Solving “Frozen Lake” and “Lunar Lander” with Q-Learning and Deep Q-Learning

Alessandro Alviani
alessandro.alviani@city.ac.uk

Dimitrios Megkos
dimitrios.megkos@city.ac.uk

School of Mathematics, Computer Science and Engineering
City, University of London

Individual work:

Solving Atari Freeway with a Double DQN (RLlib)

Deep Reinforcement Learning
(INM707)

Table of Contents

** Number of page in the PDF reader.*

1.Basic

1.1 Define an environment and a problem to solve

1.1.1. Environment

The first environment presented is a 8 X 8 custom-built grid world inspired by *OpenAI Gym's FrozenLake* [1].
An agent is placed at one end of a frozen lake (*state 1),* and it is asked to find the best way to a *goal state* placed at the other end of the grid. The environment presents several holes that the agent must avoid.
There are 64 possible *states* that the agent can take, defining the *State matrix (S)*:

1. *Start* (*state 1*): this is the grid's first square (top-left). The agent starts each episode at this position, and it is reset here if it falls into a hole or reaches the goal.
2. *Goal* (*state 64*): last square (bottom-right). If the agent reaches this state, the game is solved, and the character is restarted to *state 1*.
3. *Holes* (17 in total): their positioning allows two ways to the goal: short and long. If the agent falls into a hole, it is reset to *state 1*.
4. *Frozen tiles* (47 in total): the agent steps onto them to explore the environment. These tiles lead to holes, dead ends, and the goal.

The *states* are shown in figures 1 and 2.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

S	F	F	F	H	F	F	F
F	H	F	F	F	H	F	F
F	F	H	F	F	F	H	F
F	F	F	H	H	F	H	F
F	F	H	H	F	F	F	H
F	F	F	H	F	H	F	H
F	F	H	F	F	H	H	H
F	H	F	F	F	F	F	G

Figure 1. State Matrix (numbered) and State Matrix (right) by feature: S = start, F = frozen, H = hole, G = goal

The agent can navigate this bidimensional environment in four directions: Up, Down, Left, Right, as long as it stays within the grid; actions that would lead outside are restricted.
Figure 2 presents the *Frozen Lake* environment as displayed by our custom-made GUI:

Figure 2. The *Frozen Lake,* custom-made GUI.

1.1.2 Problem to solve

The problem to solve can be formulated as follows: given the presented environment (*S), p*rovided the *reward matrix (R)*,* and chosen the Q-learning method to solve the problem (Bellman's equation), we want to train our *agent* to learn an optimal *Q-matrix***, that is the long-term return of an action *a*, taken in a state *s,* under a certain policy π.
In other words, the agent is asked to update the *Q-matrix* by mapping *state-action* pairs to *reward (action-value)* according to the following Q-learning algorithm:

*The operator assigns a *reward* to each *state-action (64 X 4)* and stores all the values in a *reward matrix (R).*
**The agent is provided with a *Q-matrix (Q)*, *R* shaped (64 X 4), initially empty (filled with zeros).

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

The *Q-matrix* represents the agent's knowledge of the environment, in the form of *value* $Q(s_t, a_t)$ of a given action a_t *from* a *certain state* s_t.
In our problem, an optimally defined *Q-matrix,* followed by the selection of the sequence of actions that maximizes the *reward*, identifies the shortest path from *start* to *goal*.

1.2. Define a state transition function and the reward function

1.2.1 State transition function

The *state transition function* in a *Markov Decision Process* (*MDP*) is defined as follows:

$$s_{t+1} = P(s_{t+1} | \, s_t, a_t)$$

where the state s_t transitions to the state s_{t+1} with probability given by taking an action a_t from state s_t.
In our problem, the *state transition function* is implemented according to the following logic:

1. Identify available actions $a_{1t} \ldots a_{nt}$ from state s_t
2. Check the reward $ra_{1t} \ldots ra_{nt}$ associated with each action (in the *Q-matrix*).
3. Select the action with the highest *action-value,* $\max(ra_t)$.
4. Apply a ε greedy policy (handles the exploration - exploitation ratio).
5. Return the *action*.

Finally the returned *action* is used as the input of the *select_step* function that controls the *action-state* of the character in the 8X8 grid world, as illustrated below:

- Right: + 1 (moves to the next column).
- Left: - 1 (moves to the previous column).
- Up: - 8 (moves to the row above).
- Down: + 8 (moves to the row below).

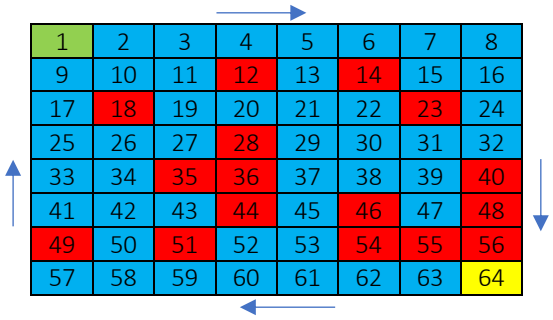


Figure 3 Transition logic

As mentioned in section 1.1.1, every *state* has its own possible actions, so that the character always stays within the grid.
In an initial implementation of this environment the icy ground was defined to have some risk, and the character stepping onto a frozen tile had a 10% probability to slip in any allowed direction. The idea was interesting, but the addition of randomness made the model(s) harder to test and benchmark, so this feature was eventually left out.

1.2.2 Reward function

In a MDP the *reward function* defines the immediate reward r_{t+1} that the agent is given for taking an action a_t from a state s_t, and it is defined by:

$$r_{t+1} = r(s_t, a_t)$$

In the presented implementation this function is represented by a 64 X 4 *reward matrix (R).* This can be thought as an expansion of the *state matrix (S)*, where each *state* is split in its 4 actions, and each *state-action* is linked to a reward. *State-actions* are rewarded as per below:

- Leading into the *hole*: – 10.
- Leading to the *final state* (*goal*): + 10.
- Leading to *frozen tiles*: –0.5.
- Leading outside the grid: *np.nan*, in that unavailable actions.

The agent is rewarded 10 points to reach the goal, and penalised -10 to fall into a hole. At each steps it occurs a penalty of -0.5, so that moving is expensive, and to maximise the *reward* it has to reach the goal in as little steps as possible. The reward for restricted actions, that are movements that would take the character outside the grid, is *undefined*.

1.3 Set up the Q-learning parameters (gamma, alpha) and policy

1.3.1 Q-learning parameters (alpha, gamma)

In the context of Q-learning, alpha (α) is known as the *learning rate*. This hyperparameter controls how fast the model learns about the environment, or in other terms, how fast the new learning changes the past one. This parameter can take values between 0 and 1. A value of α equal to zero means that the model assigns no weight to the new learning, that is it stops learning and any new *action-value* is set equal to the old one. Oppositely, a value of α equal to 1 will put all the weight on the new information, with the result of new learning overriding the previous one.
In the same context, gamma (γ) is known as the *discount rate.* This hyperparameter controls the worth of future rewards, and it takes values between 0 and 1. Value of γ close to zero means that the model assigns very little weight to future rewards, and it is therefore applying a policy of "instant gratification". On the contrary values of γ close to 1 indicates that the model will strive for long term-high reward. The presented equation clarifies the role of α and γ in the Q-learning updates:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \, max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

In our implementation we tested the model on values in the range 0.9 - 0.1 for α, and in the range 0.9 – 0.1 for γ.
α and γ were analysed in conjunction with different policies (Figure 4), and were ultimately set to: $\alpha = 0.5$ and $\gamma = 0.9$.

1.3.2 Policy

A *policy* (π) tells the *agent* what action a_t to take at any given state s_t. Goal of deep reinforcement learning is to find an *optimal policy* which maximises the *reward* in the long run. In this context, one of the challenges we face is the "*exploration-exploitation dilemma*": the agent needs to find the best compromise between exploring the environment and sticking to the best routes already learned. This behaviour can be controlled with an epsilon-greedy policy (ε), where $\varepsilon = 1$ defines an agent that always chooses at random, and $\varepsilon = 0$ an agent that always chooses greedily.
Our implementation adopts a ε-greedy policy, where ε is set to 0.9, and it is combined to a *decay rate* = 0.00001. This means that at the beginning of its training the agent mostly explores the environment, but as the number of *epochs* grows, and the agent learns, its policy shifts more and more towards a greedy one, where the agent reduce its exploring and rather follows the best paths already discovered.
The ε decay rate should be inversely proportional to the number of epochs, although not necessarily linearly. The relation between ε and number of episodes in our model is showed below:

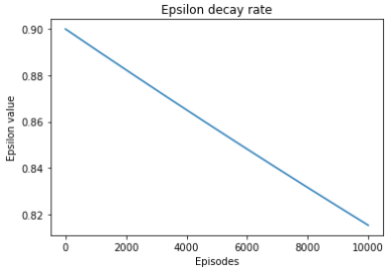


Fig. 4 Epsilon decay rate

Greedy policy and decay rate values were chosen in conjunction with α and γ, to maximise the number of times the character would reach the *goal* and the cumulative reward collected. The choice of this hyperparameters is further explained section 1.5.

1.4. Run the Q-learning algorithm and represent its performance

The performance of the Q-learning algorithm, ran on the model after hyperparameters tuning, is presented below:

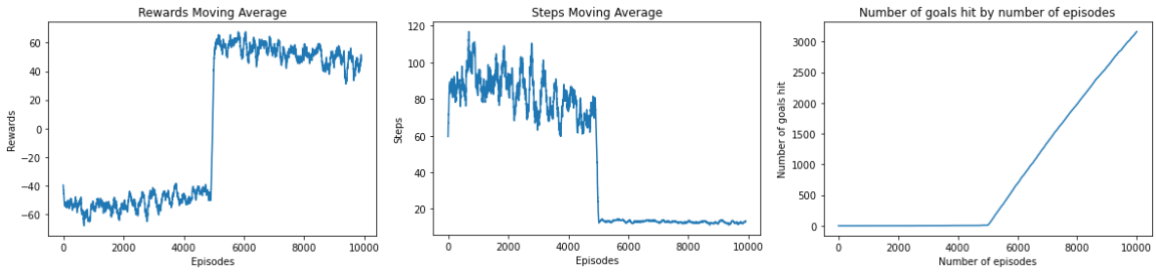


Figure. 5 Q-learning performance metrics (moving averages are on 100 episodes)

The above plots show a first phase, approximately between 0 and 5,000 epochs, where the agent explores the grid and tries to learn about the environment. In this *exploration* phase the agent takes many steps, and consequently collects high negative rewards. This pattern repeats for about 5,000 *epochs* due to the complexity of the environment, the agent's policy (high ε) and a very small ε decay rate. At around 5,000 episodes the agent starts hitting the *final state* and, given the good knowledge already accumulated and its policy getting greedier (due to the decaying ε), it starts exploiting the path to the *goal*. In the above plots this transition between exploration and exploitation is very clear. Even after having found the best path, the agent still tries to explore a little, as shown in the small changes in rewards and steps. However, the agent in this second phase, between 5,000 and 10,000 epochs, mostly sticks to the optimal actions, which we believe to be the desired behaviour.
The training was carried out over 10,000 episodes, but by looking at the plots we can conclude that this could have been early-stopped just after 5,000 epochs.

1.5. Repeat the experiment with different parameter values, and policies

The hyperparameter tuning was carried out through a custom-made grid search for α, γ, ε, and ε decay rate. This search takes a long time to complete, but we believed it could yield more robust results than testing singular hyperparameters at the time. The number of hyperparameters tested was limited by the resources available, with searching time in a grid-search growing with time complexity of $O(n^p)$, where *p* is the number of parameters tested.
The models at different parameters were tested on the number of *goals hit*, that is how many times the character reached the *final state*, the *cumulative reward* collected, and the average *number of steps* per episode.
We considered *best* the model that hit the most number of *goals,* together with the highest value of *total rewards cumulated*. An average number of steps (sitting between min and max) was considered best too, as explained by a large number of steps during *exploration* followed by a small number of steps during *exploitation*.
The grid results, and the best hyperparameters combination (highlighted) are presented in figure 6.

```
Goals hit: 2173 | Cumulative reward: -133502.0 | AVG steps: 55 | Alpha: 0.9 | Gamma: 0.9 | Epsilon_s: 0.9 | Epsilon_f: 0.82 | Decay Rate: 1e-05
Goals hit: 1081 | Cumulative reward:  -84400.5 | AVG steps: 21 | Alpha: 0.9 | Gamma: 0.9 | Epsilon_s: 0.9 | Epsilon_f: 0.34 | Decay Rate: 0.0001
Goals hit:    0 | Cumulative reward: -155280.0 | AVG steps: 11 | Alpha: 0.9 | Gamma: 0.9 | Epsilon_s: 0.9 | Epsilon_f: 0.01 | Decay Rate: 0.001
Goals hit:  329 | Cumulative reward: -414007.5 | AVG steps: 70 | Alpha: 0.9 | Gamma: 0.5 | Epsilon_s: 0.9 | Epsilon_f: 0.82 | Decay Rate: 1e-05
Goals hit:   79 | Cumulative reward: -233903.5 | AVG steps: 29 | Alpha: 0.9 | Gamma: 0.5 | Epsilon_s: 0.9 | Epsilon_f: 0.34 | Decay Rate: 0.0001
Goals hit:    0 | Cumulative reward: -154823.5 | AVG steps: 11 | Alpha: 0.9 | Gamma: 0.5 | Epsilon_s: 0.9 | Epsilon_f: 0.01 | Decay Rate: 0.001
Goals hit:   25 | Cumulative reward: -450535.0 | AVG steps: 71 | Alpha: 0.9 | Gamma: 0.1 | Epsilon_s: 0.9 | Epsilon_f: 0.82 | Decay Rate: 1e-05
Goals hit:   15 | Cumulative reward: -240911.5 | AVG steps: 29 | Alpha: 0.9 | Gamma: 0.1 | Epsilon_s: 0.9 | Epsilon_f: 0.34 | Decay Rate: 0.0001
Goals hit:    2 | Cumulative reward: -154146.5 | AVG steps: 11 | Alpha: 0.9 | Gamma: 0.1 | Epsilon_s: 0.9 | Epsilon_f: 0.01 | Decay Rate: 0.001
Goals hit: 3874 | Cumulative reward:  117198.5 | AVG steps: 42 | Alpha: 0.5 | Gamma: 0.9 | Epsilon_s: 0.9 | Epsilon_f: 0.82 | Decay Rate: 1e-05
Goals hit:  589 | Cumulative reward: -154122.0 | AVG steps: 24 | Alpha: 0.5 | Gamma: 0.9 | Epsilon_s: 0.9 | Epsilon_f: 0.34 | Decay Rate: 0.0001
Goals hit:    0 | Cumulative reward: -155141.0 | AVG steps: 11 | Alpha: 0.5 | Gamma: 0.9 | Epsilon_s: 0.9 | Epsilon_f: 0.01 | Decay Rate: 0.001
Goals hit:  155 | Cumulative reward: -441348.5 | AVG steps: 72 | Alpha: 0.5 | Gamma: 0.5 | Epsilon_s: 0.9 | Epsilon_f: 0.82 | Decay Rate: 1e-05
Goals hit:   65 | Cumulative reward: -237684.0 | AVG steps: 29 | Alpha: 0.5 | Gamma: 0.5 | Epsilon_s: 0.9 | Epsilon_f: 0.34 | Decay Rate: 0.0001
Goals hit:    1 | Cumulative reward: -154702.0 | AVG steps: 11 | Alpha: 0.5 | Gamma: 0.5 | Epsilon_s: 0.9 | Epsilon_f: 0.01 | Decay Rate: 0.001
Goals hit:   20 | Cumulative reward: -457629.0 | AVG steps: 72 | Alpha: 0.5 | Gamma: 0.1 | Epsilon_s: 0.9 | Epsilon_f: 0.82 | Decay Rate: 1e-05
Goals hit:    8 | Cumulative reward: -240334.0 | AVG steps: 28 | Alpha: 0.5 | Gamma: 0.1 | Epsilon_s: 0.9 | Epsilon_f: 0.34 | Decay Rate: 0.0001
Goals hit:    0 | Cumulative reward: -154603.5 | AVG steps: 11 | Alpha: 0.5 | Gamma: 0.1 | Epsilon_s: 0.9 | Epsilon_f: 0.01 | Decay Rate: 0.001
Goals hit:   21 | Cumulative reward: -459257.5 | AVG steps: 72 | Alpha: 0.1 | Gamma: 0.9 | Epsilon_s: 0.9 | Epsilon_f: 0.82 | Decay Rate: 1e-05
Goals hit:  103 | Cumulative reward: -225008.0 | AVG steps: 27 | Alpha: 0.1 | Gamma: 0.9 | Epsilon_s: 0.9 | Epsilon_f: 0.34 | Decay Rate: 0.0001
Goals hit:    1 | Cumulative reward: -154553.5 | AVG steps: 11 | Alpha: 0.1 | Gamma: 0.9 | Epsilon_s: 0.9 | Epsilon_f: 0.01 | Decay Rate: 0.001
Goals hit:   39 | Cumulative reward: -461010.0 | AVG steps: 73 | Alpha: 0.1 | Gamma: 0.5 | Epsilon_s: 0.9 | Epsilon_f: 0.82 | Decay Rate: 1e-05
Goals hit:   14 | Cumulative reward: -243065.5 | AVG steps: 29 | Alpha: 0.1 | Gamma: 0.5 | Epsilon_s: 0.9 | Epsilon_f: 0.34 | Decay Rate: 0.0001
Goals hit:    0 | Cumulative reward: -154604.0 | AVG steps: 11 | Alpha: 0.1 | Gamma: 0.5 | Epsilon_s: 0.9 | Epsilon_f: 0.01 | Decay Rate: 0.001
Goals hit:   23 | Cumulative reward: -448997.0 | AVG steps: 70 | Alpha: 0.1 | Gamma: 0.1 | Epsilon_s: 0.9 | Epsilon_f: 0.82 | Decay Rate: 1e-05
Goals hit:   15 | Cumulative reward: -241162.0 | AVG steps: 29 | Alpha: 0.1 | Gamma: 0.1 | Epsilon_s: 0.9 | Epsilon_f: 0.34 | Decay Rate: 0.0001
Goals hit:    1 | Cumulative reward: -154799.5 | AVG steps: 11 | Alpha: 0.1 | Gamma: 0.1 | Epsilon_s: 0.9 | Epsilon_f: 0.01 | Decay Rate: 0.001
```

Fig. 6 Hyperparameters Grid-Search

1.5.1 Parameter values

The model was tested at different values of *learning rate (*α), *discount rate (*γ), *epsilon decay rate*, and *number of timesteps.* The tests on α, γ and number of allowed timesteps are presented below (Figure 7):

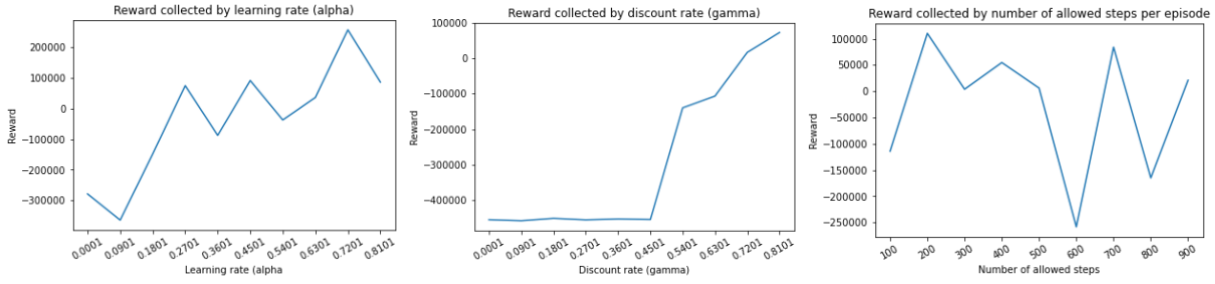


Fig. 7 Model's performance for different values of α (7a), γ (7b) and number of allowed steps (7c).

Higher values of γ yielded better results; similarly, larger α values were positively correlated with the number of goals hit and rewards collected. Values of α and γ set at the top of their range indicate a model that needed a strong tuning towards exploring, likely due to the grid size. On the contrary, increasing the number of *timesteps* above a certain threshold (200) did not bring any additional benefit but the downside of increasing the model's training times.

1.5.2 Policies

With regards to the ε-greedy policy, we experimented a range of values for ε between 0.0001 (extremely greedy) and 0.9 (markedly random). At the same time, we tried different decay rates. We had mixed results in this area, with ε values equal to

0.55, and in the range 0.8 - 0.9, yielding very similar results. However, larger decay rates, therefore smaller ε values over time, consistently generated worse model performance, and very often led to a model that could not learn at all.
In the end we chose an ε value equal to 0.9, and a decay rate equal to 0.00001. Over 10,000 episodes this resulted in ε values between 0.9 (start) and 0.82 (final), with an agent (once again) pushed to explore.

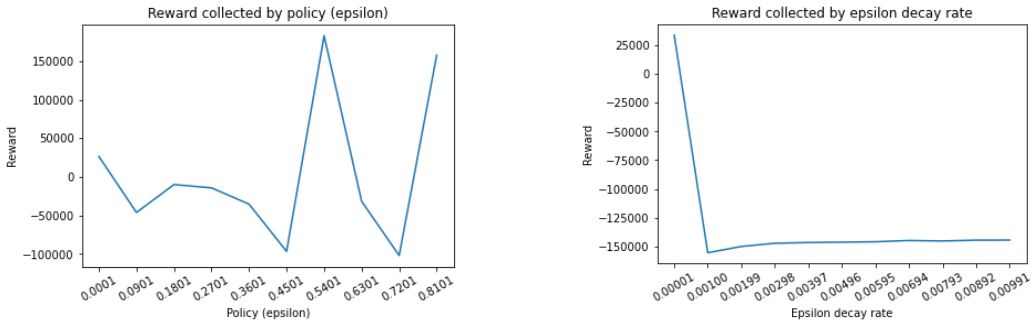


Fig. 8 Model's performance for different values of ε-greedy.

1.6 Analyze the results quantitatively and qualitatively

Our model required extensive hyperparameter tuning to generate satisfying results. Low values of α, too large decay rates, and low values of γ resulted in a model incapable of learning at all, with no *final states* reached regardless of the number of training episodes. The final hyperparameters selected: α = 0.5, γ = 0.9, ε = 0.9 and decay rate = 0.00001 explain a model that requires a high incentive to explore and benefits from big weights to future rewards. The agent appeared to be particularly challenged by the initial discovery of the *final state*, which took about 5,000 epochs at the best settings, and never occurred at sub-optimal hyperparameters' values. After reaching the *goal*, the *agent* hit it repeatedly, and its exploration cut down, although still present. This behaviour was considered ideal as the agent only reached the *final state* after extensive learning, therefore with no particular incentive to carry on exploring.
The plots in figure 5 show the moment the agent finds the *goal* and its transition from *exploration* to *exploitation*. The same plots illustrate the *reward scheme,* that negatively correlated the reward to the number of steps. Finally, they reveal that the agent had learned the *optimal policy* just after 5,000 epochs, hence its training could have been early-stopped.
As shown in figure 7a and 7b, high values of α and γ were critical for the agent to reach the *goal*, promoting a strong exploratory strategy motivated by a *final state* far from its initial position.
Similar reasoning is applied to the agent's policy, tuned to generate a strong attitude towards exploration, so to increase the likelihood of reaching the *final state*.
We originally implemented a 4 X 4 grid-world and noticed a significant difference in optimal hyperparameters: as the environment size grew the agent required more incentive to explore, therefore larger values of γ and ε.
The larger grid also proved to be much less forgiving in terms of model optimization, hence encouraged deeper insights and ultimately a better understanding of the problem.

2. Advanced

2.1 Implement DQN with two improvements. Motivate your choice and your expectations for choosing a particular improvement. Apply it in an environment that justifies the use of Deep Reinforcement Learning

2.1.1 Environment

The environment chosen to develop our DQN models was Lunar Lander Version 2 [2], an already built environment provided by Gym.OpenAi. A *lander* is required to land on a landing pad positioned at coordinates (0,0) placed between two yellow flags. The lander can take four discrete actions: fire left orientation engine, fire main engine, fire right orientation engine, or use no engine. The reward assigned to a landing depends on speed and place of landing. Each leg ground contact counts +10 points. A reward of 100 points is given to any landing; a perfectly cantered landing, at speed 0, results in a reward of 200 points. Firing the main engine costs -0.3 points per frame, resulting in a loss of points anytime the lander moves away from the landing pad or spends additional time in the air. If the lander crashes it loses 100 points. The fuel is infinite.
DQNs have been successfully used to solve the Atari Lunar Lander problem [3][4] and we believe this environment will offer a suitable level of complexity to train and test our models.

2.1.2. Implementation of the DQN

Our initial implementation of a basic Deep Q-Network (DQN) model was a *"Vanilla Deep Q-Network"*. The key-idea behind DQNs is to leverage on the artificial neural networks' (ANN) ability to be universal function approximators to resolve the Q-learning problem. This approach brings significant benefits over the standard Q-learning algorithm, and it is especially advantageous on large and continuous states that would make the iterative process of updating the Q-matrix computationally inefficient or unfeasible. A DQN takes as *state-inputs* a number of gameplay's last frames (i.e., images, usually 4), passes them through its convolutional layers, and ultimately outputs an estimated reward for each *state-action* (*Q-value).* The network learns by comparing the *outputted (policy) Q-*

values to the *target Q-values* and minimizing the resulting *loss*. With *target Q-values* we refer to values previously seen by the model (*experience)* and stored in a arbitrarily sized *replay-memory* in the form of tuples ($state_t$, $action_t$, $reward_{t+1}$, $new\ state_{t+1}$). These targets values serve as a ground truth to train the model, as explained.
Our base DQN model, used as foundation for the two improvements, is a *Vanilla DQN,* so implemented:

- Initialize *replay memory*.
- Initialize the *policy network*.
- Initialize the *target network*.
- ***For*** *each episode:*
    - Initialize the starting state.
        - ***For*** *each time step:*
            - Select an action.
            - Execute the action in the emulator and save *experience* in the *replay memory.*
            - Sample random *batch* from *replay memory* and pass it to the *policy network.*
            - Pass S_{t+1} to calculate the $arg max Q_a (S_{t+1}, a)$ as per Bellman equation.
            - Compute loss between *policy Q-values* and *target Q-values*.
            - Gradient descent to update weights in the *policy network*, to minimize the *loss*.

As previously mentioned ANNs are at the core of DQNs, and their architecture require particular attention (e.g., number of layers, number of hidden neurons, number of inputs, batch size, activation functions). In this work the founding ANN was partially built through model testing, partially by reading the relevant literature and replicating the recommended structures.
Once the model architecture was established, the original *Vanilla DQN* underwent an extensive tuning to find the optimal hyperparameters. The tests on learning rate (α), epsilon decay rate and discount rate (γ) are presented below:

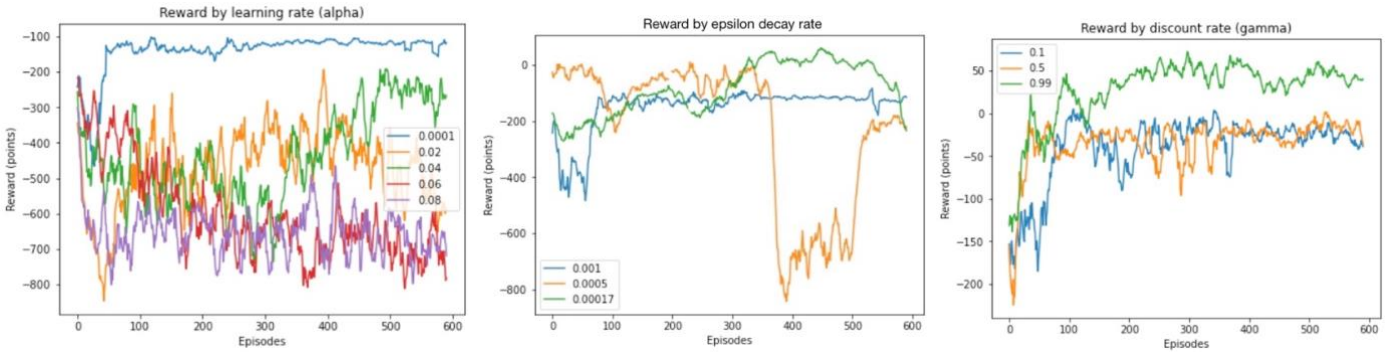


Fig. 9 Hyperparameters tuning (α, decay rate, γ) in the *Vanilla DQN*.

Table 1 presents the parameters values and architecture of the final *Vanilla DQN*:

Main Parameters - Vanilla DQN	
Number of inputs	4
Number of hidden layers	3
Number of hidden neurons	32
Batch size	256
Replay Memory size	100,000
Gamma (γ)	0.99
Alpha (α)	0.0001
Policy	ε-greedy
ε-range (min – max)	1 – 0.01
ε decay rate	0.00017
Number of episodes in tuning	600
Number of episodes in training	1000

Table 1. Models' parameters and architecture.

2.1.3 Two improvements, choice and expectations

The two chosen improvements on the *Vanilla DQN* were:

1. *Double DQN*
2. *Dueling Double DQN*

1. Double DQN has been successfully implemented on Atari games in the past [5] and solves the overestimation problem that affects *standard DQNs.* Namely, *t*he main flaw of *Vanilla DQNs* is represented by the use of the same network to estimate the *target Q-values* and the *policy Q-values*: as the net's weights are updated to minimise the loss, so are the *target Q-values* generated on arg*maxQ*$_a$ *(S*$_{t+1}$*, a),* resulting in an optimization that at each iteration "*chases its own tail"* and results in overoptimistic estimates, hance poorer policies. *Double DQN* resolves this problem by introducing a second network (*Target network*) dedicated to the estimation of the *target Q*-values, effectively decoupling the selection and evaluation actions. These decoupling is mathematically expressed by θ_t (network's weights for selection) and θ'_t (network's weights for evaluation) and presented below to define the target variables in *Double* and *Standard DQNs*:

$$Y_t^{DoubleQ} \equiv R_{t+1} + \gamma Q(S_{t+1}, arg\ max_a Q(S_{t+1}, a;\ \theta_t); \theta'_t)$$

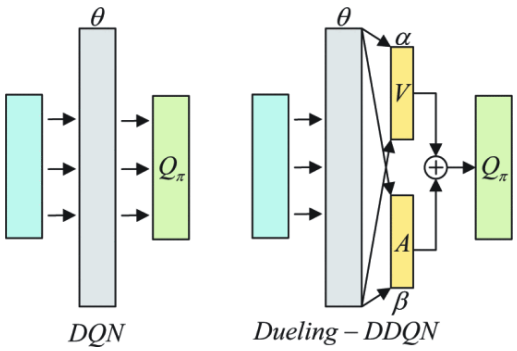
$$Y_t^{DQN} \equiv R_{t+1} + \gamma Q(S_{t+1}, argmax_a Q(S_{t+1}, a;\ \theta_t\ ); \theta_t)$$

This improvement has shown to yield significant improvement in model's performance on a range of Atari games [5] and we expect similar results in our implementation.

2. Dueling Double DQN is built on top of the *Double DQN* model and adds a *Dueling layer* between its last hidden layer and the output layer. The d*ueling* layer separates the *value function* $Q_\pi(s,a)$ into a *state value function* $V_\pi(s)$, which defines the value of being in state *s,* and an *advantage function* $A_\pi(s,a)$, which defines the advantage of taking the action *a* from state *s*.
The two functions are aggerated as follow to return $Q(s_t, a_t)$:

$$Q(s_t, a_t) = \ V(s_t) + (A(s_t, a_t) - \frac{1}{|A|}\sum A\,(s_t, a_t))$$

This results in a *dueling architecture* that differs from the *standard DQN's,* as per figure below:



Fugure 10. Structure of NN in DQN and Dueling-DDQN. The V_π (*s*) and A_π (*s, a*) are between the output layer and the last hidden layer, and the dimension of V_π (*s*) and A_π (*s, a*) are the same as the output layer. Taken from Yefeng Yang et al.[6]

*Dueling DQN*s have demonstrated to work effectively when coupled with *Double DQNs [7],* increasing model performance in a range of domains [8] and setting the new state-of-the-art on the Atari 2600 domain (2015) [9]. In line with the literature, we expect the *Dueling Double DQN* to perform better than the *Double DQN,* and dramatically better than the *Vanilla DQN.*

2.1.4 Analyse the results quantitatively and qualitatively

As expected, the *Vanilla DQN,* our baseline model, performed worse than its two improvements. This model, on 1,000 training episodes, struggled to find a satisfying policy and presented an overall unstable behaviour. This in line with Hado Van Hassel et al. [5] and their recommendation of an added layer (*Target network)* to effectively approximate the *Q-values*, as implemented in the *Double DQN* models. The *Vanilla DQN's* optimization problem was especially evident in training episodes resulting in thousands of timesteps, where the lander would fly for minutes in a seemingly confused way. In those episodes the agent collected extremely high negative results, presented in figure 11a.
When a *dueling* layer was added to the base model (forming a *Dueling DQN*)* (figure 11b), this yielded some stability improvement, however the overall performance stayed poor, and overall the agent did not seem to be able to solve the presented problem.
Double DQN (11c) was the improvement that yielded enormous improvements over the *Vanilla,* and which offered the right foundation to successfully implement the *Dueling.* This is due to resolving that main optimization flow inherent to *Vanilla DQN*s, therefore to an improvement that we may define essential to that algorithm.

*the *dueling layer* is not meant to be implemented on a *Vanilla DQN*, but rather on a *Double DNQ*. This experiment was only reported as part of an ablation study to better explain the effect of the different improvements.

Figure 11c shows the dramatic change from the previous two models in the 10 episode-rolling average reward collected, with the agent drawing an almost-ideal learning pattern. In just over 600 episodes the lander started lading successfully at most attempts, and kept increasing its rewards by using better trajectories and smoother landings. In numbers, over the entire 1,000 training episodes, this translated to a 10 episode-rolling average reward of 2 against *Vanilla*'s -525.
Adding the *dueling layer* to the *Double DQN* (figure 11d and Table 1) yielded further improvement, as anticipated: the 10 episode-rolling average reward went up from 2 to 23; the 10 episode-rolling maximum reward from 89 to 101; the 10 episode-rolling minimum reward from -375 to -299. This was a substantial leap forward and more complex environments, including more *state-actions* and the addition of enemies/obstacles, could benefit even more greatly from this architecture [9].

