Learning Complex Group Behaviours in a Multi-Agent Competitive Environment

Muhammed Kazım Sanlav Afshan Nabi

1. Introduction

The aim of this project is to use reinforcement learning algorithms to explore predator-prey dynamics, such as those seen when predator fish and prey fish interact in nature. In order to do so, we have adapted OpenAI's multiagent-particle-envs (Lowe et al.) to simulate two competing groups of fish- a predator group (has a single predator) and a prey group (2 or 10 prey). We used 2 different Q-learning algorithms (Q-Network with Experience Replay & Q-Network with Experience Replay and Fixed Q-Target) and a Monte-Carlo Policy Gradient algorithm to train these predator and prey agents. The predator is rewarded based on the number of times they collide with ("eat") prey. The prey are rewarded negatively for each collision with the prey ("every time they are eaten"). Using such reward functions, the prey group is expected to learn defense strategies and the predator is expected to learn attack strategies.

2. Methods

Reinforcement learning is a branch of machine learning along with supervised and unsupervised learning. The aim of reinforcement learning is for an agent to interact with its environment. By interaction with the environment, the agent tries to learn the sequence of actions to take in particular states that lead to the maximization of a reward signal.
In addition to an environment and an agent, there are three main elements of a reinforcement learning system. A *policy* defines the way an agent behaves at a given time by mapping states of the environment to actions to be taken when in those states. A *reward signal* defines the goal of the reinforcement learning agent. At each step, the environment outputs a scalar reward to the agent. The aim of the agent is the maximize the long term reward it receives. A *value function* of a particular state is the total amount of reward the agent can expect to accumulate in the future starting from that particular state.
In the multi-agent setting, more than one agent is present and each agent learns independently of all other agents.

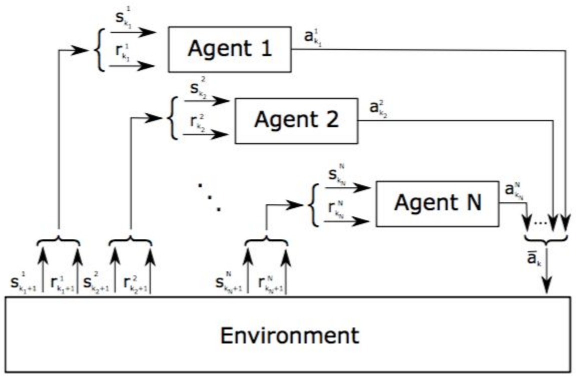

Figure 1: A multiagent environment (Chincoli et. al).

Each agent interacts with environment at discrete time steps from $t = 0, 1, 2, \ldots$ At each time step, the each agent receives a state $S_t \in S$ and based on that, selects an action $A_t \in A$. As a consequence of taking the action, each agent receives a scalar reward $R_{t+1} \in R$ and a new state S_{t+1}. In order to perform well in the long-run, the agent needs to take into account the immediate reward as well as the future rewards it expects to get. The agent tries to choose action A_t in order to maximize the expected discounted return:

$$G_t := R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where, γ is a parameter, $0 \leq \gamma \leq 1$, called the *discount rate*. The discount rate determines the present value of future rewards. If $\gamma = 0$, the agent is concerned only with maximizing the immediate rewards. If $\gamma = 1$, future rewards are taken into consideration equally. Returns at successive steps are recursively related to each other:

$$G_t := R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} \ldots = R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \ldots) = R_{t+1} + \gamma G_{t+1}$$

A *stochastic policy* is a mapping from states to the probabilities of selecting each possible action. The *action-value* function $q_\pi(s, a)$ for policy π is the value of taking an action a in state s under policy π. It is the expected return when agent starts in state s, takes action a and subsequently follows policy π:

$$q_\pi(s, a) := E_\pi[G_t | S_t = s, A_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a]$$

The action value functions can also be written recursively (*Bellman Expectation Equation* for q_π):

$$q_\pi(s, a) := E_\pi[G_t | S_t = s, A_t = a] = E_\pi[R_{t+1} + \gamma q_\pi(s_{t+1}, a_{t+1}) | S_t = s, A_t = a]$$

Solving a reinforcement learning problem means finding an *optimal policy* that achieves maximum reward. The action-value function for the optimal policy is called the *optimal action-value function* q_* :

$$q_*(s, a) := max_\pi q_\pi(s, a)$$

The *Bellman Optimality Equation* for $q_*(s, a)$ can be written as:

$$q_*(s, a) = E[R_{t+1} + \gamma max_{a'} q_*(s_{t+1,a'}) | S_t = s, A_t = a]$$

Once we know $q_*(s, a)$, the optimal policy can be followed just by choosing the action that maximizes $q_*(s, a)$ at each step.

