Better, Smarter, Faster

Author: Patrick Liang – pzl4

Design:

Environment:

Graph:

Initially planning for the Graph, I experimented with many data structures. Eventually, I realized that an ArrayList of ArrayLists is the optimal data structure to use because lookup is O(1) in time complexity and adding a new cell is also O(1) in time complexity. Even though a HashMap offers the same time complexity, there is no way to access the info of the starting cell, which is essential for this project. In addition, I decided to create a Node class for the Graph so it could store cell value and previous cell(used in BFS). Moreover, because it is an object, I decided to override the built-in equals() and hashCode() functions for this class. This is because then I could compare two different objects and see if the cell is the same. In order to do this, I imported a helper class built by Apache Commons.

Implementing the actual buildGraph() function was the core of the Graph class. I first created a function buildSkeletonGraph() which builds the initial bidirectional circular graph connecting all Nodes. I then had to build the addRandomEdges() method, which adds edges randomly which adhere to the constraints of a cell having a degree of three or less and a maximum distance of five for each connection.

I have also implemented the Serializable class in order to serialize a maze to use for V.

Agent:

The constructor of this class generates a position uniformly at random for cells 0-49(size of graph).

Prey:

The constructor for this class generates a position on the graph that is not the Agent's cell. In addition, there is a method that chooses uniformly at random what the next move will be.

Predator:

The constructor for this class generates a position on the graph that is not the Agent's cell. There is a method that chooses uniformly at random what the next move will be. There is a BFS method that searches for the Agent from the perspective of the Predator. BFS is the only valid search method because it finds the shortest path, and a heuristic could not prune out search paths. In addition, it implements a getPath() method to return the path of Predator to Agent. It is used to determine the length of the path.

Result:

Two constructors are used to return data points necessary to determine each Agent's efficacy.

State:

The constructor for this class initializes agent, Prey, and Predator locations. The helper methods are used to retrieve agent, Prey, or Predator locations.

Similarly to the static Node class, I override the equals(), hashCode(), and toString() methods. This is because then I could compare two different objects and see if the cell is the same. I imported a helper class built by Apache Commons to accomplish this.

Odd Agents Base Logic:

The odd Agents all follow the same pattern. After getting each neighboring cell of the Agent and each cell's distance to Prey and Predator, you compare each cell and add it to a priority list based on the rules given in the writeup. I decided to use a HashMap to store cells that belong to a qualified level. This way, when deciding which cell to travel towards, I will iterate through the HashMap starting from level one, and if there exists a cell that belongs to level one, I will choose that cell. If there are multiple cells at the first level that contains cells, I will choose uniformly at random which cell to travel towards. Afterward, Agent will move, the Prey will move at random. The Predator will only greedily move toward the Agent 60% of the time. The other 40% of the time, it will move uniformly at random toward one of its neighbors.

Even Agents Base Logic:

The even Agents follow a similar pattern to the Odd Agents. However, instead of following the writeup's steps to determine each cell's viability, I created my utility function to determine the value of each cell. This utility function, called bestCell(), first evaluates the average distance from each Agent's neighbors to the Preys neighbors. For example, if Agent neighbors are 1, 2, and 3 and Prey neighbors, including itself, are 9, 10, 11, and 12, then the distance of 1 to 9, 1 to 10, 1 to 11, and 1 to 12 will all be averaged out and stored. The function does this because I appreciate the closest distance to the potential future of the Prey more than the distance between the Agent and where the Prey is currently. I also find the Predator distance between it and each Agent neighbor in the same calculation. I then get the average distance for some cells and subtract it from the distance of the Predator to the Agent. This calculation is my base utility.

I iterate through Prey distances to the Agent, including itself, and Predator distances to the Agent. When the shortest distance is found to the Prey, I update the utility value for that cell by adding 75*(0.5/(number of Prey neighbors, including itself)). When the shortest distance is found to the Predator, I update the utility value for that cell by adding 100*(-0.5/(number of Predator neighbors, including itself)). The values I use are arbitrarily found through

experimentation. However, I did choose the shortest distance to the Predator to be weighted more heavily because I value survivability over attempting to win greedily.

After all utility values are calculated, I choose the utility with the greatest value when it is greater than zero. If the greatest utility value is less than or equal to zero, I opt to move away from the Predator.

