# CSCI 1410 GOT Final: Final (Written, Capstone)

Jonathan Dou

TOTAL POINTS

**100 / 100**

QUESTION 1

*1* Total **100 / 100**

✓ **- 0 pts** *Correct*

# GoT RL Bot Writeup

Jonathan Dou

November 2022

## 1 Introduction

For the ML/Capstone part of this GoT project, I used reinforcement learning with tensorflow keras to create a bot that learns and improves from playing against other bots. There were many hiccups along the way, definitely more of a challenge than I expected, but in the end I was able to beat RandomBot, SafeBot, and AttackBot most of the time on the small 13x13 emptyroom map after training and fine-tuning the reinforcemenet learning model. I unfortunately did not have time to train the model to beat the bots on the larger maps, which takes much longer to improve its performance. The actual winrates might be not displayed on gradescope because it seems to not handle importing tensorflow properly, however I have graphs of many games played over time and will show the results in this writeup.

## 2 Implementation

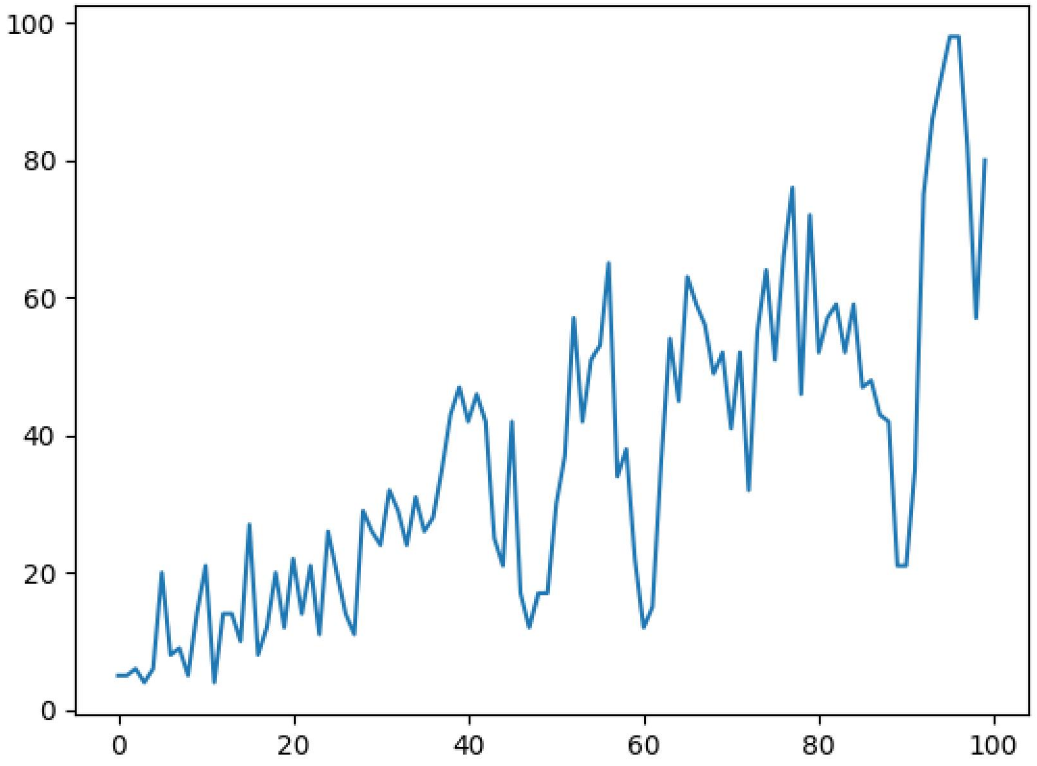
My original design for the state, action, and rewards were very simple. States were basically the map states encoded into a long one-dimnesional vector (values ranged from 1 to 8) depending on the type of the cell on the map. Next it was processed by a feed forward network which outputted a softmaxed probability vector of size 4. (one for each possible action U, D, L, R). Lastly the reward was very sparse. 1 for a win at the end, and -1 for a loss at the end. At end we plug these lists of states, actions, and rewards into our loss function and update the model accordingly. The network looked something like:

```
self.D1 = tf.keras.layers.Dense(256)

self.D2 = tf.keras.layers.Dense(128)

self.D3 = tf.keras.layers.Dense(64)

self.D4 = tf.keras.layers.Dense(4)
```

