# Project 1 – Deep Q Learning

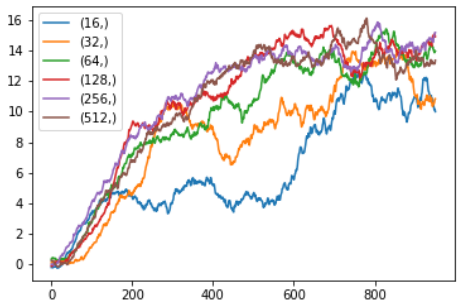
In this project, Deep Q-Learning was used to solve the environment. Improvements to the standard Q-Network included experience replay and fixed targets. An attempt at prioritized experience replay was made but problems with implementation led to poorer results and excessively long run times.

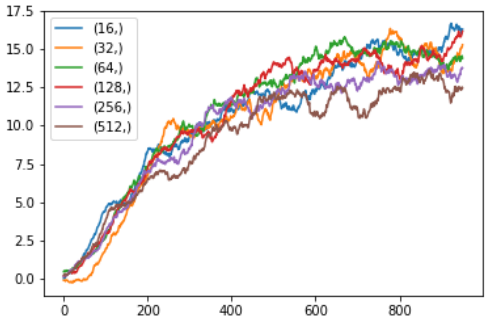
Experiments were performed to test various neural network architectures and hyperparameters are shown in the next section. To implement fixed targets, two identical networks were created at the beginning of each experiment and one was slowly updated towards the other. The following section show the performance of an optimum agent.

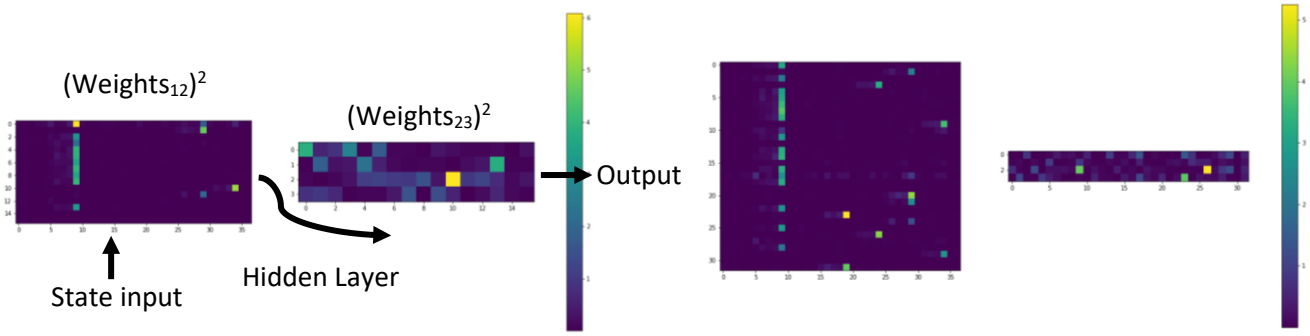
## Experiments:

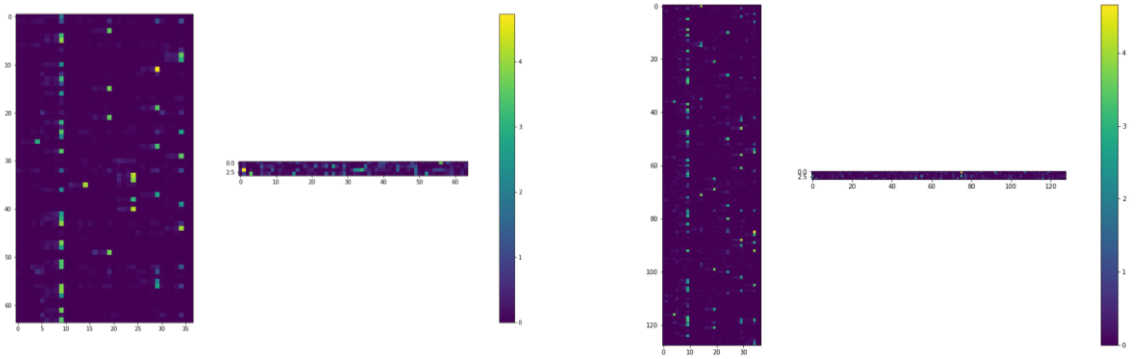
### Single hidden layer Neural Network

These networks had a single layer between input and output. To experiment with drop out, the left figure was performed with the drop out probability, drop_p = 0.5, the right with drop_p =0. The below figures show the weight matrices for 4 of the neural networks after training. Drop out was tested extensively throughout out this study for interest sake.

