520 : BETTER, SMARTER, FASTER

A Project Report by:
- Saransh Sharma (ss4368)

Environment Setup

The environment is similar to Project-2

The environment consists of a graph of 50 nodes[0-49], connected via edges in a circularly linked manner. We have added shortcut-edges for a random node ‘i’ and any node with degree 2 in the range [i-5, i+5] such that there are no nodes that with a degree > 3. This is repeated till no more edges can be added.

There are 3 players on the above mentioned graph environment – Predator, Agent, Prey – they can move along the edges, moving from any node to any of its neighbours in a single iteration. Goals for each of the players is as follows:

- Predator – Wants to catch the Agent
- Agent – Wants to catch the Prey
- Prey – Moves in a fixed manner without any motive

If the Predator or Agent fulfils their goal, the game ends – the Predator & Agent can’t exist in the same node, the Agent & Prey can’t exist in the same node.

The Predator and Prey can co-exist in the same node.

Their movements are in the following order –

1. Agent
2. Prey
3. Predator

Design Choices

I have taken an Object Oriented approach using Python (v3.7) for our implementation of the given task, this OOP approach helps us in easy refactoring of the code if required and also makes adding attributes, properties and functions easier down the line. The code is also highly functional and as such consists of a number of functions which improve ease of implementation and readability.

There are 5 major classes that have been implemented:

- **Graph** Class – Consists of the skeleton of the environment (the graph) which has been implemented using an Adjacency List (reduces time complexity while performing various functions) and consists of various helper functions like *get_deg(), get_next_moves(), BFS()* (the search algorithm that is used), *get_path(), get_valid_neighbours()* etc.

- **Prey** Class – Consists of the movement logic for the Prey (moves to one of its neighbours uniformly at random).

- **Predator** Class – Consists of the movement logic for the Predator (greedily trying to minimise its distance to the Agent - not always greedy, rather stochastically greedy – on average).

- **Agent** Classes – Consists of the logic for each of the agents' movements based on U (utility) and V (model)

- **NN** Class – Consists of the code for model 'V' which is taken to be a Neural Network with parameters like number of layers, learning rate, and number of iterations *(Explained in further detail ahead)*

