2019 International Conference on Artificial Intelligence and Computing Science (ICAICS 2019)
ISBN: 978-1-60595-615-2

Training an Agent for Third-person Shooter Game Using Unity ML-Agents

Jun LAI, Xi-liang CHEN* and Xue-zhen ZHANG

Army Engineering University of PLA, Nanjing, China

*Corresponding author

Keywords: TPS, Deep reinforcement learning, ML-agents.

Abstract. The development of deep reinforcement learning has been widely used in the field of games. FPS and TPS are very suitable. At the same time, there are many training platforms and simulation environments. In this paper, a third-person shooting environment is built under the environment of Unity ML-Agents and some agents are trained by PPO and actor-critic algorithm. Through this visual in-depth reinforcement learning, agents learn skills such as searching for enemies in unfamiliar areas, searching for cartridges after injury, supplying ammunition and multi-batch of dangerous targets. Experiments show that deep reinforcement learning can fully improve the intelligence level of agent in the field of game, which is obviously superior to the ordinary intelligent behavior based on state machine.

Introduction

(1) TPS games

Third Personal Shooting Game (TPS) is a game in which players use shooting weapons for unit confrontation. In the course of the game, players can observe the characters through the electronic screen.

The difference between TPS and First Personal Shooting Game (FPS) is that only the view of the protagonist is displayed on the FPS screen, while the protagonist in TPS is visible on the game screen. This can be more intuitive to see the role's actions, clothing and other parts of the FPS which can't be shown, more conducive to the observation of the role's injuries and surrounding things, as well as trajectory.

(2) Intensive learning

Reinforcement learning is developed from animal learning and adaptive control of parameter disturbance. Its basic principle is that if an agent's behavior strategy results in positive reward (reinforcement signal), the trend of agent's behavior strategy will be strengthened in the future. That is to say, the agent learns by trial and error, and the reward guidance behavior obtained through interaction with the environment aims to make the agent get the maximum reward.

RL is different from the supervised learning in connectionism learning, which is mainly manifested in the reinforcement signal. The reinforcement signal provided by the environment in RL is an evaluation of the quality of the generated action (usually a scalar signal), rather than telling the reinforcement learning system how to generate the correct action. Since the external environment provides little information, RLS must learn from its own experience. In this way, RLS acquires knowledge in the action-evaluation environment and improves the action plan to adapt to the environment.

The purpose of this paper is to study how RL acts as a third-person robot under Unity's ML-agents. Specific tasks include target search, collection of objects and elimination of enemies.

Platform

(1) ViZDoom

ViZDoom is an AI research platform that allows you to train robots to play Doom, a classic first-person shooting game originally released by id Software in 1993. ViZDoom uses an open source Doom engine ZDoom to interact with games. ViZDoom contains a series of sample robots written in C ++, Java, Lua and Python. In many cases, the model itself relies on various potential in-depth learning libraries, such as TensorFlow and Theano. With ViZDoom the robot will train for the scene. ViZDoom contains several scenarios in the release source code. Scenes define Doom maps, robot - usable controls (such as left turn, attack, etc.), player modes, and skill levels. However, the platform can only be used to train robots for first-person shooting.

(2) RobotSchool

Roboschool provides a new OpenAI gym environment for controlling simulated robots. Eight of these environments have been readjusted as free substitutes for the pre-existing MuJoCo to create a more realistic movement. We also include several new and challenging environments. Roboschool also makes it easy for it to train multiple agents together in the same environment. Roboschool has 12 environments, including tasks familiar to Mujoco users and new challenges, such as humanoid walkers and multiplayer ping-pong environments. However, the simulation environment is still limited and cannot be expanded freely.

