

However, after the iteration of row 3, we can find the activations have changed. This can be presented by the dramatic change of `0.9` to `-0.2`, `-1` to `-0.83`and `0.32` to `1`.

After for appearance of character **A,** we can find that **B** appears. For this transmission. This SRN can not predict the expected appearance of **B** from **A,** but we can find that when the second-dimensional decrease from `-0.32` to `-0.12`(raw **6** - raw **9**), the value increases by `0.12` for every step. When the value of 0 is about to be reached by the second dimension, it estimates that C is expected to appear in the next raw.

This scenario is as same as row 13. the third-dimensional decreases to **0**, which means that the next row is **A.**

- **Output probabilities analysis**

We can also use the case in the prior case, hidden unit analysis. In the second row, we can know that character A is expected for `0.89`. It is the right estimation.

The third dimensional of this column means the expected probability for this row is **C**. The second dimensional of this column means the expected probability for this row is **B**, and the first dimensional of this column means the expected probability for this row is **A.** For row 6, this model can not predict the first appearance of character `B`. However, cause the SRN know the prior count of the appearance of **A** is `4`, then the second appearance must be **B(row 7).** The probability can also be observed by the second dimensional(100%).

And SRN can also predict the appearance of the first character **C** after **B,** cause the number of appearances of B is as same as **A.** This scenario can also be observed from the raw 13 to 14.

In a word, the model trained this time is correct.

6. [4 marks] This question is intended to be more challenging. Train an LSTM network to predict the Embedded Reber Grammar, by typing

```
python3 seq_train.py --lang reber --embed True --model lstm --hid 4
```

You can adjust the number of hidden nodes if you wish. Once the training is successful, try to analyse the behaviour of the LSTM and explain how the task is accomplished (this might involve modifying the code so that it returns and prints out the context units as well as the hidden units).

Answer:

