CS 520 - Project 3: Better, Smarter, Faster

Akanksha Reguri (ar2085), Khyati Doshi (kbd57)

December 15, 2022

1 Abstract

The problem statement is to build an agent that captures the prey as efficiently as possible.

2 The Environment

The Environment is a graph of 50 nodes, numbered 1 to 50 connected by edges. The environment consists of 3 entities - the prey, the predator, and the agent. The agent is in search of prey and the predator is in search of the agent. The agent, the prey, and the predator move between the nodes connected by edges. The nodes are connected in the circle, and additionally, random edges are added to increase connectivity in the circle. Random edges are added based on the following rules:

1. Picking a random node with a degree less than 3
2. Adding an edge between it and one node within 5 steps forward or backward along the primary loop. (So node 10 might get connected to node 7 or node 15, but not node 16.)
3. This is repeated until no more edges can be added. Since there is already an edge from the nodes X to X-1 and X to X+1, we won't be adding another edge to these nodes while adding an additional edge from X.

When an environment is created by calling the class The_Environment, the following entities are created which are unique to each and every environment:

- Edges - Connection between the nodes. This is a dictionary where keys are the node numbers and values are the list of neighboring node numbers. We are using an adjacency list instead of an adjacency matrix to save space as in our use case each vertex will only be connected to at most 3 other vertices.
- Prey location - Location of the birth of the prey
- Predator location - Location of the birth of the predator
- Agent location - Location of the birth of the agent

The agent at birth can't occupy a node that is already occupied so, the environment created makes sure that:

agent-birth-location != predator-birth-location and agent-birth-location != prey-birth-location

2.1 Easily Distracted Predator

The goal of the predator is to catch the agent. This predator gets easily distracted - with a probability of 0.6, the predator will move to close the distance, but with a probability of 0.4, the predator moves to one of its neighbors uniformly at random. This means that on average the predator is moving to the neighbor that is closer to the agent, but it is not guaranteed in any round.

2.2 The Prey

The rules of the prey are simple. Prey chooses uniformly at random one of the locations from its neighbors and includes its own location (that is if the prey has three neighbors, there is a 1/4 probability of staying where it is). This continues, regardless of the actions or locations of the others, until the game concludes.

2.3 The Agent

Agent's goal is to catch the prey before the predator catches the agent.

3 U-Star

For a given s, U^* = minimal expected number of rounds to catch the prey. A state is something that defines the location of prey, predator, and agent. state = (agent, prey, predator). Of the entire state space (125000 states possible), states are divided into three categories.

State Space:

- Good Terminal States - A state is a good terminal state if the agent caught the prey and there is no predator in that cell. For every state belonging to a good terminal state, U*(s) = 0, since the agent and prey are in the same location, the game ends and no more rounds are needed
- Bad Terminal States - A state is a bad terminal state if the predator caught the agent. For every state belonging to a bad terminal state, U*(s) = INFINITE, since the agent is caught by the predator and it can never catch the prey.
- Intermediate States - A state which is neither good nor bad is an intermediate state

Action Space: Since the game starts with Agent's movement, followed by prey and predator. Here the action space is the movements of the agent. Hence the number of actions would be the number of nodes the agent can choose to move to. For every action the agent can take, there are respective prey and predator actions.

3.1 Algorithm to determine U^*

Since the goal is to build an agent that captures the prey as efficiently as possible(least number of rounds), to build this optimal policy, an optimal value function is required. The algorithm used to compute this optimal value is ***value iteration*** which can be shown to converge to final U^* values.

U^*(s) = R_s + $\min_{a \in S}(\Sigma_{next-s} P^a_{transition(s,next-s)}$ U*(next-s))

Here R(s) is the reward of the state s. $P_{transition}$(s, next-s) is the probability with which next-s occurs from state s and U*(next-s) is the utility of next states.

3.1.1 Value Iteration Initialization

As explained above, U^*(s) for good terminal states is 0, for bad terminal states are INFINITE, and for intermediate states, it is initialized to the distance between agent and prey (since it is an approximate estimate for the number of rounds and this would converge to a definite value at the end).
R(s) is a reward that actually depicts the cost of the state to reach the end goal since value iteration is a minimization function
Rewards of states for different states:

- Good Terminal States - For good terminal states, reward(s) = 0, since there is no cost incurred as the agent caught the prey
- Bad Terminal States - For bad terminal states, reward(s) = INFINITE as the agent can't catch the prey at any cost.

- Intermediate States - For intermediate states, reward(s) = 1, as at least 1 round is required to further catch the prey.

```
def initialize()
    for every s of state-space:
        if s is a bad-terminal state:
            U*(s) = math.inf
            reward(s) = math.inf
        else if s is a good-terminal state:
            U*(s) = 0
            reward(s) = 0
        else:
            U*(s) = dist(agent, prey)
            reward(s) = 1
    return utility, reward
```

3.1.2 Value Iteration Algorithm

A state is defined as a tuple of (agent, prey, predator). For every state s, for every given action of s, all the possible next state utilities that action leads to are calculated. To calculate this next state utilities $P_{transition(s,next-s)}$ needs to be computed. This transition probability is the product of certainties of prey transition, agent transition, and predator transition since all those are independent actions. The agent moves to the next state with certainty. Hence $P_{transition(s,next-s,agent)}$ is 1. Prey moves at random to one of the neighbors including itself. So $P_{transition(s,next-s,prey)}$ is $1/(\#neighbors + 1)$. And predator moves to optimal choices with a probability of 0.6 and one of its neighbors with a probability of 0.4.

$$P_{transition(s,next-s)} = P_{transition(s,next-s,agent)} * P_{transition(s,next-s,prey)} * P_{transition(s,next-s,predator)}$$

And finally U^*(s) of a particular state is re-computed using the below formula.

$$U^*(\text{s}) = R_s + \min_{a \in S}(\Sigma_{next-s} P^a_{transition(s,next-s)} \text{ U*(next-s)})$$

