# COMP9444 24T2 Assignment 1 Report

***Authored by Mohammad Mayaz Rakib (z5361151)***

## Declaration

I declare that:

- This assessment item is entirely my own original work, except where I have acknowledged use of source material such as books, journal articles, other published material, the Internet, and the work of other student/s or any other person/s.
- This assessment item has not been submitted for assessment for academic credit in this, or any other course, at UNSW or elsewhere.

I understand that:

- The assessor of this assessment item may, for the purpose of assessing this item, reproduce this assessment item and provide a copy to another member of the University.
- The assessor may communicate a copy of this assessment item to a plagiarism checking service (which may then retain a copy of the assessment item on its database for the purpose of future plagiarism checking).

I certify that I have read and understood the University Rules in respect of Student Academic Misconduct.

*Mohammad Mayaz Rakib*
*26/06/2024*

## Part 1: Japanese Character Recognition

**1.**
The tenth and final training epoch produced the following output after executing the training for the linear neural network.

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.823565
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.633543
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.593817
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.600265
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.323052
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.510742
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.655205
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.618373
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.351016
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.666157
<class 'numpy.ndarray'>
[[771.   5.   7.  12.  29.  63.   2.  62.  32.  17.]
 [  7. 666. 108.  18.  30.  23.  59.  12.  26.  51.]
 [  7.  59. 695.  27.  26.  21.  46.  37.  45.  37.]
 [  4.  35.  62. 757.  15.  58.  14.  18.  26.  11.]
 [ 61.  55.  79.  20. 620.  20.  34.  34.  20.  57.]
 [  8.  27. 124.  17.  20. 726.  26.   9.  32.  11.]
 [  5.  23. 144.  10.  27.  25. 722.  19.  11.  14.]
 [ 15.  28.  27.  13.  86.  17.  54. 621.  91.  48.]
 [ 11.  35.  93.  41.   6.  30.  45.   7. 711.  21.]
 [  8.  51.  86.   3.  54.  31.  19.  31.  41. 676.]]

Test set: Average loss: 1.0093, Accuracy: 6965/10000 (70%)
```

The final accuracy is exactly `70%` as required. This is indicated by the output as follows:

```
Accuracy: 6965/10000 (70%)
```

The confusion matrix is as follows:

```
[[771.   5.   7.  12.  29.  63.   2.  62.  32.  17.]
 [  7. 666. 108.  18.  30.  23.  59.  12.  26.  51.]
 [  7.  59. 695.  27.  26.  21.  46.  37.  45.  37.]
 [  4.  35.  62. 757.  15.  58.  14.  18.  26.  11.]
 [ 61.  55.  79.  20. 620.  20.  34.  34.  20.  57.]
 [  8.  27. 124.  17.  20. 726.  26.   9.  32.  11.]
 [  5.  23. 144.  10.  27.  25. 722.  19.  11.  14.]
 [ 15.  28.  27.  13.  86.  17.  54. 621.  91.  48.]
```

```
 [ 11.  35.  93.  41.   6.  30.  45.   7. 711.  21.]
 [  8.  51.  86.   3.  54.  31.  19.  31.  41. 676.]]
```

Each row is an actual class in this model, or in other words, a character in the simplified Hiragana dataset. Each column is a predicted class made by the model, or in other words, the character the model *thinks* is the right one.

Each entry of the matrix $C_{i,j}$ corresponds the number of instances of class $i$ that were predicted as class $j$. The diagonal entries $C_{i,i}$ represent the correctly classified instances for class $i$. This can be observed in our own data above.

**2.**
The tenth and final training epoch produced the following output after executing the training for the fully-connected 2-layer neural network.

```
Train Epoch: 10 [0/60000 (0%)] Loss: 0.357899
Train Epoch: 10 [6400/60000 (11%)] Loss: 0.227345
Train Epoch: 10 [12800/60000 (21%)] Loss: 0.244904
Train Epoch: 10 [19200/60000 (32%)] Loss: 0.212764
Train Epoch: 10 [25600/60000 (43%)] Loss: 0.134328
Train Epoch: 10 [32000/60000 (53%)] Loss: 0.248591
Train Epoch: 10 [38400/60000 (64%)] Loss: 0.213900
Train Epoch: 10 [44800/60000 (75%)] Loss: 0.359252
Train Epoch: 10 [51200/60000 (85%)] Loss: 0.110932
Train Epoch: 10 [57600/60000 (96%)] Loss: 0.270949
<class 'numpy.ndarray'>
[[850.   3.   1.   5.  29.  30.   5.  43.  29.   5.]
 [  5. 824.  29.   3.  17.  11.  61.   6.  18.  26.]
 [  5.   9. 840.  44.  12.  21.  25.  13.  19.  12.]
 [  6.  10.  27. 917.   0.  14.   5.   2.   8.  11.]
 [ 40.  34.  18.   8. 816.   6.  25.  16.  19.  18.]
 [ 10.  18.  81.  12.  12. 825.  21.   2.  12.   7.]
 [  3.  15.  52.   7.  15.   3. 890.   6.   1.   8.]
 [ 18.  14.  20.   7.  21.  11.  32. 826.  18.  33.]
 [  8.  22.  27.  53.   5.  10.  32.   3. 833.   7.]
 [  1.  21.  45.   6.  31.   6.  20.  12.  13. 845.]]

Test set: Average loss: 0.5032, Accuracy: 8466/10000 (85%)
```

The final accuracy is `85%` as required. This is indicated by the output as follows:

```
Accuracy: 8466/10000 (85%)
```

The confusion matrix is as follows:

```
[[850.   3.   1.   5.  29.  30.   5.  43.  29.   5.]
 [  5. 824.  29.   3.  17.  11.  61.   6.  18.  26.]
 [  5.   9. 840.  44.  12.  21.  25.  13.  19.  12.]
 [  6.  10.  27. 917.   0.  14.   5.   2.   8.  11.]
 [ 40.  34.  18.   8. 816.   6.  25.  16.  19.  18.]
 [ 10.  18.  81.  12.  12. 825.  21.   2.  12.   7.]
 [  3.  15.  52.   7.  15.   3. 890.   6.   1.   8.]
 [ 18.  14.  20.   7.  21.  11.  32. 826.  18.  33.]
 [  8.  22.  27.  53.   5.  10.  32.   3. 833.   7.]
 [  1.  21.  45.   6.  31.   6.  20.  12.  13. 845.]]
```

The structure and reperesentation of this confusion matrix is exactly the same as the one described in Question 1.

We can now calculate the total number of independent parameters used in this fully-connected network. We know that the input is a $28 \times 28$ image, flattened, thus giving us the input size $28 \times 28 = 784$.
Let the number of units in the hidden layer be $H$. We can deduce that the number of weights between the input and hidden layer is $H \times 784$ and that the number of weights between the hidden layer and output layer is $H \times 10$. Overall, the number of weights in total are $(H \times 784) + (H \times 10)$.

We must also not forget that there are bias terms for the hidden layer and output layer, amounting to $H$ and $10$ bias terms respectively, Overall, the number of biases in total are $H + 10$.

Thus, the total number of independent parameters, which is the sum of all weights and biases in the neural network, are given by the expression:

$$T_p = (794 \times H) + (10 \times H) + H + 10$$

In the above results, we used a hidden size of 200, and thus, $H = 200$.

For this reason, we can calculate the expression as follows:

$$T_p = (794 \times 200) + (10 \times 200) + 200 + 10 = 159010$$

Thus, the total number of independent parameters in our fully-connected, 2-layer network amounts to $159010$ units.

**3.**

The tenth and final training epoch produced the following output after executing the training for the convolutional neural network.

```
Train Epoch: 10 [0/60000 (0%)] Loss: 0.024619
Train Epoch: 10 [6400/60000 (11%)] Loss: 0.022218
Train Epoch: 10 [12800/60000 (21%)] Loss: 0.052050
Train Epoch: 10 [19200/60000 (32%)] Loss: 0.022322
Train Epoch: 10 [25600/60000 (43%)] Loss: 0.037489
Train Epoch: 10 [32000/60000 (53%)] Loss: 0.047339
Train Epoch: 10 [38400/60000 (64%)] Loss: 0.028575
Train Epoch: 10 [44800/60000 (75%)] Loss: 0.124471
Train Epoch: 10 [51200/60000 (85%)] Loss: 0.010505
Train Epoch: 10 [57600/60000 (96%)] Loss: 0.061329
<class 'numpy.ndarray'>
[[964.   4.   3.   0.  12.   4.   1.   4.   3.   5.]
 [  0. 918.   5.   1.   7.   0.  42.   5.   4.  18.]
 [ 11.   5. 888.  29.   8.   9.  17.   6.   8.  19.]
 [  1.   1.  16. 953.   3.   5.  11.   2.   4.   4.]
 [ 20.   8.   1.   5. 922.   0.  16.   5.  18.   5.]
 [  2.   6.  33.   8.   3. 909.  28.   0.   2.   9.]
 [  3.   0.   9.   2.   3.   0. 979.   1.   0.   3.]
 [ 10.   6.   1.   0.   2.   2.   5. 948.   5.  21.]
 [  6.  13.   5.   1.   3.   1.  11.   0. 953.   7.]
 [  6.   2.   8.   1.  10.   0.   4.   3.  10. 956.]]

Test set: Average loss: 0.2394, Accuracy: 9390/10000 (94%)
```

The final accuracy is `94%` as required. This is indicated by the output as follows:

```
Accuracy: 9390/10000 (94%)
```

The confusion matrix is as follows:

```
[[964.   4.   3.   0.  12.   4.   1.   4.   3.   5.]
 [  0. 918.   5.   1.   7.   0.  42.   5.   4.  18.]
 [ 11.   5. 888.  29.   8.   9.  17.   6.   8.  19.]
 [  1.   1.  16. 953.   3.   5.  11.   2.   4.   4.]
 [ 20.   8.   1.   5. 922.   0.  16.   5.  18.   5.]
 [  2.   6.  33.   8.   3. 909.  28.   0.   2.   9.]
```

