Better Faster Stronger Lab Report

Tej Shah
Rutgers University
tej.shah@rutgers.edu

Srinandini Marpaka
Rutgers University
sm2237@rutgers.edu

Shivam Agrawal
Rutgers University
sa1547@rutgers.edu

Abstract

Using the same Circle of Life environment from Project 2, we build a few intelligent agents for different environments. In the complete information environment, we use value iteration to iteratively solve for the Bellman Equations and calculate optimal utilities for every state in the state space. With the utilities U^*, we build an agent with the optimal policy using the Bellman Equations formulation. For data efficiency, we develop and implement a neural network V from scratch to approximate U^*; this can be queried at inference time for making decisions. In the partial prey information setting, we develop $U_{partial}$. This is the expected value of all the optimal utilities U^* for every location of the prey using our probability distribution $\mathbf{p}_{prey}$. For data efficiency, we develop and implement a generalized neural network function approximater $V_{partial}$ to predict $U_{partial}$ from data. Finally, we outline an optimal $U^*_{partial}$ using the temporal difference method of Deep-Q Learning and an alternative of value iteration with Neural Networks.

1 Introduction

There are 3 entities in an environment: the prey, the predator, and the agent. These 3 entities interact with each other on a graph environment with 50 nodes. The game's objective is for the agent to capture the prey before being killed by the predator. Until the game concludes, the prey moves with equal probability to any of its immediate neighbors, or it stays in its current location. The predator is easily distracted. It moves optimally towards the agent with a 0.6 probability and to one of its neighbors at random with a 0.4 probability. The agent moves first, then the prey, and then the predator. If the agent and prey occupy the same node, then the agent wins. If the agent and the predator occupy the same node, the predator wins. The game terminates early after 1000 time steps.

1.1 How many distinct states (predator, agent, prey) are possible in this environment?

The agent, prey, and predator can occupy 1 of 50 nodes on the graph: $\mathcal{Z} = \{1, 2, \ldots, 50\}$. Formally, $\mathcal{Z}_{agent} = \{1, 2, \ldots, 50\}$, $\mathcal{Z}_{prey} = \{1, 2, \ldots, 50\}$, $\mathcal{Z}_{predator} = \{1, 2, \ldots, 50\}$.

For this environment, the state space (including terminal states) $\mathcal{S}^+ = \mathcal{Z}_{agent} \times \mathcal{Z}_{prey} \times \mathcal{Z}_{predator}$.

$$\begin{aligned} \mid \mathcal{S}^+ \mid &= \mid \mathcal{Z}_{agent} \mid \times \mid \mathcal{Z}_{prey} \mid \times \mid \mathcal{Z}_{predator} \mid \\ &= 50 \times 50 \times 50 \\ &= 125,000 \end{aligned}$$

There are 125,000 distinct states (configurations of agent, prey, and predator) in this environment.

1.2 Graph

Because the values of U^* depend on the graph, we fixed the following graph G for the entire problem.

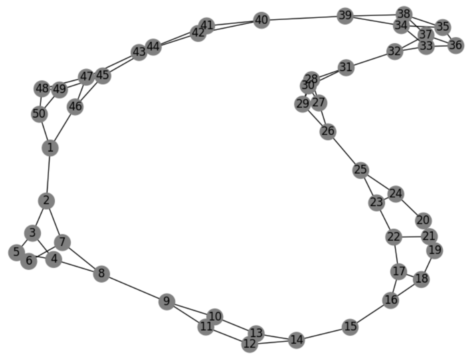

Figure 1: Game Graph G

2 Optimal Complete Information Agent

2.1 How does $U^*(s)$ relate to U^* of other states? When is U^* easy to determine?

Given s as a state and s' as the consequent state, $U^*(s)$ relates to $U^*(s')$ with the Bellman Equations. We compute the Bellman Equations with value iteration. A state is defined as $s = (z_{agent} \in \mathcal{Z}_{agent}, z_{prey} \in \mathcal{Z}_{prey}, z_{predator} \in \mathcal{Z}_{predator})$. Hence, $\forall s \in \mathcal{S}^+$, we find it easiest to compute $U^*(s)$ where $z_{agent} = z_{prey}$ or $z_{agent} = z_{predator}$, whose $U^*(s)$ values are 0 and $-\infty$ respectively. More generally, $U^*(s)$ is easiest to compute for states where the agent occupies the same node as either the prey or the predator, a terminating state of the game.

To learn the optimal value function and policy for the complete information environment with value iteration, we solved the Bellman Equation as follows. n is a function that returns the number of possible moves. n_{all} is the number of possible moves an entity can take. $n_{optimal}$ is the number of optimal neighbors that minimize the distance to the agent via BFS from the predator. The prey moves with equal probability to its current location or one of its neighbors. The predator moves to an optimally with $p = 0.6$ or to one of its neighbors with $p = 0.4$.

