Deep Reinforcement Learning for balancing CartPole-v1

Benjamin Benteke[1], **Mikhaël P. Kibinda-Moukengue**[1], **Arnaud Watusadisi Mavakala**[1], **Jeanette Nyirahakizimana**[1], **Maram A. Abdelrahman Mohamed**[1]
[1]African Masters in Machine Intelligence (AMMI), AIMS Senegal
{bbenteke, mmoukengue, amavakala, jnyirahakizimana, mmohamed}@aimsammi.org

Abstract

In this paper, some deep reinforcement learning algorithms are investigated, especially Deep Q-network (DQN) and Dueling Deep Q-network (Dueling DQN) for CartPole-v1. We apply DQN and Dueling DQN to CartPole v1 with no modification of the rewards[1]. We detect the instability of DQN agent, and we apply Early stopping to address it. We investigate Neural network architecture, Replay Buffer size and Target network update frequency. We notice overestimation bias is also one of the main causes of instability. To improve DQN, we decide to investigate Dueling DQN, in order to improve DQN and reaches a higher reward during test (environment) than DQN.

Keywords: Reinforcement Learning, CartPole-v1, Q-learning, Deep Q-Network, Dueling Deep Q-Network.

1 Introduction

Reinforcement learning (RL) is an area of machine learning in which an agent learns by interacting with its environment. In particular, reward signals are provided to the agent so it can understand and update its performance accordingly [1, 4, 10]. Learning to control agents directly from high-dimensional sensory inputs like vision and speech, which are some of the long-standing challenges of RL. Most of the applications of RL are in the self-driving cars, healthcare, industry automation, trading and finance, natural language processing, engineering, recommendation engine, gaming, robotics manipulation, etc.

Deep neural networks lead to impressive successes, e.g. in image classification, they can reliably classify 100 objects classes. These models require a large amount of labeled training data in order to succeed. RL algorithms, on the other hand, must be able to learn from a scalar reward signal that is most of the case sparse, noisy and delayed.

Another issue is that, most deep learning algorithms assume the data samples to be independently identically distributed (iid), while in RL one typically encounters sequences of highly correlated states. In RL, the data distribution changes as the algorithm learns new behaviours, which can be problematic for deep learning methods that assume a fixed underlying distribution.

In this paper, we will provide explanations, workings of two well-known RL methods, namely DQN and Dueling DQN to control a CartPole system. The objective of this paper is to provide clear, precise and intuitive explanations on the implementation of several RL concepts. Some of these concepts are the Q-learning algorithm, DQN, Replay Buffer, etc.

[1]Github : https://github.com/benjaminbenteke/Deep_RL_Project

2 Background

2.1 Reinforcement Learning Concepts

Consider tasks in which an agent interacts with an environment ξ, in a sequence of actions, observations and rewards. An agent is thus the one who lives in a world in which he interacts that we called the environment and who takes decisions based on rewards and punishments. At each time step, the agent chooses an action from the set of legal actions in the game, $A = \{a_1, ..., a_K\}$. We define an action $a \in A$, as a method of the agent allowing him to interact and to modify his environment, and thus to pass from a state to another. A model of the environment, on the other hand, is an internal, and normally simplified, representation of the environment that is used by agents to try to predict what might happen in the future.

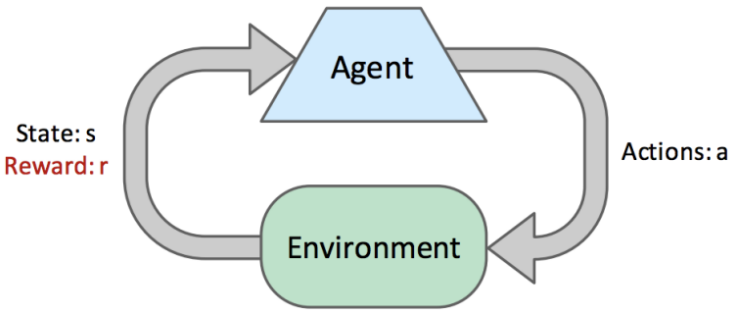


Figure 1: Reinforcement Learning Problem [source]

As we can see in figure 1, a state or experience $s \in \mathcal{S}$ is the information used to determine what happens in the next one. The reward function $r : S \times A \longrightarrow \mathbb{R}$, on the other hand, is a function that assigns a value to each state the agent can be in. Reinforcement learning agents are fundamentally reward driven, so the reward function is very important. The ultimate goal of any reinforcement learning agent is to maximize its accumulated reward over time, typically by trying to reach, through a particular sequence of actions, the states of the environment that offer the highest reward. There is a function that tells the agent how to behave at a given time, called by the policy $\pi : S \times A \longrightarrow [0, 1]$. It is essentially a function that takes the information detected in the environment and produces an action to execute.

