Report Project 1 - Navigation

Part I - Theory

The agent of the project is based on the DQN agent introduced in the paper Human-level control through deep reinforcement learning. A brief description of the algorithm as well as a brief introduction into the general reinforcement learning framework, both largely based on the book Reinforcement Learning An Introduction, Second Edition, are given in the following section.

Reinforcement Learning

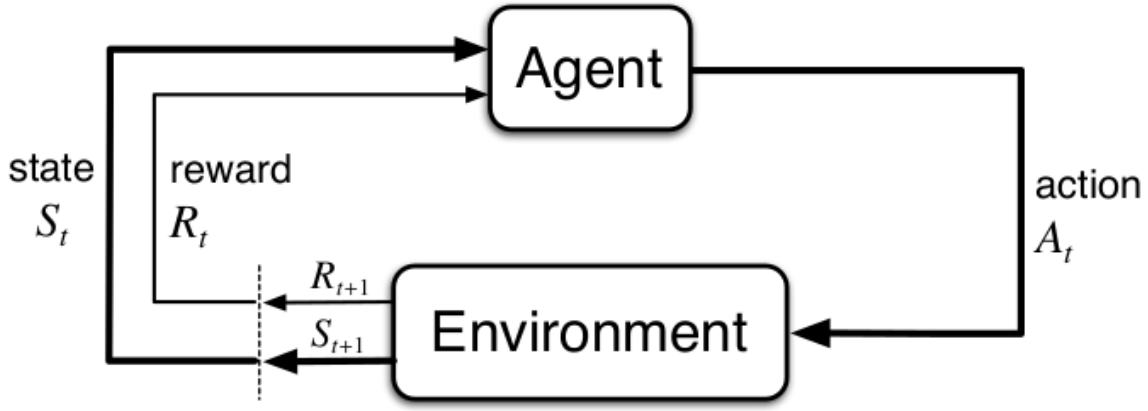
Reinforcement Learning is a computational framework to exploring and implementing goal-directed learning and decision making. It is different from other machine learning approaches like supervised learning and unsupervised learning in that the emphasis is put on an agent learning from direct interaction with its environment over sequential time steps, without requiring explicit supervision or fully fleshed out models of the environment.

The formal framework used in Reinforcement Learning to the define the interaction between an agent and its environment in terms of states, actions and rewards is the framework of Markov decision processes (MDPs). The simplicity of the framework captures the essential features of the problem of learning by an agent over time.

A **Markov decision process** (MDP) is defined as a tuple of the following components

- A **state space** $\mathbb{S}$ of states S_t of the agent.
- An **action space** $\mathbb{A}$ of actions A_t that the agent can take in the environment.
- A **transition model** $\mathbb{P}(S_{t+1}, R_{t+1}|S_t, A_t)$ that defines the distribution of states that the agent can land on and the rewards that it can obtain (S_{t+1}, R_{t+1}) by taking an action A_t in state S_t.

An agent at time t in a certain **state** S_t is interacting with the environment by performing some **action** A_t. As a result of this action the agent receives a **reward** R_{t+1} from the environment and transitions to a new state S_{t+1}.



The objective of the agent is to maximize the **expected total sum of discounted rewards** that it can receive by interacting with the environment, $\mathbb{E}(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots)$, where $0 < \gamma < 1$ is a discount factor that mainly serves two purposes: It takes account of the intuitive notion that future rewards should be less interesting to the agent than rewards closer to the present, as they are more uncertain and it ensures that the result of the (possibly infinite) sum is finite.

A sequence of states, rewards and actions is called a **trajectory**, that the agent induces by interacting with the environment in a certain manner. It is represented by a sequence of tuples $\tau = \{(S_0, A_0, R_1), (S_1, A_1, R_2), \ldots, (S_t, A_t, R_{t+1})\}$.

If what the agent is tasked to learn always gives finite trajectories, the task is called **episodic**. In contrast, if the task goes on indefinitely, it is called **continuous**. The task in the Navigation project is episodic as the length of an episode (the maximum length of any trajectory) is 300 steps.

A solution to a Reinforcement Learning problem consists of a **policy** π, which is a mapping from the current state S_t the agent is in to an action A_t the agent chooses when following the policy.

A **deterministic** policy is a mapping $\pi : \mathbb{S} \to \mathbb{A}$ that returns an action A_t for a given state S_t, $\pi(S_t) = A_t$.

If the idea is to have the agent choose probabilistically in state S_t from a given set of actions, assigning a probability to each possible action A_t, one can define a stochastic policy.

A **stochastic** policy is a mapping $\pi : \mathbb{S} \times \mathbb{A} \to \mathbb{R}$ that returns a probability distribution over all possible actions A_t in state S_t, $A_t \sim \pi(\cdot|S_t)$.

When searching for a stochastic policy, the reinforcement learning objective can be defined as **finding a policy π that maximizes the expected discounted sum of rewards**,

$$\pi^* = \arg\max_\pi \mathbb{E}_\pi(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots)$$

To extend the notation and make it more compact, we introduce additional concepts.