$$
\begin{aligned}
U^*(s) &= \max_{a \in \mathcal{A}(s)} [r_{s,a} + \beta \sum_{s'} p^a_{s,s'} \cdot U^*(s')] \\
&= \max_{a \in \mathcal{A}(s)} [-1 + \sum_{s'} p^a_{s,s'} \cdot U^*(s')] \\
&= \max_{a \in \mathcal{A}(s)} [-1 + \sum_{s'} p(z_{prey} \rightarrow z^{'}_{prey}) \cdot p(z_{pred} \rightarrow z^{'}_{pred}) \cdot U^*(s')] \\
&= \max_{a \in \mathcal{A}(s)} \left[-1 + \sum_{s'} \frac{1 \cdot}{n_{all}(z_{prey})} \cdot \left(\frac{0.40 \cdot}{n_{all}(z_{pred})} + \frac{0.60 \cdot \mathbb{1}[\texttt{isoptimal}(z^{'}_{pred})]}{n_{optimal}(z_{pred})} \right) \cdot U^*(s') \right]
\end{aligned}
$$

where:

$$
\begin{aligned}
U^*(s) &= \text{maximal utility of state s} \\
\mathcal{A}(s) &= \text{ action function that changes location of agent from } z_{agent} \rightarrow z^{'}_{agent} \\
r_{s,a} &= -1 \text{ since we want to minimize expected number of rounds} \\
p_{s,s'} &= \text{ probability of transitioning from } s \rightarrow s' \\
\beta &= \text{ 1 since the game is finite and we want minimal rounds to catch prey}
\end{aligned}
$$

2.2 Describe the algorithm to compute U^* in detail

Our algorithm in detail is value iteration:

Algorithm 1 Value iteration for finding U^*

```
Initialize the value function $U^*$ for all states $s$ to $-1, 0, -\infty$
repeat
    for all states s do
        Calculate the maximum value over all possible actions a:
        $U_{k+1}(s) = \max_a [R(s,a) + \sum_{s'} P(s'|s,a) * U_k(s')]$
    end for
    Update the value function: $U_k(s) = U_{k+1}(s)$
until convergence
$U^* = U_k$
$\pi^*(s) = \arg\max_a [R(s,a) + \sum_{s'} P(s'|s,a) * U^*(s')]$
```

An explanation for value iteration is below:

Initialize the terminal states: (1) $U_1(s) = -\infty$ if the agent and predator occupy the same location (2) $U_1(s) = 0$ if the prey and agent occupy the same location. For all other states, initialize $U_1(s) = -1$.

Until $\mid U_k(s) - U_{k-1}(s) \mid < \epsilon = 0.001$, for each state s, do:

$$U_{k+1}(s) = \max_{a \in \mathcal{A}(s)} [r_{s,a} + \beta \sum_{s'} p^a_{s,s'} \cdot U_k(s')]$$

As $k \to \infty$, we note that $U_k(s) \approx U^*(s)$ in the limit. In other words, observe that our approximations for U^* get successfully better as we do more iterations of the algorithm. See below:

$$U_1(s) \rightsquigarrow U_2(s) \rightsquigarrow \ldots \rightsquigarrow U_k(s) \approx U^*(s)$$

2.3 Are there any starting states where the agent cannot catch the prey? Why?

No, there are no starting states where the agent cannot catch the prey. The agent can start in a position as long as it is not the location of the prey or the predator. The utilities of all the states except for when $z_{agent} = z_{prey}$ or $z_{agent} = z_{predator}$ are less than 0, which means that there is a minimum expected number of rounds to catch the prey from each agent location. Those utilities are less than 0 as the predator is easily distracted, so we cannot be certain of the exact move the predator will take. Therefore, the agent has the potential to capture the prey from each valid starting state of the game.

2.4 What is the state with the largest possible finite value of U^* and give a visualization of it.

We determined $(z_{agent} = 5, z_{prey} = 24, z_{pred} = 24)$ to be the state with the largest finite possible value of $U*$, whose optimal utility is 19.94595. In other words, on average, the minimal expected number of rounds to catch the prey from $(5, 24, 24)$ is 19.94595, which is seen in Figure 2.

2.5 How does an agent based on the values of U^* compare with $A1$ and $A2$? Compare the number of steps it takes for the agents to capture the prey.

In the complete information setting, $A1$, $A2$, and an agent $A1RL$ based on the values of U^* have an average success rate of 82.30%, 99.83%, and 100.00% respectively (Figure 3). The average number of steps for the agents to capture the prey is 15.95, 73.94, and 8.82 respectively (Figure 4). We ran $A1$ and $A2$ on G and accounted for the predator being easily distracted.

The U^* agent's better performance and lower number of steps can be attributed to the fact that the agent moves based on the action with the minimal expected number of rounds to catch the prey. $A1$ and $A2$, in contrast, take more steps before capturing the prey; their movements are solely determined based on the current state of the game. $A2$, in particular, greedily selects actions that