(3) ML-Agents

ML-Agents is an open source Unity plug-in that enables us to train intelligent agents in both game and simulation environments. You can use reinforcement learning, imitation learning, neuroevolution or other machine learning methods to control and train the agent through a simple Python API. We also provide the implementation of the most advanced algorithms (based on TensorFlow) so that game developers and amateurs can easily train intelligent agent for 2D, 3D and VR/AR games. These agent can be used for a variety of purposes, including controlling NPC behavior, automatically testing game builds, and evaluating pre-released versions of different game design decisions. ML-Agents are beneficial to both game developers and AI researchers because it provides a centralized platform for us to test AI's latest progress in Unity's rich environment and make the results available to more researchers and game developers.

Method

(1) PPO

PPO algorithm selects a conservative lower bound of the objective function through a simple clip mechanism without calculating the quadratic gradient in the constraints of TRPO algorithm, thus improving the sampling efficiency and robustness of the algorithm, and reducing the complexity of super-parameter selection.

1) Implementation method based on policy probability ratio clip:

$$L^{CLIP}(\theta) = \hat{E}_t\left[min\left(r_t(\theta)\hat{A}_t\right), clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t\right]$$

θ is a policy parameter

$\hat{E}_t$ represents an empirical expectation over time

r_t is the ratio of the probability under the new strategy to the probability under the old strategy

$\hat{A}_t$ is the estimated advantage of time t

ε is a super parameter, usually 0.1 or 0.2

In the above formula, $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, the situation where $r_t(\theta)$ leaves $[1-\varepsilon, 1+\varepsilon]$ is eliminated by clip, when r_t causes the objective function to increase to a certain extent, r_t changes are ignored, and when r_t causes the objective to decrease, it is normally used for optimization. Here is the most critical point.

Complete objective function of PPO algorithm:

$$L_t^{CLIP+VF+S}(\theta) = \hat{E}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](S_t)]$$

In the above formula, where c_1 and c_2 are coefficients, S and L_t^{VF} are respectively the error terms of the strategy entropy and the value function. $L_t^{CLIP}(\theta)$ is the main objective function of strategic gradient optimization discussed earlier.

(2) Actor–Critic

Thus far, we have assumed that the internal training structure of PPO mirrors what we learned when we first looked at neural networks and DQN. However, this isn't actual the case. Instead of using a single network to derive Q values or some form of policy, the PPO algorithm uses a technique called actor-critic. This method is essentially a combination of calculating values and policy. In actor-critic, or A3C, we train two networks. One network acts as a Q-values estimate or critic, and the other determines the policy or actions of the actor or agent.

We compare these values in the following equation to determine the advantage:

$$Advantage: A = Q(s,a) - V(s)$$

However, the network is no longer calculating Q-values, so we substitute that for an estimation of rewords, as shown in Fig 1:

$$Advantage: A = R - V(S)$$

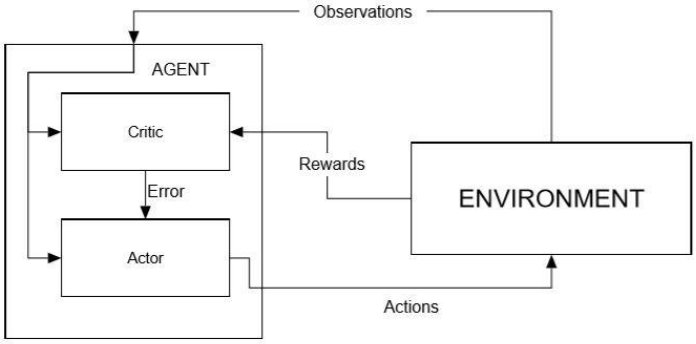

Figure 1. Diagram of actor-critic network.

