AARHUS
UNIVERSITET

Reinforcement Learning Handin

Marc Timmer Jensen
AU651548 / 201907421
Aarhus University
March 2025

Contents

1 Theoretical part

1.1 Discrete version of Gronwall's inequality

Prove the following statement: If the sequences $x_0, x_1, \ldots \geq 0$, $\alpha_0, \alpha_1, \ldots > 0$ and the scalars $C, L \geq 0$ are such that

$$x_n \leq C + L \sum_{m=0}^{n-1} \alpha_m x_m, \quad \forall n \geq 0, \tag{1.1}$$

then

$$x_n \leq C \exp\left(L \sum_{m=0}^{n-1} \alpha_m \right), \quad \forall n \geq 0 \tag{1.2}$$

To prove this a few helpful equations are needed [1]:

$$1 + \sum_{m=0}^{n-1} \left(\prod_{k=0}^{m-1} (1 + \alpha_k \cdot L) \right) \cdot \alpha_m \cdot L \leq \prod_{m=0}^{n-1} (1 + \alpha_m \cdot L) \tag{1.3}$$

$$x_n \leq C \cdot \prod_{m=0}^{n-1} (1 + \alpha_m \cdot L) \tag{1.4}$$

1.1.1 Proof of equation (1.3):

Induction is used for this proof. Lets set n = 1 and insert into equation (1.3):

$$1 + \alpha_0 \cdot L \leq 1 + \alpha_0 \cdot L$$

Lets also try and set n = 2 into (1.3):

$$1 + \sum_{m=0}^{1} \left(\prod_{k=0}^{m-1} (1 + \alpha_k \cdot L) \right) = 1 + \alpha_0 \cdot L + (1 + \alpha_0 \cdot L) \cdot \alpha_1 \cdot L = 1 + \alpha_0 \cdot L + \alpha_1 \cdot L + \alpha_0 \cdot \alpha_1 \cdot L^2$$

$$\leq$$

$$\prod_{m=0}^{1} (1 + \alpha_m \cdot L) = (1 + \alpha_0 \cdot L) \cdot (1 + \alpha_1 \cdot L) = 1 + \alpha_0 \cdot L + \alpha_1 \cdot L + \alpha_0 \cdot \alpha_1 \cdot L^2$$

Since both equations are true we assume that equation (1.3) holds for n = (0, ... n). Now we want to investigate n + 1:

$$1 + \sum_{m=0}^{n} \left(\prod_{k=0}^{m-1} (1 + \alpha_k \cdot L) \right) \cdot \alpha_m \cdot L \leq \prod_{m=0}^{n} (1 + \alpha_m \cdot L)$$

We can divide by $\prod_{m=0}^{n-1}(1 + \alpha_m \cdot L)$ on both sides. We can safely do this since $\alpha_0, \alpha_1, \ldots > 0$ and $L \geq 0$:

$$\frac{1+\sum_{m=0}^{n}(\prod_{k=0}^{m-1}(1+\alpha_k\cdot L))\cdot\alpha_m\cdot L}{\prod_{m=0}^{n-1}(1+\alpha_m\cdot L)} \leq 1+\alpha_n\cdot L$$

Now we extract the last element from the sum on the left hand side:

$$\frac{1+\sum_{m=0}^{n-1}(\prod_{k=0}^{m-1}(1+\alpha_k\cdot L))\cdot\alpha_m\cdot L}{\prod_{m=0}^{n-1}(1+\alpha_m\cdot L)} + \frac{\prod_{k=0}^{n-1}(1+\alpha_k\cdot L))\cdot\alpha_n\cdot L}{\prod_{m=0}^{n-1}(1+\alpha_m\cdot L)} \leq 1+\alpha_n\cdot L$$

By the induction assumption (that the equation holds for n = (1, ..., n)) we have that:

$$\frac{1+\sum_{m=0}^{n-1}(\prod_{k=0}^{m-1}(1+\alpha_k\cdot L))\cdot\alpha_m\cdot L}{\prod_{m=0}^{n-1}(1+\alpha_m\cdot L)} \leq 1$$

With this it becomes clear that:

$$\frac{1+\sum_{m=0}^{n-1}(\prod_{k=0}^{m-1}(1+\alpha_k\cdot L))\cdot\alpha_m\cdot L}{\prod_{m=0}^{n-1}(1+\alpha_m\cdot L)} + \alpha_n\cdot L \leq 1+\alpha_n\cdot L$$

Therefore we can conclude that this equation (1.3) holds true for n = n + 1. By induction we have proven (1.3).

1.1.2 Proof of equation (1.4):

This can also be done by induction. We can try to insert n = 0, 1 and 2 into (1.1) and (1.4):

Table 1.1: Comparing results from (1.1) and (1.4)

n	(1.1) : $x_n \leq C + L\sum_{m=0}^{n-1}\alpha_m x_m$	(1.4) : $x_n \leq C\cdot\prod_{m=0}^{n-1}(1+\alpha_m\cdot L)$
0	$x_0 \leq C$	$x_0 \leq C$
1	$x_1 \leq C + L\cdot\alpha_0\cdot x_0 \leq C + L\cdot\alpha_0\cdot C$	$x_1 \leq C\cdot(1+L\cdot\alpha_0) = C + L\cdot\alpha_0\cdot C$
2	$x_2 \leq C + L\cdot\alpha_0\cdot x_0 + L\cdot\alpha_1\cdot x_1$ $\leq C + L\cdot\alpha_0\cdot C + L\cdot\alpha_1\cdot(C + L\cdot\alpha_0\cdot C)$ $= C + L\,\alpha_0\,C + L\,\alpha_1\,C + L^2\,\alpha_0\,\alpha_1\,C$	$x_2 \leq C\cdot(1+L\cdot\alpha_0)\cdot(1+L\cdot\alpha_1)$ $= C + L\,\alpha_0\,C + L\,\alpha_1\,C + L^2\,\alpha_0\,\alpha_1\,C$

The results displayed in table 1.1 shows that equation (1.4) hold true for n=0, 1 and 2. Now assume that equation (1.4) holds true for n. For the next step in the induction we set n = n + 1 and insert this into (1.1):

$$\begin{aligned}
x_{n+1} &\leq C + L\sum_{m=0}^{n}\alpha_m x_m \\
&\leq C + L\sum_{m=0}^{n}\alpha_m\cdot C\cdot\prod_{k=0}^{m-1}(1+\alpha_k\cdot L) \\
&\leq C\cdot(1+\sum_{m=0}^{n}(\prod_{k=0}^{m-1}(1+\alpha_k\cdot L))\cdot\alpha_m\cdot L) \\
&\leq C\cdot\prod_{m=0}^{n}(1+\alpha_m\cdot L)
\end{aligned}$$

The first line is the equation given in (1.1). In the next line we use (1.4). This can be done because of our induction assumption. In line 3 we do some rewriting. In the last line we use equation (1.3). This therefore concludes the proof for n + 1. Therefore it can be concluded that we proved equation (1.4) by induction.

1.1.3 Proof of equation (1.2):

Using some of the steps from the proof of 1.1.2 it can be found that:

$$\begin{aligned} x_n &\leq C + L \sum_{m=0}^{n-1} \alpha_m x_m \\ &\leq C \cdot \prod_{m=0}^{n-1} (1 + \alpha_m \cdot L) \\ &\leq C \cdot \prod_{m=0}^{n-1} (exp(\alpha_m \cdot L)) \\ &\leq C \cdot exp(L \cdot \sum_{m=0}^{n-1} (\alpha_m)) \end{aligned}$$

From line number 2 to 3 we use that $1 + x \leq exp(x)$. With this it can be concluded that:

$$x_n \leq C \cdot exp(L \cdot \sum_{m=0}^{n-1} (\alpha_m)), \quad \forall n \geq 0$$

Under the conditions given in 1.1.

1.2 Strong LLN

Consider i.i.d. centered random variables $\xi_1, \ldots, \xi_n$. Sketch how the classical statement

$$X_n := \frac{1}{n}\sum_{i=1}^{n} \xi_i \to 0 \quad \text{a.s. as } n \to \infty$$

can be deduced from the result on the convergence of general random dynamical systems stated in Theorem 11.2 of the lecture notes. Which assumptions on (ξ_i) are required?

We use equation (1.1) that states [2, page 58]:

$$X_{n+1} = X_n + \alpha_n \cdot (h(X_n) + M_{n+1})$$

Rewriting this we get:

$$\begin{aligned} X_{n+1} - X_n &= \alpha_n \cdot (h(X_n) + M_{n+1}) \\ &\Updownarrow \\ \frac{1}{n+1}\sum_{i=1}^{n+1} \xi_i - \frac{1}{n}\sum_{i=1}^{n} \xi_i &= \alpha_n \cdot (h(X_n) + M_{n+1}) \\ &\Updownarrow \\ (\frac{1}{n+1} - \frac{1}{n})\sum_{i=1}^{n} \xi_i + \frac{1}{n+1}\xi_{n+1} &= \alpha_n \cdot (h(X_n) + M_{n+1}) \end{aligned}$$

Investigating this equation, we can try and solve for a_n in:

$$a_n \cdot (\frac{1}{n+1} - \frac{1}{n}) = \frac{1}{n}$$

It can be seen that: $(\frac{1}{n+1} - \frac{1}{n}) = -\frac{1}{n^2+n}$ therefore we get that:

$$a_n = -\frac{n^2+n}{n} = -(n+1)$$

With this we can look into this equation:

$$\begin{aligned} \frac{-(n+1)}{-(n+1)} \cdot ((\frac{1}{n+1} - \frac{1}{n})\sum_{i=1}^{n} \xi_i + \frac{1}{n+1}\xi_{n+1}) &= \alpha_n \cdot (h(X_n) + M_{n+1}) \\ &\Updownarrow \\ \frac{1}{-(n+1)} \cdot (X_n - \xi_{n+1}) &= \alpha_n \cdot (h(X_n) + M_{n+1}) \\ &\Updownarrow \\ (-X_n + \xi_{n+1}) &= (n+1) \cdot \alpha_n \cdot (h(X_n) + M_{n+1}) \end{aligned}$$

From this it can be seen that: $\alpha_n = \frac{1}{1+n}$, $h(x) = -x$ and that $M_{n+1} = \xi_{n+1}$.
We know that h is a Lipschitz function since [3]:

$$|x_2 - x_1| \leq L \cdot |x_1 - x_2|$$

Here we can set L as a real constant of minimum 1. The step size: $\alpha_n = \frac{1}{1+n}$ does also fulfill the following properties:

$$\sum_{n\in\mathbb{N}} (\alpha_n) = \infty, \quad \sum_{n\in\mathbb{N}} (\alpha_n^2) < \infty$$

See the proof of this in the: Q-learning section. Here we use this step size to make the Q-learning algorithm converge.

