RUTGERS UNIVERSITY

COMPUTER SCIENCE

520: PROJECT 3

BETTER, SMARTER, FASTER

Authors:
Sahithya Namani **(sn807)**
Shobhit Singh (**ss4363**)

Supervisor:
Prof. Wes Cowan

December 12, 2022

INTRODUCTION:

For Project 3 we are working on the same environment as Project 2. The entities are the predator, the prey, and the agent. The prey transitions in the same way by choosing a random neighbor each time it moves. While the predator transitions according to the Distracted Predator Model. The aim of this project is to build an agent that captures the prey as efficiently as possible by transitioning to states with better Utility.

1. ENVIRONMENT

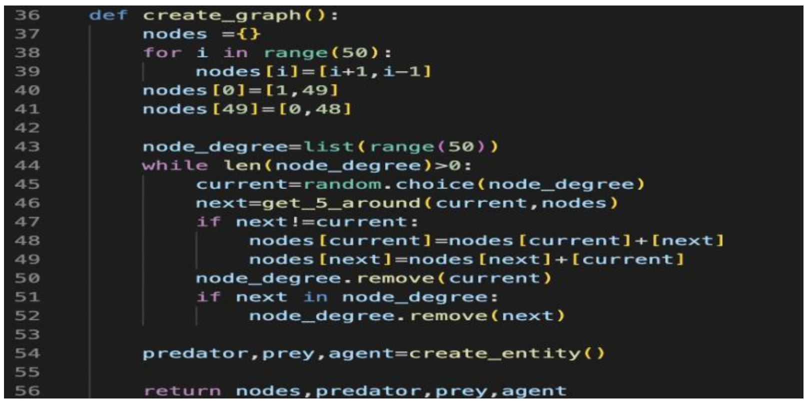
The given environment is a graph of nodes that are connected by edges. The graph consists of 50 nodes which are numbered from 1 to 50 and are connected in a circle, i.e., node-1 is adjacent to node-2, node-2 is adjacent to node-3, and similarly all the other nodes, and at the end, node-1 is connected to node-50 thus forming a closed loop. The agent, the prey, and the predator can move between nodes along the edges. To increase the connectivity of the circle, additional edges are added at random across the circle.

The following steps are followed when making an edge between two non-adjacent nodes:

1. Select a random node whose degree <3.
2. Add an edge between the selected node and a node within five steps forward or backward in the same loop.
   So, essentially node-1 can be connected with any node from either node-2 to node-6(within five steps going forward) or node-50 to node-46(within five steps going backward).
3. Repeat steps 1 and 2 until no more edges can be added.

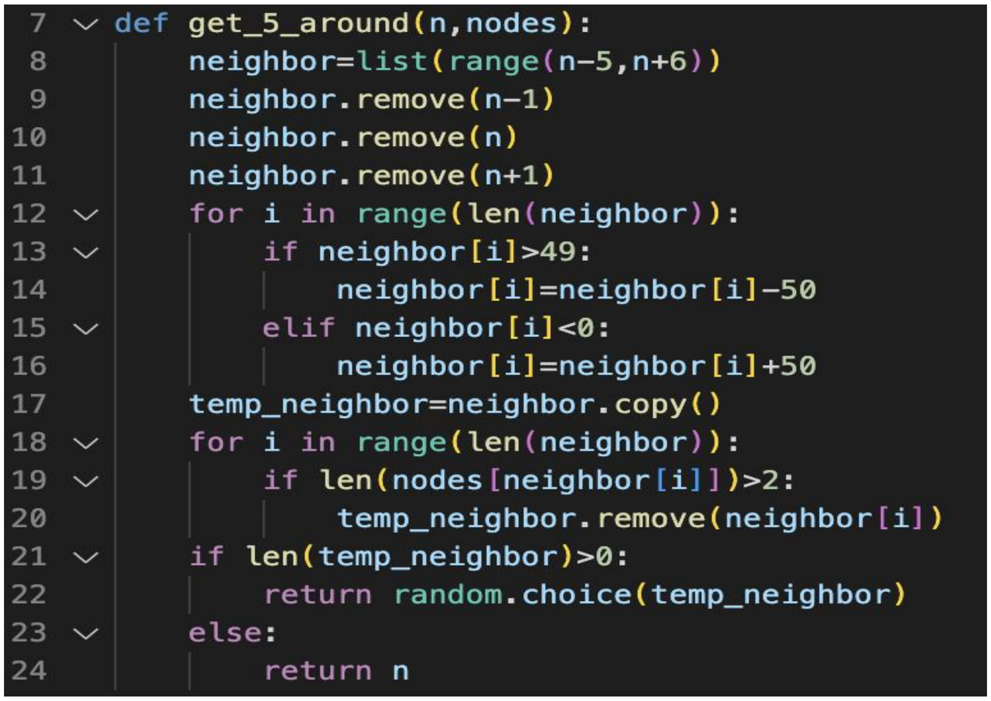
- ENVIRONMENT GENERATION:

To create an environment, we simply represent the nodes and edges as a dictionary. We create a python dictionary with keys representing the node index from 0-49 and their value represents a list of node indexes they are connected to. (Note: Nodes are represented from 0-49 and not 1-50 to make indexing easier). The following code creates the graph:



```python
def create_graph():
    nodes ={}
    for i in range(50):
        nodes[i]=[i+1,i-1]
    nodes[0]=[1,49]
    nodes[49]=[0,48]

    node_degree=list(range(50))
    while len(node_degree)>0:
        current=random.choice(node_degree)
        next=get_5_around(current,nodes)
        if next!=current:
            nodes[current]=nodes[current]+[next]
            nodes[next]=nodes[next]+[current]
        node_degree.remove(current)
        if next in node_degree:
            node_degree.remove(next)

    predator,prey,agent=create_entity()

    return nodes,predator,prey,agent
```

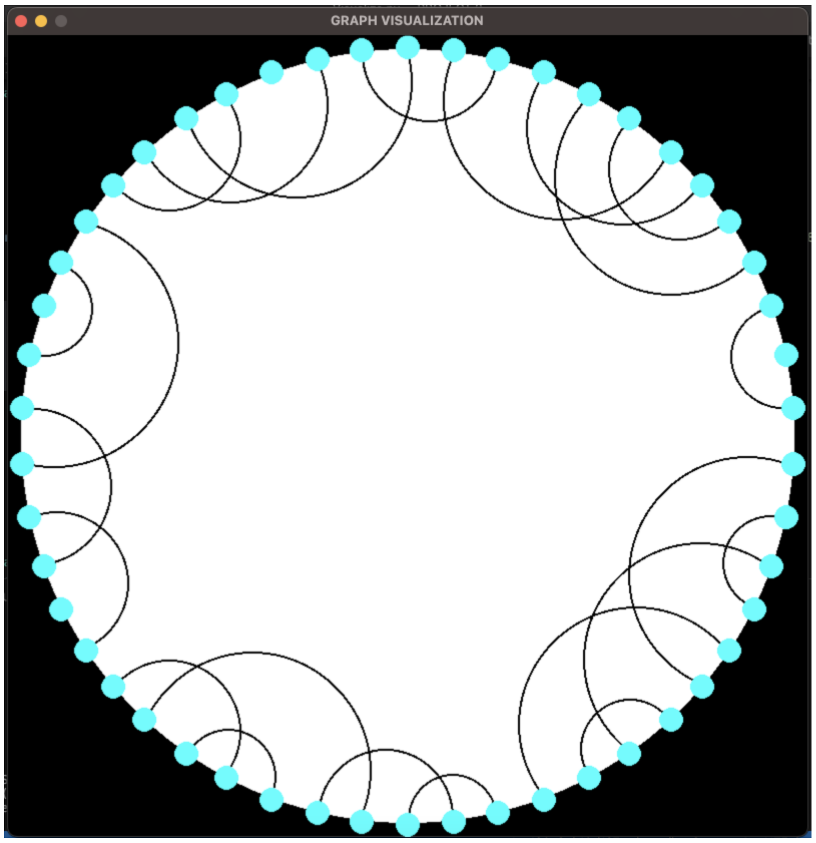
Lines 37 to 41 take care of basic node generation with connected neighbors. We then need to add the extra edge for which, we use the get_5_around function that returns a random out of all nodes at 5 steps away, does not already have degree 3, and does not already share an edge with the selected node. We then add an edge from the current to this node. If no node is available, the function returns the self-node which stays at degree 2. Node degree list in line 43 is used to keep track of the already considered node and acts as an exit condition. The get_5_around function can be found below:



```python
def get_5_around(n,nodes):
    neighbor=list(range(n-5,n+6))
    neighbor.remove(n-1)
    neighbor.remove(n)
    neighbor.remove(n+1)
    for i in range(len(neighbor)):
        if neighbor[i]>49:
            neighbor[i]=neighbor[i]-50
        elif neighbor[i]<0:
            neighbor[i]=neighbor[i]+50
    temp_neighbor=neighbor.copy()
    for i in range(len(neighbor)):
        if len(nodes[neighbor[i]])>2:
            temp_neighbor.remove(neighbor[i])
    if len(temp_neighbor)>0:
        return random.choice(temp_neighbor)
    else:
        return n
```

Note: *create_entity function is described in the next section*

For the purpose of this project, we will only create the graph once and use the same graph throughout. We created a graph which came out as follows.



Note that node numbering starts from the top center (node 0) and goes clockwise.

2. THE PREDATOR, THE PREY, AND YOU

There are three entities occupying the environment namely, the predator, the prey, and the agent who can move between nodes taking one step at a time along the edges. The agent wants to catch prey and the predator wants to catch the agent to win the game. The three players move in the following order: Agent->Prey->Predator (in circular form). A node can be occupied in the following ways:

1. The node can be empty i.e., the agent or the predator or the agent does not occupy the node.
2. The node can be occupied by the agent and the prey. And in this case the agent wins the scenario.
3. The node can be occupied by the agent and the predator. And in this case the predator wins the scenario.
4. The node can be occupied by both the predator and the prey. And in this case, nothing happens.

Rules for the entities:

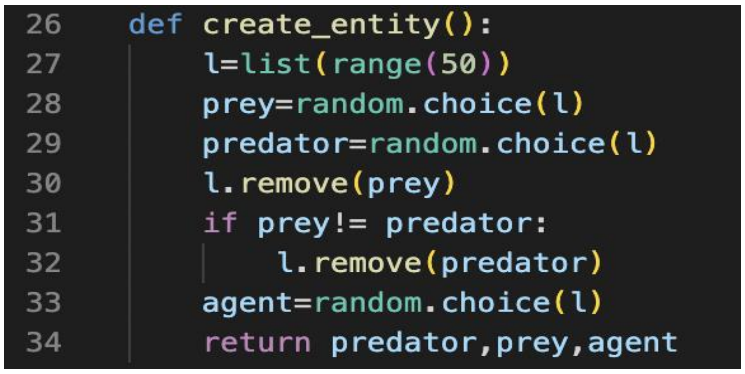
a. **The Prey:** The rules that prey take while taking a step are simple. It can choose to stay back at the same node or move to one of the adjacent connected nodes with equal probability (i.e, can choose to stay back or select one of the connected nodes with ¼ probability if the degree of that node=3). It makes a move regardless of the actions or locations of other entities until the game ends.

b. **The Predator:** The predator gets easily distracted. It tries to close the distance between itself and the agent with a probability of 0.6 and moves to one of its neighbors at random with a probability of 0.4. This implies that the predator is moving to close the distance, but this is not guaranteed in any round.

c. **The Agent:** The agent's moves depend upon the specific strategy it employs. In the case of partial information setting the agent may choose to survey the node at a distance to determine the entity present in that node. The agent is aware of how the predator and prey choose their next step, but unaware of the actual step taken.

- ENTITY CREATION:

The Prey and predator can spawn on or occupy the same node at any point of time. To disallow instant win or instant loss, we make sure that the agent does not spawn on the same node as prey or predator. This can be achieved by the following code:



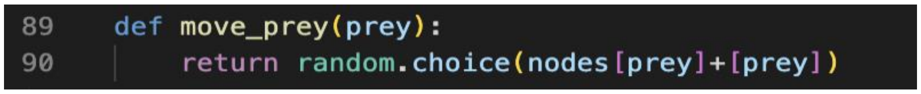
```python
def create_entity():
    l=list(range(50))
    prey=random.choice(l)
    predator=random.choice(l)
    l.remove(prey)
    if prey!= predator:
        l.remove(predator)
    agent=random.choice(l)
    return predator,prey,agent
```

We simply create a list of 50 nodes and make a random selection for prey and predator. We then remove these nodes from the selection group and do a random selection for the agent.

Note: *We need to check if the prey and predator both spawned on the same node. If that is the case, we are removing the same node twice which throws an error.*

- PREY MOVEMENT:

Prey moves to a neighbor or stays in place with equal, unbiased probability which can be achieved with the following function.

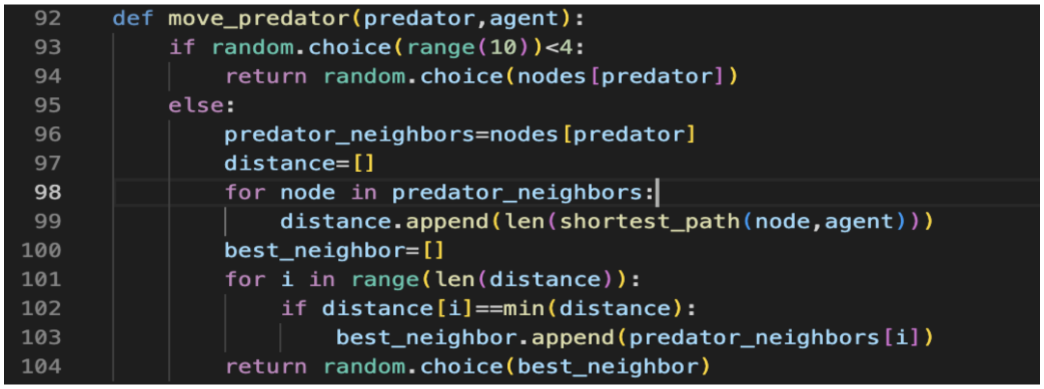


```python
def move_prey(prey):
    return random.choice(nodes[prey]+[prey])
```

We simply select a node from a pool of its neighbor nodes and itself.

- PREDATOR MOVEMENT:

Predators can have a distracted or focused movement. It has a 0.4 probability of being distracted and moving with equal probability to one of the neighbor nodes (considering it will always move and does not stay in place). It can have a focused movement with a probability of 0.6 in which case, checking which of its neighbor nodes shortens its distance from the predator. If there are multiple such nodes, it chooses one at random. This is achieved from the following function:

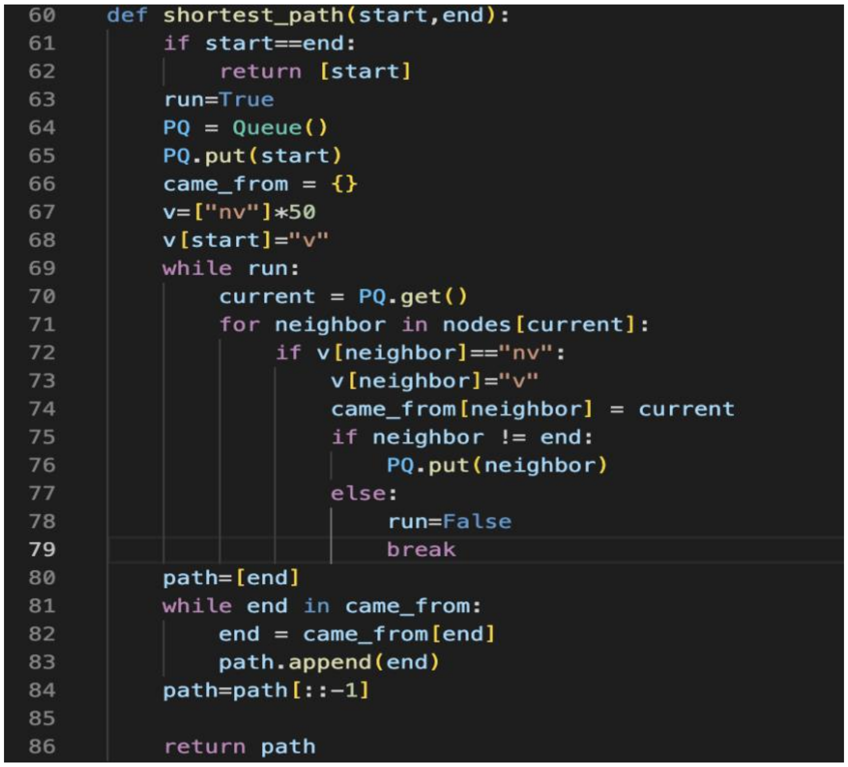


```
 92  def move_predator(predator,agent):
 93      if random.choice(range(10))<4:
 94          return random.choice(nodes[predator])
 95      else:
 96          predator_neighbors=nodes[predator]
 97          distance=[]
 98          for node in predator_neighbors:
 99              distance.append(len(shortest_path(node,agent)))
100          best_neighbor=[]
101          for i in range(len(distance)):
102              if distance[i]==min(distance):
103                  best_neighbor.append(predator_neighbors[i])
104          return random.choice(best_neighbor)
```

With a probability of 0.4, it returns a random node out of all its neighbors as in line 94. Or else, we create a list of lengths of all the distances from its neighbor node to the agent (lines 96-99). We then filter the best node (with the shortest distance) for the predator to move to in lines 100-103. Then return one of these best nodes at random.

