Better, Smarter, Faster

Priyanshu Shrivastava (ps1276), Nikhil Tammina (nt444)

1 Introduction

The previous project places a significant emphasis on using a probabilistic model of the environment whenever there is uncertainty to assist in decision-making. Agent, Prey, and Predator are the three key components of this project. The main objective is to compel the agent to catch the prey before the predator catches the agent. Although there is a predator that is moving towards the agent, the search algorithm we used, Dijkstra's, is an algorithm that is used to find the shortest paths between nodes in the graph based on the predator's, agent's, and prey's positions and makes the best possible decisions for the agent by dodging the predator and catching the prey. As in this project, we are entirely dependent on calculating the distance of the agent, prey, and predator and deciding the next step for an agent, but here instead of being concerned about the next step of the agent we are using a utility function in which it maps the entire graph into the entire state space and each state has a utility value assigned to it. These utility values maps an optimal path for the agent to catch the prey and avoid the predator.

2 Design Choices

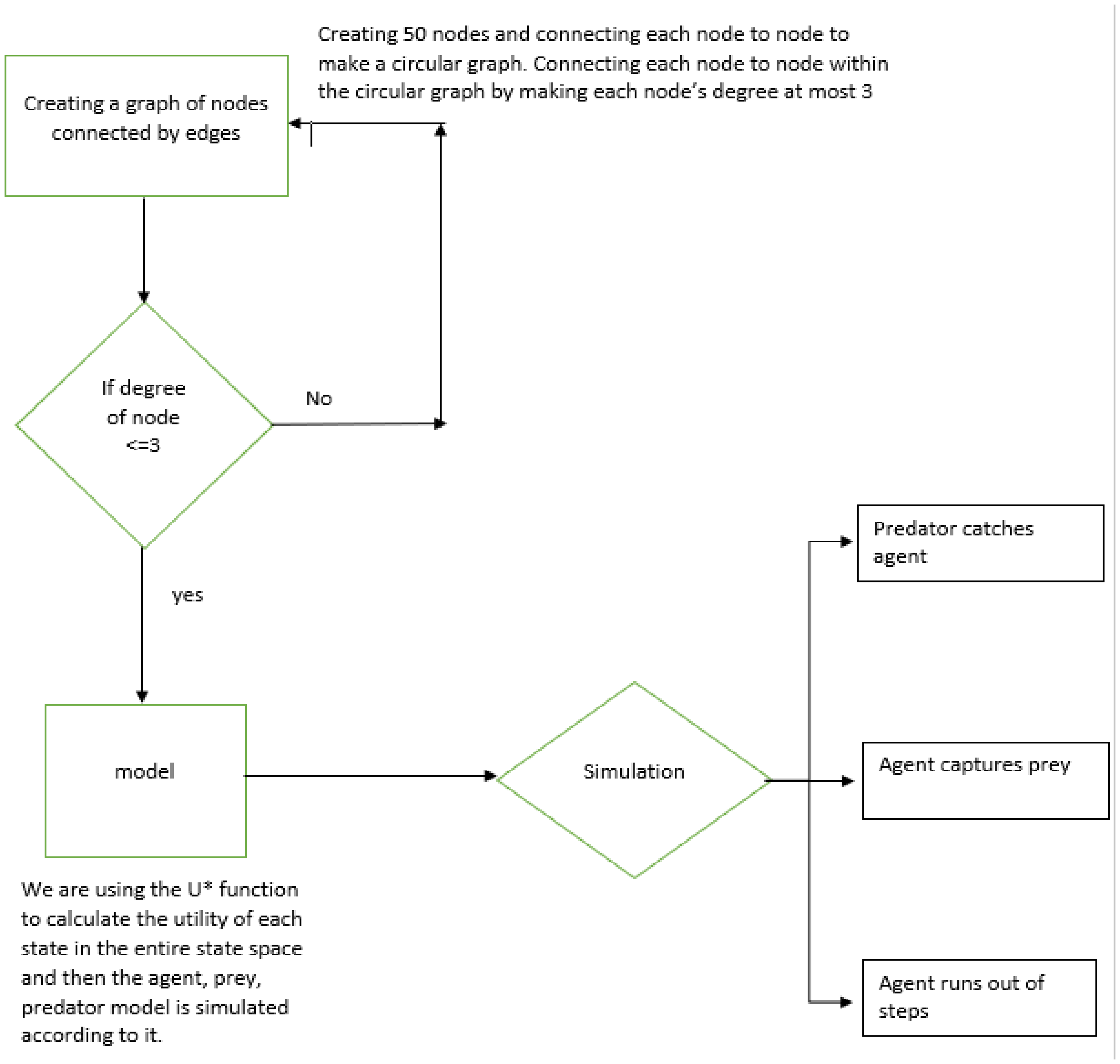

Creating 50 nodes and connecting each node to node to make a circular graph. Connecting each node to node within the circular graph by making each node's degree at most 3
Creating a graph of nodes connected by edges
If degree of node <=3
No
yes
model
Simulation
Predator catches agent
Agent captures prey
Agent runs out of steps
We are using the U* function to calculate the utility of each state in the entire state space and then the agent, prey, predator model is simulated according to it.

Figure - 1: Model Design

To begin with, we started the project by creating a graph of nodes numbered from 0 to 49 in a large circle. Now, for each randomly picked node, we have added an edge to another node that is within 5 steps forward or backward along the primary loop. We are creating three major components, Agent, predator, and prey. Prey is spawned randomly in any node. We made sure that neither agent and predator nor agent and prey are spawned in the same node as the entire system loses or wins respectively if that's the case in the first iteration. We have designed Dijkstra's algorithm in order to find the shortest path between nodes. It finds the shortest path from predator to agent and from agent to prey and the goal is to make sure that the agent avoids the predator based on different conditions of different agents and reaches the prey. In Figure-1 we have used how U* function to calculate the utility of each state in the entire state space followed by simulation.

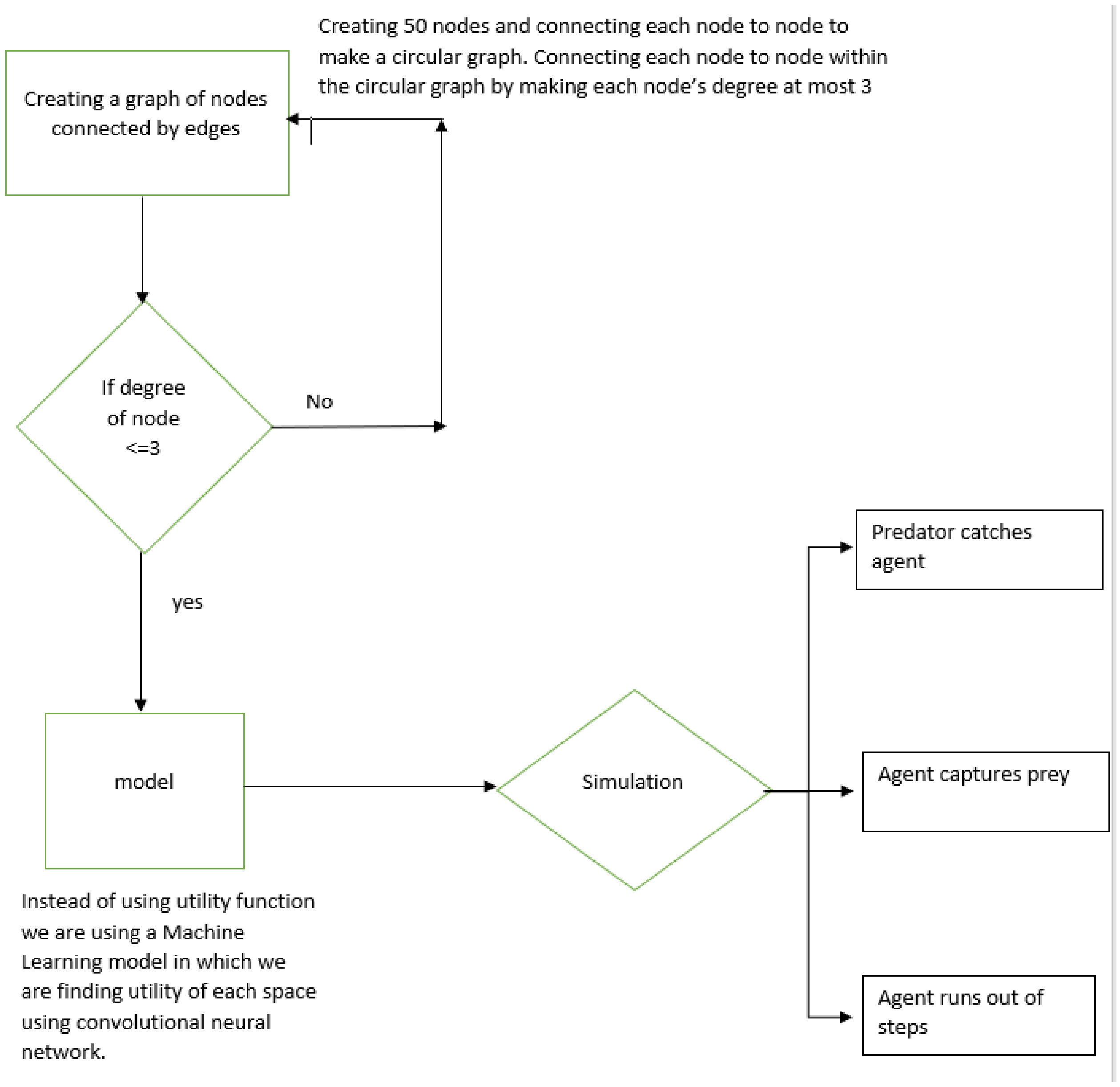

Creating 50 nodes and connecting each node to node to make a circular graph. Connecting each node to node within the circular graph by making each node's degree at most 3
Creating a graph of nodes connected by edges
If degree of node <=3
No
yes
model
Instead of using utility function we are using a Machine Learning model in which we are finding utility of each space using convolutional neural network.
Simulation
Predator catches agent
Agent captures prey
Agent runs out of steps

