Deep Reinforcement Learning: Q-Agents on Custom Environments for CSL7580

Jaisidh Singh (B20AI015)
singh.118@iitj.ac.in

Susim Mukul Roy (B20AI043)
roy.10@iitj.ac.in

Abstract

An important task of artificial intelligence is to understand an agent's environment via interactions, and in turn provide it with information leveraged to take meaningful decisions. These decisions are sequences of actions selected by observing and interacting with sequences of states. A powerful approach devised for sequential decision making is reinforcement learning, which puts an agent in an environment. The agent's interaction with the environment is parameterised by an objective to maximise a reward signal, to learn profitable decisions. In this technical project report, we present a reinforcement learning agent, following the deep Q-learning algorithm, which interacts with the given custom game environment. We present the results of the agent and the thorough analyses of our design and observations.

We would like to present an acknowledgement of the faculty and staff of CSL7580, and to state our gratitude for their teaching and this project assignment, which was a joy to work on. Further, it greatly strengthened our understanding of the course material by practically applying it here. We present the code for this work at https://github.com/jaisidhsingh/Deep-Q-Learning

1 Introduction

Reinforcement learning [1] (RL) is the process of teaching an agent to make decisions, by placing it in an interactive environment, where it observes states and takes actions. The objective of an RL agent is to maximise a reward function, which is specifically designed to teach it relevant actions for the task. We describe each of the components mentioned above in detail and with respect to our provided project environment, a snapshot of which is shown in Figure 1.

1.1 Agent

The player object is the agent in our setting. An agent interacts with the environment to learn from it and accomplish particular goals, in the form of maximising its rewards, by taking profitable decisions. Here, the player, or agent

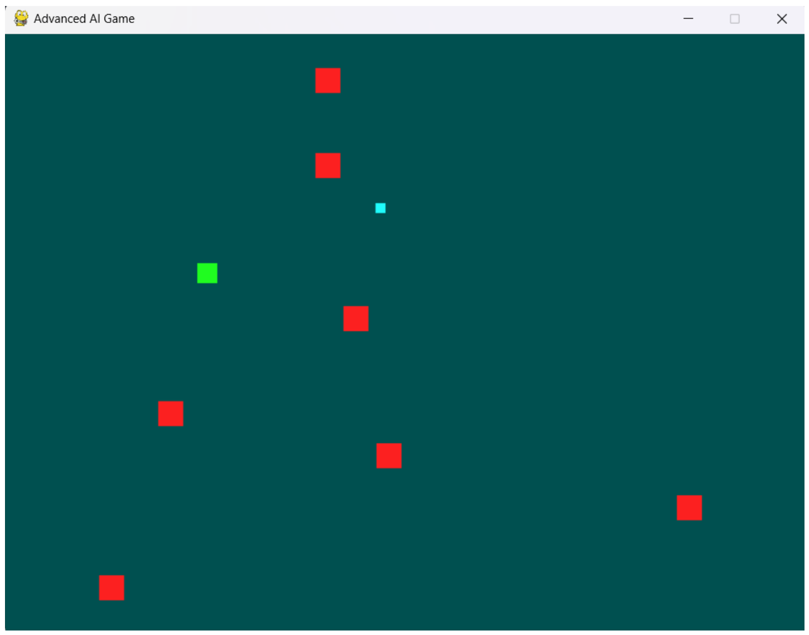

Figure 1: A snapshot of the game simulation given as the RL environment for the project

(green box) moves in the game boundary and has the objective to reach its goal, (cyan box) while avoiding enemies (red boxes). Note, that the enemies are not learnable agents, *i.e.*, their actions are randomly specified in our simulation.

1.2 Environment

An environment is a system which the agent interacts with. Here it is in the form of a game or simulation. The environment defines all rules, the state space, the action space and the reward function. In our setting, the game simulation is the environment, which defines the player and where it can go, its enemies and their actions and locations, as well as the goal state. It contains additional variables governing the game like friction, acceleration, restitution of bounce, etc. Note that our environment is fully observable, *i.e.*, we know exactly in which state we are in. It is also Markov in nature, as the future in our case is independent of the past, given the present, *i.e.*, only the current state shall affect the future state.