To prove the convergence we need to assume that there exists a k>0 such that for all n:

$$E(|M_{n+1}|^2|F_n) \leq k \cdot (1 + |X_n|) \quad \text{a.s.}$$

This assumption directly has an impact on the (ξ_i) since $M_{n+1} = \xi_{n+1}$. Additionally we also need that:

$$\sup_n(|X_n|) < \infty \quad \text{a.s.}$$

This assumption is not necessarily straight forward. Especially showing the 'stability' is not a trivial task [4, page 428].

From our setup we know that $h(x) = -x$. Therefore $h(x^*) = 0$ when $x^* = 0$. From definition 11.1 [2, page 58] we know that this is an asymptotically stable equilibrium point if, for all solutions $x(t)$, $t \geq 0$ to

$$\dot{x}(t) = h(x(t)), \quad x(0) = x_0$$

holds true for

$$\forall \varepsilon > 0 \; \exists T > 0 \; \forall t \geq T : \quad |x_0 - x^*| < \frac{1}{\varepsilon} \quad \Longrightarrow \quad |x(t) - x^*| < \varepsilon$$

In this setup the solutions to $x(t)$ are given by the equation:

$$x(t) = x_0 \cdot \exp(-t) \; \forall t$$

With this it must be true that: $|x(t)| \to 0$ as $t \to \infty$. This specifically mean that:

$$\forall \varepsilon > 0 \; \exists T > 0 \; \forall t \geq T : \quad |x_0| < \frac{1}{\varepsilon} \quad \Longrightarrow \quad |x(t)| < \varepsilon$$

This shows that $x^* = 0$ is an asymptotically stable equilibrium point [5, slide 6].

Using these five conditions we know from Theorem 11.2 [2, page 59] that:

$$X_n = \frac{1}{n} \sum_{i=1}^{n} \xi_i \to 0 \quad \text{a.s. as } n \to \infty$$

1.3 Proof of Lemma 11.3

Recall our general set-up for studying the Q learning algorithm as introduced in the lecture notes. Letting $t_0 := 0, t_n := \sum_{m \leq n-1} \alpha_m$, we consider the linear interpolation scheme

$$\bar{X}(t) := X_n + (X_{n+1} - X_n) \frac{t - t_n}{t_{n+1} - t_n}, \quad t \in [t_n, t_{n+1}) .$$

Furthermore, denote by x^s solutions of the deterministic dynamical systems starting at s, that is,

$$\dot{x}^s(t) = h\left(x^s(t)\right), \quad t \geq s, \qquad \text{with } x^s(s) = \bar{X}(s).$$

1.3.1 (a) Continuous version of Gronwall's inequality

Use the continuous version of Gronwall's inequality (Lemma 11.5 in the lecture notes) for proving that, for $s \leq t \leq s + T$,

$$\|h\left(x^s(t)\right)\| \leq \|h(0)\| + L\left(C_0 + \|h(0)\| T\right) \mathrm{e}^{LT} =: C_T. \tag{1.5}$$

Firstly use the triangle inequality and that h is Lipschitz:

$$\begin{aligned} ||h(x^s(t))|| &\leq ||h(0)|| + ||h(x^s(t) - h(0)|| \\ &\leq ||h(0)|| + L \cdot ||x^s(t) - 0|| \end{aligned}$$

Combining this with the following equation is very useful in this proof.

$$||x^s(t)|| \leq C_0 + ||h(0)|| \cdot T + L \cdot \int_s^t ||x^s(u)|| \, \mathrm{d}u \tag{1.6}$$

Proof of (1.6)

By the triangle inequality it can be seen that:

$$||x^s(t)|| \leq ||x^s(t) - x^s(s)|| + ||x^s(s)||$$

With this integral property:

$$F(b) - F(a) = \int_a^b f(x) dx$$

And the fact that we know: $\dot{x}^s(t) = h(x^s(t))$. This can be used to rewrite:

$$\begin{aligned} ||x^s(t)|| &\leq ||x^s(t) - x^s(s)|| + ||x^s(s)|| \\ &\leq || \int_s^t h(x^s(u)) \, \mathrm{d}u|| + ||x^s(s)|| \\ &\leq \int_s^t ||h(x^s(u))|| \, \mathrm{d}u + ||x^s(s)|| \\ &\leq \int_s^t ||h(x^s(u)) - h(0)|| \, \mathrm{d}u + \int_s^t ||h(0)|| \, \mathrm{d}u + ||x^s(s)|| \end{aligned}$$

Using that h is Lipschitz we can find that:

$$\int_s^t ||h(x^s(u)) - h(0)|| \, \mathrm{d}u \leq L \cdot \int_s^t ||x^s(u)|| \, \mathrm{d}u$$

We know that the maximum length of the integral is T, since $s \leq t \leq s+T$. This can be used to find that:

$$\int_s^t ||h(0)||\,\mathrm{d}u = ||h(0)|| \cdot \int_s^t 1\,\mathrm{d}u \leq T \cdot ||h(0)||$$

We can show that $||x^s(s)|| \leq C_0 = \sup_n(||x_n||)$, and we know that:

$$||x^s(s)|| = ||\bar{X}(s)|| = ||X_n + (X_{n+1} - X_n)\,\frac{s - t_n}{t_{n+1} - t_n}||, \quad s \in [t_n, t_{n+1})$$

For this equation it can happen that: $(X_{n+1} - X_n) > 0$. Trying to maximize the equation for $||\bar{X}(s)||$ yields that we should set s close to t_{n+1}. In the supremum we see that:

$$\sup_n(||X_n + (X_{n+1} - X_n)\,\frac{t_{n+1} - t_n}{t_{n+1} - t_n}||) = \sup_n(||X_{n+1}||) = \sup_n(||X_n||)$$

In the other case where $(X_{n+1} - X_n) \leq 0$ we try and minimize setting s to t_n. This result in:

$$\sup_n(||X_n + (X_{n+1} - X_n)\,\frac{t_n - t_n}{t_{n+1} - t_n}||) = \sup_n(||X_n||)$$

Using this we know that:

$$||x^s(s)|| \leq \sup_n(||X_n||) = C_0$$

Combining everything results in the inequality:

$$||x^s(t)|| \leq C_0 + ||h(0)|| \cdot T + L \cdot \int_s^t ||x^s(u)||du$$

Which conclude the proof of (1.6)

Proof of (1.5)
Gronwall's inequality states that [2, page 60] for continuous $u, v : [0, T] \to [0, \infty)$ and scalars $C, K, T \geq 0$,

$$u(t) \leq C + K \int_0^t u(s)v(s)\mathrm{d}s \quad \forall t \in [0, T]$$

implies

$$u(t) \leq C \exp\left(K \int_0^T v(s)\mathrm{d}s\right), \quad t \in [0, T].$$

Relating Gronwall's inequality to equation (1.6) yields that:

$$C = C_0 + ||h(0)|| \cdot T$$
$$K = L$$
$$u(t) = ||x^s(t)||$$
$$v(t) = 1$$

This means that:

$$||x^s(t)|| \leq (C_0 + ||h(0)|| \cdot T) \cdot \exp(L \cdot \int_s^{s+T} 1\,\mathrm{d}u) = (C_0 + ||h(0)|| \cdot T) \cdot \exp(L \cdot T)$$

We can combine this with:

$$||h(x^s(t))|| \leq ||h(0)|| + L \cdot ||x^s(t)||$$

This results in:

$$||h(x^s(t))|| \leq ||h(0)|| + L \cdot ((C_0 + ||h(0)|| \cdot T) \cdot \exp(L \cdot T))$$

Which concludes the proof for equation (1.5).

1.3.2 (b) Proving an important statement

Let $[t] := \max\{t_k : t_k \leq t\}$. Verify that, for all $m \in \mathbb{N}_0$ with $t_{n+m} \in [t_n, t_n + T]$

$$\left\| \int_{t_n}^{t_{n+m}} \left(h\left(x^{t_n}(t)\right) - h\left(x^{t_n}([t])\right)\right) \mathrm{d}t \right\| \leq LC_T \sum_{k=0}^{\infty} \alpha_{n+k}^2.$$

Firstly we can try to move the norm inside the integral and then use that h is Lipschitz:

$$\begin{aligned} || \int_{t_n}^{t_{n+m}} (h(x^{t_n}(t)) - h(x^{t_n}([t]))) \,\mathrm{d}t|| &\leq \int_{t_n}^{t_{n+m}} (||h(x^{t_n}(t)) - h(x^{t_n}([t]))||) \,\mathrm{d}t \\ &\leq \int_{t_n}^{t_{n+m}} L \cdot ||x^{t_n}(t) - x^{t_n}([t])|| \,\mathrm{d}t \end{aligned}$$

The integral can now be split into a sum of integrals over the area we integrate:

$$\int_{t_n}^{t_{n+m}} L \cdot ||x^{t_n}(t) - x^{t_n}([t])|| \,\mathrm{d}t = \sum_{k=0}^{m-1} \int_{t_{n+k}}^{t_{n+k+1}} L \cdot ||x^{t_n}(t) - x^{t_n}([t])|| \,\mathrm{d}t$$

The expression inside the integral can now be investigated (remember that $\dot{x}^s(t) = h(x^s(t))$):

$$\begin{aligned} ||x^{t_n}(t) - x^{t_n}([t])|| &= || \int_{[t]}^{t} h(x^{t_n}(u)) \,\mathrm{d}u|| \\ &\leq \int_{[t]}^{t} ||h(x^{t_n}(u))|| \mathrm{d}u \\ &\leq \int_{[t]}^{t} C_T \,\mathrm{d}u \\ &\leq \alpha_{n+k} \cdot C_T \end{aligned}$$

Where we use

$$[t] = t_{n+k} = \sum_{m \leq n+k-1} \alpha_m \leq t \leq t_{n+k+1} = \sum_{m \leq n+k} \alpha_m$$

With this we can see that: $t - [t] \leq \alpha_{n+k}$.
Then it becomes clear that:

$$\int_{t_{n+k}}^{t_{n+k+1}} (1) \,\mathrm{d}t \leq \alpha_{n+k}$$

Using this we can see that:

$$\begin{aligned} \sum_{k=0}^{m-1} \int_{t_{n+k}}^{t_{n+k+1}} L \cdot ||x^{t_n}(t) - x^{t_n}([t])|| \,\mathrm{d}t &\leq \sum_{k=0}^{m-1} \int_{t_{n+k}}^{t_{n+k+1}} L \cdot \alpha_{n+k} \cdot C_T \,\mathrm{d}t \\ &\leq \sum_{k=0}^{m-1} L \cdot \alpha_{n+k} \cdot C_T \cdot \int_{t_{n+k}}^{t_{n+k+1}} (1) \,\mathrm{d}t \\ &\leq \sum_{k=0}^{\infty} L \cdot C_T \cdot \alpha_{n+k}^2 \end{aligned}$$

With this it becomes clear that:

$$\int_{t_n}^{t_{n+m}} L \cdot ||x^{t_n}(t) - x^{t_n}([t])|| \,\mathrm{d}t \leq L \cdot C_T \cdot \sum_{k=0}^{\infty} \alpha_{n+k}^2$$

Combining everything and having the assumptions from the exercise, we can conclude that the following statement is true:

$$\left\| \int_{t_n}^{t_{n+m}} \left(h\left(x^{t_n}(t)\right) - h\left(x^{t_n}([t])\right)\right) \mathrm{d}t \right\| \leq LC_T \sum_{k=0}^{\infty} \alpha_{n+k}^2$$

2 Practical part

The code written for this assignment, is put in a repository on GitHub. I will refer to the different scripts, that I have written using hyperlinks (which the reader can click). The entire code-base can be viewed in this: GitHub-link. My code has drawn inspiration from different chapters of [6]. The coding language I have used is Python. The different dependencies for running the code can be found in this environment-file. Logging different experiments have been done with Neptune. View the different logs in these links: ReLe-final, ReLe-final-results and ReLe-opt.