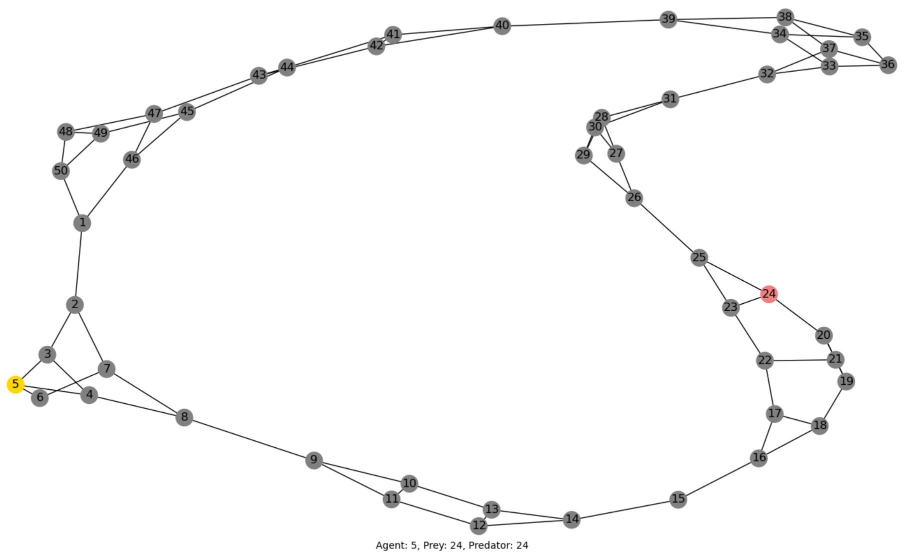

Figure 2: $U^*((5, 24, 24)) = 19.94595835427243$ is the state with largest finite value of U^*

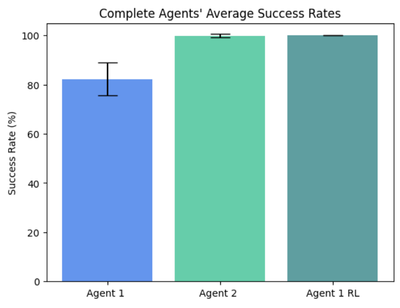

Figure 3: Average Success Rates

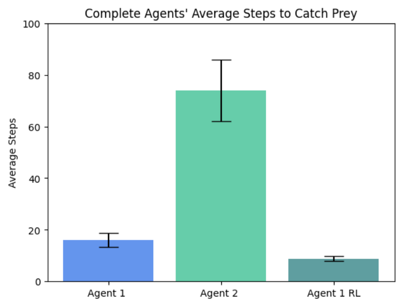

Figure 4: Average Steps

bring it closer to the prey until the predator is within 2 steps. In that case, it maximally runs away from the predator until the game ends or it is no longer threatened. The algorithm, though producing a similar success rate to that of the U^* agent, does not look into the future; $A2$ makes suboptimal movements, substantially increasing the number of steps taken to catch the prey.

2.6 Visualize and explain a state where U^* agent makes a different choices from $A1$ and $A2$.

The state $(z_{agent} = 35, z_{prey} = 46, z_{pred} = 2)$ (Figure 5) is an example of a case in which $A1$, $A2$, and the U^* agent make different choices. $A1$ chooses to move to 36 as that is farthest from the predator's current location. $A2$ chooses to move to 34 as that is the closest to the prey's current location, and there is no immediate threat of the predator. The U^* agent chooses to move to 38. The utilities for the U^* agent to move to an action in its action space are as follows $34 : -8.120, 36 : -11.869, 38 : -8.120, 35 : -9.120$. 34 and 38 have the highest utilities, indicating that the number of rounds to catch the prey is at minimum from one of these two locations. The U^* agent chooses the greatest reward action it saw last, which in this case was 38.

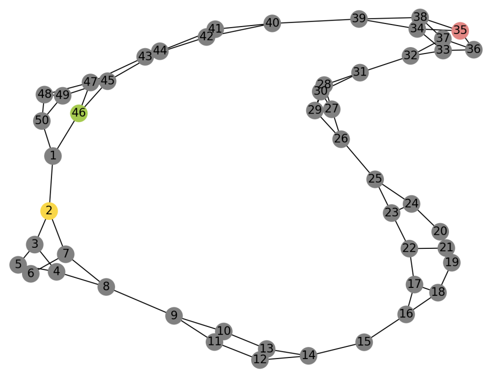

Figure 5: A1, A2, A1RL Different Choices Graph

3 Approximate Optimal Complete Information Agent

3.1 How do you represent the states s as input for the model? What features are relevant?

Naively, one might decide to take the positions of the agent, prey, and predator's location as a 1×3 vector, $[z_{agent}, z_{prey}, z_{pred}]^T$. This approach encodes spurious structures in the data that might not be present. For example, saying that $z_{pred} = 3$ and $z_{prey} = 1$ erroneously implies relative importance to the predator over the prey because $3 > 1$. Our approach is defined below:

The most relevant features are the location of the agent, prey, and predator. We generate one-hot vectors for each entity in the tuple $(z_{agent}, z_{prey}, z_{pred})$. We represent state s as a data observation by concatenating the 3 one-hot vectors of size 1×50 into a single vector $\mathbf{x}_i$ of size 1×150. So, $\mathbf{x}_i = [\mathbf{z}^{(i)}_{agent}, \mathbf{z}^{(i)}_{prey}, \mathbf{z}^{(i)}_{pred}]^T, \forall \mathbf{x}_i \in \mathcal{D}$, where $\mathcal{D} = \{(\mathbf{x}_i, y_i) : i = 0, 1, \dots, m\}$. For example, suppose $s_1 = (1, 2, 49)$. Then, $\mathbf{x}_1 = [1, 0, 0, \dots, 0, 1, 0, \dots, 0, 1, 0]^T \in \mathbb{R}^{1\times 150}$.