Figure - 2: Model Design

In figure-2, we have built a neural network that predicts the utility value for each state from the entire state space and reduces the extensive computation required to implement the utility function that is used for calculating the U* value for each state. The prey transitions to any randomly chosen neighboring node or has an equal probability of remaining there. The distracted predator transitions towards the neighboring nodes that are closer to the agent with

60% probability, and the probability of it going to any neighboring node irrespective of the distance between the predator and the agent is 40%.

Q. How many distinct states (configurations of predator, agent, prey) are possible in this environment?

As there are 50 nodes in this graph and each state is of the form (agent, prey, and predator), there are 50*50*50 combinations i.e 125,000 possible distinct states in this environment.

3 Value Iteration

Value iteration is an algorithm which helps in sloving the markov decision process (MDPs). In MDP we are concerned about the future actions and which actions are best for the said scenario. As the name suggests, it basically keeps iterating until it finds optimal estimate of action value function also known as Q-function for that MDP. The algorithm continues to iterate until the Q-Function has converged to the optimal solution. By iteravtively improving Q-function, it allows the algorithm to find the optimal policy for MDP, which is the policy that maximizes the expected total reward for the agent.

We can define Utility as the approximate cost for a state to reach the desired output. To calculate the utility we can use the U* function which is for any state s, the quantity U* (s) represents thc greatest achievable utility starting in state s under any policy.

The optimal policy can be defined as follows:

$$\pi^*(s) = \text{argmax}_{a \in A(s)} \left[r_{s,a} + \beta \sum_{s'} p^a_{s,s'} U^*(s') \right].$$

Where r is the reward can also be considered as cost wich is basically negative reward while minimizing the policy. Beta is the discounting factor which ranges between $0 < \beta < 1$. Ps,s` is the transition probability from current state to the next state and U*(s`) is the utility of the next transitioning state. Solving π* gives U* and vise versa., Bellman's Equation, a system of recurrence equations that specify the optimal utility function.

$$U^*(s) = \max_{a \in A(s)} \left[r_{s,a} + \beta \sum_{s'} p^a_{s,s'} U^*(s') \right].$$

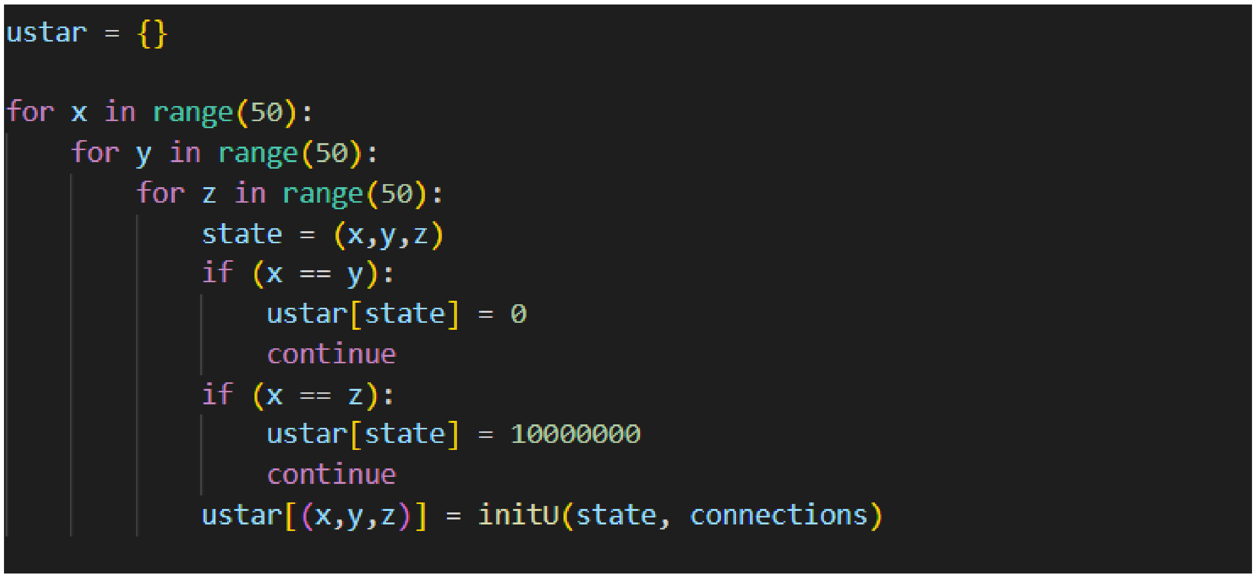
The possibility of solving this system is very difficult as the max function introduces significant nonlinearities to the system.

With value iteration we initialize some initial value of utility for each state, an estimate or guess. Then we iteratively improve this estimate using this formula:

$$U^*_{k+1}(s) = \max_{a \in A(s)} \left[r_{s,a} + \beta \sum_{s'} p^a_{s,s'} U^*_k(s') \right].$$

Q. How does U* (s) relate to U* of other states, and the actions the agent can take?

As we can see from the value iteration formula, the U*(s) at timestamp t + 1 is dependent on the U*(s`) i.e., the U* value of states neighboring to the current state. This can be clearly understood as the U* value of a state is dependent on the U* value of the neighboring state as to traverse from one state you need to know what neighboring state is the best possible state to be in.

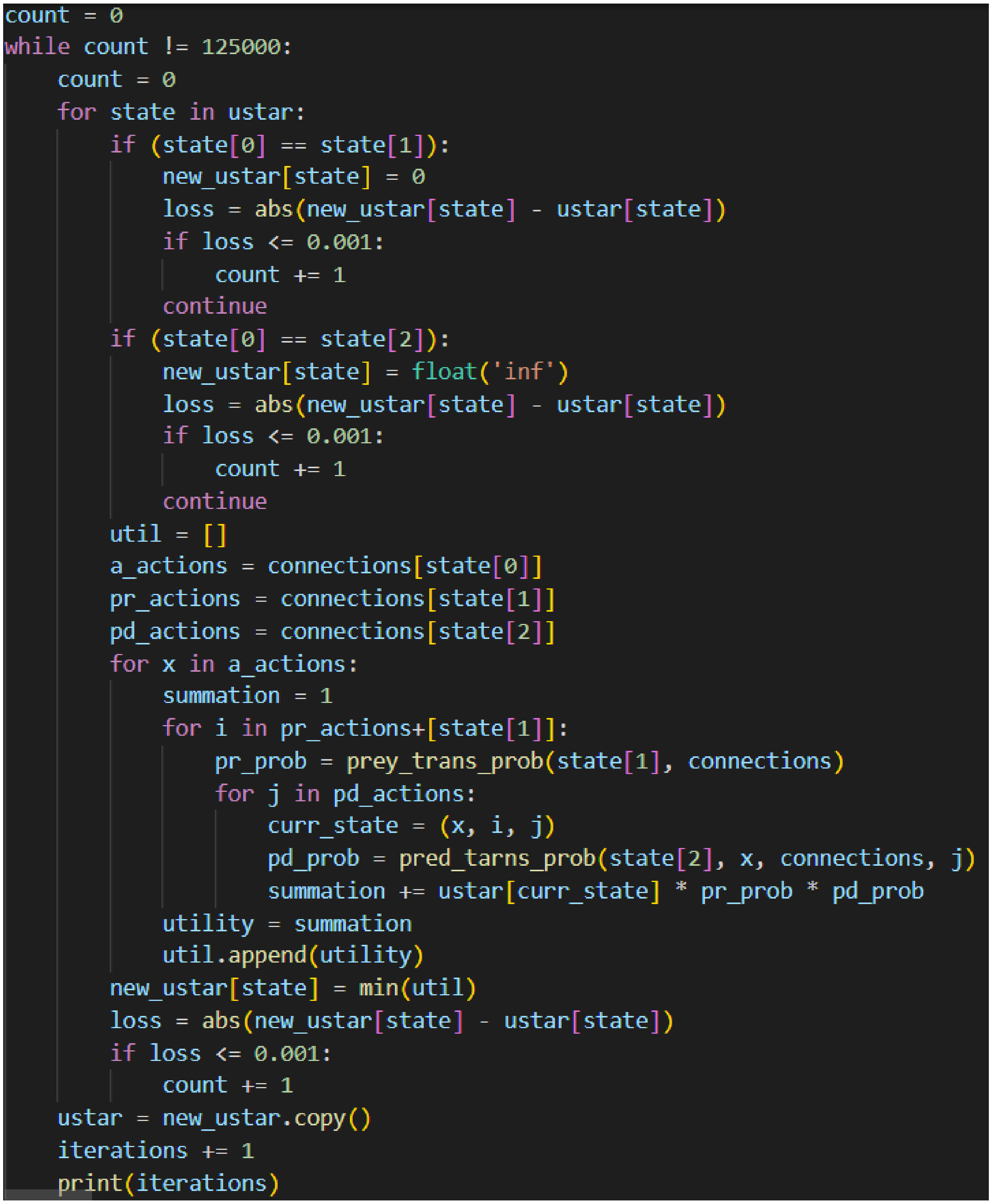


```
ustar = {}

for x in range(50):
    for y in range(50):
        for z in range(50):
            state = (x,y,z)
            if (x == y):
                ustar[state] = 0
                continue
            if (x == z):
                ustar[state] = 10000000
                continue
            ustar[(x,y,z)] = initU(state, connections)
```

