Training A 2D Pixel Tank Using Reinforcement Learning

Anavart Pandya

Mechanical Engineering | IIT Gandhinagar

anavart.pandya@iitgn.ac.in

Abstract

This paper aims to analyse the behaviour of system of two 2D pixel tanks competing each other where one tank is trained using reinforcement learning algorithm and another tank is hardcoded (or follows a set of predefined commands). The problem is divided into 2 levels where in the first level, the tanks can only move in 1 direction and in the second level, the tanks are allowed to move freely in a 2D space. The second level is further divided on the basis of action space (discrete or continuous). Agent in each level is trained using a unique approach. A general procedure for approaching the complex versions of this problem is discussed and insights on topics like hyperparameter tunning, reward shaping and reward modelling are given. Some interesting open research domains are discussed at the end.

1. Introduction

Reinforcement learning is a becoming a wide field of research now-a-days and various fields are implementing this technology to achieve super human performance or even better performance. Many countries are building powerful weapons such as completely automated tanks and human like soldiers using this technology. Many different algorithms have been invented and are applied now-a-days. But finding the most accurate algorithm for competitive tasks such as wars, implementing them in real life scenarios and achieve human like performance or super human performance is very tough and is still an open research problem. The first step for implementing and building such kind of advanced systems is analysing its behaviour in a simulation.

This paper aims to analyse the behaviour of system of two 2D pixel tanks competing each other where one tank is trained using reinforcement learning algorithm and another tank is hardcoded (or follows a set of predefined commands). The simulation environment and the system of tanks are made using the 'Pygame' library. The task is divided into two levels. In the first level the tanks are constrained to move only in one direction and have a very limited set of actions that they are allowed to perform. In the second level, the tanks are allowed to move in a 2D space and have bigger set of actions. The second level is further divided based on the action space (discrete or continuous) that each tank can use. Each level will be treated as a separate Markov's Decision Process **(MDP)**[1] and relevant procedure will be followed for each MDP based on its nature.

2. One Dimensional motion of Tanks

Here the motion of both tanks will be one dimensional and specifically can move only along the y-direction. To get an overview of how does the simulation environment look, please refer to **Fig 2.1** given below.

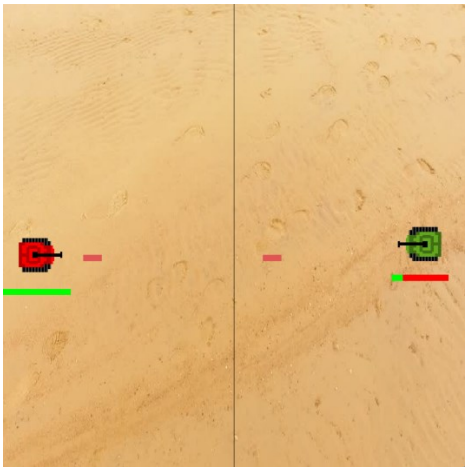

Fig 2.1

[1]Markov's Decision process (or MDP) is a method to define and formulate a Reinforcement Learning problem.

The green tank is the agent that is to be trained using reinforcement learning and the red tank is the enemy tank which is given a set of predefined instructions. We will first define the problem using the MDP. The aim of the agent is to win against the enemy tank and the constraint is that it can only move along the y-direction (i.e., up and down). The environment is a custom Open AI Gym environment. The state space is large and hence can be treated as continuous. The action space is discrete and consists of four actions (up, down, shoot, no-operation). The task is episodic (as soon as the agent kills the enemy or the enemy kills the agent, the environment is reset and a new episode begins). To make the simulation near to real life situation, the agent tank is constructed in such a manner that it can only get the coordinates of the enemy tank if the enemy tank is in a particular range of the agent (here range is defined as the difference of the y-coordinates of agent and the enemy). The state space mentioned above, is a 4-dimensional vector. The first dimension returns the y-coordinate of the agent, the second returns the health of the agent, third returns the y-coordinate of the enemy and the fourth dimension returns the health of the enemy.