Probability Distribution Logic:

To keep a valid belief vector, there exist constraints that need to be satisfied. The first is that the entire belief vector should add up to 1.0. Secondly, when a cell has been surveyed not to contain the Predator/Prey, the other belief updates must be updated in accordance with Bayes' theorem.

Bayes' theorem:

$P(B|A) = \frac{(P(A|B)*P(B))}{P(A)}$ -> $\frac{(1*P(B))}{P(A)}$; P(B) is $1/N$, N is number of ties; the length of the shortest path dictates ties; P(A) is the calculated chance of the current node. Simplified, it becomes $\frac{1}{(1-(P(B))}$.

After Bayes' theorem gets applied to the belief vector, normalization is required to bring down the scale of each node to lie between 0 and 1. The normalizing constant, in this case, would be the sum of the belief vector immediately after Bayes' rule is applied.

Transition Probability Logic:

To find the probability of Prey/Predator after moving, I used a Markov Chain to update the belief vector.

The Prey transition matrix is filled with either 0 or the random probability that the Prey would move into the cell. For example, if a Prey has three neighbors, the row in that transition matrix would have ¼ for each cell that qualifies as a move for the Prey.

The Predator has two transition matrices. The first one is nearly identical to the Prey transition matrix. Instead of considering itself as a move, it only considers its neighbors.

The second Predator transition matrix is determined by the shortest path to the Agent's current position. The denominator for each cell is the count of shortest paths from its current position. For example, if two cells are tied with the shortest path, then the probability would be ½ for each cell.

The Prey belief vector would be updated using matrix multiplication with the Prey transition matrix.

The Predator belief vector would first be cloned into two separate vectors. The first would be updated using matrix multiplication with the Predator random transition matrix. The second one would be updated using matrix multiplication with the Predator greedy transition matrix. The first one would then be multiplied by 0.6 because there is a 60% chance that the

Predator would move greedily. In the end, add clone1 and clone2 to get the final Predator belief vector. The first one would then be multiplied by 0.4 because there is a 40% chance that the Predator moves uniformly at random. In the end, add the two vectors for the final Predator belief vector.

This completes the Markov Chain.

Complete Information:

Agent One:

There is not much to add to this Agent, as it follows the Odd Agents' basic logic. It starts off knowing both Prey and Predator location.

Agent Two:

Just like Agent One, this follows the Even Agents' basic logic. It starts off knowing both Prey and Predator location.

Helper Methods:

searchPred(): Traditional BFS to search for Predator distance based on agent location

searchPrey(): Traditional BFS to search for Prey distance based on agent location

Even though they are the same code, I opted to make two different methods, so it is easier to know what each BFS is used for when reading the code.

Partial Prey:

Agent Three:

It starts only knowing where the Predator is. In addition to the basic logic, it is necessary to update beliefs and survey the location of the Prey. In order to get the best survey, I survey the cell with the highest probability of containing the Prey and choose uniformly at random from the selected cells if there are ties.

To keep a valid belief vector, there exist constraints that need to be satisfied. The first is that the entire belief vector should add up to 1.0. Secondly, when a cell has been surveyed not to contain the Prey, update the belief vector accordingly, then normalize. Lastly, a Markov Chain is used to apply the probability for the next time step accurately.

Agent Four:

Same as Agent Three, except it uses the bestCell() function to determine the utility of cells instead of the provided rules.

Agent Three and Four performs all these operations through the following helper methods:

updateProbability():

Applies Bayes' theorem.

maxIndex():

Returns the index with the greatest belief.

maxBelief():

Returns greatest belief value from beliefs vector.

initialBelief():

Initializes belief vector with $\frac{1}{49}$ for every cell except the Agent's.

beliefSum():

Returns the sum of the belief vector.

randomSurvey():

Chooses the most probabilistic cell and adds it to an ArrayList. Then, chooses uniformly at random between tied most probabilistic cell.

matmul():

Performs matrix multiplication using dot product.

dotProduct():

Dot product with belief vector as a row, not a column.

normalize():

Normalizes the belief vector.

U*:

Similarly to the even agents, I use a utility function to determine the best cell for the Agent to move into. However, now I am implementing Value Iteration to form an optimal policy for a given maze and assigning appropriate utility values to each possible state.