2.1 Q-Learning

Q-learning solves the Bellman Optimality Equation iteratively. In Q-learning, a learned action-value function Q is used to directly approximate q_*, the optimal action-value function. In each step of each episode, Q is updated as:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

In case the problem is too large to learn all action values in all states separately, it is useful to learn a parameterized action value function $Q(s, a; \theta_t)$. The standard Q-learning update for the parameters after taking action A_t in state S_t and observing immediate reward R_{t+1} and resulting state S_{t+1} is:

$$\theta_{t+1} = \theta_t + \alpha(Y_t^Q - Q(S_t, A_t; \theta_t))\nabla_{\theta_t} Q(S_t, A_t; \theta_t)$$

where α is the scalar step size and the target Y_t^Q is defined as:

$$Y_t^Q := R_{t+1} + \gamma max_a Q(S_{t+1}, a; \theta_t)$$

A Deep Q-Network (DQN) is a neural network that for a given state s outputs a vector of action values $Q(s, ., \theta)$ where θ are the parameters of the network. For an n-dimensional state space and an m-dimensional action space, the DQN is a function from R^n to R^m. In this project, two algorithms using Q-Networks were implemented.

2.1.1 Q-learning with experience replay

A memory bank with a fixed size is initialized and the agent's experiences at each time step $e_t = [S_t, A_t, R_{t+1}, S_{t+1}]$ are stored in this memory bank $D = e_1, e_2, \ldots$ over many episodes. This memory bank is called the *replay memory*. During learning, mini batch updates are applied to samples drawn at random from the replay memory.

Algorithm 1: Q-learning with experience replay

```
Initialize a replay memory to capacity N
Initialize action-value function Q with random weights $\theta$
For episode 1:M do:
    Initialize environment to get initial state $s_1$
    For step 1:T do:
        With probability $\varepsilon$ select random action $a_t$
        Otherwise select $a_t = argmax_a\ Q(s_t,\ a_t;\ \theta)$
        Execute action $a_t$ and observe reward $r_{t+1}$ , $s_{t+1}$
        Store experience $s_t, a_t, r_{t+1}, s_{t+1}$ in replay memory
        Set $s_{t+1} \leftarrow s_t$
        Sample a batch $s_j, a_j, r_{j+1}, s_{j+1}$ from replay memory
        Set target $Y_j = r_j + \gamma\ max_{a'}\ Q(s_{j+1},\ a';\ \theta\ )$
        Perform gradient descent step on $(Y_j - Q(s_j,\ a_j;\ \theta\ ))^2$
    End For
End For
```

2.1.2 Q-learning with experience replay and a fixed Q-target network

The target network is a neural network with the same architecture as the Q-network. The target network parameters θ^- are the same as the Q-network parameters except that the target network weights are updated after τ steps by copying the weights from the Q-network. At other times, the target network weights remain unchanged. The target network used by the DQN is:

$$Y_t^{DQN} := R_{t+1} + \gamma max_a Q(S_{t+1}, a; \theta_t^-)$$

DQNs successfully learn the action value function q_* corresponding to the optimal policy by minimizing the MSE loss between the Q-network and Q-learning targets. In this case, Q-learning targets are calculated with respect to the old fixed parameters θ^- :