The error term that is communicated between the critic and actor is derived from the following equations:

$$ValuesLoss: L = \sum\left(R - V(s)\right)^2 (SumSquaredError)$$
$$PolicyLoss: L = -log\left(\pi(a|s)\right) * A(s)$$

Our intention here is to minimize the error, but a better term/equation to use is the calculation of entropy:

$$H(\pi) = -\sum(P(x)log\,P(x))$$

Entropy($H(\pi)$) measures the spread of probability, while a high entropy represents an agent with multiple similar actions, which makes the agent's decisions difficult. A smaller value for entropy equates to an agent that can make better-informed decisions. This updates our loss function to the following:

$$PolicyLoss: L = -log\left(\pi(a|s)\right) * A(s) - \beta * H(\pi)$$

Finally, when we combine the two loss functions, value and policy, we get the final equation for loss, as shown in the following:

$$L = 0.5 * \sum\left(R - V(s)\right)^2 - log\left(\pi(a|s)\right) * A(s) - \beta * H(\pi)$$

This loss function is the one our network (agent) is trying to minimize. While this is the form of training we have been using since the beginning to use PPO, we have omitted one other critical improvement. Actor–critic training was derived to work with multiple asynchronous agents, each working within their own environment. We will explore the asynchronous side of training in the next section.

Experiments Setup

ML-Agents contains three advanced components: 1) Learning Environment: It contains Unity scenes and all game characters, as shown in Fig 2. 2) Python API: It contains all machine learning algorithms

for training (learning a certain behavior or policy). Note that unlike the learning environment, the Python API is not part of Unity, but is located outside and communicates with Unity through External Communicator. 3) External Communicator - It connects the Unity environment to the Python API.

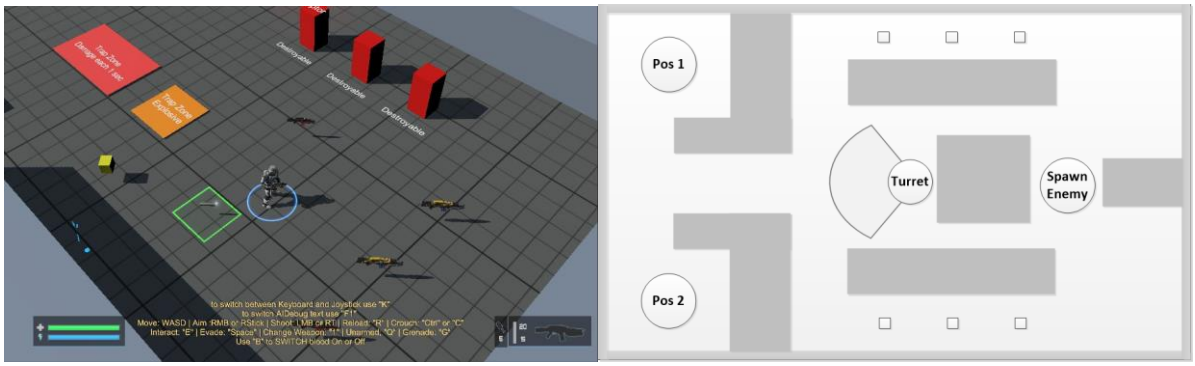

Figure 2. Three - dimensional scene and two-dimensional schematic diagram.

The learning environment contains three additional components that can help organize the Unity scene:

•Agent - It can be attached to a Unity game object (any character in the scene), responsible for generating its observations, performing the actions it receives, and allocating rewards (positive / negative) in due course. Each agent is associated with only one Brain.

•Brain - It encapsulates the decision logic of agent. Brain retains the policy of each agent, which determines the actions that agents should take in each case. More specifically, it is a component that receives observations and rewards from the agent and returns actions.

•Academy - It directs the agent's observation and decision-making process. Within Academy, you can specify several environment parameters, such as rendering quality and environment running speed parameters. External Communicator is located within Academy.

The whole learning environment has a global Academy, with multiple agent corresponding to each game character one by one. Although each agent must be connected to one Brain, multiple agents with the same role are associated to the same Brain. Brain defines all possible observation and action spaces, and the agent connected to it can each have their own unique observation and action values, as shown in Fig 3.

