2022

Project 3: Better, Smarter, Faster

GYANA DEEPIKA DASARA [GD452]
TEJASHWINI VELICHETI [TV186]

Contents

1. Introduction

1.1 Problem Statement

An undirected graph of 50 nodes contains three entities - the agent, the prey, and the predator. The agent must catch the prey before the predator catches the agent. These three entities move in the graph until the game ends. We have the following tasks at hand :

- Compute U* values, where U*(S) = minimum *#rounds taken for the agent to catch the prey, and move the agent takes decision on the basis of these values*
- Build a Model V to predict the value of U* values for all the states and move the agent based on the predicted values of U*
- Compute U* values for the partial information setting where the exact location of the prey is unknown, where U_partial*(S) = minimum *#rounds taken for the agent to catch the prey, and the agent takes decisions on the basis of these values.*
- Build a Model V to predict the value of U_partial* values for all the states and move the agent based on the predicted values of U_partial*

2. Implementation

2.1 Environment Setup

This part of the implementation requires multiple steps revolving around graph creation, modelling prey and modelling predator.

2.1.1 Graph Creation

Initially, there are 50 nodes connected in a circular fashion. And then, random nodes with degree<3 are chosen, and an edge is added between this node and another random node whose degree<3 and it is within the radius of +5 to -5 of the chosen node. The process is repeated until no more edges can be added. The function *create_graph()* is used to create this graph which returns the required graph (a dictionary consisting of keys as nodes and values as the respective neighbours).

2.1.2 Initialization of Prey, Predator, and Agent

The agent is initialized first, followed by the prey, and the predator. While initializing the prey and the predator we make sure that they are not initialized in the same node as the agent. The function *initilise_prey_pred_agent()* is used for initializing the position of the prey, the predator, and the agent.

2.1.3 Modelling Prey Movements

Every time the Prey moves, it selects among its neighbours or its current cell, uniformly at random and moves to it. The prey's movements carry on this way until the game concludes. The function *move_prey()* contains the logic to move the prey and return the next move of the prey.

2.1.4 Modelling Distracted Predator Movements

This predator has two behaviours. With a probability of 0.6, it moves to its neighbouring cell which is closest to the agent. With a probability of 0.4, it moves randomly to any of its neighbours. The function *move_distracted_predator()* follows this logic to decide the next move of the predator.

Breadth First Search has been implemented to find the shortest distance between the agent and the predator. As the graph has a limited set of nodes and is connected, we don't run into any major efficiency-based issues while computing the shortest path using BFS.

How many distinct states (configurations of predator, agent, prey) are possible in this environment?
The graph contains 50 nodes and the predator, agent, prey can exist in any of the cells. So the total possible distinct states possible in this environment are 50 * 50 * 50 which is 125000 states.

2.2 Modelling Agents in Different Environments

The different environments that we will be modelling our Agents for are:

1. Complete Information Setting: Agent U* and Agent V
2. Partial Information Setting (Prey Unknown): Agent U_Partial and Agent V_Partial

2.2.1 The Complete Information Setting

The Agent is always aware of the location of the Predator and the Prey in this scenario.

2.2.1.1 Agent U*

2.2.1.1.1 Computing Utilities

Initially the values for U* are initialized based on these base conditions:

- U* is defined as minimum expected number of rounds to catch the prey.
- When the agent, prey are in the same position and the predator is not in that position, it is considered as a win case for the agent. So the U* for such states would be 0.
- When the agent and the predator are in the same position and the agent has not caught the prey yet, this case is a loss for the agent. As the agent is dead and it cannot catch the prey anymore, U* for such states would be infinite.
- When the agent is at a distance 1 from the prey, and the predator is not in the prey's position, the U* for such cases is 1 (as only one move is required on the part of the agent to catch the prey).
- If prey and predator are in the same position and that position is the neighbour of the agent, such states have U* value of infinity.
- For all the other states for which the U* is unknown, we initialize the U* for these states as the minimum distance between the agent and the prey.
- As *min_distance(agent,prey)* is a better guess for the minimum *#rounds taken for the agent to catch the prey compared to any other arbitrary guess, we might get a faster convergence when we apply value iteration.*

What states s are easy to determine U ∗ for?