The objective is to find the policy with the highest expected reward and thus to maximize the value function V^π (the total reward we receive overtime). Following a π policy, the value function is defined as follows:

$$V^\pi(s_t) = \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...],$$

where $\gamma \in [0, 1]$ is the scalar discount factor.

The value function presented above, is an estimate of the total amount of reward the agent expects to accumulate in the future. While a reward function is usually static and tied to a specific environment, a value function is normally updated as the agent explores the environment. This updating process is a key element of most reinforcement learning algorithms. Value functions can be mapped from states or state-action pairs. A state-action pair $S \times A$ is a pair of each distinct state and each action that can be taken from that state.

2.2 Q-learning algorithm

Q-learning is a model-free, off-policy and bootstrap RL algorithm that aims to find the best action to take based on the current state. The agent represents the RL algorithm. RL has various algorithms such as Q-learning, SARSA, DQN, DDPG, etc. In this study we focused on the Q-learning algorithm which is based on the well-known Bellman Equation:

$$V^{\pi}(s) = \mathbb{E}\Big[\sum_{i\geq 0}\gamma^{i} r_{t+i} | s_t = s\Big]$$

The above equation can be re-written in the form of Q-value as:

$$Q^{\pi}(s,a) = \mathbb{E}\Big[\sum_{i\geq 0}\gamma^{i} r_{t+i} | s_t = s, a_t = a\Big]$$

The goal here is to maximize the Q-value. The unique solution to the optimal Bellman equation is:

$$Q^{*}(s,a) = \mathbb{E}[r_{t+1}|s_t = s, a_t = a] + \gamma \mathbb{E}_{s_{t+1}}[\max_{a\prime} Q^{*}(s_{t+1}, a\prime)|s_t = s, a_t = a]$$

The idea of the algorithm is to find the Q-values from the given sample s_t, a_t, r_{t+1}, s_{t+1} that are closer to satisfying the Bellman equation. To do this, one use the off-policy method (samples don't have to be from the optimal policy).

We focused on Q-learning, because it is a model-free meaning that we do not need the exact dynamic of the environment which is not available for most of the cases of RL real problems.

Exploration-exploitation trade-off

In the beginning, the agent has no idea about the environment. it is more likely to explore new things than to exploit its knowledge because it has no knowledge. The exploitation refers to give knowledge to the agent by using greedy policy, which is the selection of the action that has higher reward in the future whereas the exploration refers to give the agent the chance to select randomly an action. This lead to a trade-off, because we want to balance exploration and exploitation. To do so, we used $\epsilon-$ greedy policy, which gives ϵ probability of random selection and $1-\epsilon$ probability for performing the greedy policy. We decrease ϵ over timestep and we call this new value threshold. $\epsilon-$ greedy policy, tells us to sample a variable uniformly over $[0, 1]$, if the variable is smaller than the threshold, then the agent will explore the environment. Otherwise, he will exploit his knowledge. And the exploration will be made for example, by a Neural network.

The pseudo code of the Q-learning algorithm is:

Algorithm 1 Pseudo-code of Q-learning

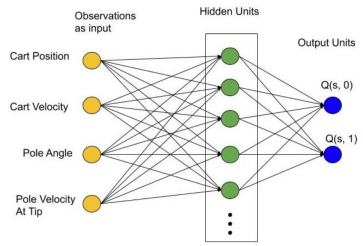
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal-state, .) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a \text{Q(S}', a) - Q(S,A)]$
    $S \leftarrow S'$;
until S is terminal

3 Deep Q-Network and Dueling Deep Q-Network

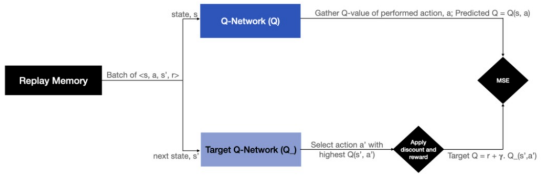
In the section we presented some popular Deep Reinforcement learning algorithms that we used in this project. We present a brief description of DQN and Dueling DQN.