Figure - 3: code snippet

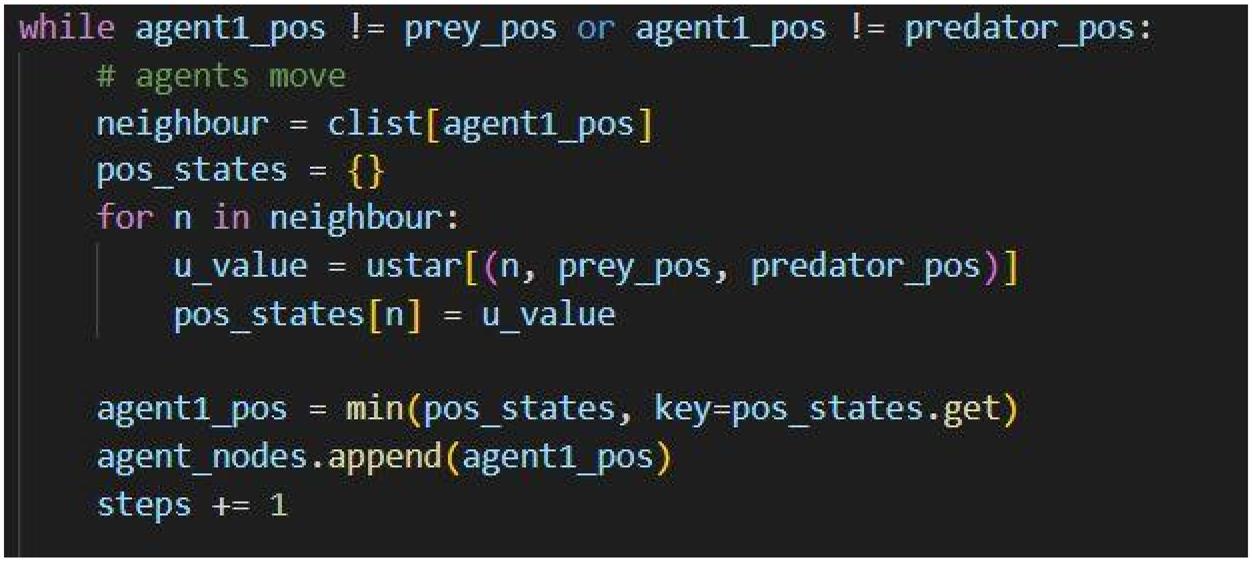
Here we have initialized the initial ustar value for each state as an estimate, which we took as the distance of prey from the agent. Utility of state should basically define the approximate number of steps required to achieve the goal state.



```
count = 0
while count != 125000:
    count = 0
    for state in ustar:
        if (state[0] == state[1]):
            new_ustar[state] = 0
            loss = abs(new_ustar[state] - ustar[state])
            if loss <= 0.001:
                count += 1
            continue
        if (state[0] == state[2]):
            new_ustar[state] = float('inf')
            loss = abs(new_ustar[state] - ustar[state])
            if loss <= 0.001:
                count += 1
            continue
        util = []
        a_actions = connections[state[0]]
        pr_actions = connections[state[1]]
        pd_actions = connections[state[2]]
        for x in a_actions:
            summation = 1
            for i in pr_actions+[state[1]]:
                pr_prob = prey_trans_prob(state[1], connections)
                for j in pd_actions:
                    curr_state = (x, i, j)
                    pd_prob = pred_tarns_prob(state[2], x, connections, j)
                    summation += ustar[curr_state] * pr_prob * pd_prob
            utility = summation
            util.append(utility)
        new_ustar[state] = min(util)
        loss = abs(new_ustar[state] - ustar[state])
        if loss <= 0.001:
            count += 1
    ustar = new_ustar.copy()
    iterations += 1
    print(iterations)
```

Figure - 4: code snippet

Here, as you can see, we are calculating the U*value for each state using the previous value of U* for each action of agent and all the neighbouring state from that action. We take the summation of product of transition probability from current state to that equivalent action state and old U* value of that equivalent action state, then we add reward to that summation to find the utility for that action state and we do this for all other action state. After finding the new U* value for all the action state we take the minimum value from those as the U* value of the current state.


```
while agent1_pos != prey_pos or agent1_pos != predator_pos:
    # agents move
    neighbour = clist[agent1_pos]
    pos_states = {}
    for n in neighbour:
        u_value = ustar[(n, prey_pos, predator_pos)]
        pos_states[n] = u_value

    agent1_pos = min(pos_states, key=pos_states.get)
    agent_nodes.append(agent1_pos)
    steps += 1
```

Figure - 5: code snippet

Here we are deciding the next best move for agent using the ustar (utility) values form the lockup table which is a hash map named Ustar that has values stored, where key is the state and values are the utility values at that state. Agent first figures the action it can take and according to the number of action the agent can take we can figure out the neighboring state of the current/present state. We look for the least utility value at those neighbouring state using the hasmap and decide the next move of agnet accordingly.

Q. What states s are easy to determine U * for?

We know that when the agent and prey are in the same position in a particular state, the agent wins and the game is over. This basically means the cost for the agent to catch the prey in this state is 0. There are 50 nodes in our graph where the agent and prey can be in the same position and the predator can be in any of the 50 positions as we are also considering agent, prey, and predator in the same node to be a success. Therefore, there are 50 * 50 possible states for this scenario, i.e., 2500 states.

Also, we know that when the agent and predator are in the same position, agent losses and the cost for this state can be considered infinite or a very large value in finite terms. This is possible only when the agent and predator are in one node and the prey is in some other nodes as discussed before, the state, where the agent, prey, and predator are in the same node, is considered to be a success. This is only possible for 50 * 49 possible states, as prey cannot be in the same spot as agent and predator, i.e., 2450 states.

So in total, we can say there are 4950 states, which are easy to determine as we can directly assign the values for these states.

Q. Are there any starting states for which the agent will not be able to capture the prey? What causes this failure?

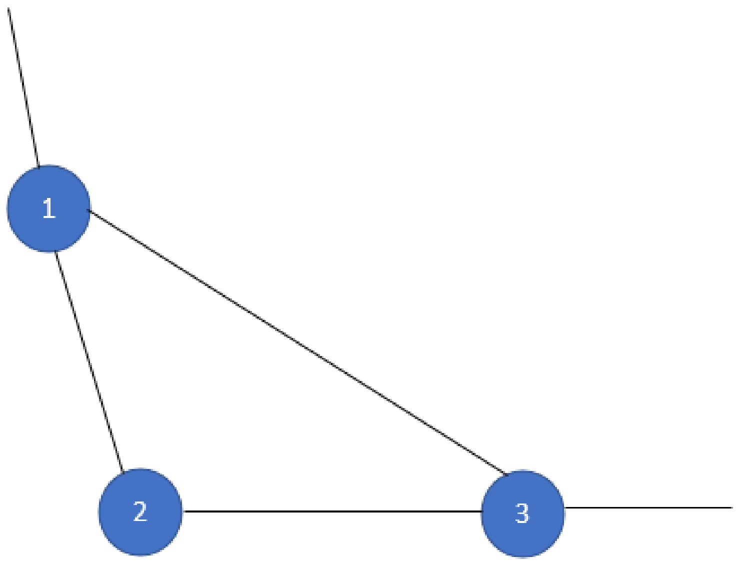


Figure - 6: graph

Yes, consider the above scenario, let's say agent stays in node 2 and the predator stays in node 3, in this case it is obvious that the starting states of the agent will lead to failure since even though the agent moves to node 1, predator is more likely to kill the agent. If the agent stays still in node 2, the predator is still more likely to kill the agent. Hence, the agent would not be able to capture the prey.