U* = minimum expected number of rounds to catch the prey
The states that U* is straightforward to determine for are:

- When the agent, prey are in the same position and the predator is not in that position, it is considered as a win case for the agent. So the U* for such states would be 0.
- When the agent and the predator are in the same position and the agent has not caught the prey yet, this case is a lose for the agent. As the agent is dead and it cannot catch the prey anymore, U* for such states would be infinite.
- When the agent is at a distance 1 from the prey, and the predator is not in the prey's position, the U* for such cases is 1 (as only one move is required on the part of the agent to catch the prey).
- If prey and predator are in the same position and that position is the neighbour of the agent, such states have U* value of infinity.

The Optimal utility can be computed using the Bellman's equations

$$U^*(s) = min\ a \in A(s)\ [\ C_{(s,a)} + \Sigma\, p_{(s,s')}\, U^*(s')]$$

where the terms are –

$C_{(s,a)}$ *is the cost to take action a*

$\Sigma\, p_{(s,s')}$ *is the product of transition probabilities of the prey and predator moving to state s'*

We use the above equation and the below update rule to carry forward value iteration

$$U_{k+1}{}^*(s) = min\ a \in A(s)\ [\ C_{(s,a)} + \Sigma\, p_{(s,s')}\, U_k{}^*(s')]$$

We have obtained the optimal U* when we have satisfied the below convergence rule

$$|\ U_{k+1}{}^*(s) - U_k{}^*(s)\ | = 0$$

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

The optimal utility can be computed using the Bellman's equations, which also denotes how U*(s) relates to U* of other states :

$$U^*(s) = min\ a \in A(s)\ [\ C_{(s,a)} + \Sigma\, p_{(s,s')}\, U^*(s')]$$

where the terms are –

$C_{(s,a)}$ *is the cost to take action a*

$\Sigma\, p_{(s,s')}$ *is the product of transition probabilities of the prey and predator moving to state s'*

In above, we have considered the value Cost (s,a) to be 1 in the equation during the update for the non-terminal states.

In our case, it's beneficial to move the agent to the state with the minimum utility, which can be defined as,

$$\Pi^*(s) = arg\ min\ a \in A(s)\ [\ C_{(s,a)} + \Sigma\, p_{(s,s')}\, U^*(s')]$$

2.2.1.1.2 Algorithm

1. We first pre-compute the optimal utilities for all the states using the formula and the process stated above.
2. Once the Agent , Prey and Pred positions are initialized, the Agent chooses to make a move to either the neighbouring cell or remains at the same position based on whichever state has the lowest U* value.
3. After the agent moves to the next position, we check if the prey/predator is present in the agent position. If the pred is in Agent's position, the Agent loses. If the Prey is in the Agent's position, the Agent wins.
4. The prey moves to either of its neighbours or stays in place. We check if the prey is in the agent position, if yes, the agent wins.
5. The predator moves, and we check if the predator catches the agent. If it catches the agent, the game ends as the agent loses, else we repeat the process from step 1 until the game ends.

2.2.1.1.3 Results

We have obtained the optimal U* when we have satisfied the below convergence rule

Are there any starting states for which the agent will not be able to capture the prey? What causes this failure?
Yes, states when the agent will not be able to capture the prey exist. During the initialization, if the agent and prey are initialized in the same node then the agent will not be able to catch the prey. If the agent, prey and predator are initialized in the same node, this would not result in a win.