Note: *the predator can move to a node with a shorter distance even if it is distracted.*

To calculate the shortest path from one node to another, we use the shortest_path function which uses the BFS algorithm to achieve it. We are not using DFS as it does not guarantee the shortest path. We do not have a reliable heuristic for A Star and Dijkstra without a weighted edge is BFS itself. The function can be found below.



```python
def shortest_path(start,end):
    if start==end:
        return [start]
    run=True
    PQ = Queue()
    PQ.put(start)
    came_from = {}
    v=["nv"]*50
    v[start]="v"
    while run:
        current = PQ.get()
        for neighbor in nodes[current]:
            if v[neighbor]=="nv":
                v[neighbor]="v"
                came_from[neighbor] = current
                if neighbor != end:
                    PQ.put(neighbor)
                else:
                    run=False
                    break
    path=[end]
    while end in came_from:
        end = came_from[end]
        path.append(end)
    path=path[::-1]

    return path
```

This is a simple BFS algorithm that breaks when it finds the target node to reduce time complexity.

DISTINCT STATE

There are 50 nodes, and the agent, prey, and predator can occupy any of them. Each state is defined by a combination of these entity locations. So, the total number of nodes is the number of unique combinations/arrangements of these 3 entities. The agent can occupy the graph in 50 ways, The predator and prey can also occupy one of the 50 nodes.

So, Total no. of arrangements = 50*50*50 = **125000**

This is the total number of states possible for a 50-node graph.

States (s) for which calculating $U^*(s)$ is easy:

There are 3 cases where it is easy to calculate Utility for corresponding states.

1. **When the agent and predator are at the same node-** In this state, The agent is already dead and can no longer proceed to catch the prey. We can simply set this utility to infinity. (For the purpose of coding, we will use 1000000 instead of infinity as since we are minimizing the utility, taking a large number assures we won't select that state)

   Total number of such states = no. of ways both predator and agent can occupy a node (50) * Number of combinations for the prey to occupy for each such case (50)
   =50*50=2500

2. **When the agent and prey are at the same node -** In this state, The agent has already caught the prey. We can simply set this utility to 0 as no more steps are needed to catch the prey.

   Total number such states = no. of ways both prey and agent can occupy a node (50)
   * Number of combinations for the predator to occupy for each such case (50) –
   no. of cases when all 3 entities will be at the same node (50)
   =50*50-50=2450

3. **When the agent is 1 step away from the prey -** In this state, since the agent moves first, it will surely move towards the prey and catch it in 1 step. So, if the agent is 1 distance away, we can simply assign the Utility for such states as one.

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

S: State space
A: Action space
T: Transition probabilities
R: Rewards

Let us assume that the initial estimates $U^*_0(s)$ for each state-s $\in$ S,
In our case, the initial estimates are taken as:

$$U_0 = |\text{Agent\_position} - \text{Prey\_position}|$$

The estimates are improved iteratively using the following equation,

$$U^*_{k+1}(s) = \min_{a \in A(s)} [R + \beta \Sigma_{s'} P_{prey} P_{predator} U^*_k(s')] \quad \text{-------} \quad \text{Equation (A)}$$

Here, since we need to catch the prey in as few moves as possible, the Reward represents the cost we need to minimize and chose the action with minimum Utility. The reward can be taken as 1 as we know we will at least one more move to catch the prey if not already caught. But in our case,

$$R = |\text{Agent\_position} - \text{Prey\_position}|/2$$

This is because we need at least half the distance to catch the prey if the prey also starts moving toward us. Also, taking a greater reward than 1 gives us a nice span of output utilities which makes training Models efficient.

Transition probability in our case is the probability of the prey being in that state and the probability of the predator being in that state too.

So,

$$T = P_{prey} P_{predator}$$

We need to keep iterating equation A till the change from U^*_k (s) to U^*_{k+1}(s) converges to 0

$$\text{Max}_{s \in S} |U^*_{k+1}(s) - U^*_k(s)| \longrightarrow 0$$

The discounting factor ranges between 0 and 1(exclusive)
i.e., $0 < \beta < 1$

The values are calculated by using the immediate reward of that state and the future estimate, which can also have an error value. The future estimate gets shrunk by a factor of β at every step. Discounting is used to tackle artificially large rewards in the future.

Given a problem we are trying to achieve a particular goal and the game ends when that goal is achieved, the infinite time horizon doesn't make sense in this case. As we know what the end state of the game is in advance the problem becomes a finite time horizon problem. And we don't have to worry about discounting. Therefore, in our problem, we have taken the value of **β=1**. And given that $U^*(s)$ is the minimal expected number of rounds to catch the prey for an optimal agent. Therefore, we calculate the minimum distance between the agent and prey.

The Equation A represents how U^* of one state relate to U^* of other states. Once we have utility of every state, we can choose the action the agent can take, by checking total utility of all possible state (with respect to how probable that state is) that each action can produce. We then choose the action that minimizes the this expected value which becomes the policy of a state.

Policy Iteration Formula:

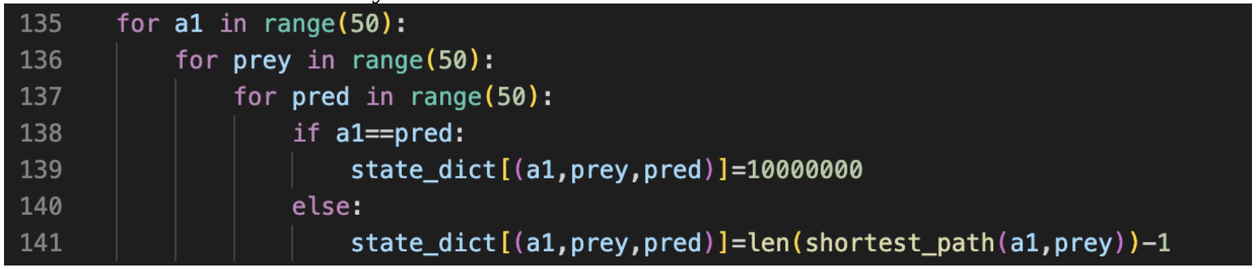
$$\pi^*(s) = argmax_{a \epsilon A(x)} \left(R + \beta \Sigma_{s'} \text{P}_{prey} \text{P}_{predator}\, U^*(s')\right)$$

THE COMPLETE INFORMATION SETTING:

In this environment, the agent has full information on the location of the prey and predator. For each agent, we call its specific function

- **Write a program to determine U^* for every 125000 states,**

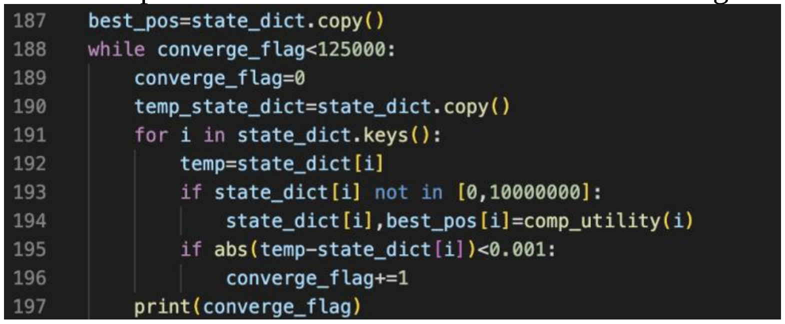
To calculate and store the utility of each state, we first initialize all 125000 states in a dictionary with a tuple of entity position as a key and its corresponding value as a guess utility. The following code initializes this dictionary.



```python
for a1 in range(50):
    for prey in range(50):
        for pred in range(50):
            if a1==pred:
                state_dict[(a1,prey,pred)]=10000000
            else:
                state_dict[(a1,prey,pred)]=len(shortest_path(a1,prey))-1
```

The 3 nested for loops generate all possible permutations of entity position creating all states. We then assign utility as infinity for states where the agent and predator are at the same node as if the agent dies, it will never catch the prey. Note that for the purpose of coding, we assigned a large

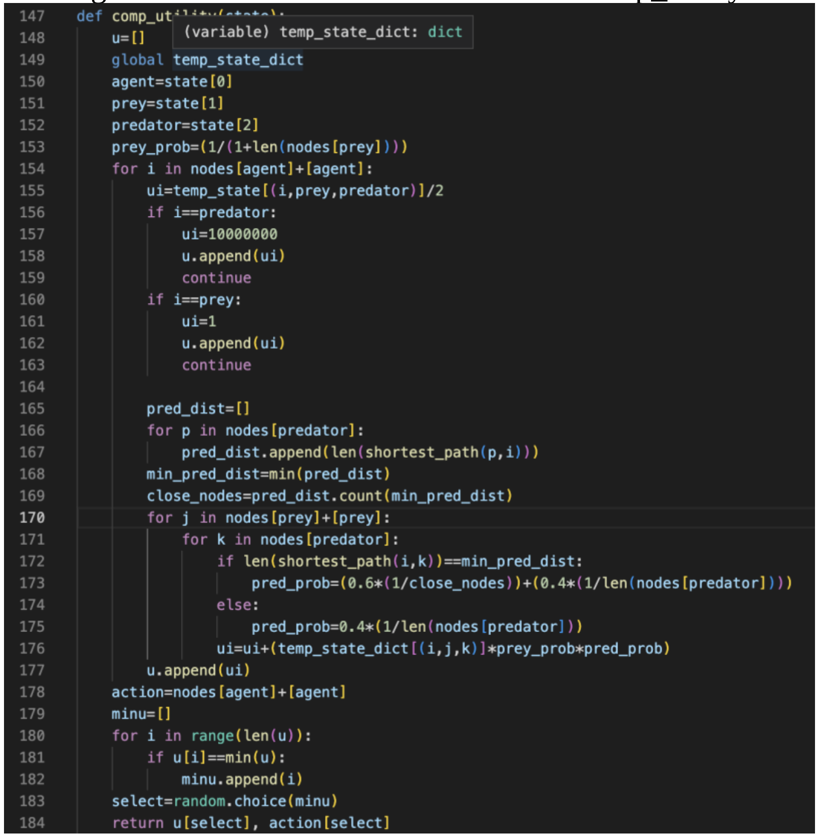
value (10000000) instead of infinity. Since we are minimizing the utility, this makes sure we never prioritize this state. Apart from this, for all other states, we initialize the shortest distance from agent to prey. This also ensures that Utility is 0 for states where agent and prey are on the same node.
Next to implement value iteration and check for convergence, we use the following code.



```python
best_pos=state_dict.copy()
while converge_flag<125000:
    converge_flag=0
    temp_state_dict=state_dict.copy()
    for i in state_dict.keys():
        temp=state_dict[i]
        if state_dict[i] not in [0,10000000]:
            state_dict[i],best_pos[i]=comp_utility(i)
        if abs(temp-state_dict[i])<0.001:
            converge_flag+=1
    print(converge_flag)
```

best_pos is another dictionary that stores the best action for the agent to take for any given state. We could compute this action after calculating the utilities but that would require some extra loops during implementation so to save complexity, we are storing the action as we are already running those loops when calculating the utility.

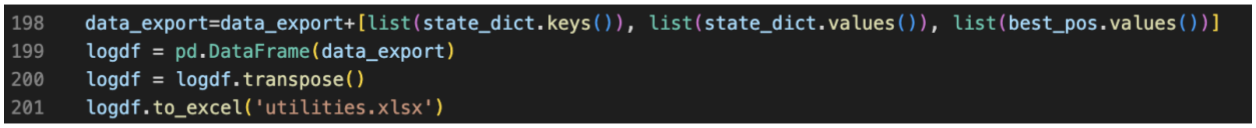
We need to make sure utility for all states converge so we run loop for Value Iteration till all utility changes by less than 0.001. We don't recalculate the Utilities of end cases of infinity and 0 as these are already accurate. The comp_utility function returns the new utility to be compared for convergence and the best action to take. The comp_utility function is described below



```python
def comp_ut
    u=[]
    global temp_state_dict
    agent=state[0]
    prey=state[1]
    predator=state[2]
    prey_prob=(1/(1+len(nodes[prey])))
    for i in nodes[agent]+[agent]:
        ui=temp_state[(i,prey,predator)]/2
        if i==predator:
            ui=10000000
            u.append(ui)
            continue
        if i==prey:
            ui=1
            u.append(ui)
            continue

        pred_dist=[]
        for p in nodes[predator]:
            pred_dist.append(len(shortest_path(p,i)))
        min_pred_dist=min(pred_dist)
        close_nodes=pred_dist.count(min_pred_dist)
        for j in nodes[prey]+[prey]:
            for k in nodes[predator]:
                if len(shortest_path(i,k))==min_pred_dist:
                    pred_prob=(0.6*(1/close_nodes))+(0.4*(1/len(nodes[predator])))
                else:
                    pred_prob=0.4*(1/len(nodes[predator]))
                ui=ui+(temp_state_dict[(i,j,k)]*prey_prob*pred_prob)
        u.append(ui)
    action=nodes[agent]+[agent]
    minu=[]
    for i in range(len(u)):
        if u[i]==min(u):
            minu.append(i)
    select=random.choice(minu)
    return u[select], action[select]
```

To find the utility of a state by value iteration, we need to find the utility of all actions and choose the one that produces the minimum expected Utility. We run a loop for actions the agent can take including staying at a place (line 154). If that action leads to death, we assign it a utility of infinity to make sure we avoid that action and if an action catches the prey, we make the utility 1 as it catches the prey in 1 action. Otherwise, we permutate through all states possible for that action (nested loop at lines 170-171) and compute the new Utility by Value iteration formula. We then find the minimum Utility of all possible actions and break the tie if any (lines 179-184)

Once we are done finding the Utility of all states, we store it for further use as we need it for agent implementation and calculating U_partial later. It takes about 40min to compute this, so we don't want to do it every time. The following code is used to store the results.



```
data_export=data_export+[list(state_dict.keys()), list(state_dict.values()), list(best_pos.values())]
logdf = pd.DataFrame(data_export)
logdf = logdf.transpose()
logdf.to_excel('utilities.xlsx')
```

We store the values to a file named "utilities.xlsx"

Find the state with largest finite U_star and visualize it.

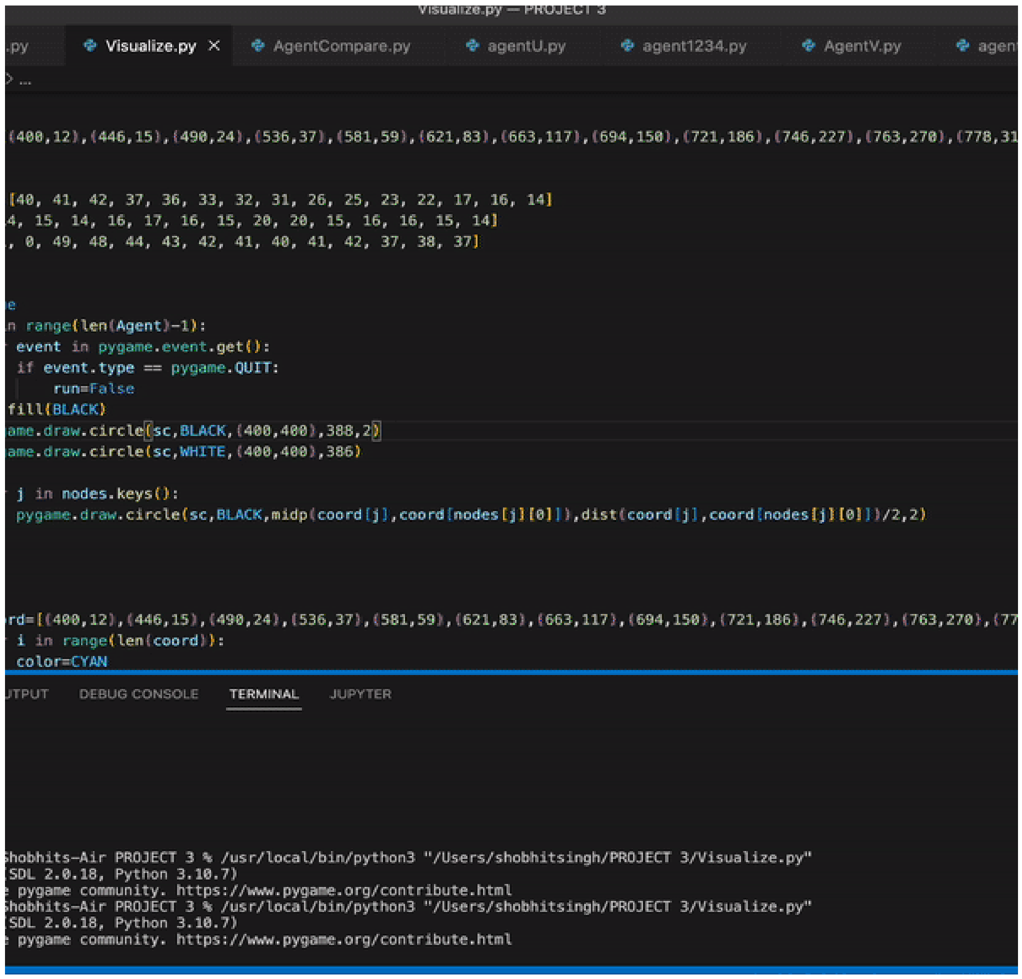
As per the data, when we sort the Utilities in descending order, the largest finite value came to be 16.85121 for the state (40,14,1) where, 40 is agent position, 14 is prey position and 1 is predator position.