Q. Find the state with the largest possible finite value of U *, and give a visualization of it.

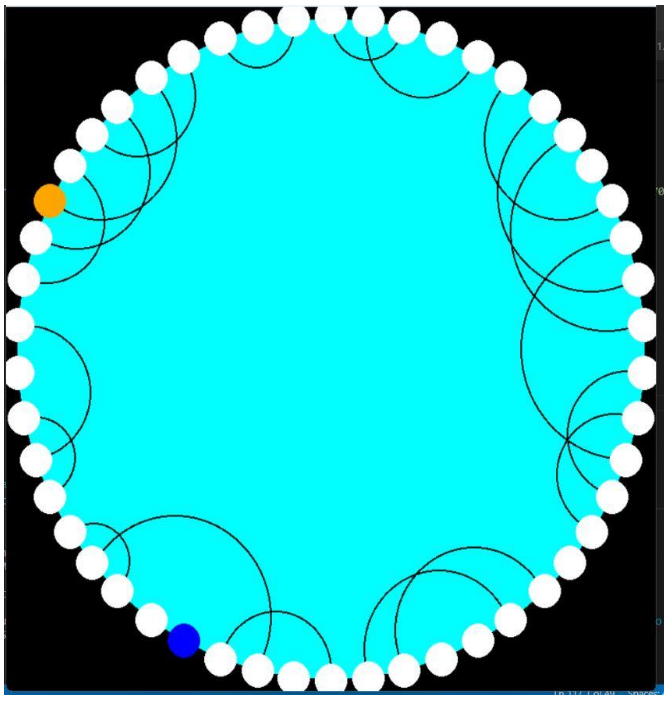


Figure - 7: visualisation

Here prey and predator are in the same node as we have visualized in orange color. When the agent tries to capture the prey and avoids the predator at the same time, it is hindered with a dilemma to whether pursue the prey or move away from the node. This can be one of the reason for the utility value to be the highest that is finite

Q. Simulate the performance of an agent based on U *, and compare its performance (in terms of steps to capture the prey) to Agent 1 and Agent 2 in Project 2. How do they compare?

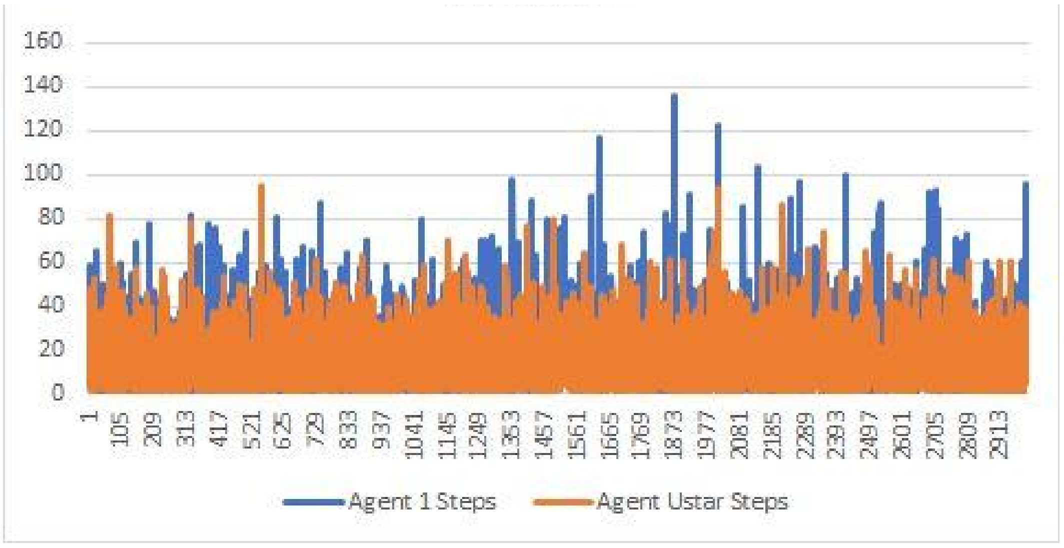


Figure - 8: graph

From the graph, Agent 1's accuracy is 87.4% and U * accuracy is 99.97% all along. In this graph the number of steps taken by both agent 1 and agent U * is comparable. In some cases, we can see that the number of steps of Ustar exceeds the agent 1 steps, that's because the agent 1 is probably dead in fewer number of steps because of it's less accuracy.

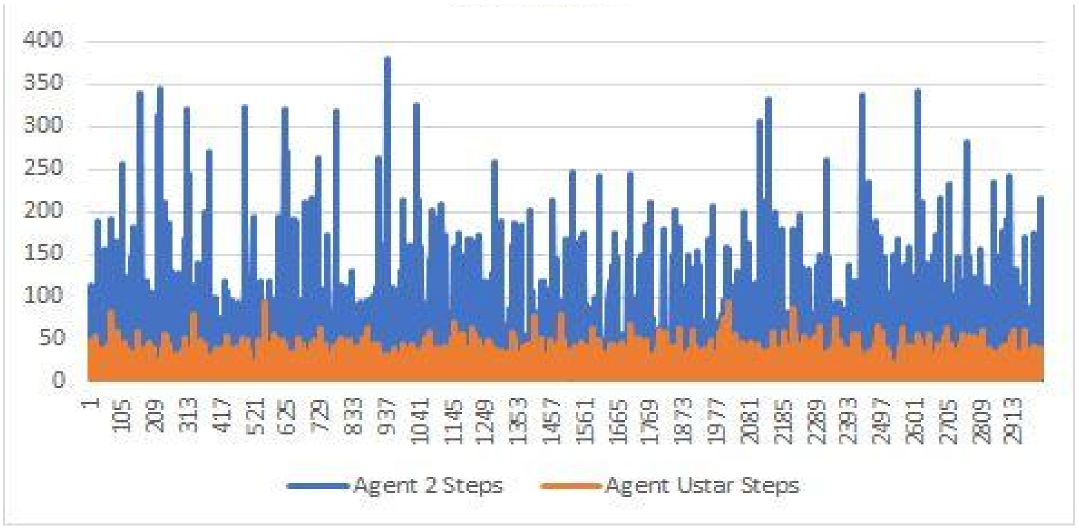


Figure - 9: graph

It’s clear from the above graph that the number of steps taken by agent U* is comparatively very low, whereas agent 2’s steps taken are way high and they both give accuracy of about 99%.

Q. Are there states where the U * agent and Agent 1 make different choices? The U * agent and Agent 2? Visualize such a state, if one exists, and explain why the U * agent makes its choice.

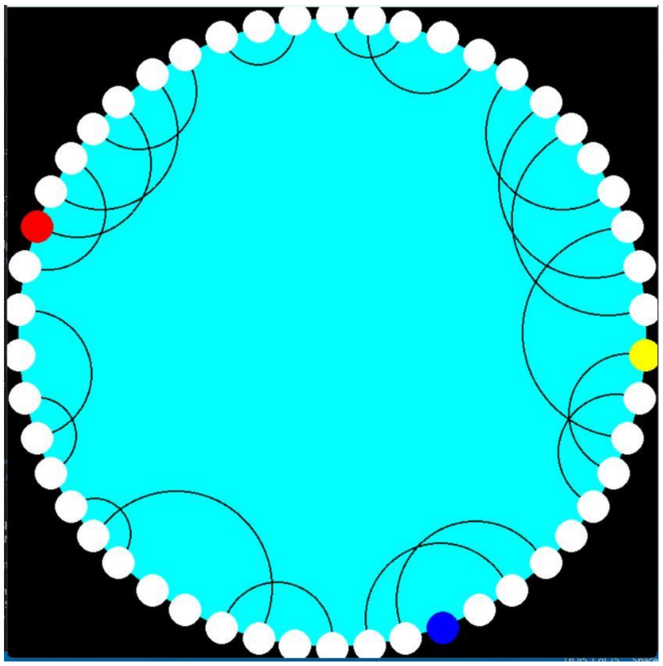


Figure - 10: visualisation

Prey Moves: [14, 13, 12, 7, 6, 7, 6, 7, 6, 11, 6, 5, 9, 10]

Agent Moves Ustar: [34, 33, 32, 27, 26, 25, 24, 23, 19, 18, 17, 16, 15, 10]

Predator Moves: [36, 34, 35, 34, 36, 34, 33, 32, 31, 32, 27, 28, 25, 24]

Prey Moves: [14, 13, 12, 7, 6, 7, 6, 7, 6, 11, 6, 5, 9, 10]

Agent Moves Project 2: [34, 33, 32, 27, 26, 25, 24, 20, 19, 18, 17, 16, 15, 10]

Predator Moves: [36, 34, 33, 32, 33, 32, 31, 32, 33, 32, 27, 32, 33, 31]

Here we can see that when agent_ustar at node 24 takes the decision of going to node 23 instead of going to node 20 which agent2 from project 2 goes to. This could be because the agentagents' to make different decision. It is prompting it to move towards the prey and away from the predator and the other is basically taking the ustar value from the lookup table and identifying which state has less cost associated with it to catch the prey.