To start, a single state is a unique set of locations of agent, Prey, and Predator. Because there are 50 different positions for each character, it results in 125,000 distinct states in any environment. Each of these states holds a utility value. I decided that the best implementation to hold these utilities would be to use a HashMap<State, Double> ustar. I use a State object to replace tuples for lookup. Additionally, it makes things easier to understand when saying State (*x, y, z)* has a utility of *u*, *u* being the expected number of steps it would take from agent to Prey.

I initialize ustar with 5 distinct states:

1) If the agent and Predator occupy the same cell without the Prey, the utility is positive infinity

2) Else if the agent and Prey occupy the same cell, the utility is 0.0
3) Else if the agent is a distance of 1 away from the Predator, the utility is positive infinity
4) Else if the agent is a distance of 1 away from the Prey, the utility is 1
5) And finally, all other scenarios start off with a utility of 0.0

*U**(s) is the expected number of steps the agent at s.agent is expected to reach s.Prey. In relation to *U** of other states, *U**(s) should hold similar utilities for similar state positions, e.g., State (1, 7, 8) and State (1, 7, 7) should have similar utilities because neither position falls into a base case and because the change in Predator cell doesn't have too big a bearing on the state space. However, if the States involve the base cases, then the *U** utilities should vary drastically.

Designing the algorithm to find the Optimal Policy was difficult. Because I had already discerned states that should be initialized, I had an easier time going about the logic of the algorithm.

To start, I modelled my algorithm after the Bellman's Equation:

$$U^*(s) = min_{a \in A(s)}[r_{s,a} + \sum_{s'} p^a_{s,s'} U^*\left(s'\right)]$$

Because I am implementing Value Iteration, I modelled my algorithm after the tweaked Bellman's Equation

$$U^*{}_{k+1}(s) = min_{a \in A(s)}[r_{s,a} + \sum_{s'} p^a_{s,s'} U_k{}^*\left(s'\right)]$$

So for every State, I would choose the min action, for all actions from State *s*. An action being an Agent's movement. The max action is denoted by the reward of State *s* choosing action *a* plus the sum of the probabilities for each possible state resulting from action *a* times the utility value of the previous iteration for each state. What this equation suggests is that eventually, $U^*{}_k$ will converge with U^* as k approaches infinity. This is how the optimal policy will be computed.

I implemented this equation with the following methods.

initU(): while generating all 125,000 unique states, add initial utility values based on the 5 distinct states I mentioned before.

updateStates(): For every possible 125,000 unique states, I either add a utility value based on the 5 distinct states I previously defined or I add a utility value based on the updateValues() method.

updateValues(): For every state, there amounts to at most 3 different actions that can be taken by the agent. For each action there are at most 12 states that results from each action. In order to find the value of each action, I first have to generate every single state possible. I use the getNextStates() method to return an ArrayList of states which result

from the state I am attempting to give a utility value for. Before I iterate through all generated states, I first determine how many cells is considered optimal for the Predator. I then iterate through all states generated. For each iteration, I check to see if the action has changed, if so I take the min value so far, e.g., *a1* has a value of 3 and *a2* has a value of 2, only the value of 2 will remain to be compared with the value of *a3*. Each actions value is the sum of U^*_k times the probability that the current generated state will occur. The probability goes like this if the current state is a shortest path for the Predator:

$$U_k^* * \frac{1.0}{numberofpreystates} * \left(\frac{0.6}{numberofshortestpaths} + \frac{0.4}{numberofpredatorstates}\right)$$

Or like this if the current state is not a shortest path for the Predator:

$$U_k^* * \frac{1.0}{numberofpreystates} * \left(\frac{0.4}{numberofpredatorstates}\right)$$

After all actions are accounted for and the min action is established, I add the reward of 1.0 to finalize the utility value for a given state. The reward value should be 1.0 because each action results in the agent moving 1.0 step.

policyIter(): I first initialize the HashMap<State, Double> ustar using initU(). I then define the convergence value I would like to use, the convergence value being how close U^*_k and U^*will get to. I opt to use 0.001. I then use an infinite while loop to create my policies until convergence. Each iteration of the while loop creates a new policy using updateStates(). I calculate the max difference between close U^*_k and $U^*$and if the max difference is smaller than the convergence value I previously defined, I have reached an optimal policy.