Simulating the agentU(explained later) starting from this state, we get the following movements

Agent = [40, 41, 42, 37, 36, 33, 32, 31, 26, 25, 23, 22, 17, 16, 14]
Prey = [14, 15, 14, 16, 17, 16, 15, 20, 20, 15, 16, 16, 15, 14]
Predator = [1, 0, 49, 48, 44, 43, 42, 41, 40, 41, 42, 37, 38, 37]

We see that it takes 15 moves for the agent to catch the prey which is close to the utility we got.

Visualization of the above simulation (Note: use Visualize.py to generate visualization)



```
Visualize.py — PROJECT 3
.py   Visualize.py ×   AgentCompare.py   agentU.py   agent1234.py   AgentV.py   agen
> ...

(400,12),(446,15),(490,24),(536,37),(581,59),(621,83),(663,117),(694,150),(721,186),(746,227),(763,270),(778,31


[40, 41, 42, 37, 36, 33, 32, 31, 26, 25, 23, 22, 17, 16, 14]
4, 15, 14, 16, 17, 16, 15, 20, 20, 15, 16, 16, 15, 14]
, 0, 49, 48, 44, 43, 42, 41, 40, 41, 42, 37, 38, 37]


e
n range(len(Agent)-1):
 event in pygame.event.get():
  if event.type == pygame.QUIT:
      run=False
fill(BLACK)
ame.draw.circle(sc,BLACK,(400,400),388,2)
ame.draw.circle(sc,WHITE,(400,400),386)

 j in nodes.keys():
 pygame.draw.circle(sc,BLACK,midp(coord[j],coord[nodes[j][0]]),dist(coord[j],coord[nodes[j][0]])/2,2)




rd=[(400,12),(446,15),(490,24),(536,37),(581,59),(621,83),(663,117),(694,150),(721,186),(746,227),(763,270),(77
 i in range(len(coord)):
 color=CYAN
```

```
hobhits-Air PROJECT 3 % /usr/local/bin/python3 "/Users/shobhitsingh/PROJECT 3/Visualize.py"
(SDL 2.0.18, Python 3.10.7)
 pygame community. https://www.pygame.org/contribute.html
hobhits-Air PROJECT 3 % /usr/local/bin/python3 "/Users/shobhitsingh/PROJECT 3/Visualize.py"
(SDL 2.0.18, Python 3.10.7)
 pygame community. https://www.pygame.org/contribute.html
```

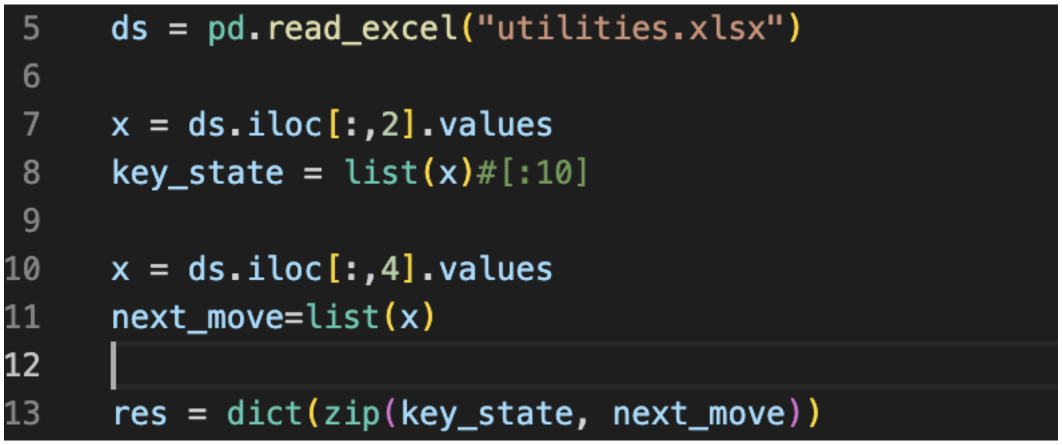
Note that node numbering starts from the top center (node 0) and goes clockwise.

BLUE - AGENT
RED - PREDATOR
GREEN - PREY

1. AGENT U

Since we already stored the next best move for every state, to implementation of agent U, we simply need to identify the current state and take the action already stored.

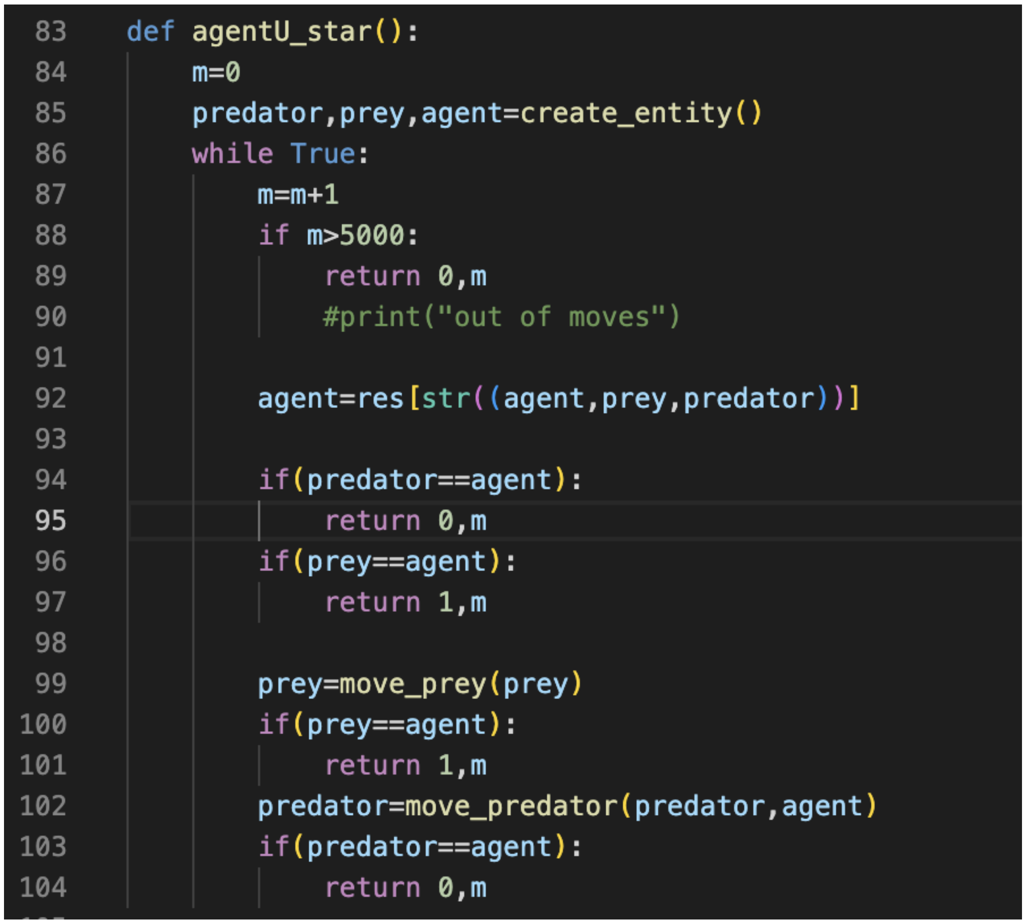
We first load our data and store it in a python dictionary using the following code.



```python
ds = pd.read_excel("utilities.xlsx")

x = ds.iloc[:,2].values
key_state = list(x)#[:10]

x = ds.iloc[:,4].values
next_move=list(x)

res = dict(zip(key_state, next_move))
```

The res dictionary stores all state as key and next best move as value.

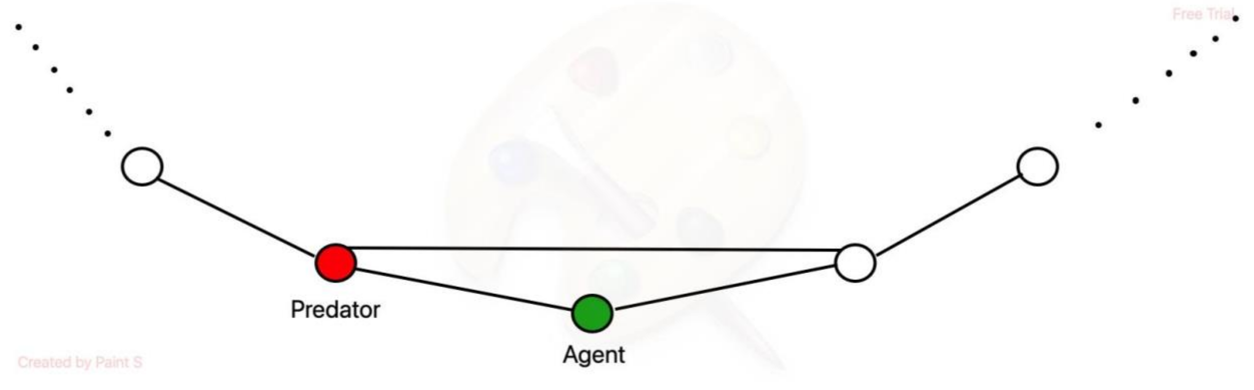
Now all we need to do is update the agents position by lookup in the res dictionary. We use the below code to do that.



```python
def agentU_star():
    m=0
    predator,prey,agent=create_entity()
    while True:
        m=m+1
        if m>5000:
            return 0,m
            #print("out of moves")

        agent=res[str((agent,prey,predator))]

        if(predator==agent):
            return 0,m
        if(prey==agent):
            return 1,m

        prey=move_prey(prey)
        if(prey==agent):
            return 1,m
        predator=move_predator(predator,agent)
        if(predator==agent):
            return 0,m
```

Once we have the 3-entity position, we update the agent position from the res dictionary. The checking of end condition and movement of prey using move_prey() and move_predator() functions remain exactly the same. We return 0,1 for success/loss (1 adds a win to the success rate whereas 0 doesn't) and **m** for the number of moves it took. Basically, we are going to test the efficiency based on how fast the agent catches the prey.

Are there any starting states for which the agent will not be able to capture the prey? What causes this failure?



Let us consider the above scenario where the agent tries to take the next step. Wherein it can either decide to stay back or go to the next connected node. If the predator takes intelligent decisions based on the 0.6 probability, then it will always move closer to the agent. Even if the agent decides to stay back in its current node or decides to move to the next step, it will always be one step away from the predator. Therefore, in the next movement, the agent will die.

For this to happen, the agent node usually has a degree 2 or the extra connection is to the node behind the predator. So, for a graph that has a lot of nodes with degree 2 and the 3rd connections are small, this fail state occurs more. For a graph with 0 node's with degree 2 and all extra edges being long connections, this type of death occurs 0 times. Unfortunately, our graph has 4 nodes with degree 2

DATA ANALYSIS:

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 1,2 can be simulated by using agent1234.py file and Agent U as explained above can be run using agentU.py

After running all 3 agents 3000 times, we got the following results.

AGENT	SUCCESS RATE	Average no. of moves to catch the prey
Agent U	100%	8.2583
Agent 1	87.2%	16.7091
Agent 2	99.46%	27.9106

Note: Agent U doesn't always give 100% success rate. It might be that the starting state corresponds to certain death cases described previously. Luckily it didn't happen

We can clearly see that Agent U is the most efficient and yields the best success rate.
Agent 2 does give near perfect success rate, but its efficiency is way too bad. This is because when the predator is close, the agent turns scared mode and prioritizes running away from the predator and takes unnecessary steps away from the prey.

Are there states where 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.

Since the underlying rules of movement are different for each of these 3 agents, they might take different path to catch the prey. Agent U transition to the next optimal state while agent 1 and 2 moves depending on the distance from the other 2 entity. Agent 2 made the decision to maximize survivability trading efficiency for that and Agent 1 trying to close the distance from prey as soon as possible.

We ran all 3 agents on same graph with same prey movement simultaneously to check how what different path they take. (Run the same using AgentCompare.py)

agentU
Agent movement - [43, 44, 48, 49, 2, 3, 7, 6]
Prey movement - [5, 4, 9, 4, 9, 8, 5, 6]
Predator movement - [3, 2, 1, 0, 49, 2, 3]

agent1
Agent movement - [43, 46, 43, 46, 43, 46, 45, 0, 1, 6, 5, 8]
Prey movement - [5, 4, 9, 4, 9, 8, 5, 6, 7, 8, 9, 8]
Predator movement - [3, 2, 49, 2, 49, 48, 44, 45, 0, 49, 0]

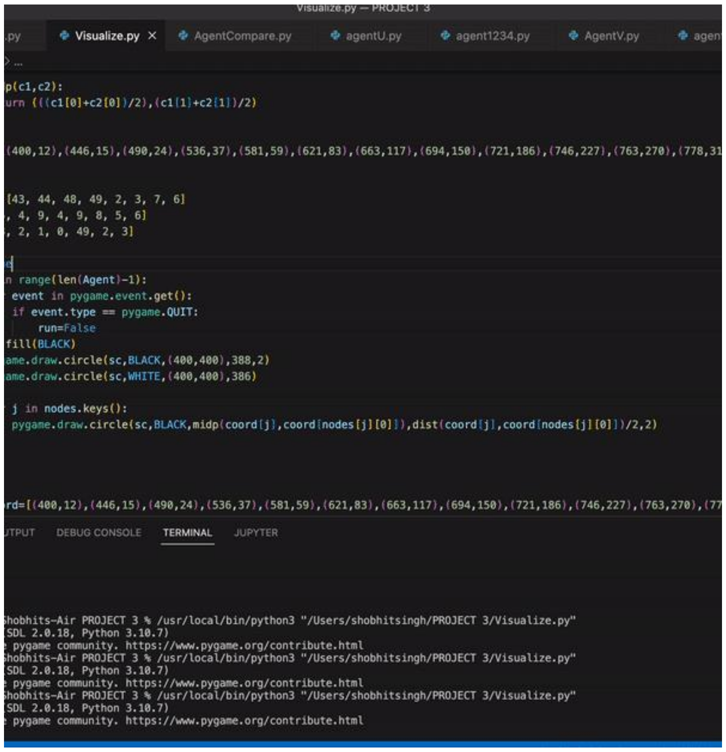
agent2
Agent movement - [43, 42, 41, 40, 39, 40, 39, 40, 39, 38, 39, 41, 42, 43, 44, 48, 49, 2, 3, 4, 9]
Prey movement - [5, 4, 9, 4, 9, 8, 5, 6, 7, 8, 9, 8, 9, 4, 9, 8, 5, 8, 5, 8, 9]
Predator movement - [3, 2, 49, 48, 44, 43, 42, 41, 42, 37, 38, 39, 40, 41, 42, 43, 44, 48, 47, 48]

We can see that all 3 agent takes different path and have different efficiency. Note that prey movement is the same for all, but a predator is different as it depends on agent location which varies between the 3

We can observe the path in the below visualization

Visualization of the above 3 cases (*Note that node numbering starts from the top center (node 0) and goes clockwise.***)**