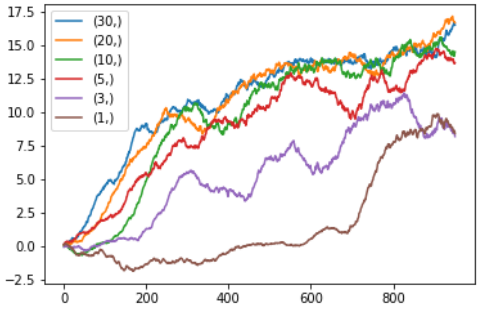


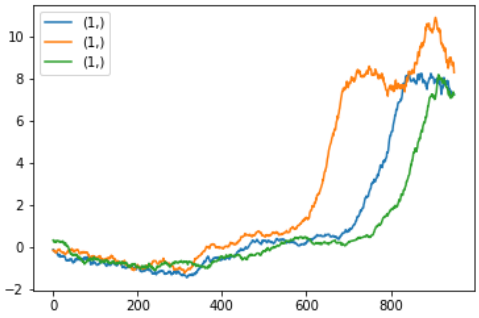



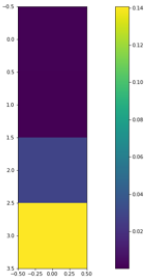



With drop_p = 0, the netowrk still managed to perform reasonably well, somehow learning with even a single neuron! The below figure shows the weights for a single neuron network.



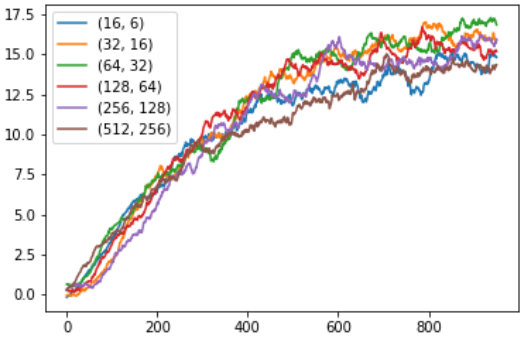



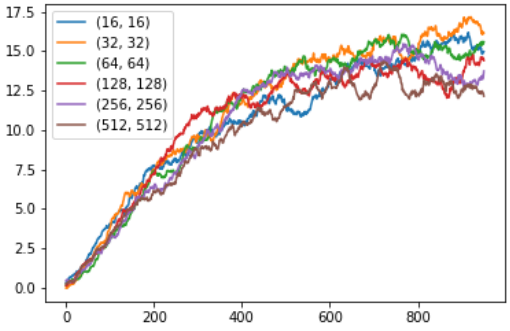


## Two hidden layer Neural Network

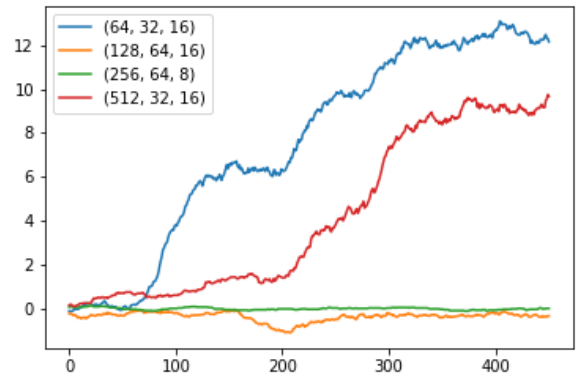
Using two fully connected layers to solve this environment showed very marginal increases in learning speed and reward. In this experiment, layers of the same size, and layers of decreasing size were tested, but show no major improvement. The main trend is that larger networks perform worse than smaller ones, however, clearly, there will be a limit to this.