Helper Methods:

getNextStates(): returns an ArrayList of all possible states given a State S.

getNextAgentStates(): returns a List of all possible Agent actions/states

getNextPreyStates(): returns a List of all possible Prey states

getNextPredatorStates(): returns a List of all possible Predator states.

copyMap(): a deep clone of a HashMap to prevent data contamination

differences(): returns an ArrayList of differences between U^*_k and U^*

oneHotEncoder(): serializes the state space, utility values, and maze to use for training in V

By using policyIter() to constantly update a new U^*_k, an optimal policy can be found which allows for the agent to both minimize steps and maximize efficiency. Additionally, because there are states where the Agent is considered “trapped”, where the Agent is a distance of 1 away from the Predator, I added a logical component that allows for the Agent to attempt to

run away from by going to the cell which is the farthest away from the Predator. This allows the Agent to increase their survival odds in those scenarios by 40% due to Predator being distracted.

The state with the largest possible finite value of U^* I discovered in my trials is about 24.45. The state is as follows:

Agent: 38

Prey: 45

Predator: 45

When run with this state, the following occurs:

Agent: 38; Prey: 45; Predator: 45

Iter: 1	Iter: 11	Iter: 21
38, 45, 45	38, 46, 39	46, 0, 39
Iter: 2	Iter: 12	Iter: 22
37, 0, 46	37, 45, 38	47, 49, 40
Iter: 3	Iter: 13	
38, 0, 41	36, 45, 37	
Iter: 4	Iter: 14	
39, 45, 40	35, 0, 38	
Iter: 5	Iter: 15	
38, 45, 43	36, 0, 35	
Iter: 6	Iter: 16	
39, 44, 40	37, 1, 36	
Iter: 7	Iter: 17	
38, 45, 43	38, 5, 35	
Iter: 8	Iter: 18	
39, 46, 40	39, 1, 38	
Iter: 9	Iter: 19	
38, 47, 43	40, 1, 39	
Iter: 10	Iter: 20	
39, 46, 40	41, 1, 38	

As expected of the optimal policy, the expected number of steps is remarkably close to reality in this scenario, where there were 22 rounds of the Agent attempting to both outsmart the predator and capture the prey, compared to the expected 24-25 steps. The reason why that specific state has the highest finite value is because the Predator is blocking the Agent from moving towards the Prey. The Agent has to be patient for the Predator to mess up to have a fighting chance at capturing the Prey.

I then simulated the Agent using U^* values.

Data:

	Survival Rate(%)	Predator Catches Agent(%)	Agent Catches Prey(%)	Agent Runs into Predator(%)	Average Steps
Agent One	82.73	17.27	53.33	0.00	15.94
Agent Two	99.13	0.87	63.00	0.00	21.107
U* Complete	100.00	0	69.47	0.00	10.55

Notes:

Survival Rate: how often the Agent catches the Prey or Prey runs into Agent

Predator Catches Agent(%): On average, out of 3000 simulations, the Predator catches the Agent

Agent Catches the Prey(%): On average, out of 3000 simulations, the Prey runs into Agent

Agent Runs into Predator(%): On average, out of 3000 simulations, the Agent runs into Predator

Average Steps: On average, out of 3000 simulations, the amount of steps the Agent takes

Analysis:

What this data tells us is that U^* completely dominates the Agents One and Two. In every category U^* performs remarkably better. Not only does the Predator not catch the Agent in U^*, but it even catches the Prey more efficiently as well at a remarkable 69.47%.

It seems that the Agent Two that I had designed is near optimal, with a Survival Rate of 99.13%. However, upon closer inspection, Agent Two, on average, takes twice as long to find the Prey and doesn't achieve the same efficiency in catch the Prey as U^*does.

It is clear that $U^*$is optimal when comparing both success and efficiency.

Moreover, when comparing the Agent in U^* and the Agents in One and Two, the Agent in U^* performs actions that the Agents in One and Two do not consider.

