Team Capture the Flag Problem

Kushagra Tiwary

I. Introduction and Problem Motivation

Games have caught the attention of a lot of AI researchers because there is an agent, clear objective and plan present, identifiable states with rewards, and a win/lose end game. This puts some of the new Machine Learning and decision making processes/techniques to test since the game environment itself provides the objective functions, the states, and the rewards.

One such game is the capture the flag. This game has two teams and two zones- Blue (player/AI) and Red (enemy). The objective of the game is to retrieve the enemy flag and bring it back in the blue territory. The blue team has 6 vehicles- 4 UGVs (circles) and 2 UAVs (squares). The UGVs can kill in their home territory and be killed in the enemy territory. The UAV vehicle are exploratory vehicles- they cannot cannot be killed, have large observation range and can travel over obstacles. The flag is present in the red territory and can only be captured by a UGV.

Moreover, each vehicle only has limited vision. The UGVs can only *see* 1-2 squares in each of the four directions. The UAVs can see up to 4-5 squares in all directions and can fly over obstacles. The state space are *all* possible combinations and positions of all the vehicles in the game environment. The actions each vehicle can take is going Forward, Backward, Right, and Left. If any UGV walks over the flag, then the flag is automatically taken by that UGV.

The whole game can be divided into accomplishing smaller objectives which will also help us formulate the solution in section 4. The objective of the game is to (in order):

1. find the enemy flag
2. pass through the enemy patrol
3. capture the enemy flag using ground units
4. Come back to home territory without being killed

Since this problem is very complex in nature with respect to the game environment, reward function and the states, we have decided to start things off simple. We make the following assumptions to keep things simple:

1. We will hard-code the enemy tanks to patrol the border (move side to side near the edge) between the territories, or have them move randomly throughout the red territory. This will allow us to focus on developing a policy for the blue team.
2. The enemy tank cannot attack us. Therefore, there will never be a state in the game where a red tank will come into blue territory.
3. The above assumption makes sure that the *only* way blue can loose the game is if all of its UGVs are killed.

4. The *score* of the game is time. The more time it takes for the blue team to find the flag, the lower the score goes. In whatever solution we chose, we will not care about this. Our main goal is to **win** the game i.e. capture the flag and come back safely into our territory.

5. The UGVs and UAVs will also have some memory and be able to pass on information to one another. For example, one possible solution discussed is that the UAVs can make a map by exploring the whole game area in some random or snake-like fashion. This information can then be accessed by the UGVs in order find the location and retrieve the flag.

Given this information, we formulate the problem in the following way: Given an environment operating with the above rules and assumptions, we want to create some feature representation based on what the blue team vehicles/agents can see and a map of the environment generated by the UAVs, *and* combine this high-dimensional input with some sort of linear value functions or policy representations to meet the objectives defined above.

This is an important problem to solve because we are modeling independent agents which will have to successfully learn control policies from a very complex environment. Moreover, the signals these models will have to interpret are very high-dimensional. This area is very important since AI agents operating in the real world will require to understand and then act on complex high-dimensional sensory inputs. This is also one of the reasons why such games are being used by AI researchers.

II. Limits of Current Practice

For this section I will explore the applications of reinforcement learning in similar problems. The very first application of reinforcement learning in games was by Deep Mind in 2013 when they showed that reinforcement learning can be used to play Atari games from 1970s.[1] From then on, reinforcement learning has been shown to be successful in playing pong [5], beating the world champion at Go [2], and in countless other video games like Doom etc.[3,4] This is because a game setting has a natural reward, punishment, and a state and action space. However, as Karpathy's blog [5] explains that even with the current success, games such as Montezuma's Revenge are very hard problems for RL because the AI fails 99% of the time (the player gets eaten by a monster) but doesn't know why. This is because it is very easy for a human to realize that the way to win is to get the key which is located at some random point, but the AI doesn't know this and with huge state spaces it fails more than it succeeds in finding the key.

There is a stark difference, however, in the input space of the games referenced above and capture the flag. One major difference is that the vehicles are only able to perceive a couple of squares therefore the input space is much smaller (even smaller for UGVs) and very sparse- partial observability. Solutions discussed try to overcome this vision problem and still use some of the well-known solutions in the area as described in section IV.