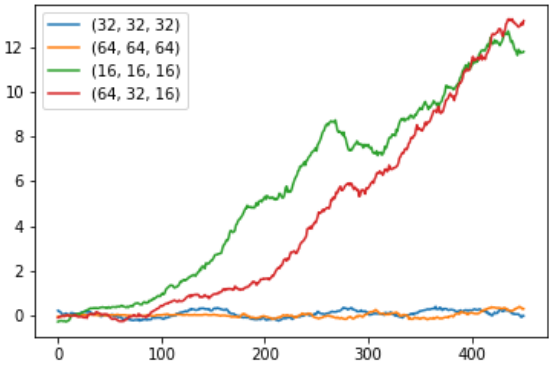


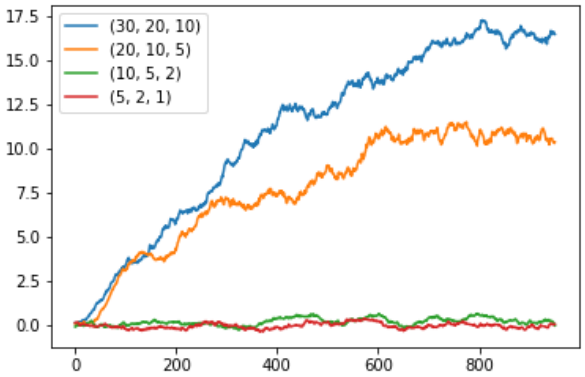



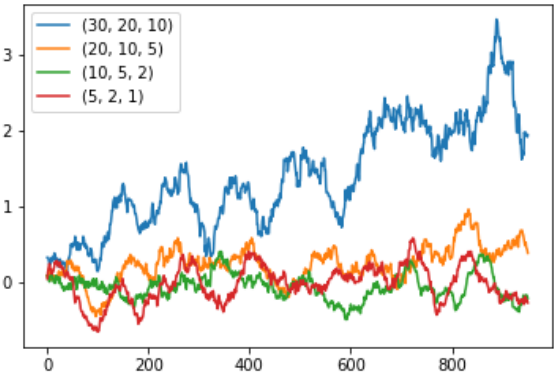
## Three hidden layer Neural Network

Here, three hidden layers was tested, with mixed results. Some networks began to learn, whereas other were unable. Architectures with the same sized and decreasing sized layers were tested. Reasonably sized networks seemed to perform the best, with first layers not too much larger than the state input and decreases to the number of actions performed the best. The top two figures and bottom right have drop_p = 0.5, bottom left has drop_p = 0.



