Project 3: Circle of Life

Course: 16:198:520

Instructor: Professor Cowan

Michael Neustater (mdn68)

Purpose:

This project attempts to improve on the previous project by computing the minimal expected number of rounds to catch the prey for every possible state (U*), resulting in the ability to make the optimal decision at every possible state. I then attempt to reduce the size requirement of storing all possible states by modeling U* using a neural network (V*). The goal then is to use the values from U* to make an approximation for partial prey scenarios called U partial and a subsequent model of it called V partial,

Notes on executing the code:

Main.py ***must*** be run on Python 3.10 in order to import the precomputed pickle files due to changes in how pickle works in this version of Python. If you are using an older version of Python you may run create_new_env.py to generate new pickle files, however this will result in recreating U* and retraining V* and V Partial which will take several hours. If you would like to re-run the visualization code, some data must be regenerated using generate_v_partial_training_data.py.

Introduction

Since my even agents in project 2 attempted to approximate an infinite time horizon utility, their longer explanations were included (it is distinctly different from the utilities used in U*)

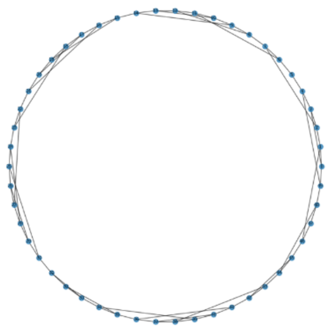
This decision-based catching and avoiding game is deployed over two information settings:

1. The complete information setting, where the agent knows the exact position of the prey and predator at every timestep in the environment.

2. The partial prey information setting, where the agent always knows the exact position of the predator but does not have information about the prey and surveys the node of highest belief for the prey

To implement this project, I use the following technologies:

1. Programming Language: Python
2. Mathematical Equations handling package: NumPy
3. Visualizations: 'Matplotlib'
4. Animations: 'PyGame'



The Environment

The environment is the same as the previous project, with 50 connected nodes in a circle represented by 50 instances of the node class. Each node in the circle is connected by edges and additional edges are generated at random between nodes with a distance of 5 or less. The max degree of a node is 3 with no duplicates.

The Agent

The Agent's goal is to catch the prey while avoiding the predator. For the partial knowledge variation, the agent keeps track of the prey using a belief vector. The agent is guaranteed to spawn in a non-game-ending position (prey or predator positions).

The logic for survey updates is as follows:

P(A is at M | Result Survey N)

=P(A is at M, Result Survey N)/P(Result Survey N)

*=P(A is at M) * P(Result Survey N | A is at M) / P(Result Survey N)*

*=P(A is at M) * P(Result Survey N | A is at M) / Sum of Probabilities*

*=P(A is at M) * P(Result Survey N | A is at M) / 1 - removed probability*

These rules are carried out by the ***survey()*** and ***agent_moved()*** functions inside of each agent class, which handle normalization based on new knowledge

The logic for transition updates is as follows:

P(A is at M after move)

= P(A is in M now) P(Prey in M Next | P in M now) + P(A is in N now) P(Prey in M Next | P in N now) +

P(A is in Z now) P(Prey in M Next | P in Z now) …

This is carried out by the **transition ()** using the dot product of the belief vector and the prey transition matrix.

prey_probabiility_ array:

- initializes to 1/49, since there is an equal probability that the prey is contained by any node except the initial agent node.
    - Transition Matrix for each node N, the edge indexes corresponding to connected nodes and itself are set to 1/(degree of N + 1). (The value is (degree of N + 1) since the edge to itself symbolizes staying still and does not actually exist, therefore it is not included in the degree).

The Prey

The prey is a goal that keeps changing its position in the environment it moves uniformly at random each time the agent moves among its neighbors or just stays in the same place. This is managed by the prey.py file.

The Predator

Predator is the actor in the project that defines the failure of a simulation. The predator is a failure node that keeps changing its position in the environment and has the goal of catching the Agent. It always has the information about the Agent's position. The predator is distracted and will only move towards the Agent's position with a probability of 0.6 for any given timestep and moves to any of the neighboring nodes with the remaining probability of 0.4. This is managed by the predator.py file.

Project Design

Testing:

All testing consists of 3000 runs on 1 environment (for this case the environment referred to is only the collection of edges and nodes not actor positions). The starting positions will be randomly chosen prior to a run. For each run all agents will start in the same initial position for a given run, with the same starting position for prey and predator. After this initial state, the movement will deviate due to randomness driving the prey movement and the dependency of predator movement on the agent position.

End Game States:

The end game states are the same as the previous project, death if collision with predator, win if collision with prey before predator and timeout if more than 5000 steps are taken.

Previous Agents:

Agent 1

Agent 1 is always aware of both actors' positions and bases its movement off these set conditions, where a lower number condition is prioritized.

1. Choose neighbors that are closer to the Prey and farther from the Predator.

2. Choose neighbors that are closer to the Prey and not closer to the Predator.
3. Choose neighbors that are not farther from the Prey and farther from the Predator.
4. Choose neighbors that are not farther from the Prey and not closer to the Predator.
5. Choose neighbors that are farther from the Predator.
6. Choose neighbors that are not close to the Predator.

