Multi-agent Reinforcement Learning Project

Delaney Dvorsky, Carter Falkenberg

1 Predator Neural Network

1.1 Architecture

Several configurations of the neural network were tested to optimize the performance of the multi-agent reinforcement learning system. The configurations included:

1. **Three-layer Dense Network Architecture (128, 64, 64 nodes):** Each dense layer (of decreasing sizes) was followed by batch normalization and a dropout layer with a rate of 20%. Batch normalization was used to normalize the activations from the dense layers, which can speed up training and improve performance by reducing internal covariate shift. Dropout was used to prevent overfitting by randomly setting a fraction of the input units to 0 during training.
2. **Four-layer Dense Network Architecture (256 nodes in hidden layers):** The model contains four dense (fully connected) layers, each with 256 neurons and the ReLU activation function. This uniform structure intends to focus on capturing a wide array of features at each level, potentially creating a rich representation of the environment's state. This specific architecture was implemented to examine the impact of additional layers on the model's efficiency.

1.2 Parameter Tuning

1.2.1 Parameter Set 1: Comparison

We tested our two network architectures against each-other using the following initial parameters:

Learning Rate: 0.0001, using Adam optimizer. A smaller learning rate ensures that the network updates its weights gradually, reducing the risk of overshooting optimal values during training. Using the Adam optimizer, which adjusts the learning rate during training, helps in converging more efficiently and effectively.
Memory: Set to the highest feasible limit. Having a large memory capacity allows the system to store a vast number of experiences. This is beneficial in experience replay, where a diverse set of past experiences is used for training, leading to more robust learning.
Target Network Update Frequency: Every 1000 steps for stability. Updating the target network less frequently helps maintain stability in training. This is because the target for the Q-value estimation remains fixed for a longer period, which can reduce the volatility of updates.

Training Initiation: After 10 steps to build sufficient memory.
Network Update Frequency: Every 30 steps to enhance training speed. Updating the network at this frequency strikes a balance between learning from new experiences and computational efficiency. Frequent updates can lead to faster learning but require more computational resources.
Double Deep Q-Network (DDQN) was employed for increased stability. DDQN helps in reducing the overestimation bias of Q-values that is common in standard DQN. This leads to more stable and reliable learning, especially in environments with noisy or complex reward structures.
Discount Factor (Gamma): 0.999, emphasizing the importance of immediate rewards. High discount factor places more emphasis on immediate rewards, encouraging the agents to consider short-term outcomes. However, it's not exactly 1 to ensure that future rewards are not completely disregarded.
Number of Agents: 4. Using multiple agents can lead to richer learning experiences, as agents can learn from the interactions and strategies of others. This can be especially beneficial in complex environments where collaborative or competitive dynamics are important.
Maximum Steps per Episode: 10. Limiting the number of steps per episode can significantly speed up training, as it reduces the time spent per episode. However, it's important to ensure that this number is sufficient for the agents to perform meaningful actions and learn from them.

We selected the Four-layer network due to it's vastly superior run-time and improved accuracy when tested on this parameter set (see 1.3.1 vs. 1.3.2). Possible reasons for superior performance could be that in the Three-Layer Network, the decreasing size of layers might limit the network's capacity to model complex relationships in later layers, as the representational power diminishes with each subsequent layer. The use of dropout, while beneficial for preventing over-fitting, can also lead to a loss of information and might require more training data to converge. On the other hand, in the Four-Layer Network, more layers with a consistent number of nodes can provide a more substantial capacity to learn and represent complex patterns within the environment. The absence of dropout in this architecture suggests that all neurons are used during training, potentially leading to faster convergence if the network can effectively capture the relevant patterns in the data. Also, with a large memory capacity as chosen in the parameters, the four-layer network might be better at utilizing a diverse set of experiences due to its greater representational capacity. The four-layer architecture might also be benefiting more from the stability offered by DDQN, especially in a multi-agent setting where the environment's dynamics can be quite complex.