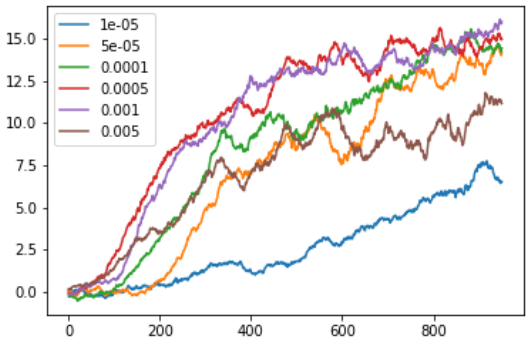



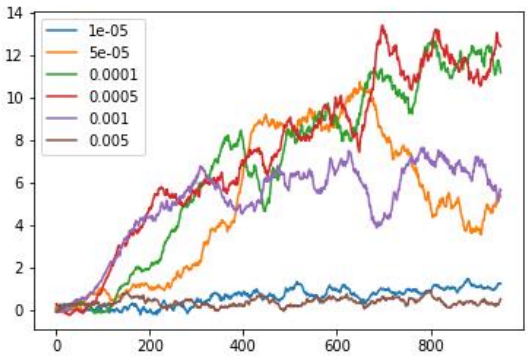




## Learning Rate

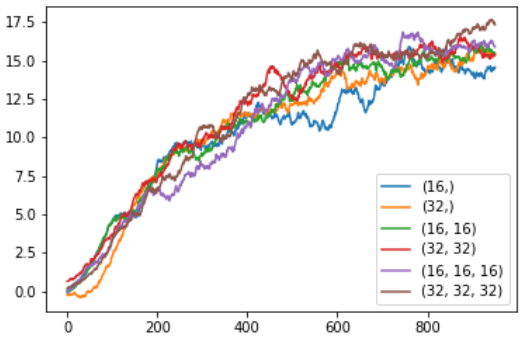
These are all performed with a single hidden layer with 32 nodes, left with drop_p = 0, right with drop_p = 0.5. The overall best performer is a learning rate of 0.0005 for both networks.

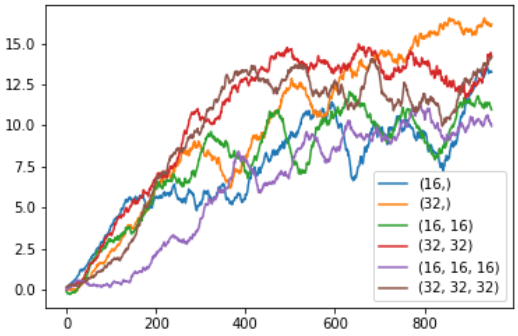


## Network Size Comparison

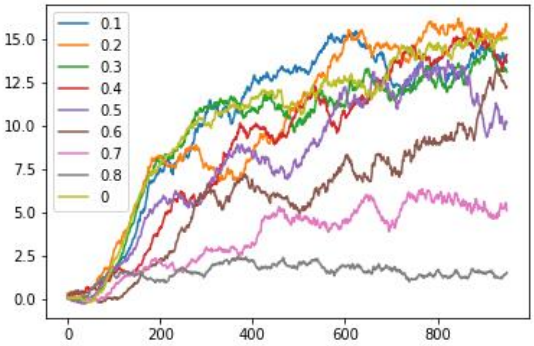
Plotting the Network sizes on the same axes to compare solutions. There is a slight improvement with an increase in layers in the left figure with drop_p = 0. However, this is not reflected in the right figure where the single hidden layer network is outperforming the three hidden layer network at drop_p = 0.2.




## Drop-out

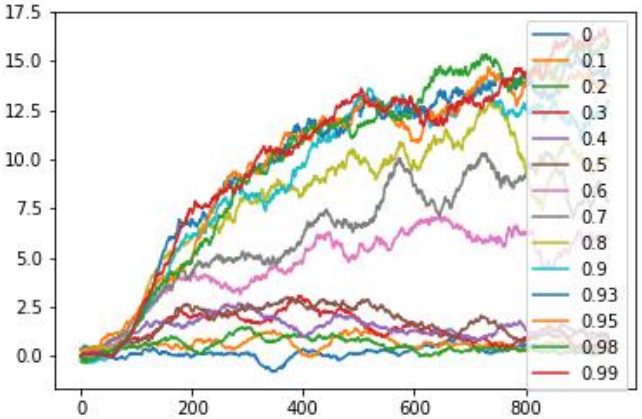
Drop out has a big impact on the performance of the agent. Below is various drop out probabilities with a single 32 neuron hidden layer.




## Discount rate

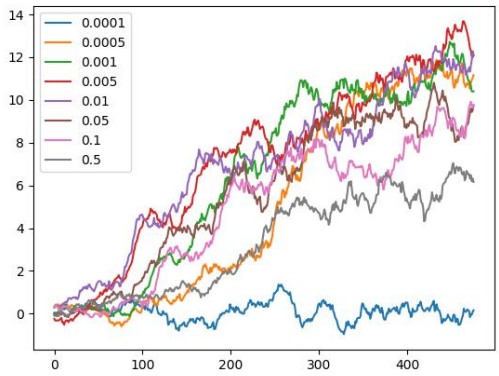
Discount rate, gamma, also has a big impact on performance. The trend here shows an increase in performance with gamma. Note that the colours in this figure are re-used, the top read line is 0.99.