When there is more than one neighbor that satisfies the highest possible condition, then the agent must choose an optimal one among them. These conditions are implemented in Get_optimal_Node.py

Agent 2

Agent 2 attempts to assess the utility of each next possible move it can take by considering the combined positive utilities of moving towards the prey and negative utility of moving towards the predator and takes the move of highest combined utility. The logic for determining belief is Bellman inspired and will be further touched upon in the improvement logic section. The in-depth explanation was copied from the previous project since this agent functioned using approximated infinite horizon utility.

Improvement logic

The bellman equation can be defined as $V(s) = max\, a(R(s,a,s') + \gamma \sum_{s'} P(s,a,s') + V(s')$, where γ denotes some decay value, R denotes the reward of taking action a to go from s to s' and P(s,a,s') is the probability of traveling from s by action a to s'. The agent considers the prey to have a reward of 1 and the predator to have a reward of -1 and uses a constant decay factor of 0.9. These values are used to create two separate utility arrays, one for the reward of going near the prey and one for the penalty of going towards the predator. Initially the utility array is set to 0 except for the position of the prey at node N which is 1. According to the bellman equation, the utility this provides to all adjacent nodes is max of the set {0, 0.9 * (1)}. Furthermore, for all nodes adjacent to those nodes are the utility provided is max of the set {nodes current values, 0.9 * 0.9} and so on. Since an agent can only pick adjacent nodes or itself as an action, the calculations for utility are only considered for adjacent cells. Nodes part of the shortest path will always have the highest utility, since they will be multiplied by the decay value fewer times. Therefore, the utility of each adjacent node can be easily simplified as $0.9^{distance\ from\ prey\ to\ action\ node}$. Such that a distance of 0, results in a direct reward of 1, and each increase in distance has a decay of 0.9. This same logic can be used for the predator however these utilities are detrimental (negative values). The predator utility for each action is then subtracted from the prey utility so that a new combined utility is created. Whichever combined utility has the largest value is the next action chosen by the agent. The resulting utilities from these calculations are only an approximation since they create the decay based of the assumption the actors stay still and do not consider the potential of the rewards moving at each timestep. This method is much faster and since the utility is calculated so frequently (which becomes more important in partial information settings). This increases in accuracy when they are closer to the agent (since they have less potential to move from their current position in the few steps the utility is calculated). Ultimately accurate utility is most important when the reward (prey) and penalty (predator) are nearby since these actions have a higher chance of ending the game.

Agent 3

Agent 3 does not know the position of the prey and makes use of a belief vector to keep track of the probability that the prey is at a certain node. It attempts to move towards the node with the current highest belief using the rules of Agent 1.

Agent 4

Agent 4 abides by the same rules as Agent 3 and follows the same belief logic to keep track of the prey and builds upon the ideas from Agent 2 but adapts them for the partial information setting of the prey.

Improvement logic

Since the exact value of the prey is unknown, for each possible next action, the agent finds the hypothetical utility of the prey from each position in the environment, using $0.9^{distance\ from\ node\ to\ action\ node}$ then multiplies it by the belief vector value that the prey is in that node. The agent multiplies by the belief in order to weight the utility the prey would give by being in that node, such that highly probable nodes have a much larger influence on utility. These utilities are then combined to create a composite prey utility for a given next action node. Since the predator locations are known, the predator utility is calculated in the same way as agent 2 and is subtracted from the new composite prey utility for each next possible action node. The next action node with the highest utility is chosen as the next agent position.

U*

U* is a finite time horizon MDP for the complete information setting implemented in the *u_ star* class. There are 50 possible positions for prey, predator and agent making for a total of 50^3 distinct states. These states are represented in U* by a 3D array called the utility_matrix. U*(s) for predator position = i, prey position = j and agent position = k is utility_matrix[i][j][k], where the values of each state U*(s) equals the minimum expected rounds to catch the prey.

The easiest states to determine for U* are states where the agent is in the same position of the predator, which is infinity since the game cannot be won, and states where the agent is in the same position as the prey (and not the predator) which is 0 since the game has been won. These states can be thought of as the base cases since they are known prior to the first iteration and result in no more actions that can be taken. Another easy to calculate state are those immediately adjacent to a prey state where predator does not share a position with the prey, since the game can be won in one move the value of the state is 1.

For the remaining states the value of U* is as follows according to the bellman equation:

$$U^*(s) = \min\ action(\,1 + \sum_{s'} P(s') + U^*(s'))$$

With max action changed to min action since the desired value of U* is the minimum expected rounds and the reward is always 1 since all actions result in 1 expected round (step). Since this is a finite time horizon, and the value wanted is minimum number of rounds, there is no decay value.

The values for these states are created using value iteration such that all states are initialized to zero and updated at each subsequent iteration the value of every state *s* is updated using the *s'* values found for U* in the previous iteration (except for the easy to determine states mentioned above).

The possible actions at state *S* are staying still or moving to any connected node. The value of an action is the sum of the utility each possible result of that action, U*(s'), times the probability that result may happen, P(s'). In this case, the potential outcomes are all possible combinations of prey and predator movements, and the probability of these outcomes are based on the combined probability of that prey and predator transition happening.

In the case of prey, the possible outcomes are moving uniformly randomly or staying still which is equal to the degree of prey node + 1 and the probabilities of each are all the same (1 / degree of prey node + 1).