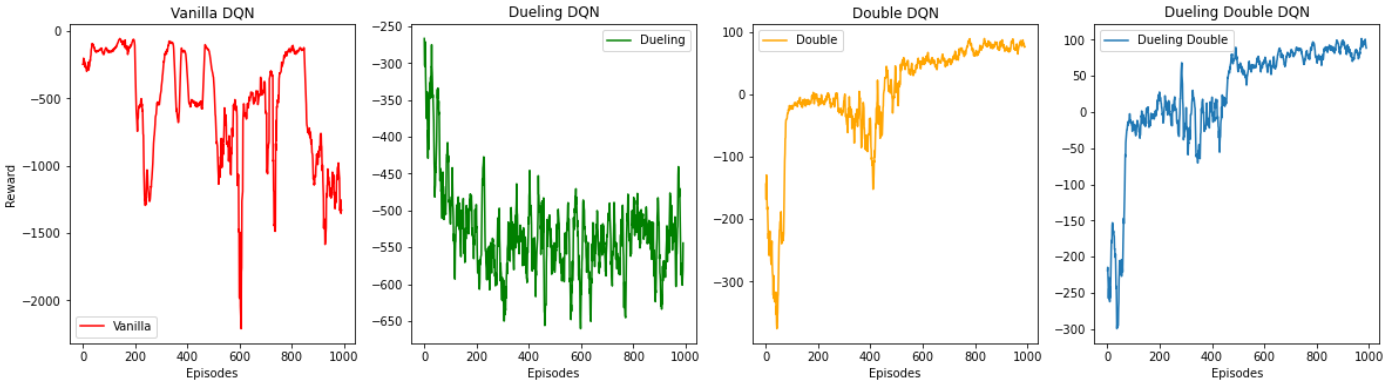
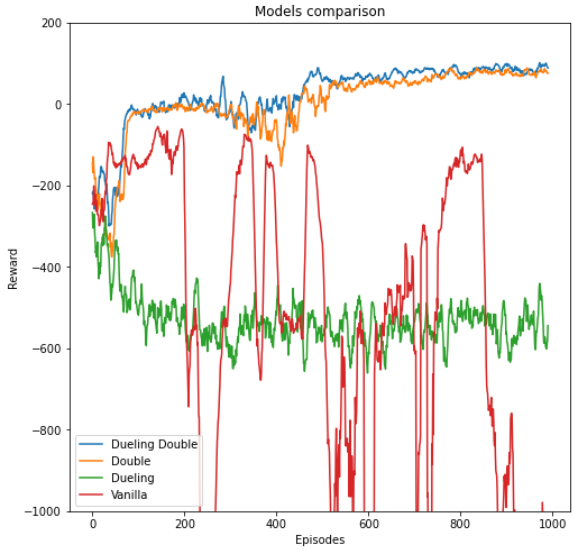


Fig. 11 DQN Models' performance. Vanilla (11a), Dueling (11b), Double (11c), Double Dueling (11d)

Figure 12 gathers the above subplots in one single figure, to better compare them:



Fugure 12. Models comparison. (The Y axis is limited to -1000 preserve a representative scale for the other models).

Table 1 presents rewards (R) and flying times (f.t.) 10 episode-rolling averages for each model:

Model	Mean R.	Max R.	Min R.	Mean f.t.	Max f.t.	Min f.t
Dueling Double	23	101	-299	466	500	85
Double	2	89	-375	458	500	100
Dueling	-531	-266	-660	67	100	57
Vanilla	-525	-55	-2,208	254	500	66

Table 2. Models' metrics comparison

Our results, in line with the literature, suggested the application of a *double* architecture to DQNs aiming to solve Atari games and similar environments. This architecture was much more capable of learning this type of problem and generating effective policies. Adding a *dueling* layer to the *double* model further improved model performance in Atari Lunar Lander V2.
The *Double Dueling DQN* resulted in the best model overall, collecting the highest reward and presenting the best stability.
To reduce training times, in our tests the number of training episodes was limited to 1,000 and the number of possible timesteps to 500. This latter parameter especially avoided spikes in training times, restricting the Lunar Lander from flying minutes per single training episode. More extensive training, and in particular a higher number of epochs, would likely improve the final models.

The models were built to run on a GPU but, when tested on Google Colab's *Tesla T4*, they trained slower than on a CPU. This could be due to the relatively small size of our models, which coupled with the overhead of invoking GPU kernels, resulted in a negative balance for the GPU over the CPU.

2.2 Apply the RL algorithm of your choice (from rllib) to one of the Atari Learning Environment. Briefly present the algorithm and justify your choice.

For this exercise, the Double DQN reinforcement learning algorithm was chosen, and was applied to the Atari Learning Environment Freeway, using Ray RLlib. Freeway is a classic atari game, where one player (the agent) controls a chicken which can be made to run across a ten-lane highway filled with traffic, to get to the other side. Every time the chicken successfully crosses the road, a point (1) is awarded to the agent. If the chicken gets hit by a car, it gets pushed back to the bottom of the screen. The point of the game is to collect as many points as possible. This is an easy game overall, and the original hypothesis was a DQN with only one improvement, will be able to learn successfully how to play it. The *Double DQN* algorithm has been thoroughly explained in section 2.1.3 and its comparison with Vanilla DQN and DQN with improvements was shown in section 2.1.4.

2.3 Analyse the results quantitatively and qualitatively

Two different RLlib configurations were tested initially, each with different hyperparameters. The first one was a Vanilla DQN, the second was a Double DQN. Epsilon greedy is used to handle the agent's exploration-exploitation rate. The decision to use Double DQN as the main algorithm of this exercise was based on the outcome of this test. The performance of each algorithm is shown in figure 13 below:

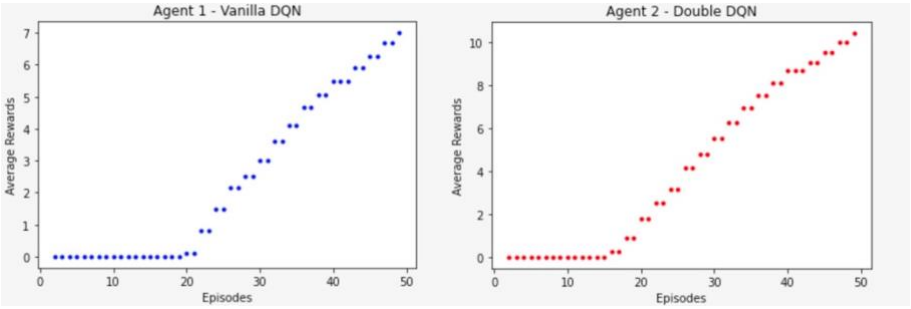


Figure 13. Agent 1 (Vanilla DQN) vs Agent 2 (Double DQN) performance

During the first trials, a network architecture of one hidden layer with 512 hidden neurons was used for both Vanilla DQN and Double DQN. Vanilla DQN did not manage to receive any rewards after 50 iterations, achieving a mean reward of zero during evaluation. Double DQN managed to receive 1.20 average rewards after 50 iterations, achieving a mean reward of 11 during evaluation. Changing the architecture of the networks to two hidden layers with 256 hidden neurons each greatly improved the performance of both Vanilla DQN and Double DQN. Having two hidden layers, the agents were able to perform more complex calculations enabling them to learn faster and better. Vanilla DQN started earning rewards after 19 iterations, reaching an average of 7 rewards after 50 iterations and achieving a mean reward of 21 during evaluation. Double DQN started earning rewards after 15 iterations, reaching an average of 10.40 rewards after 50 iterations and achieving a mean reward of 22 during evaluation. Although both agents achieved similar evaluation scores, Double DQN was the winner, not only because it started to learn faster, but also because it managed to achieve higher score after 50 training iterations. This made sense considering a Double DQN is using a second network to calculate the target values.

The Ray Tune library was used for further experimentation and fine tuning Double DQN's hyperparameters, utilizing a grid search. The parameters that seemed to affect learning the most were the learning rate, the number of hidden layers, the number of hidden neurons and the type of activation function, therefore the grid search focused on those parameters. The tuning ran until the agents reached a mean episode reward of 12 and Tune returned the best hyperparameter combination that required the least total timesteps to reach the goal.

The results of the grid search are shown on figure 14 below:

Trial name	status	loc	lr	model/fcnet_activation	model/fcnet_hiddens	iter	total time (s)	ts	reward	episode_reward_max	episode_reward_min	episode_len_mean
DQNTrainer_Freeway-ramDeterministic-v4_09513_00000	TERMINATED	127.0.0.1:69587	0.001	relu	[256, 256]	46	317.347	46000	12.2727	24	0	2048
DQNTrainer_Freeway-ramDeterministic-v4_09513_00001	TERMINATED	127.0.0.1:69589	0.0001	relu	[256, 256]	60	401.483	60000	12.069	21	0	2048
DQNTrainer_Freeway-ramDeterministic-v4_09513_00002	TERMINATED	127.0.0.1:69588	0.001	linear	[256, 256]	54	363.448	54000	12.2308	24	0	2048
DQNTrainer_Freeway-ramDeterministic-v4_09513_00003	TERMINATED	127.0.0.1:69584	0.0001	linear	[256, 256]	119	649.775	119000	12.0345	21	0	2048
DQNTrainer_Freeway-ramDeterministic-v4_09513_00004	TERMINATED	127.0.0.1:69590	0.001	relu	[256, 256, 256]	56	404.791	56000	12.2963	23	0	2048
DQNTrainer_Freeway-ramDeterministic-v4_09513_00005	TERMINATED	127.0.0.1:69583	0.0001	relu	[256, 256, 256]	101	613.861	101000	12.0612	23	0	2048
DQNTrainer_Freeway-ramDeterministic-v4_09513_00006	TERMINATED	127.0.0.1:69585	0.001	linear	[256, 256, 256]	82	525.417	82000	12.15	23	0	2048
DQNTrainer_Freeway-ramDeterministic-v4_09513_00007	TERMINATED	127.0.0.1:69586	0.0001	linear	[256, 256, 256]	113	655.175	113000	12.0182	20	0	2048

Figure 14. Grid Search results

According to the grid search, the best hyper parameter combination was learning rate 0.001, two hidden layers with 256 hidden neurons and ReLU activation function. All eight configurations reached the target. Adding more hidden layers did not benefit the network, it only added more time to learn since there were more computations happening. The reason behind this was that the problem was not complex enough, meaning it did not require three hidden layers. The configurations with smaller learning rate also required more iterations since the agent was learning slower.

Using the best configuration selected by the grid search, a training of 200 iterations was performed. The results are shown in figure 15 below:

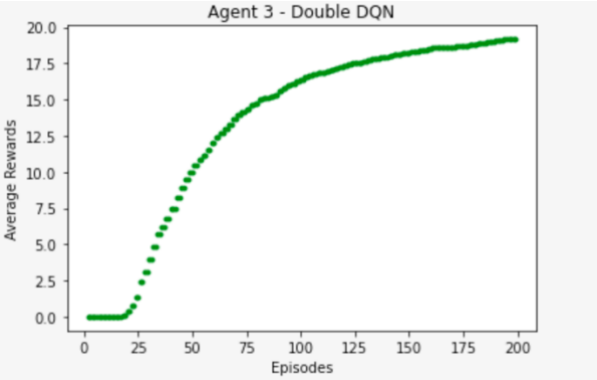


Figure 15. Double DQN results with best configuration, after 200 training iterations

Agent 3 with optimal configuration reached an average of 20 rewards after 200 iterations and from that point it started reaching plateau. During evaluation it was able to achieve an average of 24.2 rewards, after playing 5 episodes, which is impressive considering back in the day ActiVision used to send a cloth "Save The Chicken Foundation" patch to every player that managed to score 20 or more points on either Road 3 or Road 7 and sent in a photograph of their television screen. [10]

References

[1] https://gym.openai.com/envs/FrozenLake-v0/
[2] https://gym.openai.com/envs/LunarLander-v2/
[3] S. Gadgil, Y. Xin and C. Xu, "Solving The Lunar Lander Problem under Uncertainty using Reinforcement Learning," *2020 SoutheastCon*, 2020, pp. 1-8, doi: 10.1109/SoutheastCon44009.2020.9368267.
[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller, "Playing Atari with Deep Reinforcement Learning", https://doi.org/10.48550/arXiv.1312.5602
[5] Hado Van Hassel, Arthur Guez, David Silver, "Deep Reinforcement Learning With Double Q-Learning", arXiv:1509.06461v1 [cs.LG] 22 Sep 2015
[6] Y. Yang, X. Ban, X. Huang and C. Shan, "A Dueling-Double-Deep Q-Network Controller for Magnetic Levitation Ball System," *2020 39th Chinese Control Conference (CCC)*, 2020, pp. 1885-1890, doi: 10.23919/CCC50068.2020.9189157.
[7] Y. Huang, G. Wei and Y. Wang, "V-D D3QN: the Variant of Double Deep Q-Learning Network with Dueling Architecture," *2018 37th Chinese Control Conference (CCC)*, 2018, pp. 9130-9135, doi: 10.23919/ChiCC.2018.8483478.
[8] A. Iqbal, M. -L. Tham and Y. C. Chang, "Energy- and Spectral- Efficient Optimization in Cloud RAN based on Dueling Double Deep Q-Network," *2021 IEEE International Conference on Automatic Control & Intelligent Systems (I2CACIS)*, 2021, pp. 311-316, doi: 10.1109/I2CACIS52118.2021.9495912.
[9] Ziyu Wang, et al., "Dueling Network Architectures for Deep Reinforcement Learning", arXiv:1511.06581, 2015.
[10] Freeway (video game), Wikipedia https://en.wikipedia.org/wiki/Freeway_(video_game)

DRL_Part1

April 20, 2022

1 Deep Reinforcement Learning Coursework

1.0.1 Alessandro Alviani - alessandro.alviani@city.ac.uk

1.0.2 Dimitrios Megkos - dimitrios.megkos@city.ac.uk

1.0.3 Part 1: Q-Learning

Grid world: Navigating the Frozen Lake The environment is inspired by OpenAI Gym's FrozenLake. The agent is placed in a 8x8 grid world, beginning always from the same state, trying to reach the other side of the lake. The agent navigates through a frozen path, trying to avoid a number of holes the lake has. If the agent falls into a hole, the agent starts again from the initial state. Because the path is frozen, the agent will sometimes move to a different direction than the one that was picked. There are 64 available states and four available actions (Up:0, Down:1, Left:2, Right:3).

Import libraries and custom functions

[1]:
```
# Import numpy library
import numpy as np
# Import custom functions
from qlearning_functions import select_step, select_action, init_env,␣
 ↪update_qvalue, plot_cma, reward_ma, timestep_ma, display_episodes,␣
 ↪trainig_grid
import matplotlib.pyplot as plt
# Set random seed for reproducibility
np.random.seed(0)
```

Initialize the environment and parameters

[2]:
```
# Initialize our environment
S, A, R, Q, state, goal_state, hole_state = init_env()

# Initialize Q-Learning parameters
alpha = 0.5 # Learning Rate
gamma = 0.99 # Discount parameter
num_ep = 10000 # Number of episodes
num_timestep = 500 # Number of timesteps
```

```
# Exploration rate
epsilon = 0.9
max_epsilon = 0.9
min_epsilon = 0.01
epsilon_decay_rate = 0.00001
```

Initialize lists for evaluation metrics

[3]:
```
# Initialize lists for evaluation metrics
reward_ep_list = [] # List containing rewards
ts_ep_list = [] # List containing number or timesteps per episode
actions = [] # List containing the agent's actions
eps_decay = [] # List containing the updated epsilon per episode
goals = [] # List containing the number of goals hit by the agent
goal = 0
```

Train the agent

[4]:
```
#run num_episodes episodes
for episode in range(num_ep):

    #print("Starting state is '{}'".format(S[state]))

    # Initialize/Reset reward metric
    r_metric = 0

    goals.append(goal)

    for timestep in range(num_timestep):
        # Select action
        action = select_action(R,Q,S,A,state,epsilon)

        # Get immediate reward
        r = R[state,action]
        #print("Reward for taking action '{}' from state '{}': {}".
↪format(A[action], S[state], r))

        # Sum the reward
        r_metric += r

        # Update the state - move agent
        old_state = state # Store old state

        state = select_step(state,action) # Get new state
        #print("After taking action '{}' from state '{}', new state is '{}'".
↪format(A[action], S[old_state], S[state]))
```

```
        # Update Q-Matrix
        Q = update_qvalue(alpha,gamma,Q,state,old_state,r,action)

        # print('Q matrix updated: \n\n {}'.format(Q))
        if episode % 250 == 0:
            actions.append(action)

        if S[state] == goal_state:
            # print("Goal state '{}' reached at episode '{}'. Ending episode.".
↪format(goal_state, episode))
            actions.append(6)  # appends the number 6 if the agent reaches the␣
↪goal
            goal += 1
            goals[-1] = goal
            break
        elif S[state] in hole_state:
            # print("Fell into a hole :( Ending episode.")
            actions.append(5)  # appends the number 5 if the agent falls into a␣
↪hole
            break

    # Store metrics to lists
    ts_ep_list.append(timestep) # Number of timesteps
    reward_ep_list.append(r_metric) # Total episode rewards

    # Exploration rate decay
    epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.
↪exp(-epsilon_decay_rate*episode)
    eps_decay.append(epsilon) # appends the updated epsilon to a list (used to␣
↪plot this metric)

    state = S[0] # Start again from the beginning
    # print('Episode {} finished. Q matrix values:\n{}'.format(episode,Q.
↪round(1)))
```

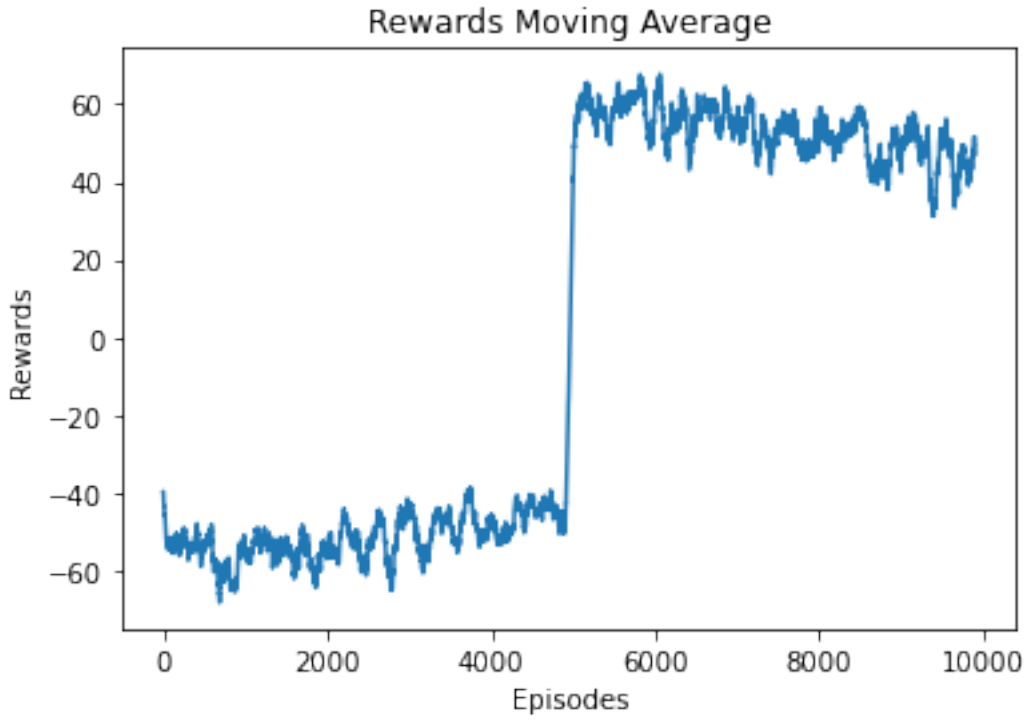
[ ]:
```
# displays the agent movements (requires pygame: pip install pygame)
# the character's runs are displayed for every 250 epochs. Initially the␣
 ↪character explores the environment \
# after about 1 minute and 30 seconds the character starts hitting the goal,␣
 ↪and it will do the same in most sequent runs.
# The whole training is displayed in about 2 minutes and 15 seconds.
display_episodes(actions)
```

Print the Q Matrix

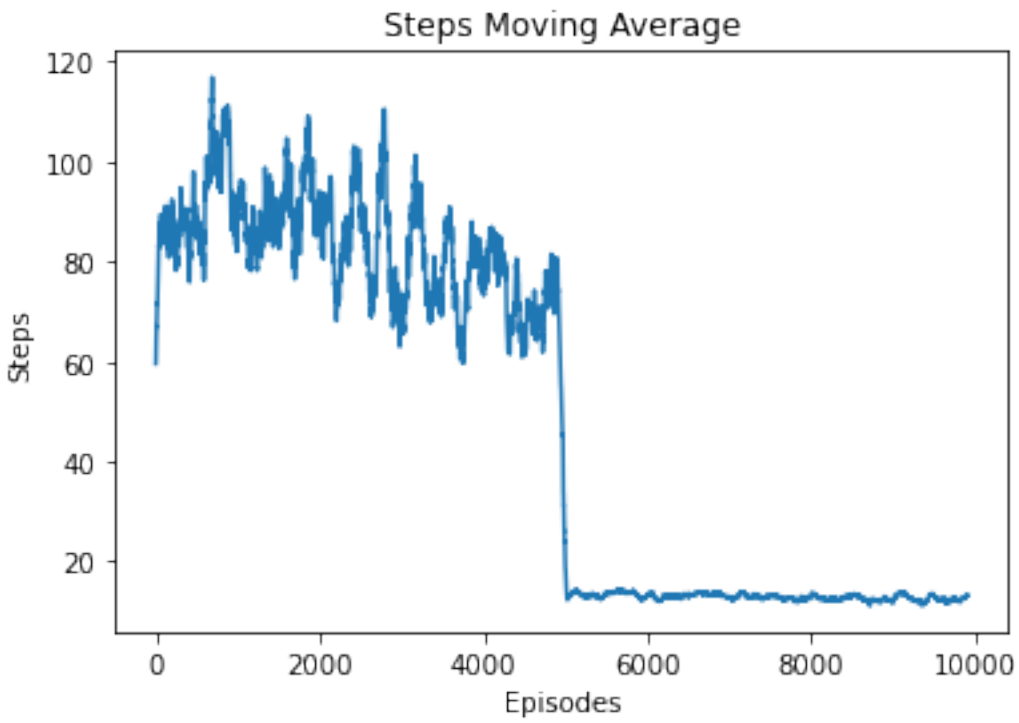
[ ]:
```
# prints the final Q matrix
print('Final Q matrix: \n{}'.format(Q.round(2)))
```

Plots

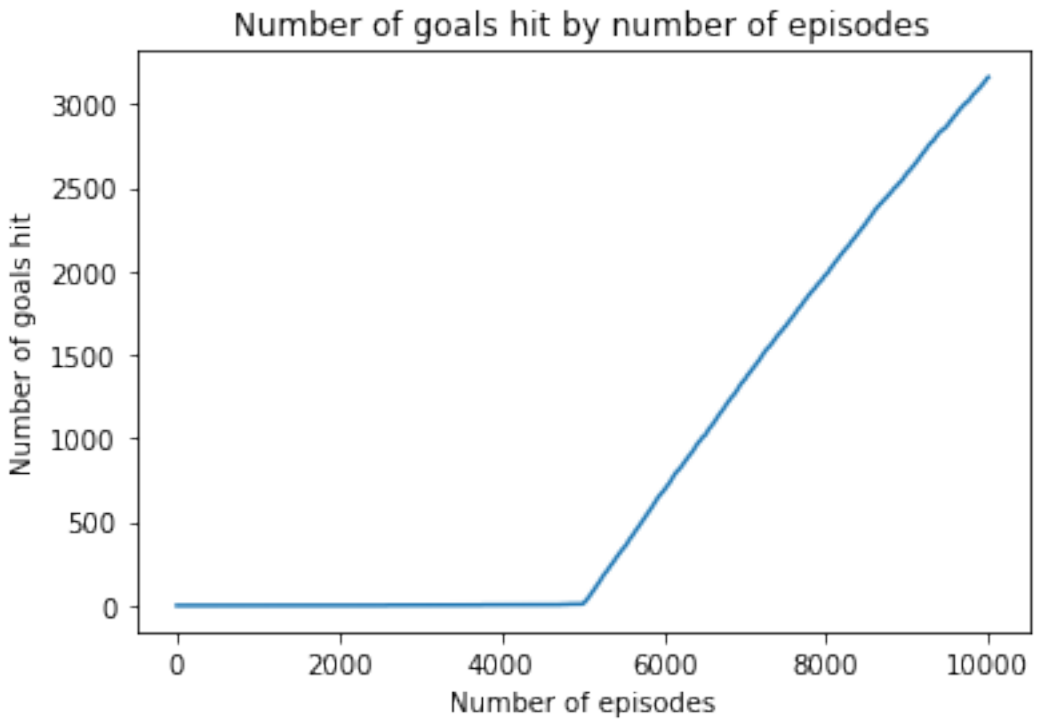
[6]:
```
# plots the 100-episode reward average
reward_ma(reward_ep_list, 100, True)
```



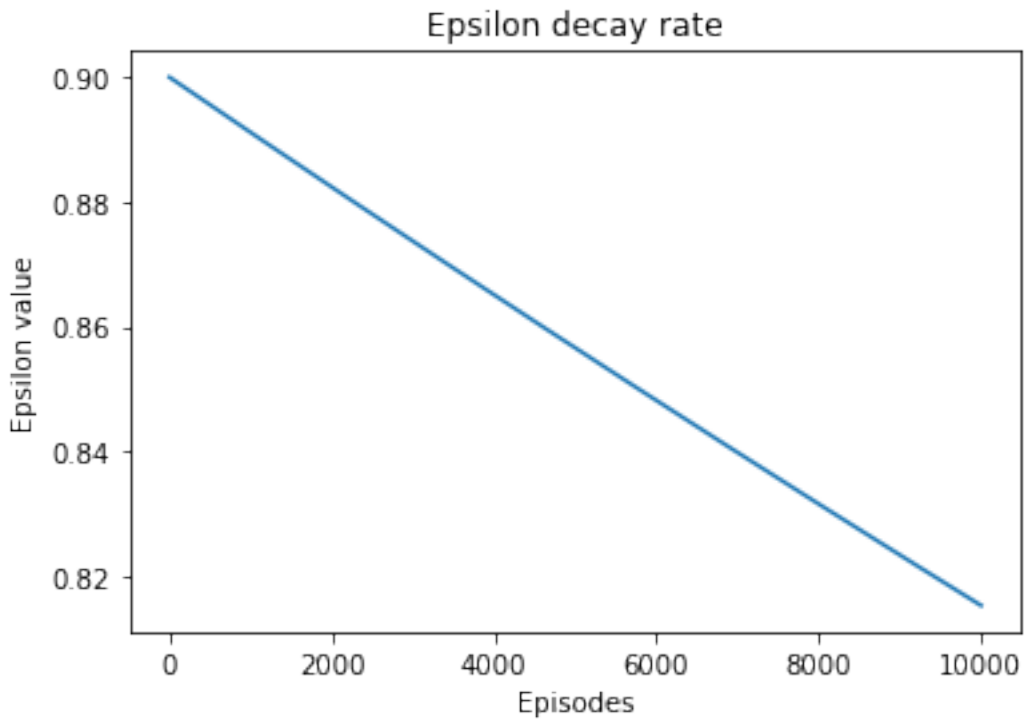
[7]:
```
# plot the number of steps by number of episodes
timestep_ma(ts_ep_list, 100, True)
```



[8]:
```
# plot the number of goals hit by number of episodes
plt.plot(np.arange(num_ep), goals)
plt.title('Number of goals hit by number of episodes')
plt.ylabel('Number of goals hit')
plt.xlabel('Number of episodes');
```



[9]:
```
# plots the epsilon-greedy policy decay rate
plt.plot(np.arange(num_ep), eps_decay)
plt.ylabel('Epsilon value')
plt.xlabel('Episodes')
plt.title('Epsilon decay rate');
```



Grid Search

[5]:
```
# sets hyperparameters for the grid search
alpha_grid = [0.9, 0.5, 0.1]
gamma_grid = [0.9, 0.5, 0.1]
eps_decay_rate_grid = [0.00001, 0.0001, 0.001]

# stars the grid search
for a in alpha_grid:
    for g in gamma_grid:
        for d in eps_decay_rate_grid:
            trainig_grid(a, g, d)  # function imported from qlearning_functions
```

```
Goals hit: 3102 | Cumulative reward:   -2605.0 | AVG steps: 49 | Alpha: 0.9 |
Gamma: 0.9 | Epsilon_s: 0.9 | Epsilon_f: 0.82 | Decay Rate: 1e-05
Goals hit:  354 | Cumulative reward: -184988.0 | AVG steps: 25 | Alpha: 0.9 |
Gamma: 0.9 | Epsilon_s: 0.9 | Epsilon_f: 0.34 | Decay Rate: 0.0001
Goals hit:    1 | Cumulative reward: -155974.5 | AVG steps: 11 | Alpha: 0.9 |
Gamma: 0.9 | Epsilon_s: 0.9 | Epsilon_f: 0.01 | Decay Rate: 0.001
Goals hit:  268 | Cumulative reward: -417933.5 | AVG steps: 69 | Alpha: 0.9 |
Gamma: 0.5 | Epsilon_s: 0.9 | Epsilon_f: 0.82 | Decay Rate: 1e-05
Goals hit:   50 | Cumulative reward: -236737.5 | AVG steps: 28 | Alpha: 0.9 |
Gamma: 0.5 | Epsilon_s: 0.9 | Epsilon_f: 0.34 | Decay Rate: 0.0001
Goals hit:    0 | Cumulative reward: -154896.5 | AVG steps: 11 | Alpha: 0.9 |
```

```
Gamma: 0.5 | Epsilon_s: 0.9 | Epsilon_f: 0.01 | Decay Rate: 0.001
Goals hit:   24 | Cumulative reward: -456919.0 | AVG steps: 72 | Alpha: 0.9 |
Gamma: 0.1 | Epsilon_s: 0.9 | Epsilon_f: 0.82 | Decay Rate: 1e-05
Goals hit:    5 | Cumulative reward: -242092.0 | AVG steps: 29 | Alpha: 0.9 |
Gamma: 0.1 | Epsilon_s: 0.9 | Epsilon_f: 0.34 | Decay Rate: 0.0001
Goals hit:    0 | Cumulative reward: -155073.0 | AVG steps: 11 | Alpha: 0.9 |
Gamma: 0.1 | Epsilon_s: 0.9 | Epsilon_f: 0.01 | Decay Rate: 0.001
Goals hit: 2991 | Cumulative reward:  -17557.5 | AVG steps: 49 | Alpha: 0.5 |
Gamma: 0.9 | Epsilon_s: 0.9 | Epsilon_f: 0.82 | Decay Rate: 1e-05
Goals hit:  477 | Cumulative reward: -166014.5 | AVG steps: 24 | Alpha: 0.5 |
Gamma: 0.9 | Epsilon_s: 0.9 | Epsilon_f: 0.34 | Decay Rate: 0.0001
Goals hit:    0 | Cumulative reward: -154875.0 | AVG steps: 11 | Alpha: 0.5 |
Gamma: 0.9 | Epsilon_s: 0.9 | Epsilon_f: 0.01 | Decay Rate: 0.001
Goals hit:  285 | Cumulative reward: -417336.0 | AVG steps: 70 | Alpha: 0.5 |
Gamma: 0.5 | Epsilon_s: 0.9 | Epsilon_f: 0.82 | Decay Rate: 1e-05
Goals hit:   36 | Cumulative reward: -239503.0 | AVG steps: 29 | Alpha: 0.5 |
Gamma: 0.5 | Epsilon_s: 0.9 | Epsilon_f: 0.34 | Decay Rate: 0.0001
Goals hit:    1 | Cumulative reward: -154638.5 | AVG steps: 11 | Alpha: 0.5 |
Gamma: 0.5 | Epsilon_s: 0.9 | Epsilon_f: 0.01 | Decay Rate: 0.001
Goals hit:   21 | Cumulative reward: -454532.5 | AVG steps: 71 | Alpha: 0.5 |
Gamma: 0.1 | Epsilon_s: 0.9 | Epsilon_f: 0.82 | Decay Rate: 1e-05
Goals hit:    7 | Cumulative reward: -242675.0 | AVG steps: 29 | Alpha: 0.5 |
Gamma: 0.1 | Epsilon_s: 0.9 | Epsilon_f: 0.34 | Decay Rate: 0.0001
Goals hit:    0 | Cumulative reward: -155649.5 | AVG steps: 11 | Alpha: 0.5 |
Gamma: 0.1 | Epsilon_s: 0.9 | Epsilon_f: 0.01 | Decay Rate: 0.001
Goals hit:  988 | Cumulative reward: -316020.0 | AVG steps: 65 | Alpha: 0.1 |
Gamma: 0.9 | Epsilon_s: 0.9 | Epsilon_f: 0.82 | Decay Rate: 1e-05
Goals hit:  142 | Cumulative reward: -220442.5 | AVG steps: 27 | Alpha: 0.1 |
Gamma: 0.9 | Epsilon_s: 0.9 | Epsilon_f: 0.34 | Decay Rate: 0.0001
Goals hit:    0 | Cumulative reward: -155055.0 | AVG steps: 11 | Alpha: 0.1 |
Gamma: 0.9 | Epsilon_s: 0.9 | Epsilon_f: 0.01 | Decay Rate: 0.001
Goals hit:   43 | Cumulative reward: -447300.0 | AVG steps: 70 | Alpha: 0.1 |
Gamma: 0.5 | Epsilon_s: 0.9 | Epsilon_f: 0.82 | Decay Rate: 1e-05
Goals hit:   14 | Cumulative reward: -243554.5 | AVG steps: 29 | Alpha: 0.1 |
Gamma: 0.5 | Epsilon_s: 0.9 | Epsilon_f: 0.34 | Decay Rate: 0.0001
Goals hit:    0 | Cumulative reward: -155479.0 | AVG steps: 11 | Alpha: 0.1 |
Gamma: 0.5 | Epsilon_s: 0.9 | Epsilon_f: 0.01 | Decay Rate: 0.001
Goals hit:   20 | Cumulative reward: -464226.5 | AVG steps: 73 | Alpha: 0.1 |
Gamma: 0.1 | Epsilon_s: 0.9 | Epsilon_f: 0.82 | Decay Rate: 1e-05
Goals hit:   15 | Cumulative reward: -241564.5 | AVG steps: 29 | Alpha: 0.1 |
Gamma: 0.1 | Epsilon_s: 0.9 | Epsilon_f: 0.34 | Decay Rate: 0.0001
Goals hit:    1 | Cumulative reward: -154119.0 | AVG steps: 11 | Alpha: 0.1 |
Gamma: 0.1 | Epsilon_s: 0.9 | Epsilon_f: 0.01 | Decay Rate: 0.001
```