2.1 The environment

The environment considered in this assignment consists of a 2d grid world. The grid is defined to a size of (x, y). Here x is the amount of rows and y is the amount of columns. The agent is spawned somewhere randomly in the grid. Two other items: a coin and a chest are also randomly placed in the grid world. The goal of the game is for the agent to pick up the coin and deliver it to the chest. Therefore the reward function (r) is defined to give a reward of 0.5 when the agent firstly picks up the coin. The agent is given a reward of 0.5 when the coin is delivered to the chest. This means that a total undiscounted reward of 1 can be achieved.
The state space (E) can be described with a vector containing 6 values:

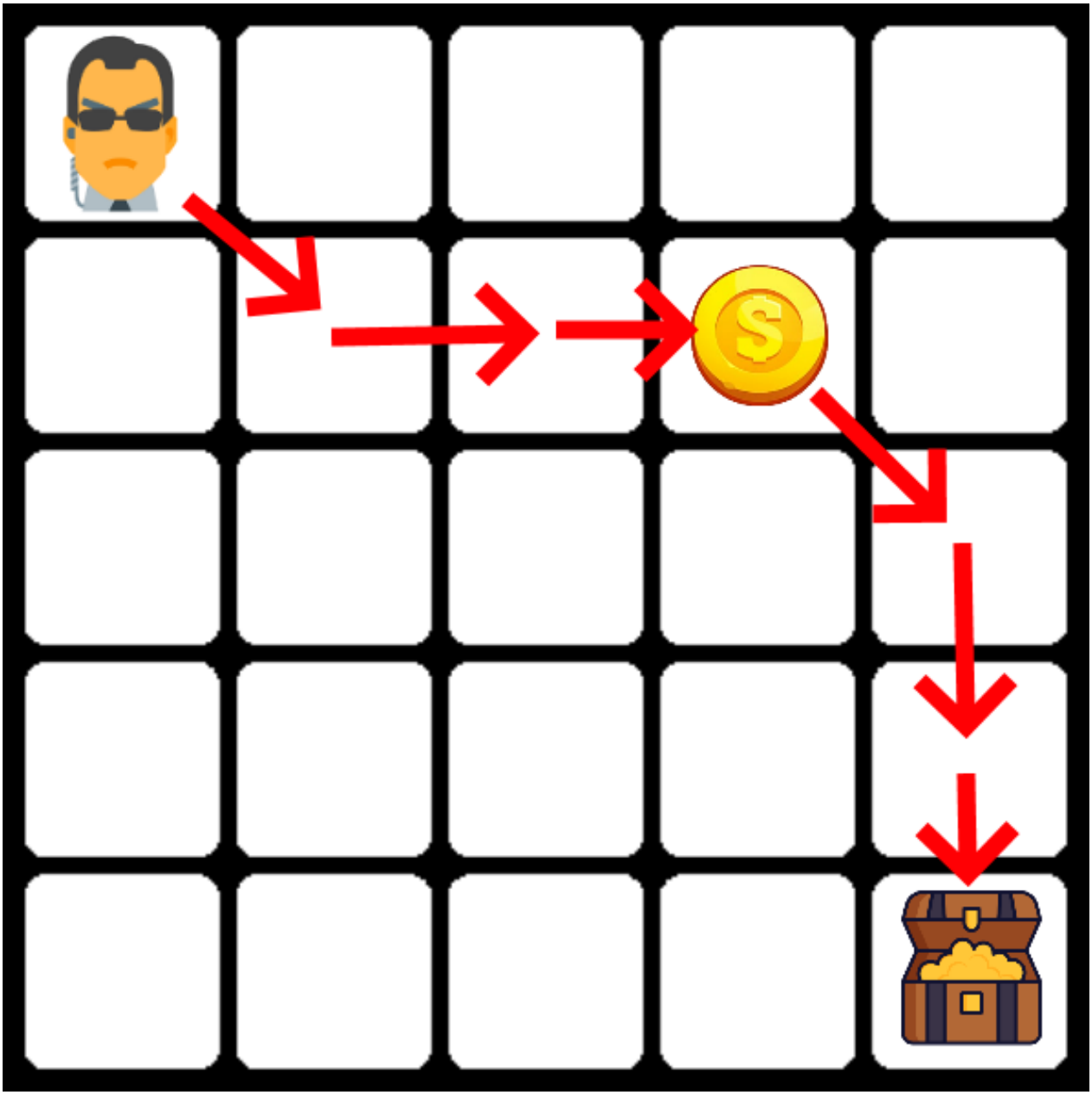
$$(agent_x, agent_y, coin_x, coin_y, chest_x, chest_y)$$

Here each value is an integer between "0" and "grid size - 1". If the agent is spawned at position $(agent_x, agent_y) = (0, 0)$ then the agent is placed at the top left corner of the world.
The action space (A) is always admissible with this tuple: (down = 2 / stand still = 1 / up = 0, right = 2 / stand still = 1 / left = 0)
0: (0, 0) -> Move up and left
1: (0, 1) -> Move up
2: (1, 0) -> Move left
3: (1, 1) -> Stand still
4: (0, 2) -> Move up and right
5: (2, 0) -> Move down and left
6: (1, 2) -> Move right
7: (2, 1) -> Move down
8: (2, 2) -> Move down and right

Each action moves the agent at max one value horizontally and/or vertically. In the case where the agent is placed alongside the edge of the world e.g. $(agent_x, agent_y) = (0, 0)$. Then the agent can call the action: $(0, 0)$. In this scenario the agent is trying to move out of the world. This is prohibited and the agent will remain in the original position $(agent_x, agent_y) = (0, 0)$. The agent could also call the action $(0, 2)$ in this case it is possible to move the agent down but not left. Therefore the new state will have the agent placed at: $(agent_x, agent_y) = (1, 0)$.
This environment is a deterministic environment, in the sense that actions always result in a deterministic outcome. Therefore the transition probability vector for each state and action contains a single one and zeros for all other entries.
The environment is written in the gym-setup and is defined in this script: chest_env.py. The amount of actions that can be called to the environment is not infinite. In the setup a variable: $number\_of\_actions$ is defined. This variable defines how many actions the agent can take in the environment. This means that the game stops when $number\_of\_actions$ is achieved or the coin

has been brought to the chest.
This setup can be summarized in the following illustration:

Figure 2.1: Illustration of the game



Here we see how our agent "Smith" employ an optimal strategy going firstly to the coin and then to the chest.
This environment specify a MDP of the form: $M = (E, A, (A(x))_{x \in E}, p, r)$ [2, page 23].

2.2 Analysis of Reinforcement Learning Algorithms

2.2.1 Deterministic expert policy

This deterministic environment has a simple optimal policy. The solution is to firstly go the direct way to the coin. When the agent has collected the coin, then the agent should go the directly to the chest.

This strategy has to check the agents placement according to its goal (coin/chest). Fore example if the agent is to the left or right of the goal. It also has to check if the agent is above or below the goal. If the agent is to the right of the goal, then an "left" action should be issued and vice versa. The same logic applies for the up-down action. Note, that the agent can move both vertically and horizontally per action (and should thus utilize this as much as possible). See the implementation of the "expert" policy in this script: expert_policy.py.

The expert policy was run with grid size 10x10. This resulted in the following output:

```
--- Start position ---
Player pos: tensor([8, 6], dtype=torch.int32)
coin pos: tensor([8, 8], dtype=torch.int32)
chest pos: tensor([6, 8], dtype=torch.int32)

--- Expert actions ---
frame index = 1   action = (2, 1)
Player pos: tensor([8, 7], dtype=torch.int32)
coin pos: tensor([8, 8], dtype=torch.int32)
chest pos: tensor([6, 8], dtype=torch.int32)
reward: tensor(0.)

frame index = 2   action = (2, 1)
Player pos: tensor([8, 8], dtype=torch.int32)
coin pos: tensor([8, 8], dtype=torch.int32)
chest pos: tensor([6, 8], dtype=torch.int32)
reward: tensor(0.5000)

frame index = 3   action = (1, 0)
Player pos: tensor([7, 8], dtype=torch.int32)
coin pos: tensor([7, 8], dtype=torch.int32)
chest pos: tensor([6, 8], dtype=torch.int32)
reward: tensor(0.)

frame index = 4   action = (1, 0)
Player pos: tensor([6, 8], dtype=torch.int32)
coin pos: tensor([6, 8], dtype=torch.int32)
chest pos: tensor([6, 8], dtype=torch.int32)
reward: tensor(0.5000)
```

This also display the reward structure of how the agent is rewarded for its actions. This is a very model-based technique, since this policy is created with this specific environment in mind. This policy was only possible to create, since we know the reward structure and the behavior of the environment.