3.1 Deep Q-Network

Previously, we presented Q-learning [3] which aims to find the best approximation of the action-state function $Q(s,a)$. Now, we will talk about DQN [7,8] which is the advanced version of Q-learning. By Universal approximation theorem statement, one uses Neural Networks to approximate the Q-value function by $Q(s,a;\theta) \approx Q(s.a)$, where $Q(s,a;\theta)$ is the Neural network output with parameter θ that we call Q-network.



(a) Neural Network for CartPole [source]



(b) Replay Buffer [source]

Figure 2: The main components of DQN

DQN algorithm has two main components: Neural network and Replay Buffer. As the figure 2 shows, the input of the neural network is the state vector s and the output is $Q(s, a)$, for all possible actions a. This is a supervised learning, to train a Neural Network we need a ground-truth. But unfortunately, we mentioned most of the RL problems do not have the exact dynamic of the environment which represents our ground-truth. Lack of exact dynamic of the environment stands for, we do not know exactly the selection of an action in a state even if the dynamics are known, we need to interact the agent with the environment for a sufficiently long time (until the episodes or number of games).

In DQN, we have a concept of Experience Replay Buffer (Replay Buffer) which tracks the agent's interactions with the environment to provide a pool of data samples from which to learn. The tuple (s, a, r, s') containing state, action, reward and next state after each iteration, called sars represent pooled data. These tuples will be stored in Replay Buffer and once a sufficient number of tuples are stored, we can train the DQN by using a batch of samples selected randomly from the replay memory. The replay buffer can store new experiences (tuples) and also the ability to sample past experiences.The Bellman Equation of action-value $Q(s, a)$ is given by:

$$Q^*(s, a) = \mathbb{E}_{(s,a)\sim P(s,a)} \left[R(s, a) + \max_{a'} Q^* \left(s', a'; \theta_{i-1} | s, a \right) \right] \tag{1}$$

where $R(s, a)$ is the reward for the current state-action pair (s, a) obtained from the environment and $Q^* \left(s', a'; \theta_{i-1} \right)$ is the Q-value for the next state from the target network which is an exact copy of Q-network. The weights of this target networks are held fixed for a fixed number of training steps after which these are updated with the weights of Q-network.

The idea is to bring close the target network close to the Q-Network. In this case, our target network becomes ground truth and Q-network our predicted.

The lose function is given by:

$$L_i(\theta_i) = \mathbb{E}_{s' \in S} \left[\left(Q^*(s, a) - Q(s, a; \theta_{i-1}) \right)^2 \right] \tag{2}$$

The experiment's repetition memory and target network are thus decisive in enabling the Neural Network to learn tasks via RL. Their disadvantage is that they slow down the learning considerably and thus increase the complexity of the sample. In addition, the DQN has stability problems because the same network may not converge in the same way in different executions.

The pseudo code of the DQN algorithm is:

Algorithm 2 Pseudo-code of DQN with experience replay

Initialize replay memory $\mathcal{D}$ to capacity $\mathcal{N}$,
Initialize action-value function Q with random weights
for episode $= 1, M$ **do** :
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**:
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action at in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal} \quad \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal} \quad \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

3.2 Dueling Deep Q-Network

To get a deeper understanding of Dueling Network, let's recap some points from Double DQN. The training process of the DQN often leads to some issues especially the bias problem, that means the vanilla DQN tends to overestimate rewards in noisy environments. The overestimation can occur during acting, meaning that the argmax will always select the best action, even if the Q-values is roughly the same to another action and learning meaning that the target network is formed using a maximum over all possible next actions. The overestimation is one of the causes of DQN instability which means, the agent was able to solve the problem (By reaching the expected rewards for the game), but after solving it, the agent often completely forgot what it has learnt and collapsed to a poor policies.