Search Algorithm

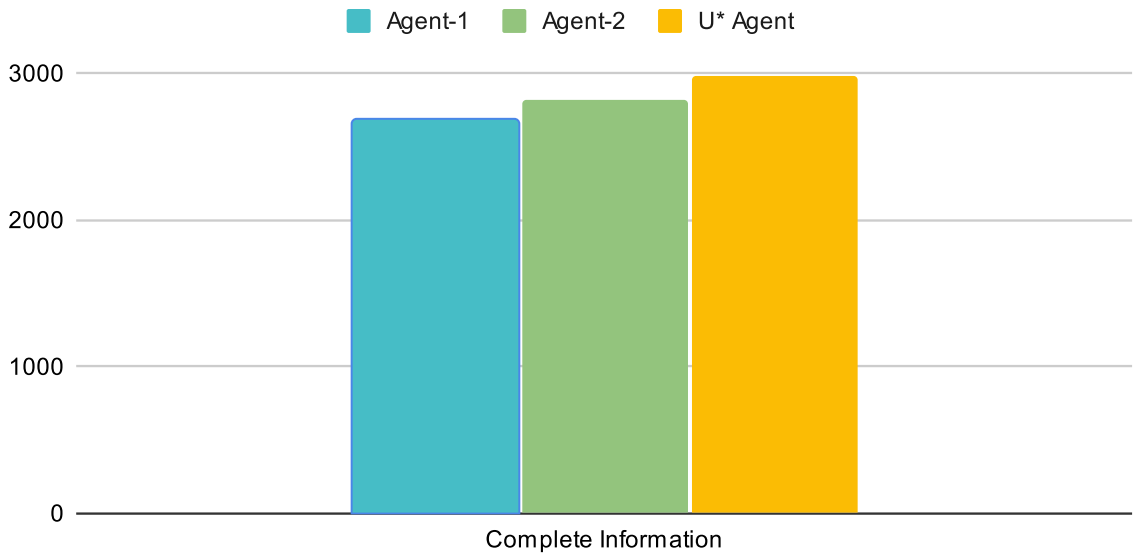
Breadth First Search

- Traverses nodes at equal depth from a parent node before moving forward in depth (level by level) and in this way, at some time, the fringe will contain all nodes that are equidistant from the root and popping these nodes, at some time the fringe will contain all nodes 1 level down.
- The data structure that we use for the *Fringe* in this case is a **Queue** as we want to explore the oldest nodes first and it's a **FIFO** (First In First Out) structure.
- It generally has a bigger memory footprint than DFS.
- If a path exists, and given BFS' property to explore all nodes on the same level first before going deeper, it guarantees that the path it finds is the **optimal/shortest** one.