2.2.2 Value iteration

The key element to this algorithm is the value function. This function is calculated recursively using:

$$V^{n+1}(x) = \max_{a \in A(x)} \left(r(x,a) + \gamma \sum_{y \in E} p(y \mid x,a) V^n(y) \right)$$

We could stay in a model based approach where we utilize, that we know the probabilities $p(y \mid x, a)$. But my approach is to use sampling from the environment to approximate these probabilities. To do this the following different tables are created

1. A "transits" table that keeps track of which states we start from and get to. This table is used for calculating the probabilities: $p(y|x,a)$. An example of the content of this table:

```
{((3.0, 3.0, 2.0, 4.0, 1.0, 3.0),
  (2, 1)): Counter({(3.0, 4.0, 2.0, 4.0, 1.0, 3.0): 9}),
 ((3.0, 4.0, 2.0, 4.0, 1.0, 3.0),
  (2, 2)): Counter({(4.0, 4.0, 2.0, 4.0, 1.0, 3.0): 1}),
 ((4.0, 4.0, 2.0, 4.0, 1.0, 3.0),
  (2, 1)): Counter({(4.0, 4.0, 2.0, 4.0, 1.0, 3.0): 1}),
 ((4.0, 4.0, 2.0, 4.0, 1.0, 3.0),
  (2, 0)): Counter({(3.0, 4.0, 2.0, 4.0, 1.0, 3.0): 4}),
 ((3.0, 4.0, 2.0, 4.0, 1.0, 3.0),
  (2, 0)): Counter({(2.0, 4.0, 2.0, 4.0, 1.0, 3.0): 12}),
 ((2.0, 4.0, 2.0, 4.0, 1.0, 3.0),
 ...}
```

Here the key (x, a) could be: ((3.0, 3.0, 2.0, 4.0, 1.0, 3.0), (2, 1)). Meaning that we are in state (3.0, 3.0, 2.0, 4.0, 1.0, 3.0) and use action (2, 1). The output is the states that has been visited after this action. Since this is a deterministic game, only one state is available. In this setup we get to state: (3.0, 4.0, 2.0, 4.0, 1.0, 3.0) always. Had there been any randomness in regards of which state we get to, then there would be multiple states for a key. Here the number of visits for each end-states y (in this example 9) is used to calculate $p(y|x,a)$.

2. A "reward" table. This table keep track of the immediate reward achieved when choosing action a. This reward is also dependent on the starting state: x and ending state: y. Here is an example of the output:

```
defaultdict(float,
{((1.0, 0.0, 1.0, 0.0, 3.0, 0.0), (0, 2), (2.0, 0.0, 2.0, 0.0, 3.0, 0.0)):
    tensor(0.),

 ((2.0, 0.0, 2.0, 0.0, 3.0, 0.0), (2, 1), (2.0, 1.0, 2.0, 1.0, 3.0, 0.0)):
    tensor(0.),

 ((2.0, 1.0, 2.0, 1.0, 3.0, 0.0), (0, 2), (3.0, 0.0, 3.0, 0.0, 3.0, 0.0)):
    tensor(0.5000),

 ((1.0, 3.0, 0.0, 3.0, 2.0, 0.0), (0, 1), (1.0, 2.0, 0.0, 3.0, 2.0, 0.0)):
    tensor(0.),

 ((1.0, 2.0, 0.0, 3.0, 2.0, 0.0), (2, 0), (0.0, 3.0, 0.0, 3.0, 2.0, 0.0)):
    tensor(0.5000),
 ...})
```

From this we see that:

```
((2.0, 1.0, 2.0, 1.0, 3.0, 0.0), (0, 2), (3.0, 0.0, 3.0, 0.0, 3.0, 0.0)):
    tensor(0.5000)
```

The agent has picked up the coin and is in state x: $(2.0, 1.0, 2.0, 1.0, 3.0, 0.0)$. This is one action $((0, 2))$ from the chest. When employing this action the agent delivers the coin to the chest in state $(3.0, 0.0, 3.0, 0.0, 3.0, 0.0)$ and gets a reward of 0.5. Another example is where the agent collects the coin:

```
((1.0, 2.0, 0.0, 3.0, 2.0, 0.0), (2, 0), (0.0, 3.0, 0.0, 3.0, 2.0, 0.0)):
    tensor(0.5000)
```

3. A "value" table that keeps track of the value in each of the states. This is also known as the value function.

```
defaultdict(float,
            {(0.0, 0.0, 0.0, 0.0, 0.0, 0.0): tensor(10.0000),
             (0.0, 0.0, 0.0, 0.0, 0.0, 1.0): tensor(4.7632),
             (0.0, 0.0, 0.0, 0.0, 0.0, 2.0): tensor(2.6280),
             (0.0, 0.0, 0.0, 0.0, 0.0, 3.0): tensor(0.4624),
             (0.0, 0.0, 0.0, 0.0, 0.0, 4.0): tensor(6.9208),
             (0.0, 0.0, 0.0, 0.0, 1.0, 0.0): tensor(9.5000),
             (0.0, 0.0, 0.0, 0.0, 1.0, 1.0): tensor(4.6938),
             ...
             })
```

Note: This table is taken from a run where the value function is not fully converged yet. Therefore some of the numbers are not correct. But it gives the indication that the value function is maximized when the game is solved e.g. the state (0.0, 0.0, 0.0, 0.0, 0.0, 0.0) has value 10. Being close to solve the game (one action away) is also very valuable e.g. the state (0.0, 0.0, 0.0, 0.0, 0.0, 1.0) has value 4.76.

The value iteration table is updated by the "value_iteration" method in the agent class. The code can be seen here (see also the code in this GitHub-link):

```
gen_all_states = self.env.iter_all_states()  # get all the states
gen_all_states = iter(gen_all_states)

for state in gen_all_states:  # for every state in the game
    gen_all_actions = self.env.iter_all_actions()  # get all the actions
    gen_all_actions = iter(gen_all_actions)

    state_values = [self.calc_action_value(state, action) for action in gen_all_actions]
    # iterate over all the possible actions:  r(x, a) + gamma * sum_y p(y|x, a) * V^n(y)

    self.values[state] = max(state_values)
    # V^(n+1)(x) = max_a (r(x, a) + gamma * sum_y p(y|x, a)*V^n(y))
```

In this code the value table is updated for each state. This means that we firstly iterate over each state of the game. For each state the state_values list is generated. The list contains an entry for each action. The entry is calculated as $r(x, a) + \gamma \cdot \sum_{y \in E} p(y|x, a) \cdot V^n(y)$. At last we can assign the maximum value to the value table for the given state. View the entire code for the value iteration algorithm in the script: value_iteration.py.

For learning the value function the following algorithm can be used [2, page 34]:

Algorithm 1 Value iteration algorithm

1: *Set* n *to* 0, *choose a tolerance level* ϵ *larger than* 0
2: *Create an initial candidate for the value function* $V^0 \in \mathbb{R}^{|E|}$
3: **for** $x \in E$ **do**
4: $\quad V^{n+1}(x) = \max_{a \in A(x)} \left(r(x,a) + \gamma \sum_{y \in E} p(y \mid x,a) V^n(y) \right)$
5: $\quad$ **if** $\left\| V^{n+1} - V^n \right\| < \frac{\epsilon(1-\gamma)}{2\gamma}$ **then**
6: $\quad\quad$ *set* $n^* = n$ *and go to step* 9
7: $\quad$ **else**
8: $\quad\quad$ *Increase* n *to* $n+1$ *and go to step* 3
9: **for** $x \in E$ **do**
10: $\quad \pi_\epsilon(x) \in \underset{a \in A(x)}{\arg\max} \left(r(x,a) + \gamma \sum_{y \in E} p(y \mid x,a) V^{n^*+1}(y) \right)$

Step 4 can be written in vector notation:

$$V^{n+1} = \mathcal{L} V^n$$

The final policy can also be written in vector notation:

$$\pi_\epsilon \in \arg\max_{\pi} \left(r_\pi + \gamma P_\pi V^{n^*+1} \right)$$

Here we know, that the value function has converged when the following condition is true:

$$\left\| V^{n+1} - V^n \right\| < \frac{\epsilon(1-\gamma)}{2\gamma}$$

The stopping criterion for the algorithm has though been chosen a bit differently in my setup. In the environment chosen, we know that the agent has solved the game when an undiscounted reward of 1 is achieved. This means that the agent has picked up the coin and brought it to the chest faster than the number of actions available. Therefore the policy can be evaluated using the game.

If the grid size is less than 5 then we evaluate the policy for all starting state spaces of the game. If the value iteration function can solve all cases (get a reward of 1 in all games). Then we know that it can solve all states of the game. With a grid size of 4 we will get: $4^6 = 4096$ different starting positions. Note though that the value iteration function might not have fully converged, but it is at this point useful. If the grid size is 5 or larger, then the policy is tested on 2000 random games. If the policy can solve these 2000 games perfectly, then we know that the policy is useful.

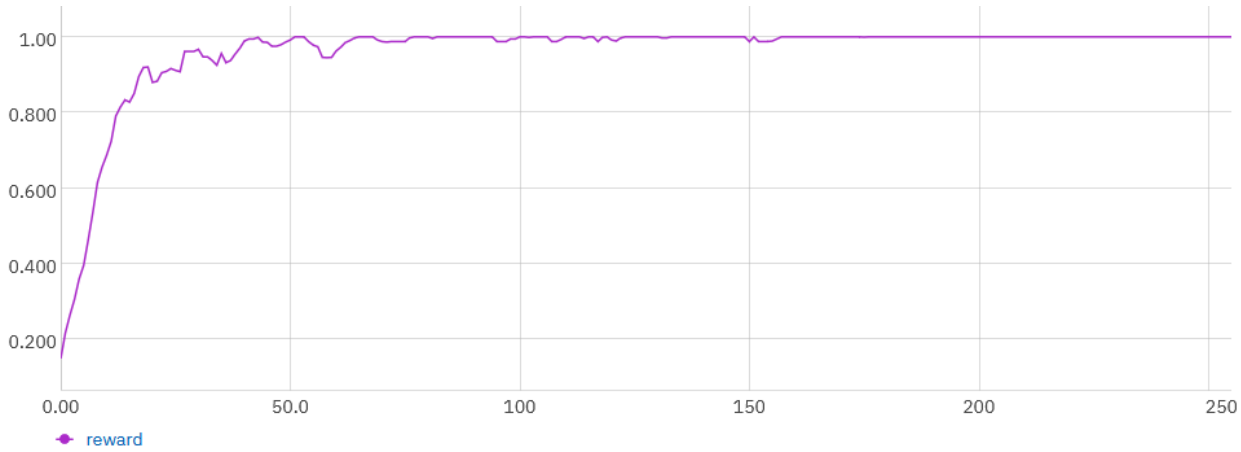
There are though a few caveats with this method. Firstly we noted that the value iteration function might not be fully converged. This means that the values it returns might not be perfect (it is not equal to the true value). Secondly the final policy that we achieve will solve the game optimally in regards to getting a total undiscounted reward of 1. But this policy might not solve the game in as few actions as possible. A fully converged value function will make sure, that we also solve the game as fast as possible. This is because of the discount factor.

This stopping criterion has some benefits. Most notably that we can stop the value iteration as early as the function can solve the game. The evaluation process is though only possible because of how the reward function is structured. We know that an undiscounted reward of 1 can always be achieved. In other setups, the reward function might not be as straight forward. In such scenarios the "normal" value iteration algorithm should be employed.

Results of running the value iteration on a 3x3 grid:

In this the value iteration function was tested for the game with grid size 3x3.
The undiscounted evaluation-reward compared to the amount of iterations can be seen in this figure:

Figure 2.2: Evolution of the reward for the value iteration algorithm



It took about 255 iterations which amounted to about 174 seconds to solve the entire game. For each iteration a total of 216 random actions are performed in the environment. It should also be noted, that this implementation also saves information to the different tables (see Value iteration) during the evaluation of the policy. It can be seen, that a rather random policy (few iterations) only achieve a low reward score. Whereas the final policy can solve the 3x3 game, no matter the starting position (we see that the reward is 1).