In addition when comparing the difference in how U^*, Agent One and Agent Two, make decisions, U^* tends to make moves that look towards the "future" whereas Agent One makes decisions that look to decrease distance towards the prey for an immediate reward. Moreover, the way my Agent Two is designed, it tends to run away at the scent of Predator if it deems that the Prey is not worth going for. Because of this, it tends to greedily go towards the Prey until the Predator is too great a concern. So in general my Agent Two would make decisions similar to U^* except when the Prey would be too close to not attempt to go towards. While, U^* would go towards the Prey in a "smarter" fashion which would involve going towards more probabilistic states for winning.

U* Partial:

In order to optimize a U^* but with partial states, I must use the same optimal policy I created for the same maze. However, because there are infinite possible belief states, an optimal utility is hard to solve for. So instead I estimate the utility of a given state using the following equation:

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

I implemented the following methods for the above equation:

updateValuesPartial(): similar to updateValues() except now there are 50 possible Prey states, which results in 3 Agent actions * (max)3 Predator states * 50 Prey states. Additionally, each actions value is now just a sum of the belief of the Prey at State *s* times the $U^*\left(s_{prey}\right)$.

updateStatesPartial(): does the exact same thing as updateStates() does, except now includes 50 Prey states

Helper Methods:

getNextPartialStates(): gets next states except now there are 50 Prey states

Data:

	Survival Rate(%)	Predator Catches Agent(%)	Agent Catches Prey(%)	Agent Runs into Predator(%)	Average Steps	Prey Survey(%)
Agent Three	84.63	15.37	49.87	0.00	30.96	6.65
Agent Four	97.57	2.40	51.17	0.00	37.24	4.70
U* Partial	99.93	0.067	68.67	0.00	23.58	17.84

Notes:

Survival Rate: how often the Agent catches the Prey or Prey runs into Agent

Predator Catches Agent(%): On average, out of 3000 simulations, the Predator catches the Agent

Agent Catches the Prey(%): On average, out of 3000 simulations, the Prey runs into Agent

Agent Runs into Predator(%): On average, out of 3000 simulations, the Agent runs into Predator

Average Steps: On average, out of 3000 simulations, the amount of steps the Agent takes

Prey Survey(%): How often the Prey survey would be correct

Analysis:

Looking at the data, it is clear that this estimate is optimal as its performance is nearly on par with Complete Information setting. It is lacking only in the Predator catches Agent category, where the predator can catch the Agent two times out of 3000 simulations. The two Moreover, $U_{partial}$ increases the survey% by a whopping 11-13 percentage points.

Agent Four seems to be very close to optimal, just like its predecessor Agent Two. In the same way, upon closer inspection, Agent Four is far from optimal. Although the survival rate is nearly as high, 97.57%, the efficiency of catching the Prey intentionally is only 51.17%. Additionally, two important statistics, average steps, and Prey survey(%) is abysmal in comparison. Average steps for Agent Four is about a 57% increase in steps taken which adds to the argument that Agent Four is not optimal. The closing case is the Prey Survey(%). $U_{partial}$ has a 17.84% success in surveying the Prey correctly. In comparison to Agent Three and Agent Four which have a 6.65% and 4.70% success rate, respectively.

In the end, it is clear that $U_{partial}$ is near optimal, if not optimal, with a few hiccups along the way.

V:

Although U^* is an optimal policy, the drawback is how much space it takes to keep every utility value. In order to mitigate this drawback, I implemented a model V, which takes the optimal policy from $U^*$and train it so that it can actively predict utility values. This way, there is no need to store 125,000 utility values.

Neural Network:

To start, there are only two relevant features for the model V. The two are 1) the state space and 2) the utility values. Originally, in U^*, I use HashMap<State, Double>. However, I will input a one-hot vector for the state space and a double[][] for the utility values, 125,000 rows and 1 column. This is because a neural net will believe that the integer inputs are categorical variables with ordinal relationships. However, for our purposes, the state space has no relationship with each other. Therefore, I use a one-hot vector to use as an input for my state space.

The one-hot vector is a 2D array of 125,000 by 150. Each row of the 2D array is a different state. Instead of a typical State of (0, 0, 49), it will now be [1, 0, 0, …., 1, 0, 0, ……., 1], where each '1' signifies the position of either an Agent, Prey, or Predator.

When the input is for $V_{partial}$, the only thing I need to change is the input. Instead of an input of position for prey, it is instead a belief vector filled with 1/49 where prey and agent do not occupy the same cell.