Below code, snippet gives a clear explanation of how the value iteration is implemented.

```
# edges denote the dictionary where key is a node and values are it's neighboring nodes
def compute_prey_probs(prey):
    prey_probs = {}
    for pr in edges[prey] + prey
        prey_probs[pr] = 1/(len(edges[prey]) + 1)

def compute_pred_probs(predator, agent):
    pred_probs = {}
    optimal_choices = []
    neighbor_choices = []
    for pred in edges[predator]:
        if pred is optimal_choice:
            # if it is the shortest distance
            optimal_choices.append(pred)
        neighbor_choices.append(pred)
    for choice in neighbor_choices:
        pred_probs[choice] = 0
    for choice in optimal_choices:
        pred_probs[choice] = pred_probs[choice] + 0.6/optimal_choice_len
    for choice in neighbor_choices:
        pred_probs[choice] = pred_probs[choice] + 0.4/neighbor_choice_len
    return pred_probs
```

```
def value_iteration(edges, iterations):
    utility = {}, reward = {}, policy = {}
    initialize(utility, reward)
    for _ in iterations:
        util_new = deep-copy(utility)
        for every state s(agent, prey, predator):
            if s is a good-terminal state or bad-terminal state:
            # There is no need to compute utilities
                continue
            actions = edges[agent]
            prey_probs = compute_prey_probs(prey)
            pred_probs = compute_prey_probs(predator, agent)
            action_utils = {}
            for action in actions:
                # for every agent action
                if action == predator:
                    # agent is caught by the predator
                    action_utils[action] = infinite
                    continue
                if action == prey:
                    # agent caught the prey
                    action_utils[action] = 0
                    continue
                sum = 0
                for new_prey in edges[prey] + [prey]:
                    for new_pred in edges[predator]:
                        if util_new[action, new_prey, new_pred] == math.inf:
                            sum = sum + math.inf
                            continue
                        sum = sum + prey_probs[new_prey]*pred_probs[new_pred]*
                        util_new[action, new_prey, new_pred]
                action_utils[action] = sum
            # sort action_utils by value
            utility[s] = reward[s] + action_utils.values()[0]
            # doing argmax
            policy[s] = action_utils.keys()[0]
        diff =  get_max_diff(utility, util_new)
        if diff < 0.001:
            break
    return utility, reward, policy
```

3.1.3 Value Iteration Termination

There is no definite number of rounds at which Value Iteration would terminate. This algorithm would get terminated if the maximum difference between the previous round utilities and the current round utilities is 0.001. For our graph Value Iteration terminated in 8 minutes(38 rounds)

4 U-star Agent

U-star Agent runs in a complete information environment. As part of the value of iteration, the best possible action(policy) is calculated for each possible state(just argmax of value iteration - capturing the agent's action(policy) that leads to the minimum utility). U-star Agent simply follows the policy for a particular state.
For 3000 simulations
Survivability: 99.99 percent

Average Step Count: 8.65
Whereas the performance of Agent 1 is:
Performance: 85.5 percent
Average Step Count: 14.15
And the performance of Agent 2 is:
Performance: 99.43 percent
Average Step Count: 18.89

Thus U-star Agent is able to catch the prey more efficiently than Agents 1 and 2 (we can see survivability increased and average step count reduced).

5 Model V

To predict the value of U^*(s) for a state s, Model V is built.

5.1 Input Data

How do you represent the states s as input for your model? A state is a sequence of Agent, prey, and predator locations. Three features, Agent-location, prey-location, and Predator-location, are given as input to the model, which uniquely identifies a given state.
What kind of features might be relevant? Following features were considered as Input.

- Agent Location - Location of the Agent
- Prey Location - Location of the Prey
- Predator Location - Location of the Predator
- Agent-Prey distance - Shortest Distance between Agent and Prey.
- Agent-Pred distance - Shortest Distance between Agent and Predator

Both Agent-Prey distance and Agent-Predator distance play a vital role in impacting the number of rounds for the Agent to capture the prey since the Agent tries to move close to the Prey escaping from the Predator.
Apart from the above features, cubic-level features are also given as input to the model to speed up the convergence. Cubic features of all the five features above are calculated and given as input.

```
for every feature:
    cubic_feature = feature*feature*feature
```

Input data given to the model is of shape [10, m] where 10 is number of features and m is number of training examples

5.2 Output

Predicting the U^* of a particular state s.

5.3 Model Space

Deep Neural Networks are implemented to predict the U^* of a particular state s. Neural Network Architecture is as follows: It is a 1 input, 1 output, and 6 hidden layer network with the number of nodes being [10, 14, 18, 16, 12, 6, 4, 1] at each layer. 10 nodes in the input layer(since 10 features are given as input - actual + cubic combined), one node in the Output layer and 14 nodes in the first hidden layer and so on...

- Weights Initialization - **He-Normal** Initialization is used since it is suitable for layers where Leaky ReLU activation function is used.

```
# Weights of layer k
W[k] = np.random.randn(layers[k-1], layers[k])*np.sqrt(2/layers[k-1])
b[k] = np.zeros((layers_dims[k], 1))
```

- Activation Functions - Leaky ReLU activation function with an alpha of 0.01 is used for all the hidden layers, and the Linear Activation function is used for the Output layer.

5.4 Loss Function

Since this is a Regression problem to compute the loss, the mean square loss is chosen.

Here m is the number of training examples

$$\text{loss} = (1/\text{m})^* \Sigma_{i=1}^{m} (Y_i - Prediction_i)^2$$

5.5 Forward and Backward Propagation

5.5.1 Forward Propagation

Following pseudo code explains the forward propagation implementation.

```
def leaky_relu(Z):
    return maximum(Z, 0.01Z)

def forward_propagate(X, weights, biases):
    # Vectorized implementation where
        W[l] - weights of the layer
        A[l] - Output of the layer
        b[l] - Biases of the layer
        Z[l] - Dot Product of weights and activation of the previous layer
        l runs from 1 to 7 here for Model V
    caches = {}
    A[0] = X
    for every layer l:
        Z[l] = np.dot(W[l], A[l-1]) + b[l]
        if l is hidden:
            A[l] = leaky_relu(Z[l])
        if output_layer:
            A[l] = Z[l]
        caches[l] = (A[l-1], W[l], b[l], Z[l])
    # Since A[l] would be the final output
    return A[l], caches
```

5.5.2 Backward Propagation

Following pseudo code explains the backward propagation implementation.

```
def compute_dAL(AL, Y):
    # Loss differentiation with respect to prediction
    return 2*(AL - Y)

def dReLU(x, alpha=.01):
    # Leaky relu differentiation
    return np.where(x>=0, 1, alpha)

def backward(dZ, cache):
    A_prev, W, b = cache
    m = A_prev.shape[1]
```

```
        dW = (1/m)*np.dot(dZ, A_prev.T)
        db = (1/m)*np.sum(dZ, axis=1, keepdims=True)
        dA_prev = np.dot(W.T, dZ)
        return dA_prev, dW, db

    def activation_backward(dA, cache, activation):
        linear_cache, activation_cache = cache
        # contains, A_prev, w, b, Z of that particular layer
        if activation == "linear":
            dZ = dA
        elif activation == "relu":
            dZ = dA*dRelu(Z)
        return backward(dZ, linear_cache)

    def backward_propagate(AL, Y, caches):
        """ Arguments:
            AL -- output of the forward propagation
            Y -- true "label" vector
            caches -- list of caches captured during
            forward propagation
        """
        grads = {}
        L = len(caches) # number of layers
        m = AL.shape[1]
        dAL = compute_dAL(AL, Y) # calculating gradient

        dA_prev, dW, db = activation_backward(dAL, caches[L-1], "linear")
        grads["dA" + str(L-1)] = dA_prev
        grads["dW" + str(L)] = dW
        grads["db" + str(L)] = db

        # Loop from l=L-2 to l=0
        for l in reversed(range(L-1)):
            dA_prev, dW, db = activation_backward(dA_prev, caches[l], "relu")
            grads["dA" + str(l)] = dA_prev
            grads["dW" + str(l+1)] = dW
            grads["db" + str(l+1)] = db
        return grads
```