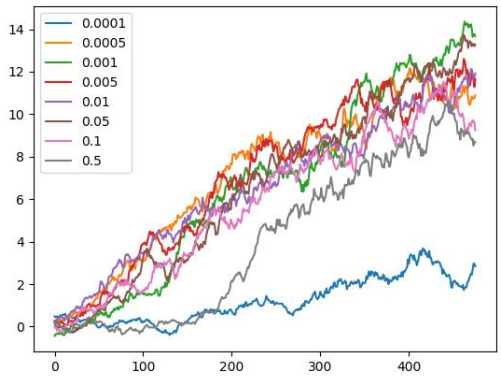
Drop_p = 0



## Soft update parameter

The soft update parameter, tau, has a big impact on learning. Here it can be seen that there are values which maximize the learning. The 'best' value for the left figure with a single 32 neuron hidden layer is 0.005. On the right, using a two hidden layer architecture with 64 and 32 neurons, the final score of tau=0.001 being the highest even though it was relatively slow to start. Regardless, all these results are not statistically significant, and a ballpark figure would probably perform fine.

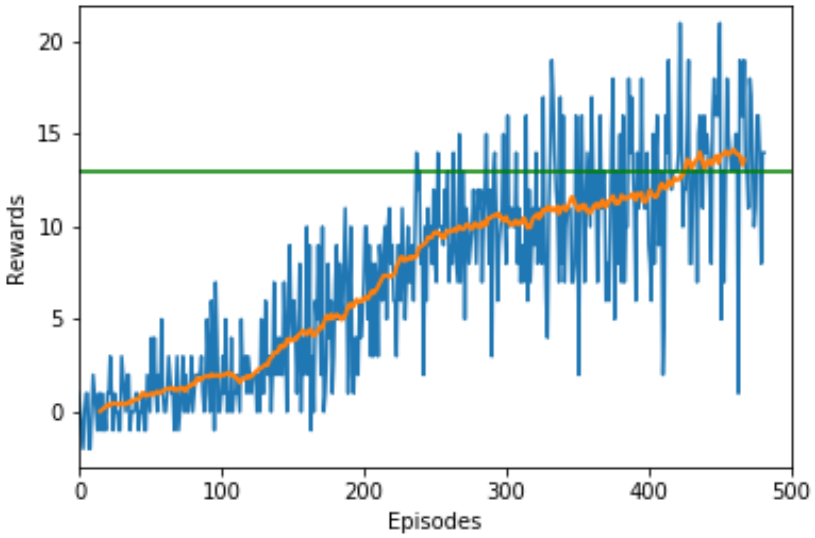


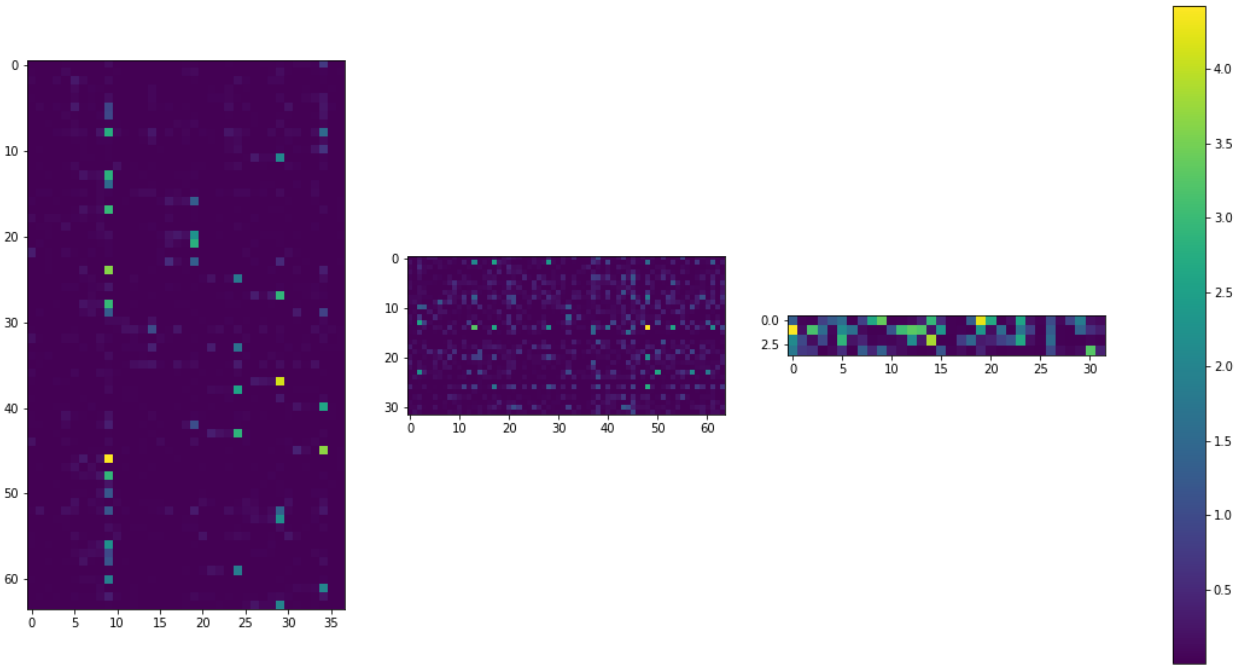




# Final Agent

The hyper parameters used for the agent were based on the generally optimal results from the above experiments. A two hidden layer architecture was used with sizes of 64 and 32 for the first and second layer. These layers were fully connected and used ReLU activation functions. The weights of this network are shown visually below and are printed with the biases in Appendix A.

Other hyper parameters include a learning rate of 0.0005, a discount of 0.99, a dropout probability of 0.2, a soft update parameter of 0.001, buffer size of 100,000 and a batch size of 64.





## Future work

- Prioritized experience replay will be implemented as the rewards are relatively sparse. This should hopefully speed up the learning of the agent with small computational and memory costs.
- I am interested in applying the 'Rainbow' algorithm that employs many improvements on deep Q-learning
- Variations in initial epsilon, its decay and its minimum were not considered but might speed up learning if prioritized experience replay is implemented.
- Finally, working from raw pixels and using convolutional layers to replicate a robot with a camera.

