Project Report
Multi-agent Reinforcement Learning

---

1. How you design the predator' neural network architecture.

   I designed a DNN architecture with series of dense layers and dropout layers. The choice of a simple Deep Neural Network (DNN) with dense and dropout layers for the predators in the simulation was driven by:

   Simplicity and Generalization: Dense layers are straightforward and effective for learning complex relationships in the game's state without the need for intricate architectures. Dropout layers help in generalizing the model by preventing overfitting.
   Flexibility and Efficiency: This architecture is adaptable to various types of input data and is computationally more efficient compared to more complex models.

   Regarding not using RNNs or CNNs:

   - RNNs are typically used for sequential or temporal data, which might not be essential for the static state information in this game scenario.
   - CNNs are best for spatial data (like images), which may be an overcomplication for scenarios where the state can be represented as a simple vector instead of a 2D spatial grid.

2. How you select the neural network training parameters.

   The selection of neural network training parameters for the predator agents in the simulation was influenced by both the requirements of the task and the flexibility to adjust these parameters as needed. Following are some parameter's selection criteria

   - **Learning Rate (`-l, --learning-rate`)**: Set to a relatively small value (0.00005) to ensure gradual and stable learning. This helps in fine-tuning the agents' strategies without drastic changes that might destabilize learning.
   - **Optimizer (`-op, --optimizer`)**: The choice between 'Adam' and 'RMSProp' offers a balance between adaptability to different problem landscapes (Adam) and stability in non-stationary objectives (RMSProp).

- **Batch Size (`-b, --batch-size`)**: Selected to optimize computational efficiency and learning stability. A moderate batch size (like 32) is a common choice that balances the need for stochastic gradient descent's efficiency with the stability of learning from more data at once.
- **Number of Episodes (`-e, --episode-number`)**: This sets the length of the training process. A higher number of episodes allows for more extensive learning but requires more computational time.
- **Gamma (`-ga, --gamma`)**: The discount factor for future rewards. A value of 0.95 indicates that future rewards are valued highly, encouraging strategies that consider long-term benefits.
- **Network Architecture (`-nn, --number-nodes`)**: The choice of 256 nodes in each layer suggests a sufficiently complex model to capture the necessary patterns in the data without being overly prone to overfitting.
- **MARL Algorithm Choice (`-MARLAlgorithm`)**: The flexibility to choose between QMIX, VDN, and IQL allows for experimentation with different multi-agent learning approaches, each with its strengths in coordinating multiple agents.

3. How you design the VDN algorithm.

The design of the Value Decomposition Networks (VDN) algorithm in the project focused on decomposing the joint action-value function into individual value functions for each agent. Key aspects of the VDN design included:

- **Individual Agent Value Functions**: Each agent had its own neural network to estimate its individual value function based on its local observations.
- **Summation of Individual Values**: The joint value function, representing the collective value of all agents' decisions, was computed as the simple sum of the individual agents' value functions. This is the core idea of VDN, facilitating the learning of cooperative behaviors without complex inter-agent communication.
- **Shared Experience Replay**: A shared replay buffer was used, storing experiences from all agents. These experiences were then sampled to update each agent's neural network, ensuring that learning was informed by both individual and group experiences.
- **Centralized Training with Decentralized Execution**: Training was centralized, using the global state and joint actions for backpropagation. However, execution was decentralized, with each agent acting based on its own value function.

4. How you design the prey escape strategy.

The design of the prey escape strategy in the "predators and prey" simulation was crafted to enhance the prey's ability to evade predators effectively. Key elements of this design included:

- **Assessing Safe Directions**: The prey first evaluated its neighboring cells to identify safe directions for movement. This involved checking for the presence of predators and obstacles in adjacent cells.
- **Maximizing Distance from Predators**: The prey aimed to move in a direction that maximized the distance from the nearest predator. If multiple safe directions were available, the prey chose the one leading it farthest from any predator.
- **Avoiding Traps**: The strategy also included logic to prevent the prey from moving into corners or dead-ends, where it could easily be trapped by the predators.
- **Fallback to Random Movement**: In scenarios where no clearly safe direction was discernable, the prey resorted to random movement. This added an element of unpredictability, making it harder for predators to anticipate the prey's moves.

5. A screen shot of the results in a successful run of your programs.

```
1/1 [==============================] - 0s 21ms/step
Episode 4, Score: 100, Final Step: 1, Goal: True
```

6. Try different configurations of training and report the results.

```
python main.py -e 7 -l 0.0001 -op Adam
```

```
Episode 6, Score: 94.0, Final Step: 3, Goal: True
```