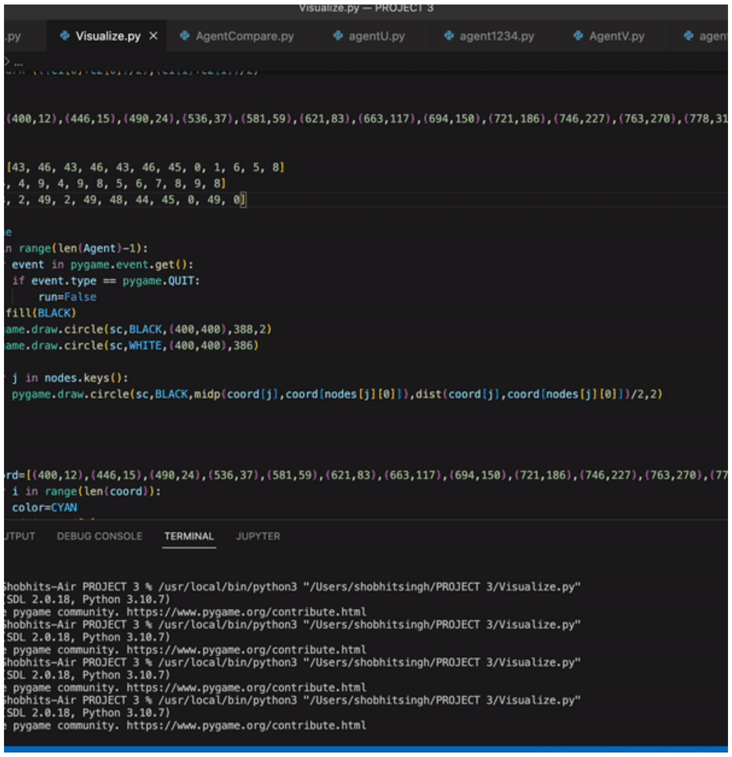
AGENT-U



```
p(c1,c2):
urn (((c1[0]+c2[0])/2),(c1[1]+c2[1])/2)


(400,12),(446,15),(490,24),(536,37),(581,59),(621,83),(663,117),(694,150),(721,186),(746,227),(763,270),(778,31


[43, 44, 48, 49, 2, 3, 7, 6]
, 4, 9, 4, 9, 8, 5, 6]
, 2, 1, 0, 49, 2, 3]

e
n range(len(Agent)-1):
 event in pygame.event.get():
 if event.type == pygame.QUIT:
     run=False
fill(BLACK)
ame.draw.circle(sc,BLACK,(400,400),388,2)
ame.draw.circle(sc,WHITE,(400,400),386)

 j in nodes.keys():
 pygame.draw.circle(sc,BLACK,midp(coord[j],coord[nodes[j][0]]),dist(coord[j],coord[nodes[j][0]])/2,2)




rd=[(400,12),(446,15),(490,24),(536,37),(581,59),(621,83),(663,117),(694,150),(721,186),(746,227),(763,270),(77
```

```
hobhits-Air PROJECT 3 % /usr/local/bin/python3 "/Users/shobhitsingh/PROJECT 3/Visualize.py"
(SDL 2.0.18, Python 3.10.7)
 pygame community. https://www.pygame.org/contribute.html
hobhits-Air PROJECT 3 % /usr/local/bin/python3 "/Users/shobhitsingh/PROJECT 3/Visualize.py"
(SDL 2.0.18, Python 3.10.7)
 pygame community. https://www.pygame.org/contribute.html
hobhits-Air PROJECT 3 % /usr/local/bin/python3 "/Users/shobhitsingh/PROJECT 3/Visualize.py"
(SDL 2.0.18, Python 3.10.7)
 pygame community. https://www.pygame.org/contribute.html
```

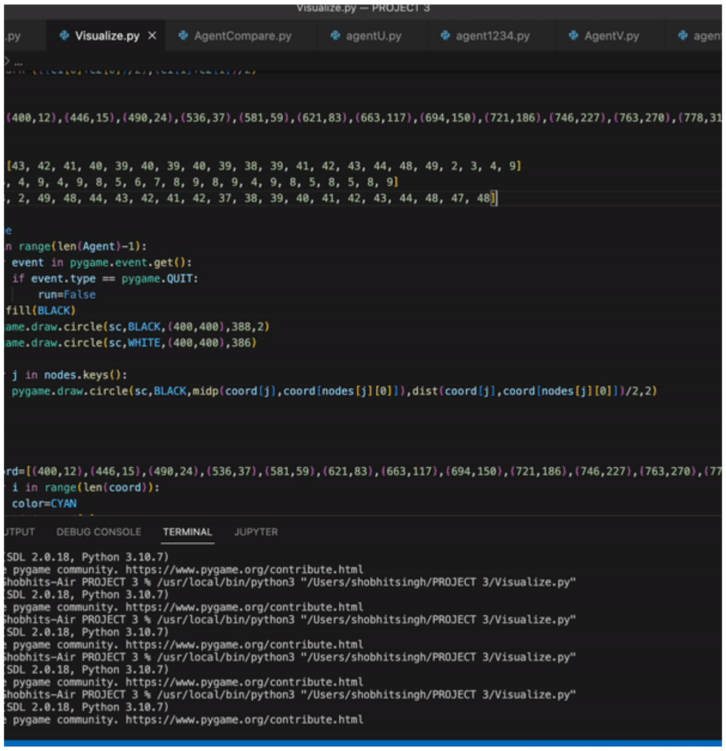
AGENT-1



```
(400,12),(446,15),(490,24),(536,37),(581,59),(621,83),(663,117),(694,150),(721,186),(746,227),(763,270),(778,31


[43, 46, 43, 46, 43, 46, 45, 0, 1, 6, 5, 8]
, 4, 9, 4, 9, 8, 5, 6, 7, 8, 9, 8]
, 2, 49, 2, 49, 48, 44, 45, 0, 49, 0]

e
n range(len(Agent)-1):
 event in pygame.event.get():
 if event.type == pygame.QUIT:
     run=False
fill(BLACK)
ame.draw.circle(sc,BLACK,(400,400),388,2)
ame.draw.circle(sc,WHITE,(400,400),386)

 j in nodes.keys():
 pygame.draw.circle(sc,BLACK,midp(coord[j],coord[nodes[j][0]]),dist(coord[j],coord[nodes[j][0]])/2,2)




rd=[(400,12),(446,15),(490,24),(536,37),(581,59),(621,83),(663,117),(694,150),(721,186),(746,227),(763,270),(77
 i in range(len(coord)):
 color=CYAN
```

```
hobhits-Air PROJECT 3 % /usr/local/bin/python3 "/Users/shobhitsingh/PROJECT 3/Visualize.py"
(SDL 2.0.18, Python 3.10.7)
 pygame community. https://www.pygame.org/contribute.html
hobhits-Air PROJECT 3 % /usr/local/bin/python3 "/Users/shobhitsingh/PROJECT 3/Visualize.py"
(SDL 2.0.18, Python 3.10.7)
 pygame community. https://www.pygame.org/contribute.html
hobhits-Air PROJECT 3 % /usr/local/bin/python3 "/Users/shobhitsingh/PROJECT 3/Visualize.py"
(SDL 2.0.18, Python 3.10.7)
 pygame community. https://www.pygame.org/contribute.html
hobhits-Air PROJECT 3 % /usr/local/bin/python3 "/Users/shobhitsingh/PROJECT 3/Visualize.py"
(SDL 2.0.18, Python 3.10.7)
 pygame community. https://www.pygame.org/contribute.html
```

AGENT-2


```
(400,12),(446,15),(490,24),(536,37),(581,59),(621,83),(663,117),(694,150),(721,186),(746,227),(763,270),(778,31

[43, 42, 41, 40, 39, 40, 39, 40, 39, 38, 39, 41, 42, 43, 44, 48, 49, 2, 3, 4, 9]
, 4, 9, 4, 9, 8, 5, 6, 7, 8, 9, 8, 9, 4, 9, 8, 5, 8, 5, 8, 9]
, 2, 49, 48, 44, 43, 42, 41, 42, 37, 38, 39, 40, 41, 42, 43, 44, 48, 47, 48]

e
n range(len(Agent)-1):
 event in pygame.event.get():
  if event.type == pygame.QUIT:
      run=False
fill(BLACK)
ame.draw.circle(sc,BLACK,(400,400),388,2)
ame.draw.circle(sc,WHITE,(400,400),386)

 j in nodes.keys():
 pygame.draw.circle(sc,BLACK,midp(coord[j],coord[nodes[j][0]]),dist(coord[j],coord[nodes[j][0]])/2,2)



rd=[(400,12),(446,15),(490,24),(536,37),(581,59),(621,83),(663,117),(694,150),(721,186),(746,227),(763,270),(77
 i in range(len(coord)):
 color=CYAN
```

```
(SDL 2.0.18, Python 3.10.7)
 pygame community. https://www.pygame.org/contribute.html
hobhits-Air PROJECT 3 % /usr/local/bin/python3 "/Users/shobhitsingh/PROJECT 3/Visualize.py"
(SDL 2.0.18, Python 3.10.7)
 pygame community. https://www.pygame.org/contribute.html
hobhits-Air PROJECT 3 % /usr/local/bin/python3 "/Users/shobhitsingh/PROJECT 3/Visualize.py"
(SDL 2.0.18, Python 3.10.7)
 pygame community. https://www.pygame.org/contribute.html
hobhits-Air PROJECT 3 % /usr/local/bin/python3 "/Users/shobhitsingh/PROJECT 3/Visualize.py"
(SDL 2.0.18, Python 3.10.7)
 pygame community. https://www.pygame.org/contribute.html
hobhits-Air PROJECT 3 % /usr/local/bin/python3 "/Users/shobhitsingh/PROJECT 3/Visualize.py"
(SDL 2.0.18, Python 3.10.7)
 pygame community. https://www.pygame.org/contribute.html
```

Why the agents make different choice?

- Agents 1 and 2 make the decision based on the distance between them and prey and predator. This does not consider the positions of the future and only considers current distance. Thus, the movement is not optimal, and the agent takes detour that does seem like getting closer to prey at that point of move only and does not guarantee best action. On the other hand, Utilities were calculated by considering all possible state and how they affect other states. Moving based on Utility guarantees the agent is transitioning to a better state every time.
- The agents 1 and 2 act based on distance which can be same for more than one action. At this point, the agent chooses one of these actions randomly with nothing to differentiate the action. So, the agent might take different action as the movement becomes random. On the other hand, Agent U choose action trying to minimize the utility which is rarely same. So, there is less randomness involved and the agent takes the best action.

2. AGENT V

For Agent U, we need to store Utilities for all 125000 states to use as a lookup table. To avoid storing all the values, we build a neural network model that trains based on the Utilities we found when implementing Agent U. The model is used to compute the Utility of a given state as and when we need them instead of storing all utilities that might not even need.

NEURAL NETWORK (**what kind of model are you taking V to be?**)

A neural network consists of one input layer, one output layer, and one or more hidden layers depending on the complexity of the network. In the illustration below, the layer Out-0 is the input layer and layer Out-N is the output layer. The training algorithm used is gradient descent.

To train a Neural network, we first initialize the weights and perform Forward propagation to find the output at each layer and the final output.
The linking equation between the node of two consecutive layers that is used for forward propagation is given by,

$$Out_i^l = \sigma(W^l(i).Out^{l-1})$$

Where, i = node in layer l and σ is activation function

Once we have the output, we can find the Error/Loss by comparing it to the training output.
Using the derivative of Loss, we can adjust the weights to minimize the loss as follows.
Here, alpha is the learning rate.

$$W(k+1) = W(k) - \alpha\nabla_w L_i\big(W(k)\big)$$

To compute the gradient descent or Stochastic gradient descent we compute the derivative of the loss. To do this, we use Back propagation.

$$\frac{dLoss}{dW^l(i)_j} = \left(\frac{dOut_i^l}{dW^l(i)_j}\right)\left(\frac{dLoss^l}{dOut_i^l}\right)$$

We calculate the derivative of the loss in two stages:

1. $\frac{dOut_i^l}{dW^l(i)_j} = \sigma'\big(W^l(i).Out^{l-1}\big).\left(\frac{d(W^l(I).Out^{l-1})}{dW^l(i)_j}\right)$
   $= \sigma'(W^l(i).Out^{l-1}).Out_j^{l-1}$

2. $\frac{dLoss}{dOut_i^l}$
   a. If the layer is the final output layer i.e.,$\frac{dLoss}{dOut_i^l}$

$$L = \Sigma_j\big(Out_j^K - y_j\big)^2$$

$$\frac{dL}{dOut_i^K} = 2(Out_i^K - y_i)$$

b. If it's any of the hidden layers i.e, L<K

$$\frac{dL}{dOut_i^l} = \Sigma_{Node\ K\ in\ layer\ L+1}\left(\frac{dOut_K^{l+1}}{dOut_i^l}\right)\left(\frac{dL}{dOut_K^{l+1}}\right)$$

$$Out_K^{l+1} = \sigma\left(W^{l+1}(k).Out^l\right)$$

$$\frac{dOut_K^{l+1}}{dOut_i^l} = \sigma'\left(W^{l+1}(k).Out^l\right).\ \frac{d}{dOut_i^L}(W^{l+1}(k).Out^l)$$

$$= \sigma'\left(W^{l+1}(k).Out^l\right).\ W^{l+1}(k)_i$$

Thus, we can use Backpropagation at each layer to get the gradient of Loss at the level and adjust the weights till Loss converges to 0

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

We are using the following inputs when training our neural network:

1. *Distance between agent and prey*: the number of steps to catch the prey depends on how far the prey is. The utility is directly proportional to the distance
2. *Distance between agent and predator*: Since the agent is trying to avoid getting caught by the predator, the predator being in proximity does affect the movement of the agent making it take longer to catch the prey.
3. *Distance between prey and predator*: While this distance doesn't seem to affect the agent much at first, in combination with the other 2 inputs above, it does tell us about the position agent, prey, and predator. In a linear path, comparing the distances does tell us if the predator is between the agent and prey or on either side. The classification is not that accurate in a circular path if the other 2 entity is on two ends of the circumference, but for most states, it is still relevant.
4. *Biased term* (i.e., 1) : We add an input as 1 to act as biased term as whatever weight gets added to it, will give that weight as constant

Note: We did try to train with actual node position as inputs included but the performance was worse. If the agent is at node 1 and prey is at 0 or if the agent is at node 49 and prey is at 0, the Utility of both cases will be 1. However, it becomes difficult for the model to assign weights that makes input 49 and 1 to generate output 1. So, we didn't include node positions as inputs.
Note: since we only use distance as inputs, we are representing different states with equal distance as same. But since the Output for these states are almost same, the Model still works.

Is OVERFITTING an issue here?

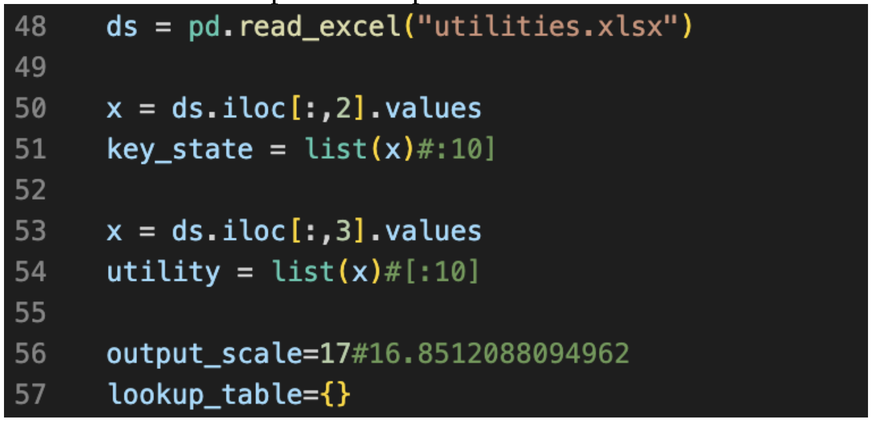
Overfitting is **not an issue** here as the scope of the project is only one graph. The better we fit our model to the states the more accurately we will find the utility as we only need to find utilities of the input states later. Basically, there is no data that we didn't train out model with. So the better it fits to all data points, the better the results.

It is an issue if we consider other randomly generated graphs. Since we are training the Model using Utilities of a single graph, if we are overfitting, it does not work very accurately for them. Value of utility is different for the relative position on as position on have degree 2 and overfitting will not give a generalized output for these exceptions. We are taking infinite Utility states out of the picture of model to reduce this problem a lot.

Note: to make overfitting less of a problem when we change graphs, we can take another input that give's insights about the connectivity of the graph. Since graphs have 50 nodes and 50 common edges, the only difference if the 25(max) nodes that interconnect. They can be well connected with long connections or poorly connected with small interconnects. We can pass an input like how many steps it takes to reach 49 from 0 taking the longer route to get an idea of how well connected the graph is. This should make the model adapt to different graphs.

TRAINING THE MODEL

To train the model, we first read the training data we stored while implementing U_star agent. The code to read and store states and respective outputs is as follows:



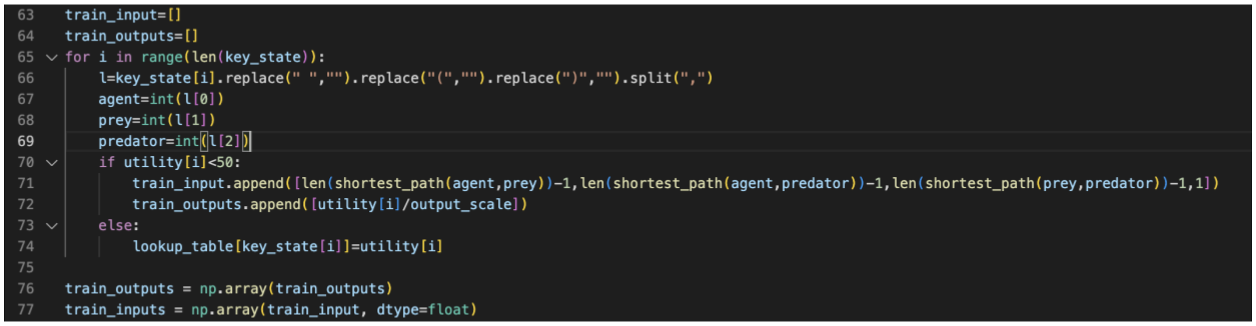
```
48    ds = pd.read_excel("utilities.xlsx")
49
50    x = ds.iloc[:,2].values
51    key_state = list(x)#:10]
52
53    x = ds.iloc[:,3].values
54    utility = list(x)#[:10]
55
56    output_scale=17#16.8512088094962
57    lookup_table={}
```

Here, key_state stores all state values (Line 51), and utility stores all output values (Line 54).
Two things to consider about the training inputs:

a. We are using sigmoid as an activation function as it gave the best accuracy. Since it generates output between 0 and 1, We need to scale down our input by dividing by the highest utility (17 rounding off). We can later scale up by the same factor to retrieve the Utility.

b. Since we have some infinite Utility for some case, it was messing up the convergence of the Model. To solve for these, we are storing this utility in a separate smaller lookup table and not using them to train or be predicted by the Model V. are only storing a small amount and still solving the problem of storing the large dataset.