$$L(\theta) = E_{s,a,r,s_{t+1}\sim D}[(Y_t^{DQN} - Q(s, a; \theta_i))^2]$$

```
Algorithm 2: Q-learning with experience replay & fixed Q-target

Initialize a replay memory to capacity N
Initialize action-value function Q with random weights $\theta$
Initialize target action-value function $\widehat{Q}$ with random weights $\widehat{\theta}$
For episode 1:M do:
    Initialize environment to get initial state $s_1$
    For step 1:T do:
        With probability $\varepsilon$ select random action $a_t$
        Otherwise select $a_t = argmax_a\ Q(s_t, a_t; \theta)$
        Execute action $a_t$ and observe reward $r_{t+1}$ , $s_{t+1}$
        Store experience $s_t, a_t, r_{t+1}, s_{t+1}$ in replay memory
        Set $s_{t+1} <- s_t$
        Sample a batch $s_j, a_j, r_{j+1}, s_{j+1}$ from replay memory
        Set target $Y_j = r_j + \gamma\ max_{a'}\ \widehat{Q}(s_{j+1}, a'; \widehat{\theta})$
        Perform gradient descent step on $(Y_j - Q(s_j, a_j; \theta))^2$
        Every C step update $\widehat{\theta} = \theta$
    End For
End For
```

2.2 Monte-Carlo Policy Gradient

In policy-gradient methods, instead of learning an action-value function, a parametrized policy will be learned that can choose actions without using a value function. Policy will be parametrized with a vector , so taking an action α in time *t* given that environment is in state **s** and have parameters θ is:

$$\pi(a|s, \theta) = Pr\{A_t = a \mid S_t = s, \theta_t = \theta\}$$

Parameters will be learned using the gradient of an performance measure $\boldsymbol{J}(\boldsymbol{\theta})$. As we want to maximize our agents performance, we will use *gradient ascent.*

$\theta_t = \theta_t + \alpha \widehat{\nabla \boldsymbol{J}(\theta_t)}$, where $\widehat{\nabla \boldsymbol{J}(\theta_t)}$ is a stochastic estimate whose expectation approximates the gradient of the performance measure, $\Delta_\theta\ \boldsymbol{J}(\boldsymbol{\theta})$.

When action space is discrete and not to big (as in our case), common way for parameterizing is to form parameterized numerical preferences $\boldsymbol{h}(s, \alpha, \boldsymbol{\theta})$ for each state-action pair. In each state, action with the highest preference will be selected, using for example exponential soft-max distribution.

The action preferences $\boldsymbol{h}(s, \alpha, \boldsymbol{\theta})$ can be parameterized with a artificial neural network, where $\boldsymbol{\theta}$ is vector of all the weights in NN. An advantage of using soft-max and policy gradient method is, actions will be chosen stochastically with arbitrary probabilities, which sometimes a better policy than a deterministic way.

It is also known that, action probabilities change smoothly with policy gradient whereas it does not hold for the ε-greedy selection.

In monte-carlo policy gradient algorithm, each increment is proportional to the product of a return G_t and a vector $\nabla \boldsymbol{J}(\theta_t)$, the gradient of the probability of taking the action actually taken divided by the probability of taking that action. $\nabla \ln \boldsymbol{\pi}(A_t \mid S_t, \theta_t)$ is used for $\nabla \boldsymbol{J}(\theta_t)$ as $\nabla \ln x = \nabla$ x/x. This is a result of Policy Gradient Theorem.

Algorithm 3: Monte-Carlo Policy Gradient

```
Initialize policy parametrization with random weights $\theta$
For episode 1:M do:
    Generate an episode following $\pi_\theta$
    For step 1:T do:
        Set $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
        Set $\theta \leftarrow \theta + \alpha\gamma^t G \nabla ln\, \pi(A_t \mid S_t, \theta_t)$
    End For
End For
```

3. Simulation Setup

Environment: In this project, we modified the multi-agent environment developed by OpenAI Gym to obtain a new predator-prey environment which we call "Swarm". In this predator-prey environment, a red predator chases green prey which are slightly faster and smaller in size than it. Multiple predators and prey can exist in the same environment and the goal of each agent is to maximize its own reward. This environment defines the state and action space for each agent in the environment. At each step, each agent performs an action and the environment returns its reward and the next state.

States: The agent's state consists of agent's position and velocity and the relative velocity and position of all there agents.

Rewards: Red predator receives a reward of +10 if they are able to collide with ('eat') any green prey. Green prey receive a negative reward of -10 if they collide with ('are eaten by') any predator. Also, all agents are rewarded negatively for moving out of the screen. This is done so that agents remain in the boundaries of the screen.

Action space: The discrete action space for each agent is the set of four elements which represent whether the agent should move left/right/up/down.

