Reinforcement Learning

Kjell Raaijmakers (1244095), Jeroen van Riel (1236068)

June 26, 2022

Introduction

This report considers the problem of using reinforcement learning for training agents to play emulated games. We will first discuss a very simple toy game to explain and compare different learning algorithms. Next, we use the well-known game *Pong* to illustrate why using these reinforcement learning techniques may be hard on more complex games. Finally, we will present our efforts at implementing a reinforement learning agent for the game *Breakout*.

Toy Game (Task 1)

We choose to explore the *Frozen Lake*[1] game, which involves a small rectangular grid world $W = \{F, H, S, G\}^{w \times h}$ containing w times h patches. Most patches are either frozen F or contain a hole H. There are two special patches S and G that indicate the *start* and *goal*, respectively, of an agent that moves in this grid world. Starting in S, the agent can choose to move a single step in one of the directions *up*, *right*, *down* or *left*, so we define the action space as $\mathcal{A} = \{U, R, D, L\}$. Due to the slippery nature of the frozen patches, the agent may not always end up in the chosen direction. The observation available to the agent is the position it actually ended up. When the agent ends up in a hole, the episode ends without any reward. If the agent is able to reach the goal patch G, the episode ends with a unit reward. Any other intermediate actions do not yield any reward.

Agents for Frozen Lake (Task 2 and 3)

Now that we have explained the game we will use the agents on, we will implement three learning agents on the game. First of all, we will look at a learning agent based on Monte Carlo (MC) prediction. For MC, our goal is to find each value for $q_\pi(s, a)$, where $q_\pi(s, a)$ is the expected return when starting in state s and taking action a, and then following policy π. The way the agent does this is by first generating an episode under a policy (in our case this will be either when the agent ends in a hole, or when he reaches the goal), and from there look for each first occurence of being in state s and taking action a, and average the reward to the estimated value we already had of $q_\pi(s, a)$ (for example, if in 5 different episodes, we take action 0 in state 0 and get a reward of 1 in 1 of the episodes (and no rewards in the other episodes), our estimated value of $q_\pi(0, 0) = \frac{\text{Total rewards}}{\text{Total number of runs}} = \frac{1}{5}$). The agent also updates their strategy based on its current estimation of the matrix $Q = (q(s, a)_{s \in S, a \in A})$ (where S is the state space and A is the action space). In particular, the agents strategy of choosing an action will always be ε-greedy, i.e., the agent takes the optimal action based on their earlier observations in $1 - \varepsilon$ cases, and in the remaning ε cases, it takes a random action (so that the agent keeps exploring). The exact form of the algorithm is based on Figure 5.4 of [4]. In Figure 1 we see the succes rate of the Monte-Carlo agent. Here, we decrease the ε over time. As we expect, we see that the succes rate increases, which is normal since we decrease ε, thus decreasing the probability of taking a non-optimal action. However, as we will later see, this is not the best agent we can get.

[1] `https://www.gymlibrary.ml/environments/toy_text/frozen_lake/`

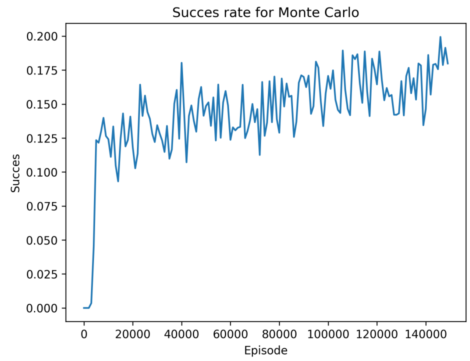
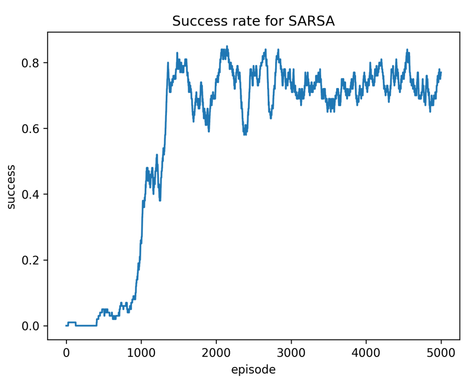


Figure 1: Succes rate of the Monte-Carlo agent

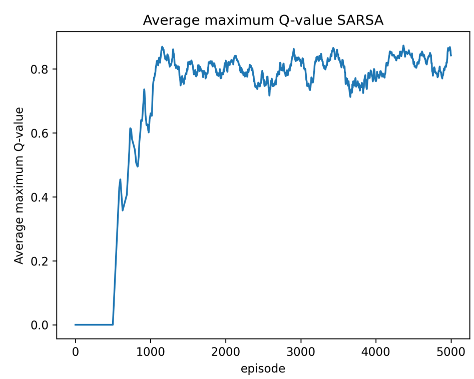
Then we will take a short look at temporal-difference (TD) learning. TD-learining, just like Monte Carlo, is a way to evaluate the values of being in a state and taking an action. However, for Monte Carlo, the agent waits for the whole run to be over, while TD-learning agents only look at the next step. This means that TD-learning agents may updates matrix Q for each step, while Monte Carlo agents update the matrix for multiple steps only once an episode is finished.

Now, TD-learning only evaluates a given strategy π, however, we need something to update this strategy. We look at two different methods: SARSA and Q-learning. Looking at SARSA, after taking a step from state s with action a and observing reward R, we look in which state we would end up (which we denote by s_n) and which action a_n we would take under our current strategy. We use this information, as well as the reward we observe, to update our matrix of Q. The algorithm used for this algorithm can be found in Figure 6.9 of [4].