Agent Comparisons

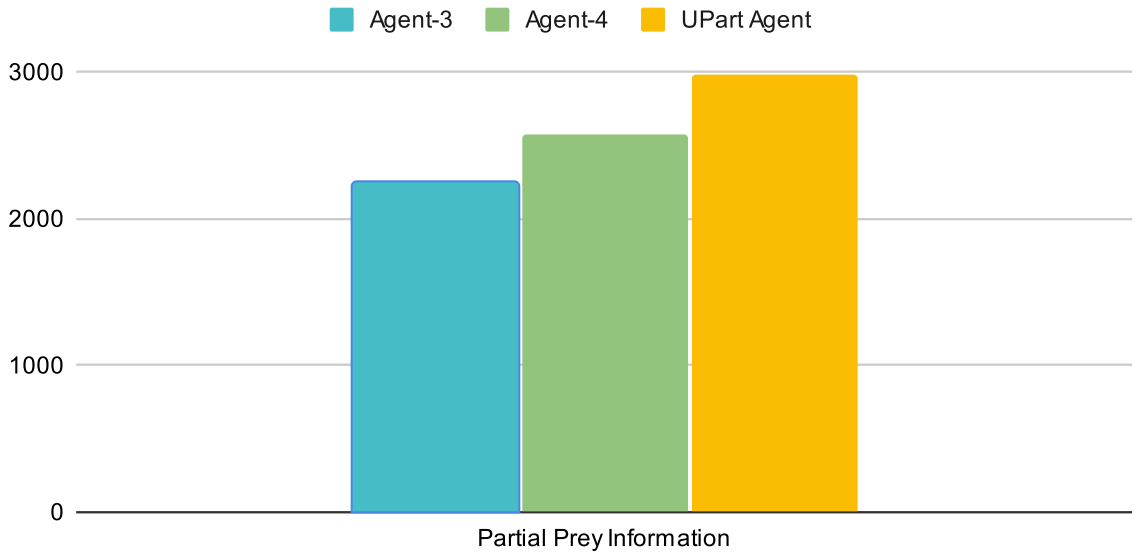
U* vs 1 vs 2

Agent 1: 16 Steps || Agent 2: 114 Steps || U* Agent: 9 Steps



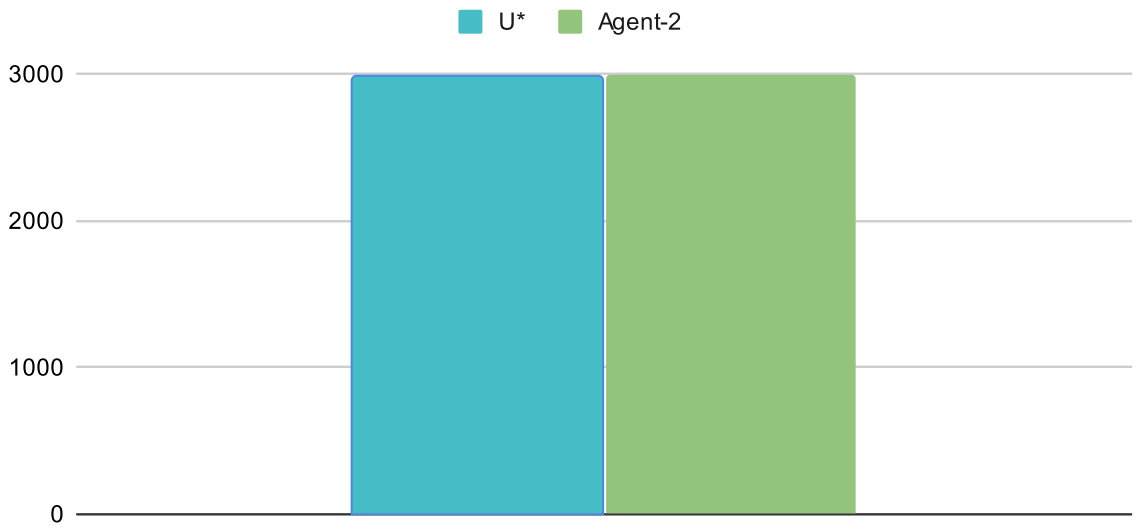
UPart vs 3 vs 4

Agent 3: 14 Steps || Agent 4: 18 Steps || UPart Agent: 25 Steps



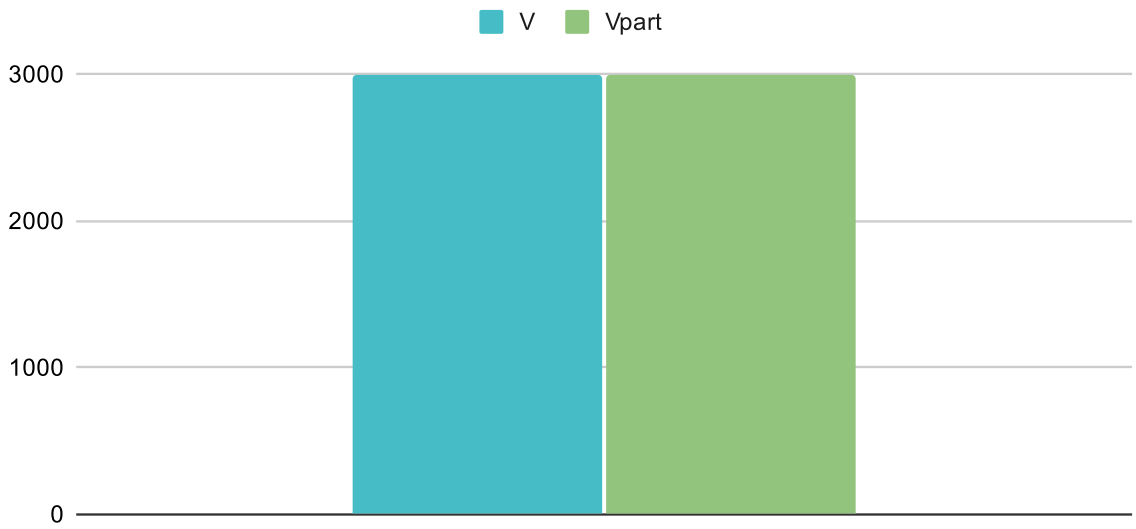
U* vs V

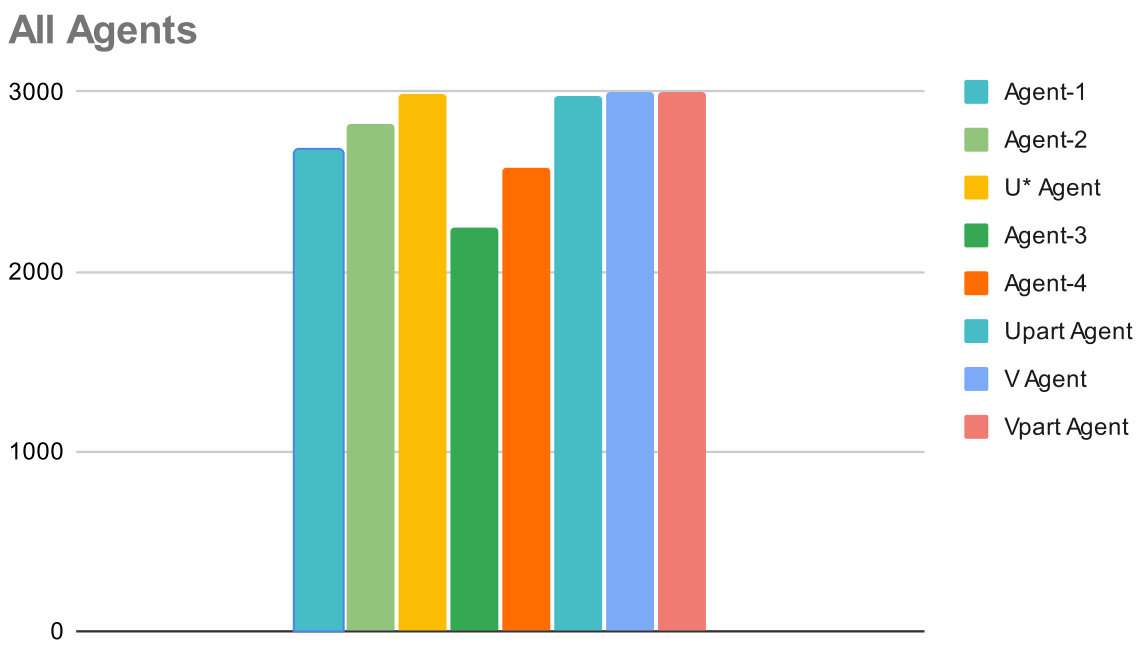
U* Agent: 9 Steps || V Agent: 12



V vs Vpartial

V Agent: 9 Steps || Vpartial Agent: 34




All Agents
3000
2000
1000
0
Agent-1
Agent-2
U* Agent
Agent-3
Agent-4
Upart Agent
V Agent
Vpart Agent

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

→ We can have 1,25,000 (50*50*50) distinct states for this environment

For a given state s, let U ∗(s) be the minimal expected number of rounds to catch the prey for an optimal agent

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

→ States where the agent and prey are in the same node, agent and predator are in the same node, agent is in close proximity to the predator (BFS Distance = 1) are states where U* is easy to determine for

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

→ We solve the Bellman System of Equations using Value Iteration:

$$U^*_{k+1}(x) = \min_{a \in A(x)} [\, R^a_x \, + \, \beta \, \textstyle\sum_{x'} P_{Prey} P_{Pred} U^*_k(x') \,]$$

Where $A(x)$ is the action space, R^a_x is the reward for taking action a, $U^*(x)$ is the optimal utility for the current state, $U^*(x')$ is the optimal policy of the state we are transitioning to. P_{Prey} is the transition probability of the prey and P_{Pred} is the transition probability of the predator.

We minimise in the above equation as we want the minimum utility for each state possible over the action space.

Q.) Write a program to determine U* for every state s, and for each state what action the algorithm should take. Describe your algorithm in detail.

I implemented Value Iteration which runs till error converges to a value less than 0.0001.

The action space is defined as the possible moves of the prey and predator for a particular agent move (and therefore the agent's transition probability is not included in the above equation).

I have initialised the value of Utilities and Rewards(Cost) for my state space as follows:

- Reward = 1 for each non-terminal state
  - Reward = 0 if agent = prey location
  - Reward = INF if agent = predator location

- Utility = Shortest distance b/w Agent & Prey
  - Utility = 0 if agent = prey location
  - Utility = INF if agent = predator location
  - Utility = INF if BFS(agent, pred) == 1

- This initialisation has been done so that Value Iteration converges faster.