***Neural Network Architecture and hyper-parameters**:* In algorithms 1 & 2, the Q-network for each individual agent has 2 hidden layers. Each hidden layer has twice the number of neurons as the input layer. For each agent these networks take as input the state of the agent and return the estimated Q-value for all possible actions. During training, mini-batches of size 32 are sampled from the replay memory of 2000 states, actions, rewards and next states tuples. We trained our agents with 2 different learning rates : 0.0001 & 0.001. In the beginning each episode consists of 100 steps. After every 500 episodes, the step size is increased by 50. We implemented this so that as agents learn more, they have more time to explore more of the state space, which can help agents learn longer. Both networks are implemented using Keras. Despite the fact that we coded the same neural network architecture from scratch ourselves(appendix), we preferred to use neural networks implemented in Keras for training for getting better performance. We used cloud computing to train our agents with different hyperparameters in parallel. Finally, we saved the statistics (loss, reward, collisions for each agent) for each episode. We also saved the weights of the neural network every 50 episodes to be able to recover any of them later.

For the policy gradient algorithm, the architecture of the network is the same as for the Q-network. In this case the output of the neural network is a soft-max layer that returns the probabilities for selecting each action. This was implemented in TensorFlow. The best performance (least loss) was obtained by using a learning rate

4. Performance

4.1 Algorithm 1: DQN + Experience Replay 2-Prey vs. 1-Predator; Trained for 16 hours

In first row of **Figure 2** (below), note that epsilon decreases as the number of episodes increase. This means that in the beginning, agents explore the environment and as the number of episodes increases, they learn and exploit what they have learnt rather than exploring the environment. Rows 2,3,4 and of Figure 2 show the number of collisions between predator (agent 0) and prey (agent 1 & agent 2) for each episode. Rows 5,6,7 show the rewards obtained by agents for each episode.

Note that in the beginning, there are no collisions between predator and prey. However, the reward obtained by all agents is negative. This is because in the beginning, agents leave the screen. At approx. episode 1000, predator begins to obtain positive reward; this means that predator learns to collide with/ “eat” the prey. Note also that near episode 4000, the number of collisions becomes close to 0 again. Simultaneously, the rewards for all agents are almost 0. This suggests that all the agents remain within the screen, but there are no collisions between predator and prey. In other words, the prey have learnt to evade the predator. Furthermore, this pattern of having a number of episodes where predator learns to catch prey followed by a number of episodes where prey learn to evade predator continues. For instance, near episodes 6000, 8000, 12000, 14000, 18000 the number of collisions between the predator and prey is large; predator learns strategies to catch prey. But in between these regions with high collisions, the number of collisions is small suggesting that the prey learn strategies to evade predator. Some other points of interest are as follows. Near episode 12000, the number of collisions between predator and prey 1 are high while there are no collisions between predator and prey 1. This suggests that while prey 2 has learnt a strategy to evade predator, prey 1 has not. This is interesting because in the beginning there was no difference between the two prey agents. In the same environment, the same prey have learnt differently. Another point of interest is near episode 18000: the predator seems to have learnt a great strategy to catch both prey as the number of collisions between each predator and prey increases upto 300. This means that in one episode the predator "eats" upto 600 prey!