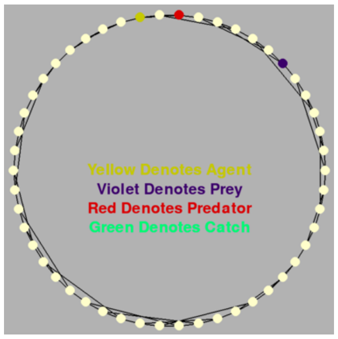
For the predator if it moves distracted it will move uniformly at random to any of the neighbors each of which has a probability of (0.4/degree of pred node) and if it moves focused it will move to a node that is closest to the agent with a probability of (0.6/number of closest nodes to new agent position) for each.

The total number of possible outcomes for an action is (degree of prey node + 1) * ((degree of pred node) + (number of closest nodes to new agent position)) and the probability for that outcome is P(of that prey state occurring from that action) * P(of that predator state occurring from that action).

Value iteration terminates when the maximum difference between the previous iteration and the current iteration is approximately 0 (1e10^{-8} is used as the threshold for 0). At that point an additional sweep of all the values occurs to create the policy based off the max possible action of each state. This is then stored in the *policy_ matrix* which uses the same lookup pattern as the *utility_ matrix.*

There are some states where the prey is initialized in a way in which survival is not guaranteed. An example is when the agent spawns in a node of degree 2 that is adjacent to the predator. If the predator's node is connected to the other node adjacent to the agent, then both the action of staying still or moving to the adjacent node could result in death. Since the predator occasionally moves in a distracted way, it is possible for the agent to survive in states like this, however it cannot be certain. These states are dependent on the environment and not all environments may have them.

The maximum non-infinity value recorded in the U* used for these results was 19.427 rounds expected. This is visualized in the picture to the right. The agent must take so many steps despite not being far from the prey because the predator between the two and the nodes between the prey and agent are highly connected meaning that in many of these nodes the agent cannot be certain of survival. The agent must move to a node where it is fully confident it can pass the predator without colliding in order to make it to the prey or go around the other side of the circle.



The agent functions by looking up the next best possible action that is stored in the policy matrix, given the current state and acts according to it until the game ends.

U Partial

U partial uses the information gathered in U* to determine the value of a state *s*. Since there are infinitely many possible states due to the use of belief vectors, the guidelines suggest approximating it at runtime. The value of a state *s* given some belief vector *v* can be approximated by executing U* of an action for every possible prey position. This means that instead of only considering all combinations of next possible prey states for one prey position, the next possible prey state from every prey position must be considered. The suggested equation to determine this is:

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

Such that s_{prey} 1 of 50 possible current state positions of the prey and $p_{s_{prey}}$ is equal to Belief(prey is in that node).

Since this is only an approximation, the partial information agent created using U Partial is not optimal but should be closer to optimal than agents 3 and 4. In order to create an optimal agent an infinite number of states would have to be considered since the belief vector elements can take an value.

At each timestep, the agent calls a helper method which determines the best possible action (where it can move to) based on the rules above. It determines the utility of an action by iterating through the belief vector and multiplying the belief of the prey being at that node times the U* of the prey being in that node. It then sums all the values to get an action value and picks the action with the lowest value (fewest number of expected rounds).

V*

V* (implemented in the *v_ star* class) consists of a neural network which attempts to use the information gathered in U* in order to predict the values of U* for a state *s.* The input to the NN is a one-hot vector of size 150. The vector can be broken into three sub vectors of size 50, where the index of each sub vector index corresponds to a node in the environment. The three sub vectors each represent the state of predator, prey, or agent. Meaning that for the predator sub vector, all indexes are 0 and the index representing the predator node is one. This holds true for the prey and agent.

This input was chosen with the hope that the model would be able find relationships between transitions and distances without having to explicitly input those features, since the vectors indices are able to represent positions in the environment. It was also chosen with the hope that it would be relevant for the V Partial (explained more in the V Partial section).

The model consists of an input layer, two hidden layers which use tanh activation functions and an output layer leaky ReLU. I found this combination worked best for me as other combinations resulted in overflows or exploding gradients. The hidden layers consist of 150 nodes to allow for adequate complexity in modeling since the input layer is large and the inputs are relatively small. The final layer consists of 1 node so that the output can be given as a single float value. Error is calculated using MSE of the output and expected output. Gradient descent is executed on each input point separately and sequentially, but the data is shuffled on each iteration to reduce biasing towards the end of the set.

The weights and biases of the network are initialized randomly as values between -0.5 and 0.5. For each "iteration" of the network, the class shuffles the input set then loops through it and performs forward propagation then backward propagation and outputs the MSE of the prediction of the entire set. This method was used instead of mini-batches which tended to not quickly decrease the loss in my implementation. The result is that an iteration takes longer (about 2 hours for 200 iterations) but the loss decreases faster.

The forward propagation takes the form of:

$$z_j^l = \sum w_{ij}^l \, out^{l-1} + b_j^l$$

$$out_j^l = \sigma(z_j^l)$$

Which takes the matrix form:

$$z^l = (w^l)^T out^{l-1} + b^l$$

$$out^l = \sigma(z^l)$$

For the first layer out^{l-1} is the input x.