```
[  3.   0.   9.   2.   3.   0. 979.   1.   0.   3.]
[ 10.   6.   1.   0.   2.   2.   5. 948.   5.  21.]
[  6.  13.   5.   1.   3.   1.  11.   0. 953.   7.]
[  6.   2.   8.   1.  10.   0.   4.   3.  10. 956.]]
```

The structure and reperesentation of this confusion matrix is exactly the same as the one described in Question 1.

We can now calculate the total number of independent parameters used in this convolutional neural network.

The first convolutional layer has $1$ input channel, $32$ output channels, and a $3 \times 3$ kernel. Thus, the number of parameters per filter is $3 \times 3 = 9$, with a total of $32$ filters, leading to $9 \times 32 = 288$ weight parameters. Each filter also has $1$ bias term, leading to $32$ total bias parameters. Thus, the total number of independent parameters for the first convolutional layer is $288 + 32 = 320$.

The second convolutional layer has $32$ input channels, $64$ output channels, and a $3 \times 3$ kernel. The number of parameters per filter is $3 \times 3 \times 32 = 288$, with a total of $64$ filters, leading to $64 \times 288 = 18432$ weight parameters. Each filter also has $1$ bias term, leading to $64$ total bias parameters. Thus, the total number of independent parameters for the second convolutional layer is $18432 + 64 = 18496$.

The first fully-connected layer has $64 \times 7 \times 7 = 3136$ input features, and $128$ output features, with one bias parameter per output feature, and thus $128$ bias parameters as well. Thus, there are $3136 \times 128 + 128 = 401536$ total independent parameters for the first fully-connected layer.

The second fully-connected layer has $128$ input features, and $10$ output features, with one bias parameter per output feature, and thus $10$ bias parameters as well. Thus, there are $128 \times 10 + 10 = 1290$ total independent parameters for the second fully-connected layer.

Overall, if we add the total number of independent parameters for the two convolutional layers and two fully-connected layers, we get $320 + 18496 + 401536 + 1290 = 421642$ independent parameters in total throughout the whole network.

**4.**
The linear neural network was the worst performing among the three models with an accuracy of `70%`. This is because linear models lack the flexibility and sophistication required to capture complex non-linear patterns in data effectively. We see this lack of complexity with the sheer lack of independent parameters, which totalled to $7840$. The confusion matrix shows significant misclassifications, such as the digit `4` being mistaken for the digit `9`, `7` being mistaken for `2` and `8` being mistaken for `3`, further highlighting the fact that linear models are not sophisticated enough to capture the intricate features needed to distinguish similar-looking digits.

The fully-connected 2-layer neural network performed moderately better than the linear network with an accuracy of `85%`. It can capture more complex patterns due to the added hidden layer and non-linearity, leading to better accuracy. We see a significant increase in independent parameters, totalling to $159010$, more than 20 times more, illustrating the sheer significance of the jump in complexity from the previous type of model. The confusion matrix indicates better performance, but some digits are still frequently misclassified, such as the `2` mistaken for `0`, or `5` mistaken for `3`. Although the fully-connected model captures more features than the linear model, it still struggles with the classification task a little.

The convolutional neural network performed the best out of the three models with an accuracy of `94%`. This is because the convolutional layers allow spatial hierarchies in the image data to be effectively captured, making it the best type of network for image classification tasks like MNIST, or in this case, K-MNIST. We see the most amount of independent parameters in all three models, totalling to $421642$, illustrating the sheer significance of the jump in complexity from the previous type of model. The confusion matrix shows the least misclassifications, with only a few instances of misclassification such as `5` being mistaken for `3` or `9` being mistaken for `4` rarely. Thus, convolutional networks have the necessary degree of complexity to capture the spatial features required to classify digits to a satisfactory degree.

## Part 2: Multi-Layer Perceptron

**1.**
The 2-layer neural network was executed with the following command:

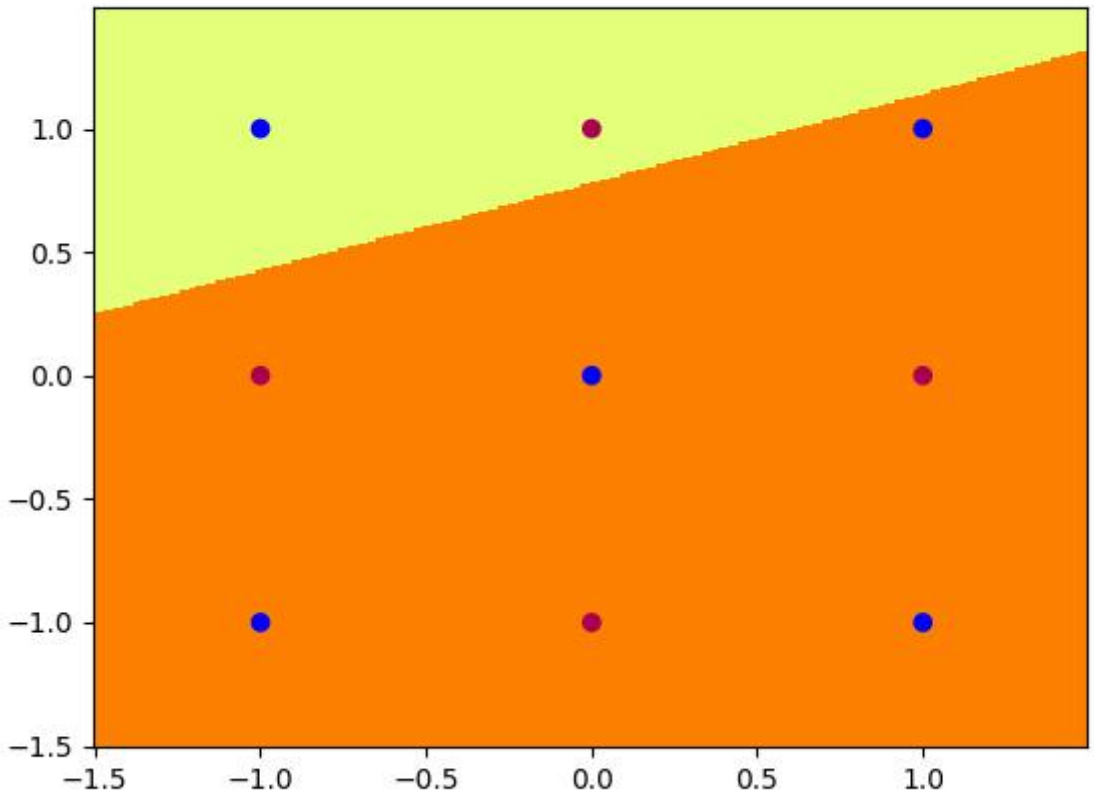
```
python3 check_main.py --hid 6 --act sig --init 0.15 --lr 0.01 --epoch 200000
```

This particular command with this particular configuration reached an accuracy of `100%`, producing the following output:

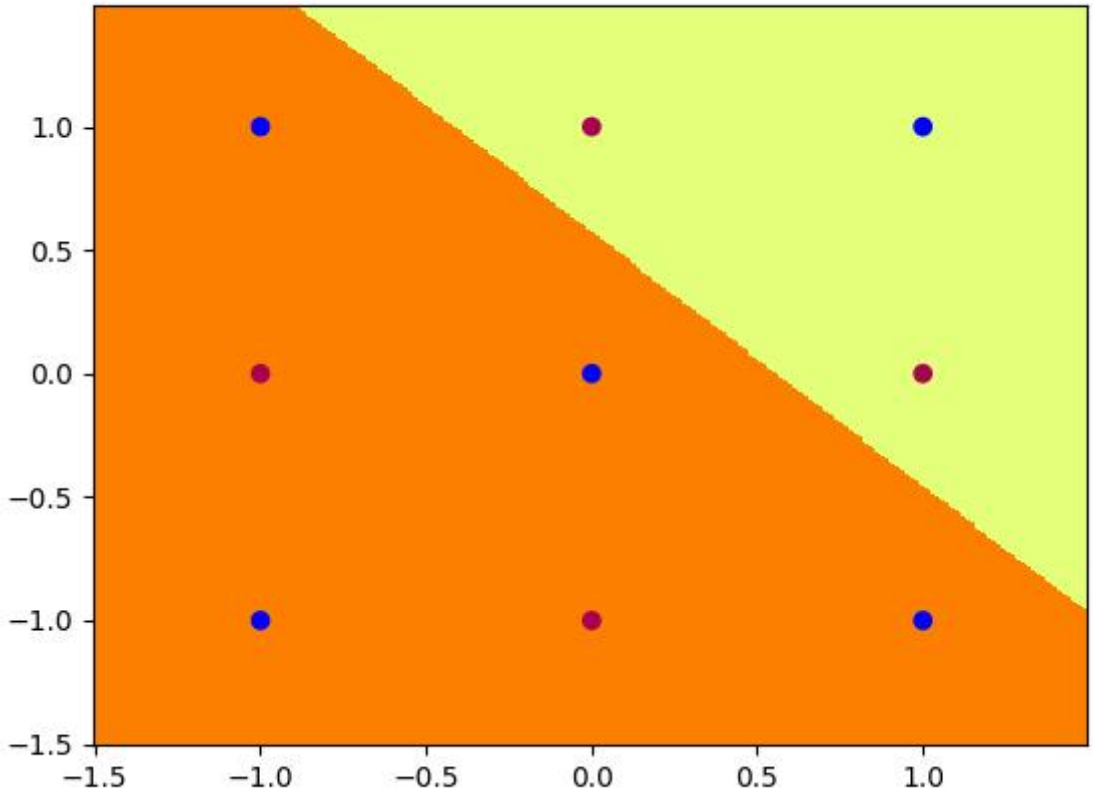
```
ep:  100 loss: 0.6858 acc: 55.56
ep:  200 loss: 0.6401 acc: 66.67
ep:  300 loss: 0.6365 acc: 66.67
ep:  400 loss: 0.6365 acc: 66.67
ep:  500 loss: 0.6365 acc: 66.67
ep:  600 loss: 0.6365 acc: 66.67
ep:  700 loss: 0.6365 acc: 66.67
ep:  800 loss: 0.6365 acc: 66.67
ep:  900 loss: 0.6365 acc: 66.67
ep: 1000 loss: 0.6361 acc: 66.67
ep: 1100 loss: 0.4336 acc: 88.89
```