The MDP is now well defined and we can further proceed to design the reward function for the MDP. Designing a reward function is a very crucial task while solving a reinforcement learning problem. If we design a very intuitive reward (i.e., rewarding the agent for every small task it performs), we bias the agent to perform on our thinking and intuition. This is not a good sign as there is a high possibility that the agent comes up with a method that is far better than our intuition. But if we build a very sparse reward function (for example, rewarding +1 for winning and -1 for losing), the agent may take a lot of time to train to achieve a very good performance. Therefore, we need to consider factors such as size of action space, size of state space, whether the action space is continuous or discrete and do we have the access to a very high computing facility to decide a suitable reward function.

For level 1 of the current problem, the action space is small and discrete. Therefore, we can afford to have a sparse reward but we will need to provide some intuition to the agent. In this case the reward function is defined as:

- Reward = -(enemy health)/100 for each time step
- Reward = -10000 for losing the game
- Reward = + (agent's health left) for defeating the enemy

As you can see that the reward function is sparse and simple but at the same time some intuition is provided to the agent. The agent is rewarded some negative reward proportional to the enemy's health at each time step, giving it an indirect command that it should kill the enemy as fast as possible. Giving the reward equal to the agent's health left on winning gives it an indirect command that it should win the game without damaging itself much. A high negative reward is given if it loses the game which gives it a direct command that losing a game is not at all favourable.

Now we have all the ingredients ready to apply a reinforcement learning algorithm for this problem. Here the state space is large and continuous (note that the state space is a 4-dimensional vector but the value of 1st and 3rd dimension is continuous and can have many values) hence applying tabular RL algorithms is not feasible. Therefore, we need to apply an algorithm that uses function approximation. Also, we do not have access to the model at every moment and hence model free function approximation algorithm will be used in this case. The action space is discrete and the algorithm supporting discrete action space should be used. The

best algorithm that suits these MDP conditions is Proximal Policy Optimization Algorithm (also known as PPO algorithm). This algorithm is an on-policy policy gradient algorithm derived from the actor-critic algorithm. It can be applied for both continuous and discrete action space. This algorithm is much better as compared to DQN (deep Q-Network) algorithm. Mlp (Multi-layer Perceptron) policy is used with this algorithm. Mlp policy uses a deep/dense neural network architecture for function approximation. The final model was trained for 1M steps using the stable baselines 3 library and Open AI Gym environment. You can access the code and the trained model through the GitHub link provided at the end of this paper.

It was observed that the agent achieved superhuman performance and was able to defeat the agent easily. One interesting point that came out of this research is that although agent played better than human against the hardcoded enemy but was not able to defeat a human player playing against it. This was due to the fact that the agent studied the movement of the hardcoded enemy so well that it was greatly biased towards those kinds of movements and hence was not able to perform well if a different set of movements was used against it. To make it play well against human, we need to train the agent on itself, i.e., we need to use competitive multiagent environment where the agent and the enemy learn together. This point is further discussed in detail in this paper.

3. Two-Dimensional motion of Tanks with Discrete Action Space

We will add a bit more complexity to the last problem. Now the tanks are free to move in any direction in a 2D space. The action space is still discrete but the number of dimensions of the space vector have been increased. The action space consists of 6 actions (forward, backward, rotate left, rotate right, shoot, no-operation). The state space remains the same, but the state space vector has now higher dimensions. To get an overview of how does the simulation environment look, please refer to **Fig 3.1** given below.