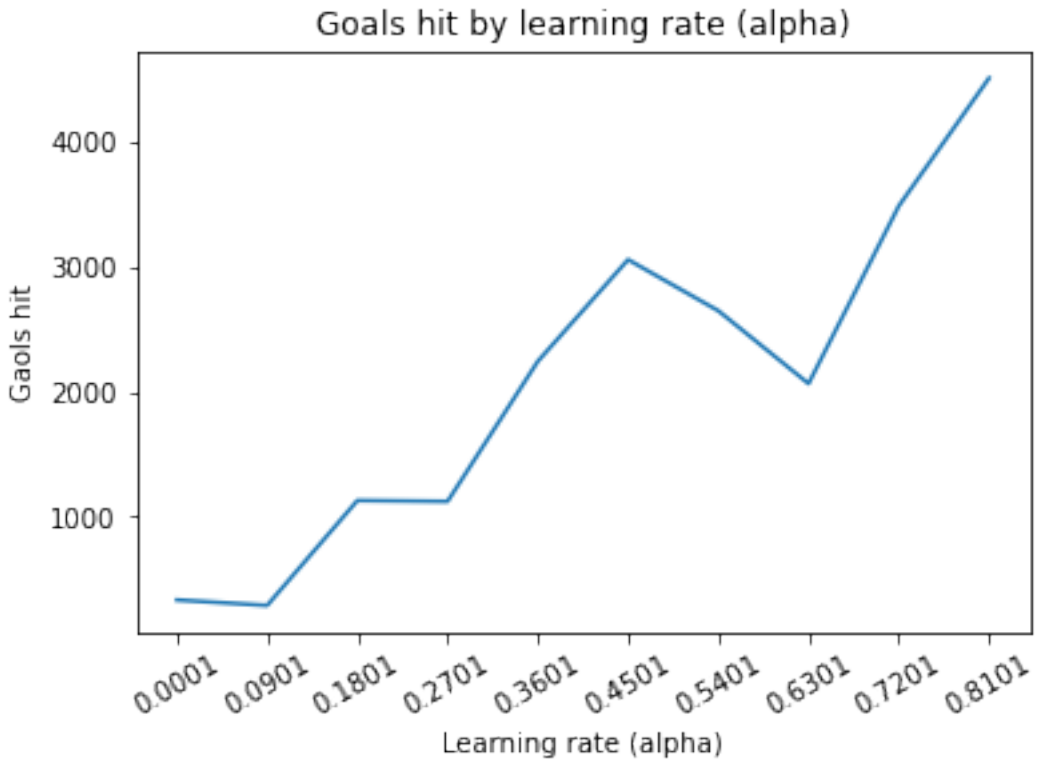
Testing the hyperparameters

[141]:
```
goals_print = [] # List containing goals to print
steps_print = [] # List containing steps to print
reward_print = [] # List containing rewards to print
gamma = 0.9
alpha = 0.9
epsilon = 0.9
d = 0.00001

num_ep = 10000
num_timestep = 500
for num_timestep in np.arange(100, 1000, 100):

    # Initialize our environment
    S, A, R, Q, state, goal_state, hole_state = init_env()

    goal = 0
    r_metric = 0

    for episode in range(num_ep):

        # Initialize/Reset reward metric

        goals.append(goal)

        for timestep in range(num_timestep):
            # Select action
            action = select_action(R,Q,S,A,state,epsilon)
            # Get immediate reward
            r = R[state,action]
            # Sum the reward
            r_metric += r
            # Update the state - move agent
            old_state = state # Store old state
            state = select_step(state,action) # Get new state
            # Update Q-Matrix
            Q = update_qvalue(alpha,gamma,Q,state,old_state,r,action)

            if S[state] == goal_state:
                goal += 1
                break
            elif S[state] in hole_state:
                break

        # Store metrics to lists
        ts_ep_list.append(timestep) # Number of timesteps
        reward_ep_list.append(r_metric) # Total episode rewards
```

```
        # Exploration rate decay
        epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-d*episode)
        eps_decay.append(epsilon) # appends the updated epsilon to a list (used␣
↪to plot this metric)

        state = S[0] # Start again from the beginning


    goals_print.append(goal)
    s = sum(reward_ep_list)
    reward_print.append(r_metric)
    t = sum(ts_ep_list)/num_ep
    steps_print.append(t)
```

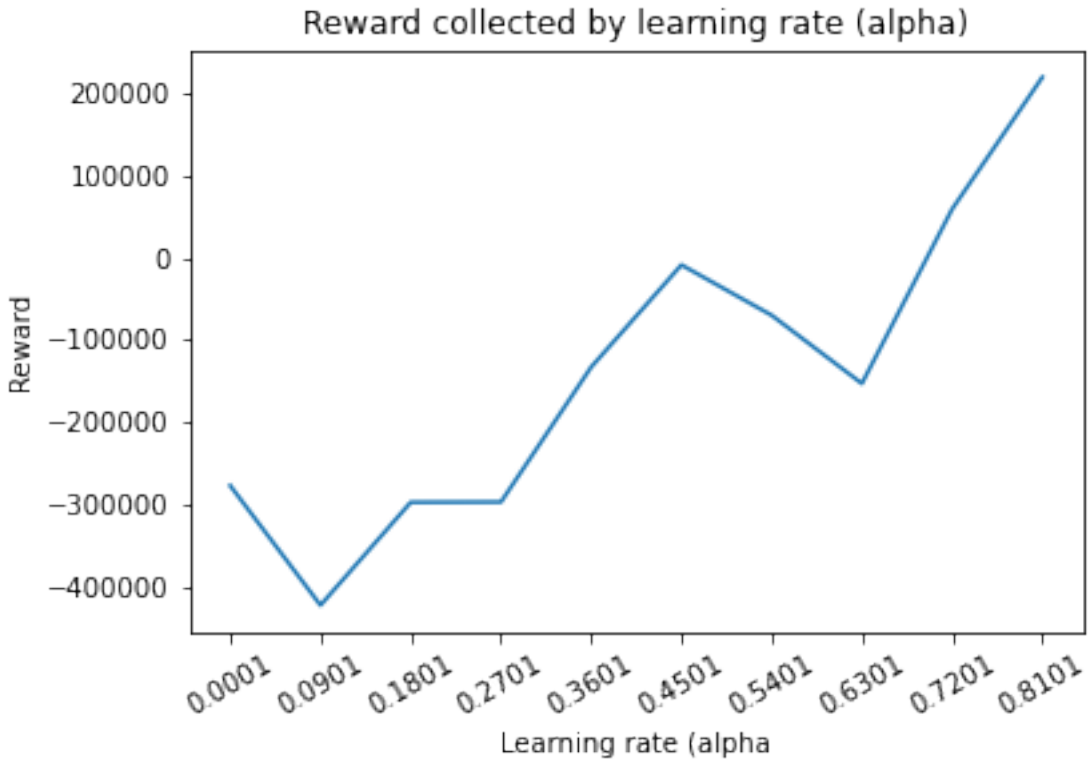
Plotting the model performance at different hyperparameters values

[120]:
```
# goals by learning rate
plt.plot(np.arange(0.0001, 0.9, 0.09), goals_print)
plt.title('Goals hit by learning rate (alpha)')
plt.ylabel('Gaols hit')
plt.xticks(np.arange(0.0001, 0.9, 0.09), rotation = 30)
plt.xlabel('Learning rate (alpha)');
```

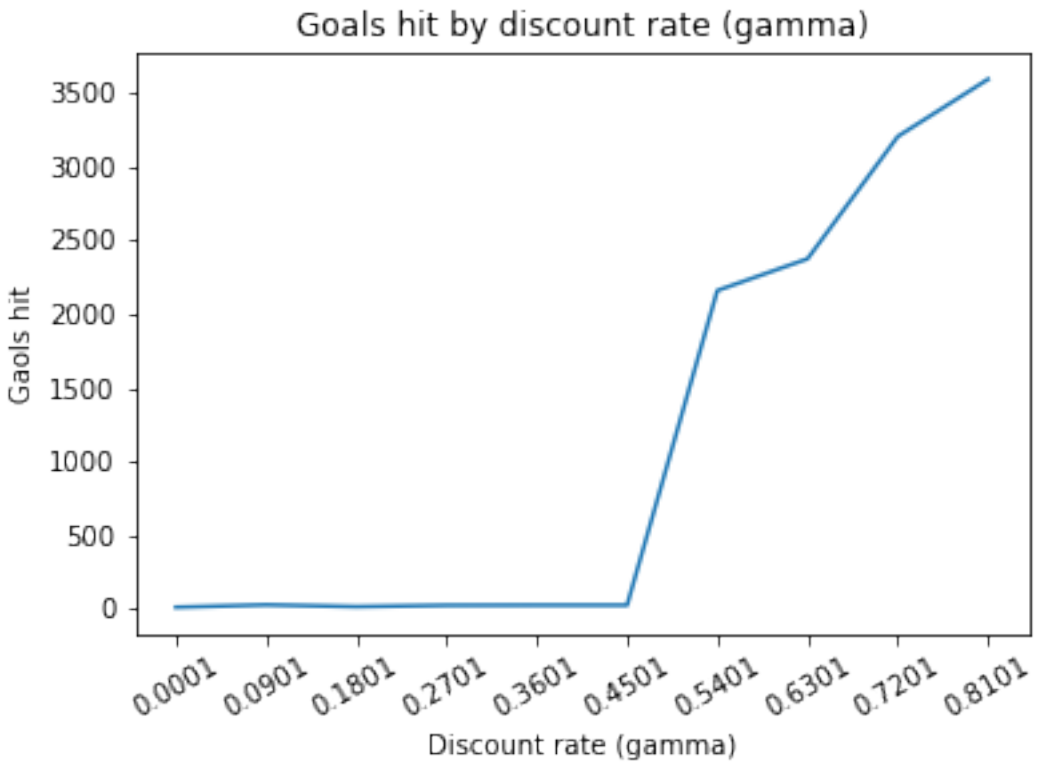


[121]:
```
# rewards by learning rate
plt.plot(np.arange(0.0001, 0.9, 0.09), reward_print)
plt.title('Reward collected by learning rate (alpha)')
plt.ylabel('Reward')
plt.xticks(np.arange(0.0001, 0.9, 0.09), rotation = 30)
plt.xlabel('Learning rate (alpha')
```

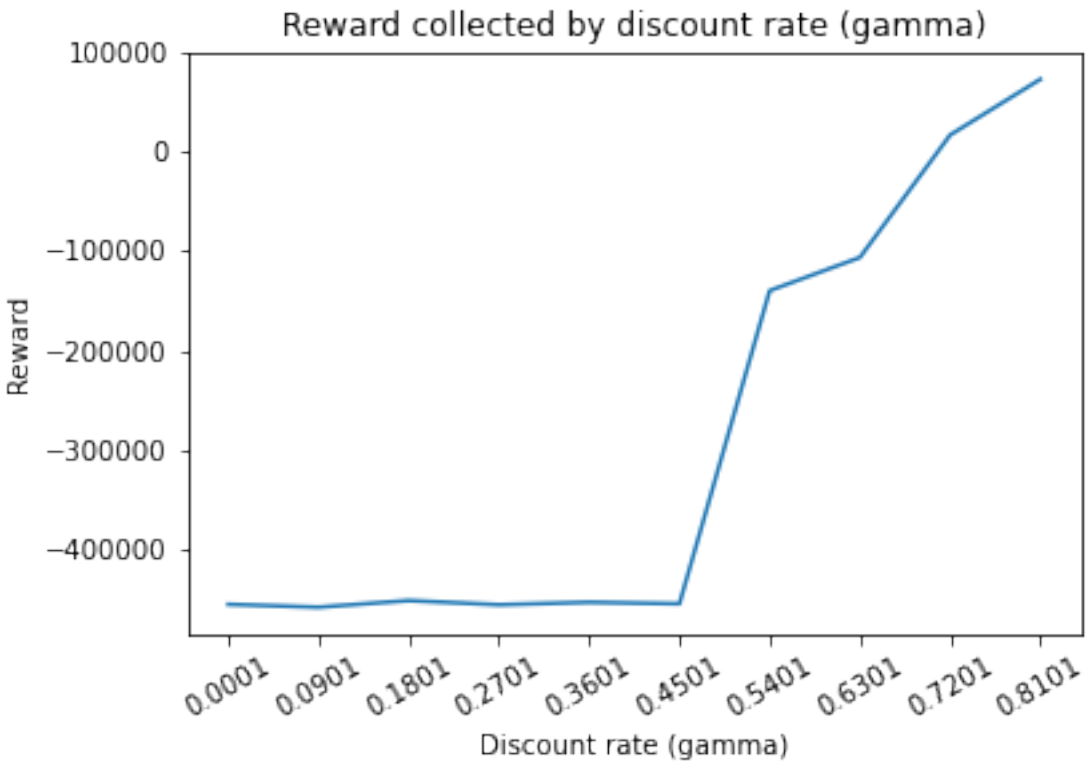
[121]:
```
Text(0.5, 0, 'Learning rate (alpha')
```



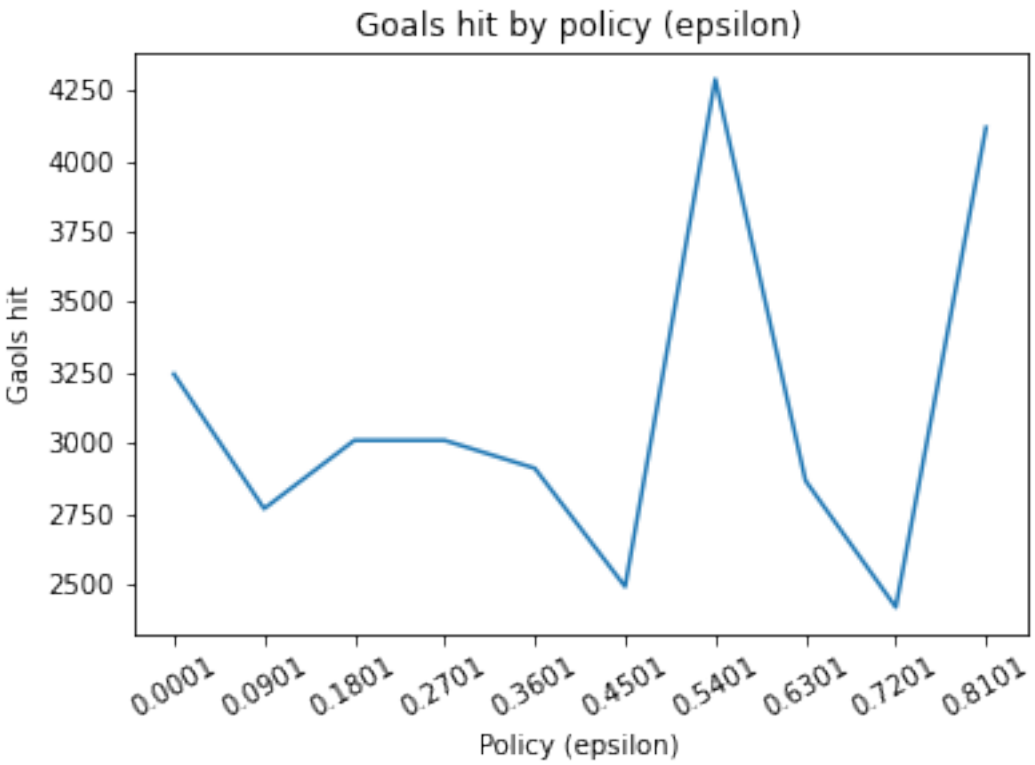
[111]:
```
# goals by gamma
plt.plot(np.arange(0.0001, 0.9, 0.09), goals_print)
plt.title('Goals hit by discount rate (gamma)')
plt.ylabel('Gaols hit')
plt.xticks(np.arange(0.0001, 0.9, 0.09), rotation = 30)
plt.xlabel('Discount rate (gamma)');
```



[113]:
```
# rewards by gamma
plt.plot(np.arange(0.0001, 0.9, 0.09), reward_print)
plt.title('Reward collected by discount rate (gamma)')
plt.ylabel('Reward')
plt.xticks(np.arange(0.0001, 0.9, 0.09), rotation = 30)
plt.xlabel('Discount rate (gamma)');
```

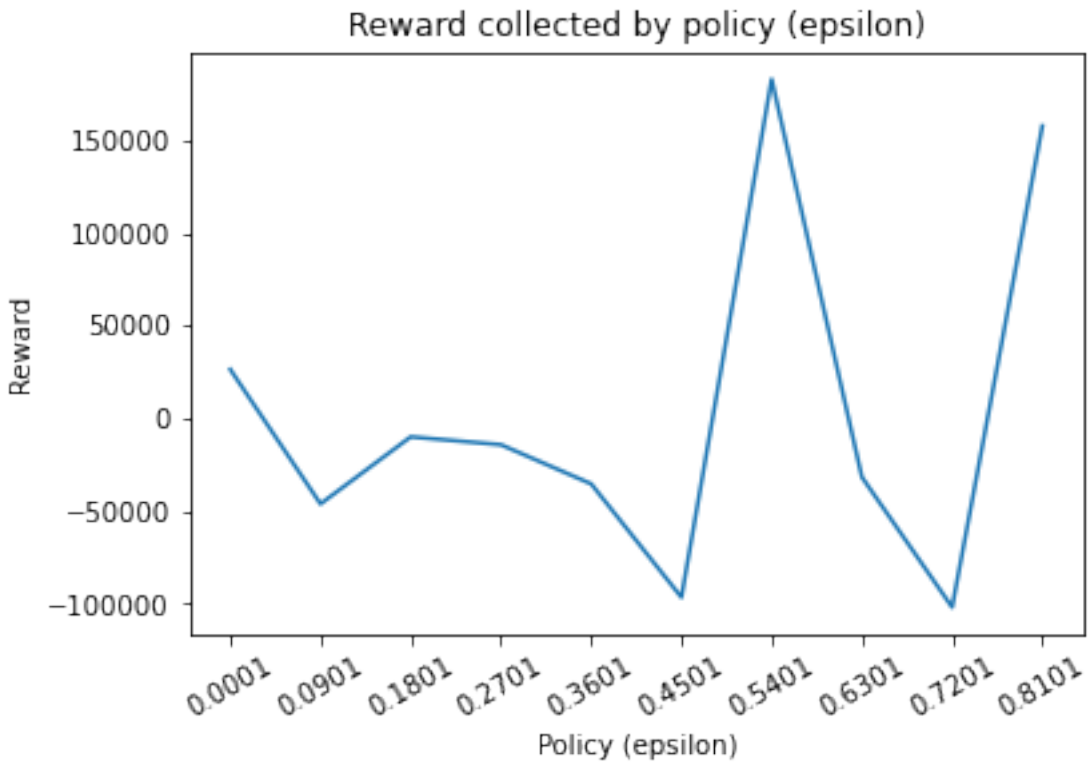


[123]:
```
# goals by epsilon
plt.plot(np.arange(0.0001, 0.9, 0.09), goals_print)
plt.title('Goals hit by policy (epsilon)')
plt.ylabel('Gaols hit')
plt.xticks(np.arange(0.0001, 0.9, 0.09), rotation = 30)
plt.xlabel('Policy (epsilon)');
```

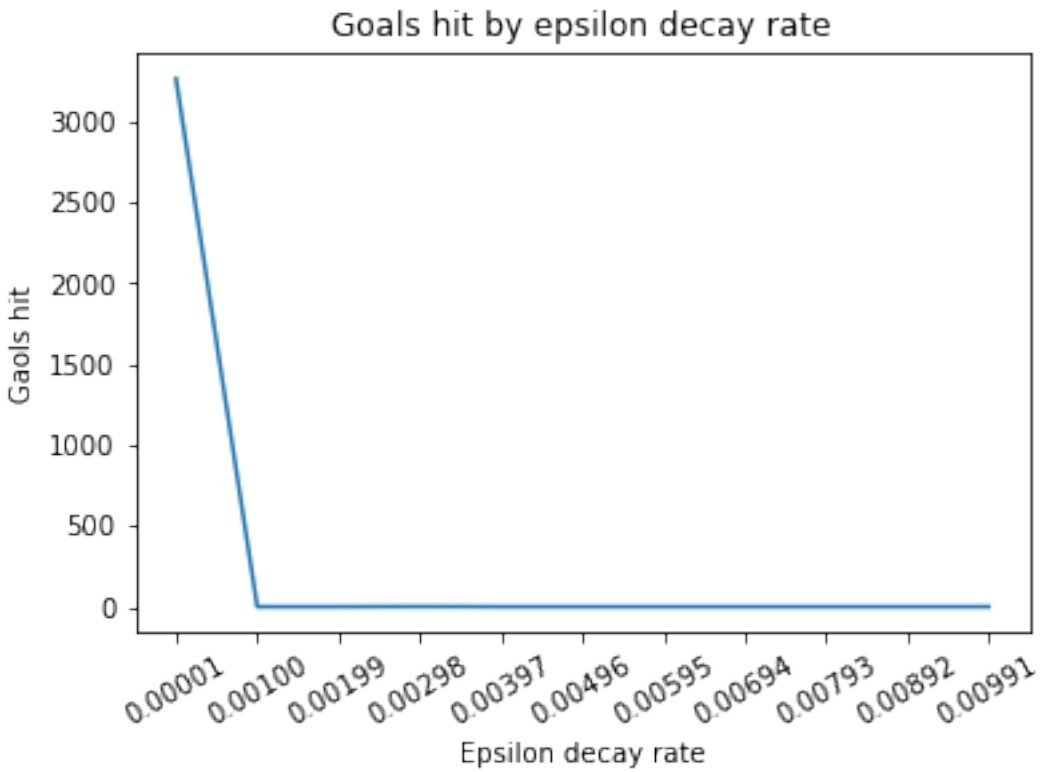


[124]:
```
# rewards by epsilon
plt.plot(np.arange(0.0001, 0.9, 0.09), reward_print)
plt.title('Reward collected by policy (epsilon)')
plt.ylabel('Reward')
plt.xticks(np.arange(0.0001, 0.9, 0.09), rotation = 30)
plt.xlabel('Policy (epsilon)')
```

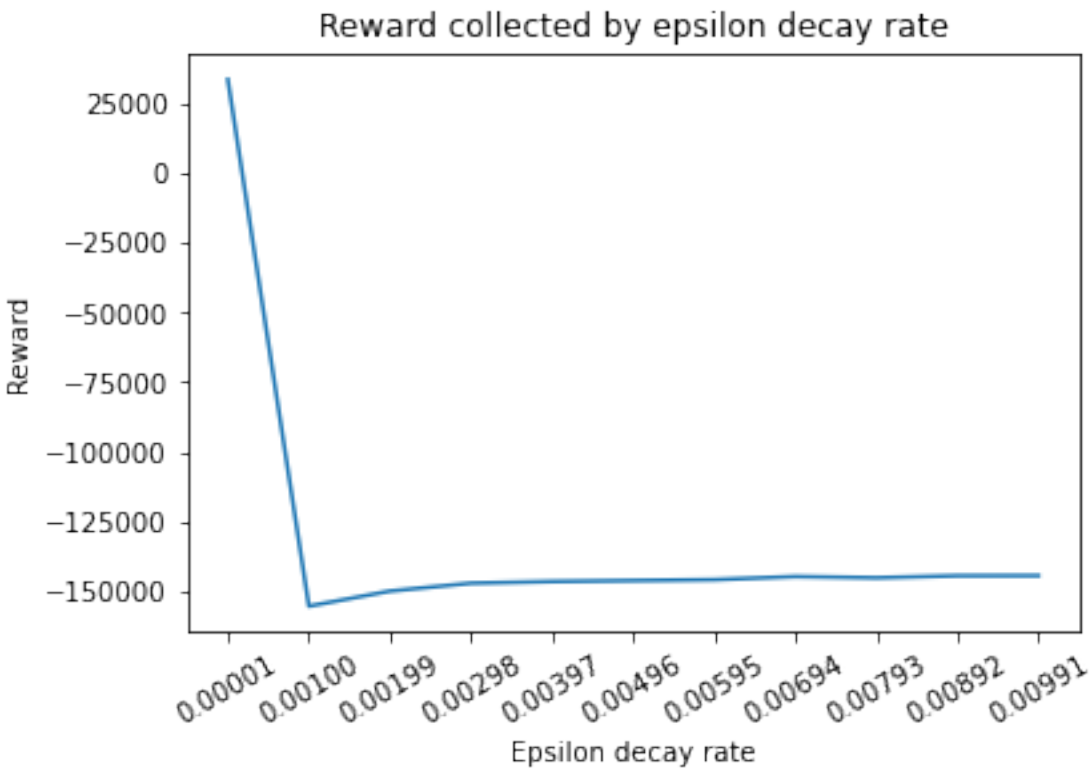
[124]:
```
Text(0.5, 0, 'Policy (epsilon)')
```



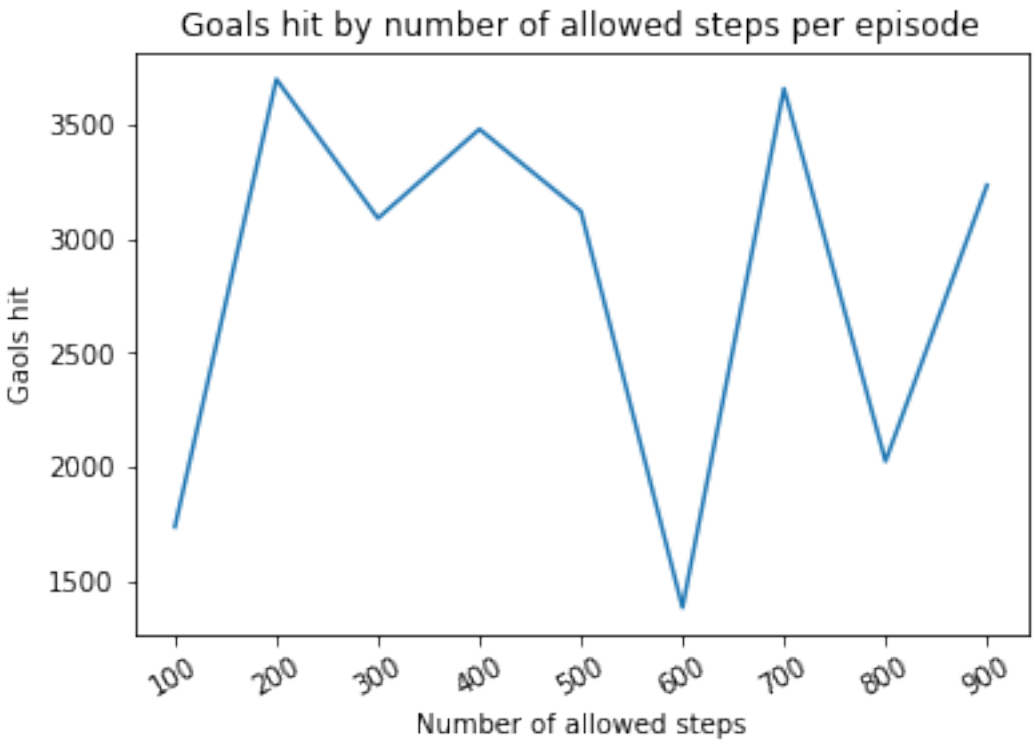
[139]:
```
# goals by decay rate
plt.plot(np.arange(0.00001, 0.01, 0.00099), goals_print)
plt.title('Goals hit by epsilon decay rate')
plt.ylabel('Gaols hit')
plt.xticks(np.arange(0.00001, 0.01, 0.00099), rotation = 30)
plt.xlabel('Epsilon decay rate');
```



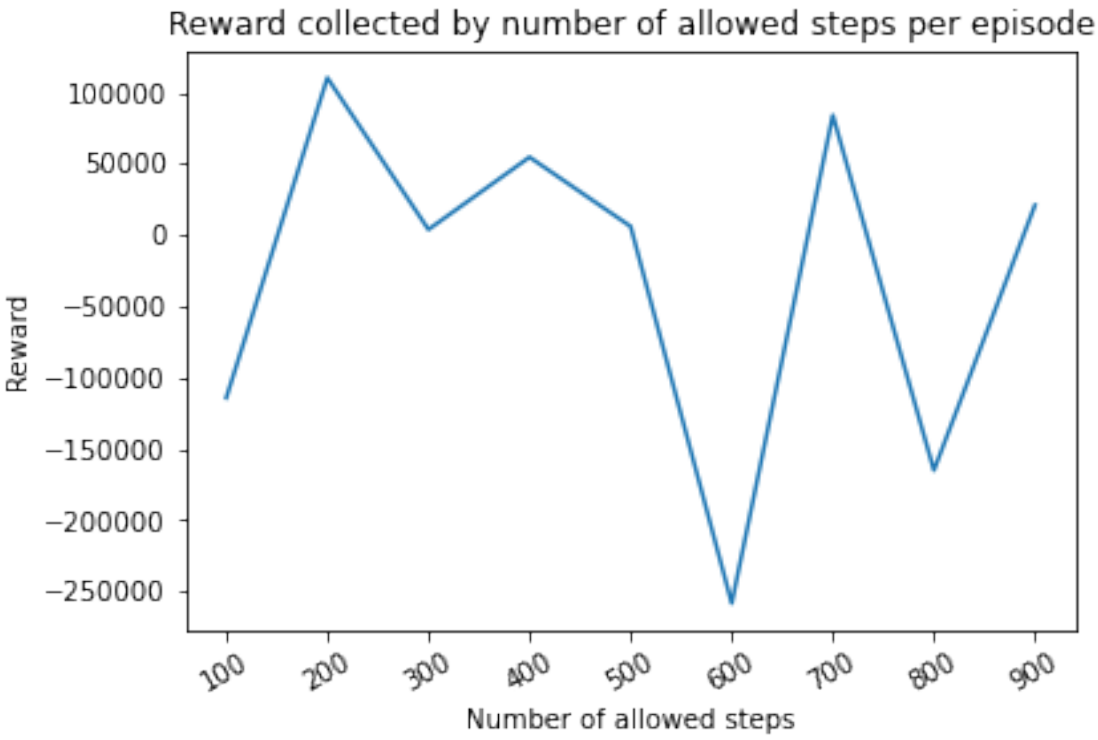
[140]:
```
# rewards by decay rate
plt.plot(np.arange(0.00001, 0.01, 0.00099), reward_print)
plt.title('Reward collected by epsilon decay rate')
plt.ylabel('Reward')
plt.xticks(np.arange(0.00001, 0.01, 0.00099), rotation = 30)
plt.xlabel('Epsilon decay rate');
```



[142]:
```
# goals by timesteps
plt.plot(np.arange(100, 1000, 100), goals_print)
plt.title('Goals hit by number of allowed steps per episode')
plt.ylabel('Gaols hit')
plt.xticks(np.arange(100, 1000, 100), rotation = 30)
plt.xlabel('Number of allowed steps');
```



[143]:
```
# rewards by timesteps
plt.plot(np.arange(100, 1000, 100), reward_print)
plt.title('Reward collected by number of allowed steps per episode')
plt.ylabel('Reward')
plt.xticks(np.arange(100, 1000, 100), rotation = 30)
plt.xlabel('Number of allowed steps');
```


Reward collected by number of allowed steps per episode
Reward
100000
50000
0
−50000
−100000
−150000
−200000
−250000
100
200
300
400
500
600
700
800
900
Number of allowed steps

DRL_Part_2

April 20, 2022

1 Deep Reinforcement Learning Coursework

1.0.1 Alessandro Alviani - alessandro.alviani@city.ac.uk

1.0.2 Dimitrios Megkos - dimitrios.megkos@city.ac.uk

1.0.3 Part 2: Deep Q-Learning

OpenAI Gym: Lunar Lander OpenAI description: Landing pad is always at coordinates (0,0). Coordinates are the first two numbers in state vector. Reward for moving from the top of the screen to landing pad and zero speed is about 100..140 points. If lander moves away from landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Solved is 200 points. Landing outside landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Four discrete actions available: do nothing, fire left orientation engine, fire main engine, fire right orientation engine.