Equations -

For the last layer

$$dA[L] = 2 * (A[L] - Y)$$
$$dZ[L] = dA[L]$$
$$dW[L] = (1/m) * dZ[L]A[L-1]^T$$
$$db[L] = (1/m) * np.sum(dZ[L], axis = 1, keepdims = True)$$

For the rest of the layers

$$dA[L-1] = W[L]^T.dZ[L]$$
$$dZ[L-1] = leaky - relu - diff(dA[L-1])$$
$$dW[L-1] = (1/m) * dZ[L-1]A[L-2]^T$$
$$\text{db[L-1]} = (1/m) * np.sum(dZ[L-1], axis = 1, keepdims = True)$$

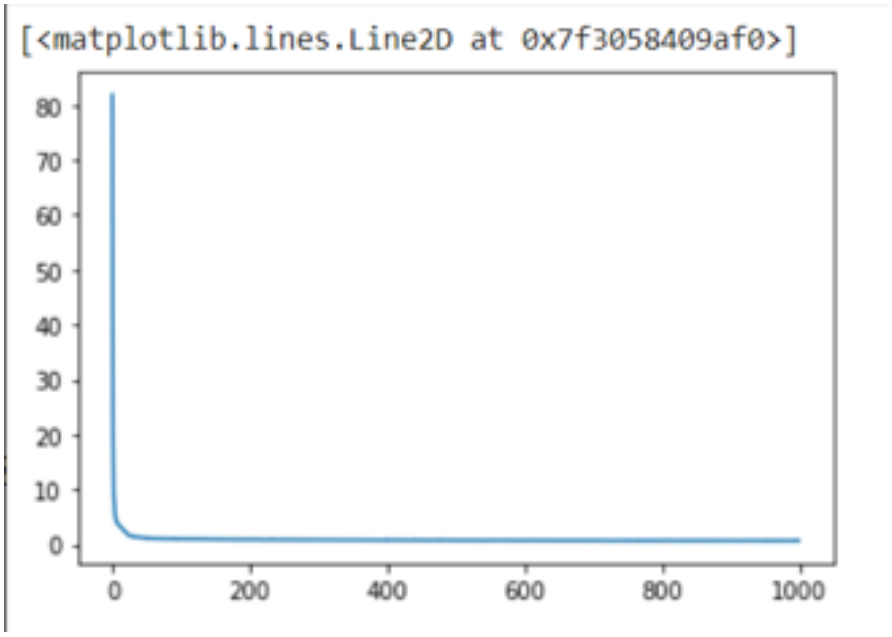

Figure 1: Cost during training is captured after every 20 iterations, Total iteration = 20K

5.6 Training Algorithm

Initially, Vanilla Gradient Descent is used for training the data. Later, Adam Optimization has been implemented for training. The model has been trained on all the 125k states available.

5.6.1 Data Cleaning

There are no duplicates and nan values, but the model isn't trained on those states where utility is infinity because the y value should be something finite to train a model. Also for the scenarios, where agent and prey are in the same location(good terminal state) or agent and predator in the same location(bad terminal state), it would be a direct decision(since they are initialization states) 0 or INFINITE respectively, so the final data to be trained doesn't contain these states.
Is overfitting an issue here?: Overfitting is not an issue here since the generalization states would be the same as training states and nothing else. Given a single graph, only 125k states are possible, and the model is trained on every state, so the model generalizes well (there is no possible scenario of unseen states). Hence, overfitting is not an issue here.

5.7 Accuracy of V

Model V converged at 0.13 loss after 20k iterations. Below described V-star Agent's accuracy defines the accuracy of Model V in more detail.

6 V-star Agent

V-star Agent runs in a complete information environment. Considering the next state possibilities, the V-star agent chooses an action that has the minimum possible utility. The pseudo-code is as follows:

```
def predict_model_v():
    # These predictions are captured once and
    # stored in a pickle for future
    utility = {}
    for every state:
        utility[state] = model_v.predict(state)
    return utility
```

```
def v_star_agent(utility):
    # Here utility is the dictionary containing state and the predicted values of Model V
    utils = {}
    if agent_loc == pred_loc:
        return "Predator caught the agent"
    if agent_loc == prey_loc:
        return "Agent caught the prey"
    for agent_choice in edges[agent]:
        if agent_choice == prey_loc:
            utils[agent_choice] = 0
            continue
        if agent_choice == pred_loc:
            utils[agent_choice] = INFINITE
            continue
        sum = 0
        for prey_choice in edges[prey]:
            for pred_choice in edges[predator]:
                if utility[agent_choice, prey_choice, pred_choice] == math.inf:
                    sum = math.inf
                    continue
                sum = sum + prey_probs[prey_choice]*pred_probs[pred_choice]*
                utility[agent_choice, prey_choice, pred_choice]
        utils[agent_choice] = sum
    sort(utils)  # by values - ascending
    agent_loc = utils.keys()[0]
    # The next step the v_star_agent takes
    move_prey()
    move_predator()
```

For 3000 simulations
Survivability: 99.99 percent
Average Step Count: 8.719
V star agent survived, as good as the U star agent. But the average step count of V star is slightly higher than average step count of U star. Difference in step count = 0.07

7 U-partial

For a given state s, U-partial is the sum of the probability of prey positions times the utility(fetched from u-star) of the given state (considering prey at every node in the graph). The agent location, predator location, and a vector of probabilities for the position of the prey define a state. The vector of prey probabilities is a 1X50 vector consisting of the probability of the node containing the prey.

