Peter Sloan

Project Report

For my project, I attempted to use deep reinforcement learning to design bots that could navigate through a 2D map towards a goal. I had two main goals in mind: 1. Teach bots to navigate to a goal, 2. Teach bots to work as a team. The topic of AI cooperation is pretty large, as algorithms are often trained to maximize their own reward, with the side effect of creating conflict in group settings. The second problem, however, quickly proved to be out of my league currently, so I settled on just training a single bot in a simple 2D environment for now.

Because deep RL typically works off game observations instead of a dataset, I had to design a game for the bot to work in. This was fairly easy to create in Godot, a game engine I am pretty familiar with. My game allowed the bot to move left, move right, and jump in an effort to reach a position on the map. This raised the question “What info do I need to make available to my bot for it to perform well?”

I wanted the bot to be able to visualize its surroundings, and perpetually be aware of its position relative to the goal. There were many good options for sending this data, such as the normalized direction to the goal, or raycasted distance to the nearest collision object, but I decided the best option was to update the bot with its global position, the position of the goal, and a grid representation of the area surrounding the bot. This way it could see around it, while still always being aware of the goal.

My choice of deep RL algorithm to solve this problem was ‘Proximal Policy Optimization, as it offers a good balance of simplicity and stability. Its input would be a flattened version of the data mentioned above, and I chose a discrete option output with 6 possibilities: left, right, stay, all with either jumping or no jumping.

To reward the bot, I had it receive a negative reward correlated to its euclidean distance to the goal (to encourage movement initially), and a larger positive reward at the end of an episode for a win, or negative for a loss (timeout).

To structure my training, I created a loop that would perform a rollout (typically 8 episodes), return the compiled data of the rollout, and send it through my training function. This repeated until I prompted it to stop. The network parameters would either be loaded from a previously saved file, or initialized with random weights, as there were really no pretrained weights that would fit my data. Each training call would calculate the advantage, iterate through 2 epochs of the collected data, shuffle the data, split it into mini batches of size 48, and compute the loss from there. To compute the loss, I just used the classic PPO loss function:

- *Total Loss = Policy Loss + c1 * Value Loss - c2 * Entropy Bonus*

Defined as follows:

- *Policy Loss: min of either the network output * advantage, or output clipped between 1 + epsilon, 1- epsilon * advantage*
- *Value Loss: Predicted Value - Returns*
- *Entropy: Calculated from probability distribution*

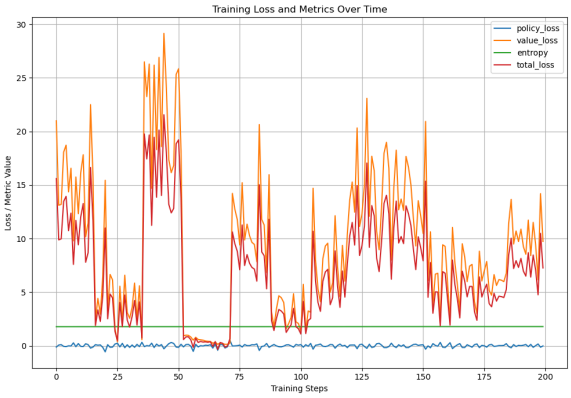
Peter Sloan

I started by trying a simple network with 3 hidden layers, totaling 21,959 parameters. My network used the Adam optimizer. Instead of having two separate networks for getting actions and calculating values, I instead included it in the same network, and gave it 2 output heads, one for each respective job. This was done for the sake of simplicity in training.

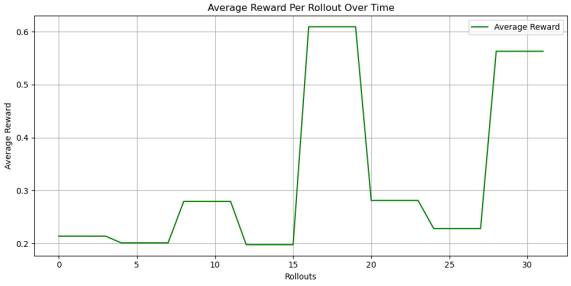
Initial training worked pretty well, but I quickly learned that it was overfitting, as the goal's position was static, and it memorized the sequence required to get to the goal, and not the relationship between input and movement. To combat this, I began to randomize the bot's position and the goal's position on every episode, which made it much harder to train properly.

To combat this, I attempted a few different networks. I tried a larger network, a smaller network, but the most effective was a dual-stream network I created. This network passed the positional inputs through one stream of 2 hidden layers, and the grid went through 3 convolutional layers and was flattened. Then the two streams were merged by two more hidden layers, and sent through each output head. The new network had significantly more parameters, totaling 181,575, but it ended up converging a lot better.

The next problem I encountered was a very unstable loss calculation. When visualizing the data, I found the value loss was orders of magnitude higher than the policy loss. To fix this, I had to lower the reward values, which led to a much better training process. I ended up having to do a lot of fine tuning, but I finished with the following values: LR = 2e-4, epsilon = .1, gamma = .99, lam =.99



I ended training by collecting data, storing policy loss, value loss, entropy, and total loss on every 1000 gradient descent steps. I also collected the average reward per rollout. Frankly, the average reward was not nearly as good as I would have expected in such a simple environment, ending over 75%. The total loss dropped to about 8, mostly due to the high value loss. From the data I collected, I think the most obvious next step I need to take is to stabilize the value loss. I believe this is due to the value head not updating effectively, so I likely will split the value function into its own network.



I think I made pretty significant progress on the first goal, as the bots were able to pretty consistently find their way to the goal. I think it will take more fine tuning to be able to work on the second goal, though I believe that it is within reach.

Peter Sloan

Total Hours Logged: 35

1. **Research**

Date	Duration	Event
4/9/25	2 hrs	Listened to a few online lectures on PPO
4/9/25	1 hr	Wrote up a basic design plan and chatted with ChatGPT on best practices for implementing the algorithm
4/14/25	1 hr	Created a UML sequence diagram outlining the training process to better understand the flow

Research Time Total: 4 hrs

2. **Data Collection** (as I am doing a deep RL algorithm, I considered creating the environment as data collection)

Date	Duration	Event
4/10/25	4 hrs	Created a basic 2d game where a robot tries to reach a goal
4/11/25	2 hrs	Modified the architecture to handle playing batches
4/11/25	3 hrs	Created a script that will download the network, and create bots to play an episode of the game.

Data Collection Time Total: 9 hrs

3. **Models**

Date	Duration	Event
4/14/25	2 hrs	Designed a simple network and implemented it in bot script
4/14/25	3 hrs	Created a trainer class, and implemented the train method
4/15/25	2 hrs	Debugged my train method
4/15/25	1 hr	Swapped continues output space for discrete outputs
4/16/25	2 hrs	Changed bots to handle multithreading for better performance
4/17/25	2 hrs	Modeled/debugged training results
4/18/25	2 hrs	Created and instantiated a dual stream network
4/18/25	1 hr	Debugged network issues
4/21/25	1.5 hrs	More debugging
4/21/25	.5 hrs	Worked with TA to isolate issues
4/21/25	1 hr	Simplified model to take more limited input
4/21/25	2 hrs	Tuned hyperparameters like learning rate, rewards, etc
4/21/25	2 hrs	Added data collection

Model Time Total: 22 hrs