3.2 What model are you taking V to be? How do you train it?

We use a highly parameterized composition of differentiable non-linear functions, colloquially known as a neural network, for our function approximation model V for $U^*(s)$. Our basic computational unit is a neuron, which is composed of an activation function $\sigma(x)$ which takes an input vector and outputs another vector. These neurons are stacked to form a layer, which is depth-wise connected to other layers; this results in the approach colloquially known as "deep learning". We train the model using a gradient descent optimizer to push information from the data into the parameters of the network with backpropagation. $V \approx U^*(s)$ is defined precisely below along with training details:

Suppose f_1, f_2, f_3, f_4 are linear functions with respect to the input $\mathbf{x}$ such that $f_i(\mathbf{x}) = \mathbf{W} \cdot \mathbf{x} + \mathbf{b}$. The non-linear activation function $a_i(\mathbf{x}) = \tanh(\mathbf{x})$. With those preliminaries, we define V, the neural network as follows: $V = f_4(a_3(f_3(a_2(f_2(a_1(f_1(x)))))))$. Note the dimensionalities:

$$\begin{tikzcd} f_1 \arrow[r] & a_1 \arrow[r] & f_2 \arrow[r] & a_2 \arrow[r] & f_3 \arrow[r] & a_3 \arrow[r] & f_4 \end{tikzcd}$$

$$\begin{array}{ll}
f_1(\mathbf{x}) : \mathbb{R}^{1\times 150} \to \mathbb{R}^{150\times 150} & a_1(\mathbf{x}) : \mathbb{R}^{150\times 150} \to \mathbb{R}^{150\times 150} \\
f_2(\mathbf{x}) : \mathbb{R}^{150\times 150} \to \mathbb{R}^{150\times 150} & a_2(\mathbf{x}) : \mathbb{R}^{150\times 150} \to \mathbb{R}^{150\times 150} \\
f_3(\mathbf{x}) : \mathbb{R}^{150\times 150} \to \mathbb{R}^{150\times 150} & a_3(\mathbf{x}) : \mathbb{R}^{150\times 150} \to \mathbb{R}^{150\times 150} \\
f_4(\mathbf{x}) : \mathbb{R}^{150\times 1} \to \mathbb{R} &
\end{array}$$

Forward propagation :

$$\mathbf{x} \to V_1(x) \to V_2(V_1(x)) \to \cdots \to V_L(V_{L-1}(\cdots V_2(V_1(x))\cdots)) \to \hat{y}$$

Loss function:

$$\mathcal{L}(\hat{y}, y) = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

Backpropagation:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \frac{1}{m}\frac{\partial}{\partial \hat{y}}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial \hat{y}}(y_i^2 - 2y_i\hat{y}_i + \hat{y}_i^2) = \frac{1}{m}(-2y + 2\hat{y}) = -\frac{2}{m}(y - \hat{y})$$
$$\frac{\partial \hat{y}}{\partial V_L(V_{L-1}(\cdots V_2(V_1(x))\cdots))} = \frac{\partial \hat{y}}{\partial V_L}\cdot\frac{\partial V_L}{\partial V_{L-1}}\cdot\frac{\partial V_{L-1}}{\partial V_{L-2}}\cdot\cdots\cdot\frac{\partial V_2}{\partial V_1}\cdot\frac{\partial V_1}{\partial x}$$

where the derivatives of the activation functions and linear functions are given by:

$$\frac{\partial a(x)}{\partial x} = f'(x), \quad \frac{\partial f(x)}{\partial x} = \mathbf{W}^T$$

We then compute gradient of the loss function with respect to the network's parameters as follows:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}V_L} = \frac{\partial \mathcal{L}}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial V_L(V_{L-1}(\cdots V_2(V_1(x))\cdots))}\cdot\frac{\partial V_L(V_{L-1}(\cdots V_2(V_1(x))\cdots))}{\partial \mathbf{W}V_L}$$
$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}V_L} = \frac{\partial \mathcal{L}}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial V_L(V_{L-1}(\cdots V_2(V_1(x))\cdots))}\cdot\frac{\partial V_L(V_{L-1}(\cdots V_2(V_1(x))\cdots))}{\partial \mathbf{b}V_L}$$

where $\frac{\partial L}{\partial \hat{y}}$ and $\frac{\partial \hat{y}}{\partial V_L}$ are computed as previously described, and $\frac{\partial x}{\partial \mathbf{W}V_L}$ and $\frac{\partial x}{\partial \mathbf{b}V_L}$ are the gradient of the input x with respect to the weights and biases of the L-th layer of the network respectively. These gradients are computed recursively using the chain rule. Once the gradient of the loss function with respect to the network's parameters is computed, we can use it to update the network's parameters using a gradient descent optimizer. This process is repeated until the model converges and is able to accurately predict the target outputs from the input features.

To summarize, we take in an observation about our environment s_i and represent that as $\mathbf{x}_i = [\mathbf{z}_{agent}^{(i)}, \mathbf{z}_{prey}^{(i)}, \mathbf{z}_{pred}^{(i)}]^T, \forall \mathbf{x}_i \in \mathcal{D}$, where $\mathcal{D} = \{(\mathbf{x}_i, y_i) : i = 0, 1, \ldots, m\}$. s_i, represented as $\mathbf{x}_i$, is fed as input to V. Initially, the weights $\mathbf{W}$ and biases $\mathbf{b}$ in each of the linear functions f_i are randomly initialized. Hence, the model $V(\mathbf{x}_i)$ is initially not good at predicting the optimal utility $U^*(s_i)$ when the model runs forward propagation. To push information from our dataset $\mathcal{D}$ of $m = 125,000$ utilities into the parameters of our network V, we implement backpropagation.