- For a good terminal state since U*(s) = 0, u-partial(s) =0.
- For a bad terminal state since U*(s) = INFINITE, u-partial(s) = INFINITE.
- For all intermediate states U-partial(agent, [prey-probs], predator) = $\Sigma_{i \in nodes} Prey\_Probability(i) * U^*(agent, i, predator)$

7.0.1 Algorithm to calculate U-partial

```
if s is a bad-terminal state:
    U-partial(s) = math.inf
else if s is a good-terminal state:
    U-partial(s) = 0
else:
    U-partial(agent, [prey-probs], predator) = For every node i [ Sum(prey probabilities(i) X
    U*(agent, i, predator)) ]
```

8 U-partial Agent

U-partial Agent runs in a partial prey information setting. The Agent is always aware of where the predator is, but not where the prey is. U-partial Agent tracks a belief state for prey for each of the available nodes. These belief states are updated every time the Agent learns something about the prey and every time the prey moves. Based on these prey probabilities U-partial of the state is calculated.

```
def calculate_u_partial_state(utility, agent_loc, pred_loc, prey_prb):
    u_partial = 0
    for prey_loc in range(1, total_nodes+1):
        u_partial = u_partial + prey_prb[prey_loc]*utility[agent_loc, prey_loc, pred_loc]
    return u_partial

def u_partial_agent(utility):
    # Utility are the U*(s)
    if agent_loc == pred_loc:
        return "Predator caught the Agent"
    if agent_loc == true_prey_loc:
        return "Agent caught the prey"
    local_utility = {} # will contain the utility for each node agent can move to.

    #calculate u_partial for the current state
    u_partial_temp = calculate_u_partial(utility, agent_loc, pred_loc, prey_prob)
    ag_pred = get_distance(agent_loc, pred_loc, env.edges)
    ag_prey = calculate_ag_prey_distance(env.edges, agent_loc, prey_prob)

    for every agent_choice :
        if agent_choice == pred_loc :
            local_utility[agent_choice, pred_loc] = INFINITE # since the utility of the state
            is INFINITE
            continue
        else if agent_choice == true_prey_loc:
            local_utility[agent_choice, pred_loc] = 0 # since the utility of the state is 0.
        else:
            pred_prob = compute_pred_probs(env,pred_loc,agent_choice)
            sum = 0
            for all pred_choice :
                if agent_choice == pred_choice :
                    sum = INFINITE
                else :
                    u_part = calculate_u_partial_state(utility, agent_choice, pred_choice,
                    prey_prob)
                    sum = sum+ pred_prob[pred_choice]*u_part
            local_utility[agent_choice, pred_loc] = sum

    local_utility = sort local_utility

     # Here data for V_partial is captured
    capture_v_partial_data(agent_loc,pred_loc,prey_probs,ag_pred,ag_prey, u_partial_temp )
    agent_location = list(local_utility.keys())[0][0] # the node with the lowest utility
    is chosen
    # Here data for Bonus Agent1 would be captured.
    # For Bonus Agent 1, U-partial agent's utility is captured at this position
    capture_bonus_agent1_utility(list(local_utility.values())[0])
    move prey()
    update prey probabilities by movement()
    move predator()
```

U partial Agent Survivability is as follows
Survivability: 96.23 percent
Average Step Count: 23.47
Whereas the performance of Agent 3 is:
Survivability: 81.5 percent
Average Step Count: 27.3
And the performance of Agent 4 is:
Survivability: 91.56 percent
Average Step Count: 29.60
Thus U partial Agent is optimal since it's survivability and step count are much better than that of Agent 3 and 4.

9 V-partial Model

To predict the value of U-partial for a state s, Model V-partial is built.

9.1 Input Data

How do you represent the states s as input for your model? A state is a sequence of Agent, predator locations, and prey probabilities. Three features, Agent-location, predator-location, and Prey-probabilities, are given as input to the model, which uniquely identifies a given state.
What kind of features might be relevant? Following features were considered as Input.

- Agent Location - Location of the Agent
- Flattened Prey Probabilities - The list of prey probabilities are flattened to feed as input to the neural network
- Predator Location - Location of the Predator
- Weighted Agent-Prey distance - Shortest Distance between Agent and Prey location multiplied by the probability of prey location.

$$\text{weighted Agent-Prey distance} = \Sigma_{prey \in nodes} P(prey) * distance(Agent, Prey)$$

- Agent-Pred distance - Shortest Distance between Agent and Predator

Both Weighted Agent-Prey distance and Agent-Predator distance play a vital role in impacting the number of rounds for the Agent to capture the prey since the Agent tries to move close to the Prey escaping from the Predator.

9.2 Output

Predicting the U-partial of a particular state s.

9.3 Model Space

Model Space of V-partial is similar to that of V*. It is a Deep Neural Network with a number of hidden layers 9, one input, and one Output layer. Layers : [54, 60, 70, 80, 70, 60, 35, 20, 10, 6, 1] Each value in the list determines the number of nodes in that layer. There are 54 inputs (50 flattened prey probabilities and four other inputs).

9.4 Loss Function

Since this is a Regression problem to compute the loss, the mean square loss is chosen.

$$\text{loss} = (1/\text{m}) * \Sigma_{i=1}^{m} (Y_i - Pred_i)^2$$

9.5 Training Algorithm

Adam Optimization is used for training. The model has been trained on the 200k states collected(by simulating U-partial Agent). To ensure the data collected is diverse, (considering different agent, prey and predator locations) following approach is followed.

```
def data_collection():
    for agent in (1, 50):
        for predator in (1, 50):
            env.generate_prey_pred_agent_locations()
            env.agent_loc = agent
            env.predator_loc = predator
            u_partial_agent(env)
```

This collected 250k samples out of which 220k states are used for training and 30k for testing. Iterating over the Prey locations consumed 4 hours and still data collection wasn't complete, so we skipped that loop. Since there are multiple simulations here, multiple environment situations are captured and fed to the model, which makes the model more robust.

9.5.1 Data Cleaning

There occurred few duplicate training samples, since states might re-occur, the number of duplicate samples were around 3000 out of 250K. **Is overfitting an issue here?**: Overfitting is an issue here since unseen data might occur in the future because it is not just 125k states here but prey-probabilities is a 50*1 vector and there might be an infinite number of states possible.
What can you do about it?: To overcome Overfitting, dropout has been implemented. This avoids in model being oversensitive to particular features. The dropout ratio is varied from layer to layer. Dropout[0, 0.3, 0.4, 0.4, 0.3, 0.3, 0.2, 0.2, 0.2, 0.2, 0] - 0 dropout means there is no dropout for those layers (in case of input and output layers since we need all the features and all the outputs) The following pseudo-code explains how dropout has been implemented

```
def dropout(A, dropout_prob):
    # For a layer l's activation and dropout ratio
    # Columns - denote number of data points
    # Rows - output of every node of layer l
    for i in range(A.shape[0]):
        if random.random() < dropout_prob:
        # Making the output zero
            A[i][:] = 0
    return A

def forward_propagate(X, weights, biases):
    # Vectorized implementation where
        W[l] - weights of the layer
        A[l] - Output of the layer
        b[l] - Biases of the layer
        Z[l] - Dot Product of weights and activation of the previous layer
        l runs from 1 to 10
    caches = {}
    A[0] = X
    for every layer l:
        Z[l] = np.dot(W[l], A[l-1]) + b[l]
        if l is hidden:
            A[l] = leaky_relu(Z[l])
        if output_layer:
            A[l] = Z[l]
        A[l] = dropout(A[l], dropout[l])
        caches[l] = (A[l-1], W[l], b[l], Z[l])
    # Since A[l] would be the final output
```