For SARSA, we see some of the results in Figure 2. First, in Figure 2a, we have the succes rate for the SARSA agent, for decreasing values for ε. Interesting to see is that the rate of succes is much higher than the results of the Monte-Carlo agent. Furthermore, we also looked at the maximum value of the Q matrix, averaged over the last 1000 values. We see that, after about 1000 episodes, this value begins to stabalize, which is logical, since we do not expect the Q-matrix to change as much when a lot of episodes are finished.



(a) Succes rate for the SARSA agent



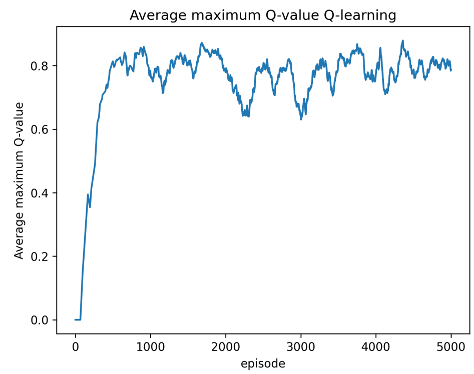
(b) Maximum Q-matrix values for the SARSA agent

Figure 2: Results for the SARSA agent

Finally, we look at Q-learning. Q-learning looks similar to SARSA, apart from the fact that we do not take a new action a_n from the new state s_n, but instead look at the best action an^*_Q we can take under our current matrix Q. The algorithm used for this algorithm can be found in Figure 6.12 of [4].

Then for the results, in Figure 3, we see similar results as for the SARSA agent. This, can be explained by the fact that, when decreasing the value for ε, the Q-learning agent and the SARSA-agent look the same.

(a) Succes rate for the Q-learning agent



(b) Maximum Q-matrix values for the Q-learning agent

Figure 3: Results for the Q-learning agent

Before moving on to some more complex games, one might ask the question: why did the Monte-Carlo agent perform so much worse than the other agents? This might be due to the fact that we might not have done enough simulations, since we would expect that, when the number of episodes increases, the performence also increases.

Pong (Task 4)

Now that we have looked at a smaller game, we take a look at some more complex games. For the first game, which is the game Pong, we will observe the difficulties we can face when dealing with games which have a large amount of observations. After that (from Task 5 and onwards) we will report on the implementation of the learning agents we have looked at in Task 2 for the game of Breakout.

For the game of Pong, we are dealing with a image of size 210×160, with color values which can range from 0 to 255.

This makes that the total observation space of Pong is big; an upperbound to the total number of observations is

$$210 \cdot 160 \cdot 256 = 8601600 \approx 8.6 \cdot 10^6$$

This is only considering the number of observations according to a single frame. However, this might not even be enough.

The agent would not be able to discriminate between situations with different velocity/acceleration from single frames alone. Since the agent has to map each state to an action at every time step, we need to incorporate information about velocity in the state representation. Assume for the moment that we have a method to extract the exact coordinates of the ball from each single frame. Taking a simple numerical approximation of velocity, as well as an approximation of the direction of the ball, we would be able to calculate where exactly we need to be at what time.

Now we can ask ourselves what the most important pieces of informations are, as taking $2.6 \cdot 10^7$ observations into account (without even looking at the speed of the ball), one might ask themselves if all this information is necessary. When looking at a single frame of the game (take for example

Figure 1 of the assignment), the most important pieces of information is the position of the ball and the position of the player. Furthermore, in the frame, a small trail can be seen, which can be used to find the direction of the ball. On top of that, we might not care that much about what happens close to the opponent; a small approximation error close to the opponent might lead to quite a large error along the way anyway. Then, since the color of the ball is constant, instead of checking all colors, we might only care about one color (or to reduce errors, a small array of colors). Finally, we still need something to deal with the speed of the ball, with which we deal after explaining our proposal of the reduced observation space.

Speaking of our reduced observation space, we might consider the following: Divide the opponents half in c_{opponent} even columns and r_{opponent} even rows, and divide the own half in c_{own} even columns and r_{own} even rows (where $c_{\text{opponent}} < c_{\text{own}}$ and $r_{\text{opponent}} < r_{\text{own}}$). Then, select an array of k colors, with the array of colors being close to the color of the ball (and by extend, the color of the own player, which is the same color). Finally, to deal with the velocity of the ball, we can take the last f frames, so that we can estimate the velocity of the ball.

An upperbound for this reduced state space is now

$$(c_{\text{opponent}} \cdot r_{\text{opponent}} + c_{\text{own}} \cdot r_{\text{own}}) \cdot k \cdot f$$

If we want to get a value for this, we can for example take $c_{\text{opponent}} = r_{\text{opponent}} = 4$, $c_{\text{own}} = r_{\text{own}} = 8$, $k = 10$ and $f = 5$, so that the upperbound is

$$(4 \cdot 4 + 8 \cdot 8) \cdot 10 \cdot 5 = 4000 = 4 \cdot 10^3$$

Given that our upperbound for the full state space was of the order 10^6, this reduced state space is a lot smaller.

Playing Atari Games

The Arcade Learning Environment (ALE) [1] provides a collection of game emulators on which reinforcement learning algorithms can be tested. We will consider the *Breakout* [2] game in which the player has to control a paddle at the bottom of the screen in order to keep bouncing a ball towards the top of the screen where rows of bricks are stacked, see Figure 4. When the ball hits a brick, it breaks and the player gets a reward. When the player fails to bounce the ball back up, it disappears out of the game screen and the player loses a *life*. Once all lives are used, the game ends.