This did not end up working well for a various number of reasons. I think the main reason was that my representation of the state was suboptimal, the encoded cell values that ranged from 1 to 8 did not really correlate with anything meaningful so the model had trouble learning. After realizing this, I switched to a more effective model that uses convolution layers instead of just feed forward layers. This made much more sense in terms of since now we are mapping the 13x13 game state to a 13x13x8 numpy array. (13x13 for the map size and 8 for each possible type of cell). The third dimension of this state were size 8 one-hot encoded vectors that represented what type of cell was on that particular location. This made much more sense intuitively and also worked better in practice. This was the network that I ended up with:

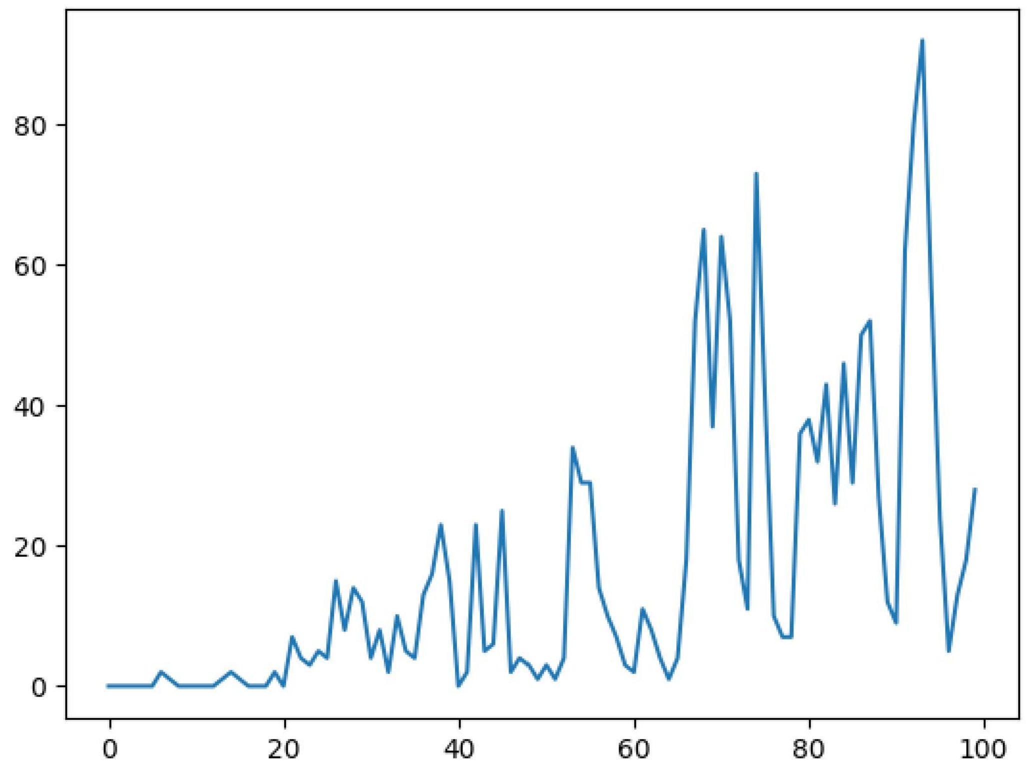
```
self.D1 = tf.keras.layers.Conv2D(filters=256, kernel_size=5, strides = (1,1), activation = '

self.D2 = tf.keras.layers.Conv2D(filters=128, kernel_size=5, strides = (1,1), activation = '

self.D3 = tf.keras.layers.Conv2D(filters=64, kernel_size=5, strides = (1,1), activation = 'r

self.D4 = tf.keras.layers.Dense(num_actions, kernel_initializer=tf.keras.initializers.Random
minval=-0.03, maxval=0.03))
```