Even though this game's solution may seem close to the ones referenced in the papers above, this is a much a harder problem if we don't make the assumptions we made in Section 1. Instead of a simple patrol, we could train an adversary to minimize the reward of the Blue team. Moreover, the objective could also be designed to take into account time spent playing the game and have a negative effect through that respect. Therefore, with even longer action sequences where there is no positive rewards, and a smart enemy make this a very hard problem to solve.

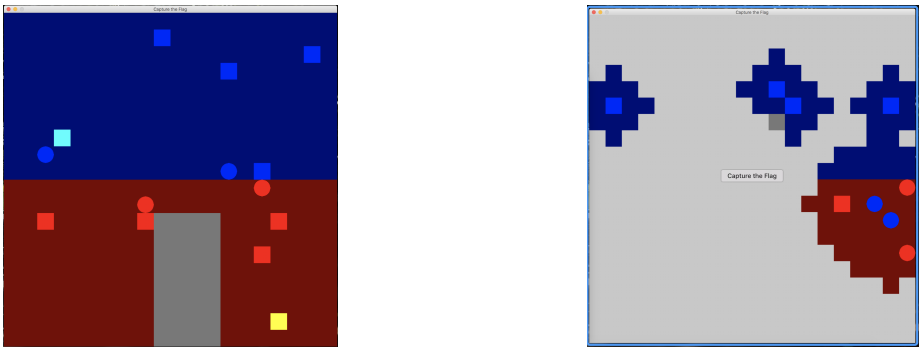

Figure 1: Figure showing how the input space is very different to the ones of many pf the ATARI games since this the agents can only some fixed amount of the state space at a given point in time.
Right shows the input that will be used in the solution techniques presented in Section IV.

III. Background Literature and State-of-the-art

There has been much recent progress in this field since 2013- when the Playing Atari with Deep Reinforcement Learning paper were published by Mnih et al. There are two main areas however, to approach a solution: Deep Q- Networks and Policy networks. A Q network is a Q-function approximator and tries to learn a state-action value function Q. The intent is to make the value of Q(s, a) with state s and action a close to $r + \gamma maxQ(s, a)$ after observing a transition (s, a, r, s), where the actions can be chosen arbitrarily. We generally use a greedy or epsilon-greedy policy.

The other method is to use a policy network. These networks generate a policy function directly- a state input, outputs an action. Now, we can simply find out which actions work well and increase their probability. We maximize the total reward by directly maximizing expected reward using gradient methods.

The state of the art in this field is the AlphaGo Policy network which beat the greatest Go Player in the world, however, it is not feasible to design something similar with the time and computation constraints. Similar policy networks are being used to compete against even *more* complex games such as star-craft. [4]

IV. Solution Techniques

Given the problem, we tried a combination of solutions - both inspired from the methods and algorithms we studied in class. The initial solution proposed until Iteration one was to make the partial observability problem into an *almost* fully observability problem using the UAVs and then use some sort of planning algorithm can be used to get to the flag- since UAVs can't get killed in the enemy territory.

I. Solution 1

The crux of the initial proposal was *given this game setting with its rules and restrictions, what is the quickest way to find the flag?* This is because there exist fast algorithms in countless research papers to get to goal state from an initial state in a most efficient way. To do so we iterated over 3 such exploration search techniques to converge over one that works the fastest. The initial assumption all the three solutions follows is that we can use the observations to create a map of the game and then use that to plan our path since if the UAVs can find a path from Blue territory to the flag, then so can the UGVs.

I.1 Map Exploration

The first two map exploration techniques were straightforward implementations of random search in enemy territory and snake like map search. We initialize an array as the same size as the grid world for the game. Each of the array cells can be marked *unknown*, or with a value like *obstacle*, *flag*, *blue territory*, *red territory* etc. A random search would be picking a random set of locations in the enemy territory and then outputting an action to the UAV which takes the UAV a step closer to that goal state. By this process, the UAV will observe all the neighboring cells and can detect the flag. This policy works well for small state spaces however, fails to work well for larger grid worlds.

The second technique is going through the enemy territory in a snake like fashion, observing nearby cells until we find a flag. One of the advantages of this exploration algorithm is that it is very thorough, therefore the path planning algorithms can find an optimal way since the UAVs will have many observations about the whole grid world before it finds the flags.