Results of the value iteration

The following table shows the convergence time of the value iteration algorithm. Here the grid size "n" is increased. The value displayed is how long it takes the algorithm to solve the game using the criteria described in the Value iteration section. The value iteration algorithm is run 5 times and the average of these are found to estimate the convergence time. See the code for this in the script: value_it_run.py. View the results in the table below:

Table 2.1: Results of the value iteration algorithm

nxn	1. time	2. time	3. time	4. time	5. time	avg. time
2x2	7 s	6 s	6 s	7 s	6 s	6.49 s
3x3	216 s	105 s	158 s	153 s	151 s	156.48 s
4x4	828 s	716 s	1450 s	1448 s	1530 s	1194.49 s
5x5	2530 s	1202 s	2841 s	1574 s	1752 s	1979.91 s
6x6	2485 s	3886 s	3691 s	3100 s	4592 s	3550.90 s
7x7	8414 s	6787 s	9825 s	7452 s	11333 s	8761.96 s
8x8	25436 s	25655 s	25097 s	25624 s	25255 s	25413.56 s
9x9	123028 s	122349 s	117213 s	116085 s	122812 s	120297.48 s

It should be noted that the policy is evaluated using all starting position of the game for the grid size n = 2, 3 and 4. When the grid size is 5 or larger we evaluate the policy on 2000 random games before the algorithm stops.

From the results it is clear that the solution time increase more than linear as a function of the environment size "n".

2.2.3 Q-learning

The Q-learning section is split into two implementations. The first implementation calculates the Q-values using estimated probabilities in the same fashion as the value iteration algorithm implemented in the: Value iteration section. The second implementation utilize Q-learning with decaying alpha-values.

The first implementation we call "Q learning with p" since we approximate the probabilities directly. Here we utilize a "rewards" and "transits" table. See an explanation of these in the: Value iteration section. Apart from these two a Q-values table is created. The difference here is that the Q-table like the Q-function is not only dependent on the state but also the action. Therefore a key in the Q-table will be (x, a). An example of the content of the Q-table can be seen here:

```
defaultdict(float,
            {((0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0, 0)): tensor(10.0000),
             ((0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0, 1)): tensor(10.0000),
             ((0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0, 2)): tensor(9.0000),
             ((0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (1, 0)): tensor(10.0000),
             ((0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (1, 1)): tensor(10.0000),
             ((0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (1, 2)): tensor(9.0000),
             ...})
```

In all cases we see that the agent is spawned in the corner on-top of the coin and chest. In this case the agent should just stay in place (action = (1, 1)). But since this is in the corner, then the agent can also take an action towards the walls. This is because the agent will not be moved outside the grid and will stay in place (thus this action is equal to the action (1, 1)). Moving away from the chest will not be optimal - we would use 2 actions to solve the game instead of 1.

The updating of the Q-values are also quite different from the value-iteration method. The code for updating the Q-values can be seen here:

```python
def q_learn(self):
    """
    Here we iterate over all states and actions and calculate the max state value.
    With this we can fill the Q-value table.

    :return: None
    """
    gen_all_states = self.env.iter_all_states()  # get all the states
    gen_all_states = iter(gen_all_states)

    for state in gen_all_states:  # for every state in the game
        gen_all_actions = self.env.iter_all_actions()  # get all the actions
        gen_all_actions = iter(gen_all_actions)
        for action in gen_all_actions:
            action_value = 0.0
            target_counts = self.transits[(state, action)]
            # get the amount of transits for this state and action

            total = sum(target_counts.values())
            for tgt_state, count in target_counts.items():
                # For every target state we can end up in and the times we have done so
                key = (state, action, tgt_state)  # get key
                reward = self.rewards[key]  # get rewards for going from state to tgt_state using action
                best_action = self.select_action(tgt_state)  # select the best action to take
                val = reward + self.gamma * self.q_values[(tgt_state, best_action)]
                # val = r(x, a) + gamma * max_a(Q(Y_n+1, a))
                action_value += (count / total) * val
                # action_value = p(y|x, a) * val
            self.q_values[(state, action)] = action_value
            # update the Q-values table using r(x, a) + gamma * sum_y p(y|x, a) * max_b(Q^n(y, b))
```

This method therefore estimate the Q-values directly using the formula [2, page 57]:

$$Q^*(x,a) = r(x,a) + \gamma \cdot \sum_{y \in E}(p(y|x,a) \cdot max_{b \in A(x)}(Q^*(y,b)))$$

Except that the Q-function is not optimal in the beginning of training but is going to converge towards Q^*. See the entire code in this script: Q_learning_with_p.py.

For the second implementation the Q-learning framework follows the lecture notes [2, page 57]. Here I use the following algorithm [7, page 131].

Algorithm 2 Q-learning(α_n, ϵ)

Initialize $Q(s,a)$, *for all* $s \in S+$, $a \in A(s)$, *arbitrarily except that* $Q(terminal, \cdot) = 0$
for *n episodes* **do**
  Initialize S
  for *for each state in episode* **do**
    Choose A *from* S *using policy derived from* Q (*e.g.* $\epsilon - greedy$)
    Take action A, *observe* R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha_n \cdot (R + \gamma \cdot max_b(Q(S',b)) - Q(S,A))$

The ϵ-greedy algorithm selects a random action with probability ϵ. The best performing action is in a greedy way selected with (1 - ϵ) probability (using the Q-values gathered so far).
α_n is a sequence that fulfill the following properties:

$$\sum_{n \in \mathbb{N}}(\alpha_n) = \infty, \quad \sum_{n \in \mathbb{N}}(\alpha_n^2) < \infty$$

A step-size that adhere to these conditions are for instance:

$$\alpha_n = \frac{1}{d \cdot n + 1}$$

Here we set $d \in (0, 1]$. It can be proven that this series adhere to the conditions. Firstly we see that:

$$\alpha_n = \frac{1}{d \cdot n + 1} \geq \frac{1}{n+1}$$

Here the last term ($\frac{1}{n+1}$) is involved in the harmonic series [8]. We know that this series is divergent, and this makes our series divergent, since all terms are larger. Therefore we can conclude that:

$$\sum_{n=1}^{\infty} \frac{1}{d \cdot n + 1} = \infty$$

Now for the second condition (see the proof for the zeta(2) function in [9]):

$$\begin{aligned}
\sum_{n=1}^{N} \frac{1}{(n \cdot d + 1)^2} &< 1 + \sum_{n=2}^{N} \frac{1}{n \cdot d \cdot (n \cdot d + 1)} \\
&= 1 + \sum_{n=2}^{N} \left(\frac{1}{n \cdot d} - \frac{1}{n \cdot d + 1} \right) \\
&= 1 + \frac{1}{2 \cdot d} - \frac{1}{N \cdot d + 1} \overset{N \to \infty}{\longrightarrow} 1 + \frac{1}{2 \cdot d}
\end{aligned}$$

This means that the sum converges as N increase to infinity. With this it can be concluded that:

$$\sum_{n=1}^{\infty}(\frac{1}{d \cdot n + 1})^2 < \infty$$

This means that the series can be used for decaying the alpha values in the Q-learning setup. With this series the Q-learning algorithm has been implemented as:

```
def q_learn(self, nr_episodes: int, epsi: float):
    """
    Here we run the Q-learning algorithm

    :return: None
    """

    is_done = False
    self.alp = self.alpha()  # get the alpha value for this iteration
    for _ in range(nr_episodes):
        while not is_done:
            action = self.select_action(self.state, epsi=epsi)  # get the action
            new_state, reward, is_done, _ = self.env.step(action)  # take the action in the env
            self.rewards[(self.state, action, new_state)] = reward  # insert in reward table

            q_now = self.q_values[(self.state, action)]
            best_action = self.select_action(new_state)  # take the best action for the next state
            q_tar = self.q_values[(new_state, best_action)]

            action_value = (reward + self.gamma * q_tar - q_now)
            # (r(x, a) + lambda * max_b(Q_n(Y_n+1(x,a), b)) - Q_n(x, a))
            action_value = q_now + self.alp * action_value  # Q_n(x, a) + action_value

            self.q_values[(self.state, action)] = action_value
            self.state = self.env.reset() if is_done else new_state  # reset or run from current state
        is_done = False
```

Note here that we do not need the transits table any more, since this algorithm calculates them indirectly by approximating the Q-function. Note that there is a transit table in the code (in this link) on Github. This table is though not directly used in the algorithm, but it can be used for debugging.

Hyper-parameter search for the decay and epsilon

The decay "d" parameter determines how fast the alpha values decay towards 0. If this happens to quickly then the algorithm will not converge to the correct Q-values. If it is to slow, then the training time will be quite large. Therefore it is an aim to find a decay value that is quite good.
In the Q-learning frame work we take ϵ-random actions when learning the Q-values. The value of ϵ should also be optimized for the framework.

For the hyper-optimization Optuna is used. Please refer this script: Q_learn_optuna.py to see the Optuna setup. For this setup we test the decay values between: 1e-09 and 0.1 and ϵ between 0.05 and 0.5.
The optimization criterion is minimizing the amount of iterations in the training loop. This implicitly makes sure that the algorithm also converge (otherwise the amount of iterations would be ∞). We do not optimize on time directly. This is because the execution time can be influenced by e.g. the work-load on the machine (that does the optimization).
This also allows us to run the process in parallel. This is done with the following script: optuna_runner.py.