We refer to a completed game as an *episode* consisting of a number of discrete *steps* at which the player observes a sreen *frame* and has to select an *action*. Each frame is encoded as a tensor of shape $(210, 160, 3)$, where the last dimension holds the intensity of each RGB color channels. The four available actions are *do nothing* (0), *fire* (1), move the paddle *right* (2) and *left* (3). The fire action starts the game and uses a life to restart after failure. The number of steps per episode lies around 100 when the agent follows a completely random policy. Our goal is to train an agent that achieves a high total reward per episode.

There are multiple approaches we could take to tackle the stated problem. As illustrated in Section , the complexity of the observation space is a main issue in practical reinforcement learning. One could try to manually engineering usefull feature for the agent, such as velocity and acceleration of the ball. A major problem with this approach is that it is does not transfer easily to other problem domains, where the observations are of a comletely different nature. Alternatively, deep learning methods have been proven useful in automatically constructing low-dimensional representation of the observations.

Deep Q-Learning

We will consider the use of a Deep Q-Network (DQN) [3], which is a Q-learning method that uses a Convolutional Neural Net (CNN) as a function approximator. Let $R_t = \sum_i^T \gamma^{i-t} r_i$ be the *discounted*

[2] `https://www.gymlibrary.ml/environments/atari/breakout/`

Figure 4: Original observation frame (without preprocessing) from a game of Breakout. The score indicated at the top left is still zero and there are four lives still available.

return at time t. The idea of Q-learning [5] is to estimate the optimal action-value function, which is defined as the expected discounted reward

$$Q^*(s,a) = \max_\pi \mathbb{E}[R_t | s_t = s, a_t = a, \pi]$$

that the agent receives when taking action a from state s and following policy π afterwards. We know that Q^* satisfies the *Bellman equation*

$$Q^*(s,a) = \mathbb{E}_{s'}[r + \gamma \max_{a'} Q^*(s', a') | s, a],$$

where the expectation is taken over next state s', which is randomly drawn from the environment. It has been shown that iterating over

$$Q(s,a) \leftarrow \mathbb{E}_{s'}[r + \gamma \max_{a'} Q(s', a') | s, a]$$

converges to the optimal action-value function.

It is common to represent the action-value $Q(s, a; \theta) \approx Q^*$ using a function approximator to enable generalization between states. Without this method, the agent has to estimate $Q^*(s, a)$ for each distinct state-action pair, which is very restrictive in the current setting because of the large number of such pairs. Like in the original papers, we use a convolutional neural network (CNN) with three layers and parameters θ, which can be trained by optimizing in each iteration the loss function

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho}[(y_i - Q(s, a; \theta_i))^2],$$

where $y_i = \mathbb{E}_{s'}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a]$ is referred to as the *target* of the iteration. The expectation in the loss function is taken over the *behavioral distribution* ρ, which is due to the *behavioral policy* that is followed during training to collect samples (s, a, r, s') to estimate both expected values. We used a simple ϵ-greedy policy π_ϵ based on the current estimate of the state-action function. Because Q-learning estimates the action-values belonging to the optimal policy π^* while following policy π_ϵ, it is referred to as an *off-policy* method.

Like the original paper, we used a *experience replay* mechanism to smooth the learning. Samples (s, a, r, s') are stored in fixed-length memory, where old samples are removed when the memory is full and a new sample is added. During each iteration, the algorithm samples a random batch from memory and performs a gradient update. This ensures that the behavioral policy does not change too fast, which smoothes the learning and prevents oscillations.

Training

We used PyTorch to implement the model In order to reduce training time, we applied several preprocessing steps. The Gym package for Python provides an easy way to *wrap* existing environments

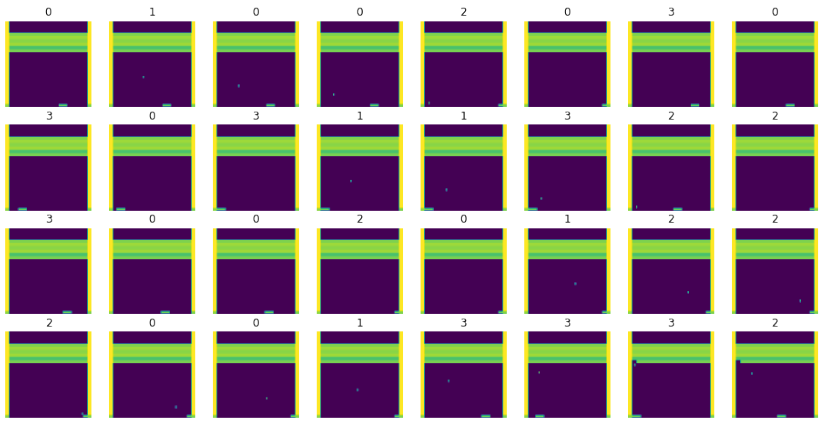


Figure 5: Some game preprocessed observations with $k = 4$ (so 3 frames are skipped). Note that the pixels have only one channel (grayscale), but we used a blue to yellow colorscheme for a clearer picture. The action number is indicated above the observation that resulted from it. As you can see, the game starts after the player has used the *fire* action.

to alter the interaction with an agent [3]. We implemented a wrapper to apply (i) image preprocessing, (ii) frame skipping, (iii) frame stacking and (iv) automatic restarts.