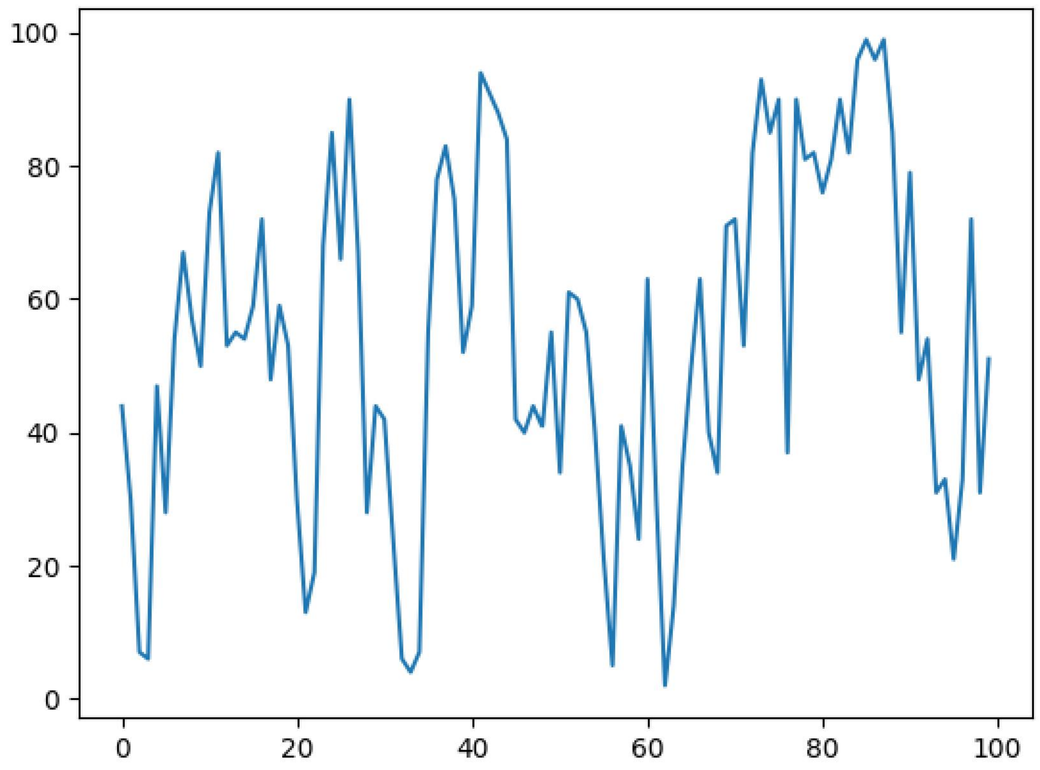
There were a few more things I had to do in order to make this work. I think one of the most important changes was to change the reward system from the sparse (win/lose only system) to a smoother one that involved calcuating the total amount of territory gained at each move. This encouraged the model to learn how to gain territory which ended up in much more efficient learning rather than immediately trying to win against the other bots. For whatever reason, punishing the bot for losing never seemed to help. I just ended up appending a reward of zero at the end for losses and a positive reward at the end for wins. These design worked very well against safebot. And this was the wins per 100 games over the first 10k games against safebot.



As we can see it was able to continously make improvements in its gameplay against Safebot and eventually beating it most of the time in the end (variance is high however). The high variance might be because I used random.choice to pick moves instead of np.argmax in order to encourage more exploration. Without exploration the model would often get stuck doing a repetitive strategy and not learn.

However, this design resulted in a problem when facing against AttackBot. For whatever reason, it would endup stalling against AttackBot to avoid getting killed and resulted in the game timing out over and over again. In order to counter this, I increased the exploration rate manually and forced the bot to pick a ”suboptimal” choice at a small percentage of the time. I just added a small value to the probabilities array and then normalized it so it made the distribution more even and even if the model predicted a near zero probability for one of the choices it would still have a small chance to be picked. I didn't want this to happen forever though because eventually I want the model to converge into an optimal strategy, so I lowered this exploration value after each game such that it dissapears completely after 10000 games. And this was the result against AttackBot: (first graph is first 10k games and second graph is the next 10k-20k games).





As we can see it makes noticeable improvements over the first 10k games and slight improvements over the next 10k games. There was very high variance in the results and I guess it is to be expected since reinforcement learning can be very unstable. There were several times when the gradients exploded initially during training against AttackBot and I ended up lowering the learning rate and created a more gradual reduction in exploration rate over time to increase stability.

## 3 Future ideas and Conclusion

If I had more time I would definitely train the model on the large maps as well, but unfortunately it took several days to get to where I am and I am currently out of time for training. I spent a lot of time trying out different model architectures because most of the time the model was not able to even learn. I want to try adding in different activation layers and maybe try different combination of filter/kernel sizes than the ones I have tested. (5x5 seemed to outperform

3x3 by alot). Overall, I still gained a good amount of experience trying to train a reinforcement learning model from scratch and make adjustments to the architecture and training process along the way to make the training more stable and effective. I'm definitely interested in researching more training techniques that increase model stability and performance; I realized from this project that tiny adjustments to how the model is trained can make very big differences, sometimes more than the architecture itself.

*1* Total **100 / 100**

✓ **- 0 pts** *Correct*