These two exploration techniques were implemented to create a baseline to test the performance of the third algorithm. The third algorithm is based on splitting the search space into quadrants at each step and then deciding an action which maximizes the amount of *new* information (states/cells we haven't explored) we gain.

The way the third algorithm does that is through dividing the subspace from its location into 4 quadrants. It then finds which of those quadrants have the most unexplored territory and then respectively divides the work between 2 UAVs such that one explores one quadrant and the other, the other quadrant. It keeps doing this until the flag is found and some fixed percentage of the game world is discovered.

I.2 Code Implementation

I implemented the above algorithm in code through the use of an Action Policy class with input as the observation (current state) and output an action in python. The class definition is defined in *action_space.py* which is attached with the files provided at the time of submission. The main file that runs this map exploration is *cap_learn.py*. The *Action_Space* class defined needs to be initialized with four parameters: number of UAVs, UGVs, positions of the UAVs, UGVs and lastly the environment instance itself. The public function is the *get_action()* function with inputs *previous_observation* and outputs the next action. The map is stored privately as a 2D array, *map*, and map operations are stored in a private dictionary called *operator_dict*. This dictionary maps the value of the current call given the current observation and previous map value. This map update takes O(dim(M)ˆ 2) time to run. The *operator_dict* dictionary is defined in *constants.py* file.

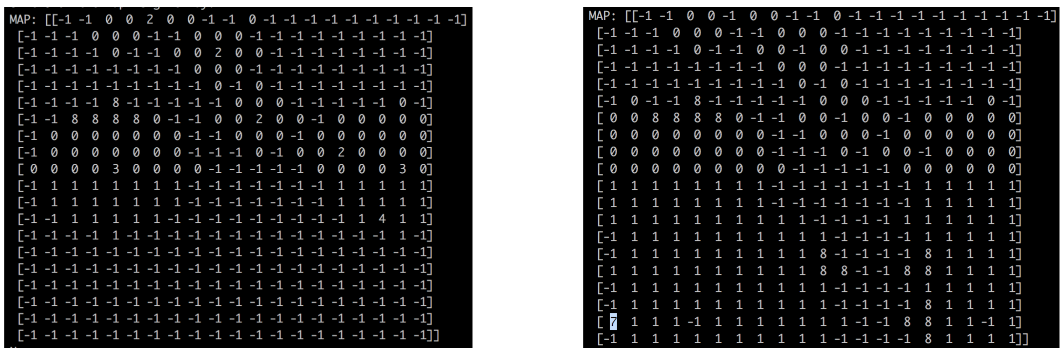

Figure 2: Figure showing how the map exploration algorithm converges to a solution by finding quadrants that are most unknown and then exploring those. The picture on the right shows that the UAV has found the flag after 13 steps. Notice that both the UAVs explored different quadrants. The quadrants they explored were the ones closet to them interms or euclidean distance.

The stopping condition of the map search is when the flag is found, *and* some pre-defined amount of the state space has been explored. After this a flag, *self.env_map_done* is set to *True*, and the policy switches to *_get_dual_input_action()* from *_get_map_based_action()* with inputs the map and the partial observations and the output as an action. This is where a path planning could be called as discussed earlier.

Once the algorithm used to locate the flag using UAVs had become robust, this problem becomes *much* easier to solve. With this new set of information, the new problem has been condensed down to using some form path planning algorithm to get to a desired state on a map while being careful about possible enemies. As countless papers in path planning area show, this problem has been solved. The ‘possible enemies’ problem can be solved by using the observation field of the UGV itself.

```
# _ means assignment is independent of a
# in the current implementation enemy vehicle dont cross the border
operator_dict= {
    (UNKNOWN, UNKNOWN): UNKNOWN,

    ('xx', TEAM1_FLAG): TEAM1_FLAG,
    ('xx', TEAM2_FLAG): TEAM2_FLAG,
    ('xx', OBSTACLE): OBSTACLE,
    ('xx', TEAM2_FLAG): TEAM2_FLAG,
    ('xx', TEAM1_UGV): UNKNOWN,
    ('xx', TEAM1_UAV): UNKNOWN,
    ('xx', TEAM1_BACKGROUND): TEAM1_BACKGROUND,
    ('xx', TEAM2_BACKGROUND): TEAM2_BACKGROUND,
    ('xx', TEAM2_UGV): TEAM2_BACKGROUND,
    ('xx', TEAM2_UAV): TEAM2_BACKGROUND,
    (OBSTACLE, 'xx'): OBSTACLE,
    (TEAM1_BACKGROUND, 'xx'): TEAM1_BACKGROUND,
    (TEAM2_BACKGROUND, 'xx'): TEAM2_BACKGROUND,

}
```