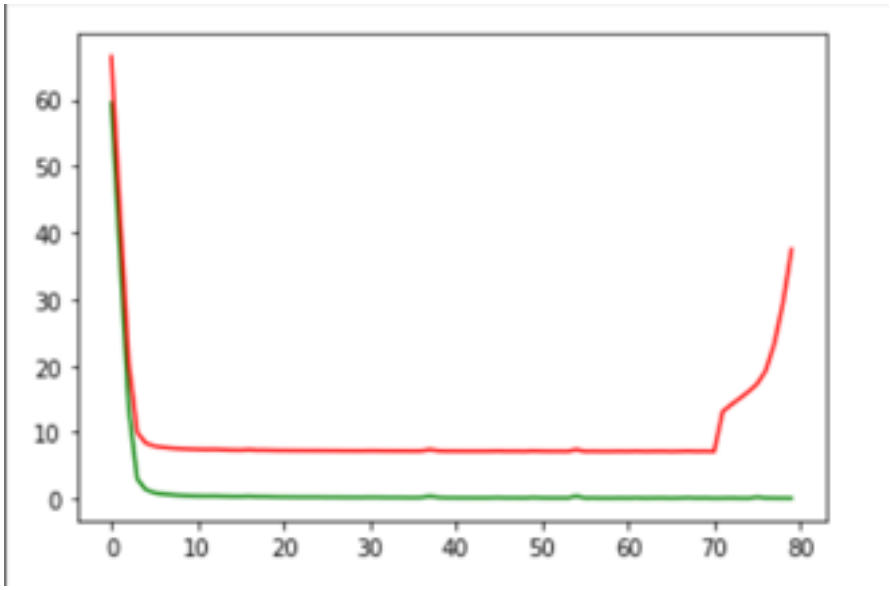

Figure 2: Red denotes testing loss; Green denotes training loss ; Cost during training is captured after every 200 iterations, Total iteration = 160K, around 14K iteration testing loss increases while training loss remains the same making this the point at which the model started overfitting.

```
    return A[l], caches
```

Dropout in the neural network occurs only during training the data, not for testing, since we don't want our predictions to be randomized.

9.6 Accuracy of V partial

Model V partial converged at 0.27 loss(for training) and 0.65 loss(for testing) after 14k iterations. The ***early stopping approach*** is followed in order to not overfit the data further which can be seen from Figure 2. The accuracy of V partial can be judged by the loss on testing data(0.65) which seems reasonable. Below described V-partial Agent's accuracy defines the accuracy of Model V-partial in more detail.
Is Vpartial more or less accurate than simply substituting V into equation (1)? By simply substituting V into equation (1) for the entire 250k samples, the loss seen(0.19) is less than that of Vpartial(0.55). Thus Vpartial is less accurate than simply substituting V into equation (1)

10 V-partial Agent

V-partial Agent runs in a partial prey information setting. The Agent knows the exact predator's location but isn't necessarily aware of where the prey is.

10.1 Pseudo code

```
def v_partial_agent(environment):
    if agent_loc == pred_loc:
        return "Predator caught the Agent"
    if agent_loc == true_prey_loc:
        return "Agent caught the prey"
    local_utility = {}

    for every agent_choice:
        ag_prey_distance = weighted-average-distance(agent_choice)
        agent_pred_distance = distance(agent, predator)
```

```
if agent_choice == pred_loc :
    # since the utility of the state is INFINITE
    local_utility[agent_choice, pred_loc] = INFINITE
  continue
else if agent_choice == true_prey_loc:
    # since the utility of the state is 0
    local_utility[agent_choice, pred_loc] = 0
else:
  pred_prob = compute_predator_probabilities(env,pred_loc,agent_choice)
  sum  = 0
  for every pred_choice:
    if agent_choice == pred_choice:
      sum = INFINITE
    else :
        # predicting v partial from NN model
        u_part = predict_model_v_partial(agent_choice, pred_choice, prey_prob,
            ag_prey_distance, agent_pred_distance)
        sum = sum+ pred_prob[pred_choice]*u_part
  local_utility[agent_choice, pred_loc] = sum

local_utility = {k: v for k, v in sorted(local_utility.items(), key=lambda item: item[1])}
# choose the node with the least utility
agent_loc = list(local_utility.keys())[0][0]
move_prey()
move_predator()
```

For 3000 simulations
Survivability: 99.9 percent
Average Step Count: 26.90
V Partial agent's survivability is better than U partial. But the average step count of V partial is slightly higher than average step count of U partial. Difference in step count = 2.64

11 Grey Box questions :

1. **How many distinct states (configurations of the predator, agent, prey) are possible in this environment?**
   With a total of 50 nodes in the graph and agent, prey, and predator having the freedom to occupy any of the nodes in the graph, the total number of nodes possible would be 50*50*50 = 125000 (50 choices for each agent, prey, and predator)

2. **What states s are easy to determine U^* for?**
   Good Terminal and Bad terminal states are easy to determine U* for, they are initialized in the start only and In case of Intermediate states, any state s whose one of the action spaces leading to good terminal states is easy to determine U^* for.

3. **Are there any starting states for which the agent will not be able to capture the prey? What causes this failure?**
   If the Agent and the Predator start in adjacent cells (refer to figure 3), and any choice that the agent can take is a neighbor to the predator, this scenario would result in the predator catching the agent irrespective of the agent's choice. And the Agent will not be able to catch the prey.

4. **Find the state with the largest possible finite value of U^*, and give a visualization of it.**
   The state with the largest possible finite value of U^* is 19.538 for state s (23, 37, 37) where 23 is the agent location and 37 is the prey and predator location. This seems reasonable since the agent plans towards prey, trying to escape from the predator, but since both of them are in the same location, it consumes more rounds. (Refer to figure 4)

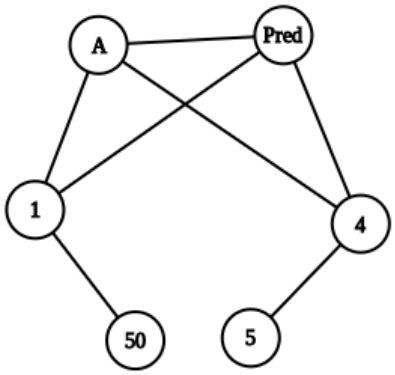

Figure 3: Edge case.