The **return** G_t is defined as the discounted sum of rewards obtained over a trajectory from time step t onwards,

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Thus the learning objective can be concisely be formulated as

$$\pi^* = \arg\max_\pi \mathbb{E}_\pi(G_t)$$

The **state-value function** $v_\pi(s)$ is defined as the expected return that an agent can get if the agent starts in state $S_t = s$ and then follows policy π,

$$v_\pi(s) = \mathbb{E}_\pi(G_t|S_t = s)$$

Intuitively speaking, the state-value function gives an indication on how good a certain state is if we follow a specific policy.

The **action-value** function $q_\pi(s, a)$ is defined as the expected return that an agent can get if the agent starts in state $S_t = s$, takes an action $A_t = a$ and then follows the policy π,

$$q_\pi(s, a) = \mathbb{E}_\pi(G_t|S_t = s, A_t = a)$$

Again intuitively speaking, the action-value function gives an indication of how good a certain action is if we apply it in a certain state and if we are following a specific policy.

Reinforcement Learning Solution Methods

There are several methods to solve the reinforcement learning problem. Based on the course material we will follow a value-based approach, in which we will try to obtain the optimal action-value function q^*.

Value-based methods are based on the **Bellman Equations**, which specify what conditions the optimal state-value and action-value functions have to satisfy in order to be optimal. The **Bellman Optimality Equation** for q^* is

$$
\begin{aligned}
q^*(s,a) &= \mathbb{E}(R_{t+1} + \gamma \max_{a'} q^*(S_{t+1}, a') | S_t = s, A_t = a) \\
&= \sum_{s',r} \mathbb{P}(s', r | s, a) \left(r + \gamma \max_{a'} q^*(s', a') \right) \quad .
\end{aligned}
$$

The fixed-point solution of the equation can be computed exactly using *Dynamic Programming* (if the environment is fully known) or approximately with *Monte Carlo* and *Temporal Difference* methods (if the environment is not fully known and cannot be modelled).

Tabular Q-learning

The method used in this project falls under the concept of Q-learning, which is a model-free *temporal difference* method that recovers q^* from experiences using the update rule

$$
Q(S_t, A_t) \leftarrow \underbrace{Q(S_t, A_t)}_{\text{current estimate}} + \alpha(\underbrace{R_{t+1} + \gamma \max_a Q(S_{t+1}, a)}_{\text{better estimate}} - Q(S_t, A_t)).
$$

It is an off-policy temporal difference control algorithm.

The learned action-value function Q directly approximates q^*, the optimal action-value function, independent of the policy being followed. The policy still has an effect in that it determines which state-action pairs are visited and updated. However, all that is required for correct convergence is that all pairs are visited and updated. Under this assumption and some additional technical assumptions, Q has been shown to converge with probability 1 to q^*.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Deep Q-learning

Deep Q-learning (see Human-level control through deep reinforcement learning) parametrizes $Q(S, A, \theta_i)$ using a deep convolutional neural network, where θ_i are the weights of the Q-network at iteration i. Prior to the discovery of Deep Q-learning, using a non-linear function approximator such as a neural network was known to make Q-learning unstable or cause it to diverge.

The instability had three causes:

- autocorrelation present in the sequence of observations
- the fact that small updates to Q may significantly change the used policy
- the correlations between the action-values Q and the target values $R + \gamma \max_{a'} Q(S', a')$

Deep Q-learning addresses these issues by introducing two central ideas:

- a mechanism termed experience replay that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution

- an iterative update method that adjusts the action-values Q towards target values that are only periodically updated, thereby reducing correlations with the target

To perform experience replay the agent's experiences $E_t = (S_t, A_t, R_t, S_{t+1})$ at each time-step t are stored in a data set $D_t = E_1, \ldots, E_t$. During learning, Q-learning updates are applied to samples/mini-batches of experience $(S, A, R, S') \sim U(D)$, drawn uniformly from the pool of stored samples. The Q-learning update at iteration i uses the loss function

$$L_i(\theta_i) = \mathbb{E}\left(\left(R + \gamma \max_{a'} Q(S', a', \theta_i^-) - Q(S, a, \theta_i)\right)^2\right)$$

where θ_i are the weights of the Q-network at iteration i and where θ_i^- are the weights used to compute the target at iteration i. The target network weights θ_i^- are only updated with the Q-network weights θ_i every C steps and are held fixed between individual updates.

Double Q-learning

The method was published in the paper Deep Reinforcement Learning with Double Q-learning building on the idea that in some stochastic environments Q-learning tends to perform poorly. This poor performance is caused by large overestimations of action values, slowing the learning. These overestimations result from a positive bias that is introduced because Q-learning uses the maximum action value as an approximation for the maximum expected action value. The Double Q-learning method introduced an alternative way to approximate the maximum expected value for any set of random variables. The obtained double estimator method sometimes underestimates rather than overestimates the maximum expected value.