Figure 3: Figure showing that given a previous cell with some value, and the current observation of that cell by the UAV, what is the value of that cell on the map in the next step. These commands had to be hard codded onto the functionality of the Action Space class using a dictionary. Since this is a more classical AI map search algorithm, we have to take into account of all such possible combinations.

II. Solution 2

The first approach we studied is a more classical way of tackling the problem. The second solution we tried- also inspired by the grid world homework- involves Reinforcement Learning. Since this is a partial observability problem and there are longer term rewards, reinforcement learning can be hard to apply on this as we explained in section II. Therefore, we believe that it is more beneficial if we attempt to solve a smaller version of the problem first before applying this to the whole problem we described in the introduction.

To apply RL, we have implemented a couple of changes to problem. Specifically, we have decreased the number of UGVs to 2 from 4, removed all obstacles, and changed the goal of the team to find the flag. This has drastically reduced the action space to 2 $*$ (# of actions by each UGV) from 6 $*$ (# of actions by each vehicle). Additionally, time does play a factor in the game now and the time taken to reach the goal will have an effect on how we structure our learning. An instance of the new game world is shown below.

II.1 Deep-Q-Networks

To start, however, I tried to implement a deep-Q- network for this problem. The reason for this choice was that we have decreased the state and the action space of the game tremendously and simplified it. Specifically, the agent won't get stuck in between obstacles and will not have to worry about navigating through a complex path to find the flag. This also means that we've shorten the long term rewards that would have made it difficult for an RL algorithm to converge i.e. it can

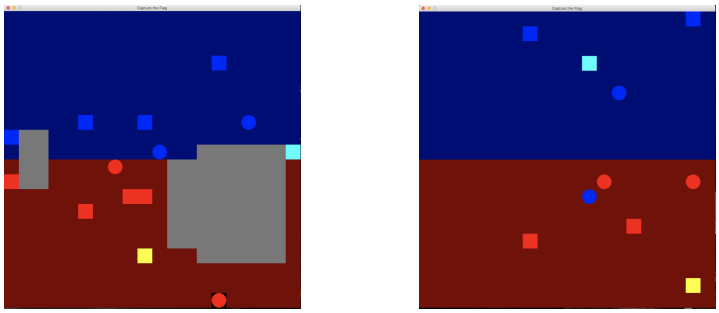

Figure 4: Figures comparing how the game setting has been changed so that we can test RL techniques on this game. By lessening the state spaces, removing obstacles, and decreasing the number of vehicles, we have made the game simpler. The rationale to do this is so that we can test if RL even works on a partial observability problem. Remember that the agent still has the same amount of observability shown in section 1.

converge faster over an optimal Q function iteratively since we have to search over smaller and less complex state and action spaces.

One of the most important aspects of Q-Learning is defining a reward function. To speed up learning, and since we know that the flag has to be in enemy territory, we have incorporated a sense of direction in the reward function as well. Specifically, if the UGV is in blue territory and takes an action towards the red territory, it is rewarded a very small positive reward. An opposite action yields a very small negative reward. As soon as it enters the red territory, this reward function yields itself to the standard reward function defined. I have talked about this in length with Denis regarding how to make a reward function that does not have local minima or be continuous or differentiable for this game. This is why I have dropped the idea (or try later) of using a reward function dependent of distance from the flag since it is recommended that not to use a distance dependent reward function in a time dependent scenario. I have also tried learning with only the standard reward function as well.

The standard reward function we have defined is pretty straightforward. Winning and losing a game yields a reward of +1 and -1 respectively. When blue agent dies, it receives a negative reward of 0.5/(# of units) and a positive reward of the same amount if a red agent dies. Another problem is that this scenario will never happen since, if you recall from section I, we have fixed the behavior of the red agents to only patrol the area.