According to the chain rule, back propagation is as follows:

$$\frac{dL}{dw_{ij}^l} = \frac{dL}{dz_j^l} \cdot \frac{dz_j^l}{dw_{ij}^l} = \Delta_j^l \cdot \frac{dz_j^l}{dw_{ij}^l} = \sum_k wj_k^{;+1} \Delta_k^{l+1} \sigma'(z_j^l)) \cdot out_i^{l-1}$$

This takes the matrix form:

$$\frac{dL}{dw^l} = (w^{l+1} \Delta^{l+1} \cdot \sigma(z^l))^T out^{l-1}$$

The according weights and bias can be updated as:

$$new\_weight^l = old\_weight^l - \alpha \frac{dL}{dw^l}$$

$$new\_bias^l = old\_bias^l - \alpha \Delta_j^l$$

Since the loss function used is MSE, the matrix form of the initial gradient of K is:

$$\Delta^K = \frac{2\,(out^K - y)}{size\ of\ out}$$

Here y is the expected output.

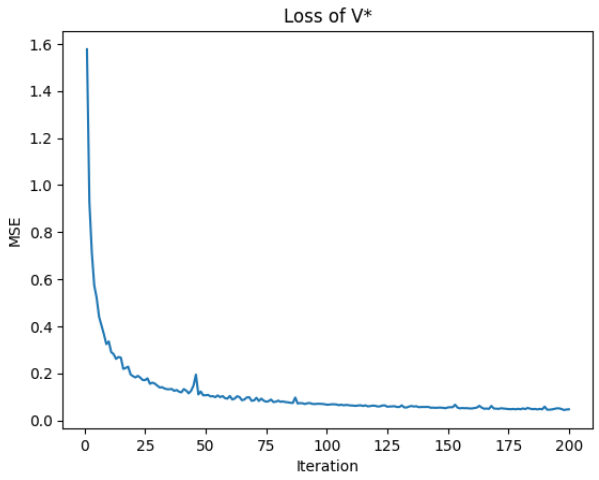
The model is trained using the entirety of U*, by converting each of the 50^3 states to a size 150 vector as training input and the corresponding U*(s) outputs of those states as the expected values. Since infinity values are not trainable the infinity values, but are easily determined, they are filtered out of the training set and considered when the agent determines its next action. In this case overfitting is not an issue since the entire expected input and output is already known.

At each time step, the agent calls a helper method, which evaluates the best possible next action (where it can move to) according to:

$$\min \ action(\sum_{s'} P(s') + V^*(s'))$$

The reward of $+1$ is insignificant here since all actions have the same reward and the value of the state is not being saved. In the case that for some action, the agent would move into the predator's current position, or some s' results in the agent colliding with the predator, these actions will automatically be considered to have an infinity utility (essentially reinserting the infinities which were filtered out).

Accuracy



The neural network was able to minimize the loss to only 1.6 after its first iteration (the initial loss with random weights was not recorded). After 200 iterations, V* becomes very accurate with a MSE of only 0.04698. At this point the improvements became minimal per iteration, however since overfitting is not an issue, theoretically it may have been possible to decrease this value more.

V Partial

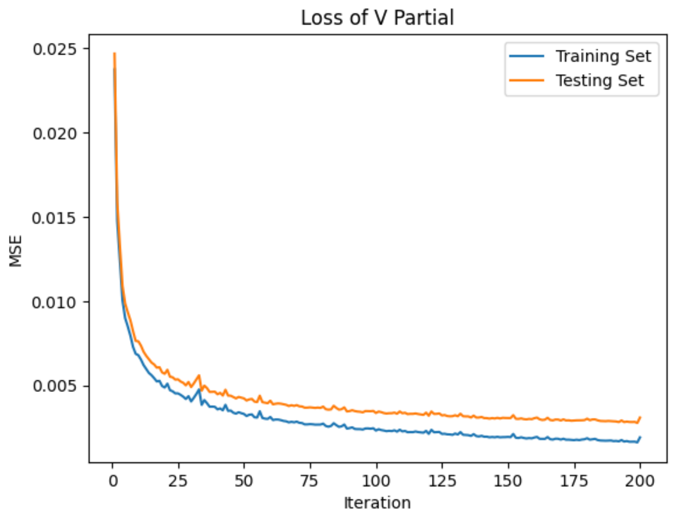
Like V*, V partial takes a combined vector of size 150, where the sub vectors for predator and agent are the same, however the sub vector corresponding to the prey is now replaced with the prey belief vector (because of this the network is also implemented in the *v_ star* class and uses the same structure and updates but has a separate training function to consider a testing set). Since the input format is the same as V*, V Partial attempts to finetune V* by using it as the initial weights with the hope that V* was able to determine certain features about the vectors that will be relevant here. Since the error became relatively low fast this appeared to be successful (more on that below).

The training and testing data for V partial was collected by running the U partial agent 10,000 times on the current environment and collecting the current prey position, predator position, belief vector and the returned value from executing U partial. Of these 10,000 simulations approximately 250,000 samples are collected, only 120,000 samples of U partial calls were used where 100,000 were reserved for training and 20,000 for testing. Some of the remaining samples are used to compare V partial and U partial with V* substituted in which is discussed in the analysis at the end. Like in V* the infinity values are filtered out and considered at the time of execution by determining if the agent will move into a node with the predator or the predator may move into the agent is in as a result of the action.

Here overfitting is a potential issue. To combat it, the elbow technique and early termination taught in class were used, such that at each iteration of gradient descent, the MSE was calculated for both the training and testing set. If the MSE of the testing set is the lowest recorded, it saves the weights and continues. If ten iterations occur in which the testing set no longer is decreasing, the network assumes the model is overfitting or has reached some local minimum and terminates, setting the weights to the saved minimum recorded for the testing set.