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

Visulization for a finite max value

```
1 visulization_max_finite_utility(states)

Max finite Utility is 24.707059472803127 and the state is (8, 13, 12)
Let's simulate the game
Initialization: Agent Position = 8 , Prey Position = 13 , Predator Position = 12
Round 0 => Current Agent, Prey and Pred positions are: 8 , 13 , 12
Anget's turn to move:
Avilable agent moves with respective utilities: {9: inf, 7: 4, 4: 4, 8: 3}
Agent moves to : 8 , Prey moves to : 32 , Predator moves to : 9
Round 1 => Current Agent, Prey and Pred positions are: 8 , 32 , 9
Anget's turn to move:
Avilable agent moves with respective utilities: {9: inf, 7: 14, 4: 14, 8: inf}
Agent moves to : 4 , Prey moves to : 33 , Predator moves to : 8
Round 2 => Current Agent, Prey and Pred positions are: 4 , 33 , 8
Anget's turn to move:
Avilable agent moves with respective utilities: {5: 12, 3: 12, 8: inf, 4: inf}
Agent moves to : 3 , Prey moves to : 27 , Predator moves to : 4
Round 3 => Current Agent, Prey and Pred positions are: 3 , 27 , 4
Anget's turn to move:
Avilable agent moves with respective utilities: {4: inf, 2: 13, 7: 11, 3: inf}
Agent moves to : 7 , Prey moves to : 29 , Predator moves to : 8
Round 4 => Current Agent, Prey and Pred positions are: 7 , 29 , 8
Anget's turn to move:
Avilable agent moves with respective utilities: {8: inf, 6: 12, 3: 12, 7: inf}
Agent moves to : 6 , Prey moves to : 27 , Predator moves to : 9
Round 5 => Current Agent, Prey and Pred positions are: 6 , 27 , 9
Anget's turn to move:
Avilable agent moves with respective utilities: {7: 11, 5: 11, 10: inf, 6: 10}
Agent moves to : 6 , Prey moves to : 27 , Predator moves to : 10
```

The largest finite value of U* is 24 as shown in the above code output. The Initial position of Agent is 8, Prey is 13 and Predator is 12.
The Agent can move to it's neighbour or stay in it's position. The Agent decides it's next move on basis of utility. The Agent moves to the node with min Utility or decides to stay in it's position if it has minimum Utility when compared to it's neighbours.
At Round 0: The current positions of Agent, Prey and Predator are 8,13,12 respectively.

The Agent can move to 9,7,4 or can stay in it's current cell i.e 8.
Among these, cell 8 has the least utility. Therefore the Agent stays in it's current cell.
The Prey moves to cell 32 and Predator moves to cell 9.
This continues till the Agent catches the prey or Predator catches the agent.

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?

Agents ->	Agent 1	Agent 2	Agent U*
Success Rate	85.86	97.38	99.13
Avg num of Steps to Catch prey	14.10	15.56	10.99

We can notice that the performance of Agent U* is significantly better than that of Agent 1. We can see that the performances of Agent 2 and Agent U* are on par with each other. There is a slight reduction in steps that is taken to catch the prey by the Agent U* compared to Agent 1. This is because Agent U* has incorporated information about the future possibilities into it's decision making process which helps it to some extent to win with less number of steps.

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.

Given:
Agent Node has two neighbours: N1 and N2.
Agent Node: Distance(Agent, Prey) = 8 Distance(Agent, Predator) = 1
N1: Distance(N1, Prey) = 7 Distance(N1, Predator) = 1
N2: Distance(N2, Prey) = 9 Distance(N2, Predator) = 2

With the above data, Agent 1 chooses N1 but Agent U* chooses N2. As Agent1 chooses N1, we can see that the agent will be caught by the predator in the next move itself. Whereas Agent U* chooses N2, and increases its chances of survival by moving away from the predator. Agent U* is able to look forward in the future and realize that making a move closer to the prey will lead to the agent getting killed before it can reach the prey. So it avoids this situation and chooses another node to move to that helps it to win the game eventually.

2.2.1.2 Agent V

2.2.1.2.1 Introduction

We need to build a model V that is able to predict U* value given a particular state.

2.2.1.2.2 Input to Model V

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

The inputs are are sent to the V model for training are as follows:

- The adjacency matrix of the graph (flattened to a 50*50 list)

- State is split into three features [agent_position, predator_position, prey_position], all the 125000 states are considered as training data

Apart from the graph and states and utilities, the following features are given as input to the model -

- Distances - dist(agent,predator) and dist(agent,prey) : This information can add value to the model's predictions as the objective of the utility is to get closer to the prey while avoiding the predator.
- Transitional_Probability(Predator, Agent) : The probability of predator moving to the agent's position
- Transitional_Probability(Prey, Agent) : The probability of prey moving to the agent's position
- Degrees of the positions - [degree_agent_position, degree_prey_position, degree_predator_position] : Using this information to the model might help the model better understand the states.