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

→

```
Max Util: 21.144623175096328 ; State (39, 47, 44)
```

In this case, the agent was closer to the prey than the predator and the environment is such that the minimum number of steps that were required to catch the prey was higher and the predator would catch the agent instantly since there was a shortcut if the agent moved towards the prey. So, the agent basically circumvented the prey for some time and took a longer route back to it which took more number of steps.

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

Agent-3 performs almost perfectly and much better than Agents 1-2

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

→ `STATE: (2, 32, 23) ; Agent1: 2 ; Agent2: 3 ; U*Agent: 1`

The U* agent goes to 1 since it estimates that the steps that it needs to catch the prey are higher than the steps required by the predator to catch it and decides that moving away from the Prey in the present (while foregoing a chance of rewards sooner) is the better decision. This is in contrast to what the numbered agents do in this situation.

Q.) Build a model to predict the value of U∗(s) from the state s. Call this model V

→ My model 'V' is a Neural Network with 4 layers. Input layer with 1 node, 2 hidden layers with 4 and 3 nodes respectively, and an output layer with 1 node. (This number of nodes for the hidden layers has been set after experimentation)

NEURAL NETWORK

The input I gave my neural network was the distance between agent-prey and distance between agent-predator. And the output is the utility.

There are 3 main functions within our NN Class: *weights(), fwd_prop(), bwd_prop()*

weights() This function initialises weights and biases with random values (These random values change as our model gets trained)

fwd_prop() This function is used for Forward Propagation through our neural network which is basically a series of mathematical operations that the NN performs with current values of weights and biases (initialised random weights and biases). The mathematical operations are as follows(how many times they're performed depends upon the number of layers):

- Weighted sum for a layer and updating that sum

```
z1 = np.dot(self.x, self.wb_dict["w1"]) + self.wb_dict["b1"]
self.wb_dict["z1"] = z1
```

- Passing the weighted sum from the previous step through an activation function (*tanh* in my case)

```
a1 = self.tanh(z1)
```

- Weighted sum between the result of the previous step and the weights of next layer (z2)
- Activation function on the result of previous step (a2)
- Weighted sum between the result of the previous step and the weights of next layer (z3)
- The result from the previous step is our *yhat* (expected value of y)

bwd_prop() This function is used for Backward Propagation through our neural network which is basically a series of mathematical operations from the output towards the input in our NN, which updates weights and biases till the loss goes below a certain threshold. If the loss goes down, that means we are on the right track.

The backward propagation after each forward propagation is essentially what is referred to as “Training” of our neural network.

Intuitively it can be seen as the ‘inverse’ operations in the reverse order from our forward propagation.

After each backward propagation, we update our weights and biases and store them to be used in the next forward propagation or if the network is trained, then for making predictions for which the model was built.

```
def bwd_prop(self, yhat):

    dl_yhat = 2 * (np.subtract(self.y, yhat))/len(self.y)
    dl_z3 = dl_yhat
    dl_a2 = np.dot(dl_z3, self.wb_dict["w3"].transpose())
    dl_w3 = np.dot(self.wb_dict["a2"].transpose(), dl_z3)
    dl_b3 = np.sum(dl_z3, axis = 0, keepdims = True)
    # dl_z2 = dl_a2  * self.drelu(self.wb_dict["z2"])####
    dl_z2 = dl_a2  * self.dtanh(self.wb_dict["z2"])
    dl_a1 = np.dot(dl_z2, self.wb_dict["w2"].transpose())
    dl_w2 = np.dot(self.wb_dict["a1"].transpose(), dl_z2)
    dl_b2 = np.sum(dl_z2, axis = 0, keepdims = True)
    # dl_z1 = dl_a1  * self.drelu(self.wb_dict["z1"])####
    dl_z1 = dl_a1  * self.dtanh(self.wb_dict["z1"])
    dl_w1 = np.dot(self.x.transpose(), dl_z1)
    dl_b1 = np.sum(dl_z1, axis = 0, keepdims = True)

    #### WB_DICT IS UPDATED
    self.wb_dict["w1"] = self.wb_dict["w1"] + self.learning_rate * dl_w1
    self.wb_dict["w2"] = self.wb_dict["w2"] + self.learning_rate * dl_w2
    self.wb_dict["w3"] = self.wb_dict["w3"] + self.learning_rate * dl_w3
    self.wb_dict["b1"] = self.wb_dict["b1"] + self.learning_rate * dl_b1
    self.wb_dict["b2"] = self.wb_dict["b2"] + self.learning_rate * dl_b2
    self.wb_dict["b3"] = self.wb_dict["b3"] + self.learning_rate * dl_b3
```