For the utility values, I have to replace Infinity because tanh() produces infinity when a number is that large. I replace the Infinity values with the largest finite value of the optimal policy + 10 for extra insurance.

Now, onto the meat of V. I designed V to be a supervised regression neural network. This is a supervised regression neural network because it uses linear regression to understand the relationship between independent and dependent variables, aka the state space relationship with the utility values.

As I mentioned, the inputs for this neural network is a one-hot vector of states and a double[][] of utility values, let's call them states and values respectively.

The output for the neural network is a List<Double> or Matrix of size 1 x 1, which is just the utility value we are attempting to model.

First, when designing my neural network, I had to decide 1) how am I going to implement my hidden layers 2) what loss functions am I going to use and 3) what data structure am I going to use to perform matrix operations.

Regarding 3), I realized that Java doesn't have any built in matrix libraries and that I would need to implement my own Matrix library. Fortunately, I am able to use a third-party Matrix library which is sourced at the end of this paper. However, even though I was using a third-party matrix library, I still had to implement additional functions.

Matrix Library:

Although the Matrix library supplied me with most of the operations I would need, there were some specifications that were in order. I had to update various params to match my own needs. I then replaced their sigmoid functions with tanh() and dtanh(). These are the activation functions I decided to use because they are easy to implement(Java has a built in tanh() function and dtanh() is not too much harder). Moreover, I had to implement a rowAdd() function that would be able to add matrices by rows for the bias vector. I had to implement a divide() function for the loss function. I had to implement a rowMultiply() function for to use in backwardPropagation(). Besides these functions, the rest of the Java class is the rooted from the source I referenced.

Loss Functions:

For 2), I landed upon using Mean Squared Error(MSE) to calculate the error of my neural network. The reason I chose MSE is because it is easy to calculate. My implementation is a doctored version of an online implementation that I sourced below.

Because I am using gradient descent in my linear regression model, I have to also calculate a gradient, which is just the derivative of the loss function, in this case MSE.

Using the chain rule for MSE results in the following formula:

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

Caption: E is error, n is size of Y, Y is original output

and Y* is predicted output

Hidden Layers:

Finally, regarding 1), I created a HiddenLayer class contains methods that maintain input, output, weights, bias, activation function, forward propagation, and backwards propagation. The class variables all have the data type Matrix and are all used to reduce error.

These are the following methods used to implement a hidden layer:

HiddenLayer(): This is a constructor that takes in input size for the hidden layer, output size for the hidden layer, and if the hidden layer will have an activation function. Additionally, the constructor will set the input and output of the HiddenLayer object to null, initialize the weights matrix of size inputSize x outputSize with random number centered around -0.5 and 0.5. The bias matrix will be size 1 x outputSize. The bias matrix is situated as such because each column represents an artificial neuron added to each row of the output. Finally, the constructor initializes whether or not an activation function will be applied to the hidden layer

forwardPropagation(): For forward propagation the math is relatively simple. The following formula encapsulates what needs to be done:

$$out_j = \sigma(\underline{w}(j) * \underline{out})$$

To break it down:

out_j represents the output for node j

σ represents the activation function

$\underline{w}(j)$ represents the vector of weights on the input to node out_j

When applied to a multiple layered neural network, each weight and output changes depending on the layer. Additionally, because not every layer should have an activation function, the formula above is only a general formula that I use for my neural network. So when I'm implementing forward propagation I first store the input. I then multiply the input with the respective Matrix of weights using Matrix library function to store in the objects output. If the current hidden layer requires an activation function, I then apply tanh() to the output and return said output.

backwardPropagation(): For backward propagation I modelled my implementation after the following algorithm:

The Backpropagation Algorithm: The derivatives of Loss with respect to any of the weights can be computed, layer by layer, as follows:

- At $t = K$, compute

$$\Delta_j^K = \frac{\partial L}{\partial \text{out}_j^K}, \tag{23}$$

  for each output node j, based on the specific loss function.