2.2.1.2.3 Model V

Steps for Training the Neural Network:

1. Feed the input data to the NN. The data flows from layer to layer and the output is obtained. This step is also called the Forward Propagation.
2. Calculation of the error between the predicted output and the true output. This error is referred to as loss and the goal is to train the model in such a way that this error is minimized.
3. Adjusting the parameters of the neural network based on the loss. This is also referred to as the backpropagation step and we have implemented it using gradient descent.
   Parameter updation using gradient descent:

$$W = W - (learning\ rate * \frac{\partial E}{\partial W})$$

$$B = B - (learning\ rate * \frac{\partial E}{\partial B})$$

4. Repeat from Step 1 again until the model is trained over multiple epochs till the required loss is obtained.

Implementation of the Neural Network:

1. Implementing the layers as classes, defining the sizes of the input and output to this layer, and initialization of the parameters.
2. Defining the Forward propagation and backward propagation for each layer.
3. Creating and defining the activation layer.
4. Implementation of the activation functions and the loss functions.

Forward Propagation

1. In this step, when an input X is given to a layer, an output Y is obtained.
2. The output of each layer can be computed as Y = WX + B, where y denotes weights, X is the input, B is the bias.
3. The activation function can be expressed with the equation Y = f(X), where f is the type of the activation function we are using.
4. The Loss Computed (Mean Squared error) can be computed using the following equation:

$$E = \frac{1}{N}\sum_i (y^*_i - y_i)^2$$

Backward propagation

1. In this step, when the derivative of the error is given with respect to the output of the layer, we compute the derivative of the error with respect to the input of the layer and other parameters.
2. To update the parameters of the model, we need to compute the derivative of the error with respect to the parameters (here the parameters are weights, biases). These values can be computed using chain rule.

Derivative of the error E w.r.t Weights after applying chain rule (for dense layer) :

$$\frac{\partial E}{\partial W} = \frac{\partial E}{\partial Y} X^T$$

Derivative of the error E w.r.t Bias after applying chain rule (for dense layer) :

$$\frac{\partial E}{\partial B} = \frac{\partial E}{\partial Y}$$

Derivative of the error E w.r.t input X after applying chain rule (for dense layer):

$$\frac{\partial E}{\partial X} = w^T \frac{\partial E}{\partial Y}$$

we compute this as the $\frac{\partial E}{\partial X}$ is the $\frac{\partial E}{\partial Y}$ for the previous layer

3. The following expressions can be used to backpropagate when there are activation functions involved.
   Derivative of the error E w.r.t Weights for activation layer using activation function f :

$$\frac{\partial E}{\partial W} = \frac{\partial E}{\partial Y} \odot f'(X), \text{ where } \odot \text{ represents element wise multiplication.}$$

4. We can compute the error $\frac{\partial E}{\partial Y}$ for the last layer using the following :

$$\frac{\partial E}{\partial Y} = \frac{2}{N}(Y - Y^*)$$

What kind of model are you taking V to be? How do you train it?
The model that is used to predict U* is a neural network. It is a combination of three fully connected dense layers with activation functions after the first two layers. Architecture of the network: Input → Dense Layer (2508,100) → Activation function (ReLu) → Dense Layer(100,50) → Activation function (ReLu) → Dense Layer (50,1) → Prediction We have chosen a neural network for our model as it better captures the non-linearities and the complexities of the underlying data than a linear model. Also, the ease and ability to introduce variation in terms of layers and activation functions makes neural networks a good choice for the problem. While trying out different activation functions with the Neural network, ReLu was able to minimize the loss comparatively faster than other non-linear activation functions such as tanh. So, we have chosen ReLu as the activation function for the intermediate layers. **Process:** 1. We have a data set consisting of 115540 rows of data that includes all the features that we have mentioned above. This has been passed to the model along with Y (which are the calculated U* values) to perform supervised learning. We have not included the states where the U* is infinity. We use a lookup table for these states when we run Agent U*. 2. The model has been run over 3000 epochs and the error metric MSE has reduced to values below 0.001. Once the training is complete, we have tested the model accuracy by sending in test data points and have noticed a MSE of 0.00156. This means that the model is highly accurate. 3. We then pass all the states as input to the model and obtain the predicted U* values which we store in a dictionary called predicted_u_values. 4. We then calculate the success rate for the Agent U_partial by using the same rules of Agent U* but using the predicted U values instead. To compute the success rate multiple simulations are carried out and the success rate is calculated.