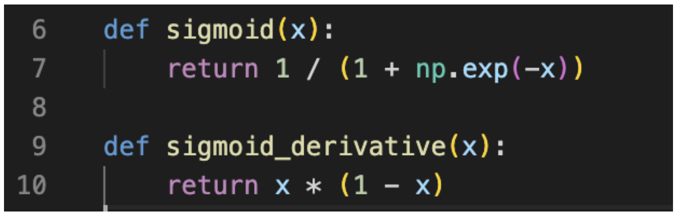
The following code then evaluate and generate the input and output datasets:



```
63  train_input=[]
64  train_outputs=[]
65  for i in range(len(key_state)):
66      l=key_state[i].replace(" ","").replace("(","").replace(")","").split(",")
67      agent=int(l[0])
68      prey=int(l[1])
69      predator=int(l[2])
70      if utility[i]<50:
71          train_input.append([len(shortest_path(agent,prey))-1,len(shortest_path(agent,predator))-1,len(shortest_path(prey,predator))-1,1])
72          train_outputs.append([utility[i]/output_scale])
73      else:
74          lookup_table[key_state[i]]=utility[i]
75
76  train_outputs = np.array(train_outputs)
77  train_inputs = np.array(train_input, dtype=float)
```

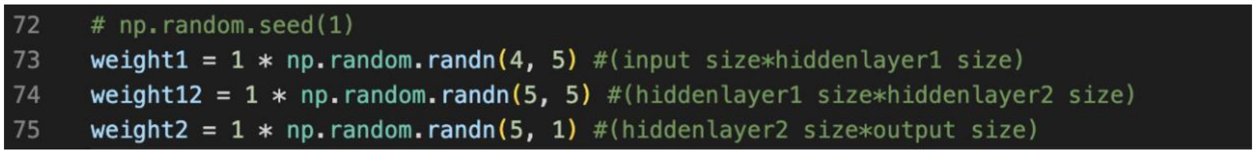
We extract all 3 entity positions and create inputs and scaled-down outputs of all finite utilities (Line 66-72). Else, we store the finite values in a lookup table.

Before we start with training the model, we need to define the sigmoid activation function and its derivative as follows:



```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)
```

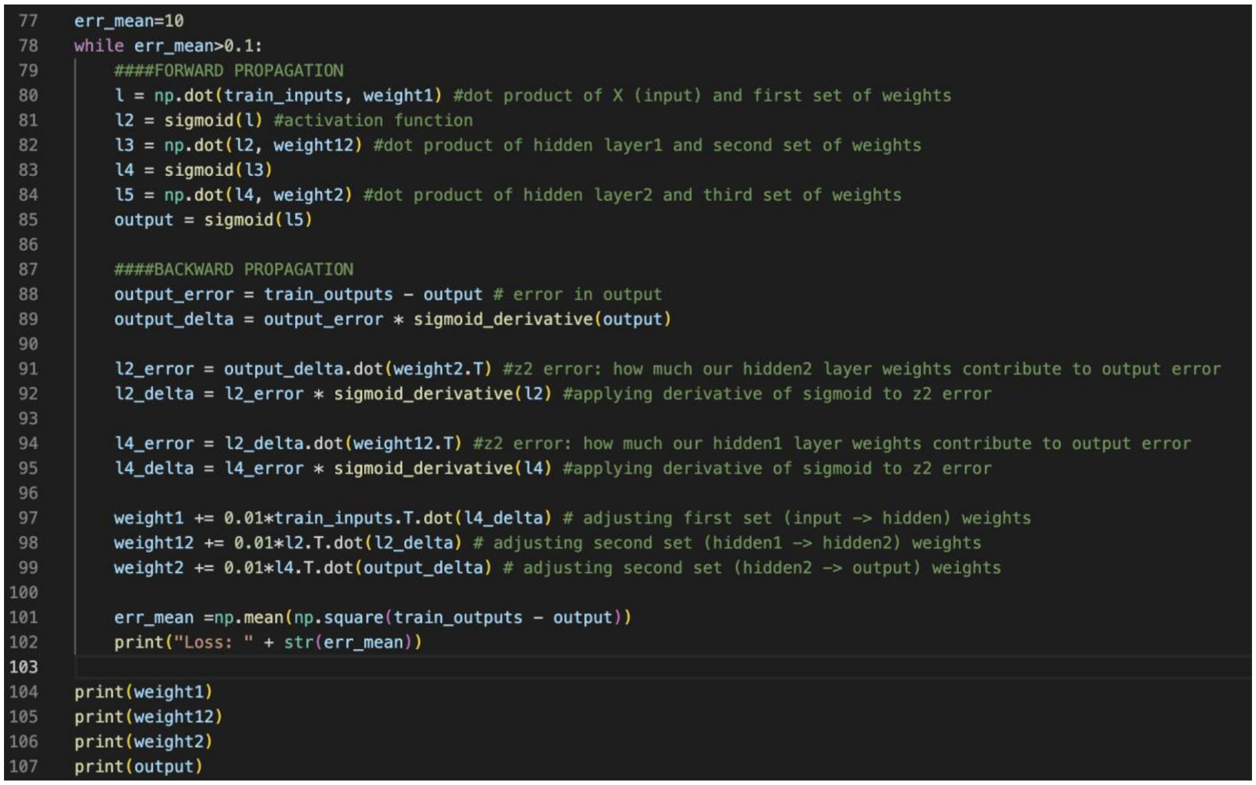
Now, we are making a neural network with the input layer having 4 nodes, 2 hidden layers each having 5 nodes and finally an output layer having 1 node. To initialize weights for this model, we use the following code.



```
# np.random.seed(1)
weight1 = 1 * np.random.randn(4, 5) #(input size*hiddenlayer1 size)
weight12 = 1 * np.random.randn(5, 5) #(hiddenlayer1 size*hiddenlayer2 size)
weight2 = 1 * np.random.randn(5, 1) #(hiddenlayer2 size*output size)
```

We just need to make sure we are taking the correct dimension.

Once we have all the dependencies, we can train the model using the below code.



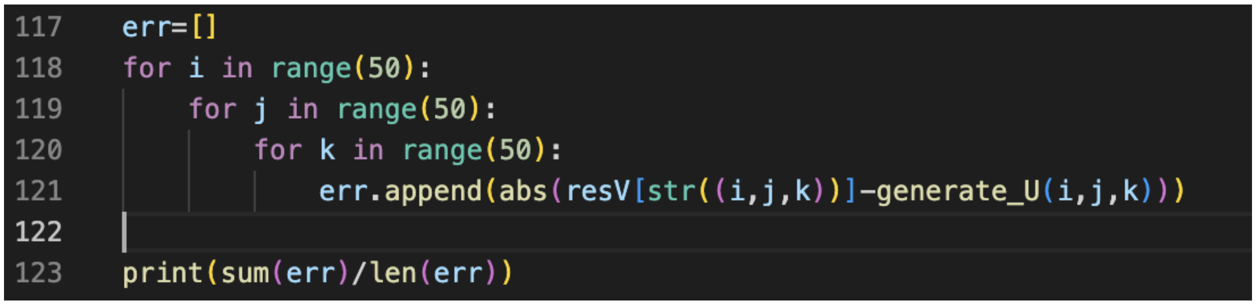
```
77  err_mean=10
78  while err_mean>0.1:
79      ####FORWARD PROPAGATION
80      l = np.dot(train_inputs, weight1) #dot product of X (input) and first set of weights
81      l2 = sigmoid(l) #activation function
82      l3 = np.dot(l2, weight12) #dot product of hidden layer1 and second set of weights
83      l4 = sigmoid(l3)
84      l5 = np.dot(l4, weight2) #dot product of hidden layer2 and third set of weights
85      output = sigmoid(l5)
86
87      ####BACKWARD PROPAGATION
88      output_error = train_outputs - output # error in output
89      output_delta = output_error * sigmoid_derivative(output)
90
91      l2_error = output_delta.dot(weight2.T) #z2 error: how much our hidden2 layer weights contribute to output error
92      l2_delta = l2_error * sigmoid_derivative(l2) #applying derivative of sigmoid to z2 error
93
94      l4_error = l2_delta.dot(weight12.T) #z2 error: how much our hidden1 layer weights contribute to output error
95      l4_delta = l4_error * sigmoid_derivative(l4) #applying derivative of sigmoid to z2 error
96
97      weight1 += 0.01*train_inputs.T.dot(l4_delta) # adjusting first set (input -> hidden) weights
98      weight12 += 0.01*l2.T.dot(l2_delta) # adjusting second set (hidden1 -> hidden2) weights
99      weight2 += 0.01*l4.T.dot(output_delta) # adjusting second set (hidden2 -> output) weights
100
101     err_mean =np.mean(np.square(train_outputs - output))
102     print("Loss: " + str(err_mean))
103
104 print(weight1)
105 print(weight12)
106 print(weight2)
107 print(output)
```

As explained before, we perform forward propagation by multiplying the weights with inputs at each layer and taking the sigmoid function to add the non-linearity to the model (Line 80-85).
Once we get an output, we check for errors at each layer using backward propagation as described above (Line 88-95).
We then adjust the weights in proportion to the contribution of each node to the error. We repeat this process until the "err_mean" converges to 0.
Once we have minimized the error, we print and store all the weights to use as our model V to compute the Utility

Note: It is advised to use recursion if you have multiple hidden layers, but since we have only 2 layers, we did the propagation in series to keep it simple.

HOW ACCURATE IS V?

The accuracy of Model V should be pretty good given we were able to converge the loss to below 0.1.
We can further verify this accuracy by comparing the Utility calculated by the model with the Utility calculated by value iteration. To find the average difference, we use the following code:

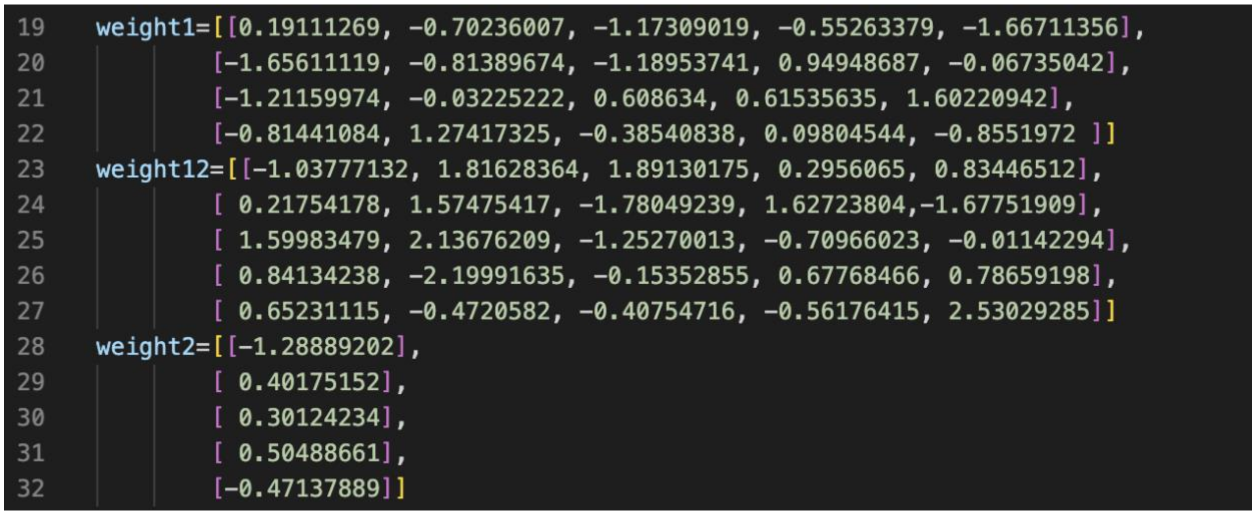


```python
err=[]
for i in range(50):
    for j in range(50):
        for k in range(50):
            err.append(abs(resV[str((i,j,k))]-generate_U(i,j,k)))

print(sum(err)/len(err))
```

The mean difference comes out to be 3.11093479. This means that that on average, the model V gives an error of around 3 which is not that good but given the decisions are made as per relative Utility, the agent still shows good success rate.

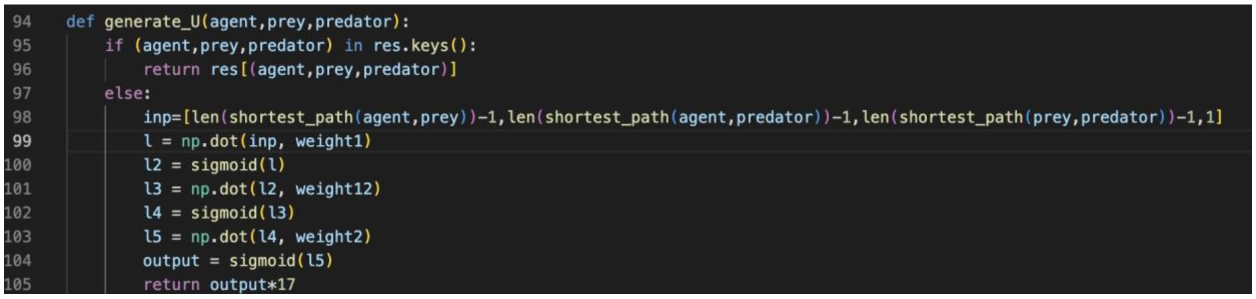
IMPLEMENTING Agent V

To implement the Agent V, we first store the weights we got from training the model.



```python
weight1=[[0.19111269, -0.70236007, -1.17309019, -0.55263379, -1.66711356],
         [-1.65611119, -0.81389674, -1.18953741, 0.94948687, -0.06735042],
         [-1.21159974, -0.03225222, 0.608634, 0.61535635, 1.60220942],
         [-0.81441084, 1.27417325, -0.38540838, 0.09804544, -0.8551972 ]]
weight12=[[-1.03777132, 1.81628364, 1.89130175, 0.2956065, 0.83446512],
         [ 0.21754178, 1.57475417, -1.78049239, 1.62723804,-1.67751909],
         [ 1.59983479, 2.13676209, -1.25270013, -0.70966023, -0.01142294],
         [ 0.84134238, -2.19991635, -0.15352855, 0.67768466, 0.78659198],
         [ 0.65231115, -0.4720582, -0.40754716, -0.56176415, 2.53029285]]
weight2=[[-1.28889202],
         [ 0.40175152],
         [ 0.30124234],
         [ 0.50488661],
         [-0.47137889]]
```

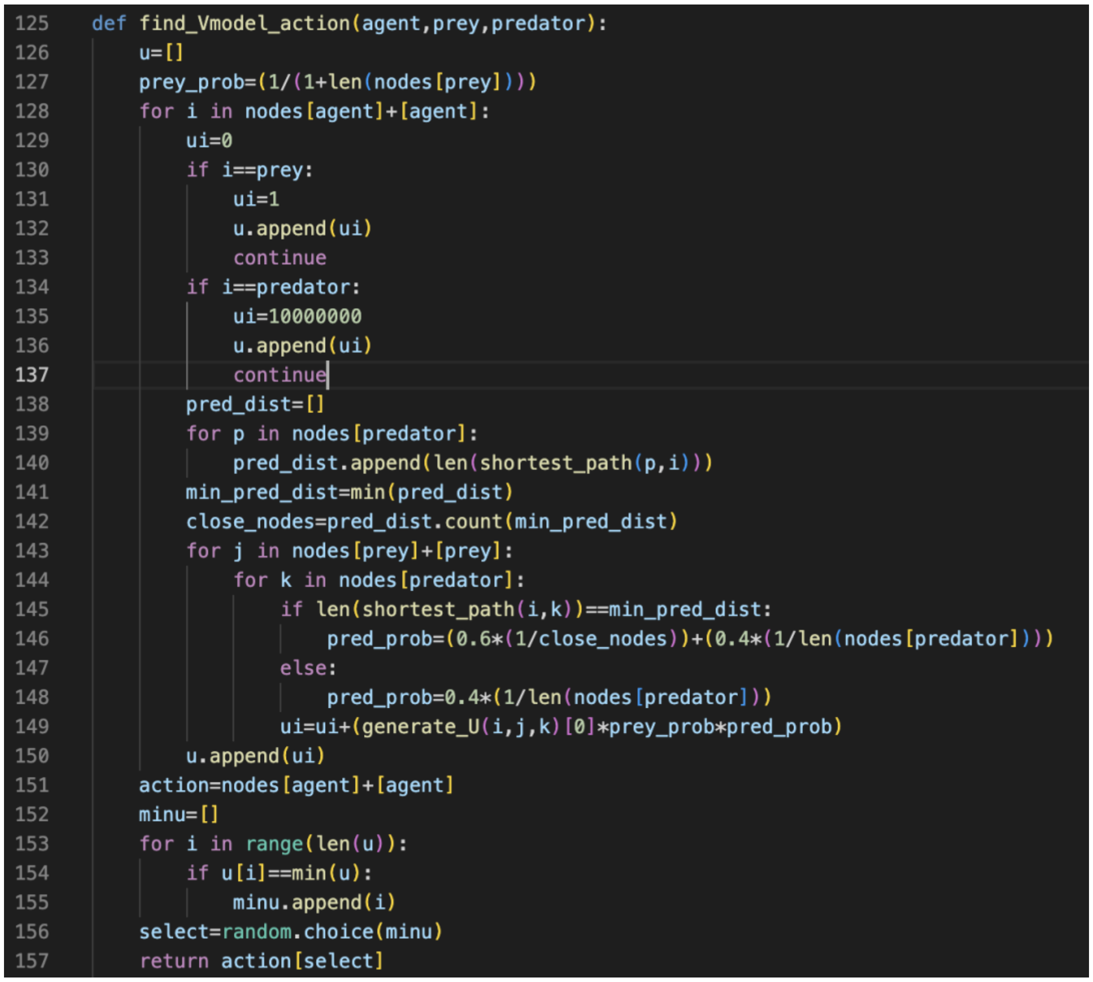
We need to then define a function that calculates the Utility based on our model. The function simply takes the above weight and implements forward propagation once.



```python
def generate_U(agent,prey,predator):
    if (agent,prey,predator) in res.keys():
        return res[(agent,prey,predator)]
    else:
        inp=[len(shortest_path(agent,prey))-1,len(shortest_path(agent,predator))-1,len(shortest_path(prey,predator))-1,1]
        l = np.dot(inp, weight1)
        l2 = sigmoid(l)
        l3 = np.dot(l2, weight12)
        l4 = sigmoid(l3)
        l5 = np.dot(l4, weight2)
        output = sigmoid(l5)
        return output*17
```