We run stochastic gradient descent to train our model since we can frame predicting optimal utilities as a supervised learning problem: $\mathbf{x}$ are the input features to the problem and $\mathbf{y}$ is the target prediction from the inputs. Given all the targets $\mathbf{y} = [y_1, y_2, \ldots, y_m]$ and predictions $\hat{\mathbf{y}} = [V(\mathbf{x}_1), V(\mathbf{x}_2), \ldots, V(\mathbf{x}_m)]$, we compute the mean-squared error loss $\mathcal{L} = \frac{1}{m}\sum_{i=0}^{m}(\hat{y} - y)^2$.

The gradient of the loss function with respect to the network's parameters is then calculated using the chain rule, and the parameters are updated in the opposite direction of the gradient. This process is repeated for each iteration of the training process until the model reaches convergence. Let V be a neural network with L layers, where V_l denotes the l-th layer of the network, and x is the input to the network. We define the mean squared error loss function as $\mathcal{L}(\hat{y}, y) = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$, where $\hat{y} = V(x)$ is the predicted output and y is the target output.

3.3 Is overfitting an issue here? How accurate is V?

No, overfitting our model V to U^* is not an issue in this scenario. This is because the dataset $\mathcal{D}$ of $m = 125,000$ examples derived from U^* contain all possible scenarios that will be seen during test-set evaluation since this a complete information setting and we can enumerate all possibilities. In other words, our training set and testing set come from the same distribution. Put differently, we want to build a data-efficient model V, which can be stored as a binary for the query at inference time, instead of storing in memory every possible state and their optimal utility. The model V predicts U^* with a mean squared error of ≈ 0.35, which is trained after 130 epochs on a dataset of $m = 125,000$ examples. This means, on average, the squared difference between U^* and V given a state is 0.35.

3.4 How does the V agent compare in performance to the U^* agent?

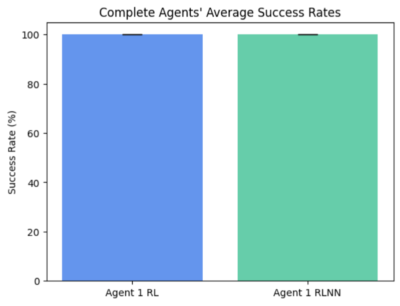

Figure 6: Average Success Rates

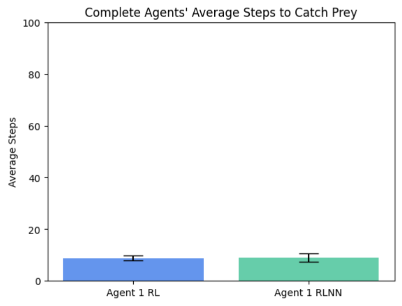

Figure 7: Average Steps

The U^* and V agents are in the complete information setting and have an average success rate of 100.00% each. The average number of steps it took for the agents to capture the prey is 8.82 and 8.98, respectively. The difference in the number of steps can be attributed to the V agent's mean squared error (MSE). It measures the average square difference between the predicted and actual utilities. The low MSE of 0.35 for V indicates that the model is predicting utilities close to the actual utility. Therefore, the V agent does not take a significant amount of more steps when compared to the to U^* agent to capture the prey.

4 Heuristic Partial Prey Agent

We estimate the value of the utility of each state $U_{partial}$ below, where $\mathbf{p}_{prey}$ is a probability distribution vector of where the prey could be with belief at each of the 50 nodes:

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

Given the utilities $U_{partial}$, we build an agent $A3RL$ to follow the following policy:

$$\pi_{U_{partial}} = \underset{a \in \mathcal{A}(s)}{\arg\max} \left[r_{s,a} + \sum_{s'} P(s'|s,a) \cdot U_{partial}(a, \mathbf{p}_{prey}, s_{predator},) \right]$$

where:

$$r_{s,a} = \begin{cases} -\infty & \text{if a} = \texttt{pred.loc} \\ -1 & \text{otherwise} \end{cases}$$

4.1 How does the Heuristic Partial Prey agent compare to $A3$ and $A4$ from Project 2?

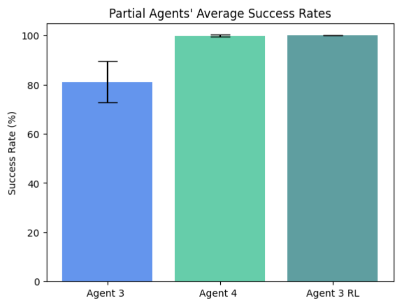

Figure 8: Average Success Rates

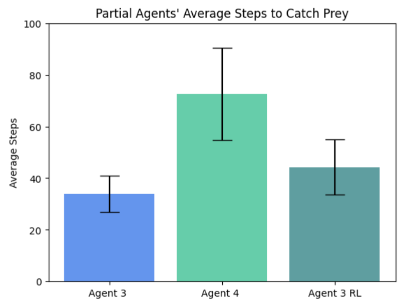

Figure 9: Average Steps

In the partial-prey information setting, $A3$, $A4$, and an agent $A3RL$ based on the values of $U_{partial}$ have success rates 81.10%, 99.93%, and 100.00% respectively (Figure 8). The average number of steps for the agents to capture the prey is 33.90, 72.50, and 44.26 respectively (Figure 9). We ran $A3$ and $A4$ on G and accounted for the predator being easily distracted.