- For $t < K$ (in decreasing order), compute

$$\Delta_j^t = \frac{\partial L}{\partial \text{out}_j^t} = \sum_k \Delta_k^{t+1} \sigma' \left(\underline{w}^t(k) . \underline{\text{out}}^t \right) w_{j,k}^t. \tag{24}$$

- Any weight can then be updated according to

$$\left(\text{new } w_{i,j}^t\right) = w_{i,j}^t - \alpha \frac{\partial L}{\partial w_{i,j}^t} = w_{i,j}^t - \alpha \Delta_j^{t+1} \sigma' \left(\underline{w}^t(j) . \underline{\text{out}}^t \right) \text{out}_i^t. \tag{25}$$

To fully implement this equation, it involves a lot of matrix multiplication. To start, I calculate the error using either the gradient of the Loss or the previous layers calculated error, lets call it the inputError, times the transposed matrix of the associated weights matrix. I then calculate the error of the weights matrix by multiply the transposed matrix of the associated input matrix times the inputError. Now, to finally update the weights matrix, I subtract the product of the calculated weights error with the learning rate. In addition, I have to update the bias vector which I do by subtracting the product of inputError times the learning rate. After these series of matrix multiplications which update both weights and bias, I have to return a Matrix. If the layer has an activation function associated with it, then I take the input associated with the current HiddenLayer object and apply dtanh(). Afterwards, I matrix multiply it with the calculated error. I then return the Matrix. However, if there is no activation function associated with the current HiddenLayer object, then I simply return the calculated error. By going through every layer, I successfully propagate the error for each neuron by using gradient descent.

Network:

This is the class which ties it all together. In this class I use a constructor to initiate an ArrayList of HiddenLayer objects so I can add as many hidden layers as I want. I have an add() function that exists for this purpose. Additionally, I implement a predict function which takes in a state vector and outputs what the neural network is trained to output. I use forward propagation for this part.

I also implement a fit() function which does all the training necessary for the neural network. It takes in the state one-hot vector and the values vector. It also has parameters for how many epochs I want to do and the learning rate that I choose.

The fit() function works like this:

I iterate through the entire state space and use forward propagation to output a predicted value. I calculate the error using MSE and calculate the gradient to use in back propagation. I then do back propagation in reverse order of layers to update weights and bias for each respective layer. I continue this process for 100 epochs which ends my trained model.

Training:

To start training my model, I had to decide the amount of hidden layers and the input output sizes for each one. I ended up using two hidden layers and an output layer, the output layer doesn't apply an activation function. Because the one-hot vector has 150 columns, its only logical that the hidden layers have an input size of 150 and an output size of 150, excluding the output layer. The output layer has an input size of 150 and an output size of 1.

I also decided to use 100 epochs because of the computational taxation of performing this neural network. Even though Java is much faster than Python when comparing performance for computations, Java is infinitely slower than python when matrix operations are involved. Because Java doesn't have an optimized matrix library, computations take an inordinate amount of time, with run time for 100 epochs at best 10 hours. This is why I sacrifice my model's performance for run time.

I chose a learning rate of 0.0001 because I learned that a good starting point is 0.1 and to continue by a factor of 1/10. Through trial and error, I ended up with 0.0001 because it seemed to minimize the loss.

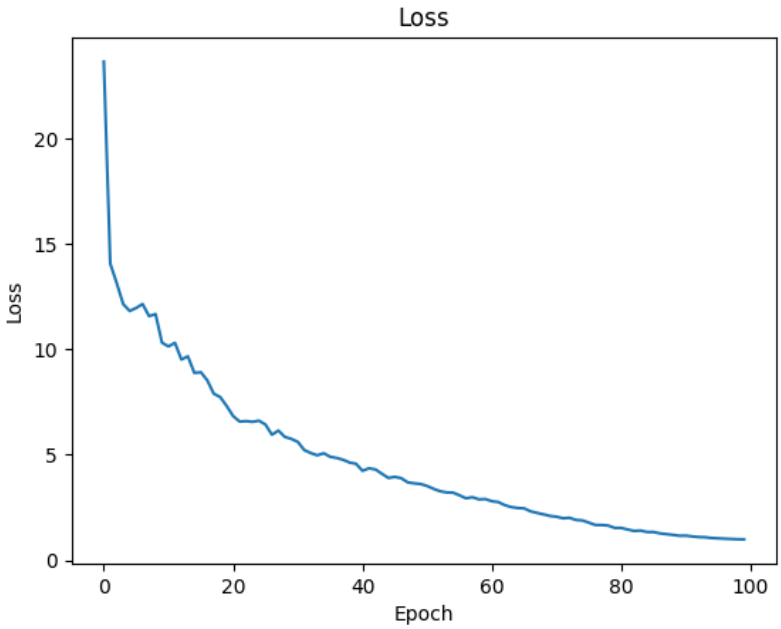
For $V_{partial}$, I choose a learning rate of 0.00005 because it ends up training my $V_{partial}$ with less error.