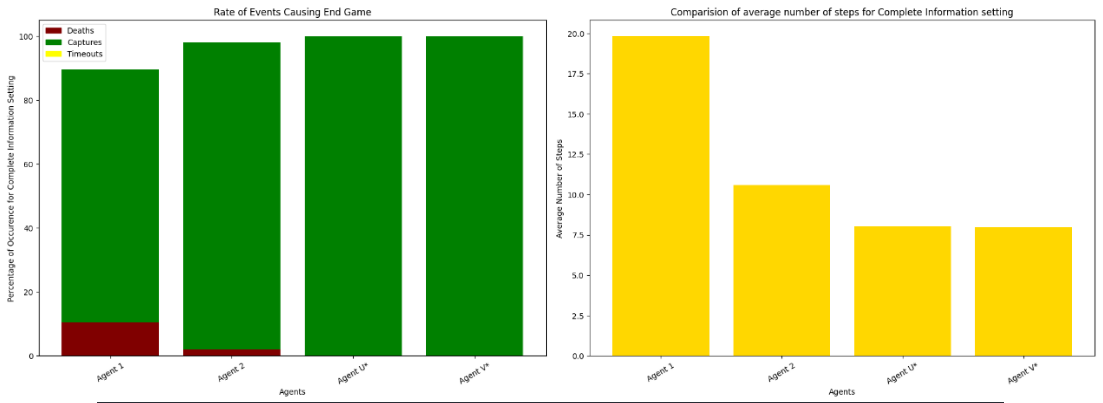
Since V partial returns a partial value, it must choose an action based off partial values, unlike U partial which combines U* values to make a single partial value. This is done using a helper method at each timestep. The helper method copies the current belief vector, then transitions it ahead one timestep. The utility of an action is then created by summing the V partial value for each possible next position of the predator given the advanced belief vector, times the probability that transition occurs. The implications of this will be discussed in the analysis at the end.

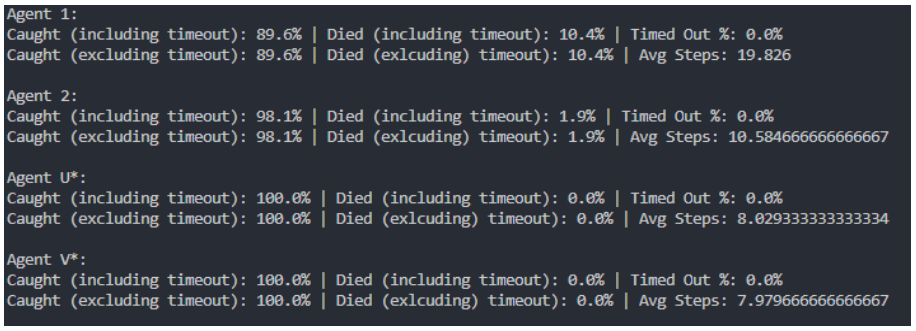
Accuracy



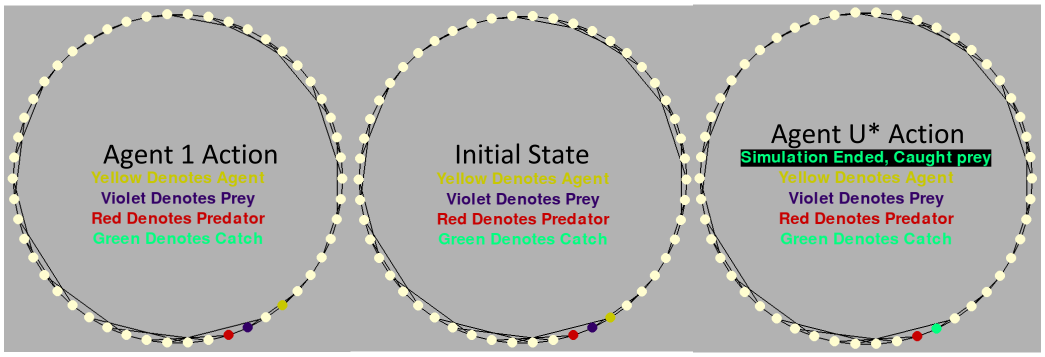
Non-finetuned V* results in a 42.415 loss for the training set and 42.933 for the testing set. V partial initializes itself using these weights and the loss drops to 0.025 in the first iteration, setting it below V* loss on the entire U* data set. This is most likely due to the increased amount of relational input data the belief vector provides which allows the neural network to better determine the relationship between certain features pertaining to the prey. At around 10 iterations the training and testing set diverge slightly, but after that the difference between the two remains consistent. Until 200 iterations the network is flat with a slight downward trend until it spikes up at the 200th iteration. The minimum recorded MSE of the testing set which was used to set the weights was 0.00277, with a corresponding training set loss of 0.00161.