Fig 3.1

Here the MDP will almost look the same as the previous one, just the difference will be that, the size of the action and state space vectors have been increased. Therefore, the reward function will need to be changed a bit. It will not be as sparse as previous one or else it will take a lot of computational power or time to train. It is observed that, increasing the action space vector by one dimension increases the complexity of the RL problem by many folds. Therefore, after many trials and errors, a suitable reward function is designed, which is:

- Reward = + (15- absolute (target angle – actual angle)) x10 if the enemy is within 30 degrees range of the agent's gun point.
  (Note: Here the target angle is the angle that gunpoint point of agent should make with respect to the ground frame such that the gun point of agent is exactly facing centre of enemy and the actual angle is the angle the agent is actually making).
- Reward = +1000 for hitting enemy each time
- Reward = -1000 for getting hit by the enemy each time
- Reward = -100000 for losing the game
- Reward = +100 x (agent's health left) for winning the game

As you can see above, this time, the reward function is even more intuitive but is still sparse. The first point in the reward function gives the agent a direct command that keep your gun point on enemy whenever possible. Other points of the reward function are almost same as those in the previous problem.

The algorithm used for training the agent will be the same as the previous problem as only the dimensions of state space and action space have increased, rest all is same as the previous problem. So again, PPO algorithm is used with Mlp policy to train the model. But, due to the fact that the complexity of the problem has increased, the hyperparameters of the algorithm needs to be tuned for getting optimal performance. Hyperparameters in this case are the number of layers of dense neural network, step size, learning rate, number of epochs, number of neurons in each layer, discount factor (gamma), batch size, etc. The hyperparameters are in the following format:

Hyperparameter	**Value**
Clip Range	$0.1\times \alpha$
vf_coeff	1
Entropy coefficient	0.01
Discount Factor (γ)	0.7
Batch Size	32×8
λ	0.95
Number of Epochs	3
Learning Rate	(2.5e-4) × α
Number of steps	128

Now the task is to find the optimal value of α. To do so, the model is trained for different values of α in range [0,1], each for 100k timesteps. Then each model is tested and a curve is plot between the average reward for 25 episodes and the value of α. The curve is shown below in **Chart 3.1**. The optimal value of α comes out to be almost 0, but learning rate becomes 0 also, which is not good, hence for learning rate, $\alpha = 0.7$ is taken and for Clip range, $\alpha = 1.38e - 16$ is taken.