Results:

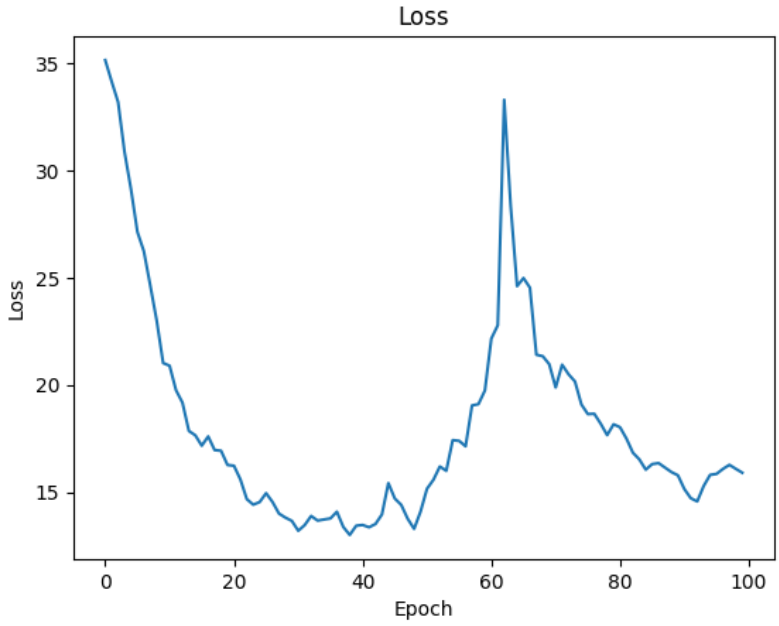
After training both V and $V_{partial}$ I get the following results:

	Survival Rate(%)	**Predator Catches Agent(%)**	**Agent Catches Prey(%)**	**Agent Runs into Predator(%)**	**Average Steps**	**Prey Survey(%)**
V Complete	99.93	0.067	62.40	0.00	20.23	N/A
V Partial	99.93	0.067	58.27	0.00	49.53	11.81

Graphs:



Caption: Loss of V



Caption: Loss of $V_{partial}$

Results Analysis:

To start, even though the loss of $V_{partial}$ was less than optimal, the loss of V is very smooth and headed to near 0 if I ran more epochs. This suggests that my model does work and suggests that my training data for $V_{partial}$ might not have been the best, although the loss may decrease further with more epochs, however, because I did not have enough time to run more than 100 epochs, I cannot definitively say so.

Regarding the results, V is near optimal, with a similar Survival Rate. However, because I only used 100 epochs, the loss only ever came down to under 1. Moreover, V is less efficient than U^* because it takes twice as many steps and is about captures the Prey at about 6 percentage points less. Regardless, it is clear that V is trained to near optimality. If I were able to have done more epochs, I would be able to overfit my data, which is not an issue in this case, which would have my V be as efficient as U^*.

$V_{partial}$, however, is a different story. Although $V_{partial}$ has the same survival rate as $U_{partial}$, $V_{partial}$ takes more than twice as long as many steps, has a worse survey rate, and is significantly worse than $U_{partial}$ at capturing prey, with 10 percentage point decrease. This suggests that $V_{partial}$ is not optimal, which I would have to agree with. However, it is better than Agent 4 because although the average steps is greater, $V_{partial}$ performs better in survey rate percentage and ability to catch prey.

$V_{partial}$ is less accurate than simply substituting V into the $U_{partial}$ because $V_{partial}$ did not particularly train well, while V did.

Source(s):

Matrix Library:

https://gist.github.com/SuyashSonawane/4bf59861a35dd6ece01f1b16918e0c63#file-matrix-java

Mean Squared Error:

https://www.oreilly.com/library/view/mastering-java-for/9781782174271/f2aad1aa-9f16-4fd1-9bbc-295cf52da842.xhtml