The hyper-parameter search resulted in:

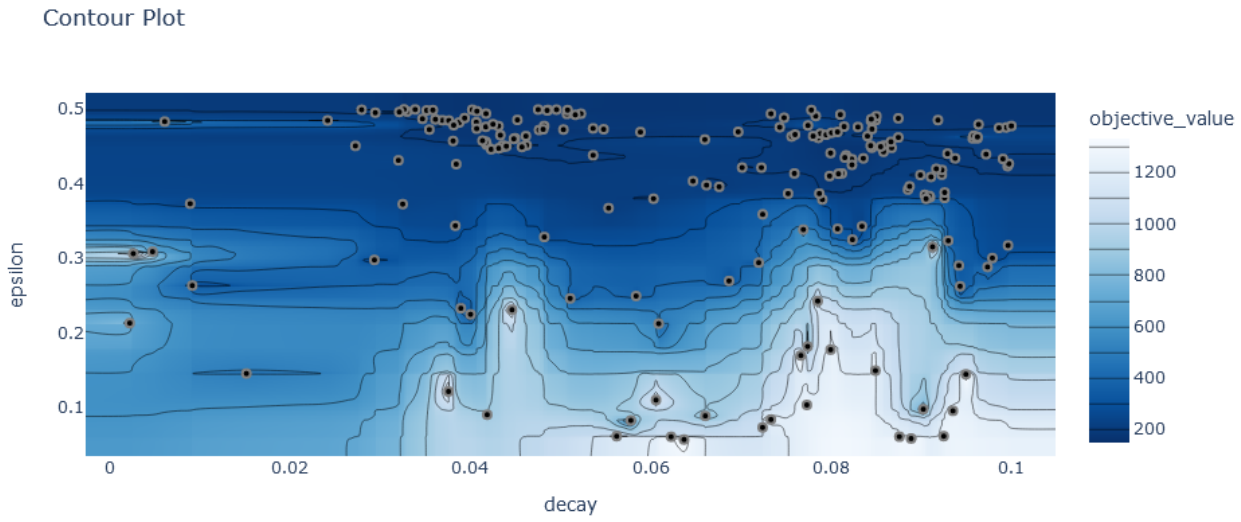
Table 2.2: Results of hyper optimization. "par. imp." is an abbreviation of "parameter importance"

grid size	number of trials	decay	ϵ	par. imp. of decay	par. imp. of ϵ
2x2	≈ 200	0.099	0.498	0.57	0.43
3x3	≈ 400	0.050	0.473	0.12	0.88
4x4	≈ 400	0.092	0.494	0.06	0.94

Note that the grid size numbers contain click-able links. These numbers links to the Neptune-runs where the hyper parameters have been found. The following plot and other information about the runs can be found in these links. The parameter importance is a measure Optuna provides. This measure is in percentage and is assigned to the parameters that we optimized over. The percentage indicate the importance of setting this parameter optimally.

The following plot illustrate the amount of iterations (objective value) in terms of the parameters we search over:

Figure 2.3: Contour plot of the hyper optimization for grid size 3x3



Here each dot represents a trail with the parameters on the axis. The amount of iterations can be read from the color of the background. This for instance tells us that a high decay value with low epsilon is not optimal. Whereas an epsilon value around 0.5 generally do quite well.

Results

For the first implementation (Q-learning with p) the following results were obtained (please refer the code in this script: Q_learn_p_run.py):

Table 2.3: Results for Q-learning method with p

nxn	1. time	2. time	3. time	4. time	5. time	avg. time
2x2	10 s	7 s	6 s	5 s	7 s	6.85 s
3x3	185 s	223 s	256 s	135 s	133 s	186.28 s
4x4	1911 s	3016 s	2092 s	2520 s	3376 s	2583.22 s
5x5	4221 s	4390 s	4690 s	7806 s	5227 s	5266.83 s
6x6	16829 s	10793 s	10526 s	12166 s	14747 s	13012.26 s
7x7	20916 s	20879 s	27291 s	41944 s	45292 s	31264.50 s
8x8	42696 s	68643 s	68137 s	49803 s	72001 s	60255.97 s

These results show that the convergence time really grows quickly. Sometimes it more than doubles in convergence time by just adding one additional layer horizontally and vertically to the grid.

The next table contains the results of the second Q-learning method. The optimized hyper-parameters for grid size 2-4 were used. It was seen from the hyper optimization that an ϵ value of 0.5 and a decay value of 0.06 do quite all right. Therefore these parameters are set for runs with a larger grid size. The code for running the algorithm can be found in this script: Q_learn_run.py.

Table 2.4: Results for second Q-learning method

nxn	1. time	2. time	3. time	4. time	5. time	avg. time
2x2	6 s	4 s	5 s	4 s	6 s	5.07 s
3x3	70 s	67 s	75 s	68 s	73 s	70.58 s
4x4	1077 s	865 s	644 s	856 s	640 s	816.42 s
5x5	431 s	571 s	480 s	471 s	532 s	496.76 s
6x6	2509 s	2200 s	2056 s	2118 s	2342 s	2245.25 s
7x7	8279 s	8917 s	9311 s	9582 s	9391 s	9095.89 s
8x8	26965 s	31283 s	34215 s	32942 s	31482 s	31377.34 s
9x9	111001 s	131986 s	124153 s	138624 s	117079 s	124568.67 s

There seems to be a speed increase for the algorithm going from a grid size of 4 to 5. One of the most prominent factors of why this might be, is the testing length. For the grid size of 4 we test all start configurations. This amounts to $4^6 = 4096$ different initial game start per iteration. Going into n=5 we only choose 20 random starting points (but do it for 100 iterations). Therefore fewer games are played for the larger environments. The larger environments take more actions to solve, since they are larger (the chest, coin and agent can be spawned further apart).

2.2.4 Deep Q-learning

The deep Q-learning setup can be summarized using this pseudo code [6, 10, page 686-687, page 138]:

Algorithm 3 Deep Q-learning

Initialize main network Q and target network Q' parameters with random values
Initialize the replay buffer D
while *not converged* **do**
  for *t in episodes* **do**
    Observe state x and select an action using the ϵ − greedy policy
    Perform the action and observe reward r and next state x'
    Store transaction (x, a, r, x') in replay buffer
    Sample mini − batch from replay buffer D
    Calculate target : $y_i = r_i$ if episode ended else $y_i = r_i + \gamma \cdot max_{a' \in A}(Q'(x'_i, a'))$
    Calculate loss $J(Q) = \frac{1}{K}\sum_{i=1}^{K}(y_i - Q(x_i, a_i))^2$
    Perform a gradient descent step for main network
    Update target network after N steps (using the main network)

For training this algorithm, the ϵ value is selected close to 1 and is then decayed toward 0. The idea is that we start with much exploration in the beginning of the training. Then we slowly start exploiting the policy when the network has learned something. This is done by lowering the ϵ value until the end of training. Therefore we expect to have a good policy toward the end of the training [6, page 136]. How fast the ϵ should decay is a hyper parameter that should be tuned.

The replay buffer used is called "ExperienceBuffer" and is located in this replay_buffer.py script. The purpose of the replay buffer is to store single transactions from state x to state x' with action a and reward r. All these values are important for the deep Q-learning algorithm. The replay buffer is very important when training the neural network (NN). When we train the neural network we use stochastic gradient descent (SGD). This assumes that the samples we show the network are i.i.d. This is why the network in this fashion cannot be trained directly on episodes [6, page 136-137]. There will be some dependence between the frames shown in the episode. The replay buffer is used to sample a large amount of random transactions. This will help by removing some of the dependence between frames.

The network used for training is defined in this net.py script. The network contain multiple linear layers. The first layer goes from the 10 observations to 50 values. Then from 50 to 500 and then from 500 to 90 and lastly from 90 to 9. Between all layers we utilize ReLU-activation functions. The 9 outputs are the different Q-values for each of the 9 actions given the state (input) to the model. See the 9 different actions when we specify the MDP in: The environment section. Additionally the representation of the state-space has been changed to:

$$\left(\frac{agent_x}{grid\_size}, \frac{agent_y}{grid\_size}, \frac{coin_x}{grid\_size}, \frac{coin_y}{grid\_size}, \frac{chest_x}{grid\_size}, \frac{chest_y}{grid\_size}, \frac{coin_x - agent_x}{grid\_size}, \frac{coin_y - agent_y}{grid\_size}, \frac{chest_x - agent_x}{grid\_size}, \frac{chest_y - agent_y}{grid\_size}\right)$$

This should scale the input values to be between 0 and 1. It also adds the additional information of how far the agent is from the coin and chest. All of this should help the network converge.
The network is utilized to approximate the Q-function in this setup. This network is also used as the target network. The reason that we need a target network lies in the fact that we use the Bellman equation to calculate $Q(x, a)$. Here we utilize $Q(x', a')$. But x and x' will be very similar since there is only one step between them. Therefore it is going to be difficult for the NN to distinguish between x and x'. That means when we update the NN to approximate $Q(x, a)$, then we indirectly effect

$Q(x', a')$ [6, page 137]. To remedy this we freeze the parameters of the target network. This result in the same value for $Q(x', a')$ (between updates). When the training has run for N-iterations, we update the target-network. Here N is a hyperparameter that is to be tuned.

The loss function for the network is called calc_loss_double_dqn and is located in this script. To calculate the loss value the loss function nn.MSELoss() from PyTorch is used. This function calculates the loss as $l_n = (x_n - y_n)^2$.
For training these different parameters can be set:

```
params = {"number_of_actions": 10,
          "grid_size": 3,
          "gamma": 0.99,
          "batch_size": 32,
          "replay_size": 2000000,
          "lr": 1e-4,
          "sync_target_frames": 10000,
          "replay_start_size": 50000,
          "epsilon_decay_last_frame": 500000,
          "epsilon_start": 0.99,
          "epsilon_final": 0.000001,
          "amount_of_eval_rounds": 450}
```

Here the "number_of_actions" and "grid_size" decide the size of the environment. "gamma" is the discount factor and is sometimes environment specific. In this case it can be freely set. The "amount_of_eval_rounds" is the amount of episodes the network should be able to perfectly solve consecutively. The last parameters are hyper-parameters. The "batch_size" determines how many transactions are sampled from the replay buffer when we update the network. The "replay_size" is used to determine how big the replay buffer is. "lr" is the learning rate used for our optimizer (in the code we use ADAM). "sync_target_frames" determine how many iterations before we update the target network. "replay_start_size" tells us how many transactions should be stored in the replay buffer before we start training the network. The ϵ parameters: "epsilon_decay_last_frame", "epsilon_start" and "epsilon_final" are used to decay the ϵ value. The following code is run for each action called to the environment:

```
epsilon = max(params["epsilon_final"],
              params["epsilon_start"] - frame_idx / params["epsilon_decay_last_frame"])
```

Here frame_idx is the amount of frames seen. This is also the total amount of states that has been seen from the environment. The implementation of the deep Q learning algorithm can be found in this script: deep_Q.py.

Hyper-optimization for Deep Q-learning
Running the Q-learning algorithm takes quite some time, therefore only few trials have been done. To minimize the search space only the parameters "sync_target_frames" and "epsilon_decay_last_frame" are optimized. These are further only tested for a subset of values. See the optimization setup in the optuna_runner.py script. For this method we stop when the algorithm can consecutively solve 450 games. See the optimization results in the table below:

Table 2.5: Results of hyper optimization. "par. imp." is an abbreviation of "parameter importance"

grid size	sync_target	epsilon_decay	par. imp. of sync_target	par. imp. of epsilon_decay
2x2	40000	100000	0.06	0.94
3x3	10000	250000	0.20	0.80
4x4	40000	500000	0.01	0.99

The table gives an indication that the epsilon_decay is an important parameter for the speed of convergence. How often the target network is synced is not as important, but it seems that a good large value (around 40 k) will make the algorithm converge. The epsilon_decay values are expected to increase, since the larger environment takes longer time to learn. The parameters in the table are used for the speed test run of the algorithm. For the tests with grid size larger than 5 these parameters are set:

```
{"sync_target_frames": 50000,
 "epsilon_decay_last_frame": 250000 * grid_size,
 "replay_size": 3500000,
 "replay_start_size": 70000}
```

These values are a bit conservatively set, but this should still allow the algorithm to converge (just slower than if optimal parameters were used).