Each screen frame includes a the score and the number of remaining lives, which is not essential for solving the problem at hand. After the top and bottom part of a frame is removed, we rescale it to 84×84 and merge the 3 color channels by taking the mean value, see Figure 5

Like the original paper, we added *frame skipping* to speed up the training process. Upon receiving an action from the agent, our wrapper environment submits the same action to the emulator for $k - 1$ steps, while accumulating rewards. The resulting state and accumulated reward is returned, such that the agent effectively observes every kth step. Another wrapper implements *frame stacking*, which involves providing the last l frames to the agent. This is a common technique to allow the agent to learn about velocity and acceleration. For a detailed explanation of how the authors of the original paper on DQN implemented both steps, we refer to [2].

One important issue that initially prevented the agent from learning anything within reasonable time is the fact that the fire action is required to start the game and to restart after the ball has been missed. Initially we considered the fire action as one of the actions the agent must choose from. Later we decided to automate the (re)starting, to speed up the learning process. The observations from the Gym environment include a number that indicates the number of lives left. Based on this number, the wrapper decides to do the fire action. The DQN agent now only has to learn how to control the paddle through the other three actions.

Evaluation

Initial development and testing of the implementation was done using Google Colab [4], but their free tier restricts use of GPU environments for extended time. Therefore, we activated free $100 Microsoft Azure [5] credit that is in the Github Student Developer Pack[6]. Using this, we were able to train and evaluate our model in Azure Machine Learning Studio [7], which also provides the popular NVIDIA

[3] `https://www.gymlibrary.ml/content/wrappers/`
[4] `https://colab.research.google.com`
[5] `https://azure.microsoft.com`
[6] `https://education.github.com/pack`
[7] `https://azure.microsoft.com/en-us/services/machine-learning/`

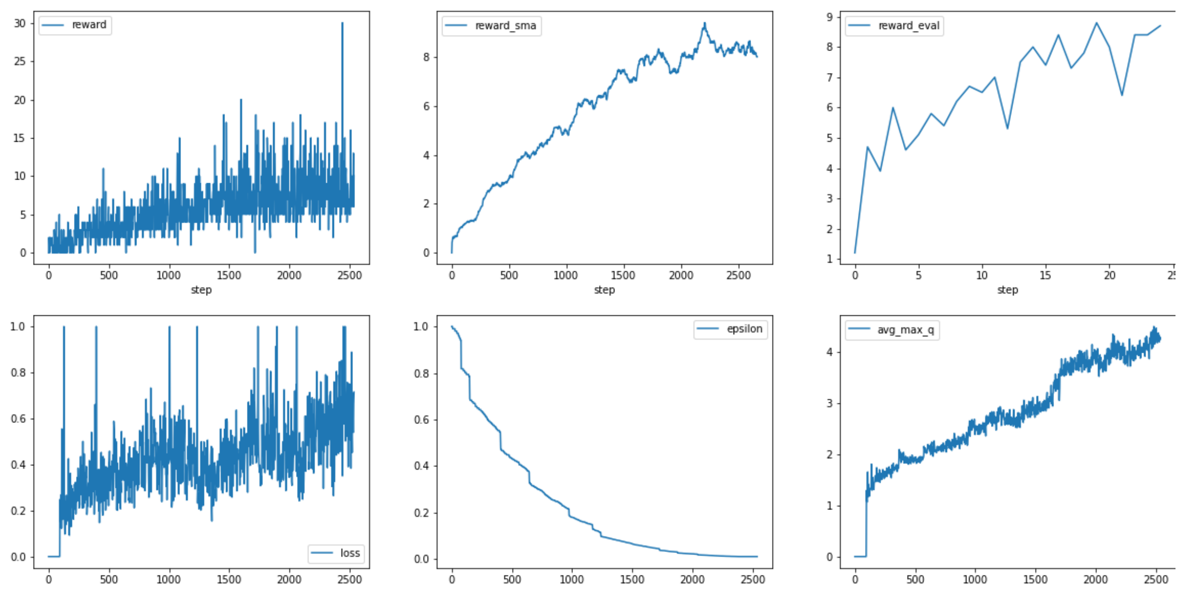


Figure 6: Evaluation of our final implementation.

Tesla K80 [8] that was available in Colab. Our source code and training logs can be found in our online repository [9]. For easy reference, the source code has also been included in Appendix .

We performed various experiments, while changing parameters for the memory size, number of frame skips and learning rate. We did observe oscillating learning behavior with small memory size of 10000, but we encountered also a situation in which a memory size of only 1000 steps clearly showed a growing learning curve. Unfortunately, we had not enough time to develop a systematic approach to have a clear comparison of results, so we showcase one training run here

We trained the model for $T = 2500$ episodes, using $\gamma = 0.95$, learning rate $\alpha = 0.0005$ and memory size of $M = 100000$ steps. We started with totally random exploration $\epsilon = 1$ and decrease after each 1000 steps using $\epsilon = 0.99^{t/1000}$. The training progress is shown in Figure 6. The top row shows various measures of the reward: reward per episode, simple moving average and the third plot shows the periodic evaluationg of a 0.05-greedy policy using the trained Q-values, averaged over 10 episodes. We dot not plot discounted reward, because we are eventually interested in the total accumulated reward. The bottom row shows various other measures that we used to ensure smooth learning. It is important that the loss does not show too large oscillations, which may indicate a too large learning rate. Finally, the last plot in the bottom row shows the average maximum Q values in a minibatch, where the maximum is taken over the actions.

For the purpose of illustration and to help during debugging, we recorded a video of an 0.05-greedy agent after each 100 episodes. We had go through a lot of Gym source code before we found a little bug that prevented us from using the provided `RecordVideo` wrapper. The bug was related to the way some metadata property was named, which prevented the recording wrapper to silently fail.