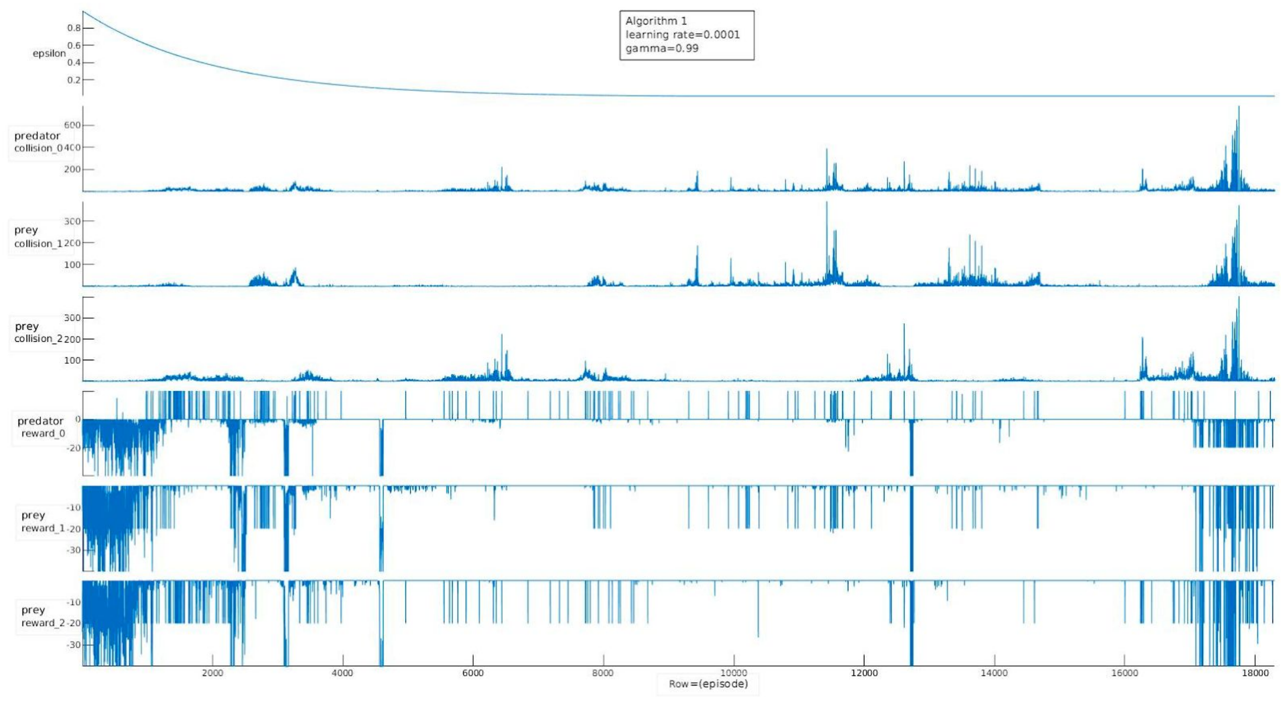

Figure 2 : Algorithm 1 *Epsilon, Collision & Reward* versus *Episode* are plotted for each agent. For this model, the learning rate = 0.0001 & gamma = 0.99

4.2 Algorithm 2: DQN + Experience Replay + Fixed Q-Target

4.2.1 2-Prey vs. 1-Predator**;** Trained for 16 hours

The first row of **Figure 3** shows that epsilon decreases with increase in number of episodes meaning that in the beginning agents explore the environment and as time passes and learning occurs, the agents exploit what they have learnt. Rows 2,3,4 and of Figure 3 show the number of collisions between predator (agent 0) and prey (agent 1 & agent 2) for each episode. Rows 5,6,7 show the rewards obtained by agents for each episode.
One interesting thing to note about Algorithm 2 is that there are no alternating sets of episodes where predator learns to catch prey and prey learn to evade predator. In fact, after approx episode 1000, when all agents learn to not leave the screen, we note that there do not seem to many collisions between predator and prey. However, near episode 15000, the number of collisions between predator and prey 2 increases upto a maximum of 800. This suggests that predator learns to catch only prey 2. Again, this is similar to what we observed in Figure 2; in the same environment, agents with similar characteristics learn differently. Close to episode 30000, we observe that all agents obtain highly negative rewards despite the absence of any collisions. This suggests that all agents once again exit the screen. This observation is also interesting since the agents had previously learnt not to exit the screen. This suggests that we might have "over-trained" the model.