4 Modeling V

Q. How do you represent the states s as input for your model? What kind of features might be relevant?

We consider the distances of the agent to the predator, the agent to the prey, and the predator to prey in the model to make sense. In a circular graph, if we assign weights based on node numbers, if the agent is in node 1 and prey is in node 0, the shortest distance between two nodes is 1, but when the agent is in node 0 and the prey is in node 49, the shortest distance between two might show it as 49, despite the fact that the shortest distance is 1. However, to overcome this case, if we consider distances between the agent, prey, and predator, atleast the model would be aware of the positions in which they are lying, hence we are taking distances into account.

Q. What kind of model are you taking V to be? How do you train it?

We considered taking 3 layer Neural network with activation function to be sigmoid to perform supervised learning with distances as inputs and utility value as the labeled output. Since we are using the sigmoid function we have to scale the output down to the values between 0 and 1 and tain the model with that values. While forward propagating we are using sigmoid with some random weights to extract features into the neural nodes and then obtain the output which is then checked for the amount of error in it and then backpropagated to refine the weights in the neural network and the backpropagation is done by using the derivative of sigmoid function. We tried to implement the model using the RELU function but were unable to converge the error. To overcome that, as suggested by few in the discussion panel to use leaky relu, we encountered an issue with the backpropagation part where the error instead of converging was diverging. We decided to stick with the sigmoid function as it was giving – fair to say – decent result. With sigmoid, we were able to bring the error down to 0.02 and when simulating the agent with UStar values obtained from the model we able to maintain the 99.98 % win rate with the previous U* agent.

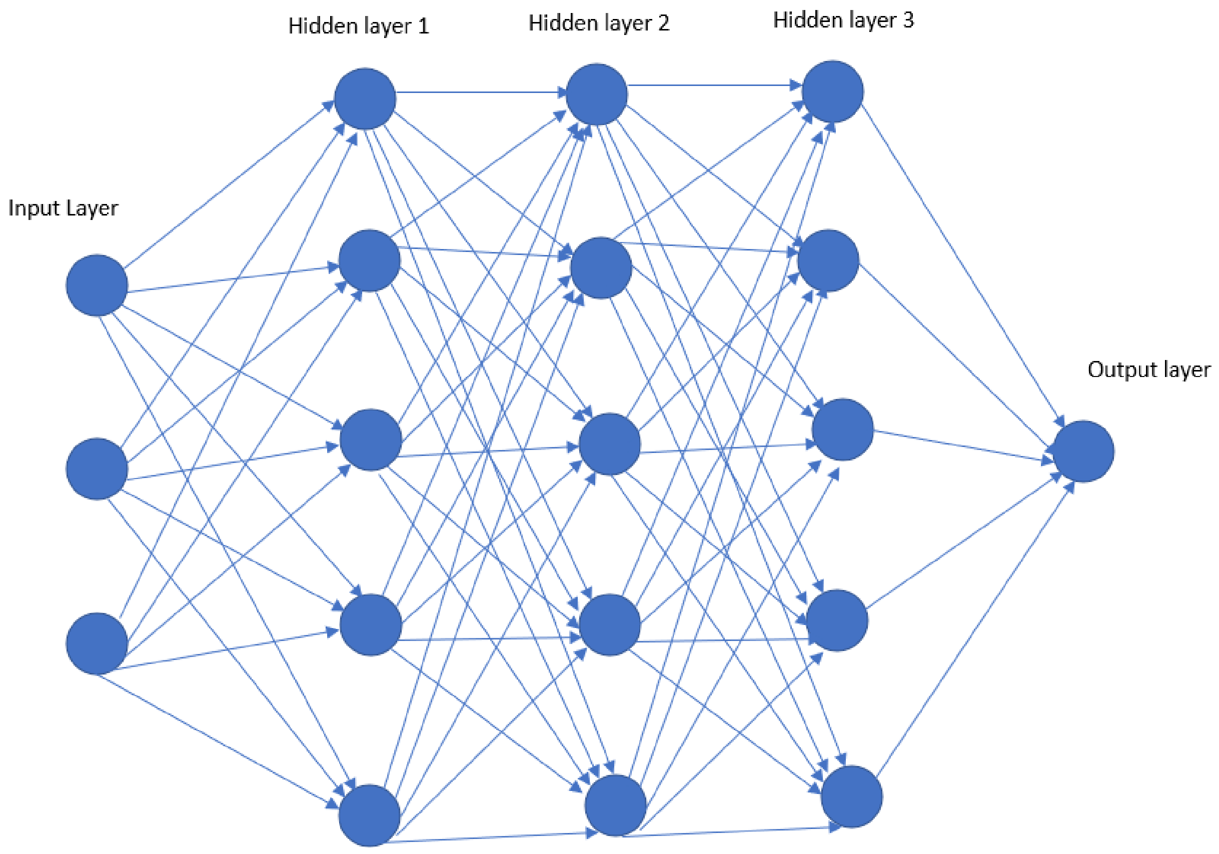


Figure - 11: Neural Network

Q. Is overfitting an issue here?

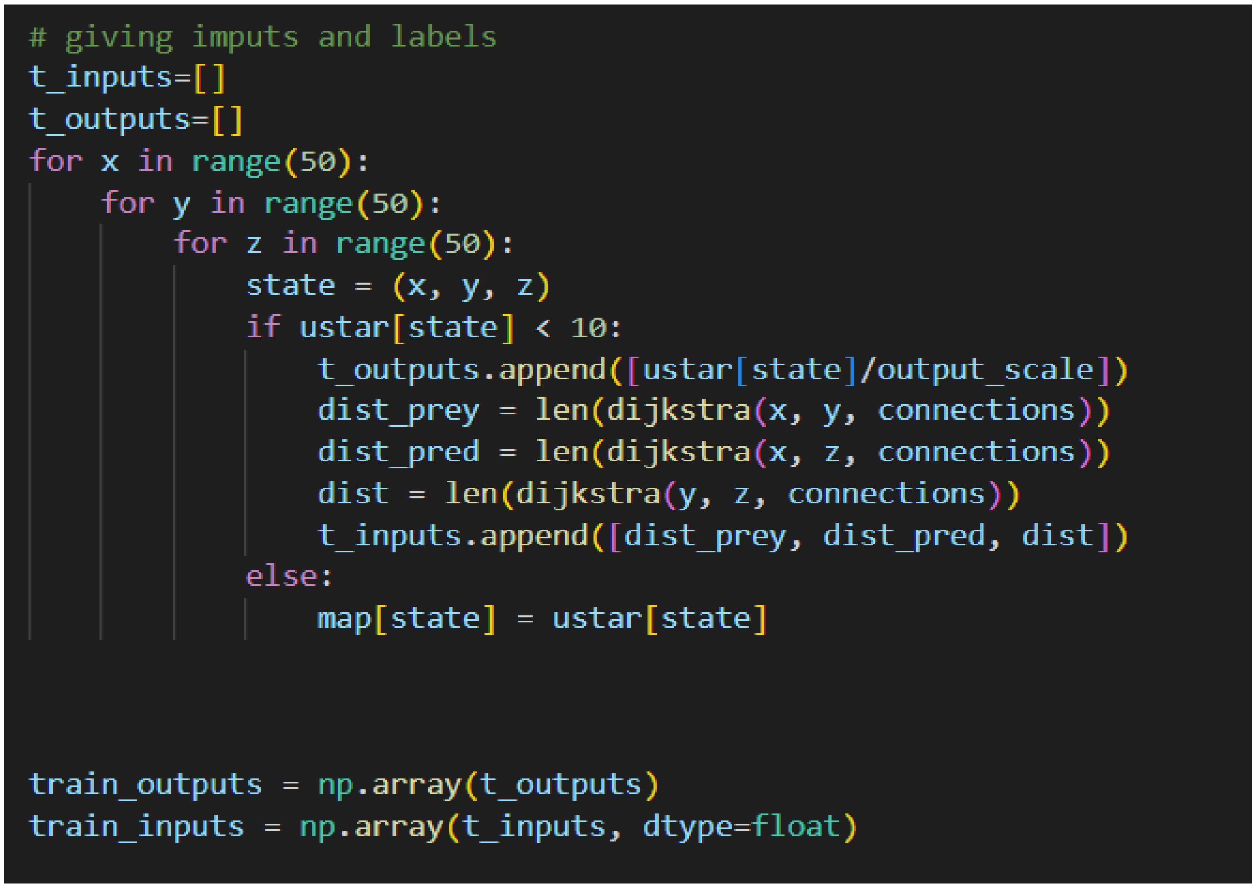
If we were to train a model that finds the utility for any graph of that dimensions then with the model we have trained we would suffer with the overfitting issue. But since the model is only designed to find the utility for a particular graph, overfitting wont be an issue. Even if it extracts features that are not irrelevant to the relationship of inputs and outputs, it will just make model reduce the error furthermore. This is a desired outcome as we are just concerned about obtaining the utility of a desired graph.