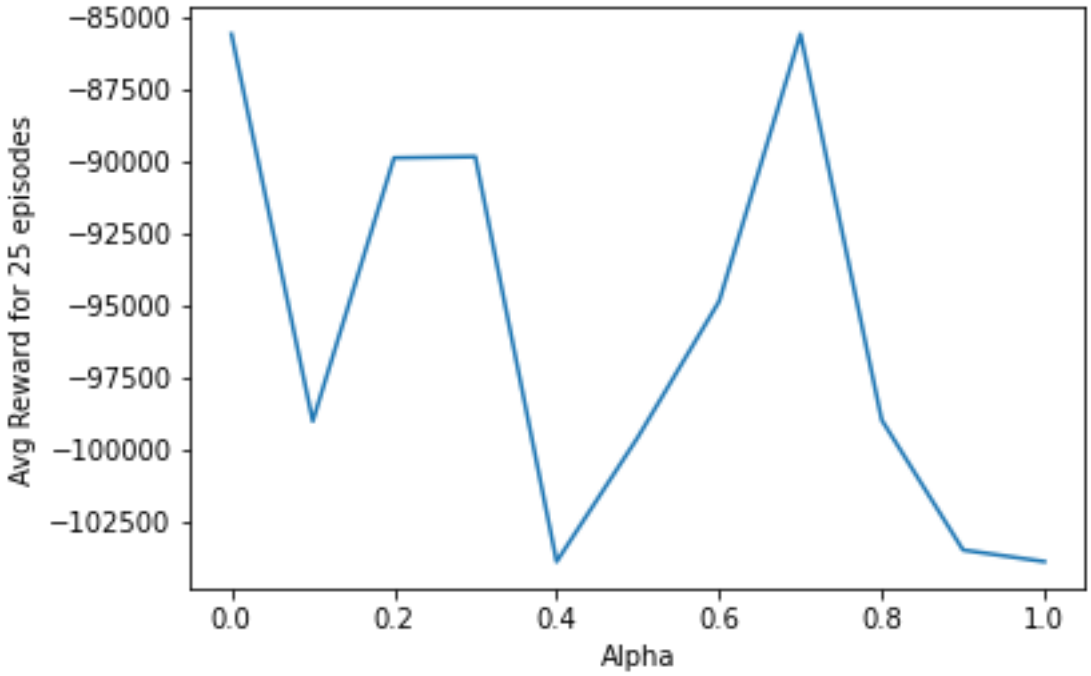

Chart 3.1

The tuned model was trained for 10M timesteps. Although the agent could not perform well against the enemy, but it performed much better as compared to other set of hyperparameters and reward functions. If the model is trained for larger number of timesteps such 50M or 100M timesteps then there is a possibility of agent winning against the enemy. But achieving a good performance from the agent by training it for a lesser number of timesteps is still an open research problem.

4. Two-Dimensional motion of Tanks with Continuous Action Space

There is not much difference in the previous problem and this problem. The only difference in this environment is that here the action space is continuous rather than discrete. Action space vector is now a 3-dimensional vector, where the first 2 dimensions are continuous and denote the velocity and angular velocity respectively. The 3 dimension denotes whether the agent should shoot the bullet or not. The state space is the same (one may want to add more dimensions in the state space such as velocity) as in the previous one. This change may look small but it is a very big modification in the problem. Though the vector is 3 dimensional but now each dimension can assume infinite number of possible values. This change makes it very tough for the agent to learn. Here if you do not have access to a very high-performance computer then it is highly recommended that your reward function is not sparse. Here I would like to introduce a term called 'Reward Shaping'. After the invention of reward shaping method, many continuous tasks have been easily implemented. Reward shaping means designing a continuous reward function rather then discretising the rewards. Let me explain you with an example. Assume that you are training a car whose action space is continuous and it is a one-

dimensional vector, denoting the speed it can give to the car at any instance. The goal of the car is to learn to start from an initial point and stop before the final point given to it as an input.

Designing a discrete reward function:

- Reward = +100 if the car reaches the goal and stops
- Reward = -1000 if the car reaches the goal with a speed greater than the threshold speed.

Designing a continuous reward function:

- Reward = -0.1 × (distance left for end point) × $(\text{velocity of car})^{\left(\text{distance from the endpoint} - \frac{\text{intinal distance from the endpoint}}{2}\right)}$ at every time step.

The above reward is a function of distance and velocity which are both continuous and hence the reward is continuous. But what is the advantage of doing so? The algorithm learns better if it gets a continuous and differentiable reward curve as at any instance it can define the slope of reward with respect to the quantity (or a related quantity) it is changing. The slope gives it a better intuition of how the term it is changing is changing the reward. This is possible in case of only continuous action space as for finding the derivative or plotting a curve of reward w.r.t to action value, you need both the action value and the reward to be continuous. Any one of them being discrete fails this concept.

For this problem, the reward shaping is done in a similar manner as shown above in the example. The reward function is:

- If distance between enemy and agent < firing range:
  - Every time the agent hits the enemy:

    $$\text{Reward} = \left(\frac{\text{agent health}}{100}\right)\left(1-\left(\frac{\text{enemy health}}{100}\right)^{0.4}\right)\times 5+1.5$$

  - Else:

    $$\text{Reward} = -\left(\frac{\text{enemy health}}{100}\right)\left(1-\left(\frac{\text{agent health}}{100}\right)^{0.4}\right)\times 5+1.5$$

- Else:
  - $\text{Reward} = \left(1 - \left(\frac{\text{distance between agent and enemy}}{\text{firing range}}\right)^{0.4}\right) \times e^{\frac{\text{agent health}}{100}}$

- Reward = -100000 for losing the game

One Point to be noted in while doing reward shaping is, despite of the fact that the function should be continuous and differentiable, the reward for the negative terminal state should be awarded a great negative reward. Setting high positive reward for positive terminal state is up to you (and also varies from problem to problem).