Double DQN refers to the use of two neural networks to learn and predict which action to take. The use of the two different networks breaks the moving target problem, the Q-network and the target network are essentially trained on different samples. Thanks to this process, any bias due to the environmental randomness should be smoothed out.

Dueling DQN is generally used to provide an additional enhancement to Double DQN while avoiding or fixing the two above mentioned problems encountered by the DQN.

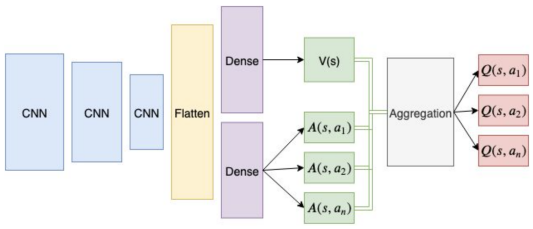


Figure 3: Dueling Learning architecture [source]

In the Dueling DQN architecture (figure 3), the evaluation of the Q function is implicitly focused on the computation of two quantities: the value of being in state s, $V(s)$, and the advantage of taking action a in state s, $A(s, a)$. The figure 3 is divided into two parts, the first part is common with CNN architecture, and the second part contains two separate densely connected layers which correspond to the value and n expressions of the advantage function respectively, followed by the aggregation layers to produce Q values estimations for each possible action in state s.

The Q-values in the Dueling DQN can be written as:

$$Q(s,a) = V(s) + A(s,a) \tag{3}$$

However, the neural network cannot be trained on the sum of the value and advantage functions. This is called the problem of identifiability. This problem can be fixed by forcing the highest Q-value to be equal to the value V. To do so, (3) can be re-written as:

$$Q(s,a) = V(s) + \left(A(s,a) - \max_{a' \in |\mathbf{A}|} A(s,a') \right) \tag{4}$$

Alternatively, as used in Wang et al. [9] experiments, (4) can also be written as:

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{\|a\|} \sum_{a'} A(s,a') \tag{5}$$

4 Experimental Results

In this section, we trained our DQN and Dueling DQN agent in *CartPole-v1* environment, provided by OpenAI Gym [5], we trained and evaluated our agents in this real environment without rewards modification. This section is divided into two main parts[2].

4.1 Game Description

The CartPole-v1 environment consists of a pole that moves along a frictionless track. The system is controlled by applying a force of $+1$ or -1 to the cart. The state space is a four-dimensional vector consists of: cart position $x \in]-4.8m, 4.8m[$, cart velocity $v \in]-\infty, \infty[$, pole angle $\theta \in]-41.8°, 41.8°[$ and pole angular velocity $w \in]-\infty, \infty[$. The figure 4, shows the cart in black color and the vertical bar attached to the cart using passive joint, and the cart can move left or right. The Cart-pole's state on the left is balanced whereas for the right is unbalanced.

Figure 4: CartPole-1 System: balanced state (left) and unbalanced state (right) [6]

In the action space, the agent can push the cart to the left or push it to the right by applying respectively a force of -1 or 1. The agent receives a reward of 1 for each timestep. The simulation ends if: (1) if the pole angle is more than $\pm 15°$ from the vertical axis, or (2) the cart position is more than ± 2.4 cm from the centre, or (3) the episode length is greater than 500.

The problem is to prevent the vertical bar from falling by moving the car left or right (these represent the action space). To solve the problem [2], the agent needs to receive an average total reward greater or equal to 475 over 100 consecutive episodes.

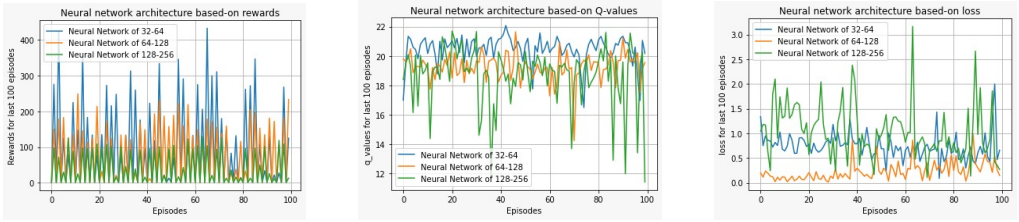
[2]Github : https://github.com/benjaminbenteke/Deep_RL_Project

4.2 Training process

In these experiments, we used the Adam algorithm with mini-batches of size 32 and Huber loss as metric. The behavior policy during training was $\epsilon-$greedy with ϵ reduced progressively from 0.5 to 0.01 over the 10000 of episodes as the agent becomes more confident at estimating $Q-$values, and fixed at 0.01. The learning rate, number of episodes, and gamma are set to 0.001, 10000, and 0.99 respectively. We attempted to tune them, and their best values will be provided.