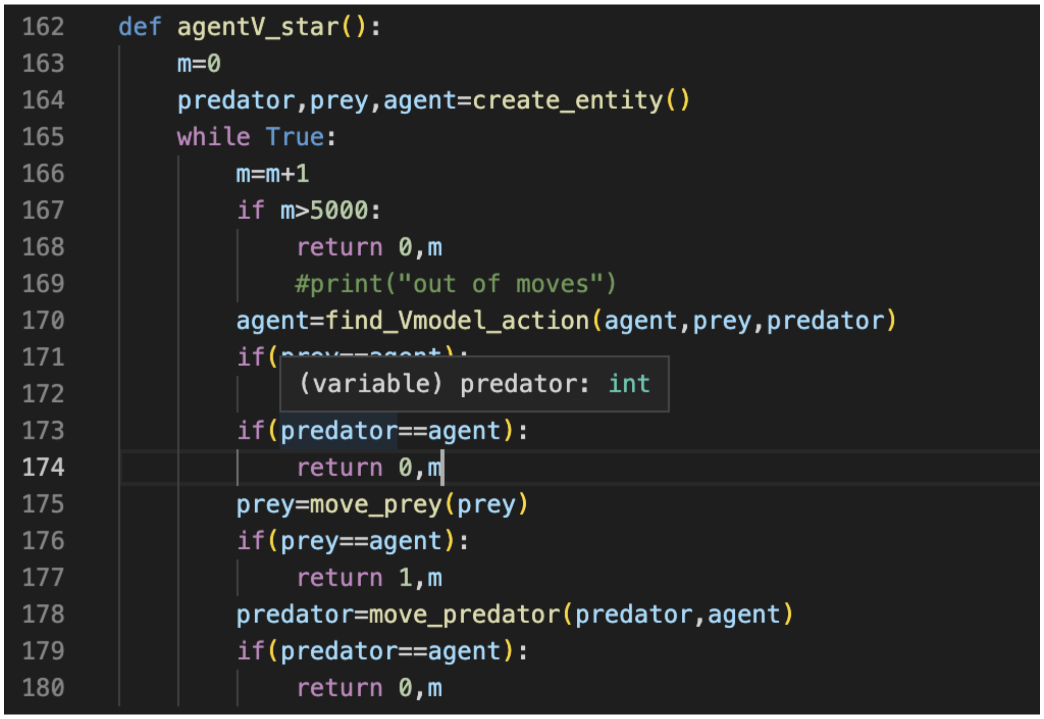
We also need to scale up the value by a factor of 17 and check in the lookup table for Utilities we didn’t consider when training.

Next, we define a function that calculates the best action based on the utility of the probable state. The code is very similar to what we used to implement Value iteration, but we don't wait for convergence.



```python
def find_Vmodel_action(agent,prey,predator):
    u=[]
    prey_prob=(1/(1+len(nodes[prey])))
    for i in nodes[agent]+[agent]:
        ui=0
        if i==prey:
            ui=1
            u.append(ui)
            continue
        if i==predator:
            ui=10000000
            u.append(ui)
            continue
        pred_dist=[]
        for p in nodes[predator]:
            pred_dist.append(len(shortest_path(p,i)))
        min_pred_dist=min(pred_dist)
        close_nodes=pred_dist.count(min_pred_dist)
        for j in nodes[prey]+[prey]:
            for k in nodes[predator]:
                if len(shortest_path(i,k))==min_pred_dist:
                    pred_prob=(0.6*(1/close_nodes))+(0.4*(1/len(nodes[predator])))
                else:
                    pred_prob=0.4*(1/len(nodes[predator]))
                ui=ui+(generate_U(i,j,k)[0]*prey_prob*pred_prob)
        u.append(ui)
    action=nodes[agent]+[agent]
    minu=[]
    for i in range(len(u)):
        if u[i]==min(u):
            minu.append(i)
    select=random.choice(minu)
    return action[select]
```

Once we can find the best action for every state, we can simulate the agent easily as follows



```
162  def agentV_star():
163      m=0
164      predator,prey,agent=create_entity()
165      while True:
166          m=m+1
167          if m>5000:
168              return 0,m
169              #print("out of moves")
170          agent=find_Vmodel_action(agent,prey,predator)
171          if(prey==agent):
172              (variable) predator: int
173          if(predator==agent):
174              return 0,m
175          prey=move_prey(prey)
176          if(prey==agent):
177              return 1,m
178          predator=move_predator(predator,agent)
179          if(predator==agent):
180              return 0,m
```

We simply use the find_Vmodel_action() to change the agent location at each iteration(line 170)

DATA ANALYSIS

Agent V was simulated 3000 times on our graph with different initial states.

The success rate came out to be **99.86%** and the average number of moves for the game to end is **9.5073.** It's efficiency is lower than Agent U as out Model cannot predict the Utility with 100% accuracy which was expected.

The reason the success rate is not affected despite there being some error in V model is that we are not using the Model to calculate infinite state utility. The agent still knows with 100% accuracy which states to avoid not dying. However, the average number of moves did increase a little compared to AgentU as the Utility was predicted with some error meaning the agent is not following optimal path. The efficiency is still pretty good compared to agent 1 and 2 so I would call this a success.

B. THE PARTIAL PREY INFORMATION SETTING:

In this setting, the Agent always knows where the Predator is, but does not necessarily know where the Prey is. The Agent gets a drone to survey a node for prey before it moves. We need to keep track of a belief state for where the Prey is, a collection of probabilities for each node that the Prey is there. This collection of probabilities is used to guess where the prey is and make decisions according

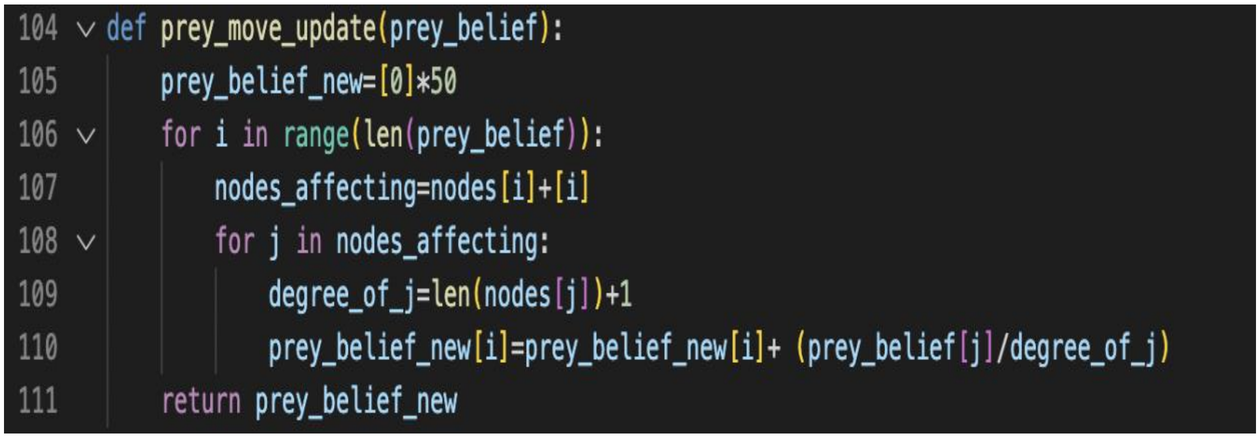
1. AGENT U partial

Since we only know the position of the agent and predator in this setting, we cannot have 50*50*50 visual states anymore. The total number of permutations of states can only be 50*50 (excluding prey combination). However, the prey is still there but we just don't know the exact 3 entity state. So, we calculate the Utility of the state based on agent and predator by summing the Utilities of all possible states in proportion to the probability of that state being the case (i.e the probability of the prey being in a position that corresponds to the state).
We use the following expression to calculate the Utility for a given agent position, predator position and a given belief of the prey.

$$U_{partial}(S_{agent}, S_{predator}, P) = \Sigma P s_{prey} U^*(S_{agent}, S_{predator}, S_{prey}). \text{ ---- Equation (1)}$$

So, we get a total of 2500 state utilities based on a belief. It should be noted that we need to recalculate the new 2500 Utilities every time the belief of prey position changes. Since the belief can be arranged in a near-infinite number of ways, the Total number of Utilities will be infinite as well. We just calculate 2500 as time as that's all we to determine the next move.

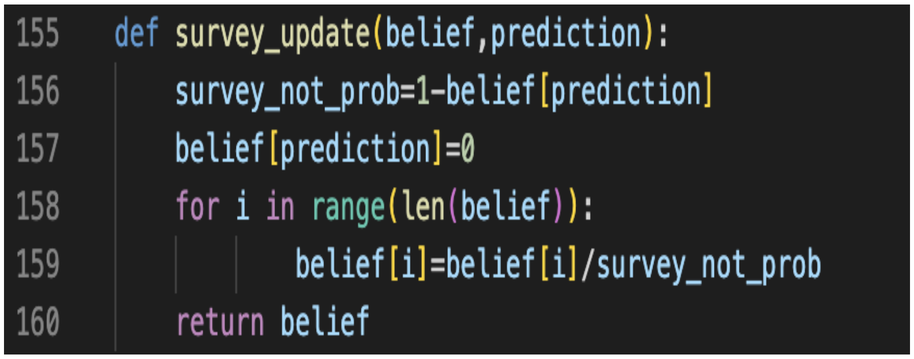
To implement and maintain the probability update of nodes, we use the following function:



```
def prey_move_update(prey_belief):
    prey_belief_new=[0]*50
    for i in range(len(prey_belief)):
        nodes_affecting=nodes[i]+[i]
        for j in nodes_affecting:
            degree_of_j=len(nodes[j])+1
            prey_belief_new[i]=prey_belief_new[i]+ (prey_belief[j]/degree_of_j)
    return prey_belief_new
```

We first initialize the prey_belief_new list where the index represents the node. Then we run a loop for each node and store all the nodes that will affect its probability (its neighbor and itself) in nodes_affecting variable. For each node, we can find the new belief as per our formula.

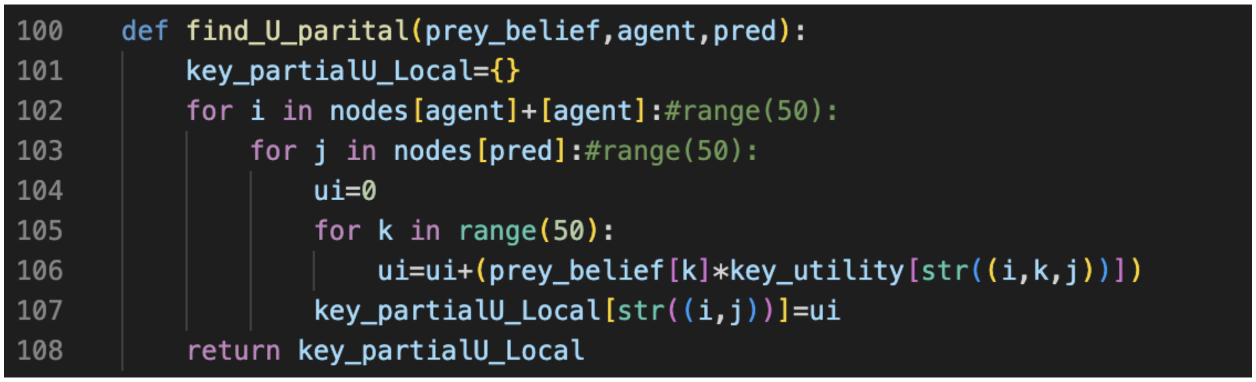
To implement redistribution of probabilities when a node is surveyed and prey wasn't there, we use the following function:



```python
def survey_update(belief,prediction):
    survey_not_prob=1-belief[prediction]
    belief[prediction]=0
    for i in range(len(belief)):
        belief[i]=belief[i]/survey_not_prob
    return belief
```

We simply divide all probabilities by (1-the probability of node surveyed) then make the node surveyed as 0. This will not affect the nodes whose probability is 0.

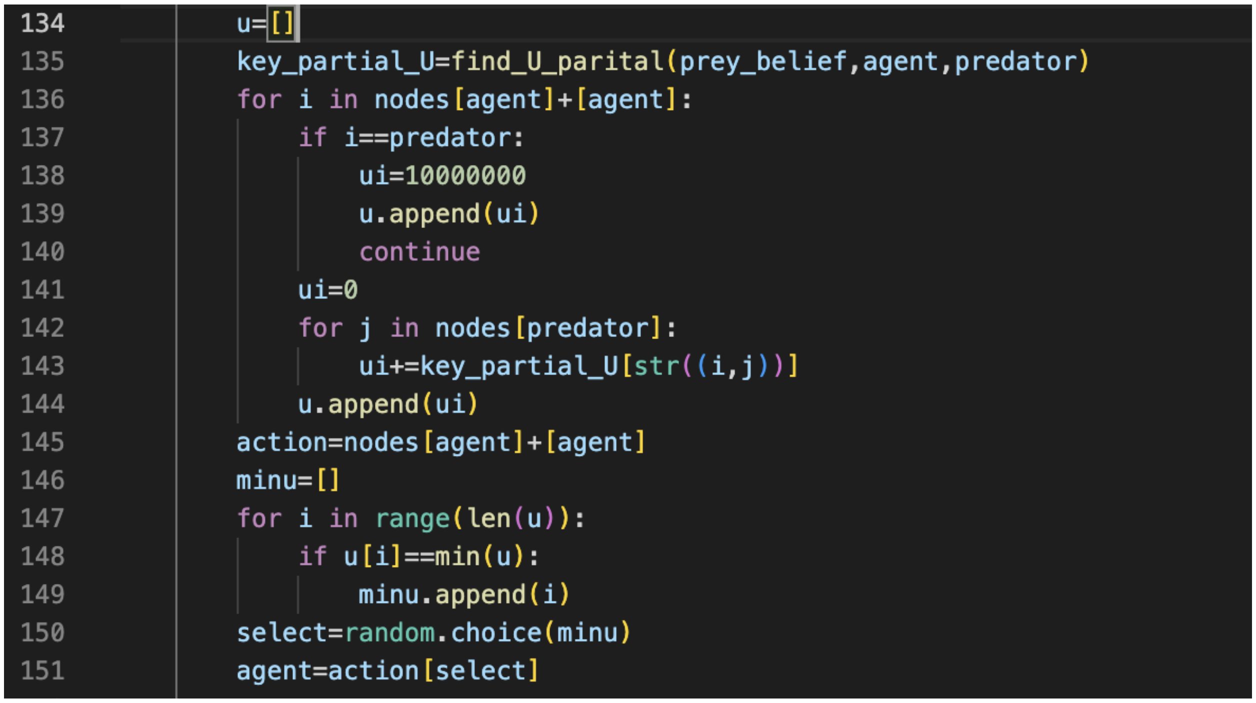
Once we have prey_belief, we can implement equation 1 as follows to get the partial information Utilities.



```python
100  def find_U_parital(prey_belief,agent,pred):
101      key_partialU_Local={}
102      for i in nodes[agent]+[agent]:#range(50):
103          for j in nodes[pred]:#range(50):
104              ui=0
105              for k in range(50):
106                  ui=ui+(prey_belief[k]*key_utility[str((i,k,j))])
107              key_partialU_Local[str((i,j))]=ui
108      return key_partialU_Local
```

The function calculate all relevant Utilites based on the agent and predator position for a given belief. It should be noted that while we can compute Utilities of all 2500 states for the belief by using the range (50) in loops (Line 102-103), We only need the utilities of states that we can reach in one move from current state, which is at max12 based on the action of agent and predator. Therefore, we are computing relevant state utilities instead of all 2500 as it makes the code significantly faster.