II.2 Code Implementation

After looking at examples of Atari games and following from the successful use of a deep-Q network on ping pong my network architecture is the following: The network is composed of a convolutional layer followed by a relu, then a maxpool, then 2 convolutional with relu, and then followed by 2 fully connected layers. I used convolutional layers since spatial information can be very useful. The input to the network is four previous partial observation matricies so that it can recognize motion. We also make use replay memory as well. The network is defined in the *_create_network()* function.

The main function is the train network function. This function trains the function using an

AdamOptimizer, and loss function defined in *_compute_cost()*. This is basically the l2 loss between target Q and the current action and Q value. This basically follows steps to get the actions, scale epsilon down, run selected actions and run observations for a selected amount of time to generate episodes and stack them for network training, start training once we have finish observing, and then update the network. The hyper-parameters were initialized as follows:

1. $\gamma = 0.99$#decay rate of past obs.
2. $\eta = 10^{-6}$# Learning Rate
3. $\epsilon_{initial} = 1.00$
4. $\epsilon_{final} = 0.05$
5. $OBSERVE = 5000$# Time steps to observe before training.
6. $REPLAYMEMORY = 590000$# decay rate of past obs.

Since the code for this problem was very long, I have not included it in the paper. Please refer to file *_rl_q_learning.py* for all the code. To run the file, simply *python _rl_q_learning.py*. Please comment out the lines 373 through 379, if no animation necessary. Animation slows down the whole thing, and training happens much much slower.

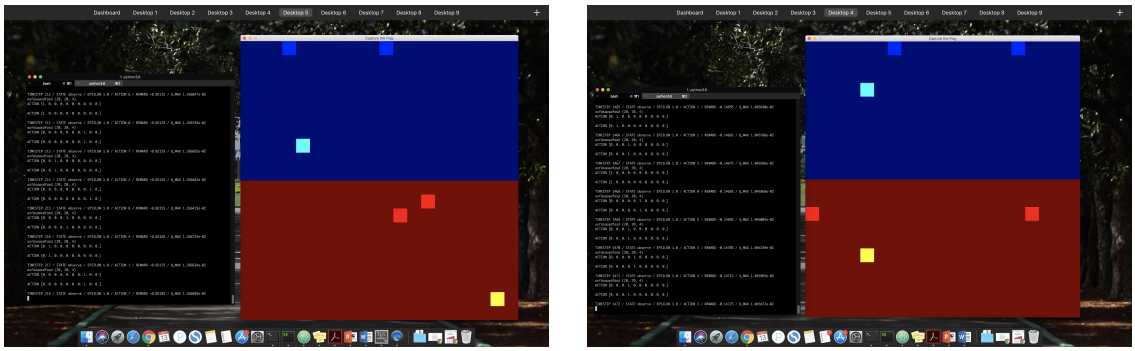

Figure 5: Figures comparing the reward functions in the game at 2000^{th} and 1500^{th} step. One of the problems that I faced was that the agent kept getting stuck at the end of the maze which is why I needed an extension to figure out the problem. Notice the output of the terminal on the right shows the current reward however, the agent still seems to not take that into consideration when outputting the next action. This is probably due to the fact that this hasn't been trained for very long time. Even in the game of ping pong, there needs to be at least 5 million such steps for the agent to start beating the human player.

Initially, the algorithm outputs actions that make the agent go in random sequences however, it seems that with the second reward function, the agent gets stuck more in the middle than in the corners as it was previously observed. The agents' are more aggressive, exploring the enemy terrotiry by the $7000^{th} - 8000^{th}$ step on average. Two videos has been attached showing the training at 2000^{th} with the first reward function and the 7000^{th} step. In the second video, the agent gets eaten around 3 minutes. Before that, there isn't much to see.