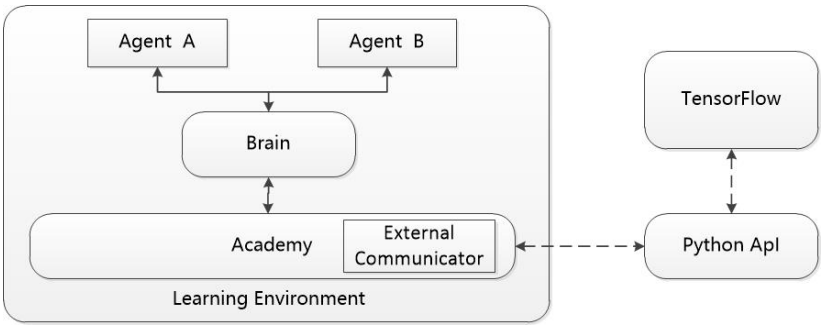

Figure 3. Structure Diagram of ML – Agents.

The logic of the agent includes four parts. One is CollectObservations. The agent collects the observation input data of agents and its own velocity through Ray Perception, as shown in Fig 4. The second is AgentAction, which obtains the input value of Visual Observation and converts it into various control quantities in Table 1. The third is that AgentReset resets the agent after each round. The fourth is to set up rewards for the agent, as shown in Table 2.

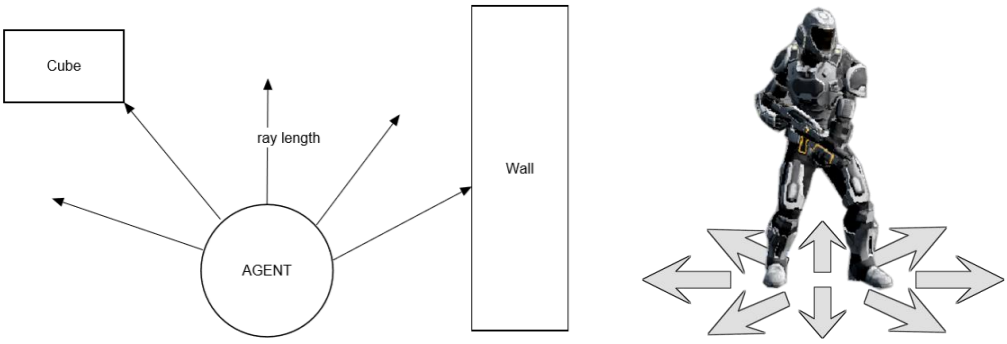

Figure 4. A schematic diagram of Collect Observers.

Table 1. Action table.

Check	Values
F/B Direction	Forward/Back/None
L/R Rotation	Left/Right/None
Jumping	Yes/No
Picking	Yes/No
Sprinting	Yes/No
Rolling	Yes/No
Shooting	Yes/No

Table 2. Reward function for the RL task.

Reward	Value
Ammo Collected (Ammo=0)	1.0
Health Collected (Health<100)	1.0
Enemy Killed	1.0
Damaged	-0.1
Died	-2.0
Collide Wall	-0.01
Distance travelled(1m/step)	-0.00001

ML-Agents comes with a variety of implementations of the most advanced algorithms for training intelligent agents. In this mode, the Brain type is set to External during training and Internal during prediction. More specifically, during training, all agent in the scene send their observations to the Python API through External Communicator. The Python API processes these observations and sends back the actions to be taken by each agent. During the training, these actions are mostly exploratory, designed to help the Python API learn the best policy for each agent. After the training, the policy learned by each agent can be exported. The implementation of training work is based on TensorFlow, so the learned policy is only a TensorFlow model file. Then in the prediction phase, we switched the Brain type to Internal and added the TensorFlow model generated from the training phase.