After selecting a Four-Layer Network we experimented with varying the learning rate and target network update frequency. The following parameter sets were implemented:

1.2.2 Parameter Set 2: Four-Layer Network

We used the same choices as in Parameter Set 1, except for selecting a *lower* **learning rate of 0.00005.** By reducing the learning rate, we aim to observe whether the agents can achieve better long-term performance through more refined, albeit slower, learning. This change is expected to further minimize the risk of overshooting optimal values during

training, potentially leading to more stable convergence. However, it may also lead to longer training times as the network would require more iterations to converge. See 1.3.2 for results.

1.2.3 Parameter Set 3: Four-Layer Network

We again used the same choices as in Parameter Set 1, except for selecting a *higher* **learning rate of 0.01.** The purpose is to investigate if the agents can still learn effectively despite the increased risk of overshooting optimal values. The main expectation is faster convergence, albeit with the risk of instability in training. See 1.3.3 for results.

1.2.4 Parameter Set 4: Four-Layer Network

We again used the same choices as in Parameter Set 1, except for selecting a lower **target network update frequency of 10.** This adjustment is likely to increase the responsiveness of the learning process to new information. However, it could also introduce more volatility into the training, as the target for Q-value estimations changes more rapidly. This could either lead to faster learning or, alternatively, to instability if the updates are too frequent for the agents to adapt effectively. See 1.3.4 for results.

We found that Parameter Set 1 had the best overall results, though other configurations were similarly successful, excluding parameter set 3, which never converged due to a high learning rate. The parameters in Parameter Set 1 were chosen for the reasons discussed in 1.2.1, and it seems the rational behind each choice was correct, albeit some parameters have a wide range for what can be successful. For example, Parameter Set 1 used a target update frequency of 1000 whereas Parameter Set 4 used a target update frequency of 10 with very similar results. The intuition that a larger target network update frequency is better was true, but it did not to be as high as initially hypothesized. The parameter that we found to be most important was the learning rate; if it was too high (0.01), the model never converged and if it was too low (0.00005), the model took much longer than needed to converge. It was also observed that a deeper network is more effective than a shallower one, but the number of nodes in each hidden layer is less important. Parameter Set 4 achieved nearly the same performance as Parameter Set 1 with roughly 60% fewer weights and biases to learn.

To run Parameter Set 1, use command line argument:

```
python main.py -MARLAlgorithm 'IQL' -l 0.0001 -m 100000 -t 1000 -fsm 10 -rs
30 -tt 'DDQN' -ga .999 -k 4 -ts 10 -e 100000
```

1.3 Results

Note: All model sizes are calculated assuming 4 agents.

1.3.1 Parameter Set 1

Three Dense hidden layers; 128, 64, 64 nodes			
100,000 episodes (ep)	Average score	Average steps to catch prey	% Reached goal in $\leq$ 10 steps
First 1000 ep	-9.2	7.6317*	94.1%
Last 1000 ep	75.762	3.554	82%

```
Score (first, last), steps (first, last), % goal (first, last)
(-9.2, 75.762, 7.631, 3.554, 0.941, 0.82)
```

Run Time: 121 minutes, 53 seconds
Model Size: 26,214,400 weights, 261 biases

Four Dense hidden layers; 256 nodes each			
100,000 episodes (ep)	Average score	Average steps to catch prey	% Reached goal in $\leq$ 10 steps
First 1000 ep	-8.96	6.187*	67.5%
Last 1000 ep	80.303	2.893	99.2%

```
Score (first, last), steps (first, last), % goal (first, last)
(-8.96, 80.303, 6.187, 2.893, 0.675, 0.992)
```