The $U_{partial}$ agent has a better performance but a greater number of steps than $A3$. The average number of steps is calculated based on the cases in which the agent succeeds, which is the lowest for $A3$. Moreover, when $A3$ succeeds, its algorithm works well - a sign that the agent can quickly move to the prey. It follows that the average of $A3$'s successes will take a smaller number of steps.

The $U_{partial}$ agent has better performance and a lower number of steps compared to $A4$. $A4$ takes more steps before it captures the prey as it moves according to the greedy algorithm followed by $A2$. Though the algorithm produces a similar success rate to that of $U_{partial}$, it prompts the $A4$ to make suboptimal movements. As a result, it takes the $A4$ agent a greater number of steps to catch the prey.

4.2 Do you think this Heuristic Partial Prey agent is optimal?

While this agent does have a 100% success rate, it is not optimal. Through empirical analysis, we observe that the expected rounds to catch the prey are not necessarily as small as possible, meaning this agent is inefficient. Simply, the agent is using the expected utility of the state for all combinations of where the prey could be, $U_{partial}$. $U_{partial} \neq U^*_{partial}$ because we are not using the optimal utility of the state for the probability distribution of where the prey could be. Hence, $U_{partial}$ is not optimal.

5 Approximate Heuristic Partial Prey Agent

5.1 How do you represent the s as input to the model?

We represent a state s very similarly for $V_{partial}$ as in V. Like before, we convert $s = (z_{agent}, z_{prey}, z_{pred})$ into a 1×150 dimensional vector. But, instead of having $\mathbf{x}_i = [\mathbf{z}^{(i)}_{agent}, \mathbf{z}^{(i)}_{prey}, \mathbf{z}^{(i)}_{pred}]^T$ for the input representation, we replace $\mathbf{z}_{prey}$ with the probability distribution vector $\mathbf{p}^{(i)}_{prey}$. Therefore, our input to the model will be $\mathbf{x}_i = [\mathbf{z}^{(i)}_{agent}, \mathbf{p}^{(i)}_{prey}, \mathbf{z}^{(i)}_{pred}]^T$. Suppose $s_1 = (1, [0.25, 0.75, \ldots], 49)$. Then, $\mathbf{x}_1 = [1, 0, 0, \ldots, 0.25, 0.75, 0, \ldots, 0, 1, 0]^T \in \mathbb{R}^{1\times 150}$.

5.2 What model is $V_{partial}$? How do you train it?

We use the same neural network and architecture from the V in the complete information setting to train the $V_{partial}$ model. Since the input features are of the same dimension, we can save on compute and training time by initializing $V_{partial}$ with the parameters of the model of V before training on the dataset of $\mathcal{D}$ of $U_{partial}$ values and the corresponding inputs $\mathbf{x}_i$. We train $V_{partial}$ using backpropagation with stochastic gradient descent.

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

Observe that there are infinite possible states for $\mathbf{p}_{prey}$. Hence, we cannot enumerate through every possible combination of the state space. Therefore, we need a generalized model that can approximate any $U_{partial}$ for any state given only a subset of the data of every possible belief state. Hence, overfitting the training data might be an issue since it might make our model not robust to generalization. Though there are various regularization methods in neural networks, like augmenting the loss objective with ridge/lasso regression or dropout, we choose to use the regularization technique of early stopping. Using the MNIST dataset as a proxy for appropriate training and testing dataset sizes, we collect a training dataset $\mathcal{D}_{partial} = \{(\mathbf{x}_i, y_i)\}_{i=1,2,\ldots,m}$ for $m = 60,000$ observations and a validation dataset $\mathcal{V}_{partial} = \{(\mathbf{x}_i, y_i)\}_{i=1,2,\ldots,m'}$ for $m' = 10,000$ observations. After training $V_{partial}$ initialized with V's parameters on $\mathcal{D}_{partial}$, we evaluate $V_{partial}$ on $\mathcal{V}_{partial}$ at every epoch. When we noticeably observe the error for $V_{partial}$ decreasing on $\mathcal{D}_{partial}$ and increasing on $\mathcal{V}_{partial}$, we terminate training the model since our model is now robust to input data variations.

5.4 How accurate is $V_{partial}$? How can you judge this?

As mentioned before, we use a validation dataset $\mathcal{V}_{partial}$ that does not include data seen in the training. Our performance on that dataset should be a good proxy for how accurate $V_{partial}$ is. We decided to save a model with the following mean squared errors for $\mathcal{D}_{partial}$ and $\mathcal{V}_{partial}$: $MSE_{\mathcal{D}} = 0.015718$ and $MSE_{\mathcal{V}} = 0.0532223$ respectively. We then collected a new dataset of observations $\mathcal{T}_{partial} = \{(\mathbf{x}_i, y_i)\}_{i=1,2,\ldots,131528}$ to be a testing set to see how accurate $V_{partial}$ is. On dataset, $\mathcal{T}_{partial}$, our model $V_{partial}$ has an average 0.10888277 mean squared error.