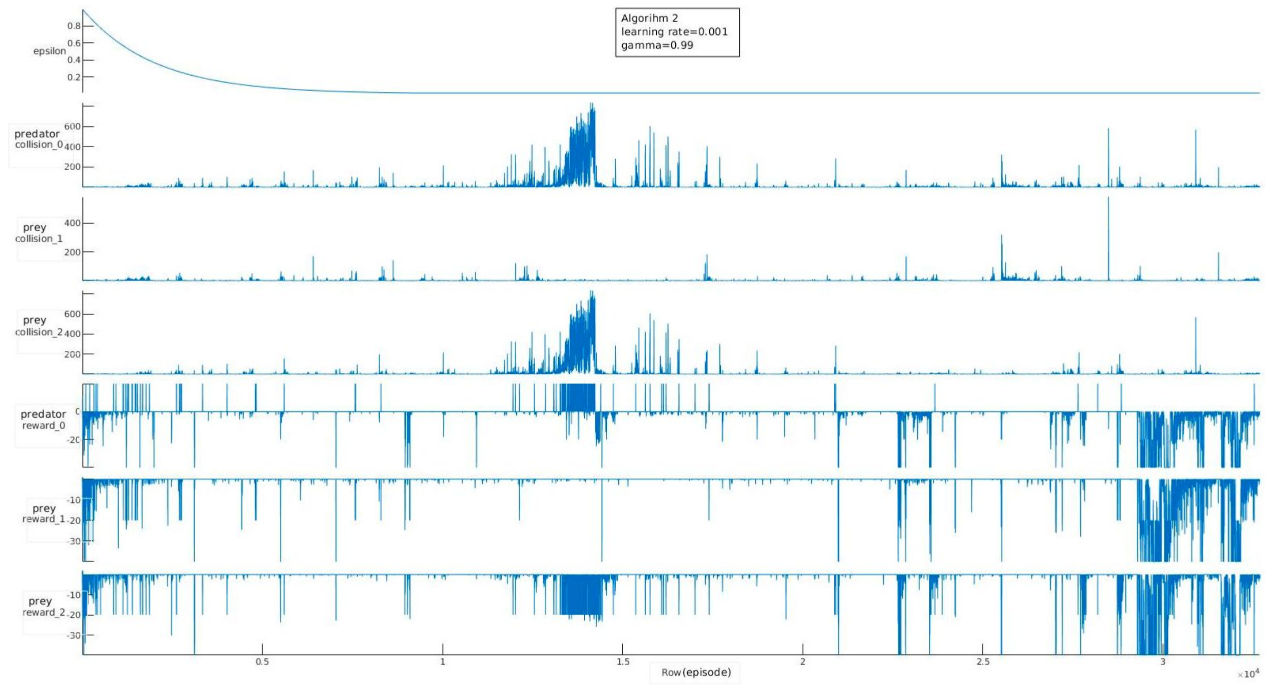

Figure 3 : Algorithm 2 *Epsilon, Collision & Reward* versus *Episode* are plotted for each agent. For this model, the learning rate = 0.001 & gamma = 0.99

4.2.1 10-Prey vs. 1-Predator**;** Trained for 20 hours

Figure 4 shows eight time-steps of a 10-vs-1 environment. The single red agent is the predator and ten green agents are the prey. At the first time step t_0, agents are placed randomly on the screen. At t_1, we notice the red predator colliding with ("eating") a green prey. Over time, we notice that the green prey group learns a defense strategy: the prey distribute themselves at equal distance from the predator. This seems to prevent the predator from catching any of the prey by causing the predator to oscillate about its position. The predator oscillates because the prey have learnt to maintain equal distance from the predator, this prevents the predator from choosing a single prey to chase. This shows how the prey learn a cooperation strategy without being explicitly coded.