5. **Simulate an agent's performance 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?**
   U* Agent has a lower average step count as compared to Agent 1 and Agent 2. Thus U* Agent takes fewer rounds to catch the prey. Also, the U* Agent captures the prey more times than Agent 1 and Agent 2. The performances of these 3 agents are as follows - U* Agent > Agent 2 > Agent 1.
   Performance Agent 1: 84 percent, Agent 2: 99.1 percent, U star agent: 99.99 percent **Avg Step Count** Agent 1: 11.56, Agent 2: 10.34, U star agent: 8.649

6. **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.**
   In the environment, when simulated, Agent 1, 2, and U^*, and of the common states encountered, In some cases U^* Agent makes different choices than Agent 1 and Agent 2.
   Agent 1 makes choice based on simple if-else conditions based on the distances from prey and predator whereas U^* makes choice considering the utility of future states(Future states would be based on future prey and predator movements). Thus in the scenario shown in figure 5, Agent 1 moves from node 19 to 23 but U^* moves from node 19 to node 20 and it is clear since prey is in 21.
   Agent 2 makes choices considering only the future prey location and tries to maintain distance from the predator's current location whereas U^* Agent along with prey movements also considers the future predator movements. In the scenario shown in Figure 6, Agent 2 moves from node 1 to node 4, whereas U^* Agent moves from node 1 to node 2

7. **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?**
   U-partial agent performs better than Agent 3 and Agent 4 from Project 2. U-partial agent takes an average step count 23.9 whereas for Agent 3 and Agent 4 takes 27.3 and 29.06 steps on an average. The U-partial agent catches the prey 96 percent of times which is more as compared to Agent 3 and Agent 4, which catches the prey 81.5 percent and 91.5 percent of time, respectively. This U-partial agent isn't optimal as U-partial is only approximating the future utility, not calculating it exactly. So as a result, it cannot guarantee that the action chosen by the u-partial Agent is truly the best one.

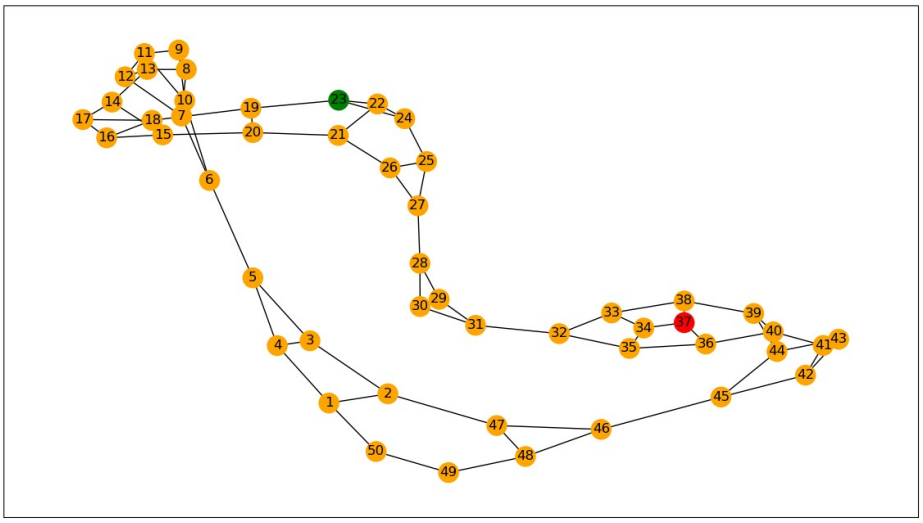

Figure 4: Maximum Utility state depiction. Red = Prey,Predator ; Green = Agent

8. **Simulate an agent based on $V_{partial}$. How does it stack against the $U_{partial}$ agent?** Survivability of $V_{partial}$ is 99.9, where as survivability of $U_{partial}$ is 96.23, but the average step count of $U_{partial}$ (23.9) is better than that of $V_{partial}$ (26.8).

12 Bonus 1 Agent

Build a model to predict the actual utility of the $U_{partial}$ agent. Where are you getting your data, and what kind of model are you using? How accurate is your model, and how do its predictions compare to the estimated $U_{partial}$ values?

12.1 Design

To predict the actual utility of the $U_{partial}$ agent, the utilities of the $U_{partial}$ agent needs to be collected. Data is collected by simulating the $U_{partial}$ agent multiple times using the following pseudo code:

```
def u_partial_agent(env):
    .... usual code described above
    utility[agent-loc, prey-probs, pred-loc] = utility_value
    capture_this_utility(utility_value)
    # described in the u_partial psuedo code
    # at what place data is being captured

def data_collection():
    for agent in (1, 50):
        for predator in (1, 50):
            env.generate_prey_pred_agent_locations()
            env.agent_loc = agent
            env.predator_loc = predator
            u_partial_agent(env)
```

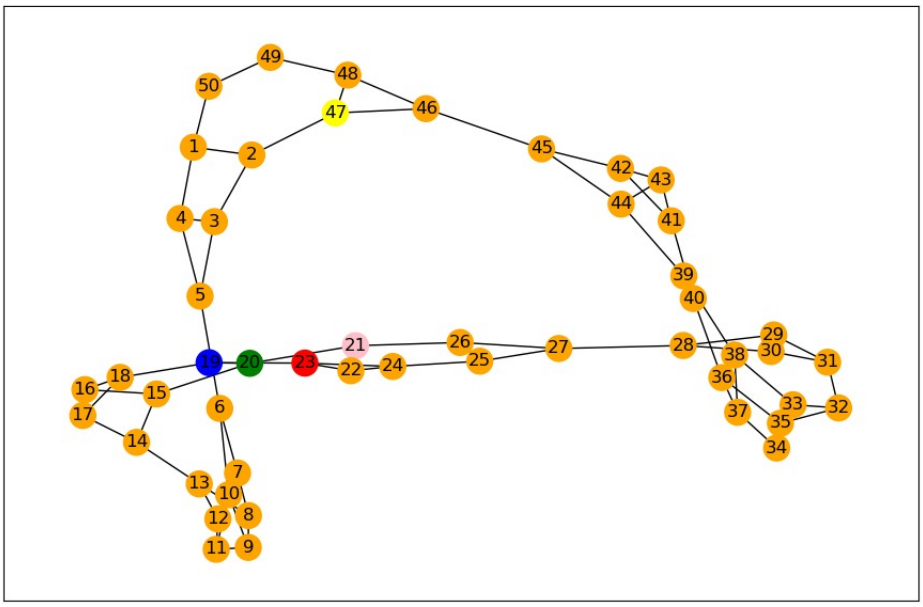

Figure 5: Agent - blue ; Prey - pink; Pred - yellow; U* agent choice - Green; Agent 1 choice - Red

At the end, where the final utility is calculated.