Experiments Result

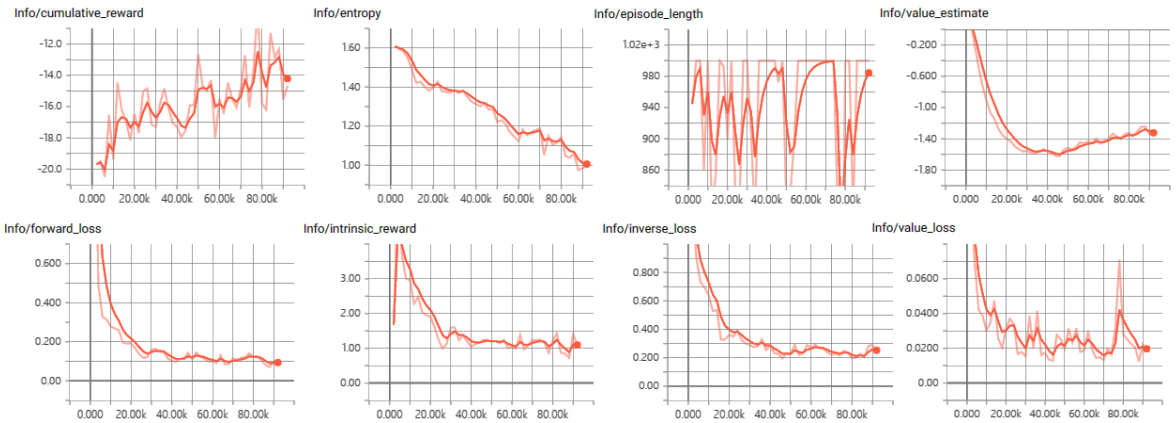

Figure 5. Training Legend for TensorBoard.

The experimental results show that there has been a steady increase in Cumulative Reward, as shown in Fig 5. Although there are slight fluctuations in the whole process which are acceptable. Entropy corresponds to how random the decisions of a Brain are, which has also dropped steadily throughout the training process. Other parameters, such as Learning Rate, Policy Loss, value estimation, Value Estimate and Value Loss, are basically in line with expectations.

The actual operation results in the simulation environment show that agent has certain tactical skills and can complete specific tasks.

Conclusion

We used Unity3d, a general game engine, to build a complex three-dimensional battlefield environment, and used ML-Agents to realize the docking between TensorFlow and agents, and used PPO+ Curiosity Search algorithm to show that a third-person shooting character in a 3D environment can learn through intensive learning. Agents have learned the skills of searching for enemies in unfamiliar areas through this visual intensive study, searching for medicine boxes after injuries, supplying ammunition and multiple batches of dangerous targets. All the work shows that Unity3d ML-agents is an excellent and free development platform for reinforcement learning. The performance of agent can be obviously improved through deep reinforcement learning. Compared with the traditional decision-making method based on state machine, it not only has less coding, but also has higher intelligence level.

References

[1] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, et al. Unity: A General Platform for Intelligent Agents. arXiv:1809.02627. (2018).

[2] R.S. Sutton and A.G. Barto, Reinforcement Learning: An Introduction. MIT Press, 1998.

[3] M. McPartland and M. Gallagher, “Reinforcement learning in first person shooter games,” in IEEE Transactions on Computational Intelligence and AI in Games, vol. 3, 2011, pp. 43–56.

[4] D. Wang and A. Tan, “Creating autonomous adaptive agents in a realtime first-person shooter computer game,” in IEEE Transactions on Computational Intelligence and AI in Games, vol. 7, 2014, pp. 123–138.

[5] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaskowski, et al. Vizdoom: A doom-based AI research platform for visual reinforcement learning. CoRR, vol. abs/1605.02097. (2016).

[6] G. Lample and D.S. Chaplot. Playing FPS games with deep reinforcement learning. CoRR, vol. abs/1609.05521. (2016).

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M.A. Riedmiller, Playing atari with deep reinforcement learning. CoRR, vol. abs/1312.5602. (2013).

[8] McPartland and Gallagher. Learning to be a bot: Reinforcement learning in shooter games. In AIIDE. (2008).