```
ep: 1200 loss: 0.2111 acc: 100.00
ep: 1300 loss: 0.1344 acc: 100.00
ep: 1400 loss: 0.0519 acc: 100.00
ep: 1500 loss: 0.0347 acc: 100.00
ep: 1600 loss: 0.0261 acc: 100.00
ep: 1700 loss: 0.0202 acc: 100.00
ep: 1800 loss: 0.0112 acc: 100.00
ep: 1900 loss: 0.0087 acc: 100.00
ep: 2000 loss: 0.0074 acc: 100.00
ep: 2100 loss: 0.0066 acc: 100.00
ep: 2200 loss: 0.0060 acc: 100.00
ep: 2300 loss: 0.0055 acc: 100.00
ep: 2400 loss: 0.0050 acc: 100.00
ep: 2500 loss: 0.0047 acc: 100.00
ep: 2600 loss: 0.0044 acc: 100.00
ep: 2700 loss: 0.0041 acc: 100.00
ep: 2800 loss: 0.0039 acc: 100.00
ep: 2900 loss: 0.0037 acc: 100.00
ep: 3000 loss: 0.0035 acc: 100.00
ep: 3100 loss: 0.0034 acc: 100.00
Final Weights:
tensor([[-7.6196, -7.4018],
        [-7.5010,  8.1321],
        [-4.7554, -5.9699],
        [ 4.4434,  5.7572],
        [ 8.7296,  8.5933],
        [ 8.1126, -2.1750]])
tensor([ 4.2481, -3.5015,  8.5662, -8.2136,  4.0343,  1.5002])
tensor([[ 12.4800, -10.8510,  -8.7114,   7.6110,  11.4031, -11.0891]])
tensor([1.7338])
Final Accuracy:  100.0
```

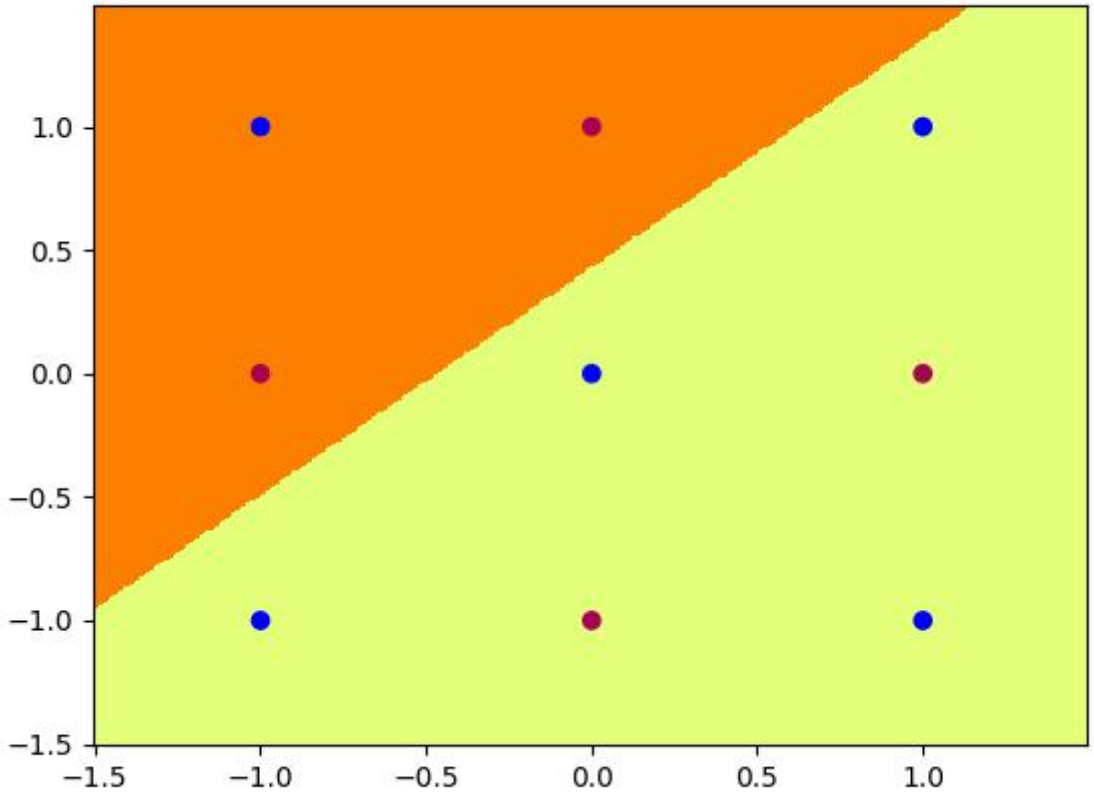
It produced the following graphs. The `check.jpg` graph is:



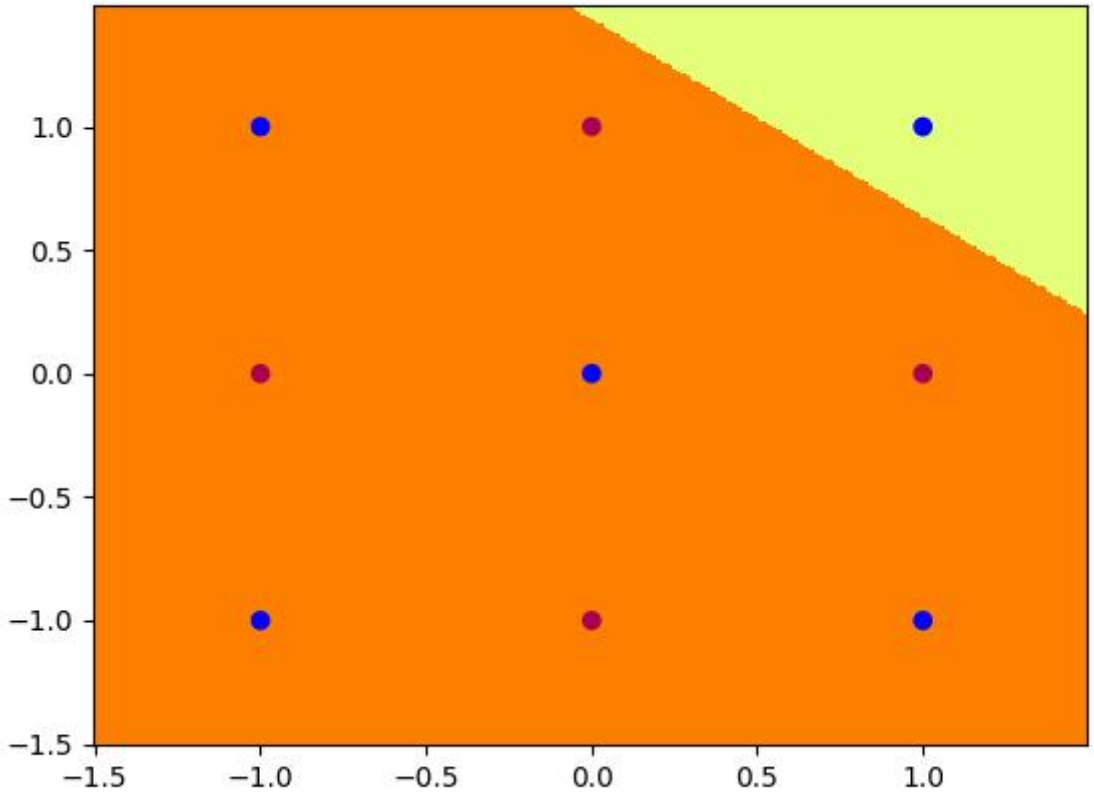

The `hid_6_0.jpg` graph is:



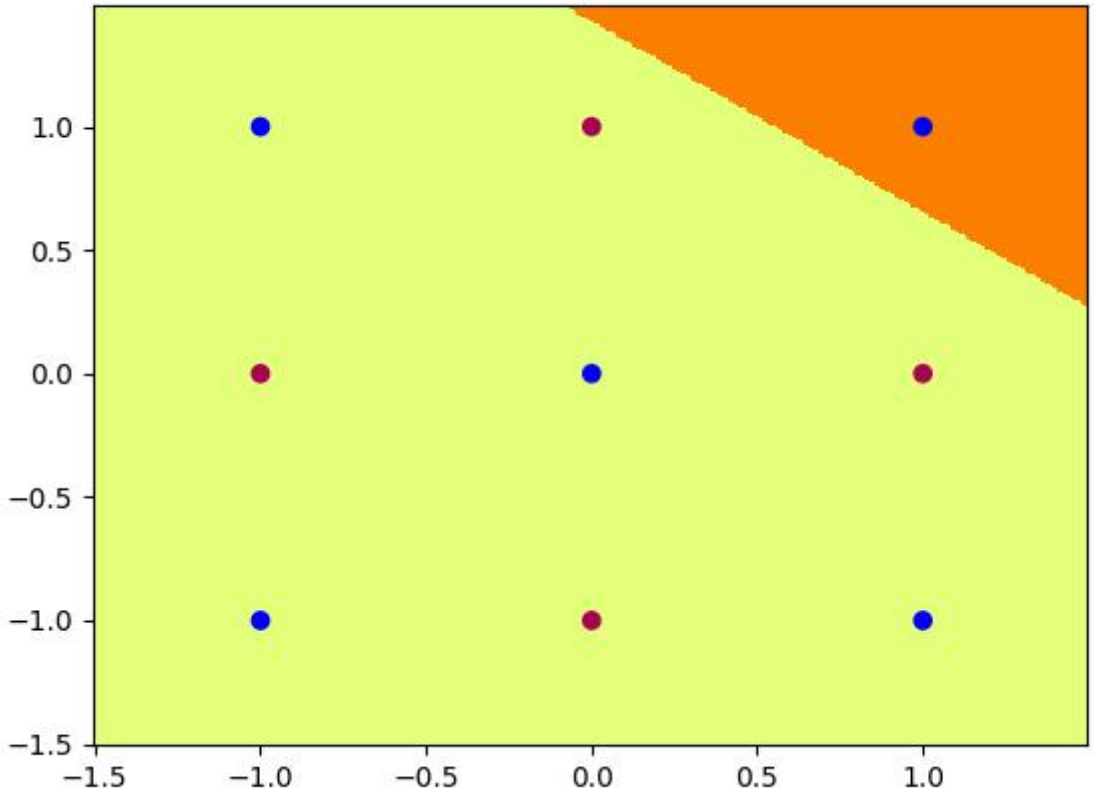

The `hid_6_1.jpg` graph is:



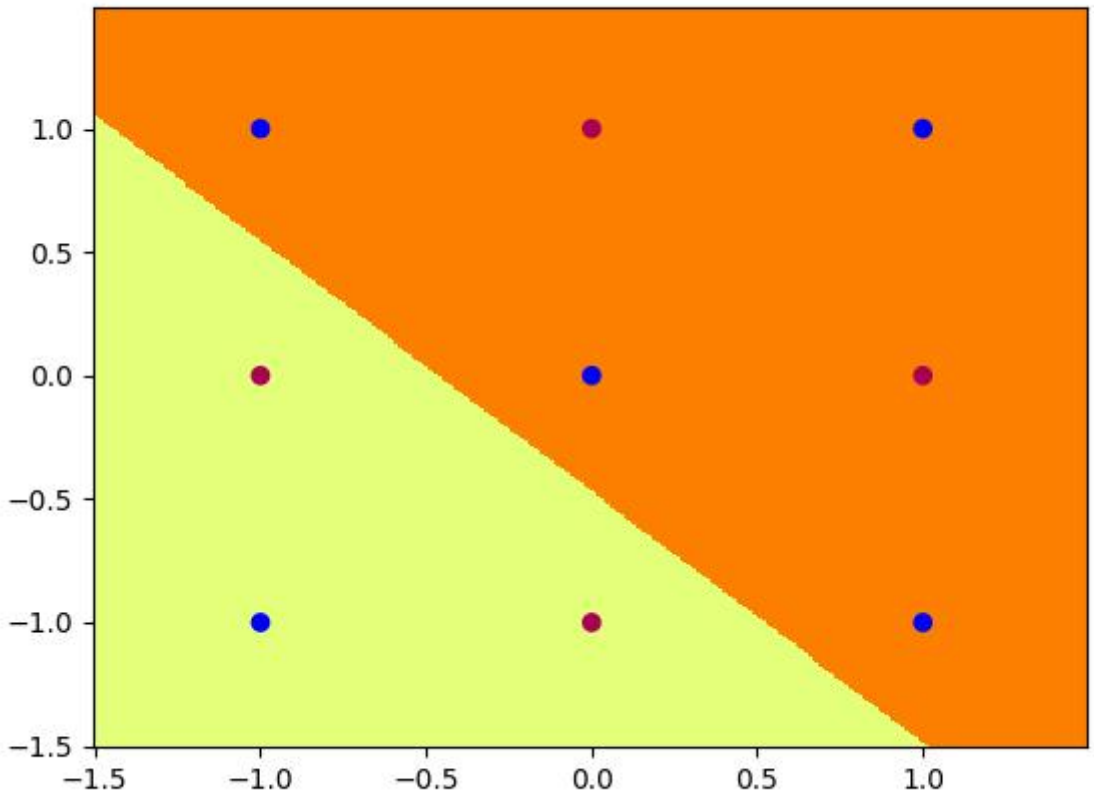

The `hid_6_2.jpg` graph is:



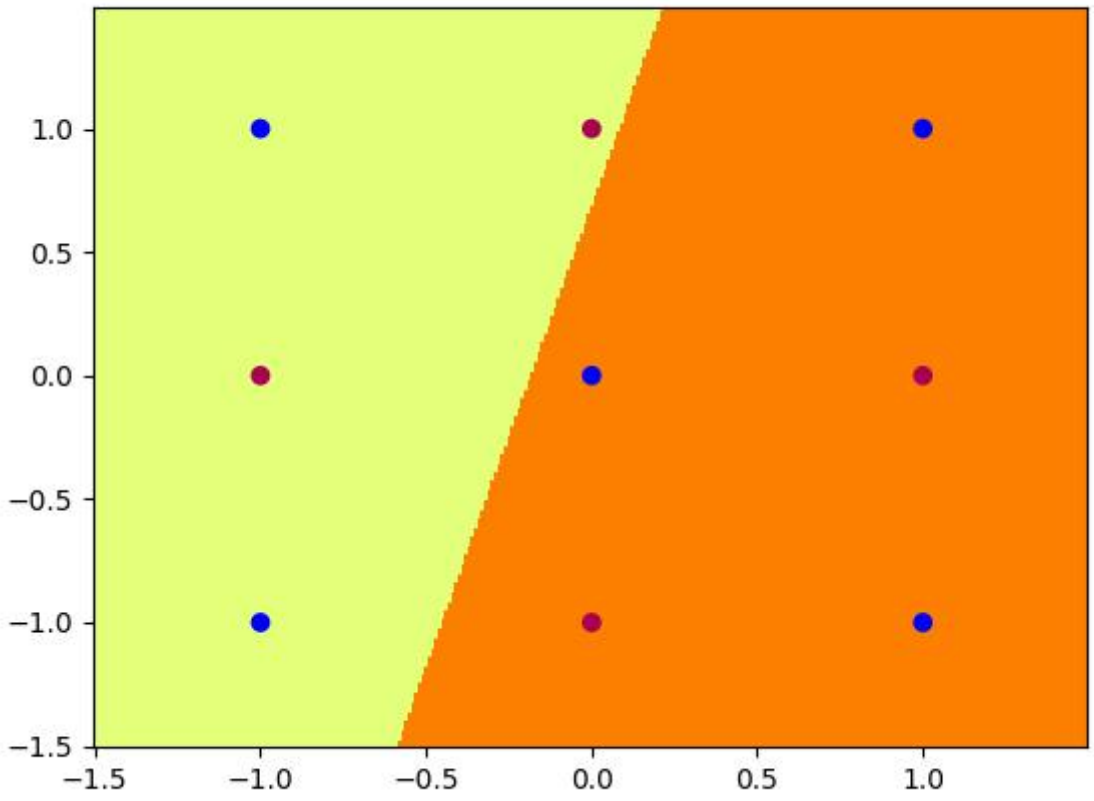

The `hid_6_3.jpg` graph is:



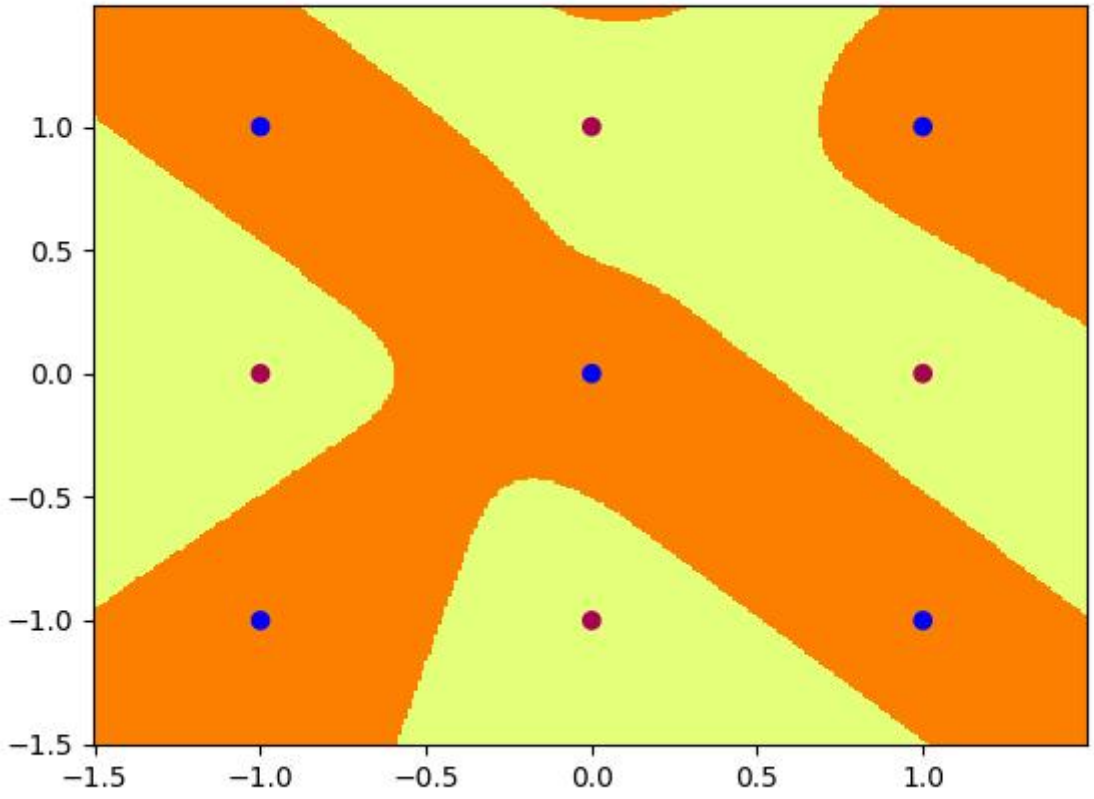

The `hid_6_4.jpg` graph is:




The `hid_6_5.jpg` graph is:




The `out_6.jpg` graph is:




Here is a brief explanation for why these particular parameters worked in generating a result with `100%` accuracy.

Firstly, we used `6` hidden units instead of `5`, increasing the overall capacity of the model and allowing more complex patterns to be captured. Of course, there is a higher risk of overfitting with a larger network, which is why we also tweaked some other parameters, but all in all ended up being a size that provided sufficient complexity.

The sigmoid activation function `sig` is used for both the hidden and output layer as required, and is a good non-linear activation function that is smooth and differentiable, allowing for easier gradient-based optimisation.

The weight initialisation value of `0.15` is relatively small to begin with, which prevents weights from becoming too large and ensures activations in a range where the sigmoid activation function is not saturated.

The learning rate of `0.01` is a common starting point for many optimisation problems, being large enough to make substantial progress during each update step but not so large that it causes drastic changes to weights introducing instability to the training.

The number of epochs was set to `200000`, which is a sufficiently high number of epochs that gives ample time for the network to converge. Once again, too many epochs can risk overfitting, but it ended up being sufficient after trial-and-error in this case, much like the network size.

**2.**
I have to now ensure that the plot of red and blue points work via a handcrafted set of weights and biases for a 4-unit hidden layer 2-layer neural network. I then have to test its accuracy with the following command:

```
python3 check_main.py --act step --hid 4 --set_weights
```

We know that the blue points are at the positions:

$$(-1.0, 1.0), (-1.0, -1.0), (0.0, 0.0), (1.0, 1.0), (1.0, -1.0)$$

We know that the red points are the positions:

$$(-1.0, 0.0), (0.0, -1.0), (0.0, 1.0), (1.0, 0.0)$$

In order to classify this to using only four hidden units, I decided to use the four corresponding decision boundaries described by the following line equations:

- $x_1 + x_2 - 1.5 = 0$
- $x_1 + x_2 - 0.5 = 0$
- $x_1 + x_2 + 0.5 = 0$
- $x_1 + x_2 + 1.5 = 0$

These are diagonal lines going from top-left to bottom-right going through the middle of each set of points.

We can observe from the blue and red data plot that we can group the blue and red points into diagonal sections. We also observe this to some degree in the previous question where we automatically learned the weights. This is why the above diagonal lines are used as the decision boundaries for our categories.

We must now handcraft the weights. Firstly, given the equations of our lines, we can use the coefficients as the values to our weight matrix and the constants as the values to our biases. Thus, we have the weight matrix:

$$W_1 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

and the bias matrix:

$$b_1 = [-1.5, -0.5, 0.5, 1.5]$$

For the output weights and biases, we have to alternate between $-1$ and $1$. This is because, for each diagonal line, we want the blues and reds to be classified on either side of the line alternatively.

This can be a bit hard to understand without an example, so lets starts from the bottom-left and go from there. In the bottom-left corner we have a blue point with the coordinate $(-1, -1)$. Diagonally above it are the red points $(-1, 0)$ and $(0, -1)$. Our bottom-left-most decision boundary, being the line $x_1 + x_2 - 1.5$, classifies these points correctly. However, when we go upwards to the blue points at $(-1, 1)$, $(0, 0)$ and $(1, -1)$, we see that all these points are also classified as red. Thus, we need to alternate the second decision boundary, represented by $x_1 + x_2 - 0.5$, such that it classifies blue *above* it and red *below* it. This alternation reflects the alternative behaviour of the existence of red and blue points in diagonal categories drawn out by the decision boundaries.

Thus, the weights of the output layer are:

$$W_2 = [1, -1, 1, -1]$$

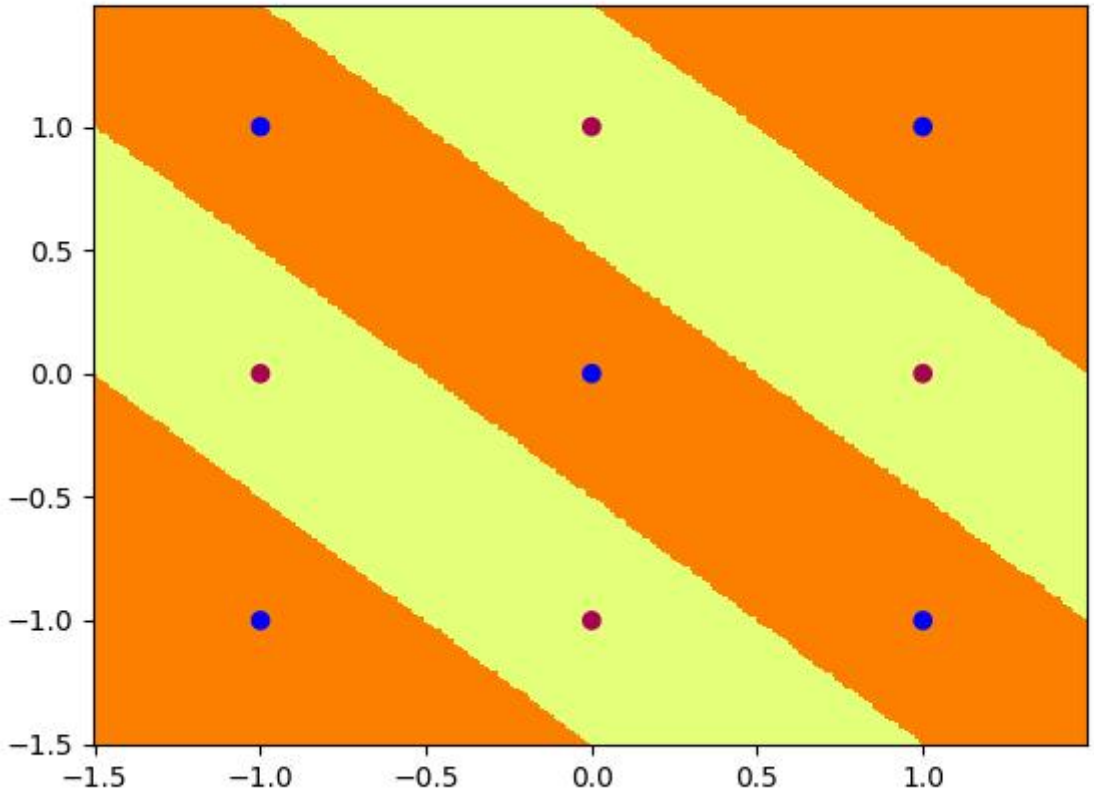
We can leave the bias alone and just set it to $0$, as:

$$b_2 = 0$$

Given this weight configuration for the network, we can test its classification accuracy, which turns out to be `100%`, thus indicating that the handcrafted weights classify the points on the graph perfectly. Here is the output:

```
Initial Weights:
tensor([[1., 1.],
        [1., 1.],
        [1., 1.],
        [1., 1.]])
tensor([-1.5000, -0.5000,  0.5000,  1.5000])
tensor([[ 1., -1.,  1., -1.]])
tensor([0.])
Initial Accuracy:  100.0
```

Our output also contains the following graph, clearly illustrating our decision boundaries:




**3.**

I now have to rescale the weights and biases that were handcrafted in Question 2 so that it works with the sigmoid activation function *as well as* the previously used Heaviside activation function.

To do this, we must first analyse the nature of the Heaviside and sigmoid activation function.

Firstly, we make the simple observation that regardless of how large a positive or negative number is in the Heaviside function, it will always get mapped to either a $0$ or a $1$. This means that we can scale up our values as much as possible, and as long as the ratios are the same, it will work with the Heaviside function the same.

Secondly, we analyse the sigmoid function. Mathematically, it is expressed as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

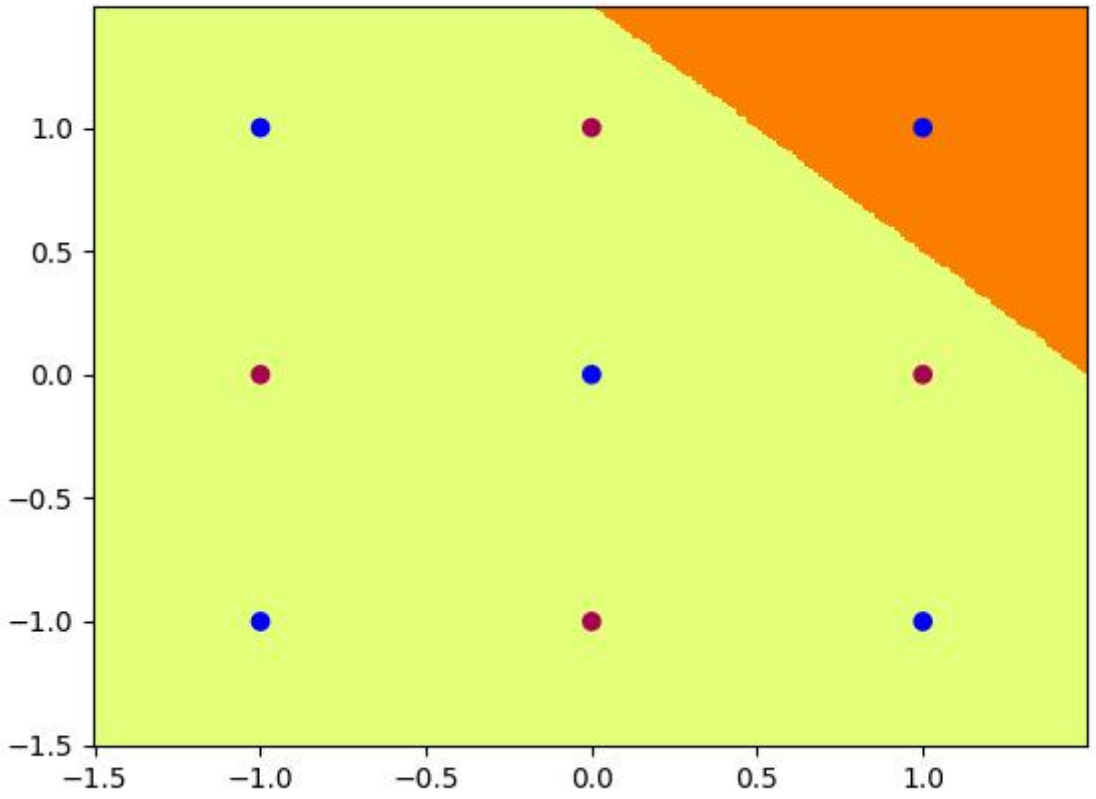
We notice some interesting properties. The sigmoid function outputs values between $0$ and $1$ and has a median value of $0.5$,similar to the Heaviside function. Not only that, but for very large positive values of $x$, typically when $x > 5$, we notice the sigmoid function approach $1$. Similary, for very large negative values of $x$, typically when $x < -5$, we notice the sigmoid function approach $0$.

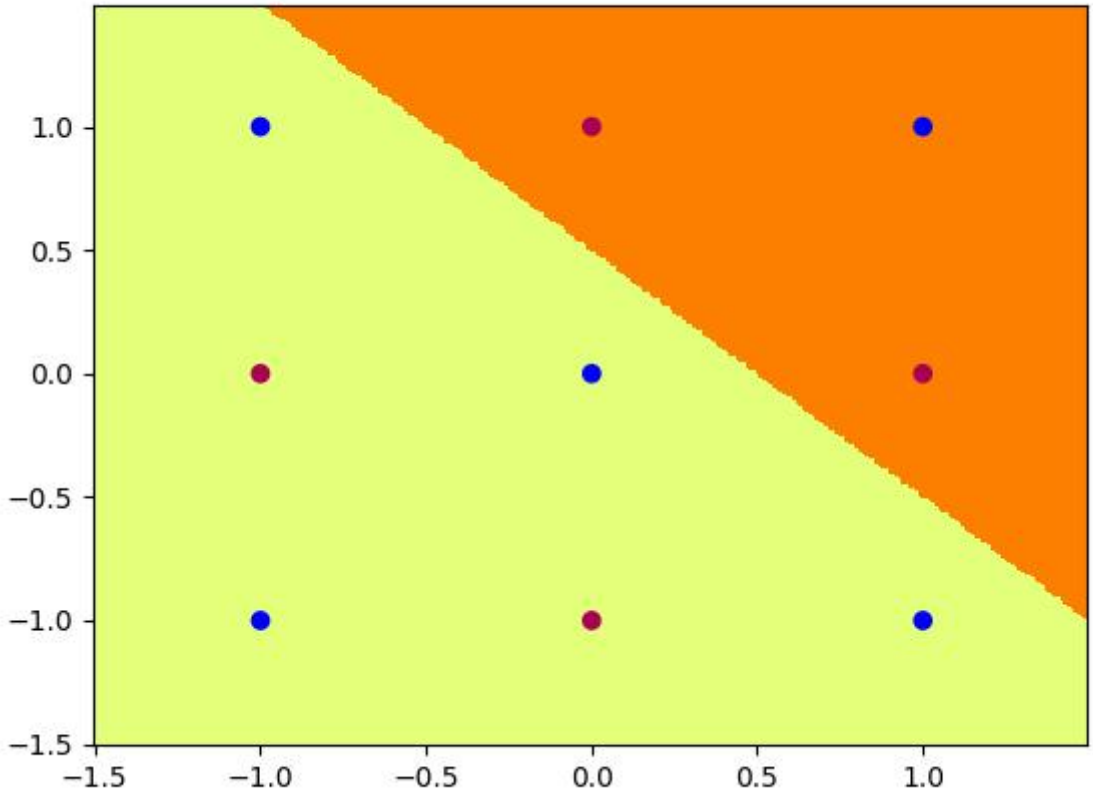
We can thus conclude we need a scaling factor $k$, or even more specifically, given the property above, a scaling factor of $s = 5k$. After a short amount of trial and error, the value $k = 100$, or $s = 500$, we achieved a configuration of scaled weights and biases that give `100%` accuracy using both the Heaviside activation function and the sigmoid activation function.

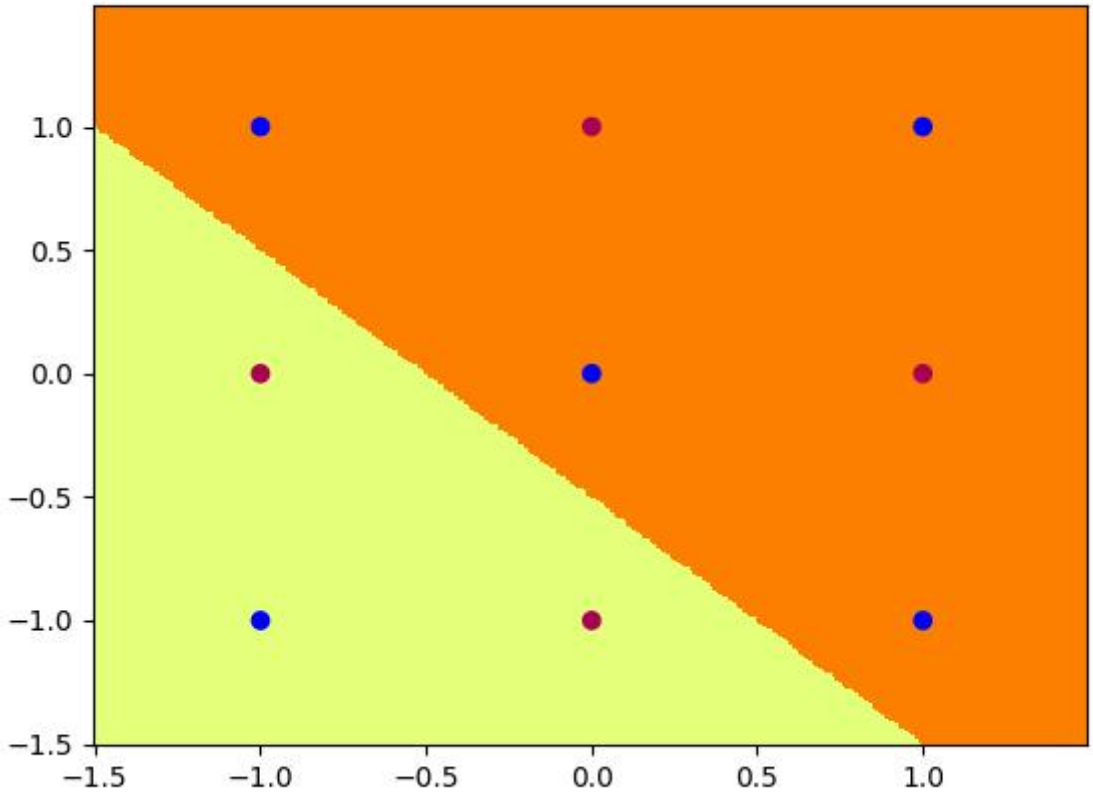
The outputs for both the Heaviside and sigmoid tests are the same, printed below:

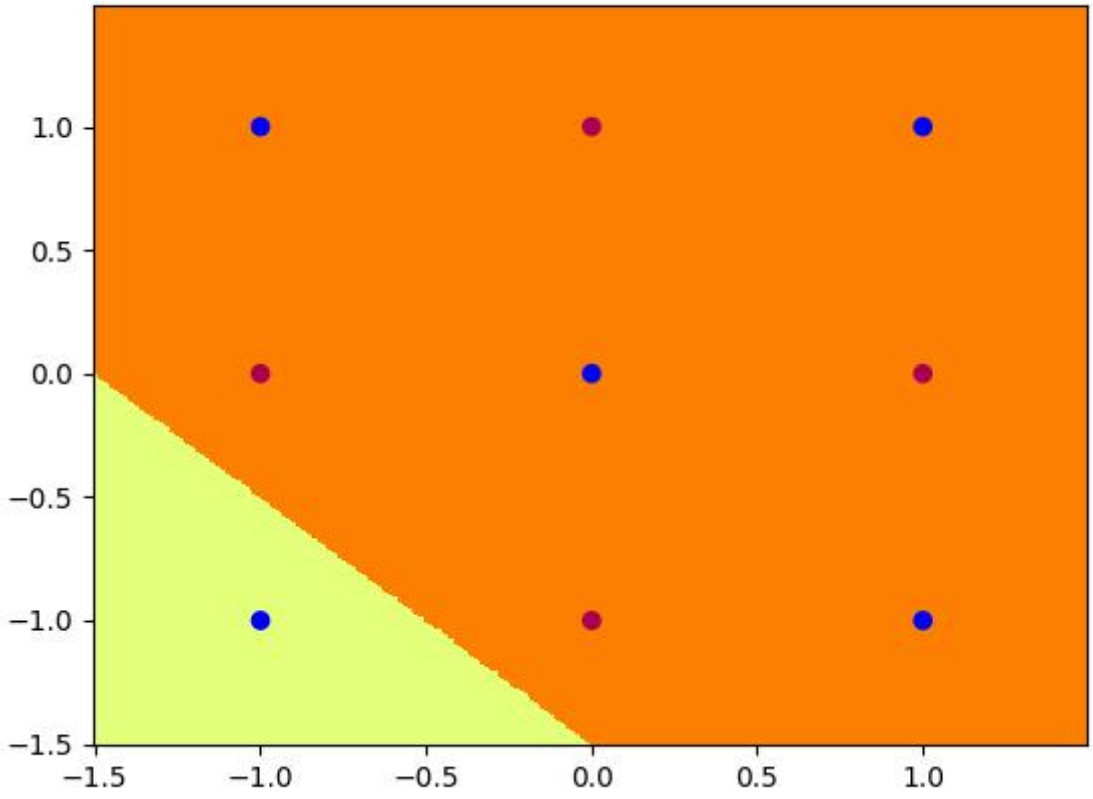
```
Initial Weights:
tensor([[500., 500.],
        [500., 500.],
        [500., 500.],
        [500., 500.]])
tensor([-750., -250.,  250.,  750.])
tensor([[ 500., -500.,  500., -500.]])
tensor([0.])
Initial Accuracy:  100.0
```

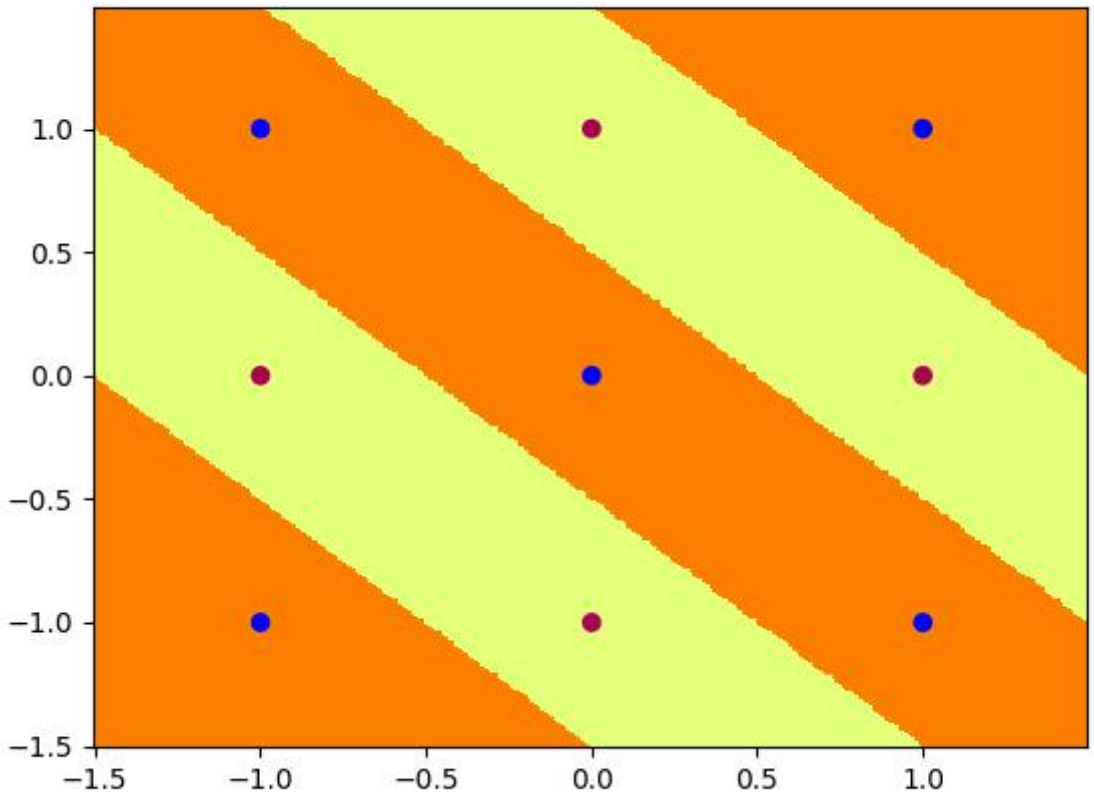
Our output also contains the following graphs, clearly illustrating how our network works:




1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0


1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0


1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0




## Part 3: Hidden Unit Dynamics for Recurrent Networks

**1.**

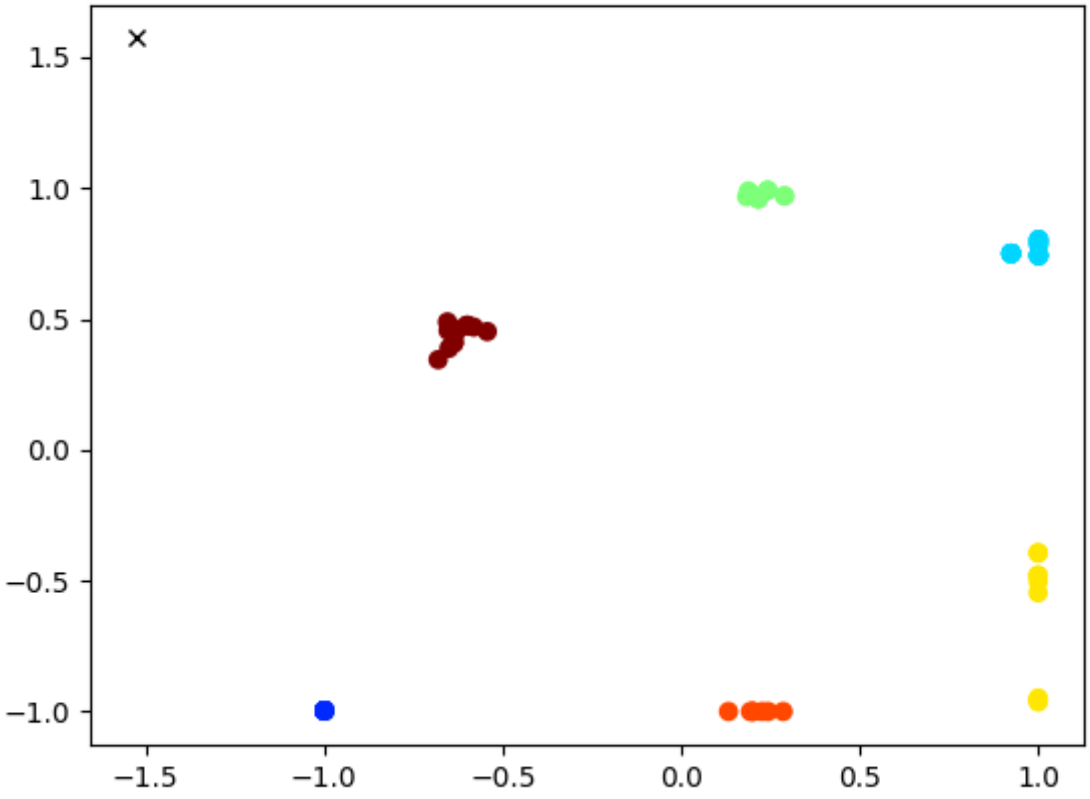
I executed the following command to train a simple recurrent network on a Reber grammar prediction task:

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

I then executed the following command to plot the hidden unit activations at epoch `50000`:

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

It produced the following graph as output, containing dots organised in clusters of discernable color:




It is safe to assume that points on the graph that are clustered together also belong to the same state in the Reber grammar. We thus observe the correlation between clusters of points, their associated color, and their Reber grammar state. More specifically, clusters are colored according to the following colormap, where blue represents lower activation values and red corresponds to higher activation values.

Here are the hidden unit activation values between the last 10000 epochs, or in other words, the last two epoch stages:

```
-----
state = 01236534
symbol= BTXXVPSE
label = 01335426
true probabilities:
     B    T    S    X    P    V    E
1 [ 0.   0.5  0.   0.   0.5  0.   0. ]
2 [ 0.   0.   0.5  0.5  0.   0.   0. ]
```

```
3 [ 0.   0.   0.5  0.5  0.   0.   0. ]
6 [ 0.   0.5  0.   0.   0.   0.5  0. ]
5 [ 0.   0.   0.   0.   0.5  0.5  0. ]
3 [ 0.   0.   0.5  0.5  0.   0.   0. ]
4 [ 0.  0.  0.  0.  0.  0.  1.]
hidden activations and output probabilities [BTSXPVE]:
1 [-1. -1.] [ 0.    0.43  0.    0.    0.53  0.04  0.  ]
2 [ 0.92  0.74] [ 0.    0.    0.45  0.54  0.    0.    0.  ]
3 [ 0.26  0.96] [ 0.    0.    0.61  0.36  0.    0.03  0.  ]
6 [-0.6  0.5] [ 0.    0.37  0.02  0.01  0.01  0.59  0.  ]
5 [ 0.14 -1.  ] [ 0.    0.03  0.    0.    0.48  0.49  0.  ]
3 [ 0.18  0.99] [ 0.    0.    0.62  0.33  0.    0.04  0.  ]
4 [ 1.   -0.53] [ 0.    0.    0.    0.    0.    0.    0.99]
epoch: 49000
error: 0.0024
-----
state = 0166534
symbol= BPTVPSE
label = 0415426
true probabilities:
     B    T    S    X    P    V    E
1 [ 0.   0.5  0.   0.   0.5  0.   0. ]
6 [ 0.   0.5  0.   0.   0.   0.5  0. ]
6 [ 0.   0.5  0.   0.   0.   0.5  0. ]
5 [ 0.   0.   0.   0.   0.5  0.5  0. ]
3 [ 0.   0.   0.5  0.5  0.   0.   0. ]
4 [ 0.  0.  0.  0.  0.  0.  1.]
hidden activations and output probabilities [BTSXPVE]:
1 [-1. -1.] [ 0.    0.46  0.    0.    0.5   0.04  0.  ]
6 [-0.64  0.43] [ 0.    0.43  0.01  0.01  0.01  0.55  0.  ]
6 [-0.63  0.46] [ 0.    0.41  0.02  0.01  0.01  0.56  0.  ]
5 [ 0.2 -1. ] [ 0.    0.02  0.    0.    0.41  0.56  0.  ]
3 [ 0.24  0.99] [ 0.    0.    0.6   0.37  0.    0.03  0.  ]
4 [ 1.   -0.48] [ 0.    0.    0.    0.01  0.    0.01  0.98]
epoch: 50000
error: 0.0016
```

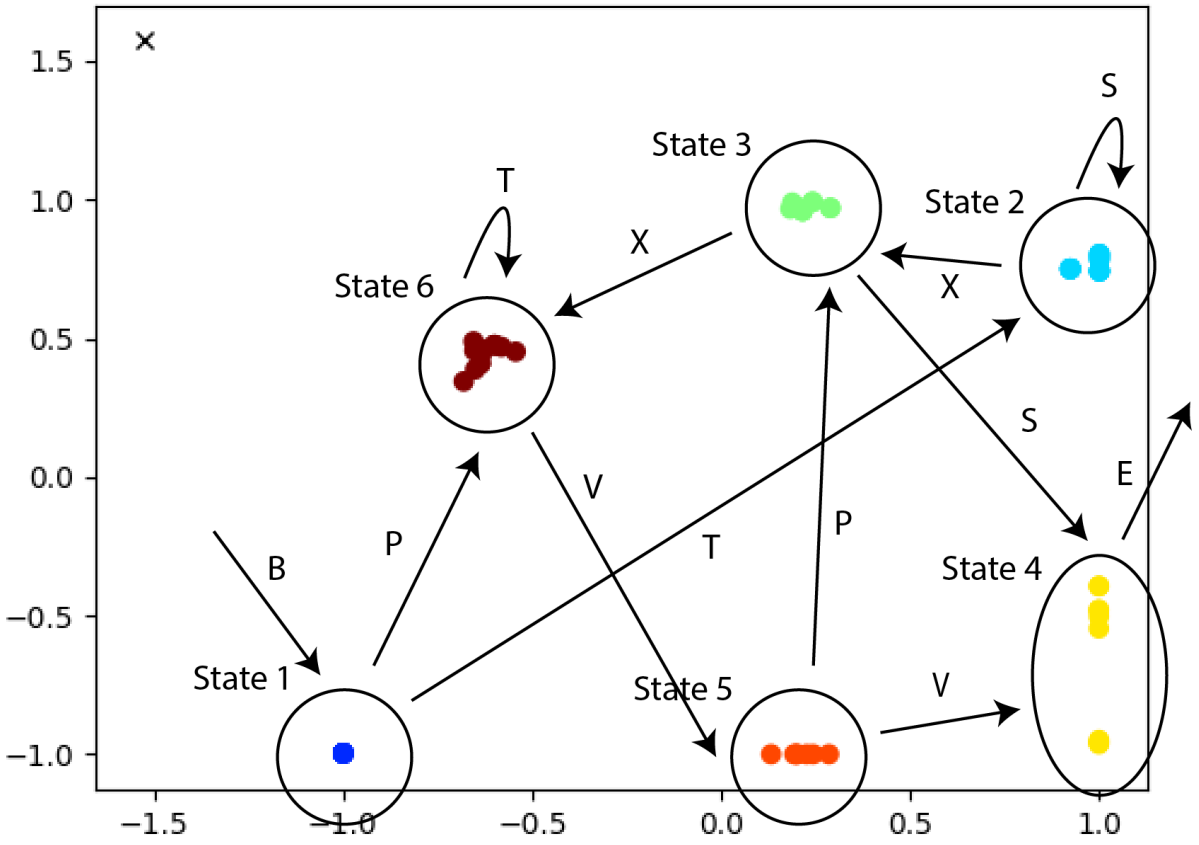
Given this data, we can correlate the hidden unit activation values with the values in the graph. For example, we see the blue dots at point $(-1, -1)$, which corresponds to the hidden activation values for State 1, which are `[-1., -1.]`. We thus know that the blue dots correspond to State

1. We see the same for the brown dots at approximately $(-0.6, 0.4)$, which correspond to the hidden activation values for State 6 such as `[-0.64, 0.43]` or `[-0.63, 0.46]`. We thus know that the brown dots correspond to State 6, and so on and so forth.

We thus get the following associations:

- State 1 is associated with the color 'Blue'
- State 6 is associated with the color 'Brown'
- State 5 is associated with the color 'Orange'
- State 3 is associated with the color 'Green'
- State 4 is associated with the color 'Yellow'
- State 2 is associated with the color 'Light Blue'

We can now annotate the graph above to redraw the Reber Grammar finite state machine, this time on the activation values graph. This is done as follows:




**2.**

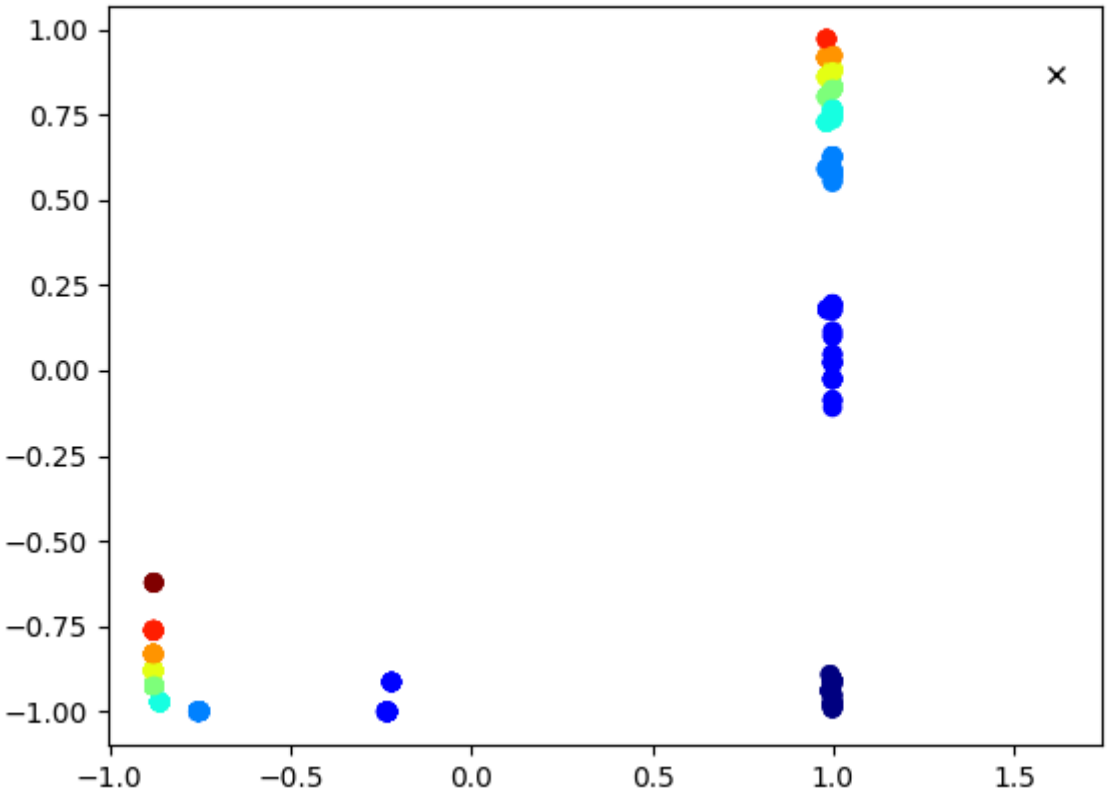
I executed the following command to train a simple recurrent network on the $a^n b^n$ language prediction task.

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

I then executed the following command to plot the hidden unit activations at epoch `100000`.

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

It produced the following graph as output:




The plot shows distinct clusters of activation values for different stages of the sequence processing. Since the network encodes the state of the sequence using its hidden units, we know that the intensity of the activation values, indicated by the color, represents the number of $a$'s seen or number of $b$'s expected to be seen.

As the recurrent network progresses along the sequence, the number of $a$'s seen and $b$'s expected to be seen grows, as seen by the vertical transition from dark blue (low intensity) to bright red or dark brown (high intensity). This is a visual indicator of the fact that our network is sucessfully working, and that the language prediction task is being properly learned.

We also observe the clustering phenomenon, where the activation values for $a$ are clustered around the point $(-1, -1)$, while the activation values for $b$ are clustered around the point $(1, 1)$. We know this because we can observe the hidden activation values for both $a$'s and $b$'s from the output of the last two training epoch stages:

```
-----
color = 012101234543210101012343210
symbol= AABBAAAAABBBBBABABAAAABBBBA
label = 001100000111110101000011110
hidden activations and output probabilities:
A [-0.23 -0.92] [ 0.86  0.14]
B [-0.76 -1.  ] [ 0.86  0.14]
B [ 0.98  0.18] [ 0.  1.]
A [ 1.   -0.91] [ 0.96  0.04]
A [-0.25 -1.  ] [ 0.92  0.08]
A [-0.76 -1.  ] [ 0.86  0.14]
A [-0.86 -0.97] [ 0.81  0.19]
A [-0.88 -0.92] [ 0.74  0.26]
B [-0.88 -0.88] [ 0.66  0.34]
B [ 0.98  0.81] [ 0.  1.]
B [ 1.    0.74] [ 0.  1.]
B [ 1.    0.56] [ 0.  1.]
B [ 1.   -0.07] [ 0.03  0.97]
A [ 1.   -0.99] [ 0.98  0.02]
B [-0.25 -1.  ] [ 0.92  0.08]
A [ 0.99 -0.93] [ 0.97  0.03]
B [-0.25 -1.  ] [ 0.92  0.08]
A [ 0.99 -0.93] [ 0.97  0.03]
A [-0.25 -1.  ] [ 0.92  0.08]
A [-0.76 -1.  ] [ 0.86  0.14]
A [-0.87 -0.97] [ 0.81  0.19]
B [-0.88 -0.92] [ 0.74  0.26]
B [ 0.98  0.73] [ 0.  1.]
B [ 1.    0.58] [ 0.  1.]
B [ 1. -0.] [ 0.02  0.98]
A [ 1.   -0.98] [ 0.98  0.02]
epoch: 99000
error: 0.0040
-----
color = 012343210123210123456787654321010123432l0
```

```
symbol= AAAABBBBAAABBBAAAAAAAABBBBBBBBBABAAAABBBBA
label = 00001111000111000000001111111101000011110
hidden activations and output probabilities:
A [-0.22 -0.91] [ 0.86  0.14]
A [-0.75 -1.  ] [ 0.87  0.13]
A [-0.86 -0.97] [ 0.83  0.17]
B [-0.88 -0.92] [ 0.76  0.24]
B [ 0.98  0.73] [ 0.  1.]
B [ 1.    0.57] [ 0.  1.]
B [ 1.   -0.04] [ 0.02  0.98]
A [ 1.   -0.99] [ 0.98  0.02]
A [-0.23 -1.  ] [ 0.93  0.07]
A [-0.76 -1.  ] [ 0.87  0.13]
B [-0.86 -0.97] [ 0.83  0.17]
B [ 0.98  0.59] [ 0.  1.]
B [ 1.    0.12] [ 0.01  0.99]
A [ 1.   -0.95] [ 0.97  0.03]
A [-0.23 -1.  ] [ 0.93  0.07]
A [-0.76 -1.  ] [ 0.87  0.13]
A [-0.86 -0.97] [ 0.83  0.17]
A [-0.88 -0.92] [ 0.76  0.24]
A [-0.88 -0.88] [ 0.69  0.31]
A [-0.88 -0.83] [ 0.6  0.4]
A [-0.88 -0.76] [ 0.46  0.54]
B [-0.88 -0.62] [ 0.22  0.78]
B [ 0.98  0.97] [ 0.  1.]
B [ 1.    0.92] [ 0.  1.]
B [ 1.    0.88] [ 0.  1.]
B [ 1.    0.82] [ 0.  1.]
B [ 1.    0.75] [ 0.  1.]
B [ 1.    0.58] [ 0.  1.]
B [ 1.    0.02] [ 0.01  0.99]
A [ 1.   -0.98] [ 0.98  0.02]
B [-0.23 -1.  ] [ 0.93  0.07]
A [ 0.99 -0.94] [ 0.97  0.03]
A [-0.24 -1.  ] [ 0.93  0.07]
A [-0.76 -1.  ] [ 0.87  0.13]
A [-0.86 -0.97] [ 0.83  0.17]
B [-0.88 -0.92] [ 0.76  0.24]
B [ 0.98  0.73] [ 0.  1.]
```

```
B [ 1.    0.57] [ 0.  1.]
B [ 1.   -0.02] [ 0.02  0.98]
A [ 1.   -0.98] [ 0.98  0.02]
epoch: 100000
error: 0.0150
```

The clear clustering and separation of activation values, as well as the low error rate, are also indicators that our network as successfully learned to predict the sequence.

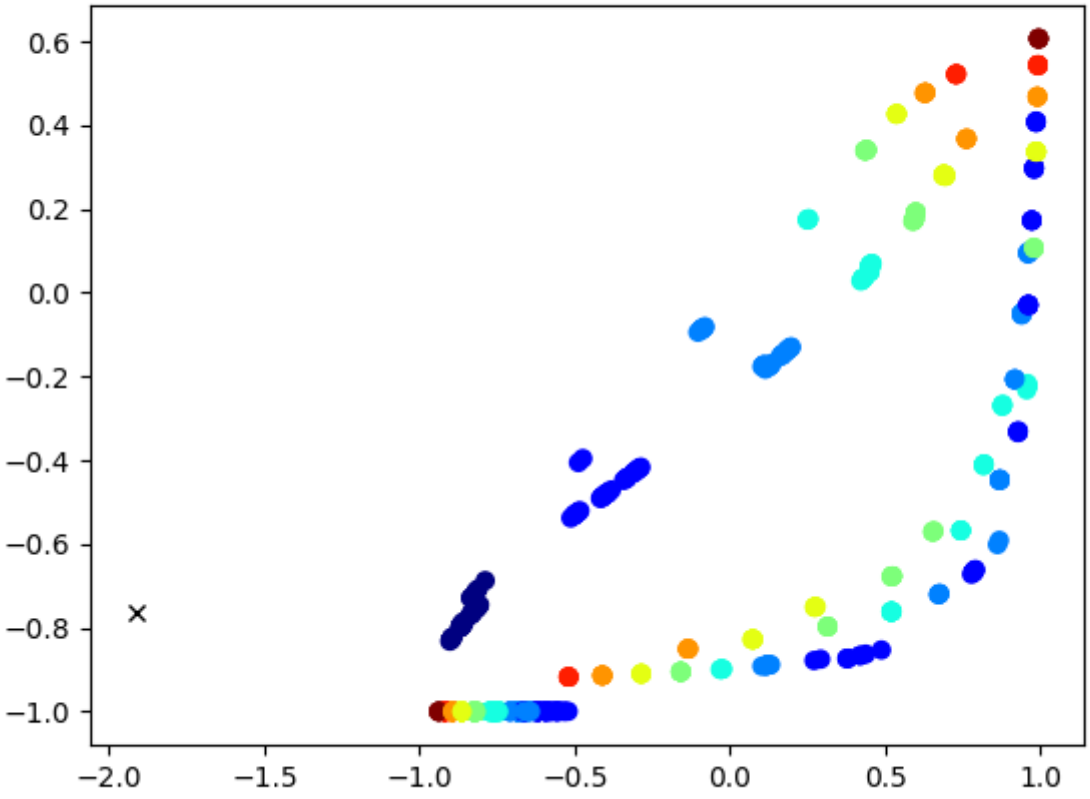
**3.**
I executed the following command to train a simple recurrent network on the $a^n b^n c^n$ language prediction task:

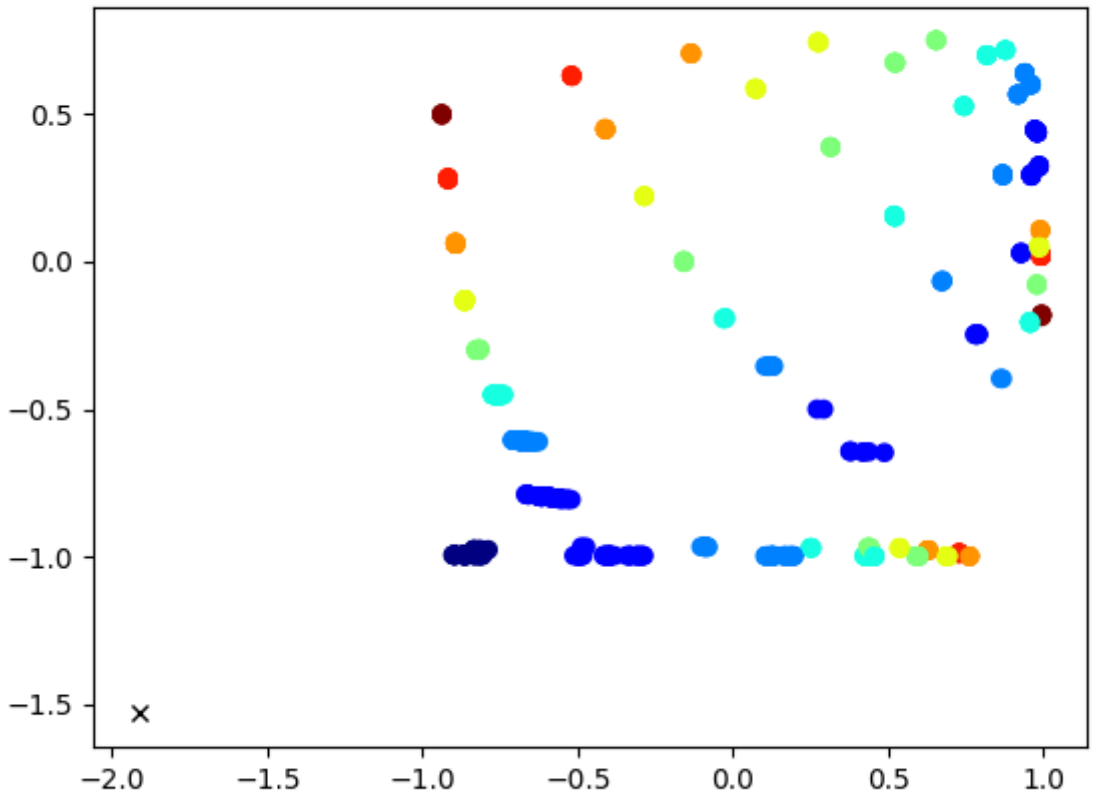
```
python3 seq_train.py --lang anbncn
```

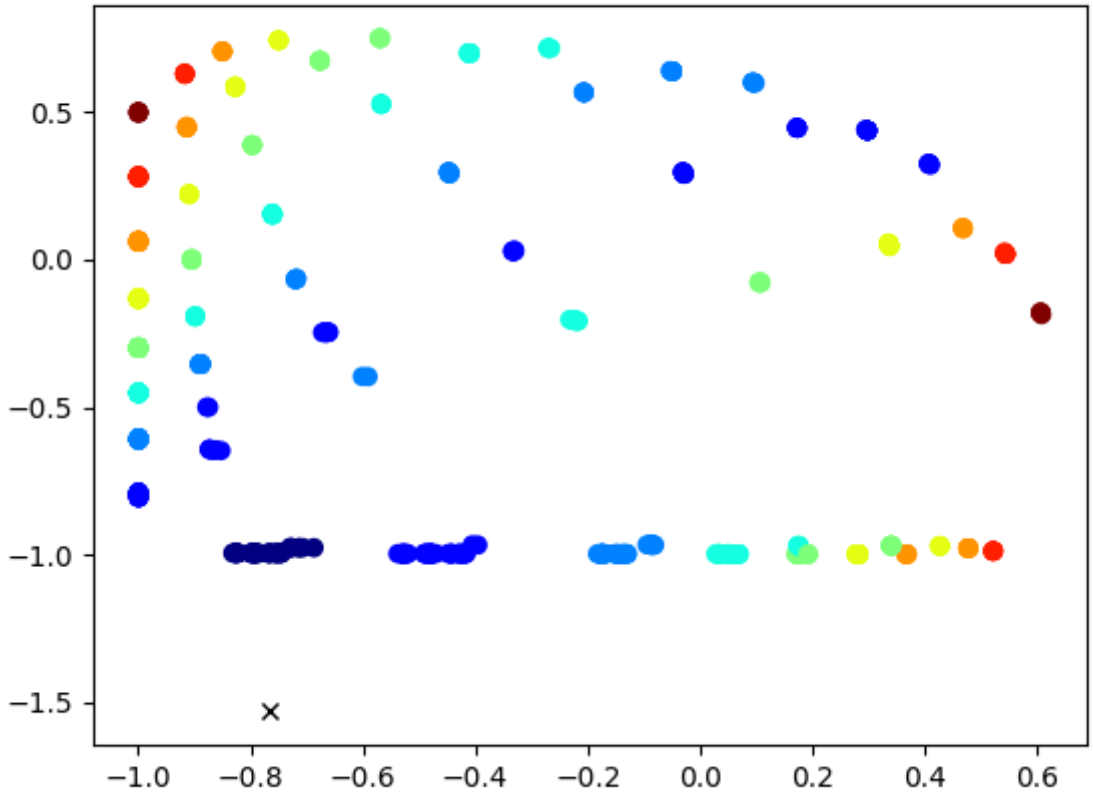
I then executed the following command to plot the hidden unit activations at epoch `200000`:

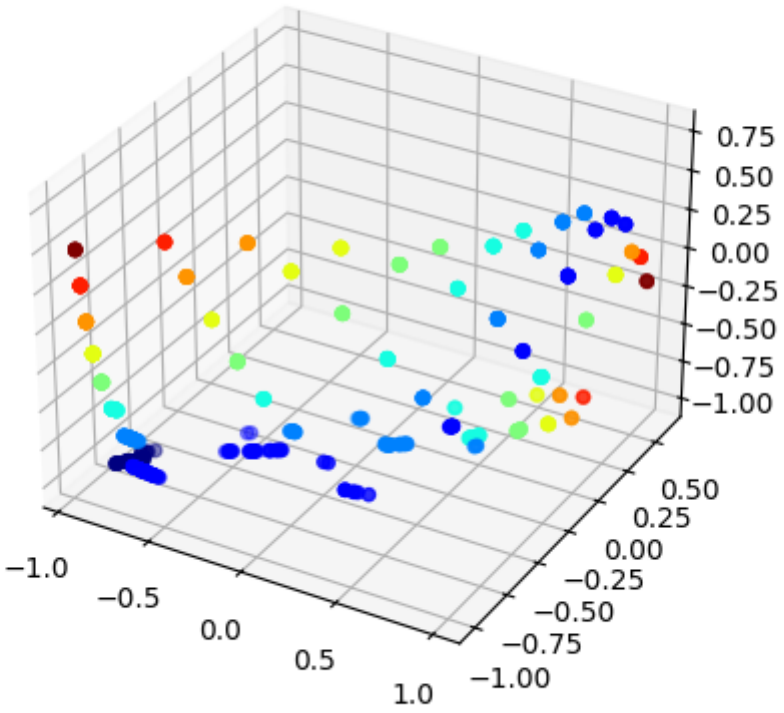
```
python3 seq_plot.py --lang anbncn --epoch 200
```

It produced the following graphs as output, and will be numbered in the order they are presented:


0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0


0.5
0.0
−0.5
−1.0
−1.5
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0


0.5
0.0
−0.5
−1.0
−1.5
−1.0
−0.8
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6




Since there are three characters: $a$, $b$ and $c$, the hidden unit activations plot is now three-dimensional. Once again, the plots show distinct clusters of activation values for different stages of the sequence processing. Since the network encodes the state of the sequence using its hidden units, we know that the intensity of the activation values, indicated by the color, represents the number of $a$'s seen, number of $b$'s expected to be seen or number of $c$'s expected to be seen.

We see in the graph that the clustering occurs around certain curves going through the three-dimensional space. These curves have observable transitions from low intensity (blue) values to high intensity (red) values, and each cluster is tightly packed and well-separated, indicating the network maintains stable states for various input patterns, and thus, successfully learned the prediction task.

Lets analyse the hidden unit activations for a better idea of the which curves correspond to which of the three characters. The following is output for the last two training epoch stages:

```
_____
color =
012345678765432187654321012321321012345678765432187654321012121012345432154321
0
```

```
symbol=
AAAAAAAABBBBBBBBCCCCCCCCAAABBBCCCAAAAAAAABBBBBBBBCCCCCCCCAABBCCAAAAABBBBBCCC
CCA
label =
000000001111111122222222000111222000000001111111222222220011220000011111222
220
hidden activations and output probabilities:
A [-0.56 -1.   -0.8 ] [ 0.84  0.16  0.  ]
A [-0.63 -1.   -0.61] [ 0.79  0.21  0.  ]
A [-0.73 -1.   -0.44] [ 0.8  0.2  0. ]
A [-0.8  -1.   -0.29] [ 0.78  0.22  0.  ]
A [-0.85 -1.   -0.12] [ 0.71  0.29  0.  ]
A [-0.89 -1.    0.08] [ 0.56  0.44  0.  ]
A [-0.92 -1.    0.31] [ 0.36  0.64  0.  ]
B [-0.94 -1.    0.52] [ 0.19  0.81  0.  ]
B [-0.52 -0.92  0.65] [ 0.  1.  0.]
B [-0.14 -0.85  0.72] [ 0.  1.  0.]
B [ 0.27 -0.75  0.75] [ 0.  1.  0.]
B [ 0.65 -0.58  0.75] [ 0.  1.  0.]
B [ 0.88 -0.28  0.72] [ 0.  1.  0.]
B [ 0.96  0.08  0.61] [ 0.  1.  0.]
B [ 0.99  0.4   0.34] [ 0.    0.97  0.03]
C [ 1.    0.6  -0.16] [ 0.  0.  1.]
C [ 0.72  0.52 -0.99] [ 0.  0.  1.]
C [ 0.76  0.36 -1.  ] [ 0.  0.  1.]
C [ 0.69  0.28 -1.  ] [ 0.  0.  1.]
C [ 0.6   0.17 -1.  ] [ 0.  0.  1.]
C [ 0.44  0.04 -1.  ] [ 0.  0.  1.]
C [ 0.15 -0.16 -1.  ] [ 0.  0.  1.]
C [-0.36 -0.46 -1.  ] [ 0.  0.  1.]
A [-0.84 -0.78 -0.99] [ 0.99  0.    0.  ]
A [-0.57 -1.   -0.8 ] [ 0.85  0.15  0.  ]
A [-0.64 -1.   -0.61] [ 0.8  0.2  0. ]
B [-0.73 -1.   -0.45] [ 0.81  0.19  0.  ]
B [ 0.16 -0.89 -0.35] [ 0.  1.  0.]
B [ 0.8  -0.65 -0.24] [ 0.    0.93  0.07]
C [ 0.96 -0.21 -0.2 ] [ 0.    0.01  0.99]
C [-0.07 -0.07 -0.97] [ 0.  0.  1.]
C [-0.46 -0.51 -1.  ] [ 0.01  0.    0.99]
A [-0.88 -0.81 -0.99] [ 1.  0.  0.]
```

```
A [-0.62 -1.   -0.79] [ 0.9  0.1  0. ]
A [-0.67 -1.   -0.6 ] [ 0.84  0.16  0.  ]
A [-0.75 -1.   -0.45] [ 0.83  0.17  0.  ]
A [-0.81 -1.   -0.29] [ 0.79  0.21  0.  ]
A [-0.86 -1.   -0.12] [ 0.71  0.29  0.  ]
A [-0.89 -1.    0.08] [ 0.56  0.44  0.  ]
A [-0.91 -1.    0.3 ] [ 0.36  0.64  0.  ]
B [-0.94 -1.    0.52] [ 0.19  0.81  0.  ]
B [-0.52 -0.92  0.64] [ 0.  1.  0.]
B [-0.13 -0.85  0.71] [ 0.  1.  0.]
B [ 0.28 -0.75  0.75] [ 0.  1.  0.]
B [ 0.65 -0.57  0.75] [ 0.  1.  0.]
B [ 0.88 -0.28  0.72] [ 0.  1.  0.]
B [ 0.96  0.08  0.6 ] [ 0.  1.  0.]
B [ 0.99  0.4   0.33] [ 0.    0.97  0.03]
C [ 1.    0.6  -0.16] [ 0.  0.  1.]
C [ 0.72  0.52 -0.99] [ 0.  0.  1.]
C [ 0.76  0.36 -1.  ] [ 0.  0.  1.]
C [ 0.69  0.28 -1.  ] [ 0.  0.  1.]
C [ 0.6   0.18 -1.  ] [ 0.  0.  1.]
C [ 0.44  0.04 -1.  ] [ 0.  0.  1.]
C [ 0.15 -0.16 -1.  ] [ 0.  0.  1.]
C [-0.35 -0.46 -1.  ] [ 0.  0.  1.]
A [-0.84 -0.77 -0.99] [ 0.99  0.    0.  ]
A [-0.57 -1.   -0.8 ] [ 0.84  0.16  0.  ]
B [-0.64 -1.   -0.61] [ 0.8  0.2  0. ]
B [ 0.33 -0.87 -0.5 ] [ 0.    0.96  0.04]
C [ 0.88 -0.58 -0.39] [ 0.    0.02  0.98]
C [-0.45 -0.38 -0.97] [ 0.  0.  1.]
A [-0.84 -0.78 -0.99] [ 0.99  0.    0.  ]
A [-0.57 -1.   -0.8 ] [ 0.85  0.15  0.  ]
A [-0.64 -1.   -0.61] [ 0.8  0.2  0. ]
A [-0.73 -1.   -0.45] [ 0.81  0.19  0.  ]
A [-0.81 -1.   -0.29] [ 0.79  0.21  0.  ]
B [-0.85 -1.   -0.12] [ 0.71  0.29  0.  ]
B [-0.14 -0.9   0.02] [ 0.  1.  0.]
B [ 0.54 -0.76  0.17] [ 0.  1.  0.]
B [ 0.87 -0.44  0.31] [ 0.  1.  0.]
B [ 0.96 -0.03  0.31] [ 0.  1.  0.]
C [ 0.99  0.33  0.07] [ 0.    0.02  0.98]
```

```
C [ 0.43  0.34 -0.97] [ 0.  0.  1.]
C [ 0.45  0.06 -1.  ] [ 0.  0.  1.]
C [ 0.19 -0.14 -1.  ] [ 0.  0.  1.]
C [-0.3  -0.43 -1.  ] [ 0.  0.  1.]
A [-0.81 -0.75 -0.99] [ 0.99  0.    0.01]
epoch: 199000
error: 0.0111
-----
color = 012321321012345676543217654321012345432154321011012121O
symbol= AAABBBCCCAAAAAAABBBBBBBCCCCCCCAAAAABBBBBCCCCCABCAABBCCA
label = 000111222000000011111112222222000001111122222012001122O
hidden activations and output probabilities:
A [-0.59 -1.   -0.8 ] [ 0.87  0.13  0.  ]
A [-0.67 -1.   -0.61] [ 0.84  0.16  0.  ]
B [-0.76 -1.   -0.45] [ 0.84  0.16  0.  ]
B [ 0.12 -0.89 -0.35] [ 0.  1.  0.]
B [ 0.79 -0.67 -0.24] [ 0.    0.94  0.06]
C [ 0.96 -0.22 -0.2 ] [ 0.    0.02  0.98]
C [-0.09 -0.09 -0.97] [ 0.  0.  1.]
C [-0.5  -0.53 -1.  ] [ 0.02  0.    0.98]
A [-0.9  -0.83 -0.99] [ 1.  0.  0.]
A [-0.66 -1.   -0.79] [ 0.92  0.08  0.  ]
A [-0.71 -1.   -0.6 ] [ 0.88  0.12  0.  ]
A [-0.77 -1.   -0.45] [ 0.86  0.14  0.  ]
A [-0.83 -1.   -0.3 ] [ 0.83  0.17  0.  ]
A [-0.86 -1.   -0.13] [ 0.75  0.25  0.  ]
A [-0.89 -1.    0.06] [ 0.6  0.4  0. ]
B [-0.92 -1.    0.28] [ 0.4  0.6  0. ]
B [-0.41 -0.91  0.45] [ 0.01  0.99  0.  ]
B [ 0.08 -0.83  0.59] [ 0.  1.  0.]
B [ 0.52 -0.68  0.67] [ 0.  1.  0.]
B [ 0.82 -0.41  0.7 ] [ 0.  1.  0.]
B [ 0.94 -0.05  0.64] [ 0.  1.  0.]
B [ 0.98  0.3   0.44] [ 0.  1.  0.]
C [ 0.99  0.54  0.02] [ 0.  0.  1.]
C [ 0.63  0.48 -0.98] [ 0.  0.  1.]
C [ 0.69  0.28 -1.  ] [ 0.  0.  1.]
C [ 0.59  0.17 -1.  ] [ 0.  0.  1.]
C [ 0.43  0.03 -1.  ] [ 0.  0.  1.]
C [ 0.13 -0.17 -1.  ] [ 0.  0.  1.]
```