1.3 Actions

The agent interacts with the environment via actions. The actions that we can take in our case are locomotive, *i.e.*, we can only go left, right, up, down, or remain stationary. Our enemies also have the same action space.

1.4 Observations

The results of an agent's interaction with its environment is the observation. The observation provides the agent with a partial view of the state, which it must use to make decisions about its actions. Here, our observations reports only if nothing happened, if an enemy attacked, or if we reached our goal.

1.5 Reward

The reward function is what provides an objective to the agent. It guides it's learning by incentivizing, or penalizing its decisions, based on the specified task in the environment. This function yields a reward at each time step to the agent, *i.e.*, the output of the reward function, or simply the reward at time step t, is r_t. We design a custom reward function for our environment, which utilizes the internal nuances of the simulation as a heuristic. In particular, we incentivize and penalize our agent using *the straight-line distance of the agent from its goal*, for different agendas, which we describe in detail in **Section 3.1**.

2 Related Works

As described in [5], a deep Q network(DQN) [3] is a multi-layered neural network that for a given states 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 action space containing m actions, the neural network is a function from $\mathbb{R}^n$ to $\mathbb{R}^m$.Two important ingredients of the DQN algorithm are the use of a target network, and the use of experience replay. Another kind of Reinforcement Learning algorithm is Passive Learning[4]. In this type of learning, the agent's policy is fixed which means that it is told what to do. As the goal of the agent is to evaluate how good an optimal policy is, the agent needs to learn the expected utility $U^\pi(s)$ for each state s. This can be done in three ways; Direct Utility Estimation, Adaptive Dynamic Programming(ADP) and Temporal Difference Learning (TD). Two more methods include REINFORCE, which is a Monte Carlo variant of a policy gradient algorithm in reinforcement learning. The agent collects samples of an episode using its current policy, and uses it to update the policy parameter θ and A2C(Actor-Critic Method) [2] where the principal idea is to split the model into two: one for computing an action based on a state and another one to produce the Q values of the action.

3 Methodology

In this section, we describe the process of implementing a deep Q-learning agent for the given game environment. We begin with specifying our reward function for the same.

3.1 Custom Reward Function

As mentioned above, the reward function presents guidance to the agent's decision making process by imposing penalties on irrelevant or possibly harmful actions, or by presenting incentives for profitable actions. Our straight-line heuristic based reward r_t, for time step t is given by:

$$R_t = R_{idle} + R_{goal} + R_{enemy} \tag{1}$$

where R_{idle} represents the reward incurred by remaining idle, which can be thought of a combination of the agent being stationary and also of not progressing towards its goal, given clear and safe paths. The term R_{goal} represents the reward (or incentive) the agent receives by moving closer to its goal, and also by achieving it. Lastly, R_{enemy} is the reward (or penalty) received when the agent intersects (or simply, is killed by) an enemy. We describe each term as follows.

$$R_{idle} = -\lambda_1 d(p, g) + c_1 \tag{2}$$

$$R_{goal} = \lambda_2 \frac{1}{d(p, g) + \epsilon} \tag{3}$$

$$R_{enemy} = \begin{cases} c_2, & \text{if } \exists e \in \text{Enemies} \mid d(p, e) = 0 \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

where $d(\cdot, \cdot)$ represents the distance between 2 entities, and λ_1 and λ_2 are constant weights for the distance based terms. Further, c_1, c_2 are constant terms, and ϵ is added for avoiding mathematical errors, in the case that the player reaches the goal, *i.e.*, $d(p, g)$ becomes 0. The values for each of these terms can be found in the implementation details section, **Section** 3.3. The intuition