5.5 Is $V_{partial}$ more or less accurate than just replacing U^* with V in $U_{partial}$?

We predict $U_{partial}(s_{agent}, \mathbf{p}_{prey}, s_{predator}) \approx \sum_{s_{prey}} p_{s_{prey}} \cdot V(\mathbf{z}_{agent}, \mathbf{z}_{prey}, \mathbf{z}_{pred})$ and evaluate it's performance on $\mathcal{T}_{partial}$, described in the prior subsection. On, $\mathcal{T}_{partial}$, we observe the total average MSE testing error on $131,528$ examples is 1.4842015808503954. Hence, since the error with respect to $\mathcal{T}_{partial}$ for $V_{partial}$ is 0.10888277, $U_{partial}(s_{agent}, \mathbf{p}_{prey}, s_{predator}) \approx \sum_{s_{prey}} p_{s_{prey}} \cdot V(\mathbf{z}_{agent}, \mathbf{z}_{prey}, \mathbf{z}_{pred})$ is less accurate than $V_{partial}$.

5.6 How does the $V_{partial}$ agent compare in performance to the $U_{partial}$ agent?

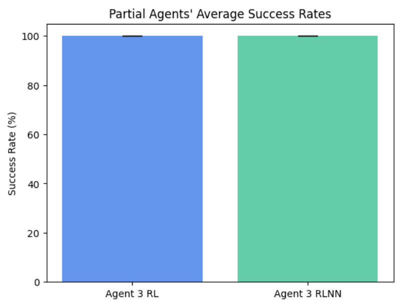

Figure 10: Average Success Rates

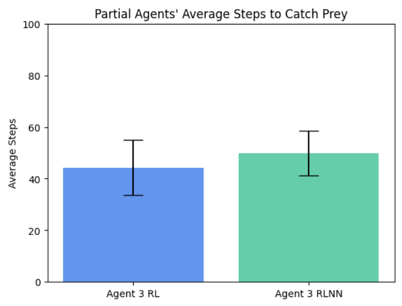

Figure 11: Average Steps

The $U_{partial}$ and $V_{partial}$ agents are in the partial-prey information setting and have an average success rate of 100.00% each. The average number of steps for the agents to capture the prey is 44.26 and 49.95 respectively. The difference in the number of steps can be attributed to the $V_{partial}$ agent's mean squared error (MSE). It measures the average square difference between the predicted and actual utilities. We save a model if its MSE is less than 5. That low MSE indicates that the model is predicting utilities close to the actual utility. Therefore, the $V_{partial}$ agent does not take a significant amount of more steps when compared to the $U_{partial}$ agent to capture the prey.

6 Optimal Partial Prey Agent

To predict the utility of the $U_{partial}$ agent, we must take a continuous input state and produce an output describing its utility.With an infinite input space, there is no way we can store tables with utility values and run a value iteration or a policy iteration algorithm.

6.1 Approach 1: Single Neural Network Deep Q Learning

The first approach we used was to use Deep Q Learning to try and estimate the utility of a state and action - that is, to estimate $Q(s, a)$ for some state s and some action a. We use a neural network that takes a 1×150 vector S. We define $\mathbf{S} = [\mathbf{x}_{agent}, \mathbf{x}_{belief}, \mathbf{x}_{predator}]^T$. We define $\mathbf{x}_{agent}$ as a 1×50 vector as such: $\mathbf{x}_{agent} = [0, 0, \ldots 0, 1, 0, \ldots, 0]^T$, where the 1 is at the agent's position. Similarly, $\mathbf{x}_{predator}$ as a 1×50 vector as such: $\mathbf{x}_{predator} = [0, 0, \ldots 0, 1, 0, \ldots, 0]^T$. We define $\mathbf{x}_{belief}$ as a 1×50 vector defining the belief system for the prey. For example, if the prey has a 50% chance of being at node 1 and a 50% chance of being at node 2, $\mathbf{x}_{belief} = [1, 1, 0, 0, \ldots, 0]^T$.

The neural network takes $\mathbf{S}$ and passes it through 3 layers, each of size 150. The output is a 1×50 vector $\mathbf{O}$. We can define $\mathbf{O} = [Q(s_1, a_1), Q(s_2, a_2), \ldots, Q(s_{50}, a_{50})]^T$. Here, a_i is the action that moves the agent to the ith node, and s_i is the system's state when the agent moves to the ith node. Of course, not all of the actions are legal. To use $\mathbf{O}$, the agent only considers legal actions.

To train this neural network, we maintain an array of live simulations J of size 1000. We then update the neural network in iterations. In each iteration, we pick 150 random simulations from J without repeats. For each simulation, let s be the current state of this simulation. Then, we use the neural network to calculate $Q(s, a)$ and get a vector I. After, we calculate $Q(s, a)$ for all legal actions in a value iteration matter. Given that action a takes state s to s', we calculate the future reward of being in s' by inputting s' into the neural network and taking the max utility of the legal actions from s'. Note that the probabilities of the predator and the prey moving are not considered in this calculation. Adding -1 to this as the current reward, we then form a new vector P. P is the exact copy of I except for the legal actions that can be taken from s. For terminal states, such as when the agent is holding the prey and when the predator is holding the agent, the utility is predetermined as 0 and -50, respectively. Note that while the utility for the predator holding the agent should be $-\infty$, we choose -50 as to not make the neural network unstable. For each iteration, we form parallel arrays of these initial network evaluations Is and these value-iteration-based evaluations Ps. After every iteration, these arrays are backpropagated, one initial network and value-iteration-based evaluation at a time.