```
C [-0.4  -0.48 -1.  ] [ 0.  0.  1.]
A [-0.86 -0.79 -0.99] [ 0.99  0.    0.  ]
A [-0.62 -1.   -0.8 ] [ 0.89  0.11  0.  ]
A [-0.68 -1.   -0.61] [ 0.86  0.14  0.  ]
A [-0.76 -1.   -0.45] [ 0.85  0.15  0.  ]
A [-0.82 -1.   -0.3 ] [ 0.82  0.18  0.  ]
B [-0.86 -1.   -0.13] [ 0.75  0.25  0.  ]
B [-0.16 -0.9  -0.  ] [ 0.01  0.99  0.  ]
B [ 0.52 -0.76  0.15] [ 0.  1.  0.]
B [ 0.87 -0.45  0.29] [ 0.  1.  0.]
B [ 0.96 -0.03  0.29] [ 0.  1.  0.]
C [ 0.99  0.34  0.05] [ 0.    0.01  0.99]
C [ 0.44  0.34 -0.97] [ 0.  0.  1.]
C [ 0.46  0.07 -1.  ] [ 0.  0.  1.]
C [ 0.2  -0.13 -1.  ] [ 0.  0.  1.]
C [-0.29 -0.42 -1.  ] [ 0.  0.  1.]
A [-0.81 -0.75 -0.99] [ 0.99  0.    0.01]
B [-0.55 -1.   -0.8 ] [ 0.83  0.17  0.  ]
C [ 0.48 -0.85 -0.65] [ 0.    0.03  0.97]
A [-0.79 -0.69 -0.98] [ 0.98  0.    0.02]
A [-0.49 -1.   -0.81] [ 0.76  0.24  0.  ]
B [-0.61 -1.   -0.61] [ 0.76  0.24  0.  ]
B [ 0.34 -0.87 -0.5 ] [ 0.    0.95  0.05]
C [ 0.88 -0.57 -0.39] [ 0.    0.01  0.99]
C [-0.44 -0.37 -0.97] [ 0.  0.  1.]
A [-0.84 -0.77 -0.99] [ 0.99  0.    0.  ]
epoch: 200000
error: 0.0033
```

From the hidden unit activations from the above output, we observe distinct clustering patterns for each character in the sequence, demonstrating how the network processes the prediction task. The activations for the character $a$ are generally clustered around negative values, indicating the 'counting up' of $a$'s. For the character $b$, activations transition from negative to positive values, reflecting the network's internal state decrementing the count ('counting down') of $b$'s after incrementing for $a$'s. The activations for the character $c$ are also around negative values but are distinctly separate from those of $a$, indicating the final stages of sequence processing.

The tight clustering of $c$ activations suggests the completion of the sequence. The network uses the transition in hidden unit activations to accurately predict the last $b$ in the sequence, ensuring it can correctly identify the final $b$ by keeping track of how many $b$'s have been processed

relative to $a$'s. After predicting all $c$'s, the network returns to processing $a$'s for the next sequence, as indicated by the activation patterns resetting to the initial state for $a$ clustering.

These patterns collectively demonstrate that the network has successfully learned to process the task by correctly encoding the sequence structure and maintaining accurate counts of each character.

**4.**
I executed the following command to train an LSTM recurrent neural network to learn the Embedded Reber Grammar:

```
python3 seq_train.py --lang reber --embed True --model lstm --hid 5
```

I decided to use $5$ hidden units instead of $4$ given the fact that the Embedded Reber Grammar is a fairly sophisticated task and may need a higher network capacity to remember and process more long-term dependencies and a higher degree of complexity. Obviously, we do not want to increase the network capacity too much to avoid overfitting, so an increment of $1$ makes sense here.

I also edited the code so that all context units are recorded and printed per epoch stage. This will help us analyse how the LSTM network works.

First, a bit about the theory behind how LSTM's work. Essentially, there are four components to an LSTM network: the input gate $i_t$, the forget gate $f_t$, the cell state $C_t$ and the output gate $o_t$. The input gate controls how much of the new information, known as the 'candidate cell state', is added to the pre-existing cell state. The forget gate how much of the previous cell state is retained The cell state is the internal memory of the LSTM network. The output gate controls how much of the cell state is used to compute the hidden state. The cell state $C_t$ is updated by combining the old cell state $C_{t-1}$ and the candidate cell state $\bar{C}_t$ such that

$$C_t = f_t * C_{t-1} + i_t * \bar{C}_t$$

We can see this in action with our neural network. Here is the output for the last three epoch stages:

```
-----
state =  0 1 2 3 4 5 6 9 18
symbol= BTBTXSETE
label = 010132616
true probabilities:
     B    T    S    X    P    V    E
1 [ 0.   0.5  0.   0.   0.5  0.   0. ]
```

```
2 [ 1.  0.  0.  0.  0.  0.  0.]
3 [ 0.   0.5  0.   0.   0.5  0.   0. ]
4 [ 0.   0.   0.5  0.5  0.   0.   0. ]
5 [ 0.   0.   0.5  0.5  0.   0.   0. ]
6 [ 0.  0.  0.  0.  0.  0.  1.]
9 [ 0.  1.  0.  0.  0.  0.  0.]
18 [ 0.  0.  0.  0.  0.  0.  1.]
hidden activations and output probabilities [BTSXPVE]:
1 [ 0.21  0.72  0.75  0.75  0.27] [ 0.    0.51  0.    0.    0.48  0.    0.
]
2 [ 0.73  0.93 -0.68  0.72  0.04] [ 1.  0.  0.  0.  0.  0.  0.]
3 [ 0.85  0.09  0.75  0.9   0.7 ] [ 0.    0.51  0.    0.    0.48  0.    0.
]
4 [ 0.83  0.77 -0.14 -0.72  0.15] [ 0.    0.    0.42  0.58  0.    0.    0.
]
5 [ 0.15 -0.13 -0.71 -0.82  0.06] [ 0.    0.    0.58  0.41  0.    0.    0.
]
6 [ 0.27 -0.83  0.49 -0.9   0.11] [ 0.  0.  0.  0.  0.  0.  1.]
9 [ 0.71 -0.96  0.89  0.51 -0.6 ] [ 0.    0.98  0.    0.    0.02  0.    0.
]
18 [ 0.5  -0.96  0.83 -0.59 -0.67] [ 0.  0.  0.  0.  0.  0.  1.]
context units:
tensor([[[ 0.2450,  0.9740,  0.9908,  0.9838,  0.5455],
         [ 1.1376,  1.8061, -0.8297,  0.9345,  0.1464],
         [ 1.4986,  1.0795,  0.9732,  1.8112,  1.0979],
         [ 1.3735,  1.0230, -0.1406, -0.9219,  0.2614],
         [ 1.0066, -0.8942, -0.9053, -1.1600,  0.8982],
         [ 0.5139, -1.2286,  0.5547, -1.4956,  0.2753],
         [ 0.9459, -1.9444,  1.5112,  0.5649, -0.7014],
         [ 0.6754, -2.1094,  1.2165, -0.6993, -1.4988]]])
epoch: 48000
error: 0.0013
final: 0.0001
-----
state =  0 1 10 11 16 15 13 16 15 14 17 18
symbol= BPBPVPXVVEPE
label = 040454355646
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. ]
13 [ 0.   0.   0.5  0.5  0.   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 [ 0.22  0.72  0.75  0.75  0.27] [ 0.    0.52  0.    0.    0.48  0.    0.
]
10 [ 0.33  0.7  -0.75  0.52  0.27] [ 1.  0.  0.  0.  0.  0.  0.]
11 [ 0.84  0.35  0.74  0.89  0.73] [ 0.    0.48  0.    0.    0.52  0.    0.
]
16 [ 0.52 -0.32 -0.73  0.64  0.53] [ 0.    0.42  0.    0.    0.    0.57  0.
]
15 [-0.68 -0.71  0.17  0.03  0.98] [ 0.    0.    0.    0.    0.44  0.55  0.
]
13 [ 0.23 -0.26 -0.7  -0.73  0.06] [ 0.    0.    0.59  0.4   0.    0.01  0.
]
16 [ 0.01 -0.54 -0.57  0.58  0.09] [ 0.    0.4   0.01  0.    0.    0.59  0.
]
15 [-0.7  -0.72  0.19  0.02  0.97] [ 0.    0.    0.    0.    0.47  0.52  0.
]
14 [-0.52 -0.9   0.79 -0.73  0.67] [ 0.  0.  0.  0.  0.  0.  1.]
17 [ 0.72 -0.97  0.95 -0.18  0.78] [ 0.    0.01  0.    0.    0.98  0.    0.
]
18 [ 0.21 -0.87  0.72 -0.76  0.03] [ 0.  0.  0.  0.  0.  0.  1.]
context units:
tensor([[[ 0.2583,  0.9736,  0.9907,  0.9834,  0.5511],
         [ 1.1133,  0.8952, -0.9791,  0.6021,  1.3670],
         [ 1.4543,  1.0338,  0.9551,  1.5496,  2.3432],
         [ 1.6038, -0.3329, -0.9358,  0.7686,  3.0076],
         [-0.8627, -0.9071,  0.1700,  0.0357,  3.1299],
         [ 0.5804, -0.2673, -0.8799, -0.9321,  2.7901],
         [ 0.0515, -0.9305, -0.7993,  0.6661,  3.6787],
         [-0.9001, -0.9228,  0.2105,  0.0185,  3.1216],
         [-0.6004, -1.4517,  1.1032, -0.9227,  2.1098],
```

```
         [ 0.9316, -2.1375,  2.0638, -0.1771,  1.0782],
         [ 0.6273, -1.3450,  0.9132, -0.9904,  0.1355]]])
epoch: 49000
error: 0.0013
final: 0.0001
-----
state =  0 1 10 11 12 12 12 12 13 14 17 18
symbol= BPBTSSSXSEPE
label = 040122232646
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. ]
12 [ 0.   0.   0.5  0.5  0.   0.   0. ]
12 [ 0.   0.   0.5  0.5  0.   0.   0. ]
12 [ 0.   0.   0.5  0.5  0.   0.   0. ]
12 [ 0.   0.   0.5  0.5  0.   0.   0. ]
13 [ 0.   0.   0.5  0.5  0.   0.   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 [ 0.21  0.72  0.75  0.75  0.26] [ 0.    0.52  0.    0.    0.48  0.    0.
]
10 [ 0.33  0.72 -0.75  0.57  0.25] [ 1.  0.  0.  0.  0.  0.  0.]
11 [ 0.84  0.35  0.74  0.9   0.73] [ 0.   0.5  0.   0.   0.5  0.   0. ]
12 [ 0.86  0.86 -0.43 -0.67  0.47] [ 0.    0.    0.42  0.57  0.    0.    0.
]
12 [ 0.58  0.74 -0.18 -0.92  0.44] [ 0.    0.    0.43  0.57  0.    0.    0.
]
12 [ 0.51  0.87 -0.24 -0.99  0.26] [ 0.    0.    0.43  0.57  0.    0.    0.
]
12 [ 0.47  0.9  -0.36 -0.99  0.25] [ 0.    0.    0.45  0.55  0.    0.    0.
]
13 [ 0.14 -0.15 -0.71 -0.98  0.03] [ 0.    0.    0.59  0.4   0.    0.    0.
]
14 [ 0.2  -0.78  0.49 -0.98  0.33] [ 0.  0.  0.  0.  0.  0.  1.]
17 [ 0.71 -0.95  0.89  0.16  0.76] [ 0.    0.04  0.    0.    0.96  0.    0.
]
```

```
18 [ 0.29 -0.84  0.36 -0.74  0.03] [ 0.    0.    0.    0.    0.    0.
0.99]
context units:
tensor([[[ 0.2388,  0.9738,  0.9905,  0.9840,  0.5544],
         [ 1.1038,  0.9258, -0.9798,  0.6810,  1.3657],
         [ 1.4402,  1.0361,  0.9577,  1.6207,  2.3414],
         [ 1.5349,  1.3138, -0.4550, -0.8180,  1.6773],
         [ 1.6705,  1.2107, -0.1783, -1.6495,  2.0547],
         [ 1.7358,  1.4959, -0.2470, -2.5711,  1.9128],
         [ 1.8857,  2.0017, -0.3833, -3.4232,  1.9755],
         [ 1.3245, -0.7608, -0.9202, -2.4704,  2.6626],
         [ 0.4075, -1.0759,  0.5592, -2.6159,  2.0061],
         [ 0.9434, -1.8093,  1.5170,  0.1604,  1.0154],
         [ 0.7022, -1.2164,  0.3750, -0.9501,  0.0964]]])
epoch: 50000
error: 0.0013
final: 0.0004
```

We can see how the context units show how the LSTM maintains and updates its memory over time. We see that the cell states tensor is a matrix of vectors, where each vector has five values given the fact that network capacity is five hidden units. We then see one vector for every letter in the symbol being trained on in that particular epoch stage.

For example, in the epoch stage `50000`, we execute our training on the symbol $BPBTSSSXSEPE$. We notice that when the network encounters the first character $B$, the cell state is `[0.2388, 0.9738, 0.9905, 0.9840, 0.5544]`. We then see the network encounter the next character $P$, where the cell state changes, becoming `[1.1038, 0.9258, -0.9798, 0.6810, 1.3657]`. This goes on.

We notice that large positive or negative values indicate strong activation at those cell states, representing strong recognition of a feature or pattern. These larger values incurred by more drastic changes are usually the ones that get carried through the network rather than forgotten. The changes between $B$ and $P$ are not too drastic since they are similar looking characters likely with similar features. However, changes between very different looking characters such as $S$ and $X$ have higher magnitudes, as they should.

This is the gradual evolution of the cell state as newer symbols and characters get processed, with the forget gate retaining a certain amount of old information from previous states, and the input gate introducing a certain amount of new information into the current state.

This is all indicative of the fact that the LSTM network is working properly. We can be confident that our network has learned the task due to the consistently low error rate and the high

correlation between output probabilities and ground truth probabilities. It is fascinating to observe the internal workings of an LSTM network as it evolves and learns the sequential data.