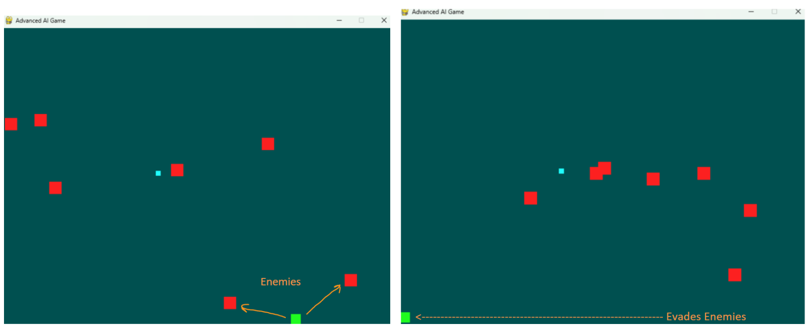

Figure 2: Snapshots of simulations showing the agent evading enemies from an initial position.

behind the formulation of the penalty for idling, R_{idle}, is that, if the agent has an opportunity to approach the goal and does not, it should incur some penalty. This penalty should reduce when it approaches the goal to allow the R_{goal} incentive to dominate and push it towards the goal. The values of the constant terms in the equations are set after empirical experiments, in order to practically observe the point at which desirable behaviour is seen.

If the idling penalty is too high, it encourages the model to be less wary of the enemies, and results in greater intersections with enemies. If the it is too low, then the agent feels too "safe" not approaching its goal, and discards the incentive given by R_{goal}. Further, the distance based idling penalty ensures that the penalty incurred while idling close to the goal is not too less, by carefully experimenting with values of λ_1 and λ_2. The value of λ_2 also plays a role as if the incentive of approaching the goal is too large, it may dominate the penalty imposed when the agent is close to the goal, but still idling, which is undesirable.

Note: Section 1 and Section 3.1, together answer "Part 1" mentioned in the assignment sheet.

3.2 Deep Q-learning Algorithm

For learning optimal learning policies in a Markov Decision Process, Q-learning is used to learn a Q function which estimates correct utilities for a particular action. Deep Q-learning involves training a neural network to learn this Q function, which is our chosen algorithm, as our environment is also Markov in nature (shown above). The Q function, also called the Q-table considers a state s and an action a, and it given by:

$$Q(s, a) = R(s, a) + \gamma \max_{a'} Q(s, a') \tag{5}$$

We formulate the state s of the agent as a vector of 54 dimensions, which consider the player's position, size, kinematic factors, the goal size, goal position, and the positions and velocities of the enemies. The Q network yields values for all actions by observing these values as its state input. This function is proven to converge, and each update $R(s, a) + \gamma \max_{a'} Q(s, a')$ is a closer, more accurate estimate of the true Q function representing the correct utilities, hence, when using a neural network to model the Q function, we use a target value of $R_t + \gamma \max_{a'} Q(s, a')$ formulate the loss as

$$L_{Qlearning} = \Big[Q(s, a) - (R_t + \gamma \max_{a'} Q(s, a'))\Big]^2 . \tag{6}$$

The second term in the loss, is weighted by a factor of η which is called the learning rate in practical implementations, which is also what we follow. The agent is placed in the environment for a certain number of episodes, where it observes an initial state. The Q network, or the neural network approximating the Q function, gives an action using the state as input. The action is used to update the environment to a new state, which is used with the previous to compute and backpropagate the loss. In the following sections, we present the details of our implementation and results, and discuss them using ablation studies.

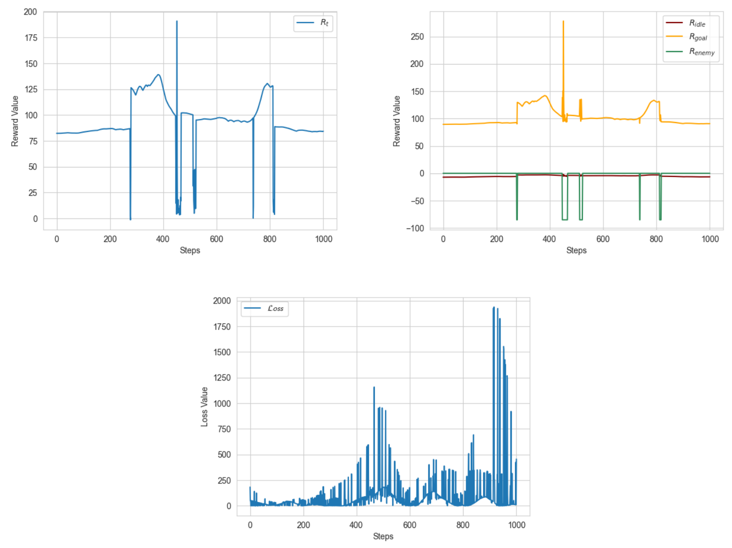

Figure 3: The landscape of the functions that we optimize during our RL agent training process, over 100 training steps.