Run Time: 46 minutes, 1 second
Model Size: 214,748,364,800 weights, 1,029 biases

1.3.2 Parameter Set 2

Four Dense hidden layers; 64 nodes each; Learning Rate .00005			
100,000 episodes (ep)	Average score	Average steps to catch prey	% Reached goal in $\leq$ 10 steps
First 1000 ep	-15.421	9.05*	95.9%
Last 1000 ep	81.011	2.965	100%

```
Score (first, last), steps (first, last), % goal (first, last)
(-15.421, 81.011, 9.05, 2.965, 0.959, 1.0)
```

Run Time: 93 minutes, 22 seconds
Model Size: 838,860,800 weights, 261 biases

1.3.3 Parameter Set 3

Four Dense hidden layers; 64 nodes each; Learning Rate .01			
100,000 episodes (ep)	Average score	Average steps to catch prey	% Reached goal in $\leq$ 10 steps
First 1000 ep	-22.187	8.396*	85.7%
Last 1000 ep	-83.639	9.512*	75.1%

```
Score (first, last), steps (first, last), % goal (first, last)
(-22.187, -83.639, 8.936, 9.512, 0.857, 0.751)
```

Run Time: 58 minutes, 9 seconds

Model Size: 838,860,800 weights, 261 biases

1.3.4 Parameter Set 4

Four Dense hidden layers; 64 nodes each; Target Network Update Frequency 10			
100,000 episodes (ep)	Average score	Average steps to catch prey	% Reached goal in $\leq$ 10 steps
First 1000 ep	-14.89	6.403*	65.9%
Last 1000 ep	77.711	2.992	99.0%

```
Score (first, last), steps (first, last), % goal (first, last)
(-14.879, 77.711, 6.403, 2.992, 0.659, 0.990)
```

Run Time: 21 minutes, 51 seconds
Model Size: 838,860,800 weights, 261 biases

**If there were no max steps, these numbers would likely be much higher, as it has the benefit of being capped at 10.*

2 VDN Algorithm

Value decomposition networks (VDNs) aim to centralize training and distribute decisions across multi-agent systems. VDNs do this by calculating Q_{tot} across the agents:

$$Q_{tot} = \sum_{i=1}^{n} Q_i$$

In the VDN replay function, a *y_list* varaible is created with shape (num agents, batch size, num moves). Since the agent can go UP, DOWN, LEFT, RIGHT, or STAY, num moves is a constant of 5. *y_list* is created by looping through all agents and finding their respective y, with size (batch size, 5). We calculated y_{tot} by isolating each batch in *y_list* and summing over each agent's Q value for each possible move. In other words, for a given batch, we summed over the 0th axis with a resulting array of length 5. For all batches, we ended with a list of all y_{tot} with shape (batch size, 5). All calculations were done using numpy for computational efficiency.

This VDN algorithm did not converge within 100,000 episodes using the same parameters as IQL with 4 dense hidden layers, each with 256 nodes. In fact, the final 1,000 episodes yielded worse results than the first 1,000 episodes.

VDN using 4 dense hidden layers with 256 nodes, 0.00001 LR			
100,000 episodes (ep)	Average score	Average steps to catch prey	% Reached goal in $\leq$ 10 steps
First 1000 ep	-11.849	6.236	65.9%
Last 1000 ep	-50.25	6.824	50%

```
Score (first, last), steps (first, last), % goal (first, last)
(-11.849, -50.259, 6.236, 6.824, 0.659, 0.5)
```

Run Time: 41 minutes, 27 seconds

Model Size: 214,748,364,800 weights, 1,029 biases

VDN using 4 dense hidden layers with 256 nodes, 0.0001 LR			
100,000 episodes (ep)	Average score	Average steps to catch prey	% Reached goal in $\leq$ 10 steps
First 1000 ep	-11.36	6.322	67.5%
Last 1000 ep	-19.413	6.447	62.9%

```
Score (first, last), steps (first, last), % goal (first, last)
(-11.36, -19.413, 6.322, 6.447, 0.675, 0.629)
```

Run Time: 36 minutes, 5 seconds
Model Size: 214,748,364,800 weights, 1,029 biases