Is overfitting an issue here?
No, overfitting is not an issue in this case. We pass all the states, its corresponding features, and their obtained optimal U* value. The search space that is sent for training is complete. Each new test point the model sees, the model has already been trained on it earlier. As the model we are building here is not graph agnostic, and is graph specific, an overfit model will not be an issue here.

2.2.1.2.4 Results

How accurate is V ?
Model V is very accurate in it's predictions of U* given an input state. During the model training, the error metric used is the Mean squared error. The training has been stopped when the error has reached a point very close to zero, that is, minimizing the error to the maximum extent within reasonable epochs. The model has been run over 3000 epochs and the error metric MSE has reduced to values below 0.0001. Once the training is complete, we have tested the model accuracy by sending in test data points and have noticed a MSE of 0.00156. This means that the model is highly accurate.

How does its performance stack against the U $*$ agent?

Agents	Agent U*	Agent V
Success Rate	99.27	98.67
Avg Steps to catch prey	11.79	12.05

Moving the agent based on the predicted U* yields a performance which on par with the agent that moves using the calculated U* values. The reason for this is the V model is highly accurate and is trained on complete data. V doesn't have to predict U* for states that it has never seen before in its training data. So U* and V model agents have very similar performances and both at around 98-99% success rates and the steps taken are approximately the same.

2.2.2 The Partial Prey Information Setting (Prey Location Unknown)

In this scenario, the Agent is always aware of the location of the Predator but is not aware of the location of the Prey. The agent must estimate the position of the prey through surveying and updating the beliefs.

2.2.2.1 Agent U Partial

2.2.2.1.1 Belief Updates

- **Initialization:** Initially, the only information we have about the prey is that it is not located in the agent's current location. Apart from that, the prey is equally likely to be present in any of the other nodes.

$$P(\text{Prey in Agent's current position}) = 0$$

$$\text{For other 49 nodes, each represented by } j\text{: } P(\text{Prey in } j) = \frac{1}{49}$$

- **Update after the survey:** When we survey the node with the maximum probability of containing the prey, we either find the prey or we don't find the prey. So we will have two cases of updates here.
  - **If survey is successful:**

$$P(\text{Prey in surveyed node}) = 1$$

$$\text{For nodes, each represented by } j\text{, where } j \text{ is not surveyed node: } P(\text{Prey in } j) = 0$$

  - **If survey is not successful:**

$$\text{For nodes, each represented by } j\text{, where } j \text{ is not surveyed node:}$$

$$P(\text{Prey in } j \mid \text{prey not in surveyed node}) = \frac{P(\text{Prey in } j) * P(\text{Prey not in surveyed node} \mid \text{Prey in } j)}{P(\text{Prey not in surveyed node})}$$

$$P(\text{Prey in } j \mid \text{prey not in surveyed node}) = \frac{P(\text{Prey in } j)}{1 - P(\text{Prey in surveyed node})}$$

$$\text{For surveyed node: } P(\text{Prey in surveyed node}) = 0$$

- **Update after the agent moves:** If the Agent finds the prey after it moves, the game ends. We need to update the belief when the agent did not find the prey in its new location.

$$\text{For nodes, each represented by } j\text{, where } j \text{ is not agent's new location and } j \text{ is not previously surveyed node :}$$

$$P(\text{Prey in } j \mid \text{Prey not in agent's location}) = \frac{P(\text{Prey in } j) * P(\text{Prey not in agent's location} \mid \text{Prey in } j)}{P(\text{Prey not in agent's location})}$$

$$P(\text{Prey in } j \mid \text{Prey not in agent's location}) = \frac{P(\text{Prey in } j)}{1 - P(\text{Prey in agent's location})}$$

$$\text{For agent's location: } P(\text{Prey in agent's location}) = 0$$

- **Update after the prey moves:** The prey can choose any of its neighbours at random to move to next or choose to remain at its position.

$$P(\text{Prey in } i \text{ after prey moves}) = \sum P(\text{Prey in } j) * P(\text{Prey moves from } j \text{ to } i \mid \text{Prey in } j)$$

$$\text{where } j \text{ consists of } i \text{ and all the neighbours of } i$$

2.2.2.1.2 Algorithm for Agent U partial

1. If it is not certain about where the prey is, the agent surveys the node with the maximum belief of containing the prey.
2. Based on the survey result, the beliefs of the prey get updated.
3. At this point, we compute the U_partial* values for the agent's position and all agent's neighboring position. The following formula is used to compute the utilities of the state when the prey's location is represented using a set of beliefs. U* values that are used in the formula

to compute U_partial are formulated using the logic stated in the previous section. The agent picks the move that has the lowest utility score and chooses to move to it.