Complete Information Setting Analysis

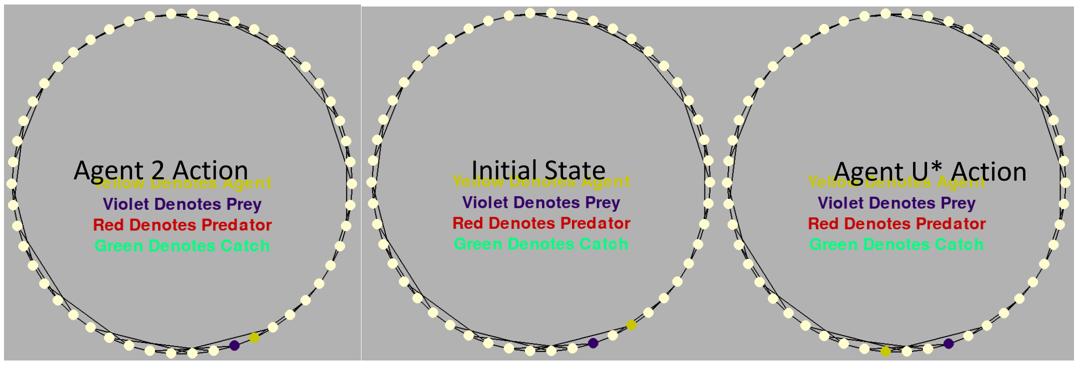



```
Agent 1:
Caught (including timeout): 89.6% | Died (including timeout): 10.4% | Timed Out %: 0.0%
Caught (excluding timeout): 89.6% | Died (exlcuding) timeout): 10.4% | Avg Steps: 19.826

Agent 2:
Caught (including timeout): 98.1% | Died (including timeout): 1.9% | Timed Out %: 0.0%
Caught (excluding timeout): 98.1% | Died (exlcuding) timeout): 1.9% | Avg Steps: 10.584666666666667

Agent U*:
Caught (including timeout): 100.0% | Died (including timeout): 0.0% | Timed Out %: 0.0%
Caught (excluding timeout): 100.0% | Died (exlcuding) timeout): 0.0% | Avg Steps: 8.029333333333334

Agent V*:
Caught (including timeout): 100.0% | Died (including timeout): 0.0% | Timed Out %: 0.0%
Caught (excluding timeout): 100.0% | Died (exlcuding) timeout): 0.0% | Avg Steps: 7.979666666666667
```

The green and red graph on the left shows the success rates of the complete information setting agents and the graph on the right shows the avg steps per iteration taken by these agents. Agents 2, U* and V* all exceed the performance of Agent 1 in both survivability, by about 10%, and in taking at least ten steps fewer. While Agent 2 performs close to U* and V* it manages to take about 2 more steps while still having a higher death rate. The star agents appear to perform optimally, surviving nearly 100% (rounded to three decimal places) of the time, while taking the least steps. Furthermore, V* appears to perform as well as U* given a large enough testing space, in this case taking 0.06 fewer steps which is a negligible difference, indicating that the model for V* is effective at replicating U*.

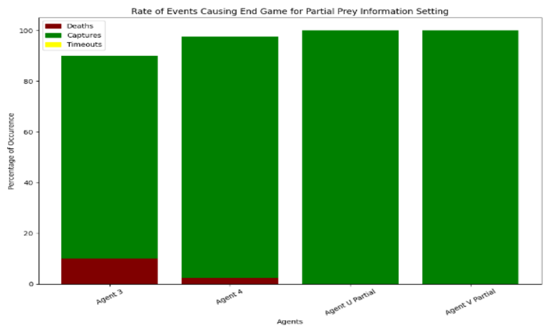


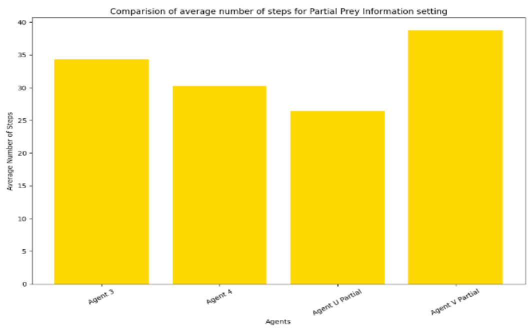
When sharing the center initial state, Agent 1 chooses to retreat due to the vicinity of the predator, even though it could win the game in one move before the predator can win. This is because the fixed conditional logic of Agent 1 does not allow the agent to move closer to the predator, always attempting to remain at least the same distance. On the other hand, when the U* Agent is in a state adjacent to the prey and the predator is not sharing that state, it considers the utility of that action to be 0 + 1 and takes it since that is the minimum possible expected rounds.

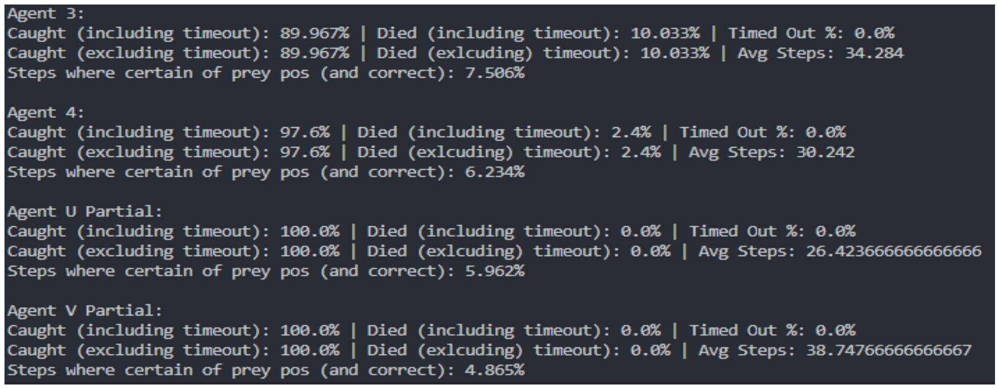


Given this slightly different initial state, Agent 2 moves into a cell where it is likely to die. This is because the "approximated utility" function that I created for this agent subtracts the decay$^{\text{distance of action pos to predator}}$ from decay$^{\text{distance of action pos to prey}}$, which in general cases results in the agent preferring states closer to the prey and further from the predator. In this setup run again, Agent 2 also took another different action not shown, since there are multiple actions with the same assessed value, and it breaks ties randomly. Agent U* always choses to move to a node that is further away unless it can win since moving to the state adjacent to the predator has a higher expected number of rounds and therefore higher chance of death.