VDN using 4 dense hidden layers with 256 nodes, 0.01 LR			
100,000 episodes (ep)	Average score	Average steps to catch prey	% Reached goal in $\leq$ 10 steps
First 1000 ep	-16.353	6.325	63.7%
Last 1000 ep	-59.79	7.084	45.2%

```
Score (first, last), steps (first, last), % goal (first, last)
(-16.353, -59.79, 6.325, 7.084, 0.637, 0.452)
```

Run Time: 29 minutes, 46 seconds
Model Size: 214,748,364,800 weights, 1,029 biases

2 possible explanations come to mind for why the VDN did not converge whereas IQL did.

One explanation is that the assumption of additivity when factorizing Q_{tot} is not valid in this particular MARL problem. Since each agent does not need to team up with other agents in this problem in order to be effective (as observed with IQL), it may not be helpful to combine Q values from each agent, especially since one agent's Q value for going a certain direction should not affect another agent's Q value to go in that direction, besides rare circumstances where they need to team up to corner the prey. It's possible that in this case "the complex interactions in MARL are unlikely to be captured by simplistic linear summations" [1].

It is also possible that the VDN implementation had too high or low of a learning rate, causing it to either get worse over time, or improve so slowly that it was never able to converge. To test this, the VDN was run with all else being equal, but a reduced learning rate to 0.00001 (reduced by a factor of 10) and an increased learning rate to 0.01 (increased by a factor of 100). Neither adjustment yielded improved performance.

3 Prey Escape Strategy

3.0.1 Algorithm

1. *Move Possibilities Assessment:* The prey looks for all adjacent positions (neighbors) that are empty. The function returns both the positions and corresponding actions to reach them.

2. *Fallback on No Options:* If there are no available moves, the prey stays still. This could happen if the prey is surrounded by obstacles or other agents.

3. *Weighted Distance Calculation:* For each potential move, the prey calculates a score based on the distance from each predator. Closer predators are weighted more heavily, meaning the prey recognizes them as a bigger threat.

- This computes the Manhattan distance (L1 norm) between a predator and the potential move.

- The weight inversely relates to the distance to give nearer predators a higher impact on the decision-making process. Adding 1 ensures there is no division by zero if a predator is on the neighboring square.

4. *Best Move Selection:* After calculating the weighted scores for each move, the one with the highest score is considered the best. This score represents the move that maximizes the distance from predators, taking into account the increased danger of closer predators. There might be multiple moves with the same highest score, so all of these are collected into 'best_moves'.

5. *Strategic Deterministic Move:* 'return random.choice(best_moves)'– Although this line uses 'random.choice', it is not introducing randomness in the sense of unpredictability. Since all 'best_moves' have the same highest score, the choice is between equally good options. This move is deterministic from the perspective of strategy because it doesn't add randomness to avoid predictability, it's just a way to select among multiple optimal moves.

3.0.2 Results

```
(MARL) C:\Users\delan\Documents\Fall 2023\Neural_Networks_ECE553\Project 2\MARL>python main.py -evm 2
pygame 2.0.3 (SDL 2.0.16, Python 2.7.18)
Hello from the pygame community. https://www.pygame.org/contribute.html
Using TensorFlow backend.
Episode 0, Score: -989.0, Final Step: 100, Goal: False
Episode 1, Score: -824.0, Final Step: 100, Goal: False
Episode 2, Score: -992.0, Final Step: 100, Goal: False
Episode 3, Score: -1078.0, Final Step: 100, Goal: False
Episode 4, Score: -844.0, Final Step: 100, Goal: False
Score (first, last), steps (first, last), % goal (first, last)
(-4.727, -4.727, 0.5, 0.5, 0.0, 0.0)
```

In all episodes the prey successfully escaped.

4 References

[1] Factorized Q-Learning for Large-Scale Multi-Agent Systems, Ming Zhou et all.