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

4. After the agent moves to the next position, we check if the prey/predator is present in the agent position. If the prey is not present in the agent position and the game has not yet ended, we update the prey's beliefs again with the new information that the prey is not present in the agent's position.
5. The prey moves to either of its neighbours or stays in place. We check if the prey is in the agent position, if yes, the agent wins.
6. The beliefs of prey are updated based on the transition model.
7. The predator moves, and we check if the predator catches the agent. If it catches the agent, the game ends as the agent loses, else we repeat the process from step 1 until the game ends.

2.2.2.1.3 Results

Simulate an agent based on U Partial, 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?

Agents ->	Agent 3	Agent 4	Agent U Partial
Success Rate	83.42	96.38	98.27
Avg num of Steps to Catch prey	15.53	16.92	11.79

We can notice that the performance of Agent U Partial is significantly better than that of Agent 3. We can see that the performances of Agent 4 and Agent U Partial are at a difference of approx. of 2% which is an improvement but not significant. There is a slight reduction in steps that is taken to catch the prey by the Agent U Partial compared to Agent 3. This is because Agent U partial has incorporated information about the future possibilities into it's decision making process which helps it to some extent to win with less number of steps.

2.2.2.2 Agent V Partial

2.2.2.2.1 Introduction

Build a model V partial to predict the value U partial for partial information states

2.2.2.2.2 Input to Model V Partial

How do you represent the states Sagent, Spredator, p as input for your model? What kind of features might be relevant?

The state is represented as input in the following way:

- A feature named agent_pos contains the position of the agent
- A feature named pred_pos contains the position of the predator
- Features labeled P_1,P_2,.....,P_50 contain the belief values of the prey at each position

The other features that might be relevant to pass to the model are -

- Distance_agent_predator : minimum distance between the agent and the predator
- Node with the maximum prey belief
- Degree_Predator : Number of nodes the predator
- Certain_where_the_prey_is : boolean column which tells based on the beliefs if the prey position is known
- Transitional_Probability(Predator, Agent) : The probability of predator moving to the agent's position
- Adjacency matrix : For the model to understand the the structure of the graph

2.2.2.2.3 Model V Partial

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

The model that is used to predict U* is a neural network. It is a combination of three fully connected dense layers with activation functions after the first two layers.
Architecture of the network:
Input → Dense Layer (2508,100) → Activation function (ReLu) → Dense Layer(100,50) → Activation function (ReLu) → Dense Layer (50,1) → Prediction

We have chosen a neural network for our model as it better captures the non-linearities and the complexities of the underlying data than a linear model. Also, the ease and ability to introduce variation in terms of layers and activation functions makes neural networks a good choice for the problem. While trying out different activation functions with the Neural network, ReLu was able to minimize the loss comparatively faster than other non-linear activation functions such as tanh. So, we have chosen ReLu as the activation function for the intermediate layers.

Data Collection:

The game has been simulated multiple times according to the rules of agent U_partial. Then the prey beliefs, state, Upartial has been captured and stored in a csv. Around 120,000 rows of data have been generated this way. Rest data has been formulated for all the states where the belief vector conveys the exact position of the prey. This data would correspond to the cases where we are certain where the prey is. Features like node_degree, dist(agent,pred), transitional probabilities, adjacency matrix have been added to the data as well to help it predict better.

Process:

1. We have a data set consisting of 150,000 rows of data that includes all the features that we have mentioned. We have split this dataset into train and test data in the ratio 80:20. The train data has been passed to the model along with Y (which are the calculated U* values) to perform supervised learning. We have not included the states where the U* is infinity. We use a lookup table for these states when we run Agent Vpartial.
2. The model has been run over 3000 epochs and stopped when error metric MSE falls below 0.05 (not reducing it beyond that (early termination in attempt to avoid overfitting)). Once the training is complete, we have tested the model accuracy by sending in test data on the data and have noticed a MSE of 0.21.The MSE is higher than what was observed in Upartial as test data contains points that the model Vpartial has never seen before.
3. We then simulate the game using the rules similar to Upartial. But while instead of calculating the Upartial* value by formula given, we use the Vpartial model to predict the value instead. The Agent chooses the position with the min utility value as the next move.
4. We then calculate the success rate for the Agent U_partial by using the same rules of Agent U* but using the predicted U values instead. To compute the success rate multiple simulations are carried out and the success rate is calculated.