Q. How accurate is V

With sigmoid as the activation function and the derivative of sigmoid as the backpropagation function, we were able to bring the error down to 0.02 and get good utility values for the graph that we are using. With the utility values obtained, the agent is performing similarly to what the ustar agent is performing with a 99.5% win rate of the agent.

Q. How does its performance stack against the U* agent?

The performance is comparably similar as the model is able to predict the utility value to the error of only 0.02. This give almost similar results to that of Ustar agent. Another import thing to talk about in this model is that when we give the utility value that are considered to be the forbidden states – Where agent is never supposed to go – the model is not given such a state as it ruins the weight assignment to each nodes and the error increases. Since these states are easy to know we can just check in the look up if such a state has arrived, if so the value is just extrtacted from the lookup instead of modeling a Neural Network that predicts the forbidden states.



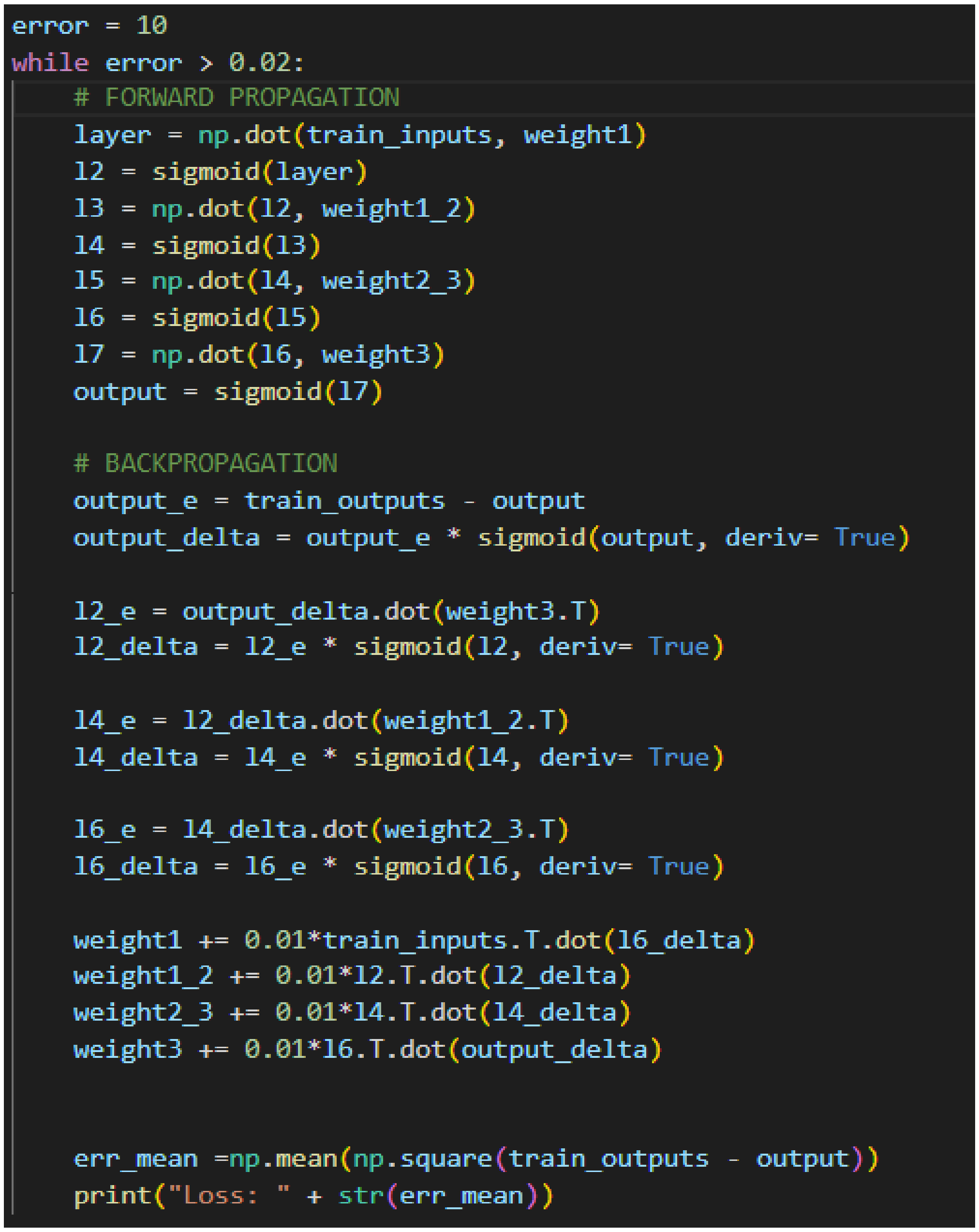
```
# giving imputs and labels
t_inputs=[]
t_outputs=[]
for x in range(50):
    for y in range(50):
        for z in range(50):
            state = (x, y, z)
            if ustar[state] < 10:
                t_outputs.append([ustar[state]/output_scale])
                dist_prey = len(dijkstra(x, y, connections))
                dist_pred = len(dijkstra(x, z, connections))
                dist = len(dijkstra(y, z, connections))
                t_inputs.append([dist_prey, dist_pred, dist])
            else:
                map[state] = ustar[state]



train_outputs = np.array(t_outputs)
train_inputs = np.array(t_inputs, dtype=float)
```

Figure - 12: code snippet

This is how we are generating the inputs for each 125000 states by calculating the distance between each other and sending that as an input to the model and the output labels are also stored in t_ouput as a list with same index to test whether the model is predicting the values accurately or not.



```
error = 10
while error > 0.02:
    # FORWARD PROPAGATION
    layer = np.dot(train_inputs, weight1)
    l2 = sigmoid(layer)
    l3 = np.dot(l2, weight1_2)
    l4 = sigmoid(l3)
    l5 = np.dot(l4, weight2_3)
    l6 = sigmoid(l5)
    l7 = np.dot(l6, weight3)
    output = sigmoid(l7)

    # BACKPROPAGATION
    output_e = train_outputs - output
    output_delta = output_e * sigmoid(output, deriv= True)

    l2_e = output_delta.dot(weight3.T)
    l2_delta = l2_e * sigmoid(l2, deriv= True)

    l4_e = l2_delta.dot(weight1_2.T)
    l4_delta = l4_e * sigmoid(l4, deriv= True)

    l6_e = l4_delta.dot(weight2_3.T)
    l6_delta = l6_e * sigmoid(l6, deriv= True)

    weight1 += 0.01*train_inputs.T.dot(l6_delta)
    weight1_2 += 0.01*l2.T.dot(l2_delta)
    weight2_3 += 0.01*l4.T.dot(l4_delta)
    weight3 += 0.01*l6.T.dot(output_delta)


    err_mean =np.mean(np.square(train_outputs - output))
    print("Loss: " + str(err_mean))
```

Figure - 13: code snippet

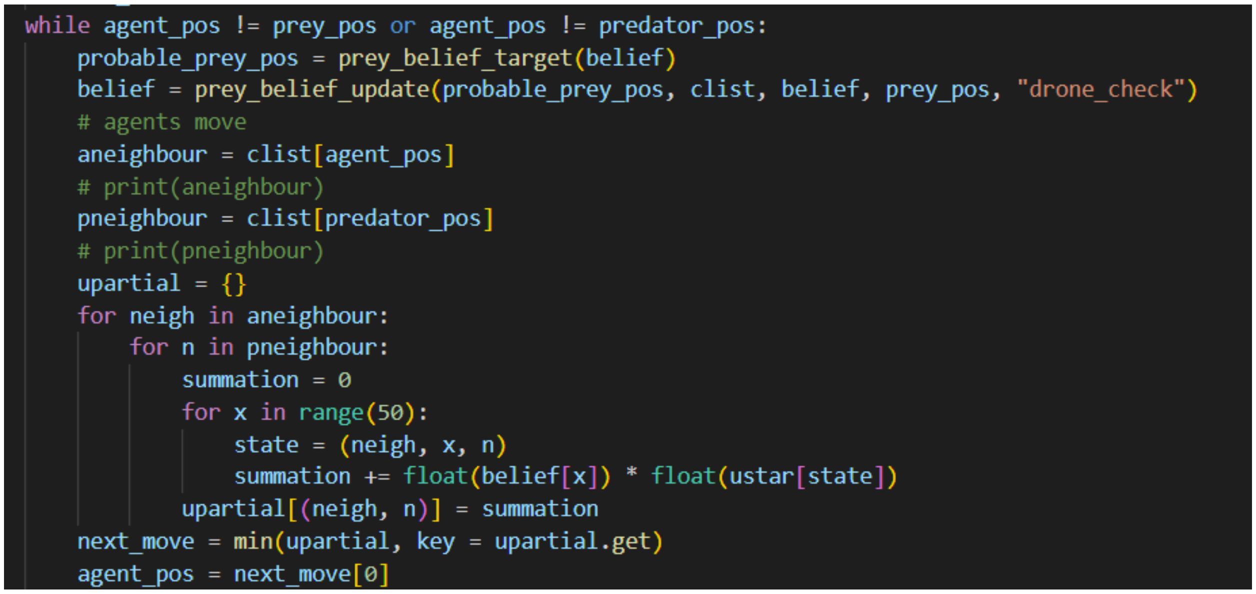
Here we can see that the model consists of 3 hidden layer and 1 input and 1 output layer. Each layer is assigned some random weights initially and as the model keeps back-propagating it keeps on adjusting the weights to the desired value. When the error is minimized to 0.02 we store the desired weights and then use it as trained modeled weights to gain the desired output which in our case is the utility value.