The next step is to implement the right algorithm for this problem. There are many algorithms supporting continuous action space but most of them are for continuing task. Many people get confused between the word continuous and continuing. Continuous signifies the nature of the space (continuous or discrete) and continuing signifies the nature of the task (episodic or continuing). Therefore, most algorithms supporting only continuous action space are continuing in nature (i.e., they tend to maximize the average reward and try to achieve the

terminal state as late as possible or never achieve the terminal state (mostly negative)). These algorithms are therefore not suitable to use here as we have an episodic task so we want to achieve the positive terminal state as soon as possible. Not many algorithms support episodic task with continuous action space. So, this is again an open research problem to build optimized algorithms for episodic tasks with continuous action space. Currently, two popular algorithms suitable for this task are PPO (as discussed in previous examples) and A2C (Advantage Actor Critic) algorithm. A2C is the deterministic version of A3C (Asynchronous Advantage Actor Critic) algorithm. Still A2C is better for continuing tasks as compared to episodic tasks. For this problem, both the algorithms were trained for 10M timesteps and it was observed that PPO did much better as compared to A2C and gained more reward but at the same time, the agent trained on A2C moved very smoothly (using the continuous action space very well) while the motion of the agent trained on PPO algorithm was very jerky (didn't use the freedom of the continuous action space well, rather took each value of the action space as a separate discrete value). But overall, although the agent was trained for such a large number of timesteps, it was not able to defeat the enemy.

As you may have now observed that slightly increasing the complexity of the problem, increases the number of parameters and hyperparameters that should be tunned exponentially or at even steeper rate. In this problem, not only selecting the right algorithm is an issue, but tunning its hyperparameters, choosing the right reward function and tunning its parameters are also major issues. You may notice that I have used power of 0.4 in the reward function at multiple places. This was a random number that was chosen between 0 and 1. But changing it to 0.5 or 0.3 may have a great impact on the training of the agent. Finding the best set of hyperparameters, meta parameters and parameters is very tough (or sometimes impossible). It needs very high-performance computing facilities and needless to say, a lot of time to train and test the results.

5. What's Next?

We analysed three different levels of the same problem. Is this the end? No there are numerous things that can be implemented in order to make this simulation more realistic. Now, the question arises that why do we want the simulation to be more complex and realistic and why are we not satisfied with the above 3 levels? The main aim is to implement this technology in real life, i.e., in real tanks. This is a very advanced task. First, we need to implement it in a simulation as it would be very dangerous and expensive to train the tanks directly in the real world. If the simulation is not realistic, then the model trained will not work properly in real life scenarios. Therefore, the primary aim is to make a robust simulation that is very similar to real life scenario and train the agent in that environment. After the agent is trained successfully in that environment (which is a very complex and time taking task in itself), we will deploy the model into real tanks with some noise added. The simulation, despite of being very realistic, will never match the real-life scenario completely and therefore we add some noise so that the agent performs well in unusual real-life scenarios. Methods for noise addition is a very later stage and is out of the scope of this paper.

Discussing further on the primary aim mentioned above, as future prospects, there are many things that can be added to problem discussed in this paper. The first and the foremost thing is

to improve the environment. The environment dynamics can be made more complex by adding friction and other physical parameters. The action space of the tanks can be more complex and the movement of the tank can be more realistic. In the above-mentioned levels, the state space is easily available to the agent and almost all the information is provided about the state. But in real life scenarios, this data is captured by different sensors (e.g., Lidar, proximity sensor, camera captures, etc.) and the working range of these sensors is also limited. So, the state space can be more complex where direct data such as velocity, angle is not known, but the readings of the sensors are known and the model has to train on such kind of data.

Making the environment more complex, makes the problem tougher from the perspective of reinforcement learning. More complex environment needs more robust policies and algorithms on which the agent can be trained. As mentioned above, a good algorithm for episodic task with large continuous state and action space is an open research problem. Reward function designing (and reward shaping) and hyperparameter tunning are very big domains in themselves. Finding the optimal reward function and set of hyperparameters can entirely change the performance of the agent. Now-a-days, due to advancement in this technology, the agents are trained without a reward function. The agents are given very basic feedback at each timestep of their training and an optimal reward function is modelled during the training itself. This strategy is called reward modelling. This is a very new domain and many researches are being done on it. This strategy is very useful at places where defining a good reward function is tough.