Import Libraries

[3]:
```
# General imports
from itertools import count
# PyTorch import
import torch
import torch.optim as optim
import torch.nn.functional as F
# Import custom classes and functions
from deepqlearning_functions import EnvManager, DQN, Experience,␣
 ↪extract_tensors, ReplayMemory, EpsilonGreedyValue, DRLAgent, QValues,␣
 ↪plot_cma, reward_ma
# Seed for reproducibility
import numpy as np
import matplotlib.pyplot as plt
import random
import os
torch.manual_seed(10)
```

[3]:
```
<torch._C.Generator at 0x7fceb06e8cf0>
```

Initialize Parameters

[5]:
```
# Set device based on wether there is a GPU or not
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Parameters for the Environment
# https://gym.openai.com/envs/LunarLander-v2/
env_name = 'LunarLander-v2'
env = EnvManager(env_name, device) # load the environment manager helper

# Parameters for Epsilon Greedy Strategy
max_epsilon = 1
min_epsilon = 0.01
epsilon_decay_rate = 0.001
strategy = EpsilonGreedyValue(max_epsilon, min_epsilon, epsilon_decay_rate)

# Parameters for Replay Memory
memory_size = 100000
batch_size = 256
memory = ReplayMemory(memory_size)

# Parameters that control the Deep Q-Learning
gamma = 0.999
lr = 0.001
num_episodes = 100


# Parameters for the architecture of the NN
hidden_neurons = 32 # number of neurons on the hidden layers
n_actions = env.env.action_space.n # for nn output, the number of actions
target_update = 10 # target nn weights update every n episodes

# Parameters for performance metrics
episode_durations = [] # variable to store the duration of each episode
episode_rewards = [] # variable to store rewards of each episode
```

Create Agent and Neural Networks

[7]:
```
# Create the Deep Reinforcement Learning agent
agent = DRLAgent(strategy, env.num_actions_available(), device)

# Create the Deep Q-Network
policy_net = DQN(env.n_state_features(),hidden_neurons,n_actions,False).
 ↪to(device) # DQN for current state's Q-Values

# Create the target network for Double Deep Q-Network
target_net = DQN(env.n_state_features(),hidden_neurons,n_actions,False).
 ↪to(device) # DQN for next state's Q-Values
```

```
target_net.load_state_dict(policy_net.state_dict()) # copy policy's net␣
 ↪parameters to target net
target_net.eval() # set target net to eval so it's weights do not update with␣
 ↪back propagation

# choose the optimizer
optimizer = optim.Adam(params=policy_net.parameters(), lr=lr) # Adam optimizer



# Create the Dueling Deep Q-Network
duel_policy_net = DQN(env.n_state_features(),hidden_neurons,n_actions,True).
 ↪to(device) # Dueling DQN for current state's Q-Values
# Create the target network for Dueling Double Deep Q-Network
duel_target_net = DQN(env.n_state_features(),hidden_neurons,n_actions,True).
 ↪to(device) # Dueling DQN for next state's Q-Values
duel_target_net.load_state_dict(duel_policy_net.state_dict()) # copy dueling␣
 ↪policy's net parameters to dueling target net
duel_target_net.eval() # set dueling target net to eval so it's weights do not␣
 ↪update with back propagation

# choose the optimizer
duel_optimizer = optim.Adam(params=duel_policy_net.parameters(), lr=lr) # Adam␣
 ↪optimizer
```

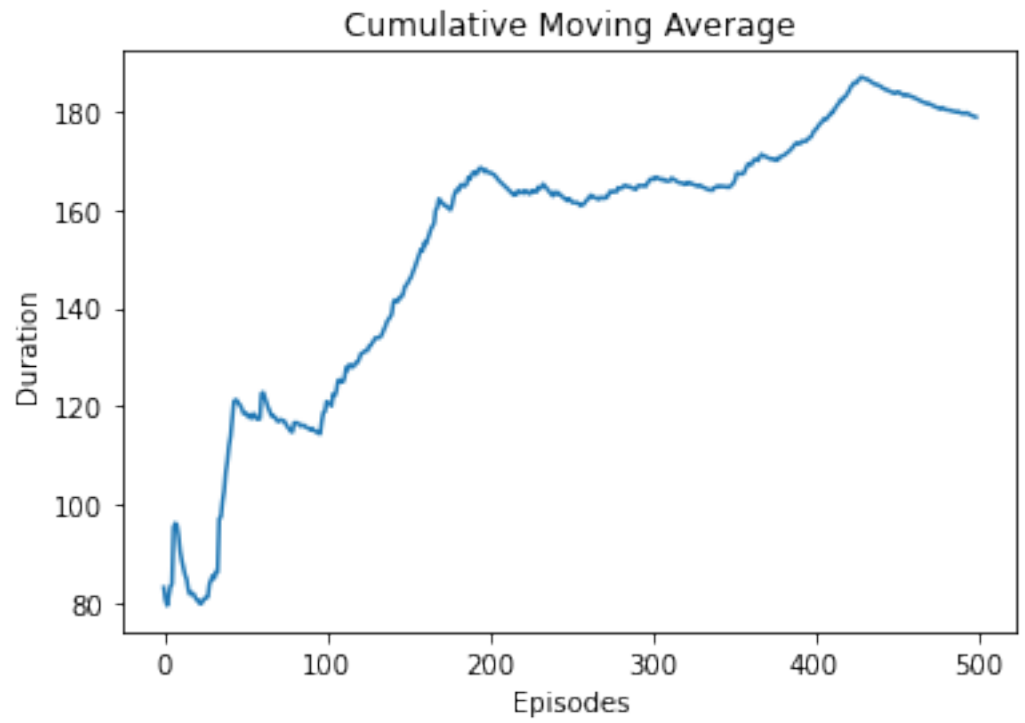
DQN - Begin Training The vanilla form of DQN. Uses Experience Replay and one neural network to calculate current and next state's Q-Values.

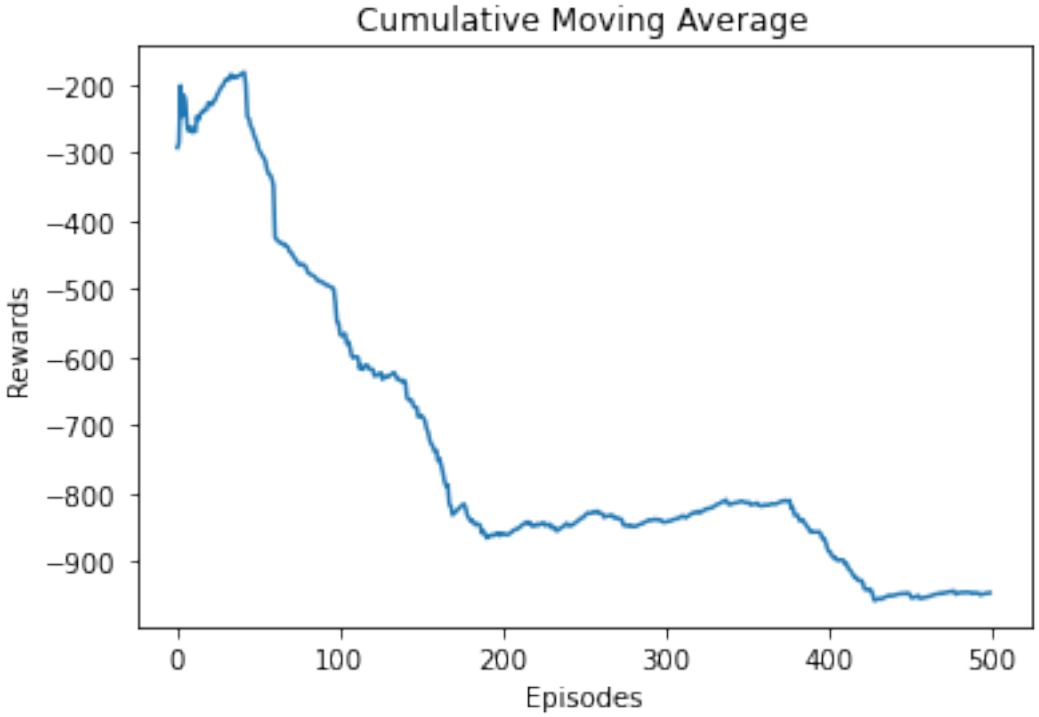
[4]:
```
# Begin training our agent on the selected gaming environment
for episode in range(num_episodes):
    env.reset() # reset the environment on each episode
    state = env.get_state() # get current (initial) state

    r_metric = 0 # initialize/Reset reward metric

    # run timesteps until state is terminal
    for timestep in count():
        #env.render() # render the game image
        action = agent.select_action(state, policy_net) # select an action with␣
 ↪epsilon greedy
        reward = env.take_action(action).float() # get the selected action's␣
 ↪reward
        r_metric += reward.item() # sum the reward
        next_state = env.get_state() # get the new state
        memory.store(Experience(state, action, next_state, reward)) # store␣
 ↪agent's experience
```

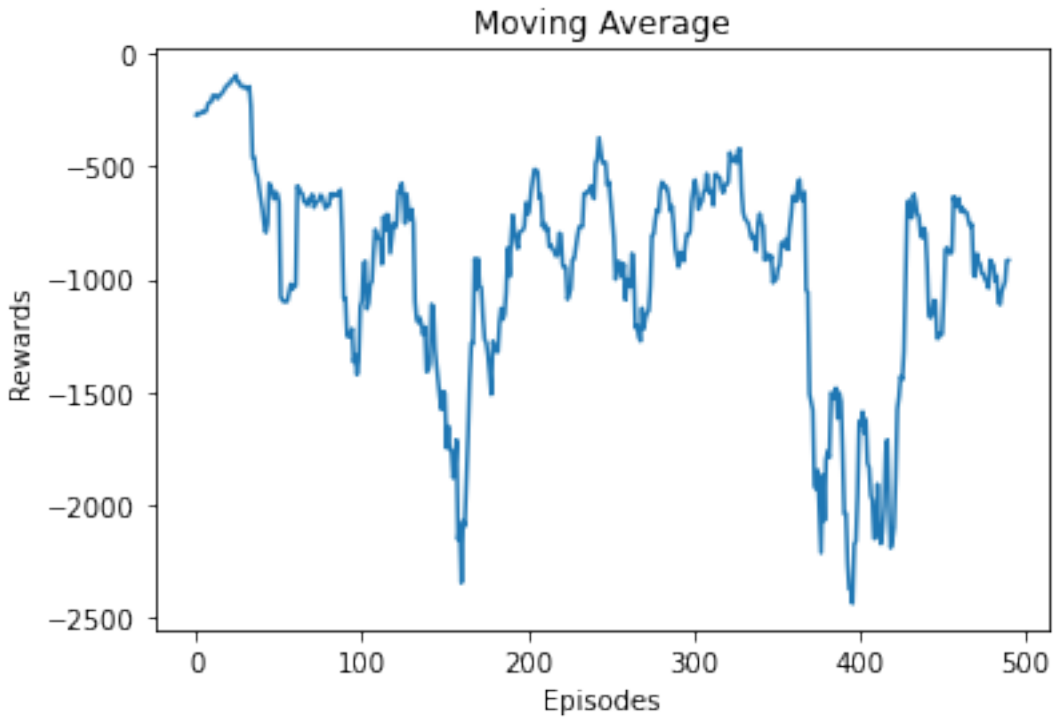
```
        state = next_state # set new state as current state

        # check if there are enough experiences in the memory to sample
        if memory.batch_available(batch_size):
            experiences = memory.get(batch_size) # get a batch of experiences
            states, actions, rewards, next_states =␣
↪extract_tensors(experiences) # convert to tensors for the neural networks
            # Neural Network part
            current_q_values = QValues.get_current(policy_net, states, actions)␣
↪# get Q-Values for current state from policy net
            next_q_values = QValues.get_next(policy_net, next_states) # get␣
↪Q-Values for next state from policy net
            target_q_values = (next_q_values * gamma) + rewards # calculate the␣
↪target Q-Values to use for back propagation
            # Back propagation
            loss = F.mse_loss(current_q_values, target_q_values.unsqueeze(1)) #␣
↪calculate the loss
            optimizer.zero_grad() # reset gradients (required by PyTorch)
            loss.backward() # back propagate loss
            optimizer.step() # update weights

        # check if last action ended the episode
        if env.done:
            episode_durations.append(timestep) # append episode duration
            episode_rewards.append(r_metric) # append episode rewards
            break

    # check if agent solved the game
    #if reward_ma(episode_rewards,100,False) >= 200:
        # break

# close the render window
#env.close()
```

Plot Performance Metrics

[18]:
```
# cumulative moving average
plot_cma(episode_durations, "Duration")
plot_cma(episode_rewards, "Rewards")
# moving average
reward_ma(episode_rewards,10,True)
```


Cumulative Moving Average
Duration
180
160
140
120
100
80
0
100
200
300
400
500
Episodes


Cumulative Moving Average
Rewards
−200
−300
−400
−500
−600
−700
−800
−900
0
100
200
300
400
500
Episodes



Double DQN - Begin Training Uses Experience Replay and two Neural Networks (Policy Network and Target Network) to calculate current state's Q-Values and next state's Q-Values. Target Network's weights are updated every 10 episodes, using Policy Network's weights. Double DQN helps us reduce the overestimation of Q-Values and helps us train faster and have more stable learning.

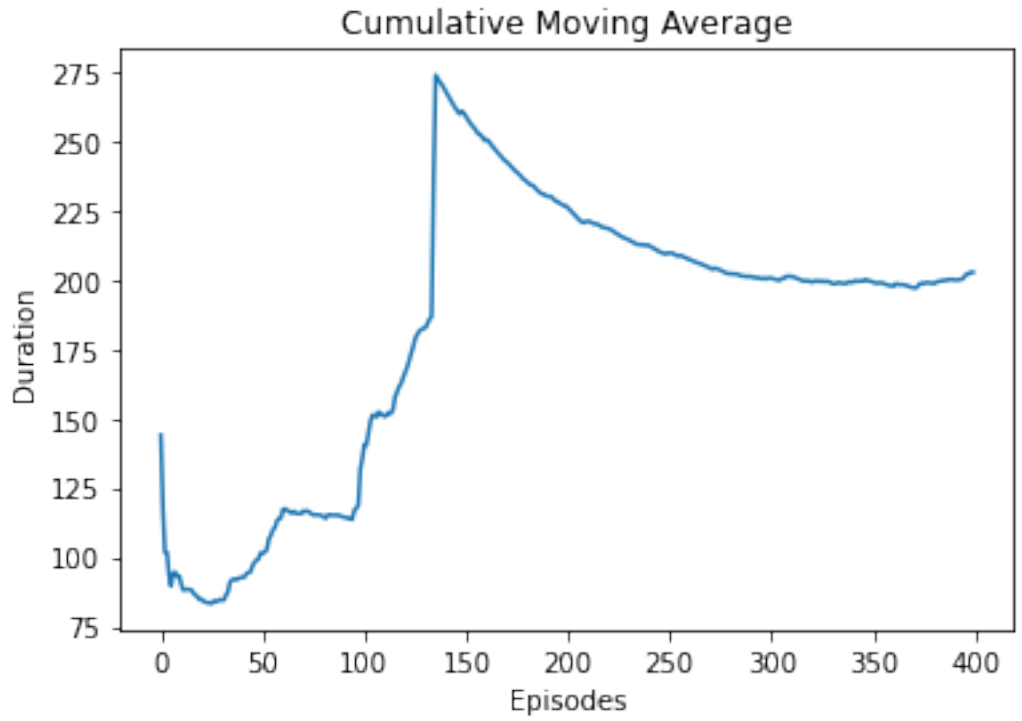
[19]:
```
# Begin training our agent on the selected gaming environment
for episode in range(num_episodes):
    env.reset() # reset the environment on each episode
    state = env.get_state() # get current (initial) state

    r_metric = 0 # initialize/Reset reward metric

    # run timesteps until state is terminal
    for timestep in count():
        #env.render() # render the game image
        action = agent.select_action(state, policy_net) # select an action with␣
↪epsilon greedy
        reward = env.take_action(action).float() # get the selected action's␣
↪reward
        r_metric += reward.item() # sum the reward
        next_state = env.get_state() # get the new state
```

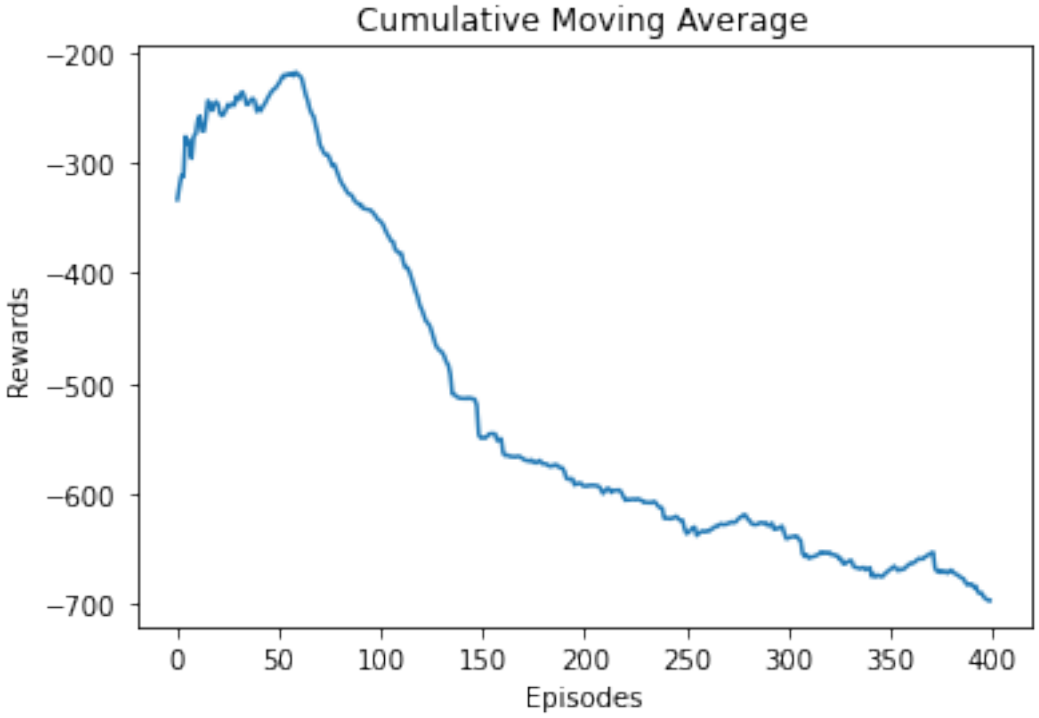
```
        memory.store(Experience(state, action, next_state, reward)) # store␣
↪agent's experience
        state = next_state # set new state as current state

        # check if there are enough experiences in the memory to sample
        if memory.batch_available(batch_size):
            experiences = memory.get(batch_size) # get a batch of experiences
            states, actions, rewards, next_states =␣
↪extract_tensors(experiences) # convert to tensors for the neural networks
            # Neural Network part
            current_q_values = QValues.get_current(policy_net, states, actions)␣
↪# get Q-Values for current state from policy net
            next_q_values = QValues.get_next(target_net, next_states) # get␣
↪Q-Values for next state from target net
            target_q_values = (next_q_values * gamma) + rewards # calculate the␣
↪target Q-Values to use for back propagation
            # Back propagation
            loss = F.mse_loss(current_q_values, target_q_values.unsqueeze(1)) #␣
↪calculate the loss
            optimizer.zero_grad() # reset gradients (required by PyTorch)
            loss.backward() # back propagate loss
            optimizer.step() # update weights

        # check if last action ended the episode
        if env.done:
            episode_durations.append(timestep) # append episode duration
            episode_rewards.append(r_metric) # append episode rewards
            break

    # update target network weights
    if episode % target_update == 0:
        target_net.load_state_dict(policy_net.state_dict()) # copy policy␣
↪network's weights to target network

    # check if agent solved the game
    #if reward_ma(episode_rewards,100,False) >= 200:
       # break

# close the render window
#env.close()
```

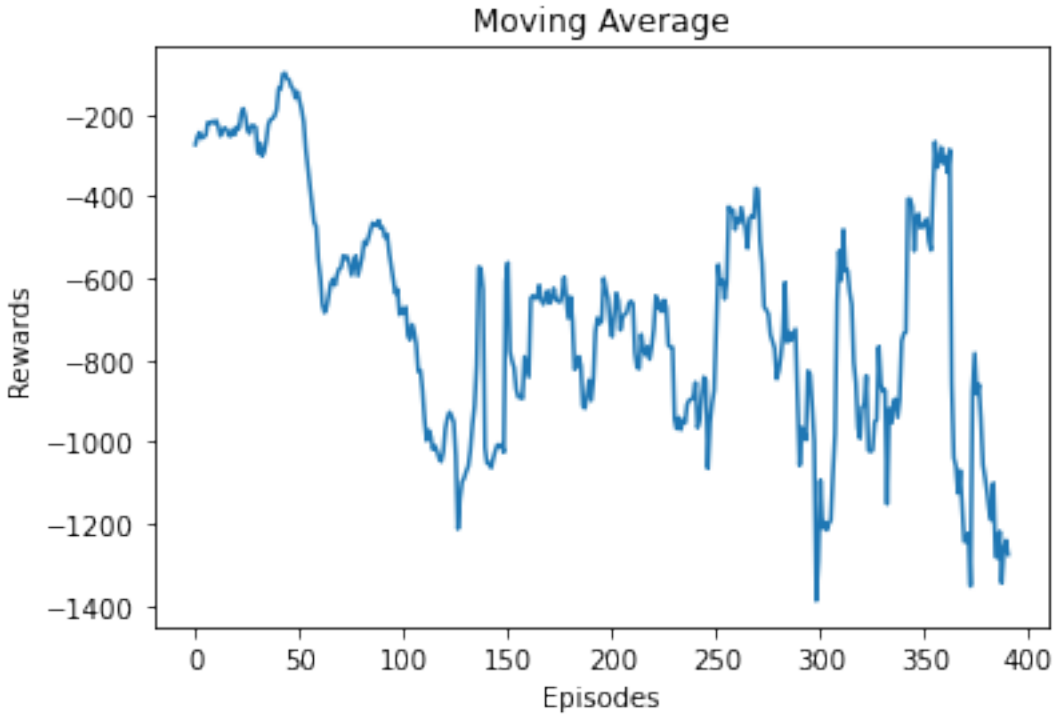
Plot Performance Metrics

[14]:
```
# cumulative moving average
plot_cma(episode_durations, "Duration")
plot_cma(episode_rewards, "Rewards")
# moving average
```

```
reward_ma(episode_rewards,10,True)
```







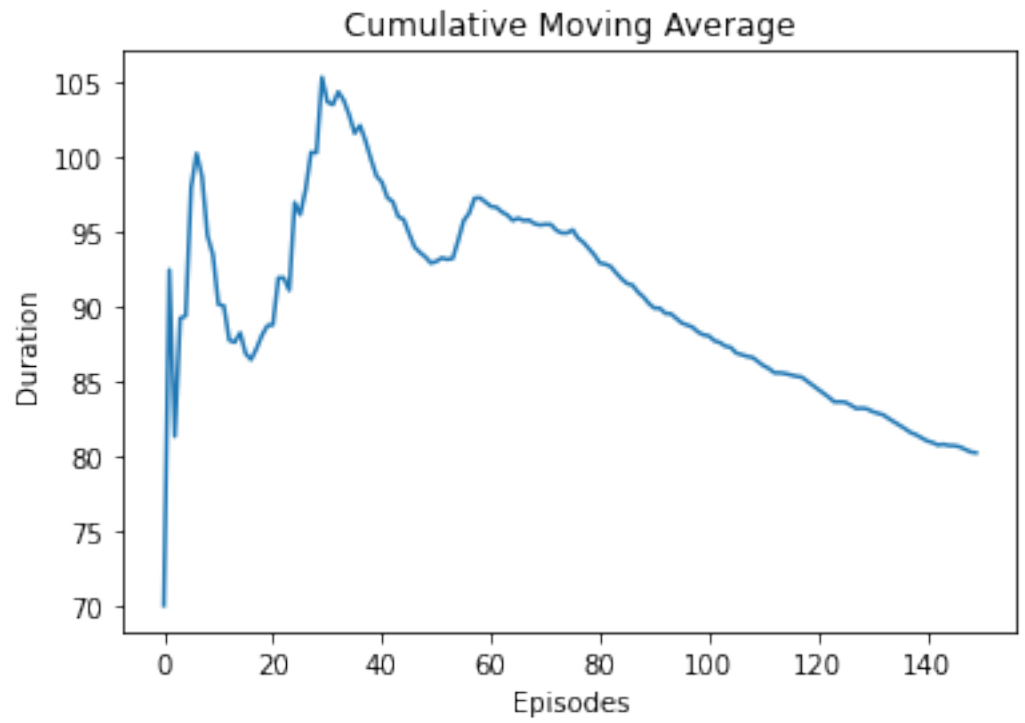
Dueling DQN - Begin Training Uses experience replay and two streams inside the Neural Network: One that estimates the state value V(s), which is the value of being at state s and one that estimates the advantage for each action A(s,a), which is the advantage of taking the action a at state s. In the end, those two streams are combined through a special aggregation to get an estimate of the Q-Value Q(s,a).

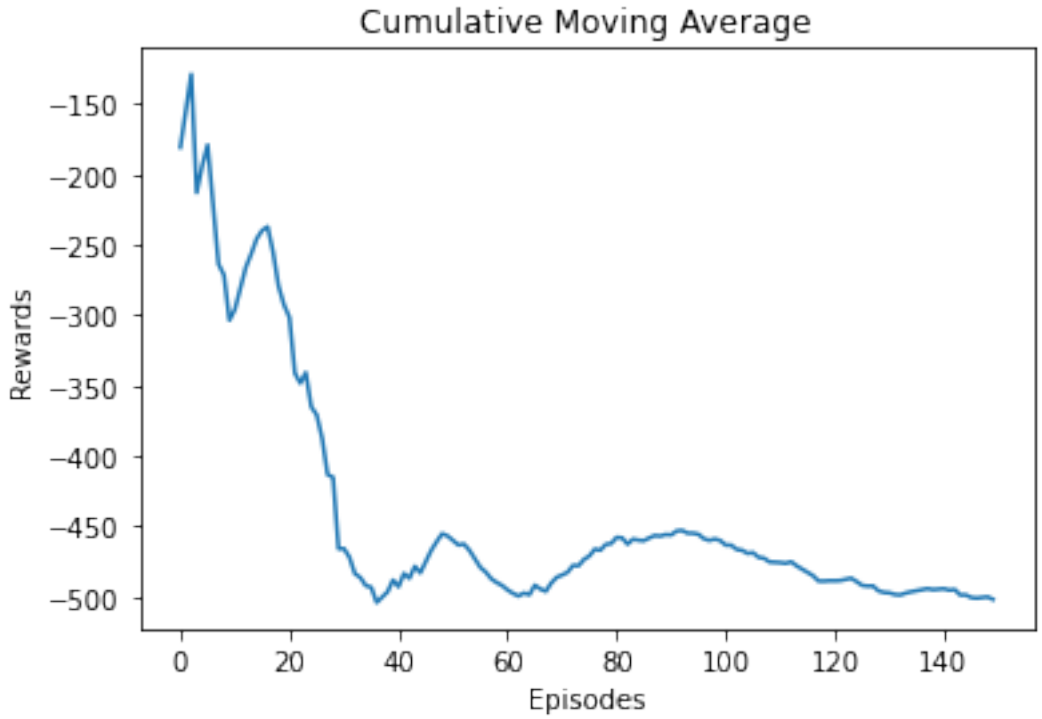
[6]:
```
# Begin training our agent on the selected gaming environment
for episode in range(num_episodes):
    env.reset() # reset the environment on each episode
    state = env.get_state() # get current (initial) state

    r_metric = 0 # initialize/Reset reward metric

    # run timesteps until state is terminal
    for timestep in count():
        #env.render() # render the game image
        action = agent.select_action(state, duel_policy_net) # select an action␣
↪with epsilon greedy
        reward = env.take_action(action).float() # get the selected action's␣
↪reward
        r_metric += reward.item() # sum the reward
```

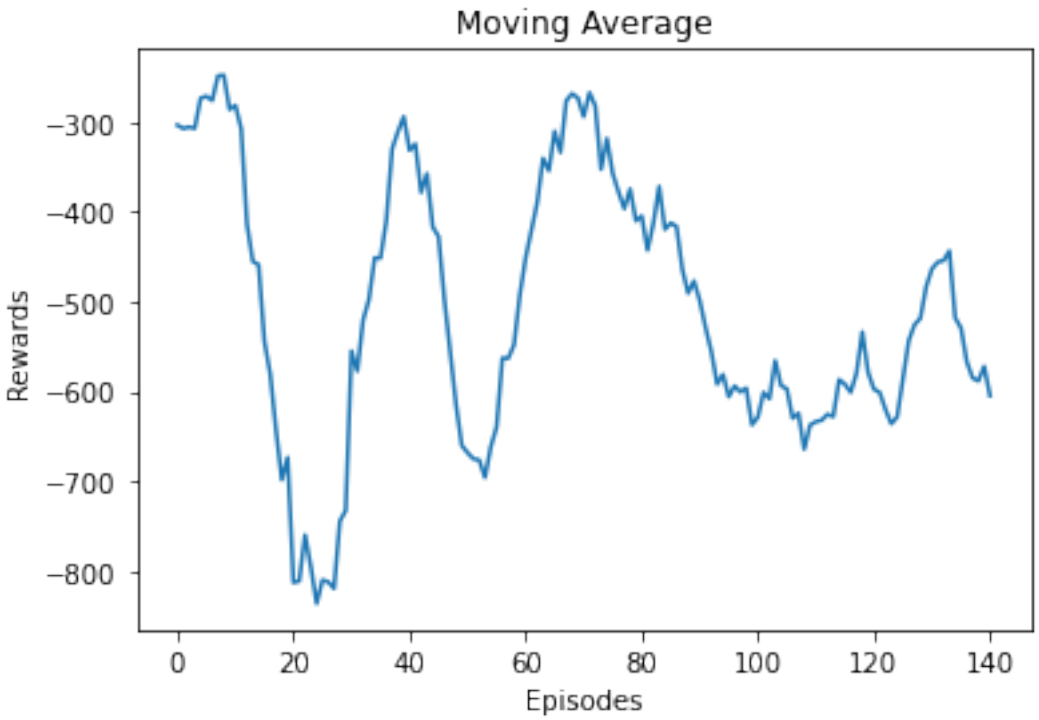
```
        next_state = env.get_state() # get the new state
        memory.store(Experience(state, action, next_state, reward)) # store␣
↪agent's experience
        state = next_state # set new state as current state

        # check if there are enough experiences in the memory to sample
        if memory.batch_available(batch_size):
            experiences = memory.get(batch_size) # get a batch of experiences
            states, actions, rewards, next_states =␣
↪extract_tensors(experiences) # convert to tensors for the neural networks
            # Neural Network part
            current_q_values = QValues.get_current(duel_policy_net, states,␣
↪actions) # get Q-Values for current state from policy net
            next_q_values = QValues.get_next(duel_policy_net, next_states) #␣
↪get Q-Values for next state from policy net
            target_q_values = (next_q_values * gamma) + rewards # calculate the␣
↪target Q-Values to use for back propagation
            # Back propagation
            loss = F.mse_loss(current_q_values, target_q_values.unsqueeze(1)) #␣
↪calculate the loss
            optimizer.zero_grad() # reset gradients (required by PyTorch)
            loss.backward() # back propagate loss
            optimizer.step() # update weights

        # check if last action ended the episode
        if env.done:
            episode_durations.append(timestep) # append episode duration
            episode_rewards.append(r_metric) # append episode rewards
            break

    # check if agent solved the game
    #if reward_ma(episode_rewards,100,False) >= 200:
       # break

# close the render window
#env.close()
```

Plot Performance Metrics

[9]:
```
# cumulative moving average
plot_cma(episode_durations, "Duration")
plot_cma(episode_rewards, "Rewards")
# moving average
reward_ma(episode_rewards,10,True)
```


Cumulative Moving Average
Duration
105
100
95
90
85
80
75
70
0
20
40
60
80
100
120
140
Episodes


Cumulative Moving Average
Rewards
−150
−200
−250
−300
−350
−400
−450
−500
0
20
40
60
80
100
120
140
Episodes



Dueling Double DQN - Begin Training Uses experience replay, two Neural Networks (Policy Network and Target Network) to calculate current state's Q-Values and next state's Q-Values and two streams inside the Neural Networks: One that estimates the state value V(s), which is the value of being at state s and one that estimates the advantage for each action A(s,a), which is the advantage of taking the action a at state s. In the end, those two streams are combined through a special aggregation to get an estimate of the Q-Value Q(s,a).