Results for Deep Q-learning

Using the parameters found in "Hyper-optimization for Deep Q-learning" we run the algorithm 5 times and time each run. This is done with increasing grid size. See the results here:

Table 2.6: Results for the deep Q-learning method

nxn	1. time	2. time	3. time	4. time	5. time	avg. time
2x2	89 s	91 s	92 s	101 s	94 s	93.30 s
3x3	1267 s	1016 s	1876 s	916 s	1096 s	1234.28 s
4x4	1754 s	1756 s	1403 s	1785 s	1919 s	1723.50 s
5x5	2394 s	3903 s	2169 s	3089 s	3732 s	3057.18 s
6x6	4507 s	2651 s	4408 s	5543 s	3176 s	4057.13 s
7x7	4297 s	4183 s	2822 s	2275 s	4970 s	3709.47 s
8x8	7110 s	6764 s	4750 s	2686 s	4105 s	5082.81 s
9x9	4944 s	5899 s	3539 s	4980 s	6037 s	5080.02 s
10x10	5229 s	3280 s	3942 s	3513 s	4390 s	4070.75 s

From these results it can with the optimal parameters (grid size 2-4) be seen that the general convergence time grows a lot with the increasing grid size. But with the conservative parameters (for grid size 5-10) only a little increase of convergence time can be seen. This maybe caused by the parameter settings. If the parameters are more optimal for larger environments, then the convergence speed should be faster for these. Using optimal parameters for each grid size, might make the correlation between convergence time and grid size more evident.
Note: Another consideration is that the smaller grid size has observations that are close to each-other, so some dependency is present (when we randomly sample). This could influence the convergence time of the network.

It was also possible to train the network with a grid size of 20. This was done in 14150 seconds which is about 4 hours. See the run in this Neptune-link. It should though be noted, that some of the test-runs of this algorithm did not converge. Meaning that the final network could not solve the game, and a restart of the process is necessary.

2.2.5 Deep Q-learning from Demonstrations (DQfD)

This method utilize supervised learning to help the neural network learn the Q-values faster. This is done with a so called expert policy. A good example of why this can be useful: We can have a setup where we want to teach an agent to drive a car. Letting the agent learn from scratch will result in a lot of traffic accidents in the initial phase of learning. To avoid some of this, then why not let the agent learn a few things from an expert driver? This is what DQfD is about [10, page 604]. This method is dependent on an expert policy, which most likely is model-based (but does not need to be). The following algorithm explains the setup, of this method [11, page 4]:

Algorithm 4 Deep Q-learning from Demonstrations

Initialize main network Q and target network Q' parameters with random values
Initialize $replay - buffer: D$ with prioritization
for *n in $expert\_episodes$* **do**
    Run expert algo and save (x, a, r, x') to D
while *pre_train_phase* **do**
    Sample $expert\ mini - batch$ from $replay\ buffer\ D$
    Calculate target: $y_i = r_i$ if episode ended else $y_i = r_i + \gamma \cdot max_{a' \in A}(Q'(x'_i, a'))$
    Calculate loss $J(Q)$
    Perform a gradient descent step for main network
    Update target network after N steps (using the main network)
while *not converged* **do**
    for *t in episodes* **do**
        Observe state x and select an action using the $\epsilon - greedy$ policy
        Perform the action and observe reward r and next state x'
        Store transaction (x, a, r, x') in replay buffer
        Sample $mini - batch$ from $replay\ buffer\ D$
        Calculate target: $y_i = r_i$ if episode ended else $y_i = r_i + \gamma \cdot max_{a' \in A}(Q'(x'_i, a'))$
        Calculate loss $J(Q)$
        Perform a gradient descent step for main network
        Update target network after N steps (using the main network)

This algorithm introduce multiple new steps. Firstly we save some of the experts transactions to the replay buffer. Then we for a finite amount of iterations train the network using these values. This should give us a good initial policy. Lastly we go into the Q-learning setup and train the NN until convergence.

Loss for DQfD

The loss function is one of the most changed functionality compared to the basic deep Q-learning setup. The loss used for DQfD is defined as:

$$J(Q) = \lambda_1 J_{DQ}(Q) + \lambda_2 J_n(Q) + \lambda_3 J_E(Q) + \lambda_4 J_{L2}(Q)$$

The different values will be described in the following:

$J_{DQ}(Q)$: Double DQN loss

It has been proven that the basic DQN tends to overestimate the Q-values. The basic DQN target Q-value is calculated as:

$$Q(x_t, a_t) = r_t + \gamma \cdot max_{a \in A}(Q'(x_{t+1}, a))$$

A solution is to select the action with largest Q-value from our main trained network, and then find the corresponding Q-value from the target-network:

$$Q(s_t, a_t) = r_t + \gamma \cdot Q'(s_{t+1}, arg\, max_{a \in A}(Q(s_{t+1}, a)))$$

This has been proven to solve the overestimation completely. This new architecture is called double "DQN" [6, page 201-202].

$J_n(Q)$: N-step DQN loss

The Bellman update used in deep Q-learning is:

$$Q(x_t, a_t) = r_t + \gamma \cdot max_{a \in A}(Q(x_{t+1}, a_{t+1}))$$

This is a recursive equation. We can therefore unroll the equation one more time:

$$Q(s_t, a_t) = r_t + \gamma \cdot max_{a \in A}(r_{a,t+1} + \gamma \cdot max_{b \in A}(Q(s_{t+2}, b_{t+2})))$$

The idea is then, if we have taken an optimal or near optimal action at time step t+1, then we can omit the first max operator [6, page 197-201]. Then the new update becomes:

$$Q(s_t, a_t) = r_t + \gamma \cdot r_{a,t+1} + \gamma^2 \cdot max_{b \in A}(Q(s_{t+2}, b_{t+2}))$$

We could in theory do this any N times. The idea for all of this is that if we have a good approximation of $Q(s_{t+2}, b_{t+2})$ then we in fewer iterations can approximate $Q(s_t, a_t)$. The problem with this method, is that we drop the max operator under the assumption that the policy acted optimally. This is not the case in the beginning of training where a lot of exploration with random actions are made. In these cases the update will be incorrect [6, page 197-201]. In the code N is fixed to 3.

$J_E(Q)$: Supervised classification loss

One of the ways the agent learns from the expert demonstrations is by the supervised classification loss ($J_E(Q)$). The loss can be defined as:

$$J_E(Q) = max_{a \in A}[Q(x, a) + l(a_e, a)] - Q(x, a_e)$$

Here the actions by the agents is denoted as "a" and the expert actions are "a_e". The marginal function "l" is defined as:

$$l(a_e, a) = \begin{cases} 0 & \text{if } a_e = a \\ 1 & \text{else} \end{cases}$$

The goal is to minimize the loss function. This loss will decrease the Q-values for the "non-optimal" actions which is the actions that are not taken by the expert. This will make the agent imitate the expert policy [11, page 4].

$J_{L2}(Q)$: L2 regularization loss

The $J_{L2}(Q)$ denote the L2 regularization loss. This should help preventing the network from over-fitting to the demonstration data [11, page 4].

$$J_{L2}(Q) = \sum_p (p^2)$$

The loss is calculated as the sum over all parameters (weights and biases in the network) "p" squared.

Lastly the emphasis of the different losses is weighted though the four different parameters: λ_1 to λ_4. You can fix λ_1 (e.g. to 1) and then only have 3 parameters to tune. The implementation of these losses can be found in the loss.py script.

Prioritized replay buffer

The basic deep Q-learning utilize uniform random sampling from the replay buffer. An alternative approach is to assign priorities to the buffer samples based on the training loss. Then we can sample proportional to these priorities. The idea behind this method, is to: "Train more on data that surprise you" [6, page 210].
The priority of every sample in the buffer is calculated as:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

Here the parameter α determines how much emphasis that is given to the priority. The larger the α the more emphasis on the priority. Uniform sampling can be achieved by setting $\alpha = 0$. It is recommended to set $\alpha = 0.6$ as a start, but this hyper parameter can be tuned.
The new sampling has introduced bias into the sampling process. This new approach samples some training samples more often than others. To deal with this weights has to be calculated for each training sample. These weights can then be used to modify the loss of the different training samples. The weight is defined as: $w_i = (N \cdot P(i))^{-\beta}$. Here β is a new hyper parameter which lies between 0 and 1. Setting $\beta = 1$ we compensate fully for the bias. But it might help the convergence of the network to start β between 0 and 1 and then increase it to 1 during training. The method of prioritized replay buffer has proven to significantly improve convergence and the policy quality of the DQN [6, page 210]. The implementation of this replay buffer can be found in the replay_buffer.py script. View the code for the DQfD setup in this script: dqfd.py.

Hyper-parameter search for DQfD

This method introduce a lot of new parameters. Here is a list of the different parameters that can be set:

```
lambda_loss = {"lambda_dq": 1, "lambda_n": 1, "lambda_je": 1, "lambda_l2": 0.0005}
params = {"number_of_actions": 256,
          "grid_size": 128,
          "gamma": 0.99,
          "batch_size": 32,
          "replay_size": 200000,
          "lr": 1e-4,
          "sync_target_frames": 10000,
          "replay_start_size": 50000,
          "epsilon_decay_last_frame": 500000,
          "epsilon_start": 0.99,
          "epsilon_final": 0.000001,
          "beta_frames": 10000000,
          "beta_start": 0.4,
          "expert_play": 50000,  # The amount of expert frames!
          "pre_train_phase": 10000,
          "lambda_loss": lambda_loss,  # the weights for the different loss-functions
          "amount_of_eval_rounds": 450,  # how many games to ace in a row.
}
```

Some of the parameters here also appear in the deep Q-learning framework please refer to the: Deep Q-learning section. The "beta_frames" and "beta_start" parameters are used to define and decay the beta parameter for the prioritized replay buffer. The "expert_play" parameter determines how many frames are sampled from the expert policy. The "pre_train_phase" parameter determines how many training iterations are necessary where only the expert data is revealed to the network. Lastly the "lambda_loss" is the different weights to the loss functions. See the optimization setup in the optuna_runner.py script.