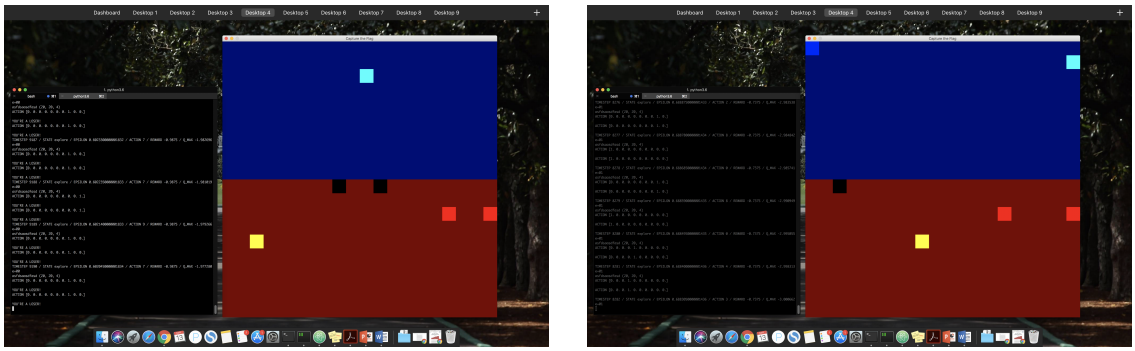

Figure 6: Figure shows after 10000 of steps by the RL algorithm with the non-standard second reward function, the agents finally made their way to the enemy territory for the first time. They get killed however, which is marked by the two black dots, however, the fact that they made it there shows that the agents reacted to the reward and changed their actions to maximize the expected reward. This gives me some hope that the Q learning algorithm implemented works well. However, more training is required for any useful output. The hypothesis is that over time the RL will realize that exploring in the goal territory gives higher reward and that will turn into a policy. While exploring the red territories and getting killed lots of times, the algorithm will realize that it is important to stay away from the red UGVs. Hopefully, it will end up near the flag eventually, and that will realize that that is the goal.

Implementing the above algorithm and understanding it fully, took longer than expected which is why I wasn't able to train on the GPUs. Although we can train the algorithm as is on a GPU, some small tweaks are required so that this can happen. Specifically, whenever we win or loose a game, the algorithm needs to reset itself without manual intervention. This is a simple if-else statement based on the *terminal* value. Please refer to the code for the entire implementation. My files are *action_space.py, cap_learn.py, constants.py, rl_cap_learn.py, and rl_q_learning.py.*

V. Discussion and Future Work

We tried two possible techniques to play the game of capture the flag using the methods learned in class. The first technique is the more classical way of doing things and we showed how the map exploration technique we implemented can help us achieve the objective of the game. However, as I realized while implementing this algorithm, that hard coding it and taking into account all the heuristics is very cumbersome. I had to take into account every possible combination of values that the UAV could find and hard code them into the algorithm as shown in section IV.1.2. While creating the map I had to code through all possible actions and tell it to pick one based on the given inputs. This was very cumbersome and has led to me appreciate deep learning even more. For larger state spaces, and more actions such a classical AI perspective of search and exploration that we talked about in the beginning of the class can lead to a lot of code and many small bugs and even then, one might miss to take all the possible combinations into account.

However, the RL approach takes care of these feature representations and decisions for us. This is why RL is a much stronger and robust solution than what I had implemented before but that's only if it works. It took me much longer than what I had expected to get started with the Q-learning code and gymcap tutorial in python and then implement it over Capture the Flag, which is why I wasn't able to meet to fully train and test at least over one game.

The RL approach that we took attempted to solve a smaller subset of the problem and with more time I could've run the code on GPUs so that training and testing would be faster. Nevertheless, regarding future work, there are so many things that we can test just because it is such a huge problem. Although, the first step would be to make sure that this deep RL works for this problem we defined above in a robust fashion, we can increase the state and action space by slowly adding more complexity, but we will have to be careful since Q-learning might fail as complexity increases.

References

[1] Playing Atari with Deep Reinforcement Learning, Volodymyr Mnih, Koray Kavukcuoglu, David Silver; arXiv:1312.5602

[2] Mastering the Game of Go without Human Knowledge, David Silver, Julian Schrittwieser, Karen Simonyan

[3] Playing FPS Games with Deep Reinforcement Learning, Guillaume Lample, Devendra Singh Chaplot; arXiv:1609.05521v1

[4] StarCraft II: A New Challenge for Reinforcement Learning, Oriol Vinyals Timo Ewalds Sergey Bartunov; Deep Mind and Blizzard

[5] Deep Reinforcement Learning: Pong from Pixels, Karpathy Andrej http://karpathy.github.io/2016/05/31/rl/