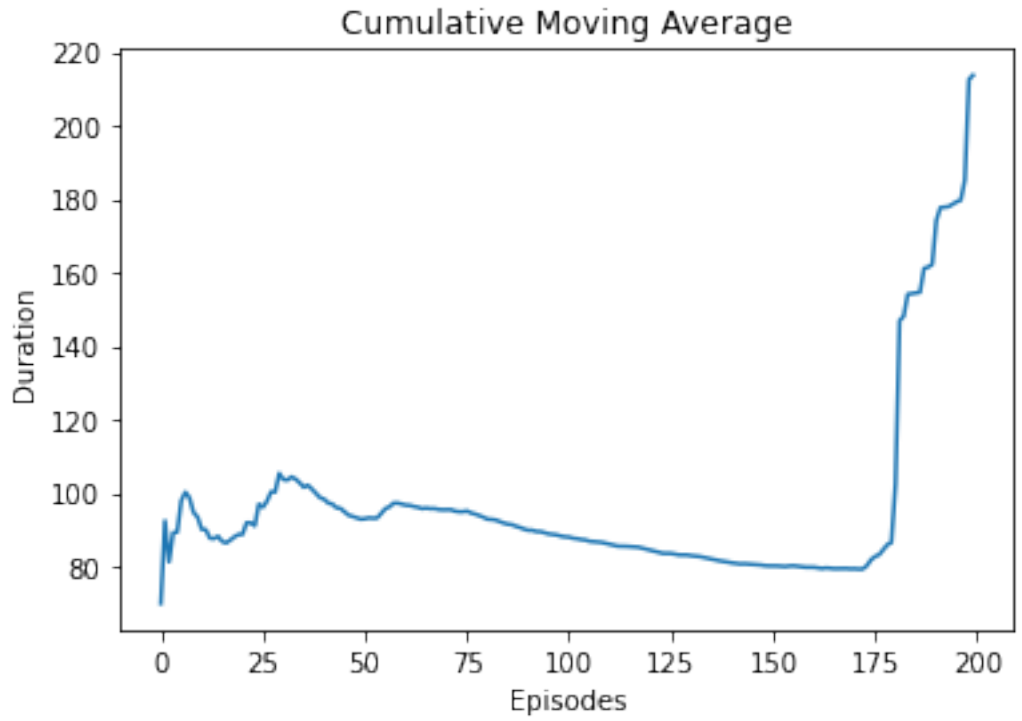
[7]:
```
# Begin training our agent on the selected gaming environment
for episode in range(num_episodes):
    env.reset() # reset the environment on each episode
    state = env.get_state() # get current (initial) state

    r_metric = 0 # initialize/Reset reward metric

    # run timesteps until state is terminal
    for timestep in count():
        #env.render() # render the game image
        action = agent.select_action(state, duel_policy_net) # select an action␣
↪with epsilon greedy
        reward = env.take_action(action).float() # get the selected action's␣
↪reward
        r_metric += reward.item() # sum the reward
        next_state = env.get_state() # get the new state
```

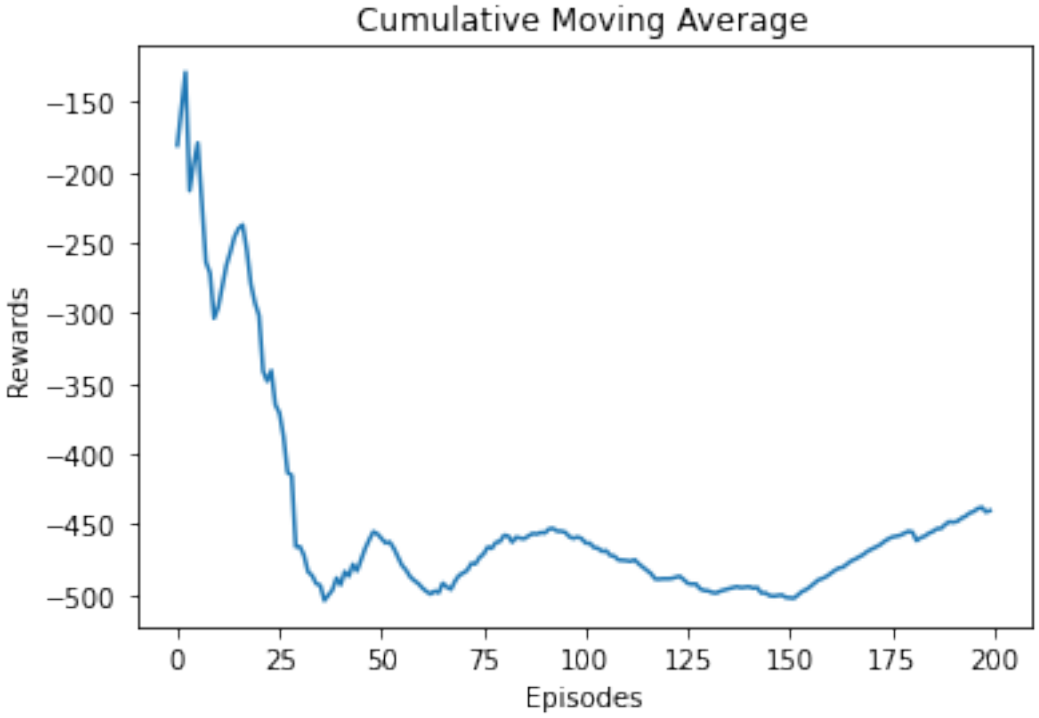
```
        memory.store(Experience(state, action, next_state, reward)) # store␣
↪agent's experience
        state = next_state # set new state as current state

        # check if there are enough experiences in the memory to sample
        if memory.batch_available(batch_size):
            experiences = memory.get(batch_size) # get a batch of experiences
            states, actions, rewards, next_states =␣
↪extract_tensors(experiences) # convert to tensors for the neural networks
            # Neural Network part
            current_q_values = QValues.get_current(duel_policy_net, states,␣
↪actions) # get Q-Values for current state from policy net
            next_q_values = QValues.get_next(duel_target_net, next_states) #␣
↪get Q-Values for next state from target net
            target_q_values = (next_q_values * gamma) + rewards # calculate the␣
↪target Q-Values to use for back propagation
            # Back propagation
            loss = F.mse_loss(current_q_values, target_q_values.unsqueeze(1)) #␣
↪calculate the loss
            duel_optimizer.zero_grad() # reset gradients (required by PyTorch)
            loss.backward() # back propagate loss
            duel_optimizer.step() # update weights

        # check if last action ended the episode
        if env.done:
            episode_durations.append(timestep) # append episode duration
            episode_rewards.append(r_metric) # append episode rewards
            break

    # update target network weights
    if episode % target_update == 0:
        duel_target_net.load_state_dict(duel_policy_net.state_dict()) # copy␣
↪policy network's weights to target network

    # check if agent solved the game
    #if reward_ma(episode_rewards,100,False) >= 200:
       # break

# close the render window
#env.close()
```

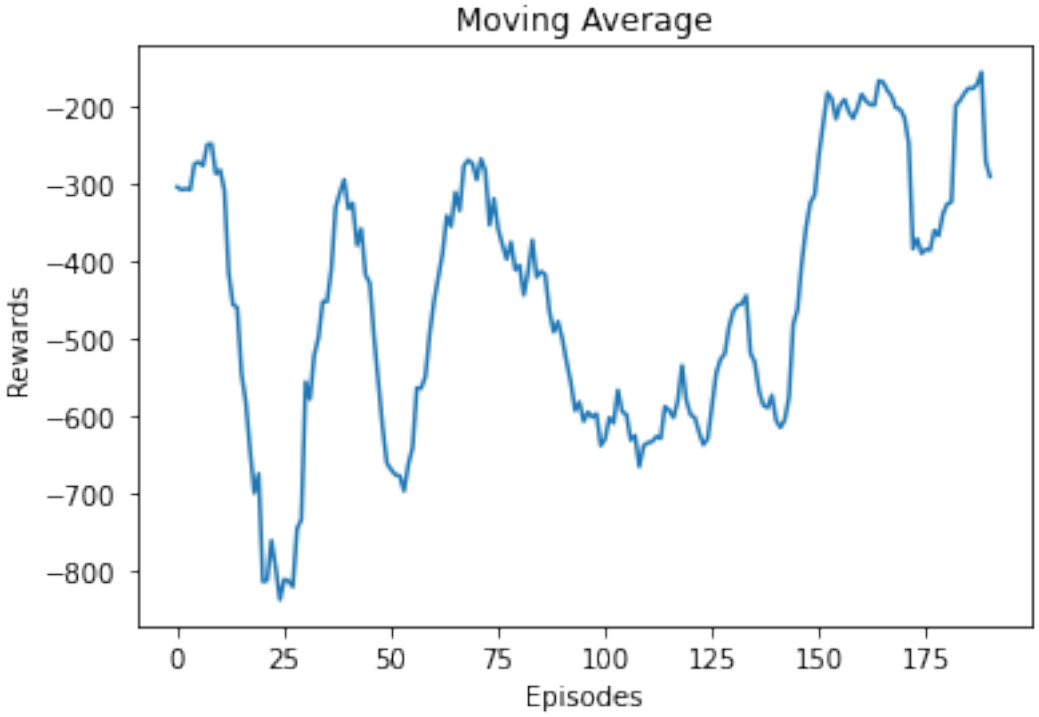
Plot Performance Metrics

[11]:
```
# cumulative moving average
plot_cma(episode_durations, "Duration")
plot_cma(episode_rewards, "Rewards")
# moving average
```

```
reward_ma(episode_rewards,10,True)
```







Testing the models' hyperparameters

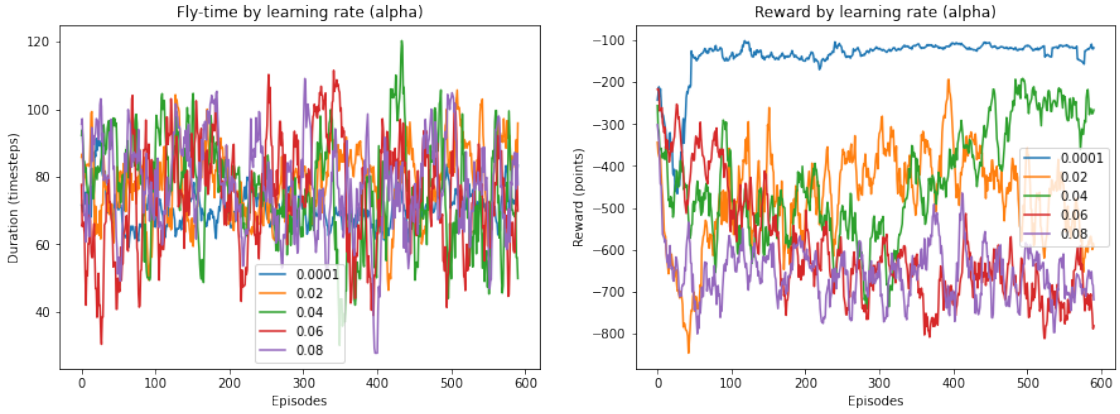
Vanilla DQN

[123]:
```
# Set device based on wether there is a GPU or not
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Parameters for Epsilon Greedy Strategy
max_epsilon = 1
min_epsilon = 0.01
epsilon_decay_rate = 0.001
strategy = EpsilonGreedyValue(max_epsilon, min_epsilon, epsilon_decay_rate)

# Parameters that control the Deep Q-Learning
gamma = 0.999
lr = 0.0001
num_episodes = 600

# Parameters for Replay Memory
memory_size = 100000
batch_size = 256
```

```
memory = ReplayMemory(memory_size)

# Parameters for the architecture of the NN
hidden_neurons = 32 # number of neurons on the hidden layers
n_actions = env.env.action_space.n # for nn output, the number of actions
target_update = 10 # target nn weights update every n episodes
# Parameters for performance metrics
# Create the Deep Q-Network
policy_net = DQN(env.n_state_features(),hidden_neurons,n_actions,False).
 ↪to(device) # DQN for current state's Q-Values

fig, ax = plt.subplots(1,2, figsize=(15,5))
# ax[0] = fig.add_subplot()
# ax[1] = fig.add_subplot()

# Begin training our agent on the selected gaming environment
for lr in np.arange(0.0001, 0.1, 0.02):
    env_name = 'LunarLander-v2'
    env = EnvManager(env_name, device) # load the environment manager helper
    agent = DRLAgent(strategy, env.num_actions_available(), device) # Create␣
 ↪the Deep Reinforcement Learning agent
    optimizer = optim.Adam(params=policy_net.parameters(), lr=lr) # Adam␣
 ↪optimizer

    training_duration = [] # initialize/Reset duration of each episode
    training_rewards = [] # initialize/Reset rewards of each episode

    for episode in range(num_episodes):
        r_metric = 0 # initialize/Reset reward metric
        duration = 0 # initialize/Reset duration metric
        env.reset() # reset the environment on each episode
        state = env.get_state() # get current (initial) state

        # run timesteps until state is terminal
        for timestep in count():
            #env.render() # render the game image
            action = agent.select_action(state, policy_net) # select an action␣
 ↪with epsilon greedy
            reward = env.take_action(action).float() # get the selected␣
 ↪action's reward
            r_metric += reward.item() # sum the reward
            next_state = env.get_state() # get the new state
            memory.store(Experience(state, action, next_state, reward)) # store␣
 ↪agent's experience
            state = next_state # set new state as current state
```

```
            # check if there are enough experiences in the memory to sample
            if memory.batch_available(batch_size):
                experiences = memory.get(batch_size) # get a batch of␣
↪experiences
                states, actions, rewards, next_states =␣
↪extract_tensors(experiences) # convert to tensors for the neural networks
                # Neural Network part
                current_q_values = QValues.get_current(policy_net, states,␣
↪actions) # get Q-Values for current state from policy net
                next_q_values = QValues.get_next(policy_net, next_states) # get␣
↪Q-Values for next state from policy net
                target_q_values = (next_q_values * gamma) + rewards # calculate␣
↪the target Q-Values to use for back propagation
                # Back propagation
                loss = F.mse_loss(current_q_values, target_q_values.
↪unsqueeze(1)) # calculate the loss
                optimizer.zero_grad() # reset gradients (required by PyTorch)
                loss.backward() # back propagate loss
                optimizer.step() # update weights

            # check if last action ended the episode
            if env.done:
                duration += timestep
                # training_duration.append(timestep) # append episode duration
                # training_rewards.append(r_metric) # append episode rewards
                break
            if timestep == 150:
                break

        training_duration.append(duration) # append training duration
        training_rewards.append(r_metric)  # append training rewards

    # 10 episodes duration moving average
    dur = np.cumsum(training_duration, dtype=float)
    dur[10:] = dur[10:] - dur[:-10]
    training_duration_average =  dur[10 - 1:] / 10
    # 10 episodes reward moving average
    rew = np.cumsum(training_rewards, dtype=float)
    rew[10:] = rew[10:] - rew[:-10]
    training_rewards_average =  rew[10 - 1:] / 10

    ax[0].set_xlabel('Episodes')
    ax[0].set_ylabel('Duration (timesteps)')
    ax[0].set_title('Fly-time by learning rate (alpha)')
    ax[0].plot(np.arange(len(training_duration_average)),␣
↪training_duration_average, label = '{:.2}'.format(lr))
```

```
    ax[0].legend()
    ax[1].set_xlabel('Episodes')
    ax[1].set_ylabel('Reward (points)')
    ax[1].set_title('Reward by learning rate (alpha)')
    ax[1].plot(np.arange(len(training_rewards_average)),␣
↪training_rewards_average, label = '{:.2}'.format(lr))
    ax[1].legend()
```



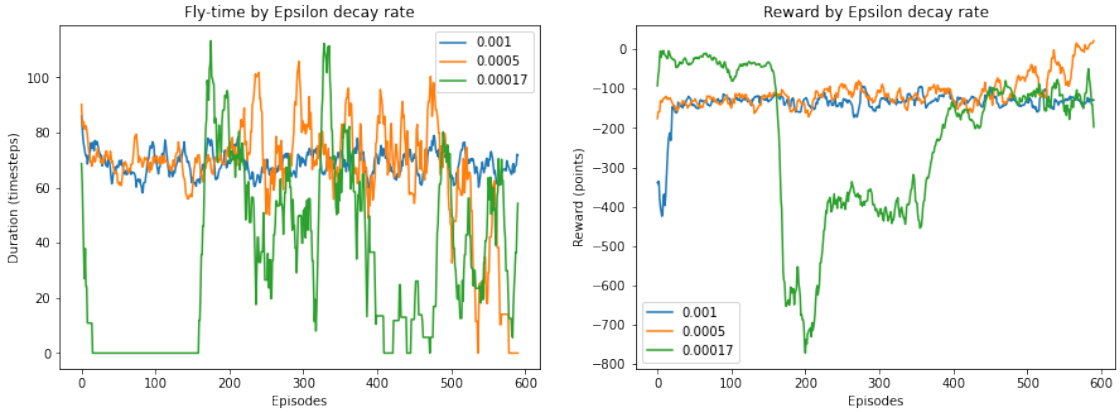
Testing the epsilon decay rate

[139]:
```
# Parameters for Epsilon Greedy Strategy
max_epsilon = 1
min_epsilon = 0.01
epsilon_decay_rate = 0.001
strategy = EpsilonGreedyValue(max_epsilon, min_epsilon, epsilon_decay_rate)

# Parameters that control the Deep Q-Learning
gamma = 0.999
lr = 0.0001
num_episodes = 600

# Parameters for Replay Memory
memory_size = 100000
batch_size = 256
memory = ReplayMemory(memory_size)

# Parameters for the architecture of the NN
hidden_neurons = 32 # number of neurons on the hidden layers
n_actions = env.env.action_space.n # for nn output, the number of actions
target_update = 10 # target nn weights update every n episodes
```

```
# Parameters for performance metrics
# Create the Deep Q-Network
policy_net = DQN(env.n_state_features(),hidden_neurons,n_actions,False).
 ↪to(device) # DQN for current state's Q-Values

fig, ax = plt.subplots(1,2, figsize=(15,5))
# ax[0] = fig.add_subplot()
# ax[1] = fig.add_subplot()

# Begin training our agent on the selected gaming environment
for epsilon_decay_rate in [0.001, 0.0005, 0.00017]:
    env_name = 'LunarLander-v2'
    env = EnvManager(env_name, device) # load the environment manager helper
    agent = DRLAgent(strategy, env.num_actions_available(), device) # Create␣
 ↪the Deep Reinforcement Learning agent
    optimizer = optim.Adam(params=policy_net.parameters(), lr=lr) # Adam␣
 ↪optimizer

    training_duration = [] # initialize/Reset duration of each episode
    training_rewards = [] # initialize/Reset rewards of each episode

    for episode in range(num_episodes):
        r_metric = 0 # initialize/Reset reward metric
        duration = 0 # initialize/Reset duration metric
        env.reset() # reset the environment on each episode
        state = env.get_state() # get current (initial) state

        # run timesteps until state is terminal
        for timestep in count():
            #env.render() # render the game image
            action = agent.select_action(state, policy_net) # select an action␣
 ↪with epsilon greedy
            reward = env.take_action(action).float() # get the selected␣
 ↪action's reward
            r_metric += reward.item() # sum the reward
            next_state = env.get_state() # get the new state
            memory.store(Experience(state, action, next_state, reward)) # store␣
 ↪agent's experience
            state = next_state # set new state as current state

            # check if there are enough experiences in the memory to sample
            if memory.batch_available(batch_size):
                experiences = memory.get(batch_size) # get a batch of␣
 ↪experiences
                states, actions, rewards, next_states =␣
 ↪extract_tensors(experiences) # convert to tensors for the neural networks
```

```
                # Neural Network part
                current_q_values = QValues.get_current(policy_net, states,␣
↪actions) # get Q-Values for current state from policy net
                next_q_values = QValues.get_next(policy_net, next_states) # get␣
↪Q-Values for next state from policy net
                target_q_values = (next_q_values * gamma) + rewards # calculate␣
↪the target Q-Values to use for back propagation
                # Back propagation
                loss = F.mse_loss(current_q_values, target_q_values.
↪unsqueeze(1)) # calculate the loss
                optimizer.zero_grad() # reset gradients (required by PyTorch)
                loss.backward() # back propagate loss
                optimizer.step() # update weights

            # check if last action ended the episode
            if env.done:
                duration += timestep
                # training_duration.append(timestep) # append episode duration
                # training_rewards.append(r_metric) # append episode rewards
                break
            if timestep == 150:
                break

        training_duration.append(duration) # append training duration
        training_rewards.append(r_metric)  # append training rewards

    # 10 episodes duration moving average
    dur = np.cumsum(training_duration, dtype=float)
    dur[10:] = dur[10:] - dur[:-10]
    training_duration_average =  dur[10 - 1:] / 10
    # 10 episodes reward moving average
    rew = np.cumsum(training_rewards, dtype=float)
    rew[10:] = rew[10:] - rew[:-10]
    training_rewards_average =  rew[10 - 1:] / 10

    ax[0].set_xlabel('Episodes')
    ax[0].set_ylabel('Duration (timesteps)')
    ax[0].set_title('Fly-time by Epsilon decay rate')
    ax[0].plot(np.arange(len(training_duration_average)),␣
↪training_duration_average, label = '{:.2}'.format(epsilon_decay_rate))
    ax[0].legend()
    ax[1].set_xlabel('Episodes')
    ax[1].set_ylabel('Reward (points)')
    ax[1].set_title('Reward by Epsilon decay rate')
    ax[1].plot(np.arange(len(training_rewards_average)),␣
↪training_rewards_average, label = '{:.2}'.format(epsilon_decay_rate))
```

```
    ax[1].legend()
```

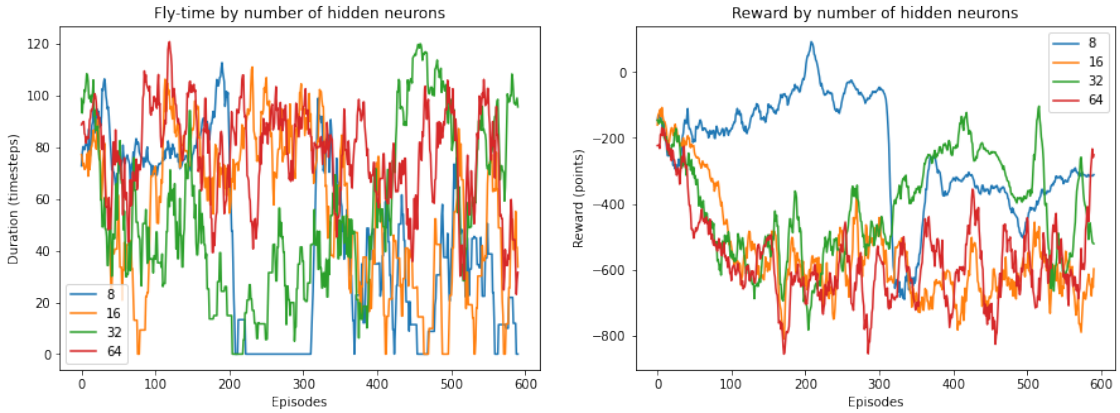


Testing the number of hidden neurons

[132]:
```
# Parameters for Epsilon Greedy Strategy
max_epsilon = 1
min_epsilon = 0.01
epsilon_decay_rate = 0.00017
strategy = EpsilonGreedyValue(max_epsilon, min_epsilon, epsilon_decay_rate)

# Parameters that control the Deep Q-Learning
gamma = 0.999
lr = 0.0001
num_episodes = 600

# Parameters for Replay Memory
memory_size = 100000
batch_size = 256
memory = ReplayMemory(memory_size)

# Parameters for the architecture of the NN
hidden_neurons = 32 # number of neurons on the hidden layers
n_actions = env.env.action_space.n # for nn output, the number of actions
target_update = 10 # target nn weights update every n episodes
# Parameters for performance metrics
# Create the Deep Q-Network
policy_net = DQN(env.n_state_features(),hidden_neurons,n_actions,False).
 ↪to(device) # DQN for current state's Q-Values

fig, ax = plt.subplots(1,2, figsize=(15,5))
# ax[0] = fig.add_subplot()
# ax[1] = fig.add_subplot()
```

```
# Begin training our agent on the selected gaming environment
for hidden_neurons in [8, 16, 32, 64]:
    env_name = 'LunarLander-v2'
    env = EnvManager(env_name, device) # load the environment manager helper
    agent = DRLAgent(strategy, env.num_actions_available(), device) # Create␣
↪the Deep Reinforcement Learning agent
    optimizer = optim.Adam(params=policy_net.parameters(), lr=lr) # Adam␣
↪optimizer

    training_duration = [] # initialize/Reset duration of each episode
    training_rewards = [] # initialize/Reset rewards of each episode

    for episode in range(num_episodes):
        r_metric = 0 # initialize/Reset reward metric
        duration = 0 # initialize/Reset duration metric
        env.reset() # reset the environment on each episode
        state = env.get_state() # get current (initial) state

        # run timesteps until state is terminal
        for timestep in count():
            #env.render() # render the game image
            action = agent.select_action(state, policy_net) # select an action␣
↪with epsilon greedy
            reward = env.take_action(action).float() # get the selected␣
↪action's reward
            r_metric += reward.item() # sum the reward
            next_state = env.get_state() # get the new state
            memory.store(Experience(state, action, next_state, reward)) # store␣
↪agent's experience
            state = next_state # set new state as current state

            # check if there are enough experiences in the memory to sample
            if memory.batch_available(batch_size):
                experiences = memory.get(batch_size) # get a batch of␣
↪experiences
                states, actions, rewards, next_states =␣
↪extract_tensors(experiences) # convert to tensors for the neural networks
                # Neural Network part
                current_q_values = QValues.get_current(policy_net, states,␣
↪actions) # get Q-Values for current state from policy net
                next_q_values = QValues.get_next(policy_net, next_states) # get␣
↪Q-Values for next state from policy net
                target_q_values = (next_q_values * gamma) + rewards # calculate␣
↪the target Q-Values to use for back propagation
                # Back propagation
```

```
                loss = F.mse_loss(current_q_values, target_q_values.
↪unsqueeze(1)) # calculate the loss
                optimizer.zero_grad() # reset gradients (required by PyTorch)
                loss.backward() # back propagate loss
                optimizer.step() # update weights

            # check if last action ended the episode
            if env.done:
                duration += timestep
                # training_duration.append(timestep) # append episode duration
                # training_rewards.append(r_metric) # append episode rewards
                break
            if timestep == 150:
                break

        training_duration.append(duration) # append training duration
        training_rewards.append(r_metric)  # append training rewards

    # 10 episodes duration moving average
    dur = np.cumsum(training_duration, dtype=float)
    dur[10:] = dur[10:] - dur[:-10]
    training_duration_average =  dur[10 - 1:] / 10
    # 10 episodes reward moving average
    rew = np.cumsum(training_rewards, dtype=float)
    rew[10:] = rew[10:] - rew[:-10]
    training_rewards_average =  rew[10 - 1:] / 10

    ax[0].set_xlabel('Episodes')
    ax[0].set_ylabel('Duration (timesteps)')
    ax[0].set_title('Fly-time by number of hidden neurons')
    ax[0].plot(np.arange(len(training_duration_average)),␣
↪training_duration_average, label = '{}'.format(hidden_neurons))
    ax[0].legend()
    ax[1].set_xlabel('Episodes')
    ax[1].set_ylabel('Reward (points)')
    ax[1].set_title('Reward by number of hidden neurons')
    ax[1].plot(np.arange(len(training_rewards_average)),␣
↪training_rewards_average, label = '{}'.format(hidden_neurons))
    ax[1].legend()
```

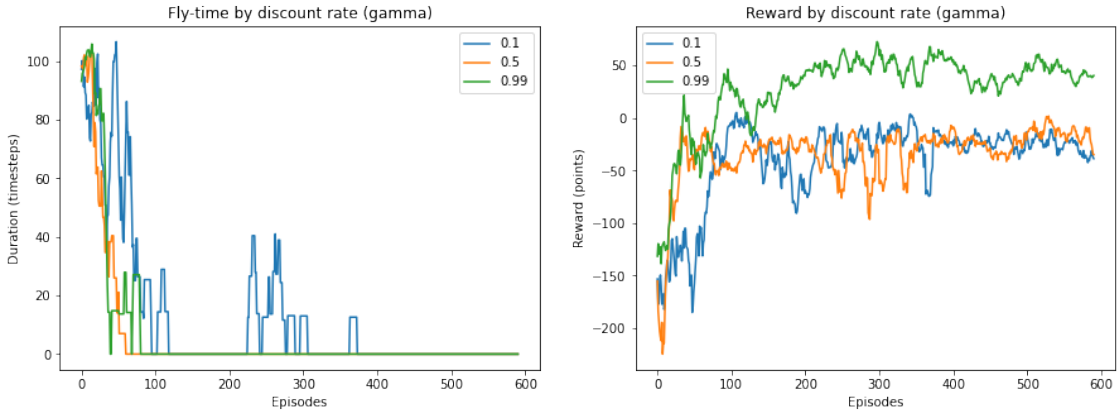


Testing the discount rate (gamma)

[133]:
```
# Parameters for Epsilon Greedy Strategy
max_epsilon = 1
min_epsilon = 0.01
epsilon_decay_rate = 0.00017
strategy = EpsilonGreedyValue(max_epsilon, min_epsilon, epsilon_decay_rate)

# Parameters that control the Deep Q-Learning
gamma = 0.999
lr = 0.0001
num_episodes = 600

# Parameters for Replay Memory
memory_size = 100000
batch_size = 256
memory = ReplayMemory(memory_size)

# Parameters for the architecture of the NN
hidden_neurons = 32 # number of neurons on the hidden layers
n_actions = env.env.action_space.n # for nn output, the number of actions
target_update = 10 # target nn weights update every n episodes
# Parameters for performance metrics
# Create the Deep Q-Network
policy_net = DQN(env.n_state_features(),hidden_neurons,n_actions,False).
 ↪to(device) # DQN for current state's Q-Values

fig, ax = plt.subplots(1,2, figsize=(15,5))
# ax[0] = fig.add_subplot()
# ax[1] = fig.add_subplot()

# Begin training our agent on the selected gaming environment
```

```
for gamma in [0.1, 0.5, 0.99]:
    env_name = 'LunarLander-v2'
    env = EnvManager(env_name, device) # load the environment manager helper
    agent = DRLAgent(strategy, env.num_actions_available(), device) # Create␣
↪the Deep Reinforcement Learning agent
    optimizer = optim.Adam(params=policy_net.parameters(), lr=lr) # Adam␣
↪optimizer

    training_duration = [] # initialize/Reset duration of each episode
    training_rewards = [] # initialize/Reset rewards of each episode

    for episode in range(num_episodes):
        r_metric = 0 # initialize/Reset reward metric
        duration = 0 # initialize/Reset duration metric
        env.reset() # reset the environment on each episode
        state = env.get_state() # get current (initial) state

        # run timesteps until state is terminal
        for timestep in count():
            #env.render() # render the game image
            action = agent.select_action(state, policy_net) # select an action␣
↪with epsilon greedy
            reward = env.take_action(action).float() # get the selected␣
↪action's reward
            r_metric += reward.item() # sum the reward
            next_state = env.get_state() # get the new state
            memory.store(Experience(state, action, next_state, reward)) # store␣
↪agent's experience
            state = next_state # set new state as current state

            # check if there are enough experiences in the memory to sample
            if memory.batch_available(batch_size):
                experiences = memory.get(batch_size) # get a batch of␣
↪experiences
                states, actions, rewards, next_states =␣
↪extract_tensors(experiences) # convert to tensors for the neural networks
                # Neural Network part
                current_q_values = QValues.get_current(policy_net, states,␣
↪actions) # get Q-Values for current state from policy net
                next_q_values = QValues.get_next(policy_net, next_states) # get␣
↪Q-Values for next state from policy net
                target_q_values = (next_q_values * gamma) + rewards # calculate␣
↪the target Q-Values to use for back propagation
                # Back propagation
                loss = F.mse_loss(current_q_values, target_q_values.
↪unsqueeze(1)) # calculate the loss
```

```
                optimizer.zero_grad() # reset gradients (required by PyTorch)
                loss.backward() # back propagate loss
                optimizer.step() # update weights

            # check if last action ended the episode
            if env.done:
                duration += timestep
                # training_duration.append(timestep) # append episode duration
                # training_rewards.append(r_metric) # append episode rewards
                break
            if timestep == 150:
                break

        training_duration.append(duration) # append training duration
        training_rewards.append(r_metric)  # append training rewards

    # 10 episodes duration moving average
    dur = np.cumsum(training_duration, dtype=float)
    dur[10:] = dur[10:] - dur[:-10]
    training_duration_average =  dur[10 - 1:] / 10
    # 10 episodes reward moving average
    rew = np.cumsum(training_rewards, dtype=float)
    rew[10:] = rew[10:] - rew[:-10]
    training_rewards_average =  rew[10 - 1:] / 10

    ax[0].set_xlabel('Episodes')
    ax[0].set_ylabel('Duration (timesteps)')
    ax[0].set_title('Fly-time by discount rate (gamma)')
    ax[0].plot(np.arange(len(training_duration_average)),␣
↪training_duration_average, label = '{}'.format(gamma))
    ax[0].legend()
    ax[1].set_xlabel('Episodes')
    ax[1].set_ylabel('Reward (points)')
    ax[1].set_title('Reward by discount rate (gamma)')
    ax[1].plot(np.arange(len(training_rewards_average)),␣
↪training_rewards_average, label = '{}'.format(gamma))
    ax[1].legend()
```



Final training and testing In the next sections the models are re-trained one at the time (with the chosen hyperparamters) and finally tested. For each model there is a first cell where all the parameters are re-initiated, and a second cell where the model is trained (and its outputs stored in lists).