5 U Partial

In this scenario, we dont have the information of where the prey is so we maintain the belief system that predicts where the location of the prey could be using a drone to survey the node. Since we only have the positions of agent and the predator, the permutation of the states is 50 * 50 as we exclude the prey combination. We can estimate the utility of a given state as the expected utility based on where the prey (belief) is based on -

$$U_{\text{partial}}(s_{\text{agent}}, s_{\text{predator}}, \underline{p}) = \sum_{s_{\text{prey}}} p_{s_{\text{prey}}} U^*(s_{\text{agent}}, s_{\text{predator}}, s_{\text{prey}}).$$

Here the utilty of the desired state is multiplied by the probability of prey being there which can be referenced from the belief model. The summation of all 50 possible different neighboring states is then compared to the other summarised action state and the value with least Upartial is then selected as the next action for the agent. This is simultaneously done with the agent, prey, and predator simulation and the values of Upartial are generated in real-time.



```
while agent_pos != prey_pos or agent_pos != predator_pos:
    probable_prey_pos = prey_belief_target(belief)
    belief = prey_belief_update(probable_prey_pos, clist, belief, prey_pos, "drone_check")
    # agents move
    aneighbour = clist[agent_pos]
    # print(aneighbour)
    pneighbour = clist[predator_pos]
    # print(pneighbour)
    upartial = {}
    for neigh in aneighbour:
        for n in pneighbour:
            summation = 0
            for x in range(50):
                state = (neigh, x, n)
                summation += float(belief[x]) * float(ustar[state])
            upartial[(neigh, n)] = summation
    next_move = min(upartial, key = upartial.get)
    agent_pos = next_move[0]
```

Figure - 14: code snippet

While simulating the agent, prey, and predator in the graph, we are first finding the neighboring states to the current state by considering the values of agent position and predator position that gives us neighbour_agent x neighbor_predator different possible states with prey position unknown. After finding Them we take the summation of all 50 possible additional states the prey can be in and multiply each state with the probability from the belief system. Then we compare each of the action states and find which one gives lowest cost.

Q. Simulate an agent based on Upartial, in the partial prey info environment case from Project 2, using the values of U ∗ from above. How does it compare to Agent 3 and 4 from Project 2? Do you think this partial information agent is optimal?

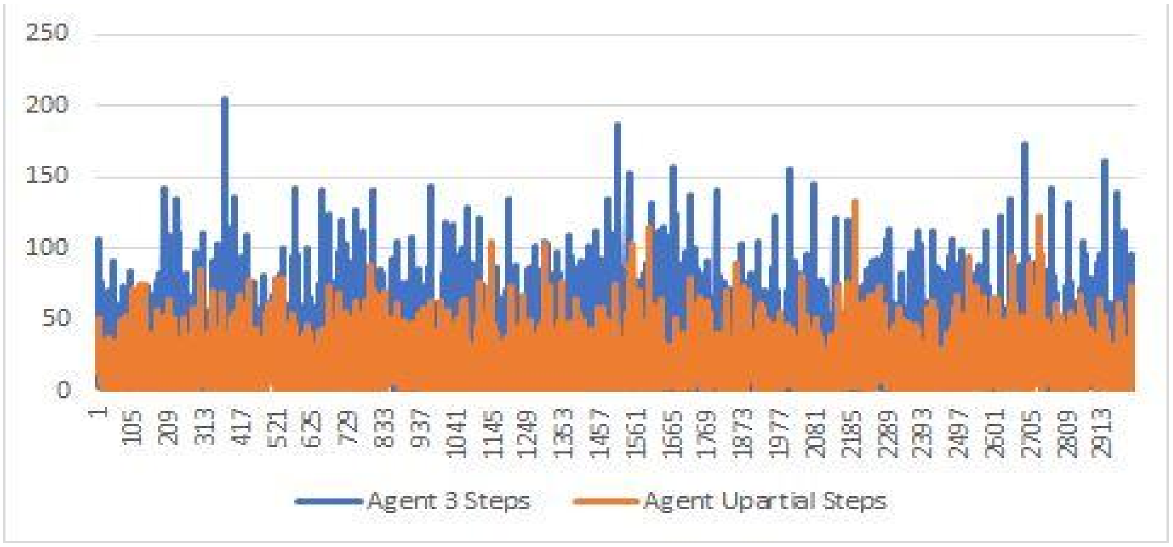


Figure - 15: graph

Agent 3’s win rate is 89.53% which is lower than agent Upartial. Agent 3 takes more steps as compared to Agent Upartial as we can clearly see from the above graphs. Agent Upartial is accurate with less number of steps compared to agent 3.

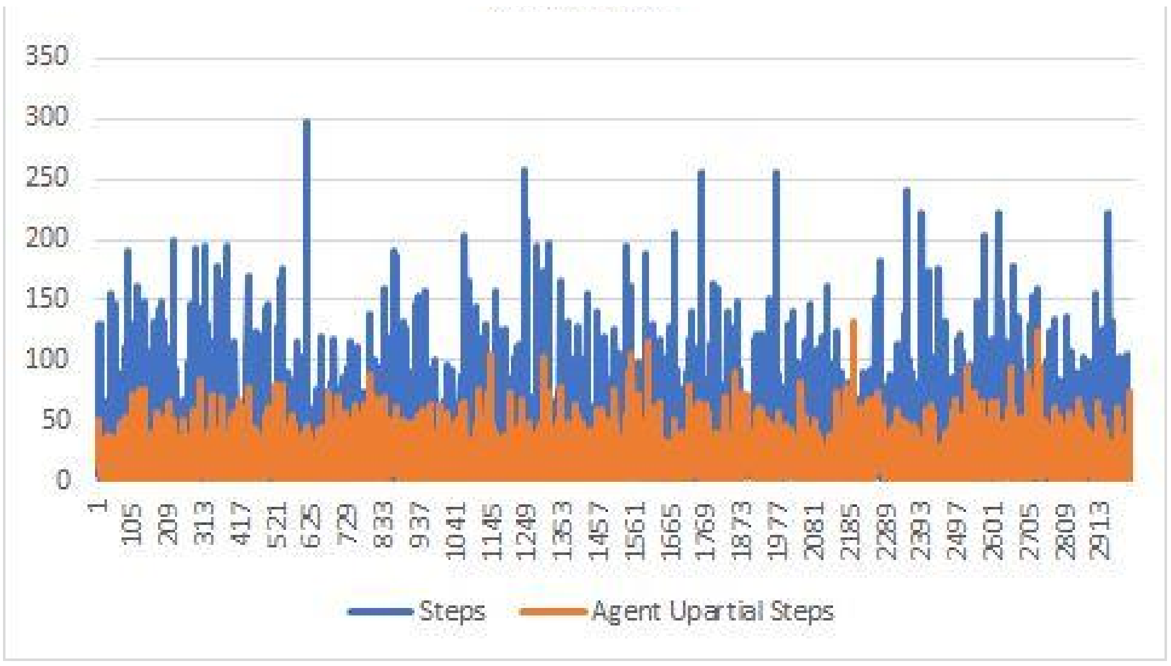


Figure - 16: Graph

Here agent 4 steps are higher than agent Upartial steps as we can see clearly. Agent Upartial is more accurate with less number of steps compared to agent 4.