Partial Prey Information Setting Analysis

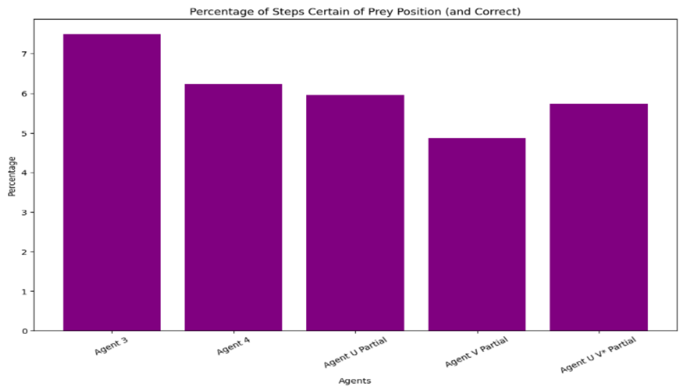






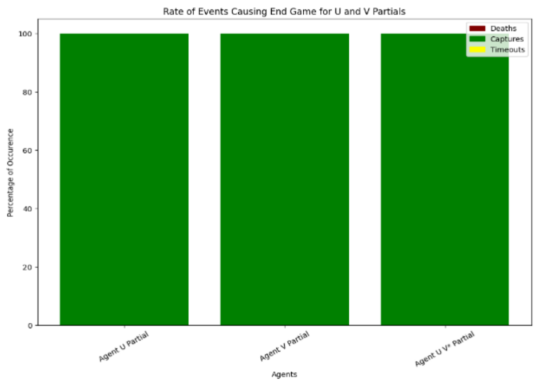
```
Agent 3:
Caught (including timeout): 89.967% | Died (including timeout): 10.033% | Timed Out %: 0.0%
Caught (excluding timeout): 89.967% | Died (exlcuding) timeout): 10.033% | Avg Steps: 34.284
Steps where certain of prey pos (and correct): 7.506%

Agent 4:
Caught (including timeout): 97.6% | Died (including timeout): 2.4% | Timed Out %: 0.0%
Caught (excluding timeout): 97.6% | Died (exlcuding) timeout): 2.4% | Avg Steps: 30.242
Steps where certain of prey pos (and correct): 6.234%

Agent U Partial:
Caught (including timeout): 100.0% | Died (including timeout): 0.0% | Timed Out %: 0.0%
Caught (excluding timeout): 100.0% | Died (exlcuding) timeout): 0.0% | Avg Steps: 26.423666666666666
Steps where certain of prey pos (and correct): 5.962%

Agent V Partial:
Caught (including timeout): 100.0% | Died (including timeout): 0.0% | Timed Out %: 0.0%
Caught (excluding timeout): 100.0% | Died (exlcuding) timeout): 0.0% | Avg Steps: 38.74766666666667
Steps where certain of prey pos (and correct): 4.865%
```

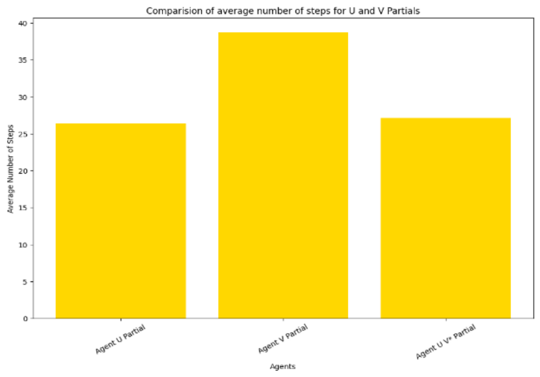
The new U Partial and V Partial agents have the highest survivability rates of the partial prey agents at nearly 100%. Like Agent 2, Agent 4 performs close but not as well in survivability, where Agent 3 does the worst by about 10% when compared to the U and V partials. V Partial manages to perform the worst in minimizing steps, taking approximately 38 per round, followed by agent 3, then 4 and lastly agent U Partial. The poor performance of Agent V Partial is also clear in the purple graph featured below, where despite the Agent taking the most steps, it manages to be the least certain of the prey position. This implies that the Agent takes many steps in a similar area, meaning that each step reveals little about the environment. This comes as a result of the agent expecting the prey to be in an area where it is not actually present. While U Partial has the second lowest certainty rate (excluding U V Partial which will be discussed later), this does not indicate poor performance, since the agent manages to take the fewest steps of the partial prey agents while also surviving the most. This means the agent can determine areas where the prey should be well enough that it manages to collide with it before it can fully determine its location. Agent U Partial is not an optimal agent since it is based on an approximation formula, however the estimation provides a more optimal agent than previous agents, allowing for the greatest survivability and fewest average steps.