Since simulations are chosen randomly and never more than once in an iteration, data used to train the neural network is 'independent'. If the same simulation was used repeatedly to train the neural network, a lot of the data could end up being very correlational. Further, an ϵ-greedy approach is used to advance a simulation. This allows us to balance the exploitation intention to fully map out optimal spaces with the exploration intention to try new options. Third, if a simulation in the game vector J ends, it is replaced by a new game. Finally, this algorithm continues until the average loss between the input and output is less than 0.1. The algorithm is more explicitly described in **Algorithm 2**

6.2 Approach 2: Double Neural Network Deep Q Learning

Unfortunately, the approach fails to be effective. The error never goes down, meaning that the neural network never converges properly. The reason for this is most likely because the neural network is trying to converge to a moving target. As the network back propagates the errors, the value iteration predictions will also change; the network can not converge if its predictions keep changing. To combat this, we used a second neural network called the 'target network.' This network, initialized to the same weights as the main network, is meant to be used as the utility function for computations relating to value iteration. The intention is to have the main network converge to this target network before updating the target network with the weights of the main network and having it try to converge again. While this is still, in some form a moving target, it does allow the main network to converge to something first before updating the target.

Using this approach, other changes were also made. With the complexity of the output space - it is not a constant size - the purpose of the neural networks shifted to simply outputting the utility of being in one state. The neural network no longer produces a vector of $Q(s, a)$, but rather one singular

Algorithm 2 Single Neural Network Deep Q Learning

```
q_func = initalize_neural_network()
J = [init_sim() for $i$ in range(0, 1000)]
repeat
    init_evals = [], predictions = []
    for $j$ states to process in iteration size do
        $s$ = pick_random_state(J)
        $I$ = q_func(s)
        $P$ = $I$
        for action $a$ in action space of $s$ do
            $P[a]$ = value_iteration()
        end for
        init_evals.append($I$)
        predictions.append($P$)
        $a$ = $\epsilon$-greedy()
        $s'$ = advance_simulation($a$)
        save_simulation_in_J($s'$)
    end for
    for ($I$, $P$) in (init_evals, predictions) do
        backpropagate $P$, $I$ through q_func()
    end for
    compute average loss
until average loss < $0.1$
```

$Q(s)$. The neural network's structure changes in accordance, taking 150 inputs with 3 hidden layers of size 150 each before outputting one value. The rest of the algorithm is the same.

Note that the target network's weights are updated every few iterations, collectively called a 'batch.' Further, the future reward is calculated by considering the places the predator could move. The predator is easily distracted, so with a 60% chance, it moves optimally and a 40% chance it moves randomly. We do not consider all of the places the prey could be based on the belief system, as we did not need to in the V_partial setting and the performance still came out well. When states are passed to the target network to calculate the future reward, the belief system for the prey isn't updated. The algorithm is described more explicitly in **Algorithm 3**

Unfortunately, this approach, while closer, doesn't converge either. This could largely be due to the way that the neural network back propagates the errors. We find that switching to a mini-batch backpropagation helps the situation by dropping the error to around 1-2 in the beginning, but this fails to converge as well. However, if the network had converged, the solution would be optimal. Value iteration ensures that our utilities our optimal; this algorithm simply generalizes value iteration to work with an infinite state space and a set of terminal states.

6.3 Other Approaches Considered

We also consider two other approaches. First, we looked at discretizing the model space. However, this doesn't build an optimal solution; an optimal solution cannot simply overextend solutions for a state to cover possibly very different states that meet specific criteria.

Another solution that was considered was using an actor-critic model. Here, one neural network - called the actor - would make a decision. A second neural network - called the critic - would judge the decision. While sounding promising to develop an optimal utility function, it is extremely similar to the double neural network deep Q learning approach used earlier. If the double neural network deep Q learning approach failed to converge, it's unlikely that the actor-critic model would converge.

Algorithm 3 Double Neural Network Deep Q Learning

```
q_func = initalize_neural_network(), target_net = copy(q_func)
J = [init_sim() for $i$ in range(0, 1000)]
**repeat**
    **for** $k$ iterations in batch size **do**
        init_evals = [], predictions = []
        **for** $j$ states to process in iteration size **do**
            $s$ = pick_random_state(J)
            $i$ = q_func(s)
            $p$ = value_iteration(target_net)
            init_evals.append($i$)
            predictions.append($p$)
            $a$ = $\epsilon$-greedy()
            $s'$ = advance_simulation($a$)
            save_simulation_in_G($s'$)
        **end for**
        **for** ($i$, $p$) in (init_evals, predictions) **do**
            backpropagate $i$, $p$ through q_func()
        **end for**
    **end for**
    target_net = copy(q_func)
    compute average loss
**until** average loss < 0.1
```