$U - partial - agent(\text{s}) = R_s + \min_{a \in S}(\Sigma_{next-s} P^a_{transition(s,next-s)}$ U-parial(next-s))

For a state s, this $U - partial(\text{s})$ is being captured as the Agent's utility. 125k samples have been collected, where 10k are used for testing and 115k for training data.

12.2 Model Space

Transfer learning is applied here, weights initialized are the same weights of Neural Network model of V-partial since the entity that needs to be predicted here is U-partial of agent, almost similar to that of U-partial of state scenario. The same dropout techniques have been followed in this scenario as well. Features are also maintained same to that of V-partial model which are Agent Location, Flattened Prey Probabilities, Predator Location, Weighted Agent-Prey distance and Agent-Pred distance.

12.3 Accuracy

Since this is Transfer Learning, Loss merged in around 250 iterations with training loss less than 0.1 and test loss 0.4. A Bonus Agent 1 has been simulated based on the predicted values to further understand this model.

```
def bonus_agent():
    if agent_loc == pred_loc:
        return "Predator caught the Agent"
    if agent_loc == true_prey_loc:
        return "Agent caught the prey"
    local_utility = {}
    for agent_choice in agent_choices:
```

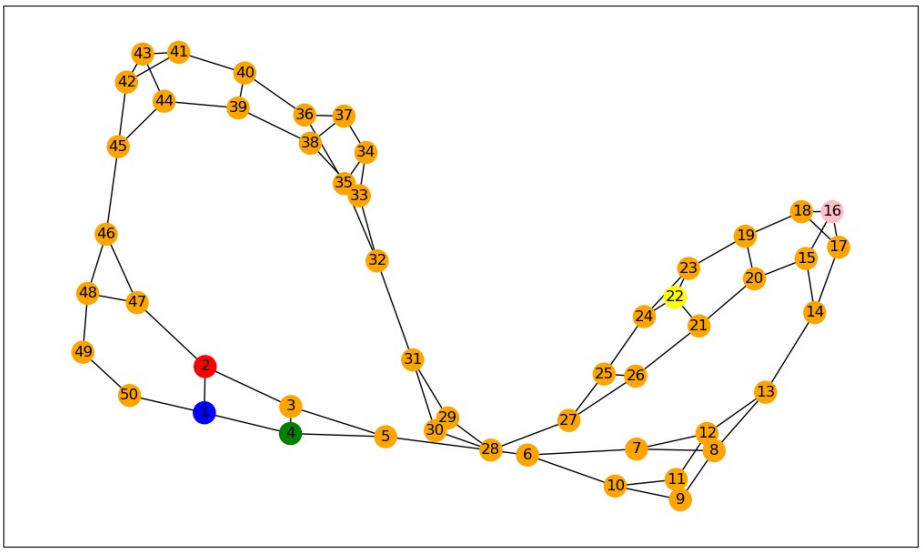

Figure 6: Agent - blue ; Prey - pink; Pred - yellow; U* agent choice - Green; Agent 2 choice - Red

```
        ag_prey_distance = calculate_ag_prey_distance(edges, agent_choice, prey_prob)
        agent_pred_distance = get_distance(agent_choice, pred_loc)
        if agent_choice == pred_loc :
          local_utility[agent_choice, pred_loc] = inf
          continue
        elif (agent_choice == true_prey_loc):
            local_utility[agent_choice, pred_loc] = 0
        else:
            local_utility[agent_choice, pred_loc] = predict_model_v_bonus_partial(agent_choi
    // sort local_utility ascending
    agent_loc = list(local_utility.keys())[0][0]
    move_prey()
    update_prey_probs()
    move_predator()
```

For 3000 simulations
Survivability: 85.3 percent
Average Step Count: 28.57

13 Bonus 2 Agent

Build an optimal partial information agent.

13.1 Design

Bonus Agent runs in partial prey information settings. This agent is an improvement over the u-partial agent. This Agent has the same information as u-partial Agent (knows the location of the predator but not necessarily of the prey). Like the u-partial agent, the bonus agent surveys

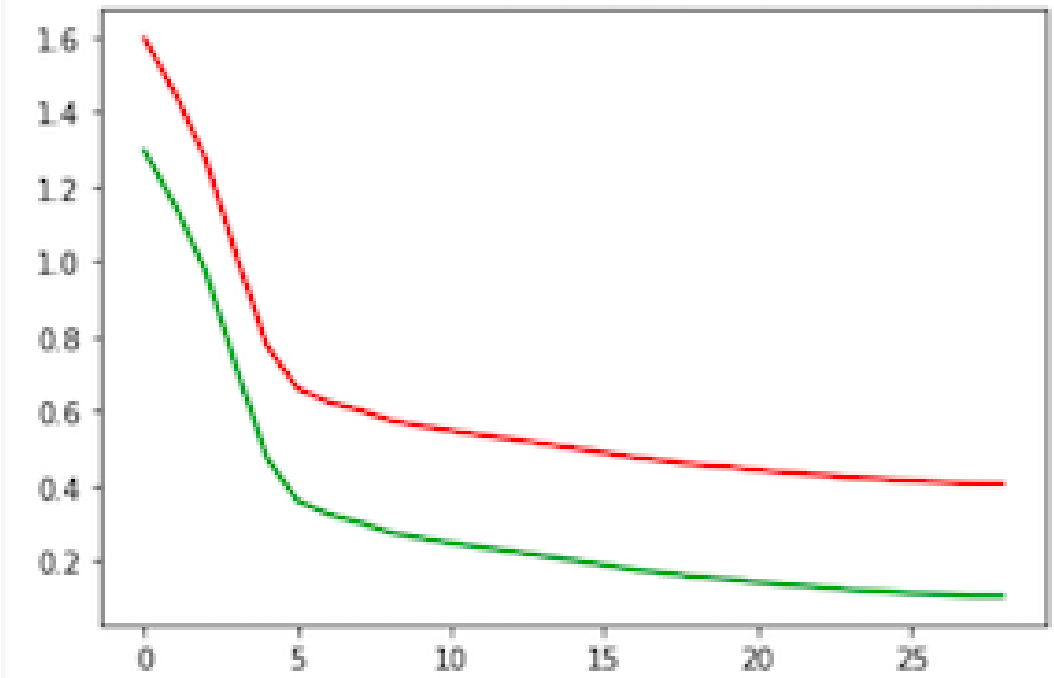

Figure 7: Red: testing loss, Green: training loss. Bonus-Model : Transfer learning from V partial, loss converged in 250 iterations. Loss is captured for every 10 iterations during the training

a node for finding prey, and tracks and updates the belief state. These belief states are updated whenever the agent surveys a node, the agent moves, and when prey moves.