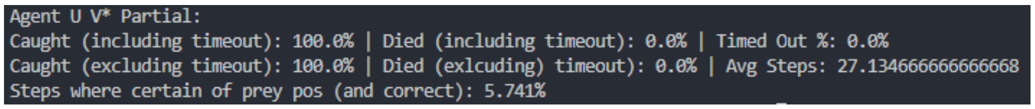


V Partial and U Partial using V*

Despite being based off U Partial and being able to predict very close approximations of its values for a given state, V Partial performs much worse than U Partial. This is because of the difference in how U Partial and V Partial create policies for an action based on the current state. U Partial uses U*, which is a very accurate value of a complete belief state, to determine the utility of possible actions, whereas V Partial must use partial approximated belief values to determine the value of an action. This means that V Partial not only suffers from the loss of the output of the neural network, but also suffers from the compounded error because it builds values of actions from approximated partial utilities.

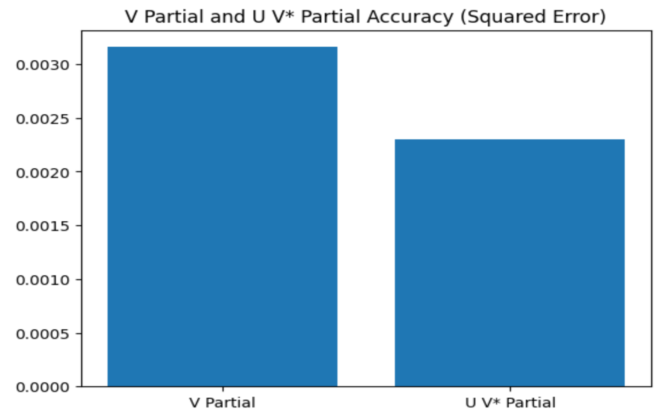






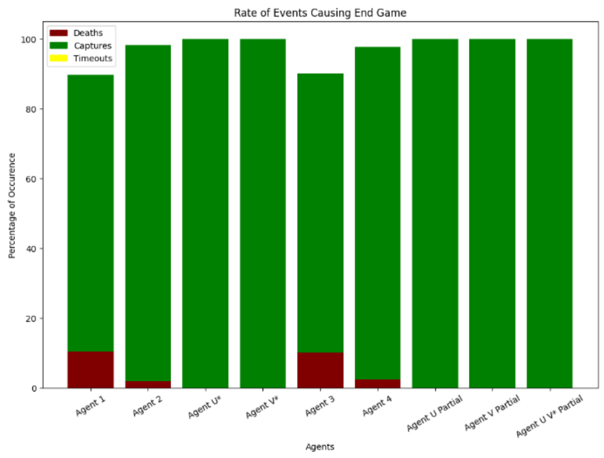
```
Agent U V* Partial:
Caught (including timeout): 100.0% | Died (including timeout): 0.0% | Timed Out %: 0.0%
Caught (excluding timeout): 100.0% | Died (exlcuding) timeout): 0.0% | Avg Steps: 27.134666666666668
Steps where certain of prey pos (and correct): 5.741%
```

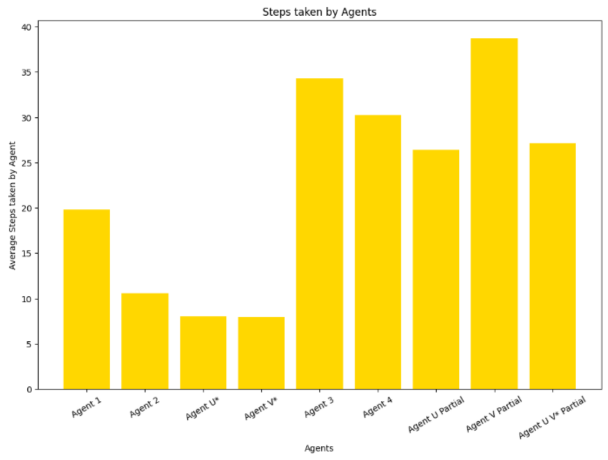
In the graphs above, agent U Partial is featured on the left, followed by V Partial then a third agent, Agent U V* Partial. Agent U V* Partial is essentially agent U partial, but instead of using values from U* to generate its approximated partial values, it makes calls to V*. The result of which is a an Agent that performs nearly identically to U Partial, with 100% survival and taking about 0.5 steps more on average. This is because unlike with the V Partial agents, U V* Partial is able to determine the values of an action based on V*, meaning that like U Partial it benefits from not having to use partial utilities to create the utiltiy of an action when determining a policy.



While the U V partial agent performs much better, its actual ability to predict the values of U Partial is only marginally better than V Partial. On a set of 10,000 inputs not seen by V Partial in training, U V* partial only outperformed V Partial by about 0.0009 MSE, with U V* Partial having a loss of 0.0023 and V Partial having a loss of 0.0031. This difference, while notable visually, is extremely small indicating that it both V and U V* Partial are accurate in predicting values of U Partial. However, these results suggest that U V* Partial is more practical when being used by an agent if performance in terms of survivability and steps is preferred. The tradeoff is that due to the increased number of calls to a neural network, the actual execution is much slower.

Overall Analysis





Unlike in the last project, where the survivability of the new agents went down in the decreased information settings, these new agents all are able to survive with about a 100% success rate. The main performance difference comes from the expected number of rounds, where all complete information agents manage to take fewer steps than the partial setting agents. In all cases the partial agents out performed the original agents and the V agents managed to do nearly as well if not on par with the U agents except for V Partial. Furthermore, despite their performance as agents, both V neural networks managed to managed to model their corresponding U variation succesfully.