During training there appeared two episodes that are particularly long. We are sure that this can only be due to the fire action not being provided, but that would mean there is a bug in our wrapper, which we have not yet been able to find. Unfortunately, it does affect the learning, because ϵ is decreased with the number of steps. This explains the large drops in the plot for ϵ in Figure 6.

[8] `https://www.nvidia.com/en-gb/data-center/tesla-k80/`

[9] `https://github.com/jeroenvanriel/decision-theory/tree/master/assignment-3`

Ideas

To account for the sparsity of rewards, we planned to apply some curriculum engineering based on the principle of *optimism in face of uncertainty* by forcing the algorithm to collect more samples with positive reward. Instead of storing all the samples that we encounter is the collection phase, we randomly skip a fraction of samples that have zero reward.

We tried using double Q learning, but lack of computation time prevented us from producing a comparison.

References

[1] M. G. Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279. DOI: 10.1613/jair.3912. URL: https://doi.org/10.1613%2Fjair.3912.

[2] *Frame Skipping and Pre-Processing for Deep Q-Networks on Atari 2600 Games*. https://danieltakeshi.github.io/2016/11/25/frame-skipping-and-preprocessing-for-deep-q-networks-on-atari-2600-games/. Accessed: 2022-06-26.

[3] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. DOI: 10.48550/ARXIV.1312.5602. URL: https://arxiv.org/abs/1312.5602.

[4] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[5] Christopher J.C.H. Watkins and Peter Dayan. "Technical Note: Q-Learning". In: *Machine Learning* 8.3 (May 1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1023/A:1022676722315. URL: https://doi.org/10.1023/A:1022676722315.

Appendices

Tasks

Jeroen van Riel typed Task 1, 5, 6 and 7 (onward from "Playing Atari games") and made the code for these chapters as well, as well as helping with the code for Task 2,3,4. Kjell Raaijmakers typed Task 2, 3, 4, and made the code for Task 2,3,4.

Code Monte-Carlo Frozen Lake

```python
import gym
import numpy as np
import matplotlib.pyplot as plt
import random as rand
from queue import Queue
from random import random, choice

env = gym.make('FrozenLake-v1', desc=["SFFF", "FHFH", "FFFH", "HFFG"])


class EpsilonSoftPolicy:
    def __init__(self, epsilon, env):
        self.epsilon = epsilon
        self.obs_space = range(env.observation_space.n)
        self.action_space = range(env.action_space.n)
        self.actions = {
                state : env.action_space.sample()
                for state in self.obs_space
        }

    def act(self, state):
```

```python
        if random() < self.epsilon:
            # choose random action
            return choice(self.action_space)
        else:
            # choose greedy action
            return self.actions[state]

    def improve(self, state, a_star):
        self.actions[state] = a_star

    def print(self):
        print(self.actions)

    def change_epsilon(self, new_epsilon):
        self.epsilon = new_epsilon


def generate_episode(policy):
    state_actions, rewards = [], []
    t = 0
    state = env.reset()
    while True:
        action = policy.act(state)
        state_actions.append((state, action))

        t += 1
        state, reward, done, info = env.step(action)
        rewards.append(reward)

        if done:
            break

    return state_actions, rewards

class SMA:
    """Simple moving average with incremental update."""
    def __init__(self, k):
        self.q = Queue(k)
        self.k = k
        self.SMA = 0

    def put(self, p):
        if self.q.full():
            self.SMA += (p - self.q.get()) / self.k
        else:
            self.SMA += p / self.k
        self.q.put(p)

    def get(self):
        return self.SMA




# on-policy Monte Carlo control

```

```

gamma = 0.95
sma = SMA(1000)

epsilon_all = [0.1]
nriterations = 150
finalrewards = np.zeros((len(epsilon_all),nriterations))


for x in range(len(epsilon_all)):
    epsilon = epsilon_all[x]
    policy = EpsilonSoftPolicy(epsilon, env)
    n = 0
    Q = np.empty((env.observation_space.n, env.action_space.n))
    N = np.ones((env.observation_space.n, env.action_space.n))
    T=0
    G = 0
    while T<nriterations:
        policy.change_epsilon(max(2*epsilon-(1/75)*epsilon*T,0))
        sa, rewards = generate_episode(policy)
        for t in range(len(sa) - 1, -1, -1):
            G = gamma * G + rewards[t]
            if sa[t] not in sa[0:t]:
                # incremental update of state action value
                Q[sa[t]] = Q[sa[t]] + (G - Q[sa[t]]) / N[sa[t]]
                N[sa[t]] += 1

                # improvement step
                state, action = sa[t]
                a_star = np.argmax(Q[state,:])
                policy.improve(state, a_star)

        sma.put(G)
        if n >= 1000:
            n = 0
            print(x,T)
            finalrewards[x][T] = sma.get()
            T=T+1
        else:
            n += 1


x_plot=range(nriterations)
x_plot = np.multiply(x_plot, 1000)

plt.plot(x_plot,finalrewards[0])
plt.title("Succes rate for Monte Carlo")
plt.xlabel("Episode")
plt.ylabel("Succes")
plt.savefig('Monte_frozen_lake_changing_epsilon_1.png', dpi=300, bbox_inches='
    tight')
plt.show
```