Is overfitting an issue here?

In this case, when the prey position is not definite, infinite states are possible. So, it is not possible for the model to be trained on all possible states. Due to this, the model will encounter test cases that it has not seen earlier. If the model overfits, the predictions on the test data that the model has not seen before might diverge a lot from the true values. This impacts the efficiency of the model and hampers it to perform well.

There are several techniques to reduce overfitting-

1. The idea of introducing a dropout layer can help the model from overfitting as though inactivating random neurons, there's no strong dependency on any one neuron for the output.
2. Through Early termination, we can stop the training when the loss reduces to a reasonably low value, but not entirely to zero. This helps the model to not overfit on the train data.

2.2.2.2.4 Results

How accurate is V Partial ? How can you judge this?

While running the game for the Agent Upartial, along with computing the Upartial utility values, we apply Model Vpartial and get the predictions for the Upartial values. This way we can compare the results of formula based Upartial value and model-generated Vpartial value as all the input features remain the same. Once we obtain the these two values, we can use a error metric (like MSE) to compute the error between the actual and predicted value. The error can be used as an indication for the the Vpartial model accuracy. We have obtained an MSE of 0.37 on the test data for Model V partial.

Is V partial more or less accurate than simply substituting V into equation (1)?

The V model has a very high accuracy in predicting the U* values. So, if we substitute V into the above equation, the obtained utility is very similar to utility computed in Upartial in terms of accuracy.
After substituting the V in the equation, the values we obtain for the utility are very close to the formulated U_partial values. As Model Vpartial is not overfit to the training data, we observe that the predictions of Vpartial are close to the values of Upartial but not exact, so comparatively less accurate. Adding more training data and more robust features to explain the graph and current states can help Vpartial perform better.

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

Agents	Agent U Partial	Agent V Partial
Success Rate	98.27	91.67
Avg Steps to catch prey	11.79	15.89

We can see that Agent Upartial performance is better than the performance of Vpartial. This maybe due to the fact that the data that is used for training Vpartial is not sufficient enough for the model to be able to estimate the possible prey location. As beliefs are involved, which in turn leads to infinite possible states, the test accuracy on unseen data points is not good enough. Enriching the train data so that it's more diverse and adding features that can give more information about the location of the prey might help improve the models performance.

2.2.6 Bonus

Bonus 2: Build an optimal partial information agent.

.

In the partial information setting, we are not sure of the prey's position. We approximate the position of the prey based on the beliefs we maintain regarding the prey's position.

As the agent's location and the predator's location is already known, we can try to always prioritize moving away from the predator if it comes too close to the agent. This way we can avoid the predator killing the agent. However, we also have to minimize the number of rounds it takes to catch the prey. To do that we identify the area within the graph where the prey might be by repeatedly applying transition updates on the present belief. This way we identify where the prey will be in the near future and can make an attempt to move closer to that area directly at the current timestep.

1. *Identify Top 3 nodes that have the highest belief of containing the prey.*
2. *Calculate the distances from the agent and each of the top 3 nodes.*
3. *For each distance d in distances_from_agent_to_top3nodes, apply transition update formula of the prey on the current belief state d times. This way we identify the region where the prey might be present in the future.*
4. *Once we have identified the region where the prey might be in the future, we should attempt to move closer to that region.*
5. *To do that we look at the agent's neighbors and see which neighbour is close to the region where we assume the prey will be in the near future.*
6. *Once we have identified that neighbour, we check if moving to this neighbour is risky in terms of predator catching the agent. If the risk is low we move towards the agent. If the risk isn't low, then we make other moves that will help the agent to move away from the predator.*
7. *We repeat this process till the time the game concludes. This way we prevent the agent from getting killed and make an attempt to move closer to the place where the prey might be in the near future from the current timestep itself. This would help the agent conclude the game successfully with minimal steps.*