# Appendix – Sample of paramteres

```
hidden_layers.0.weight tensor([[-3.8878e-02, -1.1056e-02, -2.5220e-02,
..., -9.7521e-01,
         -4.6170e-02,  6.3563e-03],
        [ 1.7586e-03,  5.1074e-03,  5.9755e-02,  ...,  1.5996e-01,
         -1.8828e-02, -4.1401e-03],
        [-1.0363e-01,  1.6017e-01,  1.2556e-01,  ...,  1.5733e-01,
          6.1716e-03, -2.8323e-04],
        ...,
        [-1.4985e-02,  2.9117e-02,  4.9675e-03,  ..., -1.3941e+00,
         -9.3745e-04,  1.0899e-03],
        [-7.0097e-02,  1.6726e-01, -1.0107e-02,  ..., -4.8520e-01,
         -2.0390e-02,  1.2777e-02],
        [-9.6498e-02, -1.0170e-01, -8.6159e-02,  ...,  8.5763e-02,
         -1.1463e-02,  8.9471e-04]], device='cuda:0')
hidden_layers.0.bias tensor([ 0.1140, -0.1332, -0.1624, -0.0923,  0.019
5, -0.1280, -0.1916,
        -0.0149, -0.0970, -0.2504,  0.0860,  0.1002,  0.0339,  0.0169,
        -0.1155, -0.2093,  0.0235, -0.0256, -0.0815, -0.1957, -0.1474,
```

```
         0.0716, -0.1194, -0.0414, -0.1215,  0.1582,  0.0008, -0.1123,
         0.1071,  0.0940, -0.0563, -0.1345,  0.0610, -0.0251, -0.2175,
         0.0200, -0.1753, -0.0401, -0.0724, -0.1224, -0.0978, -0.0942,
        -0.1028,  0.0987, -0.0977, -0.0772,  0.0679, -0.1668, -0.0872,
         0.0223, -0.2361, -0.0267, -0.0439, -0.1047, -0.2928, -0.0945,
         0.0676, -0.1021, -0.1143, -0.0668, -0.0578, -0.0337,  0.0538,
         0.0230], device='cuda:0')
hidden_layers.1.weight tensor([[ 0.0381,  0.0225, -0.0410,  ...,  0.254
6, -0.0103, -0.0340],
        [-0.1010,  0.0960,  0.1245,  ..., -0.4553,  0.0285, -0.2296],
        [ 0.0124, -0.0376, -0.2000,  ...,  0.0149, -0.0417, -0.0057],
        ...,
        [ 0.0111, -0.0283, -0.0031,  ...,  0.1521, -0.0732,  0.0845],
        [-0.1624, -0.0761, -0.0835,  ..., -0.2743, -0.0216,  0.2312],
        [ 0.0316, -0.0592, -0.1459,  ...,  0.1516, -0.1300,  0.0898]],
device='cuda:0')
hidden_layers.1.bias tensor([ 0.1233,  0.0589,  0.2342,  0.1157,  0.065
5,  0.0365, -0.0338,
         0.0775,  0.1336, -0.0152,  0.2585,  0.1856,  0.1931,  0.2600,
         0.1220,  0.0562, -0.1207,  0.1645,  0.1016,  0.2218,  0.1150,
         0.1302,  0.1299,  0.2781,  0.1530, -0.1880,  0.2312, -0.0710,
         0.1527,  0.0690,  0.1022,  0.1523], device='cuda:0')
output.weight tensor([[ 0.1636, -0.1817,  0.1273,  0.1053,  0.0897, -0.
0880,  0.0926,
         -0.1211, -0.2654, -0.3213,  0.1691,  0.1316,  0.1536,  0.2111,
         -0.3609,  0.1525, -0.1280,  0.0826,  0.0037,  0.1460, -0.4316,
          0.1040,  0.0856,  0.2507,  0.0972, -0.0364, -0.2103, -0.0706,
          0.1064,  0.1043,  0.0715,  0.0859],
        [ 0.1827,  0.0252,  0.1464,  0.1228,  0.0801,  0.2582,  0.3445,
          0.0538, -0.0690, -0.0767,  0.1731,  0.1404,  0.1460,  0.2045,
         -0.0674,  0.1192, -0.0725,  0.0922,  0.1686,  0.1836, -0.0917,
          0.1270,  0.0938,  0.2613,  0.0864, -0.1368, -0.2307, -0.0161,
          0.1036,  0.0961,  0.2242,  0.1148],
        [ 0.1967, -0.0046,  0.1155,  0.1210,  0.1442,  0.1748, -0.0970,
         -0.2454, -0.1242, -0.0981,  0.0128,  0.2090,  0.0662,  0.0055,
         -0.1585, -0.0008, -0.1059,  0.1435,  0.2312,  0.1675, -0.1543,
          0.0843,  0.0816,  0.0833,  0.1286, -0.1121, -0.2774,  0.0191,
          0.1328,  0.1484, -0.1656,  0.1244],
        [-0.0502,  0.0086,  0.1419,  0.1430,  0.1398, -0.2452, -0.1061,
          0.1811, -0.2311, -0.2420,  0.0577,  0.1388,  0.1882, -0.0149,
         -0.1151,  0.2848, -0.0164,  0.0878, -0.0844,  0.1750, -0.1693,
          0.1471,  0.1304,  0.0867,  0.1143,  0.0896, -0.2198, -0.0937,
          0.0629,  0.1704,  0.2644,  0.1126]], device='cuda:0')
output.bias tensor([ 1.1374,  0.9647,  1.0413,  1.0303], device='cuda:0
')
```