[21]:
```
#INITIALISE VANILLA-DQN WITH ALL OPTIMIZED HYPERPARAMETERS

# Parameters for Epsilon Greedy Strategy
max_epsilon = 1
min_epsilon = 0.01
epsilon_decay_rate = 0.00017
strategy = EpsilonGreedyValue(max_epsilon, min_epsilon, epsilon_decay_rate)

# Parameters that control the Deep Q-Learning
gamma = 0.999
lr = 0.0001
num_episodes = 1000

# Parameters for Replay Memory
memory_size = 100000
batch_size = 256
memory = ReplayMemory(memory_size)

# Parameters for the architecture of the NN
hidden_neurons = 32 # number of neurons on the hidden layers
n_actions = env.env.action_space.n # for nn output, the number of actions
target_update = 10 # target nn weights update every n episodes
# Parameters for performance metrics
# Create the Deep Q-Network
policy_net = DQN(env.n_state_features(),hidden_neurons,n_actions,False).
 ↪to(device) # DQN for current state's Q-Values
```

```
# Begin training our agent on the selected gaming environment

# Create the Deep Reinforcement Learning agent
agent = DRLAgent(strategy, env.num_actions_available(), device)

# Create the Deep Q-Network
policy_net = DQN(env.n_state_features(),hidden_neurons,n_actions,False).
 ↪to(device) # DQN for current state's Q-Values

# Create the target network for Double Deep Q-Network
target_net = DQN(env.n_state_features(),hidden_neurons,n_actions,False).
 ↪to(device) # DQN for next state's Q-Values
target_net.load_state_dict(policy_net.state_dict()) # copy policy's net␣
 ↪parameters to target net
target_net.eval() # set target net to eval so it's weights do not update with␣
 ↪back propagation

# choose the optimizer
optimizer = optim.Adam(params=policy_net.parameters(), lr=lr) # Adam optimizer

# Create the Dueling Deep Q-Network
duel_policy_net = DQN(env.n_state_features(),hidden_neurons,n_actions,True).
 ↪to(device) # Dueling DQN for current state's Q-Values
# Create the target network for Dueling Double Deep Q-Network
duel_target_net = DQN(env.n_state_features(),hidden_neurons,n_actions,True).
 ↪to(device) # Dueling DQN for next state's Q-Values
duel_target_net.load_state_dict(duel_policy_net.state_dict()) # copy dueling␣
 ↪policy's net parameters to dueling target net
duel_target_net.eval() # set dueling target net to eval so it's weights do not␣
 ↪update with back propagation

# choose the optimizer
duel_optimizer = optim.Adam(params=duel_policy_net.parameters(), lr=lr) # Adam␣
 ↪optimizer
```

[22]:
```
# TRAIN VANILLA DQN

env_name = 'LunarLander-v2'
env = EnvManager(env_name, device) # load the environment manager helper
agent = DRLAgent(strategy, env.num_actions_available(), device) # Create the␣
 ↪Deep Reinforcement Learning agent
optimizer = optim.Adam(params=policy_net.parameters(), lr=lr) # Adam optimizer

vanilla_episode_durations = [] # variable to store the duration of each episode
vanilla_episode_rewards = [] # variable to store rewards of each episode

# Begin training our agent on the selected gaming environment
```

```
for episode in range(num_episodes):
    env.reset() # reset the environment on each episode
    state = env.get_state() # get current (initial) state

    r_metric = 0 # initialize/Reset reward metric

    # run timesteps until state is terminal
    for timestep in count():
        #env.render() # render the game image
        action = agent.select_action(state, policy_net) # select an action with␣
↪epsilon greedy
        reward = env.take_action(action).float() # get the selected action's␣
↪reward
        r_metric += reward.item() # sum the reward
        next_state = env.get_state() # get the new state
        memory.store(Experience(state, action, next_state, reward)) # store␣
↪agent's experience
        state = next_state # set new state as current state

        # check if there are enough experiences in the memory to sample
        if memory.batch_available(batch_size):
            experiences = memory.get(batch_size) # get a batch of experiences
            states, actions, rewards, next_states =␣
↪extract_tensors(experiences) # convert to tensors for the neural networks
            # Neural Network part
            current_q_values = QValues.get_current(policy_net, states, actions)␣
↪# get Q-Values for current state from policy net
            next_q_values = QValues.get_next(policy_net, next_states) # get␣
↪Q-Values for next state from policy net
            target_q_values = (next_q_values * gamma) + rewards # calculate the␣
↪target Q-Values to use for back propagation
            # Back propagation
            loss = F.mse_loss(current_q_values, target_q_values.unsqueeze(1)) #␣
↪calculate the loss
            optimizer.zero_grad() # reset gradients (required by PyTorch)
            loss.backward() # back propagate loss
            optimizer.step() # update weights

        # check if last action ended the episode
        if env.done:
            vanilla_episode_durations.append(timestep) # append episode␣
↪duration
            vanilla_episode_rewards.append(r_metric) # append episode rewards
            break

        if timestep == 500:
```

```
            vanilla_episode_durations.append(timestep) # append episode␣
 ↪duration
            vanilla_episode_rewards.append(r_metric) # append episode rewards
            break


# 10 episodes duration moving average
dur = np.cumsum(vanilla_episode_durations, dtype=float)
dur[10:] = dur[10:] - dur[:-10]
vanilla_training_duration_average =  dur[10 - 1:] / 10
# 10 episodes reward moving average
rew = np.cumsum(vanilla_episode_rewards, dtype=float)
rew[10:] = rew[10:] - rew[:-10]
vanilla_training_rewards_average =  rew[10 - 1:] / 10
```

[24]:
```
#INITIALISE DOUBLE DQN WITH ALL OPTIMIZED HYPERPARAMETERS

# Parameters for Epsilon Greedy Strategy
max_epsilon = 1
min_epsilon = 0.01
epsilon_decay_rate = 0.00017
strategy = EpsilonGreedyValue(max_epsilon, min_epsilon, epsilon_decay_rate)

# Parameters that control the Deep Q-Learning
gamma = 0.999
lr = 0.0001
num_episodes = 1000

# Parameters for Replay Memory
memory_size = 100000
batch_size = 256
memory = ReplayMemory(memory_size)

# Parameters for the architecture of the NN
hidden_neurons = 32 # number of neurons on the hidden layers
n_actions = env.env.action_space.n # for nn output, the number of actions
target_update = 10 # target nn weights update every n episodes
# Parameters for performance metrics
# Create the Deep Q-Network
policy_net = DQN(env.n_state_features(),hidden_neurons,n_actions,False).
 ↪to(device) # DQN for current state's Q-Values

# Begin training our agent on the selected gaming environment

# Create the Deep Reinforcement Learning agent
agent = DRLAgent(strategy, env.num_actions_available(), device)
```

```
# Create the Deep Q-Network
policy_net = DQN(env.n_state_features(),hidden_neurons,n_actions,False).
 ↪to(device) # DQN for current state's Q-Values

# Create the target network for Double Deep Q-Network
target_net = DQN(env.n_state_features(),hidden_neurons,n_actions,False).
 ↪to(device) # DQN for next state's Q-Values
target_net.load_state_dict(policy_net.state_dict()) # copy policy's net␣
 ↪parameters to target net
target_net.eval() # set target net to eval so it's weights do not update with␣
 ↪back propagation

# choose the optimizer
optimizer = optim.Adam(params=policy_net.parameters(), lr=lr) # Adam optimizer

# Create the Dueling Deep Q-Network
duel_policy_net = DQN(env.n_state_features(),hidden_neurons,n_actions,True).
 ↪to(device) # Dueling DQN for current state's Q-Values
# Create the target network for Dueling Double Deep Q-Network
duel_target_net = DQN(env.n_state_features(),hidden_neurons,n_actions,True).
 ↪to(device) # Dueling DQN for next state's Q-Values
duel_target_net.load_state_dict(duel_policy_net.state_dict()) # copy dueling␣
 ↪policy's net parameters to dueling target net
duel_target_net.eval() # set dueling target net to eval so it's weights do not␣
 ↪update with back propagation

# choose the optimizer
duel_optimizer = optim.Adam(params=duel_policy_net.parameters(), lr=lr) # Adam␣
 ↪optimizer
```

[25]:
```
# TRAIN DOUBLE DQN

env_name = 'LunarLander-v2'
env = EnvManager(env_name, device) # load the environment manager helper
agent = DRLAgent(strategy, env.num_actions_available(), device) # Create the␣
 ↪Deep Reinforcement Learning agent
optimizer = optim.Adam(params=policy_net.parameters(), lr=lr) # Adam optimizer

double_episode_durations = [] # variable to store the duration of each episode
double_episode_rewards = [] # variable to store rewards of each episode

# Begin training our agent on the selected gaming environment
for episode in range(num_episodes):
    env.reset() # reset the environment on each episode
    state = env.get_state() # get current (initial) state

    r_metric = 0 # initialize/Reset reward metric
```

```
    # run timesteps until state is terminal
    for timestep in count():
        #env.render() # render the game image
        action = agent.select_action(state, policy_net) # select an action with␣
↪epsilon greedy
        reward = env.take_action(action).float() # get the selected action's␣
↪reward
        r_metric += reward.item() # sum the reward
        next_state = env.get_state() # get the new state
        memory.store(Experience(state, action, next_state, reward)) # store␣
↪agent's experience
        state = next_state # set new state as current state

        # check if there are enough experiences in the memory to sample
        if memory.batch_available(batch_size):
            experiences = memory.get(batch_size) # get a batch of experiences
            states, actions, rewards, next_states =␣
↪extract_tensors(experiences) # convert to tensors for the neural networks
            # Neural Network part
            current_q_values = QValues.get_current(policy_net, states, actions)␣
↪# get Q-Values for current state from policy net
            next_q_values = QValues.get_next(target_net, next_states) # get␣
↪Q-Values for next state from target net
            target_q_values = (next_q_values * gamma) + rewards # calculate the␣
↪target Q-Values to use for back propagation
            # Back propagation
            loss = F.mse_loss(current_q_values, target_q_values.unsqueeze(1)) #␣
↪calculate the loss
            optimizer.zero_grad() # reset gradients (required by PyTorch)
            loss.backward() # back propagate loss
            optimizer.step() # update weights

        # check if last action ended the episode
        if env.done:
            double_episode_durations.append(timestep) # append episode duration
            double_episode_rewards.append(r_metric) # append episode rewards
            break

        if timestep == 500:
            double_episode_durations.append(timestep) # append episode duration
            double_episode_rewards.append(r_metric) # append episode rewards
            break

    # update target network weights
    if episode % target_update == 0:
```

```
        target_net.load_state_dict(policy_net.state_dict()) # copy policy␣
 ↪network's weights to target network


# 10 episodes duration moving average
dur = np.cumsum(double_episode_durations, dtype=float)
dur[10:] = dur[10:] - dur[:-10]
double_training_duration_average =  dur[10 - 1:] / 10
# 10 episodes reward moving average
rew = np.cumsum(double_episode_rewards, dtype=float)
rew[10:] = rew[10:] - rew[:-10]
double_training_rewards_average =  rew[10 - 1:] / 10
```

[18]:
```
#INITIALISE DUELING-DQN WITH ALL OPTIMIZED HYPERPARAMETERS

# Parameters for Epsilon Greedy Strategy
max_epsilon = 1
min_epsilon = 0.01
epsilon_decay_rate = 0.00017
strategy = EpsilonGreedyValue(max_epsilon, min_epsilon, epsilon_decay_rate)

# Parameters that control the Deep Q-Learning
gamma = 0.999
lr = 0.0001
num_episodes = 1000

# Parameters for Replay Memory
memory_size = 100000
batch_size = 256
memory = ReplayMemory(memory_size)

# Parameters for the architecture of the NN
hidden_neurons = 32 # number of neurons on the hidden layers
n_actions = env.env.action_space.n # for nn output, the number of actions
target_update = 10 # target nn weights update every n episodes
# Parameters for performance metrics
# Create the Deep Q-Network
policy_net = DQN(env.n_state_features(),hidden_neurons,n_actions,False).
 ↪to(device) # DQN for current state's Q-Values

# Begin training our agent on the selected gaming environment

# Create the Deep Reinforcement Learning agent
agent = DRLAgent(strategy, env.num_actions_available(), device)

# Create the Deep Q-Network
```

```
policy_net = DQN(env.n_state_features(),hidden_neurons,n_actions,False).
 ↪to(device) # DQN for current state's Q-Values

# Create the target network for Double Deep Q-Network
target_net = DQN(env.n_state_features(),hidden_neurons,n_actions,False).
 ↪to(device) # DQN for next state's Q-Values
target_net.load_state_dict(policy_net.state_dict()) # copy policy's net␣
 ↪parameters to target net
target_net.eval() # set target net to eval so it's weights do not update with␣
 ↪back propagation

# choose the optimizer
optimizer = optim.Adam(params=policy_net.parameters(), lr=lr) # Adam optimizer

# Create the Dueling Deep Q-Network
duel_policy_net = DQN(env.n_state_features(),hidden_neurons,n_actions,True).
 ↪to(device) # Dueling DQN for current state's Q-Values
# Create the target network for Dueling Double Deep Q-Network
duel_target_net = DQN(env.n_state_features(),hidden_neurons,n_actions,True).
 ↪to(device) # Dueling DQN for next state's Q-Values
duel_target_net.load_state_dict(duel_policy_net.state_dict()) # copy dueling␣
 ↪policy's net parameters to dueling target net
duel_target_net.eval() # set dueling target net to eval so it's weights do not␣
 ↪update with back propagation

# choose the optimizer
duel_optimizer = optim.Adam(params=duel_policy_net.parameters(), lr=lr) # Adam␣
 ↪optimizer
```

[19]:
```
# TRAINING THE DUELING DQN

env_name = 'LunarLander-v2'
env = EnvManager(env_name, device) # load the environment manager helper
agent = DRLAgent(strategy, env.num_actions_available(), device) # Create the␣
 ↪Deep Reinforcement Learning agent
optimizer = optim.Adam(params=policy_net.parameters(), lr=lr) # Adam optimizer

dueling_episode_durations = [] # variable to store the duration of each episode
dueling_episode_rewards = [] # variable to store rewards of each episode

# Begin training our agent on the selected gaming environment
for episode in range(num_episodes):
    env.reset() # reset the environment on each episode
    state = env.get_state() # get current (initial) state

    r_metric = 0 # initialize/Reset reward metric
```

```
    # run timesteps until state is terminal
    for timestep in count():
        #env.render() # render the game image
        action = agent.select_action(state, duel_policy_net) # select an action␣
↪with epsilon greedy
        reward = env.take_action(action).float() # get the selected action's␣
↪reward
        r_metric += reward.item() # sum the reward
        next_state = env.get_state() # get the new state
        memory.store(Experience(state, action, next_state, reward)) # store␣
↪agent's experience
        state = next_state # set new state as current state

        # check if there are enough experiences in the memory to sample
        if memory.batch_available(batch_size):
            experiences = memory.get(batch_size) # get a batch of experiences
            states, actions, rewards, next_states =␣
↪extract_tensors(experiences) # convert to tensors for the neural networks
            # Neural Network part
            current_q_values = QValues.get_current(duel_policy_net, states,␣
↪actions) # get Q-Values for current state from policy net
            next_q_values = QValues.get_next(duel_policy_net, next_states) #␣
↪get Q-Values for next state from policy net
            target_q_values = (next_q_values * gamma) + rewards # calculate the␣
↪target Q-Values to use for back propagation
            # Back propagation
            loss = F.mse_loss(current_q_values, target_q_values.unsqueeze(1)) #␣
↪calculate the loss
            optimizer.zero_grad() # reset gradients (required by PyTorch)
            loss.backward() # back propagate loss
            optimizer.step() # update weights

        # check if last action ended the episode
        if env.done:
            dueling_episode_durations.append(timestep) # append episode␣
↪duration
            dueling_episode_rewards.append(r_metric) # append episode rewards
            break

        if timestep == 500:
            dueling_episode_durations.append(timestep) # append episode␣
↪duration
            dueling_episode_rewards.append(r_metric) # append episode rewards
            break

# 10 episodes duration moving average
```

```
dur = np.cumsum(dueling_episode_durations, dtype=float)
dur[10:] = dur[10:] - dur[:-10]
dueling_training_duration_average =  dur[10 - 1:] / 10
# 10 episodes reward moving average
rew = np.cumsum(dueling_episode_rewards, dtype=float)
rew[10:] = rew[10:] - rew[:-10]
dueling_training_rewards_average =  rew[10 - 1:] / 10
```

[9]:
```
#INITIALISE DUELING DOUBLE WITH ALL OPTIMIZED HYPERPARAMETERS

# Parameters for Epsilon Greedy Strategy
max_epsilon = 1
min_epsilon = 0.01
epsilon_decay_rate = 0.00017
strategy = EpsilonGreedyValue(max_epsilon, min_epsilon, epsilon_decay_rate)

# Parameters that control the Deep Q-Learning
gamma = 0.999
lr = 0.0001
num_episodes = 1000

# Parameters for Replay Memory
memory_size = 100000
batch_size = 256
memory = ReplayMemory(memory_size)

# Parameters for the architecture of the NN
hidden_neurons = 32 # number of neurons on the hidden layers
n_actions = env.env.action_space.n # for nn output, the number of actions
target_update = 10 # target nn weights update every n episodes
# Parameters for performance metrics
# Create the Deep Q-Network
policy_net = DQN(env.n_state_features(),hidden_neurons,n_actions,False).
 ↪to(device) # DQN for current state's Q-Values

# Begin training our agent on the selected gaming environment

# Create the Deep Reinforcement Learning agent
agent = DRLAgent(strategy, env.num_actions_available(), device)

# Create the Deep Q-Network
policy_net = DQN(env.n_state_features(),hidden_neurons,n_actions,False).
 ↪to(device) # DQN for current state's Q-Values

# Create the target network for Double Deep Q-Network
target_net = DQN(env.n_state_features(),hidden_neurons,n_actions,False).
 ↪to(device) # DQN for next state's Q-Values
```

```
target_net.load_state_dict(policy_net.state_dict()) # copy policy's net␣
 ↪parameters to target net
target_net.eval() # set target net to eval so it's weights do not update with␣
 ↪back propagation

# choose the optimizer
optimizer = optim.Adam(params=policy_net.parameters(), lr=lr) # Adam optimizer

# Create the Dueling Deep Q-Network
duel_policy_net = DQN(env.n_state_features(),hidden_neurons,n_actions,True).
 ↪to(device) # Dueling DQN for current state's Q-Values
# Create the target network for Dueling Double Deep Q-Network
duel_target_net = DQN(env.n_state_features(),hidden_neurons,n_actions,True).
 ↪to(device) # Dueling DQN for next state's Q-Values
duel_target_net.load_state_dict(duel_policy_net.state_dict()) # copy dueling␣
 ↪policy's net parameters to dueling target net
duel_target_net.eval() # set dueling target net to eval so it's weights do not␣
 ↪update with back propagation

# choose the optimizer
duel_optimizer = optim.Adam(params=duel_policy_net.parameters(), lr=lr) # Adam␣
 ↪optimizer
```

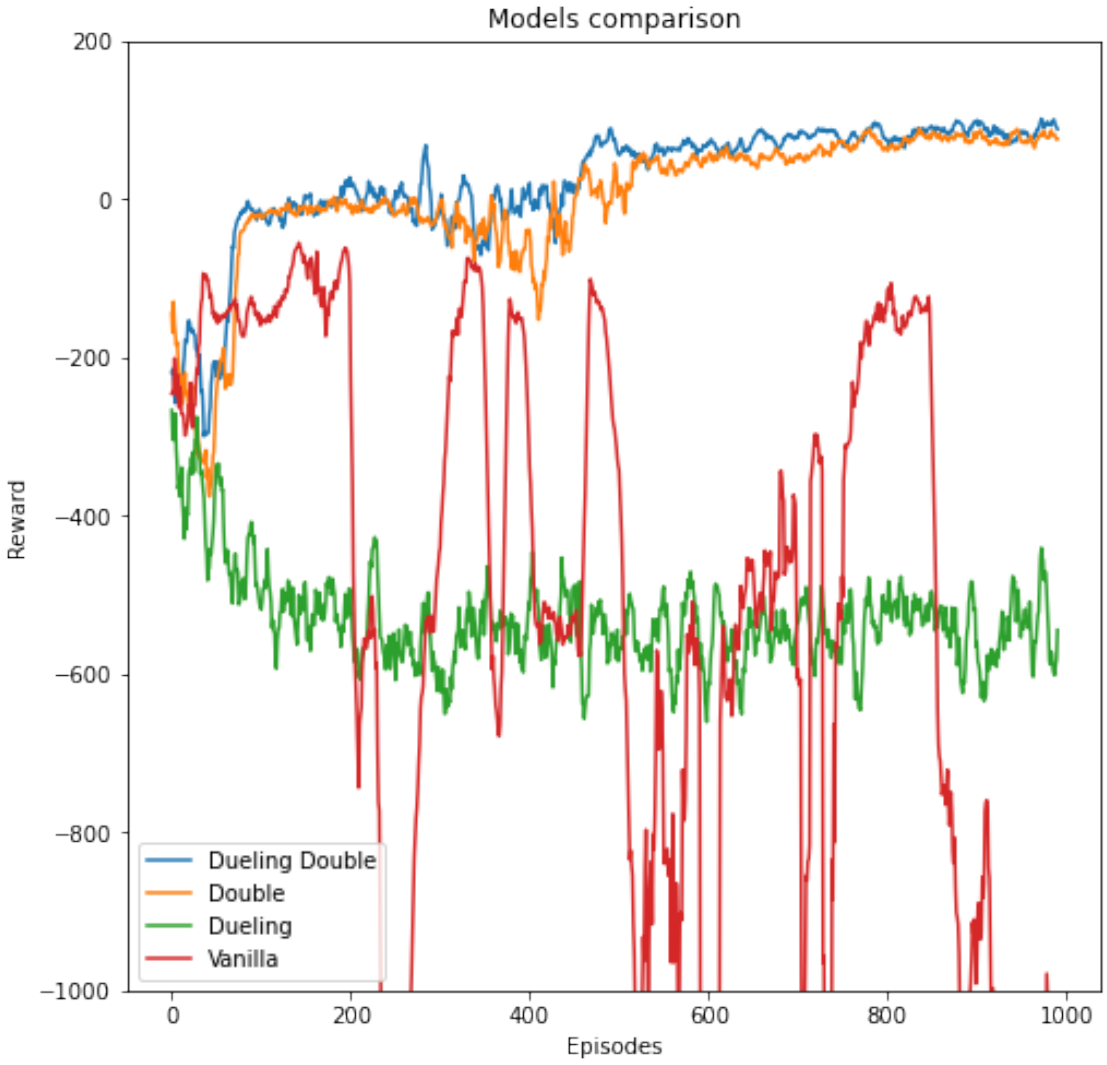
[10]:
```
#TRAINING THE DUELING DOUBLE

dueling_double_episode_durations = [] # variable to store the duration of each␣
 ↪episode
dueling_double_episode_rewards = [] # variable to store rewards of each episode

# Begin training our agent on the selected gaming environment
for episode in range(num_episodes):
    env.reset() # reset the environment on each episode
    state = env.get_state() # get current (initial) state

    r_metric = 0 # initialize/Reset reward metric

    # run timesteps until state is terminal
    for timestep in count():
        #env.render() # render the game image
        action = agent.select_action(state, duel_policy_net) # select an action␣
 ↪with epsilon greedy
        reward = env.take_action(action).float() # get the selected action's␣
 ↪reward
        r_metric += reward.item() # sum the reward
        next_state = env.get_state() # get the new state
        memory.store(Experience(state, action, next_state, reward)) # store␣
 ↪agent's experience
```

```
        state = next_state # set new state as current state

        # check if there are enough experiences in the memory to sample
        if memory.batch_available(batch_size):
            experiences = memory.get(batch_size) # get a batch of experiences
            states, actions, rewards, next_states =␣
↪extract_tensors(experiences) # convert to tensors for the neural networks
            # Neural Network part
            current_q_values = QValues.get_current(duel_policy_net, states,␣
↪actions) # get Q-Values for current state from policy net
            next_q_values = QValues.get_next(duel_target_net, next_states) #␣
↪get Q-Values for next state from target net
            target_q_values = (next_q_values * gamma) + rewards # calculate the␣
↪target Q-Values to use for back propagation
            # Back propagation
            loss = F.mse_loss(current_q_values, target_q_values.unsqueeze(1)) #␣
↪calculate the loss
            duel_optimizer.zero_grad() # reset gradients (required by PyTorch)
            loss.backward() # back propagate loss
            duel_optimizer.step() # update weights

        # check if last action ended the episode
        if env.done:
            dueling_double_episode_durations.append(timestep) # append episode␣
↪duration
            dueling_double_episode_rewards.append(r_metric) # append episode␣
↪rewards
            break

        if timestep == 500:
            dueling_double_episode_durations.append(timestep) # append episode␣
↪duration
            dueling_double_episode_rewards.append(r_metric) # append episode␣
↪rewards
            break

    # update target network weights
    if episode % target_update == 0:
        duel_target_net.load_state_dict(duel_policy_net.state_dict()) # copy␣
↪policy network's weights to target network

# 10 episodes duration moving average
dur = np.cumsum(dueling_double_episode_durations, dtype=float)
dur[10:] = dur[10:] - dur[:-10]
dueling_double_training_duration_average =  dur[10 - 1:] / 10
# 10 episodes reward moving average
```

```
rew = np.cumsum(dueling_double_episode_rewards, dtype=float)
rew[10:] = rew[10:] - rew[:-10]
dueling_double_training_rewards_average =  rew[10 - 1:] / 10
```

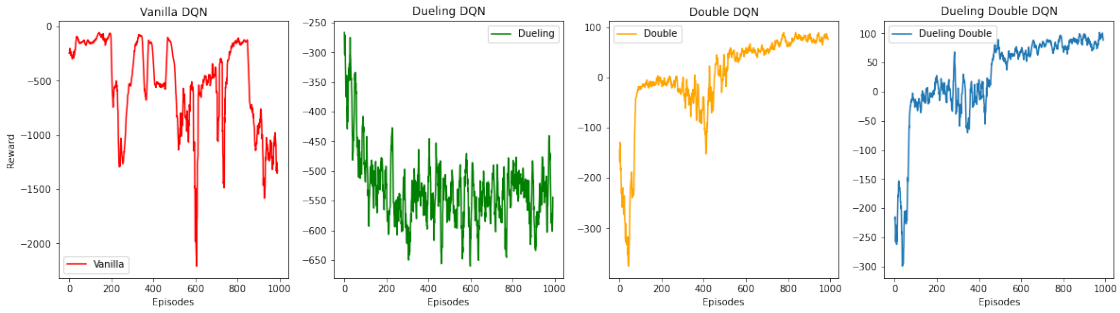
Plotting the results

[52]:
```
# plot the 10 episode rolling average reward of each model

fig = plt.subplots(figsize = (8,8))
plt.title('Models comparison')
plt.xlabel('Episodes')
plt.ylim(-1000,200)
plt.ylabel('Reward')
plt.plot(np.arange(len(dueling_double_training_rewards_average)),␣
 ↪dueling_double_training_rewards_average, label = 'Dueling Double')
plt.plot(np.arange(len(double_training_rewards_average)),␣
 ↪double_training_rewards_average, label = 'Double')
plt.plot(np.arange(len(dueling_training_rewards_average)),␣
 ↪dueling_training_rewards_average, label = 'Dueling')
plt.plot(np.arange(len(vanilla_training_rewards_average)),␣
 ↪vanilla_training_rewards_average, label = 'Vanilla')
plt.legend();
```



[37]:
```
# plot the models' results, separately

fig, ax = plt.subplots(1,4, figsize = (20,5))
ax[0].plot(np.arange(len(vanilla_training_rewards_average)),␣
 ↪vanilla_training_rewards_average, label = 'Vanilla', c = 'red')
ax[0].legend()
ax[0].set_title('Vanilla DQN')
ax[0].set_xlabel('Episodes')
ax[0].set_ylabel('Reward')
ax[1].plot(np.arange(len(dueling_training_rewards_average)),␣
 ↪dueling_training_rewards_average, label = 'Dueling', c = 'green')
ax[1].legend()
ax[1].set_title('Dueling DQN')
ax[1].set_xlabel('Episodes')
```

```
ax[2].plot(np.arange(len(double_training_rewards_average)),␣
 ↪double_training_rewards_average, label = 'Double', c = 'orange')
ax[2].legend()
ax[2].set_title('Double DQN')
ax[2].set_xlabel('Episodes')
ax[3].plot(np.arange(len(dueling_double_training_rewards_average)),␣
 ↪dueling_double_training_rewards_average, label = 'Dueling Double')
ax[3].legend()
ax[3].set_title('Dueling Double DQN')
ax[3].set_xlabel('Episodes');
```



Metrics

[66]:
```
# rewards and timesteps averages for each model

print('Vanilla: mean episode duration = {} | min episode duration = {} | max␣
 ↪episode duration = {}'.format(np.mean(vanilla_episode_durations), np.
 ↪min(vanilla_episode_durations), np.max(vanilla_episode_durations)))
print('Vanilla: mean reward = {} | min reward = {} | max reward = {}'.format(np.
 ↪mean(vanilla_episode_rewards), np.min(vanilla_episode_rewards), np.
 ↪max(vanilla_episode_rewards)))
print('Dueling: mean episode duration = {} | min episode duration = {} | max␣
 ↪episode duration = {}'.format(np.mean(dueling_episode_durations), np.
 ↪min(dueling_episode_durations), np.max(dueling_episode_durations)))
print('Dueling: mean reward = {} | min reward = {} | max reward = {}'.format(np.
 ↪mean(dueling_episode_rewards), np.min(dueling_episode_rewards), np.
 ↪max(dueling_episode_rewards)))
print('Double: mean episode duration = {} | min episode duration = {} | max␣
 ↪episode duration = {}'.format(np.mean(double_episode_durations), np.
 ↪min(double_episode_durations), np.max(double_episode_durations)))
print('Double: mean reward = {} | min reward = {} | max reward = {}'.format(np.
 ↪mean(double_episode_rewards), np.min(double_episode_rewards), np.
 ↪max(double_episode_rewards)))
```

```
print('Dueling Double: mean episode duration = {} | min episode duration = {} |␣
 ↪max episode duration = {}'.format(np.mean(dueling_double_episode_durations),␣
 ↪np.min(dueling_double_episode_durations), np.
 ↪max(dueling_double_episode_durations)))
print('Dueling Double: mean reward = {} | min reward = {} | max reward = {}'.
 ↪format(np.mean(dueling_double_episode_rewards), np.
 ↪min(dueling_double_episode_rewards), np.max(dueling_double_episode_rewards)))
```

```
Vanilla: mean episode duration = 253.675 | min episode duration = 49 | max
episode duration = 500
Vanilla: mean reward = -527.4211718478566 | min reward = -6448.277877458371 |
max reward = 239.5725368415824
Dueling: mean episode duration = 67.214 | min episode duration = 46 | max
episode duration = 137
Dueling: mean reward = -528.8123826997648 | min reward = -1129.3000551462173 |
max reward = -61.28033232362941
Double: mean episode duration = 456.424 | min episode duration = 65 | max
episode duration = 500
Double: mean reward = 1.7109668719920774 | min reward = -597.8412189818919 | max
reward = 128.45568719971925
Dueling Double: mean episode duration = 464.202 | min episode duration = 57 |
max episode duration = 500
Dueling Double: mean reward = 22.974367201545558 | min reward =
-485.236411975231 | max reward = 227.0166863068007
```

[68]:
```
# rewards and timesteps 10 episodes rolling averages for each model
print('Vanilla: mean episode duration = {} | min episode duration = {} | max␣
 ↪episode duration = {}'.format(np.mean(vanilla_training_duration_average), np.
 ↪min(vanilla_training_duration_average), np.
 ↪max(vanilla_training_duration_average)))
print('Vanilla: mean reward = {} | min reward = {} | max reward = {}'.format(np.
 ↪mean(vanilla_training_rewards_average), np.
 ↪min(vanilla_training_rewards_average), np.
 ↪max(vanilla_training_rewards_average)))
print('Dueling: mean episode duration = {} | min episode duration = {} | max␣
 ↪episode duration = {}'.format(np.mean(dueling_training_duration_average), np.
 ↪min(dueling_training_duration_average), np.
 ↪max(dueling_training_duration_average)))
print('Dueling: mean reward = {} | min reward = {} | max reward = {}'.format(np.
 ↪mean(dueling_training_rewards_average), np.
 ↪min(dueling_training_rewards_average), np.
 ↪max(dueling_training_rewards_average)))
print('Double: mean episode duration = {} | min episode duration = {} | max␣
 ↪episode duration = {}'.format(np.mean(double_training_duration_average), np.
 ↪min(double_training_duration_average), np.
 ↪max(double_training_duration_average)))
```

```
print('Double: mean reward = {} | min reward = {} | max reward = {}'.format(np.
 ↪mean(double_training_rewards_average), np.
 ↪min(double_training_rewards_average), np.
 ↪max(double_training_rewards_average)))
print('Dueling Double: mean episode duration = {} | min episode duration = {} |␣
 ↪max episode duration = {}'.format(np.
 ↪mean(dueling_double_training_duration_average), np.
 ↪min(dueling_double_training_duration_average), np.
 ↪max(dueling_double_training_duration_average)))
print('Dueling Double: mean reward = {} | min reward = {} | max reward = {}'.
 ↪format(np.mean(dueling_double_training_rewards_average), np.
 ↪min(dueling_double_training_rewards_average), np.
 ↪max(dueling_double_training_rewards_average)))
```

```
Vanilla: mean episode duration = 254.39182643794146 | min episode duration =
65.9 | max episode duration = 500.0
Vanilla: mean reward = -525.3551800246779 | min reward = -2208.742346156994 |
max reward = -54.9341601362903
Dueling: mean episode duration = 67.08435923309789 | min episode duration = 57.0
| max episode duration = 100.5
Dueling: mean reward = -530.5758897486406 | min reward = -660.4861410105601 |
max reward = -266.4408089472912
Double: mean episode duration = 457.83148335015136 | min episode duration =
100.4 | max episode duration = 500.0
Double: mean reward = 2.0992352925827658 | min reward = -375.5150165274739 | max
reward = 89.39634154002415
Dueling Double: mean episode duration = 465.72653884964683 | min episode
duration = 85.3 | max episode duration = 500.0
Dueling Double: mean reward = 23.77182850561909 | min reward =
-299.40705822137534 | max reward = 101.8286828314449
```

Individual_part

April 21, 2022

0.0.1 RLlib with Atari Learning Environment

Dimitrios Megkos - dimitrios.megkos@city.ac.uk

Environment: Atari Freeway (Ram Version) https://gym.openai.com/envs/Freeway-ram-v0/
"Why did the chicken cross the road?"

A classic atari game where one player (the agent) controls a chicken who can be made to run across a ten lane highway filled with traffic in order to get to the other side. Every time the chicken successufly crosses the road, a point (1) is awarded to the agent. If the chicken is hit by a car it is pushed back to the bottom of the screen. The point of the game is to collect as many points as possible.

0.0.2 Import the libraries

[1]:
```
# Visualization imports
import matplotlib.pyplot as plt
# Ray and RLlib import
import ray
# import tune for hyper parameter experimentation
from ray import tune
from ray.tune import grid_search
# Import a Trainable (one of RLlib's built-in algorithms):
import ray.rllib.agents.dqn as dqn # Deep Q-Network
```

0.0.3 Setting up and trying multiple configurations

There are two ways to experiment with different hyperparameres. The first way is to create different configurations and pass to the trainer inside a for loop that handles the training and stores the rewards which can then be plotted for comparisons. The second and more efficient way is to use Ray's Tune library for experiment execution and hyperparameter tuning. With this method, a grid search can be used to try different hyperparameter combinations, implementing a stoping criteria to control the training.

0.0.4 Method 1: Passing different configurations to the Trainer

Define configurations Two different configurations are defined below, each with different hyperparameters. The first one is a Vanilla DQN, the second is a Double DQN. Epsilon greedy is

used to handle the agent's exploration-exploitation rate.