We survey the node and update the belief on movement and survey each time similar to project 2. We then compute Utilities for the belief before moving the agent at each step. The following code implements Agent U_partial.



```
134        u=[]
135        key_partial_U=find_U_parital(prey_belief,agent,predator)
136        for i in nodes[agent]+[agent]:
137            if i==predator:
138                ui=10000000
139                u.append(ui)
140                continue
141            ui=0
142            for j in nodes[predator]:
143                ui+=key_partial_U[str((i,j))]
144            u.append(ui)
145        action=nodes[agent]+[agent]
146        minu=[]
147        for i in range(len(u)):
148            if u[i]==min(u):
149                minu.append(i)
150        select=random.choice(minu)
151        agent=action[select]
```

The “find_U_partial” function in Line 135 calculates and store the utilities of all relevant states based on the current prey_belief and agent, predator position. Once we have the Utilities of all the states that can be reached from current state, we calculate the transitional Utility for each action that the agent can take and chose the minimum of all action (Line 136-144). If we have more than 1 action with same minimum Utility, we break ties at random (Line 146-150).

Order of movement and belief update for Agent U_partial:

1. Predict the prey location (Find max of all probability). Our prediction is 100% true if max probability is 1
2. Survey the prediction and Call survey_update function to reflect the feedback.
3. Update the utilities based on the updated prey belief
4. Agent moves using the utilities
5. Call survey_update function at agent’s location (if the hunt isn’t already over).
6. Prey Moves (and Predator moves as well)
7. Call the prey_move_update function to update the probabilities.
8. Call survey_update function at agent’s location (if the hunt isn’t already over).

DATA ANALYSIS:

Agent 3,4 can be simulated by using agent1234.py file and Agent U_partial as explained above can be run using agentpartial.py

After running all 3 agent 3000 times, we got the following results.

AGENT	SUCCESS RATE	Average no. of moves to catch the prey
Agent U_partial	99.93%	23.594
Agent 3	89.36%	30.947
Agent 4	99.66%	40.1216

Note: Agent U_partial doesn't give 100% success rate might be because it initialized 2 times to a certain death condition. It should still perform at 100% as we are avoiding states with infinite utility as we know the predator location for sure.

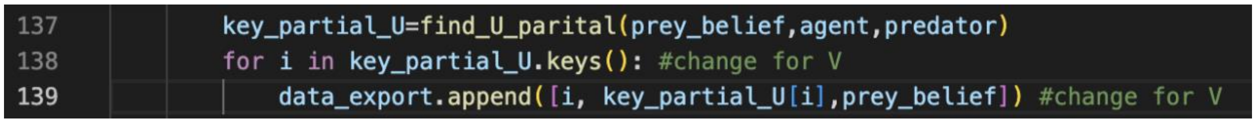
We can clearly see that Agent U is the most efficient and yields the best success rate.
Agent 2 does give near perfect success rate, but its efficiency is bad. This is because when the predator is close, the agent turns scared mode and prioritizes running away from the predator and takes unnecessary steps away from the prey.

Is the U_Partial agent optimal?

The U_partial agent managed to beat agents 3 and 4 in terms of success rate and efficiency so surely is better than project 2 but can we say it is optimal? Well, given we don't know the exact location of prey, it is expected to find it in more, number of step as we cannot always head towards an entity whose location is unknown.
I believe that agent U_partial is optimal as this is the best, we can do given the information we have. We know our utilities for all states are correct and we are using all information about the prey's location (survey and belief) to compute the weighted Utility based on the belief to computing the partial utilities, it is the best value we can produce. Another way would be to consider the max probability like in project 2 and calculate the utilities assuming the prey is there. But this is not better than considering the entire belief matrix when calculating. So, using equation 1 is optimal.

2. AGENT V_Partial

To build the V partial model, since there can be infinite U_partial values based on belief, we take training data states (including belief states) are used from what occur during simulations of the Upartial agent.
Let's change the code a bit to store the entity position, output, and the prey_belief values to a file.


```
        key_partial_U=find_U_parital(prey_belief,agent,predator)
        for i in key_partial_U.keys(): #change for V
            data_export.append([i, key_partial_U[i],prey_belief]) #change for V
```

We store this data to "partial_utilities.xlsx" file to train out data.

Once our dataset is ready, we start training our neural network.

INPUTS

We are using the following inputs when training our neural network:

1. Distance between agent and predator: Since the agent is trying to avoid getting caught by the predator, the predator being in proximity does affect the movement of the agent making it take longer to catch the prey.
2. The prey belief probabilities: The belief of the prey position is very important in calculating the Utility as it was adding weightage to the Utility of a given state. To train the model, we are using all 50 beliefs as input.
3. Biased term (i.e., 1)

Is Overfitting an issue here?

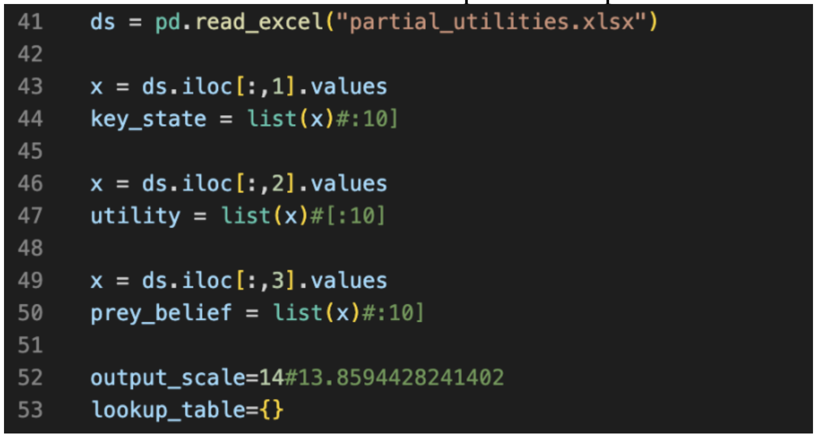
Overfitting is an issue here as we are only training the model using about 75000 input points but depending on the belief of prey, we can have infinite combination of Utilities just for a single graph. If we overfit the Model for the belief we did train it for, it will not perform well for the belief combination that was left out that might come up during agent simulation. Basically we don't want to overfit to input data as this time, the input data is not the only dataset we might run the Model on.

To solve for overfitting,

- we can train the model using more input points that will take more belief combination into consideration making the model more generalized.
- We can use smaller and simpler neural network architecture to prevent the model from having more weights. Less weights means the model will not be able to fit to the training dataset perfectly. We need to test out different architecture until we find the best one.
- We can stop training a little before the error converges to 0. For example, the loop ends when error is less than 0.1 instead of less than 0.0001.
- Use Regularization to prevent the model from fitting to the noise in the training data.

TRAINING THE MODEL

To train the model, we first read the training data we stored while implementing U_star agent. The code to read and store states and respective outputs is as follows:

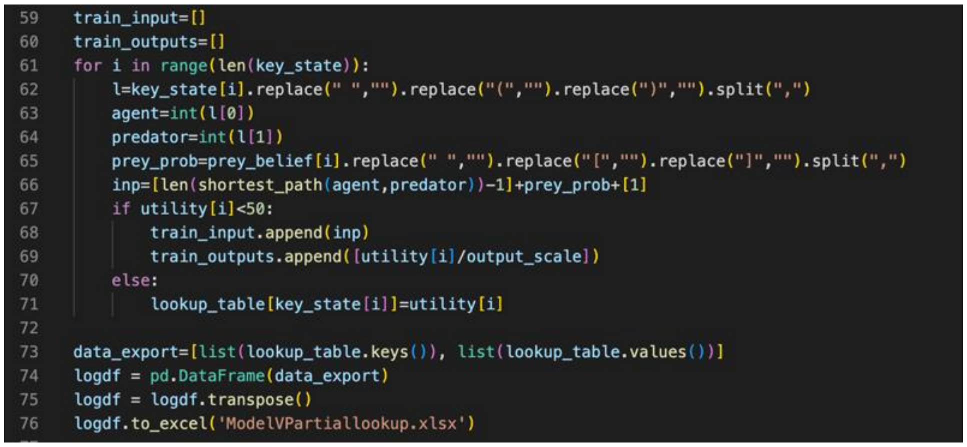


```
ds = pd.read_excel("partial_utilities.xlsx")

x = ds.iloc[:,1].values
key_state = list(x)#:10]

x = ds.iloc[:,2].values
utility = list(x)#[:10]

x = ds.iloc[:,3].values
prey_belief = list(x)#:10]

output_scale=14#13.8594428241402
lookup_table={}
```

Here, key_state stores all state values (Line 44) and utility stores all outputs values(Line54).
Two things to consider about the training inputs:

a. We are using sigmoid as an activation function as it gave the best accuracy. Since it generates output between 0 and 1, We need to scale down our input by dividing by the highest utility (14 rounding off). We can later scale up by the same factor to retrieve the Utility.
b. Since we have some infinite Utility for some cases, it was messing up the convergence of the Model. To solve these, we are storing these utilities in a separate smaller lookup table and not using them to train or be predicted by Model V. are only storing a small amount and still solving the problem of storing the large dataset.

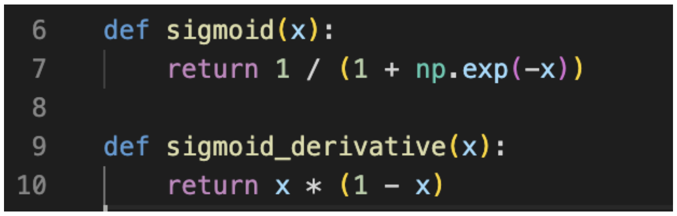
The following code then evaluate and generate the input and output datasets:



```
59    train_input=[]
60    train_outputs=[]
61    for i in range(len(key_state)):
62        l=key_state[i].replace(" ","").replace("(","").replace(")","").split(",")
63        agent=int(l[0])
64        predator=int(l[1])
65        prey_prob=prey_belief[i].replace(" ","").replace("[","").replace("]","").split(",")
66        inp=[len(shortest_path(agent,predator))-1]+prey_prob+[1]
67        if utility[i]<50:
68            train_input.append(inp)
69            train_outputs.append([utility[i]/output_scale])
70        else:
71            lookup_table[key_state[i]]=utility[i]
72
73    data_export=[list(lookup_table.keys()), list(lookup_table.values())]
74    logdf = pd.DataFrame(data_export)
75    logdf = logdf.transpose()
76    logdf.to_excel('ModelVPartiallookup.xlsx')
```

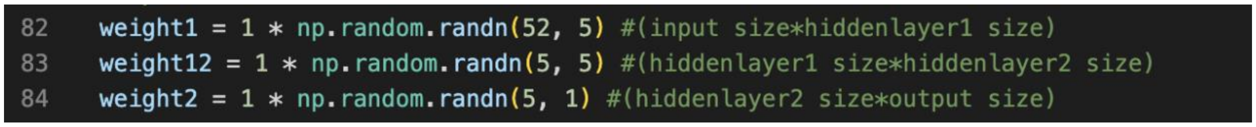
We extract both entity position and create inputs and scaled down outputs of all finite utilities(Line 62-69). Else, we store the finite values to a lookup table.

Before we start with training the model, we need to define sigmoid activation function and it's derivative as follows:



```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)
```

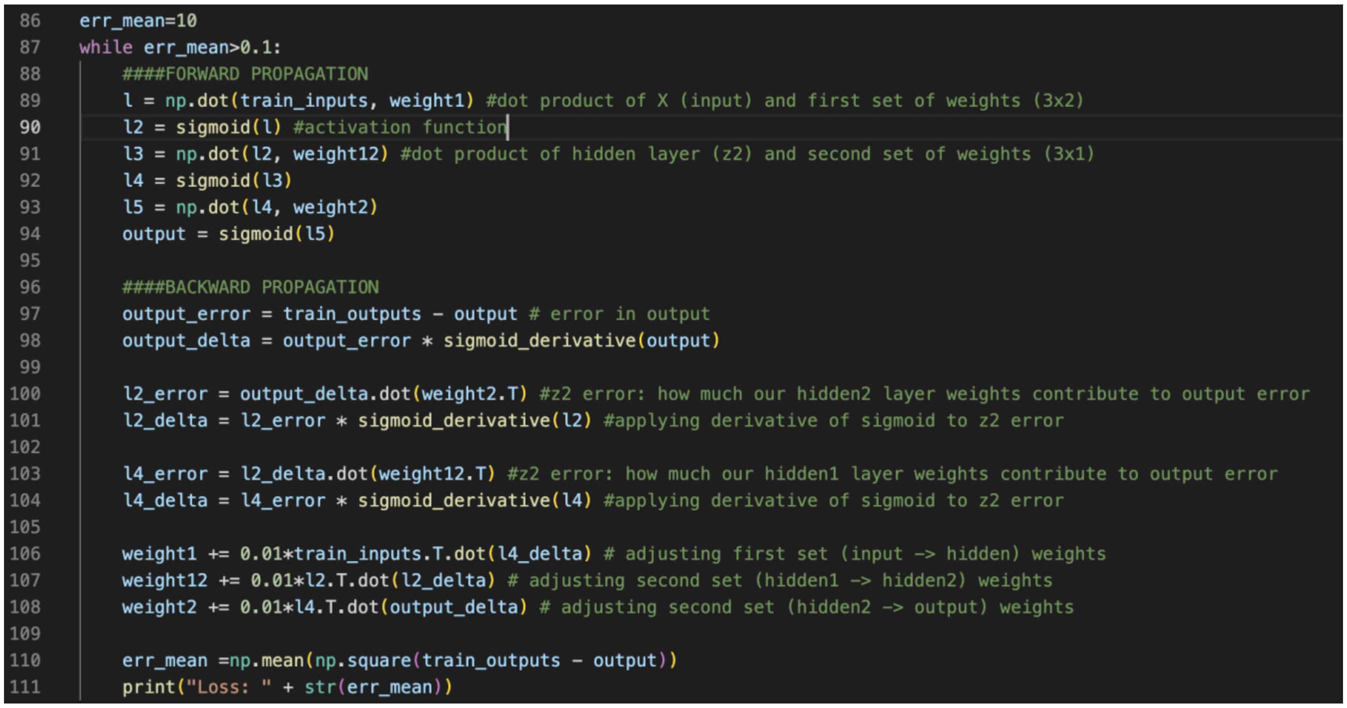
Now, we are making a neural network with the input layer having 52 nodes, 2 hidden layers each having 5 nodes and finally an output layer having 1 node. The reason we didn't make complex architecture even with 52 inputs is because we don't want to overfit the model. To initialize weights for this model, we use the following code.



```python
weight1 = 1 * np.random.randn(52, 5) #(input size*hiddenlayer1 size)
weight12 = 1 * np.random.randn(5, 5) #(hiddenlayer1 size*hiddenlayer2 size)
weight2 = 1 * np.random.randn(5, 1) #(hiddenlayer2 size*output size)
```

We just need to make sure we are taking the correct dimension.