From the previous section, we pointed out about the instability of DQN during training. So, to address it, we instigated some of causes among them, we have Replay Buffer size, Neural network architecture, and Target network update frequency. And we attempted to use some of techniques that prevent them. To address the overestimation bias, we used Dueling DQN agent.

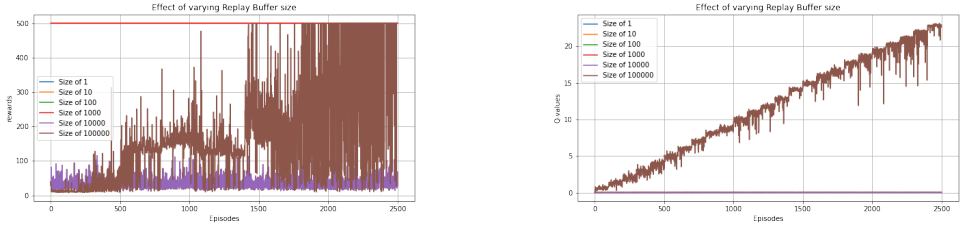
As in Cart-Pole problem, the state vector is four dimensional which is small. So we used a Neural network with two hidden layers. The figure 5 shows that the best architecture is the one with $(32, 64)$[3].



(a) Rewards values per episode (b) Q-values per episode (c) Loss values episode

Figure 5: Neural Network architecture effect

The replay Buffer plays important role, so we need to use the right value. The figure 7 shows that the Q-values or Rewards are higher even if the agent is unstable. So, we set the value to 100000.

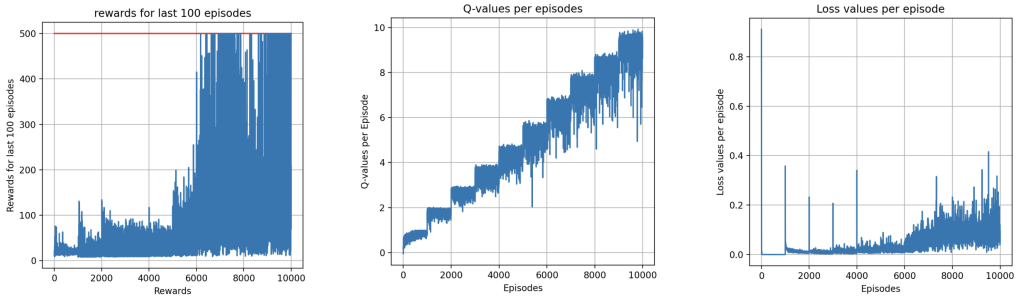


(a) Rewards values episode (b) Q-values episode

Figure 6: Replay Buffer size effect

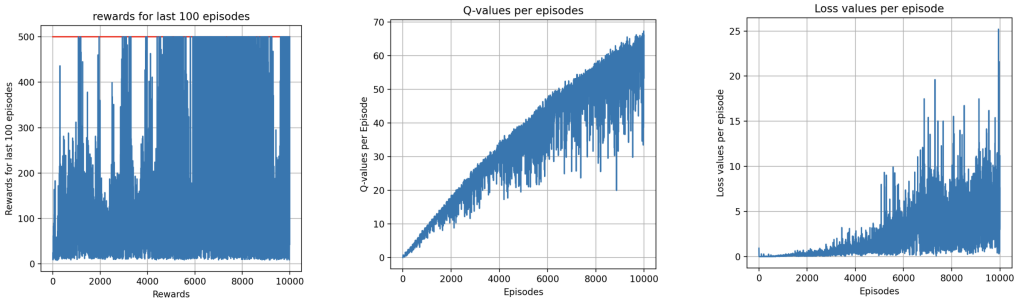
The target network update frequency is crucial because it is not good to update it frequently, so the figure shows that, when we update the target network every 100 steps, the Q-values leave 10 to 60. This gap is significant and brings a good performance.