After all these improvements, there is still one crucial thing that is not discussed yet. It is the behaviour of the enemy. If the behaviour of the enemy is predefined, the agent will learn that behaviour and will not perform well on other behaviours. Therefore, we need a way in which the enemy tank also performs equally good as the agent or slightly better than agent. Also, the actions taken by the tank should not be a predefined set of tasks. This is achievable if the enemy also learns with the agent using the same reward function or a different reward function. This can be done using multiagent reinforcement learning. Here the enemy and the agent are treated as two different agents and are trained together in a competitive way. Agent 1's task is to defeat agent 2 and vice versa. Multiagent RL is also a new domain and many researches are conducted worldwide to optimize the performance of the agents. There are mainly three types of multiagent RL methods: competitive, collaborative and mixed. In our problem, we need to use competitive multiagent method. Mixed method can also be implemented where two agents collaborate with each other to defeat the third agent. But we are currently dealing with 1 vs 1 problem. As mentioned above, multiagent RL is a new domain, hence, there is a specific domain which needs much research and is also related to our problem. Our problem will come under episodic competitive multiagent RL with continuous action space. There are currently very few algorithms that can be implemented on such kind of RL problems and hence it is again an open research problem.

6. Conclusion

We analysed the problem of training a pixel tank and applied reinforcement learning in 3 different levels. Every level has a different approach. As the RL problem gets more complex, the training of the agent becomes tougher. As the action space and state space increases, the

need for parameter tuning and reward function optimization increases. In continuous action space problems, shaping the reward function is greatly advised and parameter tunning becomes more sensitive. Small changes in the hyperparameters can change entire functioning of the agent. Reinforcement Learning is a new domain, especially multiagent RL and large state and action space RL. There is a great scope of research in this domain. One such RL domain which is connected with the problem discussed in this paper is finding an optimal RL algorithm for episodic competitive multiagent RL with continuous action space.

7. Acknowledgements

I am very grateful to Professor Madhu Vadali (Professor in discipline of Electrical and Mechanical engineering at IIT Gandhinagar) and Suraj Borate (PhD fellow at IIT Gandhinagar) for their valuable insights.

8. Resources

Here is the GitHub Repository link for your reference and better understanding of this papers. You can access the code files and trained models in the GitHub Repository for the methods discussed in this paper.

I highly recommend you to refer the book named 'Reinforcement Learning: An Introduction' written by Richard S. Sutton and Andrew G. Barto.

References

- https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf
- https://arxiv.org/pdf/1911.10635.pdf
- https://arxiv.org/pdf/1707.06347.pdf
- https://blog.paperspace.com/creating-custom-environments-openai-gym/
- https://stable-baselines3.readthedocs.io/en/master/index.html#
- https://github.com/openai/gym/tree/master/gym
- https://arxiv.org/pdf/1909.07528.pdf
- https://github.com/openai/multi-agent-emergence-environments/blob/master/randomized_uncertain_social_preferences/rusp/env_prisoners_buddy.py
- https://youtu.be/PYylPRX6z4Q
- https://youtu.be/kopoLzvh5jY
- https://openai.com/blog/emergent-tool-use/
- https://towardsdatascience.com/what-is-deep-reinforcement-learning-de5b7aa65778
- https://towardsdatascience.com/getting-an-ai-to-play-atari-pong-with-deep-reinforcement-learning-47b0c56e78ae
- https://youtu.be/93M1l_nrhpQ
- https://youtu.be/Q-__8Xw9KTM
- https://github.com/philtabor/Youtube-Code-Repository/blob/master/ReinforcementLearning/Fundamentals/gridworld.py
- https://www.coursera.org/specializations/reinforcement-learning