[2]:
```
# based on lab 07 material
# === FIRST CONFIGURATION ===
config_1 = dqn.DEFAULT_CONFIG.copy() # copy the default configuration
# === Deep Learning Framework Settings ===
config_1["framework"] = "torch" # set pytorch for the framework
# === Environment Settings ===
config_1["env"] = "Freeway-ramDeterministic-v4" # load Atari Freeway from␣
 ↪OpenAI Gym
# === Settings for the Trainer process ===
config_1["gamma"] = 0.99 # MDP discount factor
config_1["lr"] = 0.0001 # the learning rate
config_1["double_q"] = False # whether to use double dqn
# neural network architecture
config_1["model"] = {
    "fcnet_hiddens": [256,256], # hidden layers and neurons
    "fcnet_activation": "relu" # activation function
}
# === Exploration Settings ===
config_1["explore"] = True
config_1["exploration_config"] = {
    "type": "EpsilonGreedy",
    # Parameters for the Exploration class' constructor:
    "initial_epsilon": 1.0,
    "final_epsilon": 0.01,
    "epsilon_timesteps": 20000
}
# === Evaluation Settings ===
config_1["evaluation_duration"] = 5
config_1["evaluation_duration_unit"] = "episodes"
config_1["evaluation_num_workers"] = 1
config_1["evaluation_config"] = {
    "render_env": True,
    "explore": False,
}
# === Resource Settings ===
config_1["num_gpus"] = 0
config_1["num_cpus_per_worker"] = 1



# === SECOND CONFIGURATION ===
config_2 = dqn.DEFAULT_CONFIG.copy() # copy the default configuration
# === Deep Learning Framework Settings ===
config_2["framework"] = "torch" # set pytorch for the framework
# === Environment Settings ===
```

```
config_2["env"] = "Freeway-ramDeterministic-v4" # load Atari Freeway from␣
 ↪OpenAI Gym
# === Settings for the Trainer process ===
config_2["gamma"] = 0.99 # MDP discount factor
config_2["lr"] = 0.0001 # the learning rate
config_2["double_q"] = True # whether to use double dqn
# neural network architecture
config_2["model"] = {
    "fcnet_hiddens": [256,256], # hidden layers and neurons
    "fcnet_activation": "relu" # activation function
}
# === Exploration Settings ===
config_2["explore"] = True
config_2["exploration_config"] = {
    "type": "EpsilonGreedy",
    # Parameters for the Exploration class' constructor:
    "initial_epsilon": 1.0,
    "final_epsilon": 0.01,
    "epsilon_timesteps": 20000
}
# === Evaluation Settings ===
config_2["evaluation_duration"] = 5
config_2["evaluation_duration_unit"] = "episodes"
config_2["evaluation_num_workers"] = 1
config_2["evaluation_config"] = {
    "render_env": True,
    "explore": False,
}
# === Resource Settings ===
config_2["num_gpus"] = 0
config_2["num_cpus_per_worker"] = 1
```

0.0.5 Train with each configuration

Configuration 1 - Train Agent 1

[3]:
```
# based on lab 07 material
# initialise ray
ray.init()

# create our RLlib Trainer.
agent_1 = dqn.DQNTrainer(config=config_1)

# create list to store rewards
avg_rewards_1 = []

# begin training
```

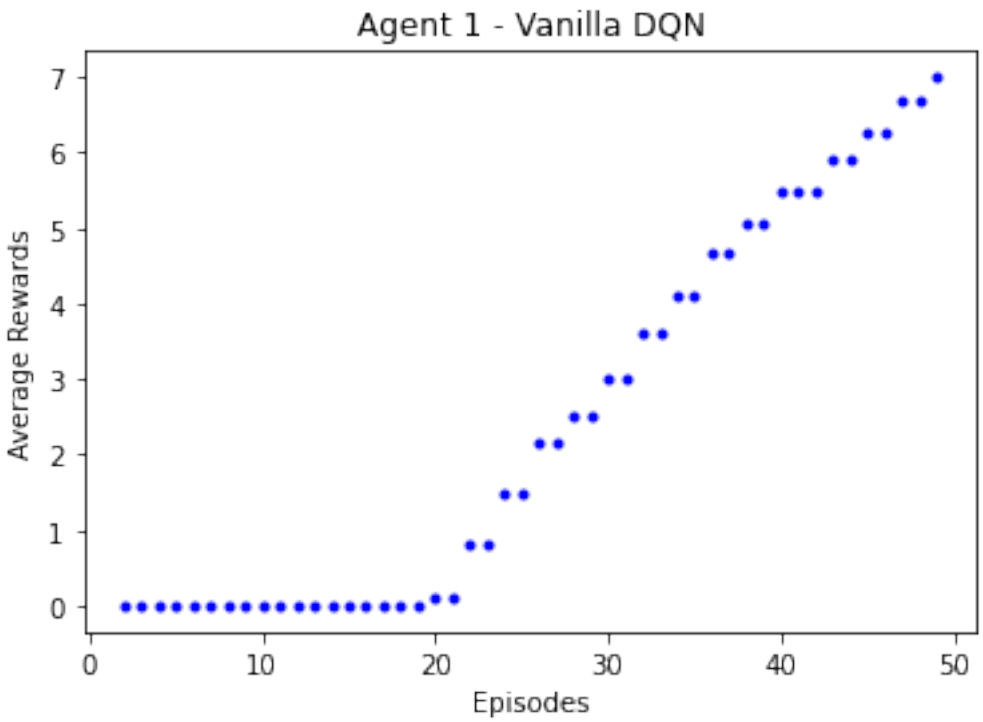
```python
for i in range(50):
    # Perform one iteration of training the policy with DQN
    result = agent_1.train()
    #print(pretty_print(result))
    print(result['episode_reward_mean'])
    avg_rewards_1.append(result['episode_reward_mean'])

# shutdown ray
#ray.shutdown()
```

```
2022-04-13 17:57:23,529 INFO services.py:1412 -- View the Ray dashboard at
http://127.0.0.1:8265
2022-04-13 17:57:26,024 WARNING trainer.py:2347 -- You have specified 1
evaluation workers, but your `evaluation_interval` is None! Therefore,
evaluation will not occur automatically with each call to `Trainer.train()`.
Instead, you will have to call `Trainer.evaluate()` manually in order to trigger
an evaluation run.
2022-04-13 17:57:26,026 INFO simple_q.py:154 -- In multi-agent mode, policies
will be optimized sequentially by the multi-GPU optimizer. Consider setting
`simple_optimizer=True` if this doesn't work for you.
2022-04-13 17:57:26,026 INFO trainer.py:779 -- Current log_level is WARN. For
more information, set 'log_level': 'INFO' / 'DEBUG' or use the -v and -vv flags.
2022-04-13 17:57:26,194 WARNING deprecation.py:45 -- DeprecationWarning:
`simple_optimizer` has been deprecated. This will raise an error in the future!
2022-04-13 17:57:26,195 WARNING trainer.py:2347 -- You have specified 1
evaluation workers, but your `evaluation_interval` is None! Therefore,
evaluation will not occur automatically with each call to `Trainer.train()`.
Instead, you will have to call `Trainer.evaluate()` manually in order to trigger
an evaluation run.
2022-04-13 17:57:26,242 WARNING util.py:55 -- Install gputil for GPU system
monitoring.

nan
nan
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
```

```
0.0
0.0
0.0
0.0
0.1
0.1
0.8181818181818182
0.8181818181818182
1.5
1.5
2.1538461538461537
2.1538461538461537
2.5
2.5
3.0
3.0
3.625
3.625
4.117647058823529
4.117647058823529
4.666666666666667
4.666666666666667
5.052631578947368
5.052631578947368
5.5
5.5
5.5
5.904761904761905
5.904761904761905
6.2727272727272725
6.2727272727272725
6.695652173913044
6.695652173913044
7.0
```

Plot Agent 1 Rewards

[4]:
```
# plot the rewards of agent 1
plt.title("Agent 1 - Vanilla DQN")
plt.xlabel("Episodes")
plt.ylabel("Average Rewards")
a1=plt.plot(avg_rewards_1, 'b.')
```



Evaluate Agent 1

[8]:
```
# evaluate agent for 5 episodes
agent_1.evaluate()
# shutdown ray
ray.shutdown()
#show agent_1 evaluation metrics
agent_1.evaluation_metrics
```

[8]:
```
{'evaluation': {'episode_reward_max': 21.0,
  'episode_reward_min': 21.0,
  'episode_reward_mean': 21.0,
  'episode_len_mean': 2048.0,
  'episode_media': {},
  'episodes_this_iter': 5,
  'policy_reward_min': {},
  'policy_reward_max': {},
  'policy_reward_mean': {},
  'custom_metrics': {},
  'hist_stats': {'episode_reward': [21.0, 21.0, 21.0, 21.0, 21.0],
   'episode_lengths': [2048, 2048, 2048, 2048, 2048]},
  'sampler_perf': {'mean_raw_obs_processing_ms': 0.08932406426686676,
   'mean_inference_ms': 0.5525182832580632,
   'mean_action_processing_ms': 0.02557385562628161,
```

```
    'mean_env_wait_ms': 0.6681889158803357,
    'mean_env_render_ms': 1.1005202560566352},
   'off_policy_estimator': {},
   'timesteps_this_iter': 0}}
```

Configuration 2 - Train Agent 2

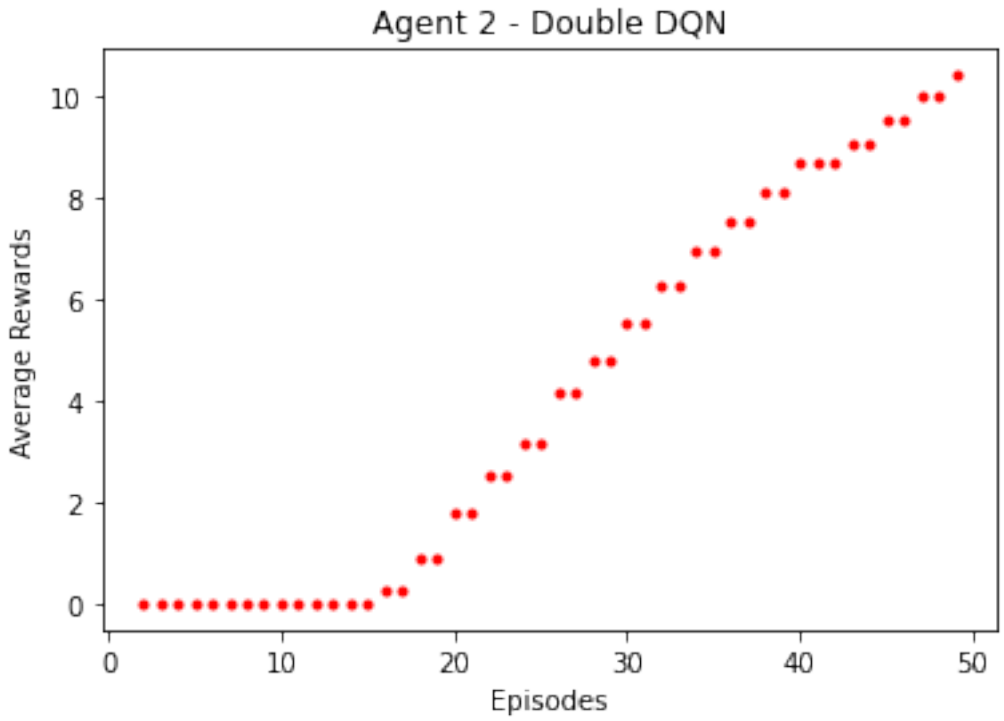
[5]:
```
# based on lab 07 material
# initialise ray
ray.init()

# create our RLlib Trainer.
agent_2 = dqn.DQNTrainer(config=config_2)

# create list to store rewards
avg_rewards_2 = []

# begin training
for i in range(50):
    # Perform one iteration of training the policy with DQN
    result = agent_2.train()
    #print(pretty_print(result))
    print(result['episode_reward_mean'])
    avg_rewards_2.append(result['episode_reward_mean'])

# shutdown ray
#ray.shutdown()
```

```
2022-04-13 18:01:28,354 WARNING trainer.py:2347 -- You have specified 1
evaluation workers, but your `evaluation_interval` is None! Therefore,
evaluation will not occur automatically with each call to `Trainer.train()`.
Instead, you will have to call `Trainer.evaluate()` manually in order to trigger
an evaluation run.
2022-04-13 18:01:28,502 WARNING deprecation.py:45 -- DeprecationWarning:
`simple_optimizer` has been deprecated. This will raise an error in the future!
2022-04-13 18:01:28,502 WARNING trainer.py:2347 -- You have specified 1
evaluation workers, but your `evaluation_interval` is None! Therefore,
evaluation will not occur automatically with each call to `Trainer.train()`.
Instead, you will have to call `Trainer.evaluate()` manually in order to trigger
an evaluation run.
2022-04-13 18:01:28,543 WARNING util.py:55 -- Install gputil for GPU system
monitoring.

nan
nan
0.0
0.0
0.0
```

```
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.25
0.25
0.8888888888888888
0.8888888888888888
1.8
1.8
2.5454545454545454
2.5454545454545454
3.1666666666666665
3.1666666666666665
4.153846153846154
4.153846153846154
4.785714285714286
4.785714285714286
5.533333333333333
5.533333333333333
6.25
6.25
6.9411764705882355
6.9411764705882355
7.555555555555555
7.555555555555555
8.105263157894736
8.105263157894736
8.7
8.7
8.7
9.047619047619047
9.047619047619047
9.545454545454545
9.545454545454545
10.0
10.0
10.416666666666666
```

Plot Agent 2 Rewards

[6]:
```
# plot the rewards of agent 2
plt.title("Agent 2 - Double DQN")
plt.xlabel("Episodes")
plt.ylabel("Average Rewards")
a2 = plt.plot(avg_rewards_2, 'r.')
```



Evaluate Agent 2

[7]:
```
# evaluate agent for 5 episodes
agent_2.evaluate()
# shutdown ray
ray.shutdown()
#show agent_2 evaluation metrics
agent_2.evaluation_metrics
```

[7]:
```
{'evaluation': {'episode_reward_max': 22.0,
  'episode_reward_min': 22.0,
  'episode_reward_mean': 22.0,
  'episode_len_mean': 2048.0,
  'episode_media': {},
  'episodes_this_iter': 5,
  'policy_reward_min': {},
  'policy_reward_max': {},
  'policy_reward_mean': {},
```

```
'custom_metrics': {},
'hist_stats': {'episode_reward': [22.0, 22.0, 22.0, 22.0, 22.0],
 'episode_lengths': [2048, 2048, 2048, 2048, 2048]},
'sampler_perf': {'mean_raw_obs_processing_ms': 0.0861592968394195,
 'mean_inference_ms': 0.54730375860017,
 'mean_action_processing_ms': 0.025069873065946623,
 'mean_env_wait_ms': 0.6753676874238804,
 'mean_env_render_ms': 1.1064958297674363},
'off_policy_estimator': {},
'timesteps_this_iter': 0}}
```

Analysis of results

- During the first trials, a network architecture of one hidden layer with 512 hidden neurons was used for both Vanilla DQN and Double DQN. Vanilla DQN did not manage to receive any rewards after 50 iterations, achieving a mean reward of zero during evaluation. Double DQN managed to receive 1.20 average rewards after 50 iterations, achieving a mean reward of 11 during evaluation
- Changing the architecture of the networks to two hidden layers with 256 hidden neurons each greatly improved the performance of both Vanilla DQN and Double DQN. Having two hidden layers, the agents were able to perform more complex calculations enabling them to learn faster and better. Vanilla DQN started earning rewards after 19 iterations, reaching an average of 7 rewards after 50 iterations and achieving a mean reward of 21 during evaluation. Double DQN started earning rewards after 15 iterations, reaching an average of 10.40 rewards after 50 iterations and achieving a mean reward of 22 during evaluation.
- Although both agents achieved similar evaluation scores, Double DQN is the winner here because it not only because it started learning faster, but also because it managed to achieve higher score after 50 training iterations. This makes sense considering a Double DQN is using a second network to calculate the target values.

0.0.6 Method 2: Using the Tune Library

Using Agent 2 configuration (Double DQN), a grid search is going to be performed in order to experiment more and fine tune the hyperparameters. The parameters that seem to affect learning the most are the *learning rate*, the *number of hidden layers*, the *number of hidden neurons* and the *type of activation function*, therefore the grid search will focus on those parameters.

Define the grid search configuration

[16]:
```
# === GRID SEARCH CONFIGURATION ===
gs_config = dqn.DEFAULT_CONFIG.copy() # copy the default configuration
# === Deep Learning Framework Settings ===
gs_config["framework"] = "torch" # set pytorch for the framework
# === Environment Settings ===
gs_config["env"] = "Freeway-ramDeterministic-v4" # load Atari Freeway from␣
 ↪OpenAI Gym
# === Settings for the Trainer process ===
gs_config["gamma"] = 0.99 # MDP discount factor
```

```
gs_config["lr"] = grid_search([0.001,0.0001]) # the learning rate
gs_config["double_q"] = True # whether to use double dqn
# neural network architecture
gs_config["model"] = {
    "fcnet_hiddens": grid_search([[256,256], [256,256,256]]), # hidden layers␣
 ↪and neurons
    "fcnet_activation": grid_search(["relu","linear"]) # activation function
}
# === Exploration Settings ===
gs_config["explore"] = True
gs_config["exploration_config"] = {
    "type": "EpsilonGreedy",
    # Parameters for the Exploration class' constructor:
    "initial_epsilon": 1.0,
    "final_epsilon": 0.01,
    "epsilon_timesteps": 10000
}
```

Run the Tuning The tuning will run until the agents reach a mean episode reward of 12 and Tune will return the best hyperparameter combination that required the least total timesteps to reach the goal.

[ ]:
```
# inspired by https://www.anyscale.com/blog/
 ↪an-introduction-to-reinforcement-learning-with-openai-gym-rllib-and-google
# initialize ray
ray.init()

# begin tune analysis
gs_analysis = tune.run(
    dqn.DQNTrainer, # the trainer
    config=gs_config, # use the grid search configuration
    stop={"episode_reward_mean": 12}, # stop criteria
    metric="timesteps_total", # comparisons based on total timesteps
    mode="min" # choose based on the least total timesteps
)

# shutdown ray
ray.shutdown()
```

Printing the best configuration

[18]:
```
# print out best hyperparameters
print(
  "Best hyperparameters found:",
  gs_analysis.best_config,
)
```

Best hyperparameters found: {'num_workers': 0, 'num_envs_per_worker': 1,
'create_env_on_driver': False, 'rollout_fragment_length': 4, 'batch_mode':
'truncate_episodes', 'gamma': 0.99, 'lr': 0.001, 'train_batch_size': 32,
'model': {'fcnet_hiddens': [256, 256], 'fcnet_activation': 'relu'}, 'optimizer':
{}, 'horizon': None, 'soft_horizon': False, 'no_done_at_end': False, 'env':
'Freeway-ramDeterministic-v4', 'observation_space': None, 'action_space': None,
'env_config': {}, 'remote_worker_envs': False, 'remote_env_batch_wait_ms': 0,
'env_task_fn': None, 'render_env': False, 'record_env': False, 'clip_rewards':
None, 'normalize_actions': True, 'clip_actions': False, 'preprocessor_pref':
'deepmind', 'log_level': 'WARN', 'callbacks': <class
'ray.rllib.agents.callbacks.DefaultCallbacks'>, 'ignore_worker_failures': False,
'log_sys_usage': True, 'fake_sampler': False, 'framework': 'torch',
'eager_tracing': False, 'eager_max_retraces': 20, 'explore': True,
'exploration_config': {'type': 'EpsilonGreedy', 'initial_epsilon': 1.0,
'final_epsilon': 0.01, 'epsilon_timesteps': 10000}, 'evaluation_interval': None,
'evaluation_duration': 10, 'evaluation_duration_unit': 'episodes',
'evaluation_parallel_to_training': False, 'in_evaluation': False,
'evaluation_config': {'explore': False}, 'evaluation_num_workers': 0,
'custom_eval_function': None, 'always_attach_evaluation_results': False,
'sample_async': False, 'sample_collector': <class
'ray.rllib.evaluation.collectors.simple_list_collector.SimpleListCollector'>,
'observation_filter': 'NoFilter', 'synchronize_filters': True,
'tf_session_args': {'intra_op_parallelism_threads': 2,
'inter_op_parallelism_threads': 2, 'gpu_options': {'allow_growth': True},
'log_device_placement': False, 'device_count': {'CPU': 1},
'allow_soft_placement': True}, 'local_tf_session_args':
{'intra_op_parallelism_threads': 8, 'inter_op_parallelism_threads': 8},
'compress_observations': False, 'metrics_episode_collection_timeout_s': 180,
'metrics_num_episodes_for_smoothing': 100, 'min_time_s_per_reporting': 1,
'min_train_timesteps_per_reporting': None, 'min_sample_timesteps_per_reporting':
None, 'seed': None, 'extra_python_environs_for_driver': {},
'extra_python_environs_for_worker': {}, 'num_gpus': 0, '_fake_gpus': False,
'num_cpus_per_worker': 1, 'num_gpus_per_worker': 0,
'custom_resources_per_worker': {}, 'num_cpus_for_driver': 1,
'placement_strategy': 'PACK', 'input': 'sampler', 'input_config': {},
'actions_in_input_normalized': False, 'input_evaluation': ['is', 'wis'],
'postprocess_inputs': False, 'shuffle_buffer_size': 0, 'output': None,
'output_compress_columns': ['obs', 'new_obs'], 'output_max_file_size': 67108864,
'multiagent': {'policies': {}, 'policy_map_capacity': 100, 'policy_map_cache':
None, 'policy_mapping_fn': None, 'policies_to_train': None, 'observation_fn':
None, 'replay_mode': 'independent', 'count_steps_by': 'env_steps'},
'logger_config': None, '_tf_policy_handles_more_than_one_loss': False,
'_disable_preprocessor_api': False, '_disable_action_flattening': False,
'_disable_execution_plan_api': False, 'simple_optimizer': -1, 'monitor': -1,
'evaluation_num_episodes': -1, 'metrics_smoothing_episodes': -1,
'timesteps_per_iteration': 1000, 'min_iter_time_s': -1,
'collect_metrics_timeout': -1, 'target_network_update_freq': 500, 'buffer_size':
-1, 'replay_buffer_config': {'type': 'MultiAgentReplayBuffer', 'capacity':

Extra_part

April 20, 2022

0.0.1 PPO Implementation on Lunar Lander environment using RLlib

0.0.2 Alessandro Alviani - alessandro.alviani@city.ac.uk

0.0.3 Dimitrios Megkos - dimitrios.megkos@city.ac.uk

The PPO algorithm was implemented using Ray RLlib on the OpenAI Gym Environment: Lunar Lander

Import Libraries

[1]:
```
# Visualization imports
import matplotlib.pyplot as plt
# Ray and RLlib import
import ray
# Import a Trainable (one of RLlib's built-in algorithms):
import ray.rllib.agents.ppo as ppo # PPO Algorithm
```

Load default configuration and change parameters

[3]:
```
# based on lab 07 material
ppo_config = ppo.DEFAULT_CONFIG.copy() # copy the default configuration

# Choose framework
ppo_config["framework"] = "torch" # set pytorch

# Environment setting
ppo_config["env"] = "LunarLander-v2" # LunarLander from OpenAI Gym

# Training settings
ppo_config["lr"] = 0.001 # the learning rate
# neural network architecture
ppo_config["model"] = {
    "fcnet_hiddens": [32,32], # hidden layers and neurons
    "fcnet_activation": "linear" # activation function
}
# Exploration Settings
ppo_config["explore"] = True
ppo_config["exploration_config"] = {
```

```
    "type": "StochasticSampling"
}
# Evaluation Settings
ppo_config["evaluation_duration"] = 5
ppo_config["evaluation_duration_unit"] = "episodes"
ppo_config["evaluation_num_workers"] = 1
ppo_config["evaluation_config"] = {
    "render_env": True,
    "explore": False,
}
# Resource Settings
ppo_config["num_gpus"] = 0
ppo_config["num_cpus_per_worker"] = 1
```

Train PPO Trainer

[3]:
```
# based on lab 07 material
# initialise ray
ray.init()

# create our RLlib Trainer.
ppo_agent = ppo.PPOTrainer(config=ppo_config)

# create list to store rewards
ppo_avg_rewards = []

# begin training
for i in range(50):
    # Perform one iteration of training the policy with DQN
    result = ppo_agent.train()
    #print(pretty_print(result))
    print(result['episode_reward_mean'])
    ppo_avg_rewards.append(result['episode_reward_mean'])
```

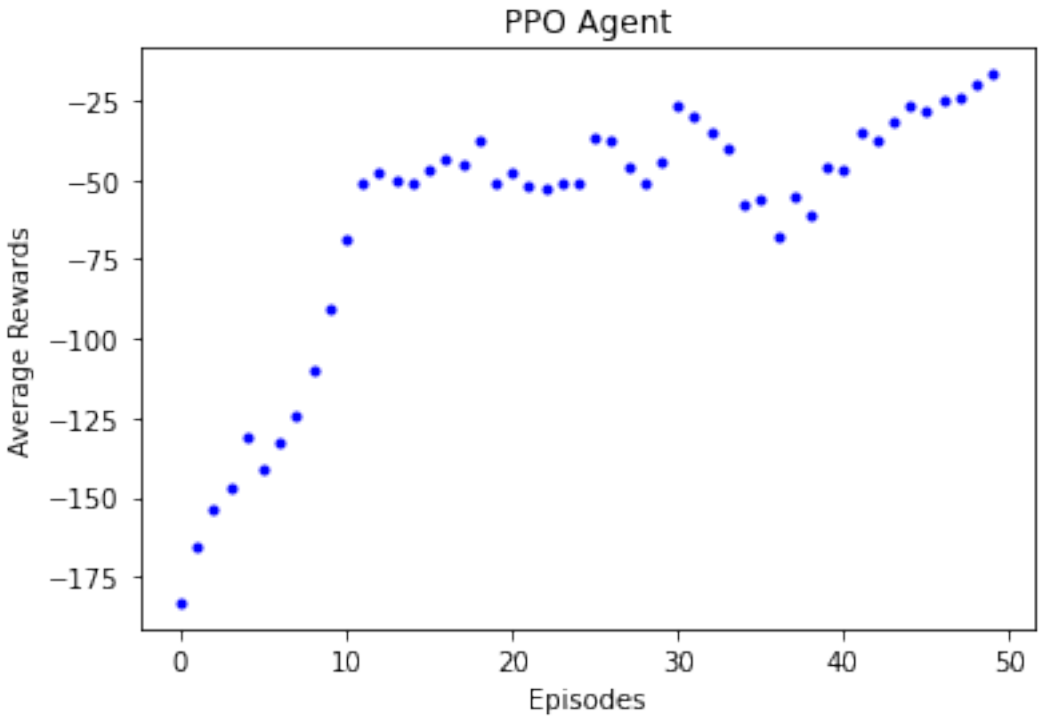
```
2022-04-20 20:12:27,962 INFO services.py:1412 -- View the Ray dashboard at
http://127.0.0.1:8265
2022-04-20 20:12:31,569 WARNING trainer.py:2347 -- You have specified 1
evaluation workers, but your `evaluation_interval` is None! Therefore,
evaluation will not occur automatically with each call to `Trainer.train()`.
Instead, you will have to call `Trainer.evaluate()` manually in order to trigger
an evaluation run.
2022-04-20 20:12:31,571 INFO ppo.py:249 -- In multi-agent mode, policies will be
optimized sequentially by the multi-GPU optimizer. Consider setting
simple_optimizer=True if this doesn't work for you.
2022-04-20 20:12:31,571 INFO trainer.py:779 -- Current log_level is WARN. For
more information, set 'log_level': 'INFO' / 'DEBUG' or use the -v and -vv flags.
2022-04-20 20:12:32,908 WARNING deprecation.py:45 -- DeprecationWarning:
`simple_optimizer` has been deprecated. This will raise an error in the future!
```

```
2022-04-20 20:12:32,908 WARNING trainer.py:2347 -- You have specified 1
evaluation workers, but your `evaluation_interval` is None! Therefore,
evaluation will not occur automatically with each call to `Trainer.train()`.
Instead, you will have to call `Trainer.evaluate()` manually in order to trigger
an evaluation run.
2022-04-20 20:12:32,944 WARNING util.py:55 -- Install gputil for GPU system
monitoring.

-183.04556853923742
-165.8394753462788
-153.45996959954866
-146.65110893172488
-131.09945448903346
-140.8902431263759
-132.9459810350038
-124.32291093312102
-109.8856436322202
-90.32204458139859
-68.49888691471476
-51.46861014658593
-47.41790033276456
-50.44706572335351
-51.4560472220311
-46.91478109893111
-43.87973794767229
-45.536730330573576
-37.83821304486864
-51.02406342112563
-47.365202859936005
-52.0823743006126
-52.83816927775186
-50.68179497300291
-50.88423758379716
-36.648375272751146
-37.799695718245054
-45.7301017177828
-51.40944233853542
-44.034784263889215
-27.106279199932537
-29.753629417095294
-34.69300875551924
-40.54422644242287
-58.19031412400278
-55.7576897576254
-67.59906624168647
-55.245746782858184
-61.42868410698823
-45.786434237032516
```

```
-46.5938352389612
-35.398563853021955
-37.75025574222145
-31.50348963070189
-26.951574493635526
-28.7807665419684
-25.045941408783815
-23.803775043072473
-20.03284633082973
-16.804886616995937
```

Plot PPO Rewards

[4]:
```
# plot the rewards of agent 1
plt.title("PPO Agent")
plt.xlabel("Episodes")
plt.ylabel("Average Rewards")
ppo_r=plt.plot(ppo_avg_rewards, 'b.')
```



Play the Game and Evaluate

[5]:
```
# evaluate agent for 5 episodes
ppo_agent.evaluate()
# shutdown ray
ray.shutdown()
```

```
#show agent_1 evaluation metrics
ppo_agent.evaluation_metrics
```

[5]:
```
{'evaluation': {'episode_reward_max': -48.91031093714949,
  'episode_reward_min': -196.96654140575785,
  'episode_reward_mean': -85.2119269865243,
  'episode_len_mean': 115.8,
  'episode_media': {},
  'episodes_this_iter': 5,
  'policy_reward_min': {},
  'policy_reward_max': {},
  'policy_reward_mean': {},
  'custom_metrics': {},
  'hist_stats': {'episode_reward': [-196.96654140575785,
    -50.732946942923014,
    -48.91031093714949,
    -65.94235703186516,
    -63.50747861492594],
   'episode_lengths': [249, 96, 80, 94, 60]},
  'sampler_perf': {'mean_raw_obs_processing_ms': 0.16737066466232825,
   'mean_inference_ms': 0.7357601461739375,
   'mean_action_processing_ms': 0.037945961130076436,
   'mean_env_wait_ms': 0.22187972890919655,
   'mean_env_render_ms': 16.486269030077704},
  'off_policy_estimator': {},
  'timesteps_this_iter': 0}}
```

Training the agent on fifty iterations returned an average training reward of -16.80, up from an initial average training reward of -183. The plot shows the agent is learning and we expect the model will keep improving by increasing the number of training iterations. The agent evaluation on five episodes returned a mean episode reward of -85.

Architecture and hyperparameters optimization In the cell below we: - increased the number of iterations: from 50 to 200 - added more hidden neurons to each layer: from 32 to 64 - changed activation function: from linear to ReLU - decreased the learning rate: from 0.001 to 0.0001

[2]:
```
# based on lab 07 material
ppo_config = ppo.DEFAULT_CONFIG.copy() # copy the default configuration

# Choose framework
ppo_config["framework"] = "torch" # set pytorch

# Environment setting
ppo_config["env"] = "LunarLander-v2" # LunarLander from OpenAI Gym

# Training settings
ppo_config["lr"] = 0.0001 # the learning rate
# neural network architecture
ppo_config["model"] = {
```

```
    "fcnet_hiddens": [64,64], # hidden layers and neurons
    "fcnet_activation": "relu" # activation function
}
# Exploration Settings
ppo_config["explore"] = True
ppo_config["exploration_config"] = {
    "type": "StochasticSampling"
}
# Evaluation Settings
ppo_config["evaluation_duration"] = 5
ppo_config["evaluation_duration_unit"] = "episodes"
ppo_config["evaluation_num_workers"] = 1
ppo_config["evaluation_config"] = {
    "render_env": True,
    "explore": False,
}
# Resource Settings
ppo_config["num_gpus"] = 0
ppo_config["num_cpus_per_worker"] = 1
```

Training the PPO Agent

[3]:
```
# based on lab 07 material
# initialise ray
ray.init()

# create our RLlib Trainer.
ppo_agent = ppo.PPOTrainer(config=ppo_config)

# create list to store rewards
ppo_avg_rewards = []

# begin training
for i in range(200):
    # Perform one iteration of training the policy with DQN
    result = ppo_agent.train()
    #print(pretty_print(result))
    print(result['episode_reward_mean'])
    ppo_avg_rewards.append(result['episode_reward_mean'])
```

```
2022-04-20 20:35:56,861 INFO services.py:1412 -- View the Ray dashboard at
http://127.0.0.1:8265
2022-04-20 20:35:59,413 WARNING trainer.py:2347 -- You have specified 1
evaluation workers, but your `evaluation_interval` is None! Therefore,
evaluation will not occur automatically with each call to `Trainer.train()`.
Instead, you will have to call `Trainer.evaluate()` manually in order to trigger
an evaluation run.
```

```
2022-04-20 20:35:59,415 INFO ppo.py:249 -- In multi-agent mode, policies will be
optimized sequentially by the multi-GPU optimizer. Consider setting
simple_optimizer=True if this doesn't work for you.
2022-04-20 20:35:59,416 INFO trainer.py:779 -- Current log_level is WARN. For
more information, set 'log_level': 'INFO' / 'DEBUG' or use the -v and -vv flags.
2022-04-20 20:36:00,605 WARNING deprecation.py:45 -- DeprecationWarning:
`simple_optimizer` has been deprecated. This will raise an error in the future!
2022-04-20 20:36:00,606 WARNING trainer.py:2347 -- You have specified 1
evaluation workers, but your `evaluation_interval` is None! Therefore,
evaluation will not occur automatically with each call to `Trainer.train()`.
Instead, you will have to call `Trainer.evaluate()` manually in order to trigger
an evaluation run.
2022-04-20 20:36:00,642 WARNING util.py:55 -- Install gputil for GPU system
monitoring.

-197.21366791781733
-184.94749155126982
-162.4880308185099
-136.74975010776453
-121.39607656610356
-110.6031179518207
-97.85391948953007
-72.41645864476261
-73.3090285307518
-68.7427816808708
-58.19292857901161
-50.279543553578534
-45.726307422059854
-37.2915672877457
-33.603718778114214
-28.4738301531513
-22.807589630562163
-22.088263530286472
-13.250492337909892
-5.782938846276974
2.4914114412346904
6.7738415089066155
18.115394305060313
26.661643764143697
38.09131652635957
52.590692722752244
75.78862418665392
93.6191031657765
112.15827228526534
128.17954998078278
150.54705199450092
161.69398714005794
162.1107733427962
```

167.27299569946845
179.6378932371005
168.9153461567753
176.05437797868188
177.56676639272973
180.899166011306
184.8274047130174
196.8442527245489
201.92882112905926
196.70421967300442
203.69519651374085
209.43030961157638
208.33069442204666
217.69161063506442
221.4768854472209
223.7180754801493
219.7321988254545
228.0130977287696
229.5566703380244
227.13443268099937
227.7450674128464
225.6261295541008
225.5882005383353
220.6489035812464
229.33965642293342
228.23301017105987
227.72654000161216
233.5438978616678
235.18736789337584
233.40194715786242
236.03937454563427
240.42114814131793
243.98762755746938
250.13235754955855
249.3782249985487
250.0089992455786
251.53789077376513
244.41177897603234
243.42585132641875
231.2246374686544
224.36008057973007
222.65441995661826
218.8905170490888
209.8855107722831
207.24555322974516
197.63382195437666
197.5222659782326
201.10880788066436