[3]Github : https://github.com/benjaminbenteke/Deep_RL_Project



(a) Rewards values per episode (b) Q-values per episode (c) Loss values per episode

Figure 7: Target network update frequency effect



(a) Rewards values per episode (b) Q-values episode (c) Loss values episode

Figure 8: Target network update frequency effect

Training the models too long they always end up catastrophically forgetting, this is instability. Early stopping is a potential solution to the stability problem. Rather than training forever, once a good performance has been achieved training should stop. Cart-Pole-v1 is considered solved if average reward is ≥ 475 over 500 and 475 is the threshold. In this experiment, we used Early stopping to address instability of DQN. The figure 9 shows that, with Early stopping, the Cart-Pole v 1 Problem is solved in about 7800.

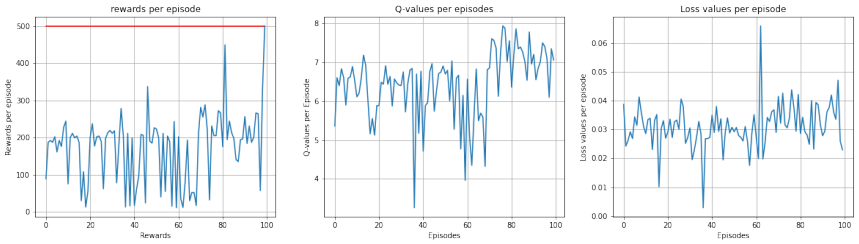


Figure 9: DQN with Early stopping results

At the test time we succeeded to move the cart from left to right with a reward of 210. This need an improvement. To do so, we used Dueling DQN. The figure 10, shows the improvement of DQN by Dueling. By using Dueling, at test time the agent reached 401 rewards, this is an improvement of DQN. The figure 10 shows[4].

[4]Github : https://github.com/benjaminbenteke/Deep_RL_Project

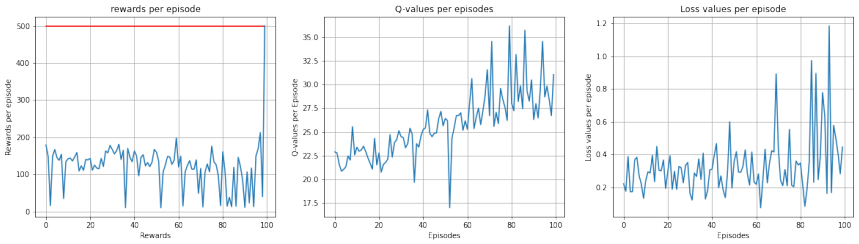


Figure 10: Dueling DQN results

5 Conclusions

In this paper, the tasks where to balance CartPole v 1, we have applied some deep RL algorithms, especially DQN and Dueling DQN to achieve this goal. We used Dueling DQN in order to improve DQN even though both solve the problem. We provided explanations about these algorithms and CartPole v1. On way to extend this work is to use other deep RL algorithms and investigate on their improvements.

References

[1] Andrew G Barto and Richard S Sutton. Reinforcement learning. *Handbook of brain theory and neural networks*, pages 804–809, 1995.

[2] Leonardo Lucio Custode and Giovanni Iacca. Evolutionary learning of interpretable decision trees. *arXiv preprint arXiv:2012.07723*, 2020.

[3] Jesse Farebrother, Marlos C Machado, and Michael Bowling. Generalization and regularization in dqn. *arXiv preprint arXiv:1810.00123*, 2018.

[4] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning. *arXiv preprint arXiv:1811.12560*, 2018.

[5] OpenAI Gym. Toolkit for developing and comparing reinforcement learning algorithms.

[6] Swagat Kumar. Balancing a cartpole system with reinforcement learning–a tutorial. *arXiv preprint arXiv:2006.04938*, 2020.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[8] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped dqn. *Advances in neural information processing systems*, 29:4026–4034, 2016.

[9] Ziyu Wang, Nando Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. 11 2015.

[10] Marco A Wiering and Martijn Van Otterlo. Reinforcement learning. *Adaptation, learning, and optimization*, 12(3), 2012.