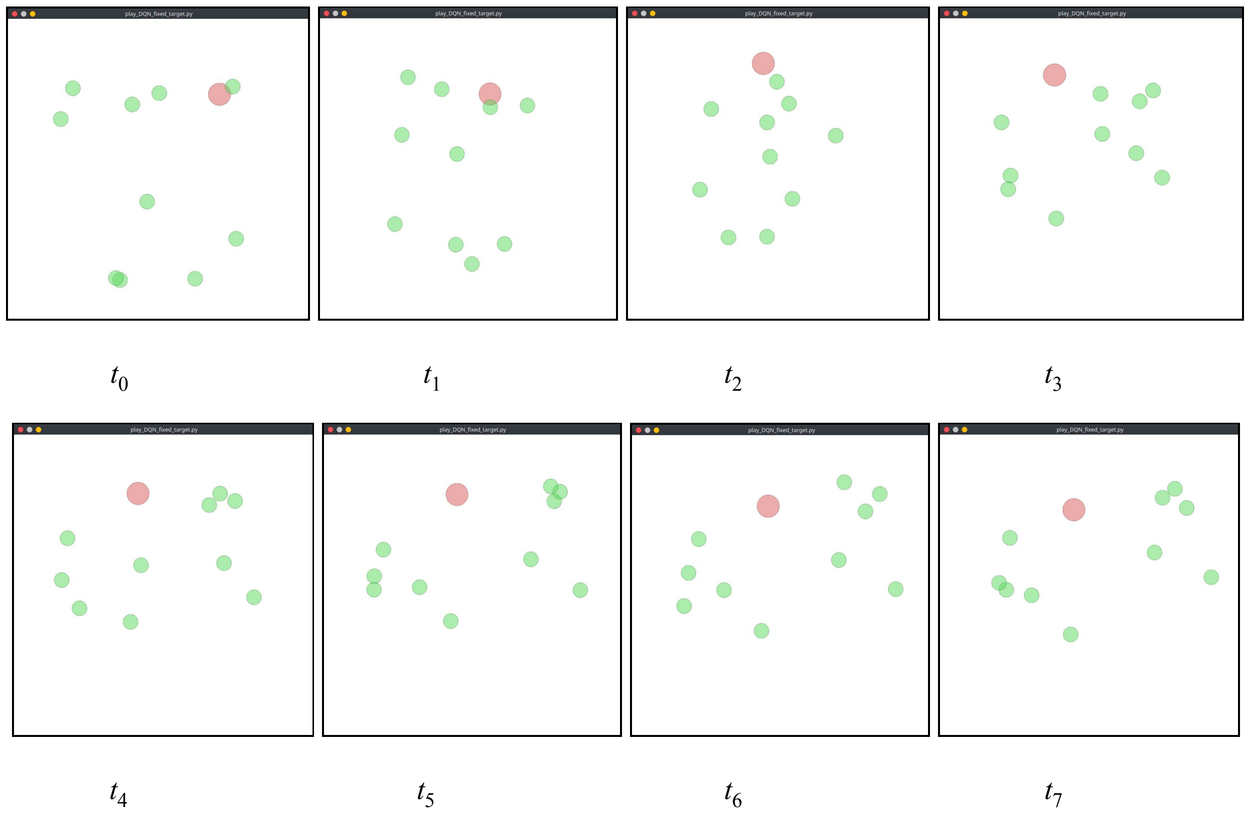

Figure 4: From Left-Right and Top-Bottom, we see 8 time-steps of an episode of 10vs1 game and how agents behave during these time steps.

4.3 Algorithm 3: Monte-Carlo Policy Gradient

Figure 5 shows the collisions, rewards & loss. Agent 0 is the predator and agent 1 and agent 2 are the prey. One point of interest about row 1,2,3 of Figure 5 is that the predator seems to be collide with a single prey (prey 1) most of the time. This observation also explains the rewards obtained by the agent (rows 4,5,6). The predator consistently obtains positive reward for colliding with the prey, while prey 1 consistently obtains negative reward for being "eaten" by the predator. Reward for prey 2 is more negative in the beginning and afterwards, it is rarely caught by the predator. This suggests that prey 2 has learnt a strategy to evade the predator, while prey 1 has not. Loss functions (rows 7,8,9) for each of the three agents approach 0 as the number of episodes increases. A difference between algorithm 3 and algorithm 1 is that in the former, there are no alternating sets of episodes where the predator or the prey learn one-by-one. Rather, learning seems to occur constantly. The predator and prey 2 seem to be able to maximize their rewards easily, but prey 1 does not.