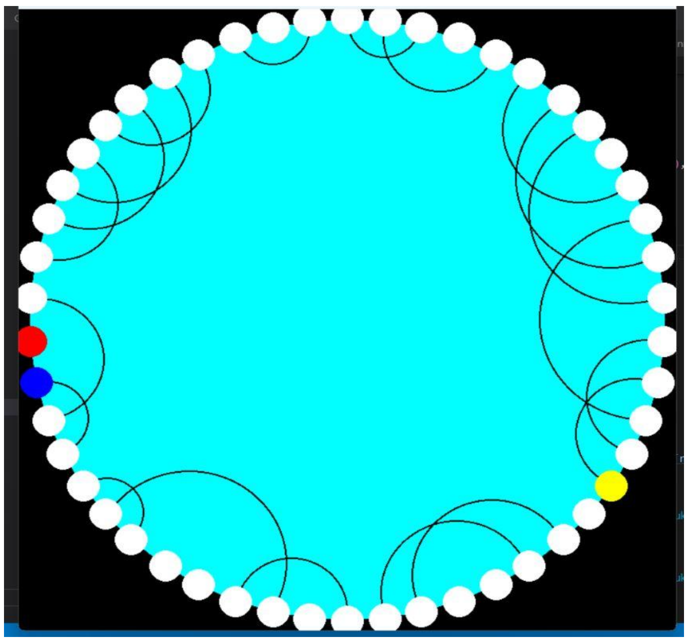


Figure - 17: visualisation

Prey Moves: [13, 14, 13, 12, 7, 6, 11, 10, 15, 14, 13, 12, 11, 12]

Agent upartial Moves: [22, 23, 19, 18, 17, 16, 13, 12, 11, 10, 15, 14, 13, 12]

Predator Moves: [40, 39, 38, 35, 36, 37, 36, 34, 33, 32, 27, 32, 31, 32]

Prey Moves: [13, 14, 13, 12, 7, 6, 11, 10, 15, 14, 13, 12, 11, 12, 11, 10, 9, 10, 11, 10, 9, 5, 6, 11, 10, 15, 10, 11, 12, 11]

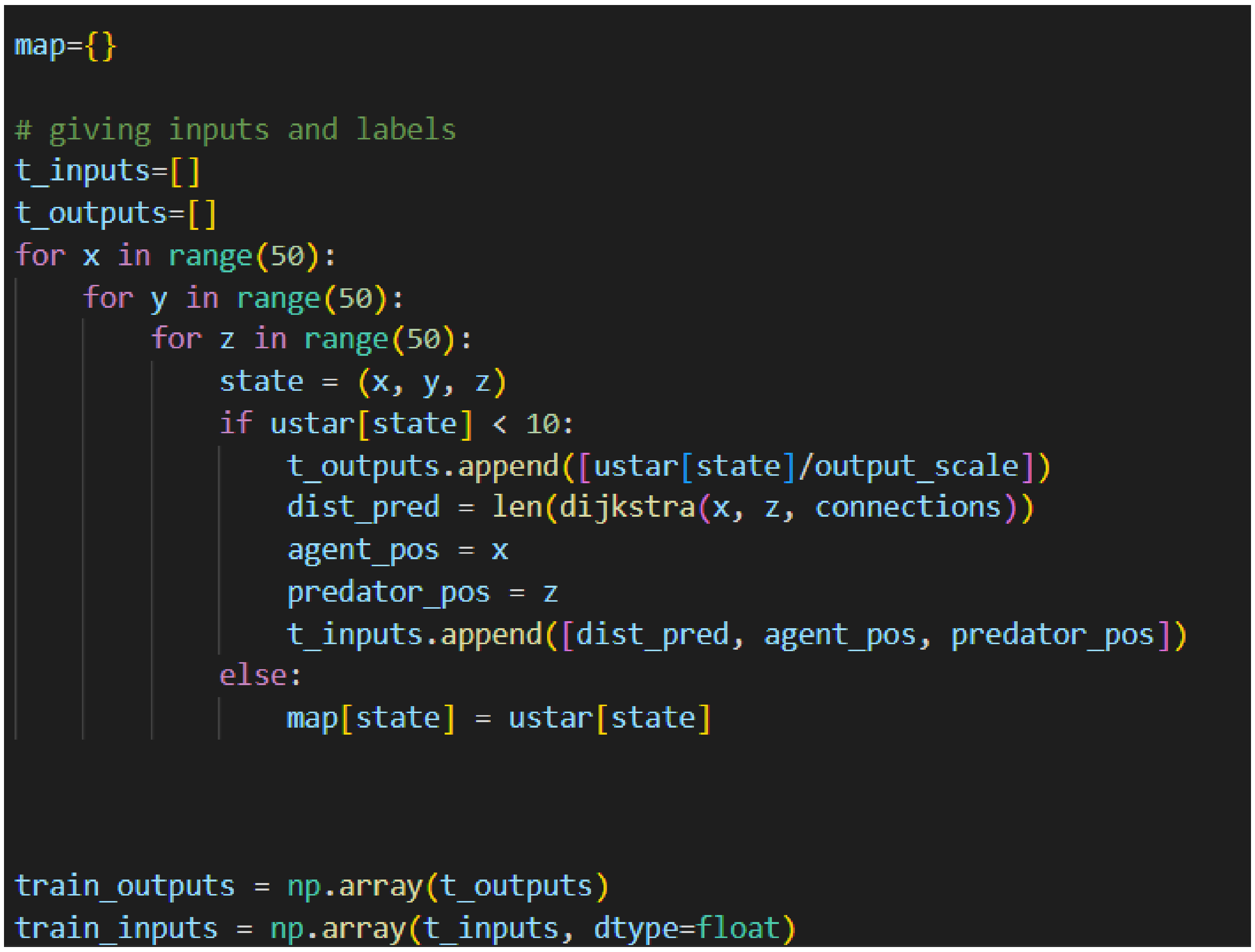
Agent 3 Moves: [22, 21, 22, 21, 22, 21, 22, 21, 22, 21, 22, 21, 22, 21, 22, 21, 22, 21, 22, 23, 22, 21, 20, 19, 18, 17, 14, 15, 10, 11]

Predator Moves: [40, 39, 38, 39, 38, 35, 36, 34, 33, 34, 33, 32, 27, 26, 27, 26, 25, 24, 20, 19, 23, 22, 21, 20, 24, 25, 24, 20, 19, 18]

Here we can see that the movement of prey and predator are completely different from the second step. This is majorly because the upatrtial agent is majorly concerned about the utility and not where the prey and predator are whereas the agent 3 is concerned about the distances from prey and predator. Since Upartail agent is taking optimal states as the next step will reach the prey faster as compared to Agent 3.

6. V Partial

In this, we are generating a model that can predict the Upartial values, which are infinite as the number of state is constant but the position of prey is unknown. To do this we are using neural network with the sigmoid function as the activation function and the derivate of it as the backpropagation function. Similar to V we then find the weights as the error keeps convering. In this model the error convered to no lower than 0.06 and hence the model gave uprtaiul which dose not perform optimally and has a win rate of 87%



```
map={}

# giving inputs and labels
t_inputs=[]
t_outputs=[]
for x in range(50):
    for y in range(50):
        for z in range(50):
            state = (x, y, z)
            if ustar[state] < 10:
                t_outputs.append([ustar[state]/output_scale])
                dist_pred = len(dijkstra(x, z, connections))
                agent_pos = x
                predator_pos = z
                t_inputs.append([dist_pred, agent_pos, predator_pos])
            else:
                map[state] = ustar[state]



train_outputs = np.array(t_outputs)
train_inputs = np.array(t_inputs, dtype=float)
```

Figure - 18: code snippet

Here we have just changed the inputs from the previous V model to agent and predator position and distance between predator and agent. We tried playing with different number of layers and different activation function but we unsuccessful with implementation. Sigmoid function was always performing better when scaled down and was giving a better accuracy for the simulation. The error only converged till 0.06

Q. How do you represent the states s_{agent}, $s_{predator}$, p as input for your model? What kind of features might be relevant?

We consider the distances of the agent to the predator, the agent location, and the predator location. By inputting belief we were unable to get the error converged. Due to that fact we tried implementing the model with input as distance of agent to predator, agnet location and predator location.

Q. What kind of model are you taking Vpartial to be? How do you train it?

We considered taking 3 layer Neural network with activation function to be sigmoid to perform supervised learning with distances as inputs and utility value as the labeled output. Since we are using the sigmoid function we have to scale the output down to the values between 0 and 1 and tain the model with that values. For this model, we take 3 inputs, has 5nodes5 nodes in all 3 hidden layer, and one output node. While forward propagating we are using sigmoid with some random weights to extract feature into the neural nodes and then obtain the output which is then checked for the amount of error in it and then backpropagated to refine the weights in the neural network and the backpropagation is done by using the derivative of sigmoid function.

Q. Is overfitting an issue here? What can you do about it?

Unlike the case of Model V, we don't train this model with all possible input states. This is because depending on the the belief distribution of prey, we can have infinite possible utilities and training them on all is not possible. So, if we overfit the model, the model will not work properly for other states that the model wasn't trained with

Q. How accurate is Vpartial? How can you judge this?

We are able to converge the error to 0.06 so the values of utility obtained are somewhat similar to the upartial but the win rate of the simulation is a bit lower than upartial agent. We are getting accuracy in the range of 85%**.** We can judge the accuracy by find the deviation in the value estimated by model and the value we calculated by equation 1.

Q. Is Vpartial more or less accurate than simply substituting V into equation (1)?

With the model that we tried to simulate, it seems that it is less accurate than substituting V into Equation (1) as the model V which we trained had really good convergence to the error and it will give better result than the Vpartial model. The error convergence of Vpartail is around 0.06 as compared to that of V which is 0.02.

Q. Simulate an agent based on Vpartial. How does it stack against the Upartial agent?

After running the simulation 3000 times, we get the value of win rate for Vpartial to be in the range of 80-90% whereas the accuracy of win rate for Upartial is way better with the accuracy of the win rate to be around 99%.