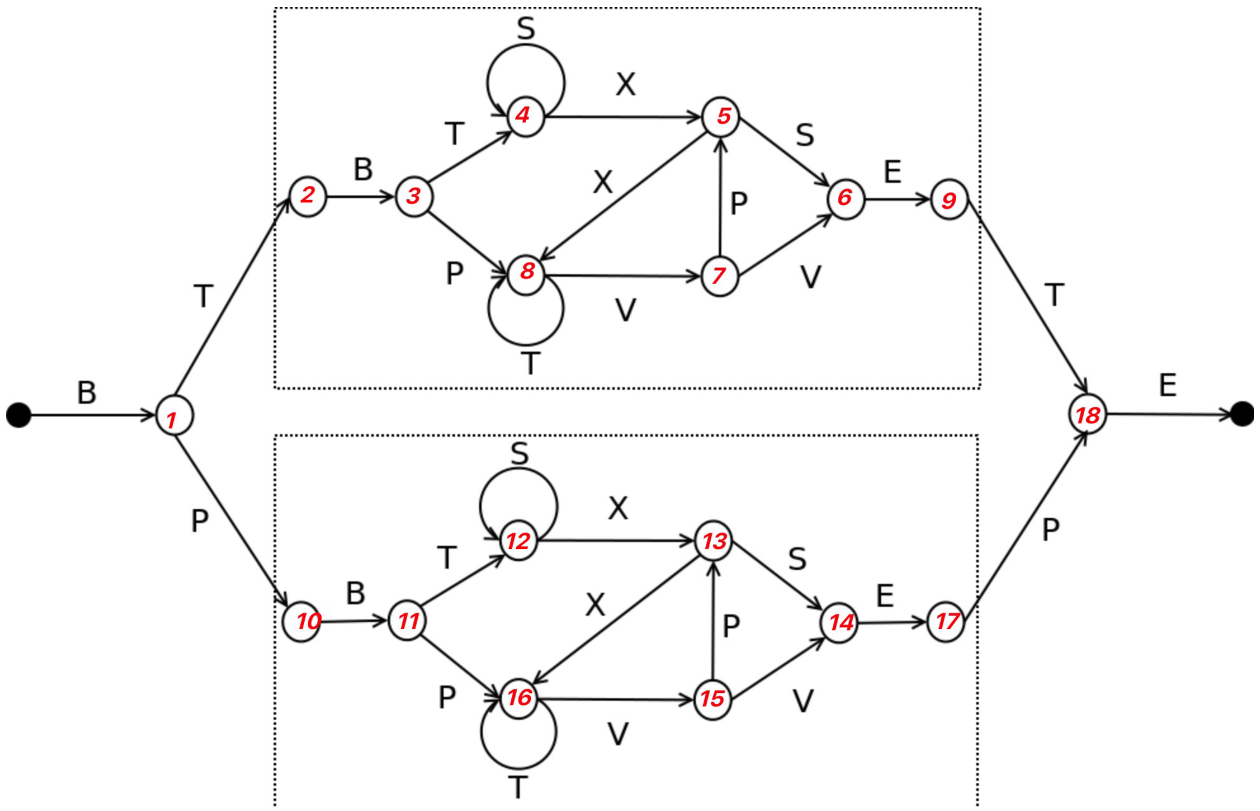


Figure: reference from [1]

For the choice of parameters, I use `5` as the number of hidden layer.

Then we can get the result below:

```
state =  0 1 10 11 16 15 14 17 18
symbol= BPBPVVEPE
label = 040455646
true probabilities:
     B    T    S    X    P    V    E
1 [ 0.   0.5  0.   0.   0.5  0.   0. ]
10 [ 1.  0.  0.  0.  0.  0.  0.]
11 [ 0.   0.5  0.   0.   0.5  0.   0. ]
16 [ 0.   0.5  0.   0.   0.   0.5  0. ]
15 [ 0.   0.   0.   0.   0.5  0.5  0. ]
14 [ 0.  0.  0.  0.  0.  0.  1.]
17 [ 0.  0.  0.  0.  1.  0.  0.]
18 [ 0.  0.  0.  0.  0.  0.  1.]
hidden activations and output probabilities [BTSXPVE]:
1 [-0.56  0.27  0.75 -0.46 -0.73] [ 0.   0.5  0.   0.   0.5  0.   0. ]
10 [-0.05  0.83 -0.39 -0.8  -0.29] [ 1.  0.  0.  0.  0.  0.  0.]
11 [-0.94  0.91  0.74  0.48 -0.77] [ 0.    0.48  0.    0.    0.52  0.    0.  ]
16 [-0.08  0.89  0.62  0.81  0.66] [ 0.    0.42  0.    0.    0.    0.58  0.  ]
15 [-0.87 -0.73  0.89  0.95  0.51] [ 0.   0.   0.   0.   0.4  0.6  0. ]
14 [-0.87 -0.87 -0.68  0.84 -0.36] [ 0.  0.  0.  0.  0.  0.  1.]
17 [-0.86 -0.47  0.72  0.02 -0.83] [ 0.    0.01  0.    0.    0.99  0.    0.  ]
18 [-0.03 -0.81 -0.65  0.77 -0.64] [ 0.  0.  0.  0.  0.  0.  1.]
```

```
epoch: 50000
error: 0.0016
final: 0.0000
```

For the first hidden layer, we can know the probability of choosing `T` or `P` is equal. The reason of the action during the beginning is LSTM can choose gate randomly. According to the result, we can know the model chooses the gate 10 at the beginning, and choose the gate **17** at the end.

The model of LSTM is long term memory, which can predict as same result as the beginning.

In order to dig more information from the code, we can store the paths below in the `seq_plot.py`:

```
['B', 'P', 'B', 'P', 'V', 'P', 'S', 'E', 'P', 'E', 'B', 'T', 'B', 'T', 'S', 'X', 'S', 'E', 'T', 'E']
```

The first half is:

```
['B', 'P', 'B', 'P', 'V', 'P', 'S', 'E', 'P', 'E']
```

which is belongs to the up part.

And the second part is:

```
['B', 'T', 'S', 'X', 'S', 'E', 'T', 'E']
```

Both of them have the same second gate and the penultimate gate individually.

References:

[1]:- http://christianherta.de/lehre/dataScience/machineLearning/neuralNetworks/reberGrammar.php