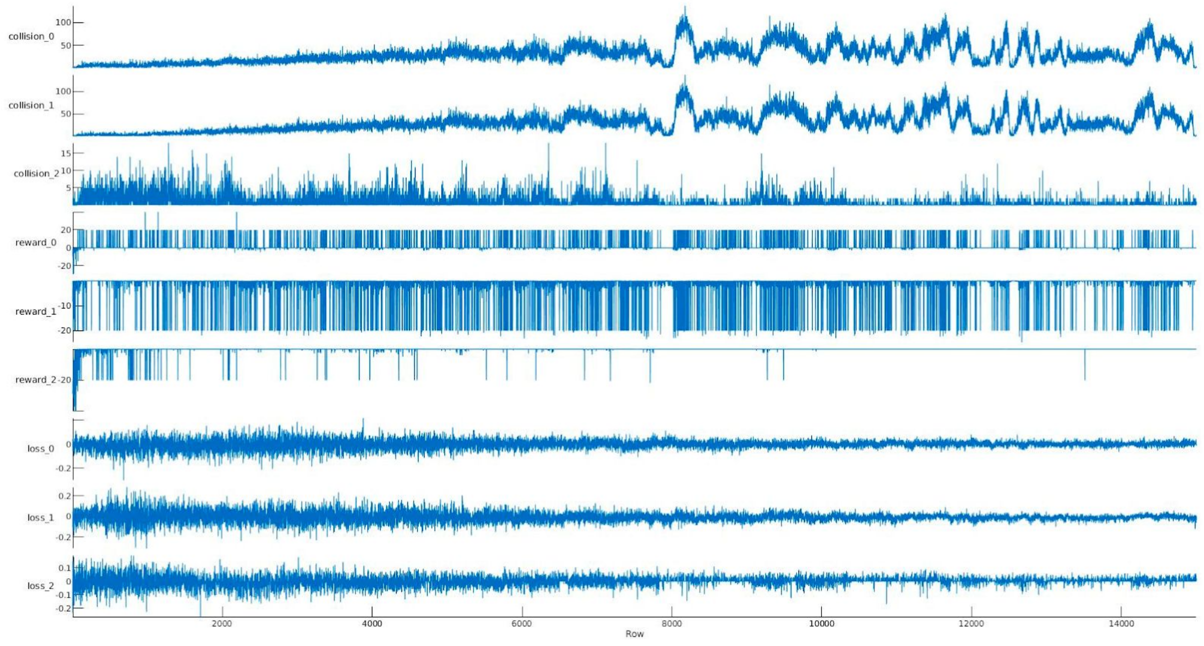

Figure 5: Algorithm 3 *Collision, Reward & Loss* versus *Episode* are plotted for each agent. Agent 0: predator; agent 1 & agent 2: prey.

Discussion

We were able to train our agents in the multi-agent environment using three different reinforcement learning algorithms. From the figures provides we note that though learning occurs in all algorithms, it seems that algorithm 3 Monte-Carlo Policy Gradient seem to perform best in terms of the predator consistently being able to "eat" prey. However, in this algorithm, as seen from Figure 5, prey 1 does not seem to learn at all and consistently obtains negative rewards. We observe such an issue in Algorithm 2 as well, (Figure 3) where prey 1 seems to learn evasion better than prey 2. The difference though is that the in Algorithm 2, such an effect is not seen throughout, instead it is only observed before episode 15000. Thus, in terms of the all agents learn to increase the reward they collect over time in both the models.

We note that the prey agents develop different strategies over time to evade the predator. One such strategy is shown in Figure 4, where we the green prey agents learn to distribute themselves equally from the predator. This seems to confuse the predator and causes it to oscillate about its position unsure of which prey to chase. We also tried different number of predators and prey (1-predator-vs-2-prey & 1-predator-vs-10 prey) and in each case, found that different strategies were learnt by agents. These will be shown in the demo, in the form of videos.

In conclusion, we note that using reinforcement learning algorithms and simple reward functions for agents, it is possible for the prey agents to discover cooperative strategies that prevent the predator from "eating" them. It is also interesting to note that though the prey agents are the same in the beginning, as training proceeds, the same prey agents learn different strategies. While some are able to successfully evade predators, others are not. Such a concept is similar to the concept of "fitness" observed in real biological systems, where some individuals are able to survive better than others. Extensions of this work could incorporate more agents to study the different strategies developed as the number of individuals in the group increases. Moreover, instead of a 2-D environment, a 3-D environment might be used where the strategies developed by agents might resemble those observed in real predator-prey interactions.

References

Chincoli, Michele, and Antonio Liotta. "Self-learning power control in wireless sensor networks." Sensors 18.2 (2018): 375.

Lowe, Ryan, et al. "Multi-agent actor-critic for mixed cooperative-competitive environments." Advances in Neural Information Processing Systems. 2017.

Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).

Matiisen, Tambet. “Demystifying Deep Reinforcement Learning.” 2015

Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. MIT press, 2018.

Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning." AAAI. Vol. 2. 2016.