Code SARSA Frozen Lake

```
import gym
import numpy as np
```

```
import matplotlib.pyplot as plt
import math
from queue import Queue
from random import random, choice


class EpsilonSoftPolicy:
    def __init__(self, epsilon, env):
        self.epsilon = epsilon
        self.obs_space = range(env.observation_space.n)
        self.action_space = range(env.action_space.n)
        self.actions = {
                state : env.action_space.sample()
                for state in self.obs_space
        }

    def act(self, state):
        if random() < self.epsilon:
            # choose random action
            return choice(self.action_space)
        else:
            # choose greedy action
            return self.actions[state]

    def improve(self, state, a_star):
        self.actions[state] = a_star

    def print(self):
        print(self.actions)

    def change_epsilon(self, new_epsilon):
        self.epsilon = new_epsilon

class SMA:
    """Simple moving average with incremental update."""
    def __init__(self, k):
        self.q = Queue(k)
        self.k = k
        self.SMA = 0

    def put(self, p):
        if self.q.full():
            self.SMA += (p - self.q.get()) / self.k
        else:
            self.SMA += p / self.k
        self.q.put(p)

    def get(self):
        return self.SMA


env = gym.make('FrozenLake-v1', desc=["SFFF", "FHFH", "FFFH", "HFFG"])

Q = np.zeros((env.observation_space.n, env.action_space.n))
```

```
gamma = 0.95
alpha = 0.5

sma = SMA(1000)

epsilon_all = [0.2]
nriterations = 250
finalrewards = np.zeros((len(epsilon_all),nriterations))

for x in range(len(epsilon_all)):
    epsilon = epsilon_all[x]
    policy = EpsilonSoftPolicy(epsilon, env)
    n = 0
    T=0
    G = 0
    while T<nriterations:
        policy.change_epsilon(max(epsilon-(1/nriterations)*epsilon*T,0))
        s = env.reset()
        while True:
            a = policy.act(s)
            sn, reward, done, info = env.step(a)
            G = gamma * G + reward
            an = policy.act(sn)

            Q[s,a] = Q[s,a] + alpha * (reward + gamma * Q[sn,an] - Q[s,a])

            # update greedy action
            a_star = np.argmax(Q[s,:])
            policy.improve(s, a_star)

            s = sn
            a = an
            if done:
                sma.put(G)
                if n >= 1000:
                    n = 0
                    finalrewards[x][T] = sma.get()
                    print(T)
                    T=T+1
                else:
                    n += 1
                break



x_plot=range(nriterations)
plt.plot(x_plot,finalrewards[0])
plt.title("Succes rate for SARSA")
plt.xlabel("Episode")
plt.ylabel("Succes")
plt.savefig('SARSA_frozen_lake_changing_epsilon.png', dpi=300, bbox_inches='
    tight')
plt.show
```

Q-learning Frozen Lake

```
import gym
```

```python
import numpy as np
import matplotlib.pyplot as plt
from queue import Queue
from random import random, choice


class EpsilonSoftPolicy:
    def __init__(self, epsilon, env):
        self.epsilon = epsilon
        self.obs_space = range(env.observation_space.n)
        self.action_space = range(env.action_space.n)
        self.actions = {
                state : env.action_space.sample()
                for state in self.obs_space
        }

    def act(self, state):
        if random() < self.epsilon:
            # choose random action
            return choice(self.action_space)
        else:
            # choose greedy action
            return self.actions[state]

    def improve(self, state, a_star):
        self.actions[state] = a_star

    def print(self):
        print(self.actions)

class SMA:
    """Simple moving average with incremental update."""
    def __init__(self, k):
        self.q = Queue(k)
        self.k = k
        self.SMA = 0

    def put(self, p):
        if self.q.full():
            self.SMA += (p - self.q.get()) / self.k
        else:
            self.SMA += p / self.k
        self.q.put(p)

    def get(self):
        return self.SMA


env = gym.make('FrozenLake-v1', desc=["SFFF", "FHFH", "FFFH", "HFFG"])

Q = np.zeros((env.observation_space.n, env.action_space.n))
gamma = 0.9
alpha = 0.5
episodes = 1000
T = 10000 # maximum number of steps in one episode

```

```
sma = SMA(100)
episode_sma = np.zeros((episodes))

def update_epsilon(epsilon):
    #return min(epsilon, max(0, 2 * epsilon - (1 / 50) * epsilon * episodes))
    return epsilon - 0.001

policy = EpsilonSoftPolicy(1, env)
for episode in range(episodes):
    total_reward = 0
    s = env.reset()
    for t in range(T):
        a = policy.act(s)
        sn, reward, done, info = env.step(a)
        total_reward += reward

        Q[s,a] = Q[s,a] + alpha * (reward + gamma * np.max(Q[sn,:]) - Q[s,a])

        # update greedy action
        a_star = np.argmax(Q[s,:])
        policy.improve(s, a_star)

        s = sn

        if done:
            sma.put(total_reward)
            episode_sma[episode] = sma.get()
            policy.epsilon = update_epsilon(policy.epsilon)
            print(f'epsilon: {policy.epsilon}, episode: {episode}, sma: {sma.get()}, steps: {t}')
            break

x_plot = range(episodes)
plt.plot(x_plot, episode_sma)
plt.title('Success rate for Q-learning')
plt.xlabel('episode')
plt.ylabel('success')
plt.savefig('q-frozen-lake.png', dpi=300, bbox_inches='tight')
plt.show
```

Atari Breakout

train.py

```
import gym, torch, cv2, random
import numpy as np
from collections import deque
from datetime import datetime
from gym.wrappers import TransformObservation, FrameStack, RecordVideo

from gym import logger
logger.set_level(logger.INFO)

# for logging in azure machine learning studio
from azureml.core import Run
run = Run.get_context()