The parameters: beta_frames, epsilon_decay, expert_play and sync_target was optimized for the setup with grid size 2 and 3. The results can be viewed in these tables:

Table 2.7: Results of the hyper optimization

grid size	beta_frames	epsilon_decay	expert_play	sync_target
2x2	250000	40000	5000	40000
3x3	50000	50000	10000	20000

Table 2.8: Parameter importance

grid size	beta_frames	epsilon_decay	expert_play	sync_target
2x2	0.04	0.16	0.75	0.05
3x3	0.22	0.40	0.28	0.10

The parameters and their importance is quite different between the grid sizes. This might be because very few iterations were run for the hyper parameter tuning. From the few hyper optimization runs it is clear that the sync_target parameter is not the most influential. When we test this method we will use the parameters specified in "params" above. Apart from the optimized parameters from Table 2.8 for grid size 2 and 3. For the rest of the grid sizes we set these parameters with:

```
lambda_loss2 = {"lambda_dq": 1, "lambda_n": 1, "lambda_je": 1, "lambda_l2": 0.001}
params_to_try = params_to_try | {"sync_target_frames": 40000,
                                 "epsilon_decay_last_frame": 40000 + 1000 * (grid_size - 4),
                                 "beta_frames": 50000 + 10000 * (grid_size - 4),
                                 "expert_play": 10000 + 2500 * (grid_size - 4),
                                 "pre_train_phase": 10000 + 2500 * (grid_size - 4),
                                 "lambda_loss": lambda_loss2}
```

See the script for running the test of DQfD in the dqfd_run.py file.

Results for the DQfD

The following table summarize the convergence time of the DQfD setup:

Table 2.9: Results for the DQfD method

nxn	1. time	2. time	3. time	4. time	5. time	avg. time
2x2	90 s	48 s	105 s	72 s	108 s	84.54 s
3x3	417 s	518 s	427 s	381 s	510 s	451.70 s
4x4	474 s	481 s	505 s	506 s	384 s	470.00 s
5x5	661 s	548 s	5400 s	5400 s	1605 s	937.87 s
6x6	232 s	5400 s	135 s	5400 s	5400 s	183.24 s
7x7	139 s	185 s	139 s	139 s	143 s	148.96 s
8x8	158 s	178 s	199 s	156 s	154 s	169.06 s
9x9	188 s	265 s	224 s	186 s	190 s	210.61 s
10x10	230 s	225 s	229 s	248 s	250 s	236.39 s

Note that if the convergence time was more than 5400 s, then the algorithm was stopped. This is the cases where the algorithm simply did not converge. In those situations it will be better to restart the algorithm and hope that the new run converge. These situations can arise because the network needs more information about the environment (it needs to explore more). But since we decrease the ϵ value the exploration phase is limited.
The average was calculated form the successful runs only. The general observation that you can make from Table 2.9, is that the convergence time does not seem to increase much as the grid size increase.

It was also possible to solve the game for a grid size of 128x128. This took about 5380 s which is about 1.5 hours. Check the run information in this Neptune-link. Note that many of the other algorithms would not have the ability to solve the game so quickly. This shows how the expert policy helps the DQfD algorithm converge faster.

2.2.6 Comparison between the algorithms

In the following tables the different algorithms can be compared in terms of execution time. It would also be possible to compare the algorithms on other parameters such as number of action calls to the environment. The problem with this, is that the Q-learning and value iteration algorithms rely very much on learning directly from the environment. Whereas the deep Q-learning methods utilize a replay buffer that allows us to draw the same samples multiple times (without interaction with the environment). Since drawing samples from the environment is quite efficient, convergence time becomes more important in our case.

Table 2.10: Comparison of convergence time for the different algorithms

nxn	Algorithm	1. time	2. time	3. time	4. time	5. time	avg. time
2x2	**Value**	7 s	6 s	6 s	7 s	6 s	**6.49 s**
	1. Q-learn with p	10 s	7 s	6 s	5 s	7 s	6.85 s
	2. Q-learn	6 s	4 s	5 s	4 s	6 s	5.07 s
	Deep Q	89 s	91 s	92 s	101 s	94 s	93.30 s
	DQfD	90 s	48 s	105 s	72 s	108 s	84.54 s
3x3	Value	216 s	105 s	158 s	153 s	151 s	156.48 s
	1. Q-learn with p	185 s	223 s	256 s	135 s	133 s	186.28 s
	2. Q-learn	70 s	67 s	75 s	68 s	73 s	**70.58 s**
	Deep Q	1267 s	1016 s	1876 s	916 s	1096 s	1234.28 s
	DQfD	417 s	518 s	427 s	381 s	510 s	451.70 s
4x4	Value	828 s	716 s	1450 s	1448 s	1530 s	1194.49 s
	1. Q-learn with p	1911 s	3016 s	2092 s	2520 s	3376 s	2583.22 s
	2. Q-learn	1077 s	865 s	644 s	856 s	640 s	816.42 s
	Deep Q	1754 s	1756 s	1403 s	1785 s	1919 s	1723.50 s
	DQfD	474 s	481 s	505 s	506 s	384 s	**470.00 s**
5x5	Value	2530 s	1202 s	2841 s	1574 s	1752 s	1979.91 s
	1. Q-learn with p	4221 s	4390 s	4690 s	7806 s	5227 s	5266.83 s
	2. Q-learn	431 s	571 s	480 s	471 s	532 s	**496.76 s**
	Deep Q	2394 s	3903 s	2169 s	3089 s	3732 s	3057.18 s
	DQfD	661 s	548 s	5400 s	5400 s	1605 s	937.87 s
6x6	Value	2485 s	3886 s	3691 s	3100 s	4592 s	3550.90 s
	1. Q-learn with p	16829 s	10793 s	10526 s	12166 s	14747 s	13012.26 s
	2. Q-learn	2509 s	2200 s	2056 s	2118 s	2342 s	2245.25 s
	Deep Q	4507 s	2651 s	4408 s	5543 s	3176 s	4057.13 s
	DQfD	232 s	5400 s	135 s	5400 s	5400 s	**183.24 s**
7x7	Value	8414 s	6787 s	9825 s	7452 s	11333 s	8761.96 s
	1. Q-learn with p	20916 s	20879 s	27291 s	41944 s	45292 s	31264.50 s
	2. Q-learn	8279 s	8917 s	9311 s	9582 s	9391 s	9095.89 s
	Deep Q	4297 s	4183 s	2822 s	2275 s	4970 s	3709.47 s
	DQfD	139 s	185 s	139 s	139 s	143 s	**148.96 s**

Table 2.11: Comparison of convergence time for the different algorithms

nxn	Algorithm	1. time	2. time	3. time	4. time	5. time	avg. time
8x8	Value	25436 s	25655 s	25097 s	25624 s	25255 s	25413.56 s
	1. Q-learn with p	42696 s	68643 s	68137 s	49803 s	72001 s	60255.97 s
	2. Q-learn	26965 s	31283 s	34215 s	32942 s	31482 s	31377.34 s
	Deep Q	7110 s	6764 s	4750 s	2686 s	4105 s	5082.81 s
	DQfD	158 s	178 s	199 s	156 s	154 s	**169.06 s**
9x9	Value	123028 s	122349 s	117213 s	116085 s	122812 s	120297.48 s
	1. Q-learn with p	...	...	...	...	...	...
	2. Q-learn	111001 s	131986 s	124153 s	138624 s	117079 s	124568.67 s
	Deep Q	4944 s	5899 s	3539 s	4980 s	6037 s	5080.02 s
	DQfD	188 s	265 s	224 s	186 s	190 s	**210.61 s**
10x10	Value	...	...	...	...	...	...
	1. Q-learn with p	...	...	...	...	...	...
	2. Q-learn	...	...	...	...	...	...
	Deep Q	5229 s	3280 s	3942 s	3513 s	4390 s	4070.75 s
	DQfD	230 s	225 s	229 s	248 s	250 s	**236.39 s**

It might not be totally fair to compare these convergence times directly. The methods of 2. Q-learn, Deep Q and DQfD has been trained with optimized parameters. Here we utilized Optuna to get a good idea of what the parameters should be set as. This process takes time and resources that is not given to the Value and 1. Q-learn methods. Secondly these implementations utilize different amount of computing resources. The Value and 1. Q-learn methods can be optimized more if we had implemented sampling in parallel (sampling from multiple environments). This would generate more samples in less time, and therefore speed up our convergence of these methods. Thirdly an unfair advantage is given to the "deep" approximation methods. These are determined to have solved the game after 450 successfully played games. Whereas the Q learning and value iteration algorithms are expected to consecutively solve 2000 games.

A few insights can be seen from the tables. Firstly it can for the Q-learning and value iterations methods be seen that the convergence time increase fast when the grid size increase. This means that the larger environments takes way more time to solve. Secondly it can be observed that the approximation methods of Deep-Q learning and DQfD can solve the larger environments in a time-frame that is more acceptable.
Another observation is that the the Q-learning and value iterations methods are quite fast at solving the very small environment: (grid size of 2, 3, 4 and 5). Of these methods it seems that the 2. Q-learning method (that utilize a decreasing step size α) is the fastest at solving the game.
Of the approximation methods it is clear that the DQfD method is the fastest to converge. But this method sometimes had issues with convergence. The deep Q method did not have these issues.
Another key observation is that the faster algorithms have more hyperparameters. These parameters need to be tuned so that the algorithms can converge (and hopefully as fast as possible).

Bibliography

[1] user43158. Gronwall's lemma (discrete version). Mathematics Stack Exchange. Accessed: 2023-12-20. URL: https://math.stackexchange.com/q/325565.

[2] Strauch Claudia. Lecture notes reinforcement learning, 2023.

[3] From Wikipedia. Lipschitz continuity, 2023. Accessed: 2024-01-04. URL: https://en.wikipedia.org/wiki/Lipschitz_continuity.

[4] B Bharath and VS Borkar. Stochastic approximation algorithms: Overview and recent trends. *Sadhana (Bangalore)*, 24(4-5):425–452, 1999.

[5] Strauch Claudia. Q learning (slides), 2023.

[6] Lapan Maxim. *Deep Reinforcement Learning Hands-On - Second Edition*. Packt Publishing, 2020.

[7] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning : an introduction*. Adaptive computation and machine learning series. The MIT Press, Cambridge, MA, second edition. edition, 2018.

[8] From Wikipedia. Harmonic series (mathematics), 2023. Accessed: 2023-12-27. URL: https://en.wikipedia.org/wiki/Harmonic_series_(mathematics).

[9] From Wikipedia. Basel problem, 2023. Accessed: 2023-12-27. URL: https://en.wikipedia.org/wiki/Basel_problem.

[10] Ravichandiran Sudharsan. *Deep Reinforcement Learning with Python - Second Edition*. Packt Publishing, 2020.

[11] Hester Todd, Vecerik Matej, Pietquin Olivier, Lanctot Marc, Schaul Tom, Piot Bilal, Horgan Dan, Quan John, Sendonaris Andrew, Dulac-Arnold Gabriel, Osband Ian, Agapiou John, Leibo Joel Z., and Gruslys Audrunas. Deep q-learning from demonstrations, 2017. arXiv:1704.03732.

[12] Sergio Martínez-Losa Del Rincón. Unofficial LaTeX template for reports/books/thesis with corporate logos of Universidad de Zaragoza with a beautiful look and feel. https://github.com/sergiomtzlosa/latex-template-report-unizar, 2021.