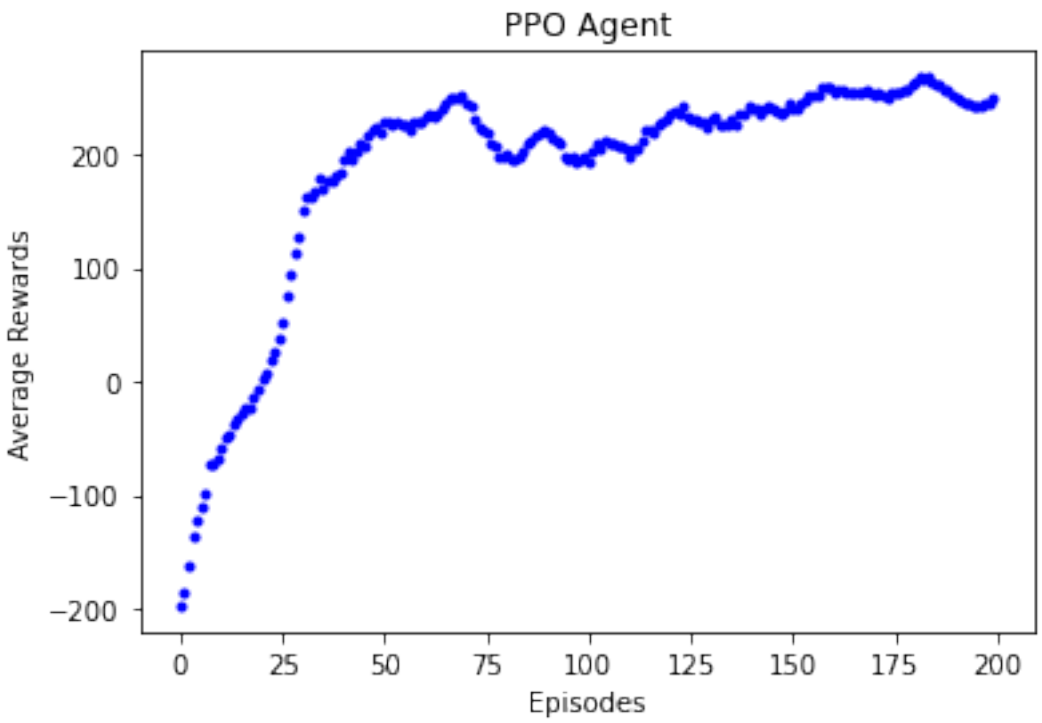
196.5730661745098
195.25458516678214
196.95659259954738
201.88772545118343
208.83051853454484
211.11889042302153
216.87878230895487
220.35713035019336
222.00756505933236
218.20440725410586
214.9352578415801
211.9755648565371
209.535048596104
197.0043059927224
196.73565161859034
197.18365836130278
194.40387521643098
195.0628252075818
198.001449821235
194.3497898666953
202.86498811783105
210.04281223066428
205.05157775320615
212.1777423540008
210.18576416631674
210.3068481015036
207.792881878037
207.44550686835865
204.0349073148542
198.2281028067164
204.44296545386854
203.9784795039259
212.99490159495778
221.40091566691677
222.40517832229426
220.01979282711488
226.35372409852738
228.4950956709154
232.0120100993407
234.51193634408446
236.93470237442727
235.56275873052778
243.22315518237545
232.20513631572123
230.2157032252234
231.37625770467363
229.6782522500494
228.92064042125563

224.24120670590116
231.6168869703373
234.23508128564387
227.0594703809386
227.37894934173582
225.43460689994157
231.23580234758938
226.7989913434838
234.5426607732786
234.93047458687712
243.6497897392551
240.94075988543472
239.69436894294486
235.82054416686557
239.62554379540606
242.2066789285332
240.10621789121714
238.675493295146
236.44818752773946
237.97204990429498
245.20134642881138
241.5455170760054
239.92754177354186
243.97226248887492
248.35020168251708
251.11580972456343
253.2292146047913
251.5934530023598
259.6664114773828
259.47074128580533
258.0832668664926
254.78913226996502
256.63710925756635
257.82307868503864
255.3148607696738
255.36590644547243
253.8690990691766
253.65041885154776
254.9732382094283
257.1959177691289
255.13195785290364
252.17235318771523
254.1822640619529
252.0526308947984
250.2927732223405
253.43192535312318
253.43812833008647
254.4122135436679

```
257.9180146051916
257.6731928256974
262.7223439821329
264.869431738095
268.06586510373137
267.22558279437845
267.80007218949106
264.2833971292542
262.5287272354645
260.79295378781865
256.5499045693005
255.73216485493674
251.62774194182583
249.74288461492407
247.9323751218445
244.43342377905373
244.21049206935243
241.82810002297146
242.66729838308254
241.60129232051648
243.93444267662
244.9354280337674
248.96321322577595
```

Plot PPO Rewards

[4]:
```
# plot the rewards of agent 1
plt.title("PPO Agent")
plt.xlabel("Episodes")
plt.ylabel("Average Rewards")
ppo_r=plt.plot(ppo_avg_rewards, 'b.')
```



Play the Game and Evaluate

[5]:
```
# evaluate agent for 5 episodes
ppo_agent.evaluate()
# shutdown ray
ray.shutdown()
#show agent_1 evaluation metrics
ppo_agent.evaluation_metrics
```

[5]:
```
{'evaluation': {'episode_reward_max': 284.8348203507567,
  'episode_reward_min': 163.2844277081189,
  'episode_reward_mean': 250.58314113528596,
  'episode_len_mean': 418.2,
  'episode_media': {},
  'episodes_this_iter': 5,
  'policy_reward_min': {},
  'policy_reward_max': {},
  'policy_reward_mean': {},
  'custom_metrics': {},
  'hist_stats': {'episode_reward': [284.8348203507567,
    163.2844277081189,
    259.8906202710482,
    281.80117146538663,
    263.10466588111933],
```

```
 'episode_lengths': [275, 1000, 232, 231, 353]},
'sampler_perf': {'mean_raw_obs_processing_ms': 0.15922919286140746,
 'mean_inference_ms': 0.8781022369063833,
 'mean_action_processing_ms': 0.03863782773282286,
 'mean_env_wait_ms': 0.23275898701830072,
 'mean_env_render_ms': 15.583472201746691},
'off_policy_estimator': {},
'timesteps_this_iter': 0}}
```


Conclusions The optimized model resulted in a much more effective PPO Agent. Over just 200 training iterations, the model in evaluation reached an average training rewards of 250 points. The smaller learning rate helped achieve a steadier training, and the higher number of hidden neurons perfected the agent's movement, hence generated higher rewards. The model in training plateaud after about 75 games (as shown in the rewards plot), indicating that the training could be early-stoppped and the agent successfuly trained with less than 100 training iterations. In evaluation, the agent collected a mean episode reward of 250 points over five episodes, a nearly maximum score, and result of an agent that carried out an almost perfect landing (right in the middle of the goal) at each attempt.

```
# These functions are used in DRL_Part_1 (Q-learning)

# Required imports
import numpy as np
import matplotlib.pyplot as plt

# A custom function that initializes our environment. The environment is inspired by OpenAI Gym's FrozenLake.
# The agent is placed in a 8x8 grid world, beginning always from the same state, trying to reach the other side
# of the lake. The agent navigates through a frozen path, trying to avoid a number of holes the lake has. If the
# agent falls into a hole, the agent starts again from the initial state. Because the path is frozen, the agent
# will sometimes move to a different direction than the one that was picked. There are 64 available states
# and four available actions (Up:0, Down:1, Left:2, Right:3). The 8x8 grid is defined below:
#
#  "8x8": [    SFFFHFFF
#              FHFFFHFF
#              FFHFFFHF
#              FFFHHFHF
#              FFHHFFFH
#              FFFHFHFH
#              FFHFFHHH
#              FHFFFFFG
#              ]
#
# Where S=Safe Spot (Agent follows the action that was chosen), F=Frozen, H=Hole, G=Goal
# Which can be mapped to the below states:
#
#  "8x8": [   0,1,2,3,4,5,6,7,
#             8,9,10,11,12,13,14,15
#             16,17,18,19,20,21,22,23
#             24,25,26,27,28,29,30,31,
#             32,33,34,35,36,37,38,39,
#             40,41,42,43,44,45,46,47,
#             48,49,50,51,52,53,54,55,
#             56,57,58,59,60,61,62,63
#             ]

# Rewards for R matrix: S=0, F=-0.5, H=-1, G=1
# Output: S (set of states), A (set of actions), R Matrix, Q Matrix, state (starting state)
# goal state (the exit), hole state (the states that have holes)


def init_env():
    # Define set of states
    S = np.arange(64)

    # Define the goal
    goal_state = 63

    # Define the holes
    hole_state = [11, 13, 17, 22, 26, 27, 28, 34, 43, 45, 47, 48, 50, 53, 54, 55]

    # Define set of actions and map to numbers
    UP = 0
    DOWN = 1
    LEFT = 2
    RIGHT = 3
    A = [UP, DOWN, LEFT, RIGHT]

    # Initialize R matrix with dimensions A x S
    R = np.empty((len(S), len(A)))

    # Populate R matrix with possible actions and rewards
    R[:] = -0.5  # Populate with -0.5 (F)

    # Populate with 0s (Safe spot), action without random probability for the movement

    # R[8, 0] = 0
```

```
    # R[1, 2] = 0

    # Populate with 1s, the goal
    R[62, 3] = 100

    # Populate with -1s, the holes
    for i in hole_state:
        # the if statements avoid the index being out bounds
        if i - 1 < 0:
            continue
        if i - 8 < 0:
            continue
        if i + 1 > 63:
            continue
        if i + 8 > 63:
            continue
        # sets the states that lead to the holes to -10
        R[i-1, 3] = -10
        R[i+1, 2] = -10
        R[i-8, 1] = -10
        R[i+8, 0] = -10


   # Populate with NaNs, actions that can't be executed (there's a wall)
    for i in range(8):  # upper wall - agent can't go up
        R[i, 0] = np.nan
    for i in np.arange(0, 57, 8):  # left wall - agent can't go left
        R[i, 2] = np.nan
    for i in np.arange(7, 64, 8):  # right wall - agent can't go right
        R[i, 3] = np.nan
    for i in np.arange(56, 64):  # lower wall - agent can't go down
        R[i, 1] = np.nan

    # Initialize Q matrix with dimensions A x S and populate with zeros
    Q = np.zeros(R.shape)

    # Initialize starting state
    state = S[0]  # We always begin from state 0
    #print("Starting state is '{}', which is row {} in the Q and R matrices".format(S[state], state))
    return S, A, R, Q, state, goal_state, hole_state


# A custom function that handles the selection of the action to be executed by the agent
# Input: R matrix, Q matrix, S (the list of all states), A (the list of actions), state (the current state of the agent)
# Output: action (the action the agent is going to execute)


def select_action(R, Q, S, A, state, epsilon):
    # Check all available actions
    available_actions = np.where(~np.isnan(R[state]))[0]  # Identify available actions
    # print("The available actions from state '{}' are: {}".format(
    # S[state], [A[x] for x in available_actions]))

    # The current Q values for each action
    q_values = [Q[state, action] for action in available_actions]
    #print('The Q values for those actions from current state are: {}'.format(q_values))

    # Select an action
    # Check the best actions
    best_actions = available_actions[np.where(q_values == np.max(q_values))[0]]

    # The current Q values of the best actions
    #best_actions_q_values = [Q[state, a] for a in best_actions]

    # if len(best_actions) > 1:  # In case there are more than one best actions
    #print('Detected multiple actions with identical Q values. Agent will randomly select one of these.')
```

```
    # print('Our best available actions from here are: {} with current q values: {}'.format(
    # [A[x] for x in best_actions], best_actions_q_values))

    # Epsilon-greedy
    if np.random.uniform() > epsilon:
        action = np.random.choice(available_actions)
        #print("Selecting random action '{}' with current Q value {}".format(A[action], Q[state, action]))
    else:
        action = np.random.choice(best_actions)
        #print("Selecting greedy action '{}' with current Q value {}".format(A[action], Q[state, action]))

    # Implement the Freezing Random action - this option has been deactivated as too challenging for the agent on our grid.
    # if state in range(64):
    #     if np.random.uniform() > 0.9:  # There is a 10% chance the agent "slips" and chooses a random action
    #         action = np.random.choice(available_actions)

    return action  # Return the action the agent is going to execute

# A custom function that handles the movement of the agent inside the 8x8 grid.
# Input: state (the current state of the agent), action (the action the agent is going to take)
# Output: state (the new state of the agent after executing the given action)


def select_step(state, action):
    if action == 0:
        state = state - 8  # Move up
    elif action == 1:
        state = state + 8  # Move down
    elif action == 2:
        state = state - 1  # Move left
    else:
        state = state + 1  # Move right
    return state

# A custom function that updates the relevant Q-Value of the agent.
# Input: alpha (learning rate), gamma (the discount parameter), Q matrix, state (current state),
# old_state (the previous state), r (the immediate reward), action
# Output: Q Matrix (the updated version)


def update_qvalue(alpha, gamma, Q, state, old_state, r, action):
    Q[old_state, action] = Q[old_state, action] + alpha * \
        (r + gamma * np.max(Q[state])-Q[old_state, action])

    return Q


# Function that calculates and plots cumulative moving average
# Inputs: list containing rewards or timesteps per episode, x and y labels
# Outputs: CMA Plot
def plot_cma(rt_ep_list, ylabel):
    # Program to calculate cumulative moving average
    # Inspired by https://www.geeksforgeeks.org/how-to-calculate-moving-averages-in-python/

    # Convert list to array
    rt_ep_array = np.asarray(rt_ep_list)

    i = 1
    # Initialize an empty list to store cma
    cma = []

    # Store cumulative sums of array in cum_sum array
    cum_sum = np.cumsum(rt_ep_array)

    # Loop through the array elements
    while i <= len(rt_ep_array):
```

```
        # Calculate the cumulative average by dividing
        # cumulative sum by number of elements till
        # that position
        window_average = round(cum_sum[i-1] / i, 2)

        # Store the cumulative average of
        # current window in moving average list
        cma.append(window_average)

        # Shift window to right by one position
        i += 1

    # Plot evaluation metrics
    plt.figure()
    plt.title("Reward - Cumulative Moving Average")
    plt.xlabel("Episodes")
    plt.ylabel(ylabel)
    pcma = plt.plot(cma)


#### Function that calculates rewards moving average ####
# Inputs: list containing rewards, window size and a boolean for plotting
# Outputs: Final Moving Average or Plot
def reward_ma(rt_ep_list, w_size, plot_metric):
    # Program to calculate moving average using numpy
    # Inspired by https://www.geeksforgeeks.org/how-to-calculate-moving-averages-in-python/

    # Convert list to array
    rt_ep_array = np.asarray(rt_ep_list)
    window_size = w_size

    i = 0
    # Initialize an empty list to store moving average
    moving_averages = []

    # Loop through the array
    # consider every window of size w_size
    while i < len(rt_ep_array) - window_size + 1:

        # Calculate the average of current window
        window_average = round(np.sum(rt_ep_array[
            i:i+window_size]) / window_size, 2)

        # Store the average of current
        # window in moving average list
        moving_averages.append(window_average)

        # Shift window to right by one position
        i += 1

    # if flagged to plot the moving average
    if plot_metric:
        # Plot evaluation metrics
        plt.figure()
        plt.title("Rewards Moving Average")
        plt.xlabel("Episodes")
        plt.ylabel("Rewards")
        pma = plt.plot(moving_averages)
    else:
        # return the last reward moving average
        return moving_averages[-1]


#### Function that calculates timestep moving average ####
# Inputs: list containing number of timesteps, window size and a boolean for plotting
# Outputs: Final Moving Average or Plot
```

```
def timestep_ma(ts_ep_list, w_size, plot_metric):
    # Program to calculate moving average using numpy
    # Inspired by https://www.geeksforgeeks.org/how-to-calculate-moving-averages-in-python/

    # Convert list to array
    rt_ep_array = np.asarray(ts_ep_list)
    window_size = w_size

    i = 0
    # Initialize an empty list to store moving average
    moving_averages = []

    # Loop through the array
    # consider every window of size w_size
    while i < len(rt_ep_array) - window_size + 1:

        # Calculate the average of current window
        window_average = round(np.sum(rt_ep_array[
            i:i+window_size]) / window_size, 2)

        # Store the average of current
        # window in moving average list
        moving_averages.append(window_average)

        # Shift window to right by one position
        i += 1

    # if flagged to plot the moving average
    if plot_metric:
        # Plot evaluation metrics
        plt.figure()
        plt.title("Steps Moving Average")
        plt.xlabel("Episodes")
        plt.ylabel("Steps")
        pma = plt.plot(moving_averages)
    else:
        # return the last reward moving average
        return moving_averages[-1]


#### Function used to generate the hyperparameters grid search ####
# Inputs: alpha, gamma, decay rate, epsilon
# Output: grid search on gols hit, number of rewards collected, average steps, hyperparameters
def trainig_grid(a, g, d, epsilon = 0.9):
    S, A, R, Q, state, goal_state, hole_state = init_env()
    max_epsilon = 0.9
    min_epsilon = 0.01
    num_ep = 10000
    num_timestep = 500
    goal = 0
    goals = []
    reward_ep_list = [] # List containing rewards
    ts_ep_list = [] # List containing steps
    eps_decay = []  # List containing epsilons
    goals_print = [] # List containing goals to print
    steps_print = [] # List containing steps to print
    reward_print = [] # List containing rewards to print
    #run num_episodes episodes
    for episode in range(num_ep):

        #print("Starting state is '{}'".format(S[state]))

        # Initialize/Reset reward metric
        r_metric = 0

        goals.append(goal)
```

```
    for timestep in range(num_timestep):
        # Select action
        action = select_action(R,Q,S,A,state,epsilon)

        # Get immediate reward
        r = R[state,action]
        #print("Reward for taking action '{}' from state '{}': {}".format(A[action], S[state], r))

        # Sum the reward
        r_metric += r

        # Update the state - move agent
        old_state = state # Store old state

        state = select_step(state,action) # Get new state
        #print("After taking action '{}' from state '{}', new state is '{}'".format(A[action], S[old_state], S[state]))

        # Update Q-Matrix
        Q = update_qvalue(a,g,Q,state,old_state,r,action)

        if S[state] == goal_state:
            goal += 1
            goals[-1] = goal
            break
        elif S[state] in hole_state:
            break

    # Store metrics to lists
    ts_ep_list.append(timestep) # Number of timesteps
    reward_ep_list.append(r_metric) # Total episode rewards

    # Exploration rate decay
    epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-d*episode)
    eps_decay.append(epsilon) # appends the updated epsilon to a list (used to plot this metric)

    state = S[0] # Start again from the beginning
    # print('Episode {} finished. Q matrix values:\n{}'.format(episode,Q.round(1)))
  # the below if statements are used to adjust the outputs lenghts -> display a better grid
  if episode == num_ep-1:
    m = max(goals)
    goals_print.append(m)
    if len(str(m)) == 1:
        m = "   "+str(m)
    elif len(str(m)) == 2:
        m = "  "+str(m)
    elif len(str(m)) == 3:
        m = " "+str(m)
    s = sum(reward_ep_list)
    reward_print.append(s)
    if len(str(s)) == 8:
        s = " "+str(s)
    elif len(str(s)) == 7:
        s = "  "+str(s)
    t = sum(ts_ep_list)/num_ep
    steps_print.append(t)
    # prints outputs to the grid
    print('Goals hit: {} | Cumulative reward: {} | AVG steps: {:.0f} | Alpha: {} | Gamma: {} | Epsilon_s: {} | Epsilon_f: {:.2} | Decay
Rate: {}'\
    .format(m,
    s,
    t,
    a,
    g,
    max_epsilon,
    epsilon,
    d))
```

```
#### Function used to display the episodes to screen ####
# Inputs: agents' actions
# Output: visual grid with agent moving onto it.
def display_episodes(actions):

    # Creates the frozen lake matrix

    S = [['S', 'F', 'F', 'F', 'F', 'F', 'F', 'F'],
         [ 'F', 'F', 'F', 'H', 'F', 'H', 'F', 'F'],
         [ 'F', 'H', 'F', 'F', 'F', 'F', 'H', 'F'],
         [ 'F', 'F', 'F', 'H', 'F', 'F', 'F', 'F'],
         [ 'F', 'F', 'H', 'H', 'F', 'F', 'F', 'F'],
         [ 'F', 'F', 'F', 'H', 'F', 'H', 'F', 'H'],
         [ 'H', 'F', 'H', 'F', 'F', 'H', 'H', 'H'],
         [ 'F', 'F', 'F', 'F', 'F', 'F', 'F', 'G']]

    # Import and initialize the pygame library
    import pygame
    from pygame.locals import QUIT
    clock = pygame.time.Clock()

    # Set the parameters for the main window
    SCREEN_WIDTH = 1000
    SCREEN_HEIGHT = 1000

    # Set up the drawing window
    screen = pygame.display.set_mode([SCREEN_WIDTH, SCREEN_HEIGHT])

    # Display window name
    pygame.display.set_caption("Frozen Lake")

    # Initialize pygame
    pygame.init()

    # Run until the user asks to quit

    running = True

    while running:
        # Did the user click the window close button?
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

        # Fill the background with white
        screen.fill((255, 255, 255))

        # LOAD THE ENVIRONMENT IMAGES

        hole = pygame.image.load('hole.jpeg')  # load image
        hole = pygame.transform.scale(hole, (SCREEN_WIDTH/8, SCREEN_WIDTH/8)) # rescale the image to be a perfect
square that fits 1/4 of the screen.
        frozen = pygame.image.load('frozen.jpeg')
        frozen = pygame.transform.scale(frozen, (SCREEN_WIDTH/8, SCREEN_WIDTH/8))
        goal = pygame.image.load('goal2.jpeg')
        goal = pygame.transform.scale(goal, (SCREEN_WIDTH/8, SCREEN_WIDTH/8))

        # CREATE ENVIRONMENT (sets the photos into the right slots) FROM THE MATRIX
        def print_screen():
            for r in range(8):
                for c in range(8):
                    x = c * 125 # each slot is 125 x 125 pixels
                    y = r * 125
                    if S[r][c] == 'S':
```

```
                image = frozen
            elif S[r][c] == 'F':
                image = frozen
            elif S[r][c] == 'G':
                image = goal
            else:
                image = hole

            screen.blit(image, (x, y)) #display the images to screen

    # Define a player object by extending pygame.sprite.Sprite
    # The surface drawn on the screen is now an attribute of 'player'
    class Player(pygame.sprite.Sprite):
        def __init__(self):
            super(Player, self).__init__()
            self.surf = pygame.image.load('ninja2.png') # load the image
            self.surf = pygame.transform.scale(self.surf, (SCREEN_WIDTH/12, SCREEN_WIDTH/12)) # rescale the image
            self.rect = self.surf.get_rect() # this gets the images property (position and size)
            self.rect.x = 0 # set player horizontal position more central (by default the images are set with their up left corner to position 0,0)
            self.rect.y = 0 # set player vertical position more central

        # set the player picture to move properly from slot to slot, depending on the action.
        def move(self, actions):
            if actions == 0: # move up
                self.rect.x += 0
                self.rect.y += -125
            elif actions == 1: # move down
                self.rect.x += 0
                self.rect.y += 125
            elif actions == 2: # move left
                self.rect.x += -125
                self.rect.y += 0
            elif actions == 3: # move right
                self.rect.x += 125
                self.rect.y += 0
            elif actions == 5: # restart
                self.rect.x = 0
                self.rect.y = 0


    player = Player()   # initialise a player instance

    # this is a loop to try and see if the players moves okay.
    for i in actions: # for now this takes the actions from the list of actions defined at the top
    # screen.blit(image, (x, y)) #display the images to screen
        player.move(i) # uses the above function to move the player
        print_screen()
        screen.blit(player.surf, (player.rect.x, player.rect.y)) # make the player move
        clock.tick(100)  # this sets the speed of the movements. If you don't set it it finishes all the movements before you even see them.
        pygame.display.update() # this updates the frames on the screen.

    # this stops the game when the player has reached the final goal.
        if i == 6:
            running = False

# Done! Time to quit.
pygame.quit()
```

```
# These functions are used in DRL_Part_2 (Deep Q-Learning)

# Import libraries
# General imports
import numpy as np
import random
from collections import namedtuple
# Neural Networks imports
import torch
import torch.nn as nn
import torch.nn.functional as F
# Visualization imports
import matplotlib.pyplot as plt
# Environment import
import gym
# Seed for reproducibility
torch.manual_seed(10)


#### Define Environment functionality class ####
# Code inspired by https://deeplizard.com/learn/video/jkdXDinWfo8
# A class that loads and handles an OpenAI Gym game
class EnvManager():
    def __init__(self, env_name, device):
        self.env = gym.make(env_name).unwrapped  # load environment
        self.env.reset()  # reset to initial state
        self.done = False  # track if the episode has ended
        self.current_state = None  # storing the current state
        self.device = device  # cpu or gpu

    # reset environment and set the current state
    def reset(self):
        self.current_state = self.env.reset()

    # close the environment
    def close(self):
        self.env.close()

    # render game image
    def render(self, mode='human'):
        return self.env.render(mode)

    # get the number of environment's available actions
    def num_actions_available(self):
        return self.env.action_space.n

    # excecute the given action and get the reward
    def take_action(self, action):
        self.current_state, reward, self.done, _ = self.env.step(action.item())
        return torch.tensor([reward], device=self.device)

    # get the state after excecuting the action
    def get_state(self):
        if self.done:  # if the action ended the episode
            return torch.zeros_like(torch.tensor(self.current_state), device=self.device).float()
        else:
            # return the new state after the action
            return torch.tensor(self.current_state, device=self.device).float()

    # number of features in a state, for the NN
    def n_state_features(self):
        return self.env.observation_space.shape[0]


#### Define our Deep Q Network class ####
```

```
# Code inspired by Lab 06
# Three hidden layers
# takes as input the state features
# outputs q values for each action
class DQN(nn.Module):
    def __init__(self, n_state_features, hidden_size, n_actions, isdueling):
        super().__init__()
        self.isdueling = isdueling  # Boolean for Dueling architecture
        # Input -> First hidden layer
        self.fc1 = nn.Linear(n_state_features, hidden_size)
        # First hidden layer -> Second hidden layer
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        # Second hidden layer -> Third hidden layer
        self.fc3 = nn.Linear(hidden_size, hidden_size)
        # Check if network architecture is Dueling Network
        if self.isdueling:
            # The Value V(s) of being at the state
            self.V = nn.Linear(hidden_size, 1)
            # The Advantage A(s,a) of taking the action at the state
            self.A = nn.Linear(hidden_size, n_actions)
        else:
            # Third hidden layer -> Output
            self.out = nn.Linear(hidden_size, n_actions)

    def forward(self, x):
        # Pass input through layers and activate it
        # using ReLu activation function
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        # Check if network architecture is Dueling Network
        if self.isdueling:
            V = self.V(x)  # The Value V(s)
            A = self.A(x)  # The Advantage A(s,a)
            x = V + (A - A.mean())  # Combine streams
            return x
        else:
            x = self.out(x)
            return x


#### Define an Experience Class for Experience Replay ####
# Code inspired by Lab 06
# Agent's experiences will be saved in a named tuple
# Each experience will have the state, action, the new state and the reward
Experience = namedtuple(
    'Experience', ('state', 'action', 'new_state', 'reward'))


#### Process Tensors ####
# Code inspired by https://deeplizard.com/learn/video/kF2AlpykJGY
# and https://deeplizard.com/learn/video/ewRw996uevN
def extract_tensors(experiences):
    # Convert batch of Experiences to Experience of batches
    # using zip
    batch = Experience(*zip(*experiences))

    t1 = torch.stack(batch.state)
    t2 = torch.cat(batch.action)
    t3 = torch.cat(batch.reward)
    t4 = torch.stack(batch.new_state)

    return (t1, t2, t3, t4)


#### Define Replay Memory Class for storing and sampling experiences ####
```

```python
# Code inspired by Lab 06
# This class will store agent's new experiences
# and sample experiences based on a batch size
class ReplayMemory():
    def __init__(self, capacity):
        self.capacity = capacity  # the capacity of the replay memory
        self.memory = []
        self.position = 0  # counter for position

    # store an experience to the memory
    def store(self, experience):
        if len(self.memory) < self.capacity:  # if there is room available
            self.memory.append(experience)  # store experience
        else:  # if the experience memory is full
            # replace oldest experience
            self.memory[self.position % self.capacity] = experience
        self.position += 1  # increase counter

    # sample a batch of experiences from the memory
    def get(self, batch_size):
        return random.sample(self.memory, batch_size)

    # check if there are enough experiences to sample
    def batch_available(self, batch_size):
        return len(self.memory) >= batch_size


#### Define Epsilon Greedy Class ####
# Gets as input the minimum, maximum and decay rate values of epsilon
# Returns epsilon greedy strategy based on current step
class EpsilonGreedyValue():
    def __init__(self, max_epsilon, min_epsilon, epsilon_decay_rate):
        self.max_epsilon = max_epsilon
        self.min_epsilon = min_epsilon
        self.epsilon_decay_rate = epsilon_decay_rate

    # get epsilon greedy strategy
    def epsilon_strategy(self, current_step):
        return self.min_epsilon + (self.max_epsilon - self.min_epsilon) * \
            np.exp(-1. * current_step * self.epsilon_decay_rate)


#### Define our Agent Class ####
# The agent will choose an action from the list of
# available actions based on the epsilon greedy strategy
class DRLAgent():
    def __init__(self, strategy, n_actions, device):
        self.current_step = 0  # store n of steps
        self.strategy = strategy  # epsilon greedy strategy
        self.n_actions = n_actions  # available actions
        self.device = device  # cpu or gpu

    # agent selects an action using the policy network
    # based on the current state
    def select_action(self, state, policy_net):
        # get epsilon greedy strategy
        rate = self.strategy.epsilon_strategy(self.current_step)
        self.current_step += 1

        # exploration vs exploitation
        if rate > random.random():  # explore
            # select a random action
            action = random.randrange(self.n_actions)
            return torch.tensor([action]).to(self.device)
        else:  # exploit
            with torch.no_grad():
```

```
            # pass the state to the network and get the max q value action
            return policy_net(state).unsqueeze(dim=0).argmax(dim=1).to(self.device)


#### Define Q-Values Calculator class ####
# This class will handle the calculation the Q Values
# Gets states and actions sampled from replay memory
# Policy network will return the current state Q-Values
# Target network will return the next state Q-Values
# Code source https://deeplizard.com/learn/video/ewRw996uevM
class QValues():
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    @staticmethod
    # Use the policy network to get current state q values
    def get_current(policy_net, states, actions):
        return policy_net(states).gather(dim=1, index=actions.unsqueeze(-1))

    @staticmethod
    # Use the target network to get next state q values
    def get_next(target_net, next_states):
        # find if there are final states to the given batch
        # get their location
        final_state_locations = next_states.flatten(start_dim=1) \
            .max(dim=1)[0].eq(0).type(torch.bool)
        # get the location of states that are not final
        non_final_state_locations = (final_state_locations == False)
        # filter the non final states
        non_final_states = next_states[non_final_state_locations]
        # get the batch size
        batch_size = next_states.shape[0]
        # create torch to store the values of the next states
        values = torch.zeros(batch_size).to(QValues.device)
        # pass the states that are not final to the target net
        # get the output and store to values
        values[non_final_state_locations] = target_net(
            non_final_states).max(dim=1)[0].detach()
        return values


#### Function that calculates and plots cumulative moving average ####
# Inputs: list containing rewards or timesteps per episode, x and y labels
# Outputs: CMA Plot
def plot_cma(rt_ep_list, ylabel):
    # Program to calculate cumulative moving average
    # Inspired by https://www.geeksforgeeks.org/how-to-calculate-moving-averages-in-python/

    # Convert list to array
    rt_ep_array = np.asarray(rt_ep_list)

    i = 1
    # Initialize an empty list to store cma
    cma = []

    # Store cumulative sums of array in cum_sum array
    cum_sum = np.cumsum(rt_ep_array)

    # Loop through the array elements
    while i <= len(rt_ep_array):

        # Calculate the cumulative average by dividing
        # cumulative sum by number of elements till
        # that position
        window_average = round(cum_sum[i-1] / i, 2)

        # Store the cumulative average of
```

```
        # current window in moving average list
        cma.append(window_average)

        # Shift window to right by one position
        i += 1

    # Plot evaluation metrics
    plt.figure()
    plt.title("Cumulative Moving Average")
    plt.xlabel("Episodes")
    plt.ylabel(ylabel)
    pcma = plt.plot(cma)


#### Function that calculates rewards moving average ####
# Inputs: list containing rewards, window size and a boolean for plotting
# Outputs: Final Moving Average or Plot
def reward_ma(rt_ep_list, w_size, plot_metric):
    # Program to calculate moving average using numpy
    # Inspired by https://www.geeksforgeeks.org/how-to-calculate-moving-averages-in-python/

    # Convert list to array
    rt_ep_array = np.asarray(rt_ep_list)
    window_size = w_size

    i = 0
    # Initialize an empty list to store moving average
    moving_averages = []

    # Loop through the array
    # consider every window of size w_size
    while i < len(rt_ep_array) - window_size + 1:

        # Calculate the average of current window
        window_average = round(np.sum(rt_ep_array[
            i:i+window_size]) / window_size, 2)

        # Store the average of current
        # window in moving average list
        moving_averages.append(window_average)

        # Shift window to right by one position
        i += 1

    # if flagged to plot the moving average
    if plot_metric:
        # Plot evaluation metrics
        plt.figure()
        plt.title("Moving Average")
        plt.xlabel("Episodes")
        plt.ylabel("Rewards")
        pma = plt.plot(moving_averages)
    else:
        # return the last reward moving average
        return moving_averages[-1]
```

Contribution to the team tasks

The presented project is the result of our shared work and was organised to split the workload equally.
It mostly unfolded in live sessions where we would discuss our ideas, code them on the spot and draft them in the report. The final writing was shared.

High level workload split

1. Define an environment and the problem to be solved
Dimitris: proposed the *Frozen Lake*-like environment.
Alessandro: how this could be displayed.
Both: discussed how to implement the Q-learning algorithm to solve it.

2. Define a state transition function and the reward function
Alessandro: had an idea on how to make the agent move on the grid, and implement the *transition function*.
Dimitris: built the *reward function.*
Both: built the *state function.*

3. Set up the Q-learning parameters (gamma, alpha) and policy
Alessandro: looked at α and γ.
Dimitris: looked at ε-greedy policy and ε decay rate.

4. Run the Q-learning algorithm and represent its performance
Dimitris: ran the Q-learning algorithm and represented its base performance.

5. Repeat the experiment with different parameter values, and policies
Alessandro: did hyperparameters tuning and testing.

6. Analyze the results quantitatively and qualitatively
Both*:* discussed the results and contribute to writing the report.

7. Implement DQN with two improvements, presented during the Lectures/Labs or from the literature. Motivate your choice and your expectations for choosing a particular improvement. Apply it in an environment that justifies the use of Deep Reinforcement Learning.
Dimitris: chose the two improvements.
Alessandro: selected a suitable environment.

8. Analyse the results quantitatively and qualitatively
Dimitris: implemented the models.
Alessandro: did hyperparameters tuning and testing.
Both: discussed the results and contribute to writing the report.

9. Apply the RL algorithm of your choice (from rllib) to one of the Atari Learning Environment. Briefly present the algorithm and justify your choice
Individual

10. Analyse the results quantitatively and qualitatively
Individual