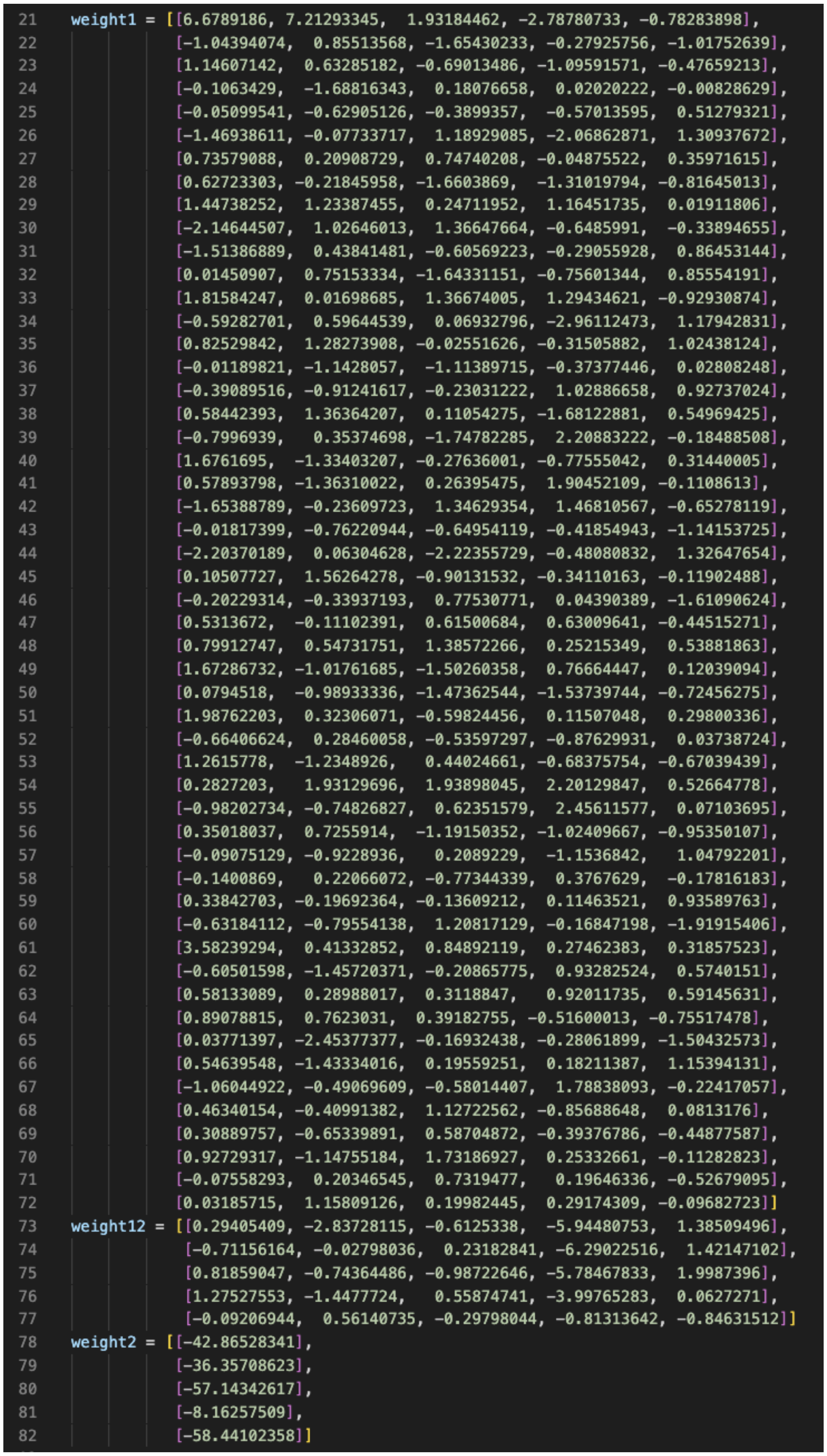
Once we have all the dependencies, we can train the model using the same structure as we used to train Model V. We have already taken care of changes in weight dimension and inputs.



```python
err_mean=10
while err_mean>0.1:
    ####FORWARD PROPAGATION
    l = np.dot(train_inputs, weight1) #dot product of X (input) and first set of weights (3x2)
    l2 = sigmoid(l) #activation function
    l3 = np.dot(l2, weight12) #dot product of hidden layer (z2) and second set of weights (3x1)
    l4 = sigmoid(l3)
    l5 = np.dot(l4, weight2)
    output = sigmoid(l5)

    ####BACKWARD PROPAGATION
    output_error = train_outputs - output # error in output
    output_delta = output_error * sigmoid_derivative(output)

    l2_error = output_delta.dot(weight2.T) #z2 error: how much our hidden2 layer weights contribute to output error
    l2_delta = l2_error * sigmoid_derivative(l2) #applying derivative of sigmoid to z2 error

    l4_error = l2_delta.dot(weight12.T) #z2 error: how much our hidden1 layer weights contribute to output error
    l4_delta = l4_error * sigmoid_derivative(l4) #applying derivative of sigmoid to z2 error

    weight1 += 0.01*train_inputs.T.dot(l4_delta) # adjusting first set (input -> hidden) weights
    weight12 += 0.01*l2.T.dot(l2_delta) # adjusting second set (hidden1 -> hidden2) weights
    weight2 += 0.01*l4.T.dot(output_delta) # adjusting second set (hidden2 -> output) weights

    err_mean =np.mean(np.square(train_outputs - output))
    print("Loss: " + str(err_mean))
```

Once we get the weights, we can use them as our V_partial model just like V Model.

The weights came as follows and we can implement the agent exactly like V model but using the below weights.



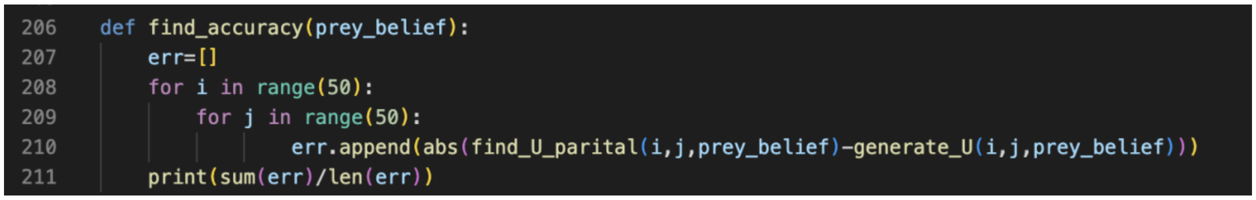
```
weight1 = [[6.6789186, 7.21293345,  1.93184462, -2.78780733, -0.78283898],
           [-1.04394074,  0.85513568, -1.65430233, -0.27925756, -1.01752639],
           [1.14607142,  0.63285182, -0.69013486, -1.09591571, -0.47659213],
           [-0.1063429,  -1.68816343,  0.18076658,  0.02020222, -0.00828629],
           [-0.05099541, -0.62905126, -0.3899357,  -0.57013595,  0.51279321],
           [-1.46938611, -0.07733717,  1.18929085, -2.06862871,  1.30937672],
           [0.73579088,  0.20908729,  0.74740208, -0.04875522,  0.35971615],
           [0.62723303, -0.21845958, -1.6603869,  -1.31019794, -0.81645013],
           [1.44738252,  1.23387455,  0.24711952,  1.16451735,  0.01911806],
           [-2.14644507,  1.02646013,  1.36647664, -0.6485991,  -0.33894655],
           [-1.51386889,  0.43841481, -0.60569223, -0.29055928,  0.86453144],
           [0.01450907,  0.75153334, -1.64331151, -0.75601344,  0.85554191],
           [1.81584247,  0.01698685,  1.36674005,  1.29434621, -0.92930874],
           [-0.59282701,  0.59644539,  0.06932796, -2.96112473,  1.17942831],
           [0.82529842,  1.28273908, -0.02551626, -0.31505882,  1.02438124],
           [-0.01189821, -1.1428057,  -1.11389715, -0.37377446,  0.02808248],
           [-0.39089516, -0.91241617, -0.23031222,  1.02886658,  0.92737024],
           [0.58442393,  1.36364207,  0.11054275, -1.68122881,  0.54969425],
           [-0.7996939,   0.35374698, -1.74782285,  2.20883222, -0.18488508],
           [1.6761695,  -1.33403207, -0.27636001, -0.77555042,  0.31440005],
           [0.57893798, -1.36310022,  0.26395475,  1.90452109, -0.1108613],
           [-1.65388789, -0.23609723,  1.34629354,  1.46810567, -0.65278119],
           [-0.01817399, -0.76220944, -0.64954119, -0.41854943, -1.14153725],
           [-2.20370189,  0.06304628, -2.22355729, -0.48080832,  1.32647654],
           [0.10507727,  1.56264278, -0.90131532, -0.34110163, -0.11902488],
           [-0.20229314, -0.33937193,  0.77530771,  0.04390389, -1.61090624],
           [0.5313672,  -0.11102391,  0.61500684,  0.63009641, -0.44515271],
           [0.79912747,  0.54731751,  1.38572266,  0.25215349,  0.53881863],
           [1.67286732, -1.01761685, -1.50260358,  0.76664447,  0.12039094],
           [0.0794518,  -0.98933336, -1.47362544, -1.53739744, -0.72456275],
           [1.98762203,  0.32306071, -0.59824456,  0.11507048,  0.29800336],
           [-0.66406624,  0.28460058, -0.53597297, -0.87629931,  0.03738724],
           [1.2615778,  -1.2348926,   0.44024661, -0.68375754, -0.67039439],
           [0.2827203,   1.93129696,  1.93898045,  2.20129847,  0.52664778],
           [-0.98202734, -0.74826827,  0.62351579,  2.45611577,  0.07103695],
           [0.35018037,  0.7255914,  -1.19150352, -1.02409667, -0.95350107],
           [-0.09075129, -0.9228936,   0.2089229,  -1.1536842,   1.04792201],
           [-0.1400869,   0.22066072, -0.77344339,  0.3767629,  -0.17816183],
           [0.33842703, -0.19692364, -0.13609212,  0.11463521,  0.93589763],
           [-0.63184112, -0.79554138,  1.20817129, -0.16847198, -1.91915406],
           [3.58239294,  0.41332852,  0.84892119,  0.27462383,  0.31857523],
           [-0.60501598, -1.45720371, -0.20865775,  0.93282524,  0.5740151],
           [0.58133089,  0.28988017,  0.3118847,   0.92011735,  0.59145631],
           [0.89078815,  0.7623031,  0.39182755, -0.51600013, -0.75517478],
           [0.03771397, -2.45377377, -0.16932438, -0.28061899, -1.50432573],
           [0.54639548, -1.43334016,  0.19559251,  0.18211387,  1.15394131],
           [-1.06044922, -0.49069609, -0.58014407,  1.78838093, -0.22417057],
           [0.46340154, -0.40991382,  1.12722562, -0.85688648,  0.0813176],
           [0.30889757, -0.65339891,  0.58704872, -0.39376786, -0.44877587],
           [0.92729317, -1.14755184,  1.73186927,  0.25332661, -0.11282823],
           [-0.07558293,  0.20346545,  0.7319477,   0.19646336, -0.52679095],
           [0.03185715,  1.15809126,  0.19982445,  0.29174309, -0.09682723]]
weight12 = [[0.29405409, -2.83728115, -0.6125338,  -5.94480753,  1.38509496],
            [-0.71156164, -0.02798036,  0.23182841, -6.29022516,  1.42147102],
            [0.81859047, -0.74364486, -0.98722646, -5.78467833,  1.9987396],
            [1.27527553, -1.4477724,   0.55874741, -3.99765283,  0.0627271],
            [-0.09206944,  0.56140735, -0.29798044, -0.81313642, -0.84631512]]
weight2 = [[-42.86528341],
           [-36.35708623],
           [-57.14342617],
           [-8.16257509],
           [-58.44102358]]
```

The code for implementing V_partial is AgentVPar.py and it simply forward propagates using the above weights like in V model implementation.

Note: not explaining the simulation code again to reduce redundancy of the report as it is same as Agent V

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

The accuracy of Model V should be pretty good given we were able to converge the loss to below 0.1.
We can further verify this accuracy by comparing Utility_partial calculated by the model with Utility_partial calculated by equation 1. To find the average of difference, we use the following code:



```
def find_accuracy(prey_belief):
    err=[]
    for i in range(50):
        for j in range(50):
                err.append(abs(find_U_parital(i,j,prey_belief)-generate_U(i,j,prey_belief)))
    print(sum(err)/len(err))
```

Where, find_U_partial() calculates the partial utility by computing equaltion1 and generate_U estimates the value via Model V_partial

The mean difference comes out to be 7.16541384. This means that that on average, the model V gives an error of around 7 which is not that good given we cannot train with all infinite possible partial Utility as input. The agent still shows good success rate as knowing the location of predator, we were able to avoid death states. But the efficiency falls off a bit because the agent is not always moving towards the prey.

Is V_partial more or less accurate than simply substituting V into the equation?

V_partial is less accurate than simply substituting V into the equation. Since it is not possible to design a Model that produces estimates Utility with 100% accuracy, calculating the Utility using Value iteration or Partial Utility using weighted summation should ideally be more accurate but we are using models to reduce space complexity.

Let’s assume not using a model gives no error in calculation and let errorV be the error in model V and errorVP be the error in model V_partial. Since summing V to estimates produces no additional error, Substituting V into the equation contribute to only errorV deviation. And estimating through V_partial Model produces errorVP error. As in our case, errorVP is greater than errorV, it is better to substitute V into equation if we want more accuracy.

DATA ANALYSIS:

Agent V_partial was simulated 3000 times on our graph with different initial states.

The success rate came out to be **99.66%** and the average number of moves for game to end is **35.6241.** So, the efficiency did decrease as the Model was not able to predict the U_partial Utilities with 100% accuracy.

The reason success rate is not affected despite there being some error in V model is because we are not using the Model to calculate infinite state utility. The agent still knows with 100% accuracy which states to avoid to not die. However, the efficiency average number of moves did increase a little compared to AgentU as the Utility was predicted with some error meaning the agent is not following optimal path. The efficiency is still pretty good compared to agent 1 and 2 so I would call this a success.

COMPARISON OF ALL THE AGENTS

AGENT	SUCCESS RATE	Average no. of moves to catch the prey
Agent U	100%	8.2583
Agent 1	87.2%	16.7091
Agent 2	99.46%	27.9106
Agent V	99.86%	9.5073
Agent U_partial	99.933%	23.594
Agent 3	89.36%	30.947
Agent 4	99.66%	40.1216
Agent V_partial	99.66%	35.6241

We ended up getting near perfect success rate for all agents because we always knew the location of the predator and knew the states with infinite Utility and avoided them.

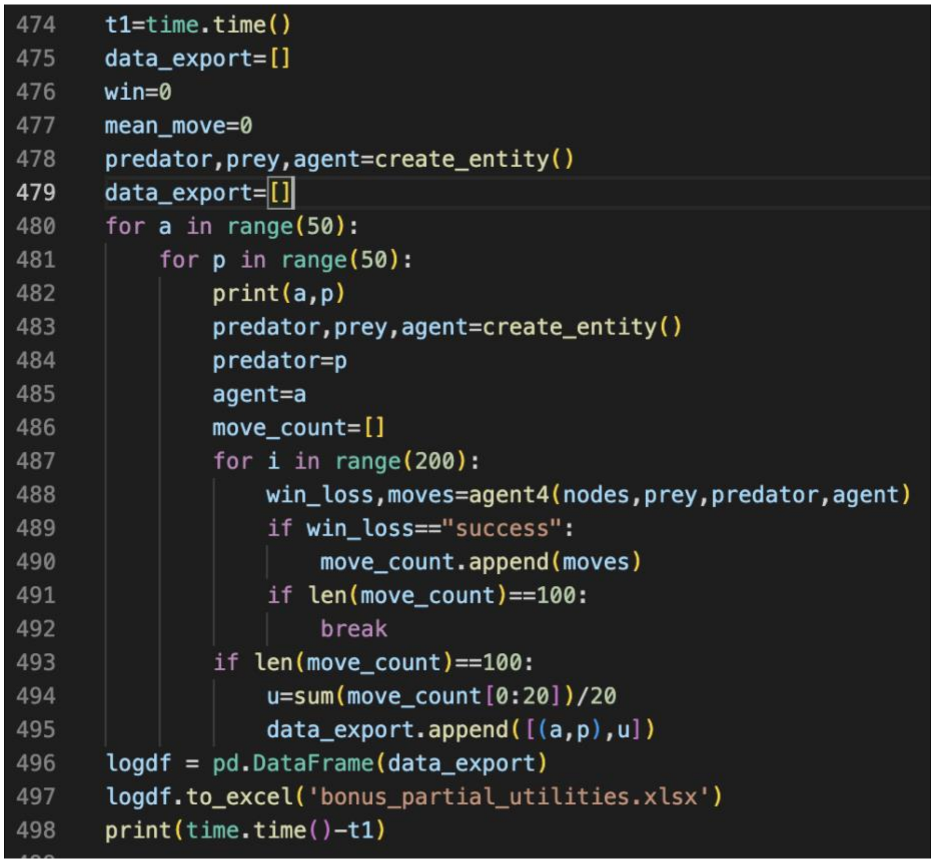
BONUS AGENT:

The value of U_partial that we get from equation 1 does not represent the actual Utility for a particular agent and predator position as it is based on the belief of the prey and not the actual location of the prey. Since we don’t know the actual location, mathematically, we did calculate optimal values. To find the actual Utility we need to simulate an agent starting from every state multiple time and take the average number of moves it takes to catch the prey.
But, to do this with simulation, we need an optimal agent to know the values are correct. We have our agent 4 whose success rate is above 99% but it still doesn’t take the optimal path. To solve for this non-optimal behavior, we do the following:

- To solve for a not 100% success rate, we only consider the values if the agent wins. This ensures that we are collecting data when the agent can succeed (i.e good agent)
- To solve for the agent being not efficient, we only calculate the average of steps needed for top 20% performance. This was, we only average out the no. of steps when the non-optimal agent was moving optimally.

This should solve the problem of using a somewhat non-optimal agent 4 for simulation.

To compute the utilities like this, we use the below code:



```
474  t1=time.time()
475  data_export=[]
476  win=0
477  mean_move=0
478  predator,prey,agent=create_entity()
479  data_export=[]
480  for a in range(50):
481      for p in range(50):
482          print(a,p)
483          predator,prey,agent=create_entity()
484          predator=p
485          agent=a
486          move_count=[]
487          for i in range(200):
488              win_loss,moves=agent4(nodes,prey,predator,agent)
489              if win_loss=="success":
490                  move_count.append(moves)
491              if len(move_count)==100:
492                  break
493          if len(move_count)==100:
494              u=sum(move_count[0:20])/20
495              data_export.append([(a,p),u])
496  logdf = pd.DataFrame(data_export)
497  logdf.to_excel('bonus_partial_utilities.xlsx')
498  print(time.time()-t1)
```

The code runs for all 2500 combination of agent and predator (Line 480-481) and simulates the agent 4 200 times (line 487-492). The loop ends when first 100 success occurs and the top 20 results of those 100 successful runs gets averaged for Utility. We store this data on an external file(bonus_partial_utilities.xlsx) to train our model.

Implementation of Bonus agent.

The model can be trained using the same code as Agent V by taking inputs as agent position, predator position, distance between agent and predator and biased term.