Bot Fight - Multi-Agent cooperation

Matteo Maggiolo Henry Mauranen

March 28, 2018

1 Introduction

2 Bot-Fight Environment

For the purposes of the experiments we implemented a game environment we called "Bot-Fight". It consists of a 2D environment with two teams of bots and some obstacles. The idea was to implement different types of agents controlling these bots, which can move around the environment and fire straight forward. The obstacles in the environment could block bullets and prevent bots from moving through them. This game was also human playable. All of these game elements are showcased in Figure

3 Rule-Based Bots

The first iteration of an agent in our game was a simple rule based bot that always faced towards the first opponent and fired when the opponent was visible. This then evolved into an early version of the state-based agent by implementing A* pathfinding. The bot moved towards the opponent if they were not visible and fired at them if they were.
When playing against humans, these bots were too rudimentary to be of challenge. Even though they didn't struggle with controls like humans sometimes do, they were not avoiding bullets or using strategies to approach the enemy. The bots also knew everything about the game world, which made boring game experience. In video games, bots typically know their immediate surroundings, not the entire world.

4 State-Based Bot

Improving on these ideas, we discretized the bots to 4 states: dodge, roam, shoot and search. Using simple rules for state changes and discrete states, we could improve the performance and game play experience. We also planned this as the training controller for learning bots, which we hoped to allow us to learn more complex behaviour than what we could implement with simple rules.
Dodge state was the most important change for this bot. It made the bot significantly more challenging to play against. We used simple vector algebra to determine whether the closest bullet was going to collide with the agent and to determine the best direction to dodge to. Bots prioritize survival by putting dodge above other states in importance.
Roam and search states were improvements on the omniscience of the bot. We

decided to only give bot knowledge of the closest visible opponent. If it didn't know about any opponents, it simply roamed around to random locations on the map. After spotting and potentially losing sight of the enemy, the agent would go to the location where the enemy was last seen, before returning to roam.
Shoot state remained the same as with rule-based bot: Fire at enemy if they are visible.
Two important limitations of this bot were its lack of coordination with teammates and its inability to multi-task (i.e. to be in multiple states at the same time). We planned on experimenting with rule-based team controllers, which would be responsible of state-based agents' state changes, but our first priority was to create a bot that would learn on its own.

5 Deep Q-Learning

Based on Google's DeepMind paper on Deep Q-Learning with Atari games, we decided to try a similar approach to create another bot for our game. Q-Learning is a model free, reinforcement learning technique to find the optimal policy π^* that maximizes the sum of cumulative rewards obtained from the environment. The way this works is that the agent picks an action based on the value of the Q function (which approximates the sum of cumulative rewards) and the state in which the environment is s_t, then it performs this action on the environment and receives a reward r_t and a new state s_{t+1}. Finally, the Q function is updated based on the tuple (s_t, a_t, r_t, s_{t+1}). With these definition, the sum of cumulative rewards is defined as:

$$\sum_{t=t_0}^{\infty} \gamma^t \cdot r_t \tag{1}$$

Where γ is the discount factor for future rewards. The final optimal policy π^* for choosing an action is then:

$$\pi^*(s_t) = \arg\max_{a'} Q^*(s_t, a') \tag{2}$$

where

$$Q^*(s_t, a) = r_t(s_t, a) + \lambda \cdot \max_{a'} Q^*(tr(s_t, a), a') \tag{3}$$

In this last equation, $r_t(s_t, a)$ is the reward obtained executing action a when in state s_t, and $tr(s_t, a)$ is the state obtained after executing action a on state s_t. In order to learn the optimal Q function, a training step needs to be executed, where we update the Q function based on the obtained reward. That is:

$$Q(s_t, a) \leftarrow Q(s_t, a) + \alpha \cdot [r_t(s_t, a) + \lambda \cdot \max_{a'} Q(s_{t+1}, a) - Q(s_t, a)] \tag{4}$$

where α is the learning rate.

In order to apply deep learning to classical Q learning, some changes need to be made to the algorithm. First, for the Q function (classically represented as a table with state, action pairs) a neural network is used. Then, a choice for a representation of the game state needs to be made. In the paper, DeepMind used a deep Convolutional Neural Network, which took the entire screen as input, and returned a Q-values vector with an entry for each possible action. In our experiment, however, we decided to go for a simpler model (Feed-forward Net) and a simpler state representation to test whether it was descriptive enough for our game. The network would receive at each time step a vector with information about:

- The x position of the bot
- The y position of the bot
- The rotation of the bot
- The health of the bot
- The rotation the bot should match to face towards the closest enemy
- The distance from the closest enemy
- Boolean representing if the closest enemy is visible (no obstacles in the way)
- The x position of the closest enemy bullet (if any)
- The y position of the closest enemy bullet (if any)
- The dot product between bullet speed and vector between bullet and bot (if any)
- Boolean representing if there is an obstacle right in front
- Boolean representing if there is an obstacle on the right
- Boolean representing if there is an obstacle on the left

We normalized each of these values in the range $[0, 1]$, and as result of the feed-forward pass we obtained the expected Q values for each action. The training step for such a network consists of an update of the network parameters by applying a gradient descent step, minimizing the following loss function at time step t:

$$L(\theta_t) = [r_t + \lambda \cdot \max_{a'} Q(tr(a, s), a'; \theta_t) - Q(a, s; \theta_t)]^2 \tag{5}$$

where a is the chosen action at t.

Another change to be made to the classical algorithm was the introduction of an episodic memory D of fixed size. The motivation for this is that if we simply apply a network update on the results obtained at each step, subsequent

gradients are going to be very similar (updating every 1/fps $\approx$ 0.16 sec), preventing the network to well generalize the training. Therefore, at each update step, the transition tuple (s_t, a, r_t, s_{t+1}) is stored in the memory, then a random mini-batch of episodes is sampled from the memory (size of mini-batch was 32), and an update step is applied over the mini-batch.

Unfortunately, however, the network was not able to learn in a reasonable amount of time (200 full games) starting from no training, in neither 1v1 (Deep QL against itself) nor 2v2 and 3v3 (one Deep QL on each team, and the rest chosen randomly among the State-Based and Pathfinder controllers). Finally, as last experiment, we tried to first pre-train the network by watching the State-Based controller play and simply trying to approximate its actions. This led to a better result than the previous: in this one the bot was able to better roam around the map, and it did give some responses when the enemy bot enter into sight (twitching, starting to shoot at a fixed point). However it was still not able to learn how to turn to face the opponent, even after Deep Q learning was applied to the pre-trained network.

6 Monte Carlo Tree-Search

One of our initial approaches on improving the state-based bot was implementing Monte Carlo Tree-Search (MCTS) in some of its states, namely the shoot and dodge states. The game as whole was too large to be efficiently simulated in real time. The timeframe for dodging or shooting to hit an enemy was always fairly small, which is why decided on splitting the game to subtasks.
Our approach here was to simulate only one or two seconds of the game and then evaluate the state based on if the agent had hit an enemy or gotten hit by one. We further attempted to optimize the simulations by making them run at larger time step than the game itself. It is not necessary to make a decision 60 times each second to achieve good game play against humans.
This approach was proven to be unsuccessful as a single simulation for a second long game took 0.07 seconds on average. This was too slow to run multiple simulations and make effective decisions and so MCTS approach was abandoned.
We briefly discussed making a more rough grained simulation of the game to speed up decision making, but it was not feasible to make another entire game engine for the scope of this project. This is something to look into however, if one would be interested in creating more efficient bots for this style of games.

7 Conclusion