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

I represent the states ‘s’ as the distance between agent-prey and agent-pred which implicitly also gives the network some information about the states. I use a matrix to feed this data to the neural network.

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

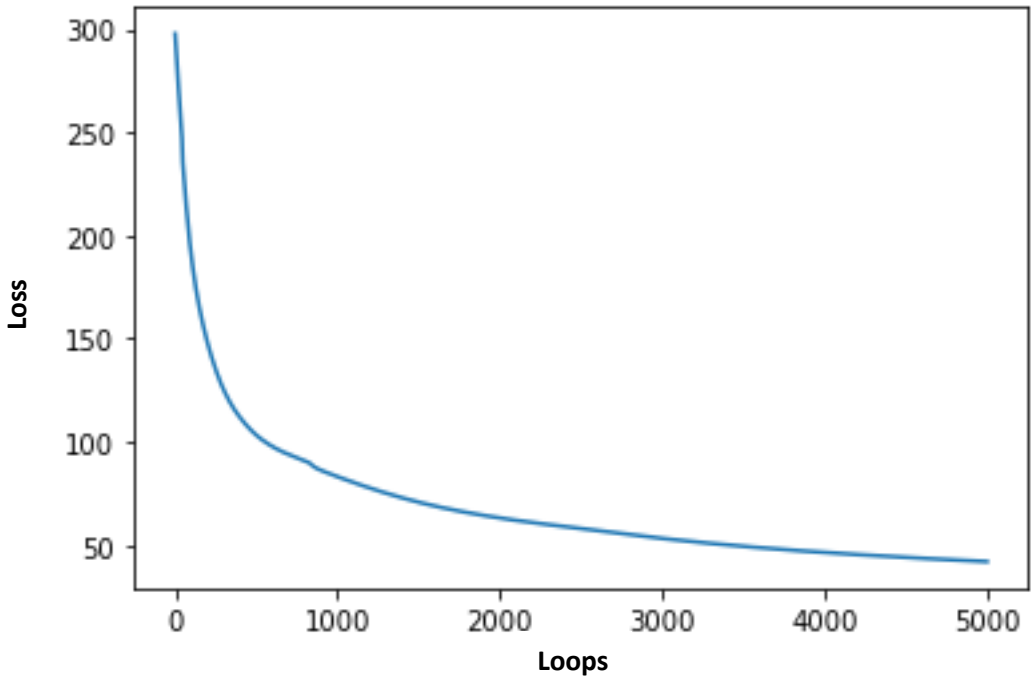
The Model V is a Neural Network. I use Gradient Descent as the training algorithm. The weights get trained/adjusted via multiple iterations of gradient descent.

Q.) Is overfitting an issue here?

→ Overfitting isn’t an issue in this case since our data set is finite and we don’t predict for new/unknown cases. I would go so far as to say that overfitting is actually a good thing in this case.

Q.) How accurate is V ?

→ My model produced a Mean Squared Error of 49.9 which means that my predictions were off by an average of 0.02 for each data point. (e.g. If model V predicted the Utility of a state to be 10, the actual utility, on average was between 9.98 and 10.02.



Once you have a model V , simulate an agent based on the values V instead of the values U∗

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

V has 100% success rate while U* was successful for 99.2% of the cases from a total of 3000 iterations.