3.3 Implementation Details

We set our game seed as a unique number of 22019043. The constants defined for the reward formulation in Section 3.1 are described here. The values of λ_1, λ_2 are $-6e^{-3}$ and $8e^3$ respectively, and the values for c_1 and c_2 are -2 and -85 respectively. The learning rate in our experiments is kept as $1e^{-9}$, using Stochastic Gradient Descent (SGD) without momentum as our optimizer.

We start the model off in an initial state in the top left corner of the game simulation and train for 1000 episodes, by propagating the loss every episode. Further, the agent is encouraged to explore so as to learn new information, by generating a random number at every step, and using the learnt Q network only if the random number is above a threshold of 0.5. Lastly, our Q network is composed of 6 fully connected layers with $164,485$ parameters.

Note: Section 3 answers "Part 2" mentioned in the assignment sheet.

4 Results

We train and evaluate our model using the given boilerplate and achieve approximately 12 wins and 1685 deaths, (averaged over 10 separate evaluation runs) using this setting. We also present snapshots of inference on the game with our model, to show its behaviour, in Figure 2. Further, Figure 3 reports the landscapes of the reward and loss functions during a training run on our game simulation. To supplement the extensiveness of our work, we also provide ablation studies and discussions in the next section.

5 Ablation Studies and Discussion

We present ablation studies and discussions on the observations in this section. In order to explain the effect of the variables in the game, we present an ablation study on how the results would be affected if we change the following game constants: Game Seed, Game Width and Game Height, Goal Size, Enemy Count, Game Friction and FPS. The results observed were as follows:

1. **Game Seed**: We don't observe major changes on changing the game seed, as the simulation seems to run in a fairly uniform manner. However, it is possible that a model trained extensively on a particular seed value can show perturbed results in the event of a new seed. Our model faces unstable gradients during its training, which is typical of deep Q-learning agents, hence it does not overfit to a particular seed state.

2. **Enemy Count**: On changing the enemy count, the number of wins should decrease. This is because we trained the model where the input states have been created by taking into account the number of enemies. Hence, if we change this term, and use the model for inference only, then theoretically our model hasn't trained on those number of factors and will produce erratic results. In other words, it might sometimes give high win rate and sometimes very low.

3. **Goal Size**: On increasing the goal size, our trained model should perform better and vice-versa as now it has a higher chance of colliding with the goal. Since our model has learned to collide successfully with smaller goal size (as one of our training state is the height and width of the agent), hence it would still try to be precise with respect to the old smaller goal size which would make the task easier.

4. **FPS**: On increasing the FPS, we should observe an increase in the win rate of the agent and vice-versa. This could be because the RL agent can be fed with more up-to-date information about the game environment, allowing it to make more informed and accurate decisions.

5. **Game Width, Game Height**: There shouldn't be any significant change in changing these factors. This is because we are not training our RL model based on these factors and our input state is not dependent on the play area. The model learns to understand distance between entities as we give it coordinates of the entities, however, the bounds on these have no effect on the model.

6. **Friction**: The no. of wins increases on increasing the friction on an average as compared to the default friction value. This is because an increased friction reduces the momentum of the agent more and helps in smoother loss backpropagation as the agent is getting trained.

Note: Section 5 answers “Part 3” mentioned in the assignment sheet.

References

[1] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. A brief survey of deep reinforcement learning. *CoRR*, abs/1708.05866, 2017.

[2] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013.

[4] G. Ostrovski, P. S. Castro, and W. Dabney. The difficulty of passive learning in deep reinforcement learning. *CoRR*, abs/2110.14020, 2021.

[5] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.