```

```
DEV = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'using device {DEV}')

def get_time():
    now = datetime.now()
    return now.strftime("%H:%M:%S")

def preprocess(frame):
    frame = frame[34:-16, :, :] # crop
    frame = cv2.resize(frame, (84, 84))
    frame = frame.mean(-1) # to grayscale
    frame = frame.astype('float') / 255 # scale pixels
    return frame

class FrameSkip(gym.Wrapper):
    def __init__(self, env, frames):
        super().__init__(env)
        self.env = env
        self.frames = frames
        self.lives = 5

    def step(self, action):
        action = {0:0, 1:2, 2:3}[action] # map to NOPE, LEFT, RIGHT
        reward = 0
        for _ in range(self.frames):
            s, r, done, info = self.env.step(action)
            lives_next = info['lives']
            reward += r
            if done:
                break

        if self.lives != lives_next and not done:
            s, r, done, info = self.env.step(1) # FIRE to restart

        return s, reward, done, info

    def reset(self):
        super().reset()
        s, _, _, _ = self.env.step(1) # FIRE to start
        return s

def wrap(env):
    env = TransformObservation(env, preprocess)
    env = FrameSkip(env, 2)
    env = FrameStack(env, num_stack=4)
    return env

env = wrap(gym.make('ALE/Breakout-v5'))

# maximum number of emulation steps
T_MAX = 100000

# evaluation is done in a different environment, with recording enabled
EVAL_INTERVAL = 100
EVAL_EPISODES = 10
env_eval = gym.make('ALE/Breakout-v5')
```

```
env_eval.metadata['render.modes'] = env_eval.metadata.get('render_modes', []) # fix a gym bug
vid = RecordVideo(env_eval, 'outputs/', episode_trigger=lambda: True)
env_eval = wrap(vid)

def evaluate(agent, episode):
    epsilon = agent.epsilon
    agent.epsilon = 0.05

    rewards = []
    for t in range(EVAL_EPISODES):
        if t == 0: # big hack to only record first with the episode number
            vid.episode_id = episode
            vid.episode_trigger = lambda x: True
        else:
            vid.episode_trigger = lambda x: False
        s = env_eval.reset()
        reward = 0
        for _ in range(T_MAX):
            a = agent.get_action(s)
            s, r, done, _ = env_eval.step(a)
            reward += r
            if done:
                break
        rewards.append(reward)

    agent.epsilon = epsilon # reset back
    return np.mean(rewards)

if __name__ == "__main__":
    from agent import Agent
    EPISODES = 10000
    MEMORY_SIZE = 100000 # steps
    MIN_MEMORY = 10000 # steps
    bs = 64
    lr = 5e-5
    gamma = 0.95
    agent = Agent(env, lr, gamma, MEMORY_SIZE, MIN_MEMORY, bs, DEV)

    reward_sma = deque(maxlen=100) # simple moving average

    total_step = 0 # number of steps over all episodes
    for episode in range(EPISODES):
        s = env.reset()

        total_reward = 0
        total_loss = 0
        total_max_q = 0
        for t in range(T_MAX): # max number of actual emulation steps
            a = agent.get_action(s)
            s_next, r, done, info = env.step(a)
            agent.store(s, a, r, s_next, done)
            s = s_next

            loss, max_q = agent.train()

```

```
            total_loss += loss
            total_max_q += max_q
            total_reward += r

            total_step += 1
            if total_step % 1000 == 0:
                agent.epsilon = max(0.01, agent.epsilon * 0.99)

            if done: # end of episode
                agent.update_target_dqn()

                # periodic evaluation and model saving
                if total_step > MIN_MEMORY and episode % EVAL_INTERVAL == 0:
                    torch.save(agent.online_dqn, f'outputs/online-{episode}.pt')
                    # torch.save(agent.target_dqn, f'outputs/target-{episode}.pt')
                    print('model saved!')

                    reward_eval = evaluate(agent, episode)
                    print(f'reward_eval: {reward_eval}')
                    run.log('reward_eval', reward_eval)

                # logging
                reward_sma.append(total_reward)
                avg_max_q = total_max_q / t
                run.log('reward', total_reward)
                run.log('reward_sma', np.mean(reward_sma))
                run.log('loss', total_loss)
                run.log('avg_max_q', avg_max_q)
                run.log('epsilon', agent.epsilon)
                print(f'[{get_time()}] episode: {episode} reward: {int(total_reward)} reward_sma: {np.mean(reward_sma):.3f} loss: {total_loss:.3f} avg_max_q: {avg_max_q:.3f} epsilon: {agent.epsilon:.2f} last_step: {t} total_step: {total_step}')
                break
```

agent.py

```
import torch
import random
import numpy as np
from torch.optim import Adam
from torch.nn.functional import mse_loss, smooth_l1_loss
from collections import deque

from model import DQN, DQN1, DQN3

def q_for_action(qs, actions):
    """Get the q values from (64, 3) corresponding to the action."""
    return qs.gather(1, actions.unsqueeze(1)).squeeze(1)

class Agent:
    def __init__(self, env, lr, gamma, memory_size, min_memory, batch_size, device):
        self.epsilon = 1