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

→ Upartial Agent is much better than Agents 3-4 even with limited information, it performs

Build a model Vpartial to predict the value Upartial for these partial information states. Use as the training data states (including belief states) that actually occur during simulations of the Upartial agent.

Q.) How do you represent the states S(agent), S(predator), P as input for your model? What kind of features might be relevant?

→ I have provided the Vpartial model with 2 inputs – Expected Agent-Prey distance and Agent-Predator Distance.

The given input implicitly provides information about the state.

The expected distance to the prey was calculated using the Upartial agent.

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

→ Vpartial is a Neural Network. Gradient Descent has been used as the training algorithm for the NN.

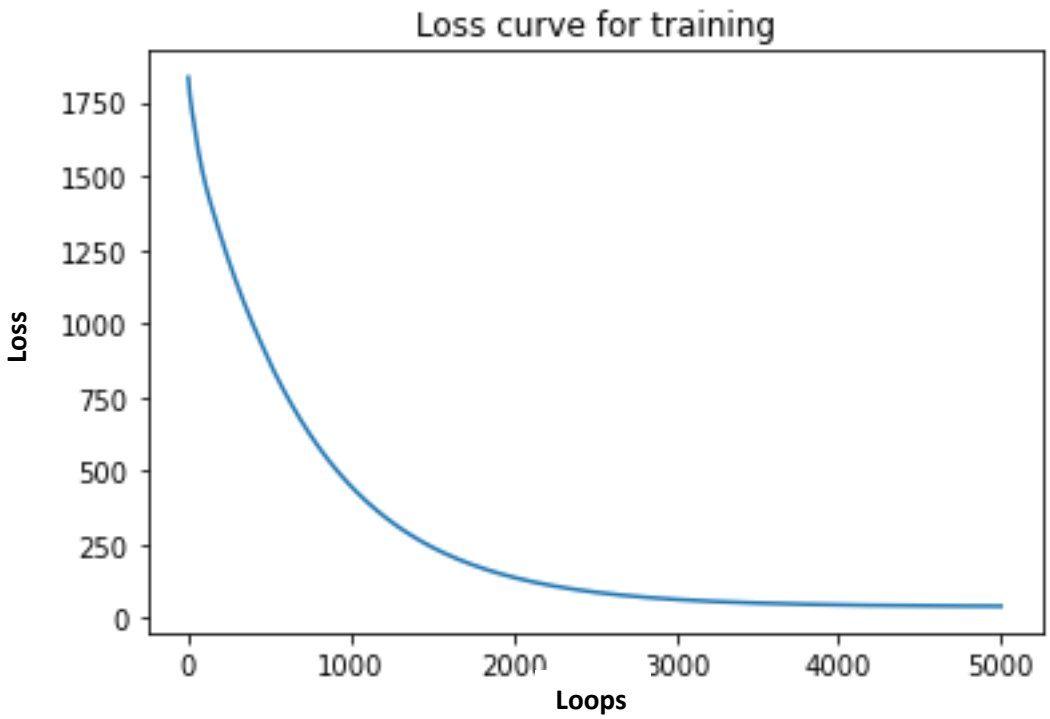
Q.) Is overfitting an issue here? What can you do about it?

Yes, Overfitting is an issue here since we don't know the location of the prey and its movement is probabilistic so our state space is essentially infinite.

This problem of overfitting can be solved by training on larger data.

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

→ Vpartial gave a Mean Squared Loss of 39.85 which means on average the predictions it made were off by 0.013 for each data point.



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

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

→ Vpartial is a marginally more accurate than Upartial

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

Vpartial stacked up evenly against the Upartial agent. This could be due to the fact that we were operating in a partial information setting and we ran our tests for relatively smaller iterations.

References

[1] https://heartbeat.comet.ml/building-a-neural-network-from-scratch-using-python-part-1-6d399df8d432