The improvement of this agent over the u-partial agent is that it selects the node to move not only based on the current agent and predator location. The bonus agent calculates the distance from all its choices to all the choices of the predator, thus trying to move away not just from the predator's current location but from the future region of the predator's location. Along with this it also takes into consideration the node with the highest probability of containing the prey. And calculates the distance from all its choices to all the nodes the prey(with the highest probability) can move to in future. The agent chooses the node based on the largest weighted average predator distance, smallest weighted average prey distance, and u-partial value of the state.

```
utility of bonus agent = weighted average agent-prey distance *
    U-partial(s) / weighted average agent-predator distance
```

This suggests the bonus agent to move to a position which has least distance to the prey future and farthest distance from the predator's future.
Edge Case: The only scenario where this bonus agent is caught by the predator before it can catch the prey is that the initial location of the agent is adjacent to the predator and all the nodes the agent can move to are the adjacent node to the predator.

13.1.1 Pseudo code

```
def optimal_partial_agent(utility):
    for all agent_choice:
```

```
# node with the highest probability of containing the prey
high_prey_loc = get_highest_prob(prey_prob)

# agent prey weighted distance initial
ag_pr_dist = 0

for prey_nodes in all nodes prey can move from high_prey_loc :
    if probability of prey_nodes == 0 :
        # the reason being the future probability might not be zero
        ag_pr_dist += get_distance(agent_choice,prey_nodes, edges)
    else :
        ag_pr_dist += probability[prey_nodes]*get_distance(agent_choice,
        prey_nodes, edges)
    ag_pr_dist = ag_pr_dist / No. of nodes, prey can move to

if agent_choice == pred_loc :
    #since the utility of the state is INFINITE
    local_utility[agent_choice, pred_loc] = INFINITE
    continue
else if agent_choice == true_prey_loc:
    # Since the utility of the state is 0.
    local_utility[agent_choice, pred_loc] = 0
else:
  pred_prob = compute_predator_probs(env,pred_loc,agent_choice)
  sum = 0
  for all pred_choice:
    # calculating distance from the predator's choice to the agent's choice
    ag_pred_choice_dist = get_distance(agent_choice, pred_choice, env.edges)
    if agent_choice == pred_choice:
        #The utility is INFINITE
        sum = INFINITE
    else :
      u_part = calculate_u_partial_state(utility, agent_choice, pred_choice, prey_prob)
      sum += pred_prob[pred_choice]*u_part
    if ag_pred_choice_dist > 0:
        sum = sum / ag_pred_choice_dist
    else :
        # as the distance from pred choice to agent choice is 0.
        sum = INFINITE

  local_utility[agent_choice, pred_loc] = sum*ag_pr_dist

# sorting the list of local utilities.
local_utility = {k: v for k, v in sorted(local_utility.items(), key=lambda item:
item[1])}

#  agent location is the node with the least local utility.
agent_loc = list(local_utility.keys())[0][0]
```

13.1.2 Performance

Comparison of the partial bonus agent with u-partial agent based on 3000 iterations:

Performance of U-partial Agent:

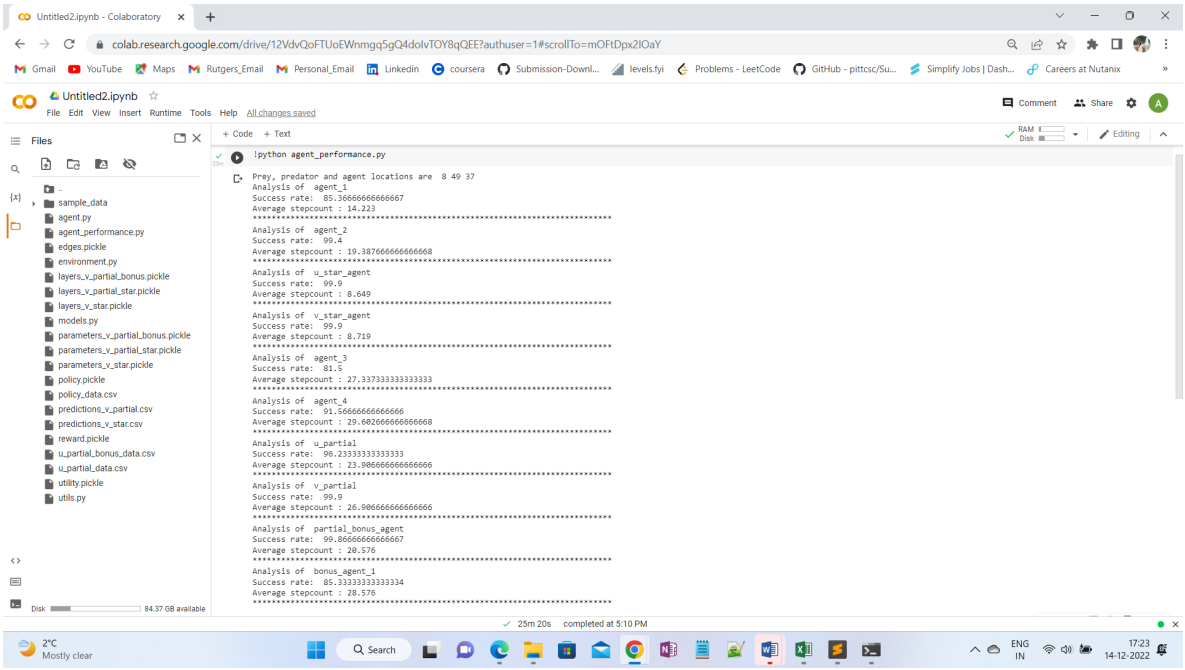

Figure 8: Results

Performance: 96.23 percent
Average Step Count: 23.90
Performance of Bonus Partial Agent:
Performance: 99.86 percent
Average Step Count: 20.576

14 Results

To conclude the survivability of agents looks like the following:

```
V* Agent = U* Agent > Agent 2 = Bonus Agent 2 = V-partial > U-partial Agent > Agent 4
> Agent 1 = Bonus Agent 1 > Agent 3.
```

The average step counts of agents follow the following trend :

```
U* Agent = V* Agent < Agent 1 < Agent 2 < Bonus Agent 2 < U-partial Agent <
V-partial Agent < Agent 3 < Bonus Agent 1 = Agent 4
```