```

```
        self.dev = device
        self.online_dqn = DQN(3).to(device).train()
        self.target_dqn = DQN(3).to(device)
        self.target_dqn.load_state_dict(self.online_dqn.state_dict())
        self.target_dqn.eval()

        self.gamma = gamma

        self.memory = deque(maxlen=memory_size)
        self.batch_size = batch_size
        self.min_memory = min_memory

        self.optimizer = Adam(self.online_dqn.parameters(), lr=lr)

        self.opt_exp_count = 0
        self.just_exp_count = 0

    def store(self, s, a, r, s_next, done):
        self.memory.append([s, a, r, s_next, done])

        # optimistic experience
        # if len(self.memory) > self.min_memory:
        #     self.memory.append([s, a, r, s_next, done])
        #     print('just store')
        #     self.just_exp_count += 1
        # else:
        #     if r > 0:
        #         self.memory.append([s, a, r, s_next, done])
        #         self.opt_exp_count += 1
        #     elif random.random() < 0.5:
        #         self.memory.append([s, a, r, s_next, done])
        #         self.just_exp_count += 1

    def get_action(self, state):
        if random.random() < self.epsilon:
            # choose random action
            return random.choice(range(3))
        else:
            # return greedy action
            x = torch.unsqueeze(torch.from_numpy(np.asarray(state)), dim=0).
float().to(self.dev)
            return torch.argmax(self.online_dqn(x)).item()

    def double_q_loss(self, s_b, a_b, r_b, s_next_b, done_b):
        # get q values for current states
        q_vals = self.online_dqn(s_b)
        # actual q values for selected actions
        actual_q_vals = q_for_action(q_vals, a_b)

        # target q values
        # TODO: maybe no_grad() is required here!
        next_q_vals = self.online_dqn(s_next_b) # shape: N x 4
        next_actions = next_q_vals.max(1)[1]
        next_target_q_vals = q_for_action(self.target_dqn(s_next_b),
next_actions)

```

```
        # double Q learning target
        target = r_b + self.gamma * next_target_q_vals * (1 - done_b)

        return mse_loss(actual_q_vals, target.detach()), q_vals

    def td_loss(self, s_b, a_b, r_b, s_next_b, done_b):
        q_vals = self.online_dqn(s_b)
        # select q values corresponding to actions
        q_vals = torch.squeeze(torch.take_along_dim(q_vals, a_b.unsqueeze(1),
    1))

        with torch.no_grad():
            qnext = self.online_dqn(s_next_b)
            m, _ = torch.max(qnext, dim=-1)
            target = r_b + self.gamma * m * (1 - done_b)

        # return mse_loss(q_vals, target), q_vals
        return smooth_l1_loss(q_vals, target), q_vals

    def train(self):
        if len(self.memory) < self.min_memory:
            return 0, 0

        # sample minibatch
        s_b, a_b, r_b, s_next_b, done_b = zip(*random.sample(self.memory, self
    .batch_size))

        s_b = torch.from_numpy(np.asarray(s_b)).float().to(self.dev)
        a_b = torch.tensor(a_b, dtype=torch.long, device=self.dev)
        r_b = torch.tensor(r_b, dtype=torch.float, device=self.dev)
        s_next_b = torch.from_numpy(np.asarray(s_next_b)).float().to(self.dev)
        done_b = torch.tensor(done_b, dtype=torch.float, device=self.dev)

        self.optimizer.zero_grad()
        loss, q_vals = self.td_loss(s_b, a_b, r_b, s_next_b, done_b)
        loss.backward()
        self.optimizer.step()

        return loss.item(), q_vals.max().item()

    def update_target_dqn(self):
        self.target_dqn.load_state_dict(self.online_dqn.state_dict())
        self.target_dqn.eval()
```

model.py

```
import torch.nn as nn

class DQN(nn.Module):
    def __init__(self, n_actions):
        super().__init__()
        self.n_actions = n_actions
        self.model = nn.Sequential(
            # 4 input frames
            nn.Conv2d(4, 32, 8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, 4, stride=2),
```

```
                nn.ReLU(),
                nn.Conv2d(64, 64, 3, stride=1),
                nn.ReLU(),
                nn.Conv2d(64, 1024, 7, stride=1),
                nn.ReLU(),
                nn.Flatten(),
                nn.Linear(1024, n_actions),
            )

    def forward(self, x):
        return self.model(x)


class DQN1(nn.Module):
    def __init__(self, n_actions):
        super().__init__()
        self.n_actions = n_actions
        self.model = nn.Sequential(
            # 4 input frames
            nn.Conv2d(4, 32, 8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, 4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, 3, stride=1),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(7 * 7 * 64, 512),
            nn.ReLU(),
            nn.Linear(512, n_actions),
        )
        # for i in [0, 2, 4, 7, 9]:
        #     nn.init.xavier_uniform(self.model[i].weight)

    def forward(self, x):
        return self.model(x)

class DQN2(nn.Module):
    def __init__(self, n_actions):
        super().__init__()
        self.n_actions = n_actions
        self.model = nn.Sequential(
            # 4 input frames
            nn.Conv2d(4, 32, 8, stride=4),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.Conv2d(32, 64, 4, stride=2),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, 64, 3, stride=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Flatten(),
            nn.LazyLinear(256),
            nn.ReLU(),
            nn.Linear(256, n_actions),
        )
```

```python

    def forward(self, x):
        return self.model(x)

class DQN3(nn.Module):
    def __init__(self, n_actions):
        super().__init__()
        self.n_actions = n_actions
        self.model = nn.Sequential(
            # 4 input frames (84, 84)
            nn.Conv2d(4, 32, 8),
            nn.MaxPool2d(2, stride=2),
            nn.ReLU(),
            nn.Conv2d(32, 64, 4),
            nn.MaxPool2d(2, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, 3),
            nn.MaxPool2d(2, stride=2),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(7 * 7 * 64, 512),
            nn.ReLU(),
            nn.Linear(512, n_actions),
        )

    def forward(self, x):
        return self.model(x)
```