Two separate value functions are trained in a symmetric way using separate action-value functions, Q^A and Q^B. The double Q-learning update is then given by

$$Q^A(S_t, A_t) \leftarrow Q^A(S_t, A_t) + \alpha\left(R_{t+1} + \gamma Q^B(S_{t+1}, \operatorname*{argmax}_a Q_t^A(S_{t+1}, a)) - Q^A(S_t, A_t)\right)$$
$$Q^B(S_t, A_t) \leftarrow Q^B(S_t, A_t) + \alpha\left(R_{t+1} + \gamma Q^A(S_{t+1}, \operatorname*{argmax}_a Q_t^B(S_{t+1}, a)) - Q^B(S_t, A_t)\right)$$

Double Q-learning sometimes underestimates the action values, but does not suffer from the overestimation bias that Q-learning does. In a roulette
game and a maze problem in the original paper introducing the method, Double Q-learning was shown to reach good performance levels much more quickly.

Dueling Q-Networks

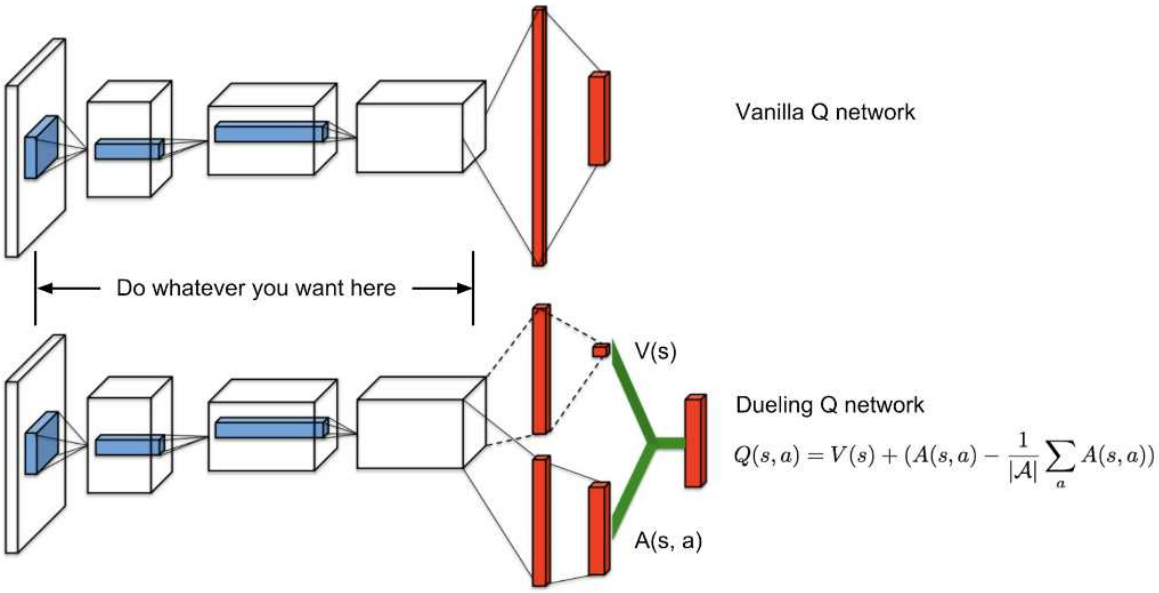
The method was published in the paper Dueling Network Architectures for Deep Reinforcement Learning. The action-value function $q_\pi(s, a)$ and the value function $v_\pi(s)$ are defined as above. Additionally the description of a Dueling Q-Network necessitates the definition of the advantage function

$$a_\pi(s, a) = q_\pi(s, a) - v_\pi(s).$$

As q_π intuitively represents the value of choosing a specific action a in a given state s and v_π represents the value of a given state s regardless of the action taken, a_π is a measure of how advantageous it is to select a specific action a in a given state s relative to choosing any of the other actions.

The dueling architecture explicitly separates the representation of the value function v_π and the state-dependent action advantage function a_π via two separate streams. For some reinforcement learning problems it is unnecessary to know the value of each action at every timestep. By separating the two estimators, the dueling architecture can learn which states are or are not valuable, without having to learn the effect of each action for each state.

The architecture is especially relevant for tasks where actions might not always affect affect the environment in meaningful ways.



The straightforward way to aggregate the advantage values and state values seems to be

$$Q(S, A, \theta_i, \alpha, \beta) = V(S, \theta_i, \beta) + A(S, A, \theta_i, \alpha)$$

where the value function V is parameterized with the parameter β and where the advantage function A is parameterized by the parameter α. However, the naive sum of the two functions is unidentifiable in that given Q, we cannot recover V and A uniquely. This lack of identifiability leads empirically to poor practical performance.

Therefore, the last module of the neural network implements the forward mapping

$$Q(S, A, \theta_i, \alpha, \beta) = V(S, \theta_i, \beta) + \left(A(S, A, \theta_i, \alpha) - \max_{a \in |\mathbb{A}|} A(S, a, \theta_i, \alpha)\right)$$

to force the advantage function estimator to have zero advantage at the chosen action.

Alternatively, the mapping

$$Q(S, A, \theta_i, \alpha, \beta) = V(S, \theta_i, \beta) + \left(A(S, A, \theta_i, \alpha) - \frac{1}{|\mathbb{A}|}\sum_a A(S, a, \theta_i, \alpha)\right)$$

can be used with the action chosen via $a^* = \text{argmax}_{a \in \mathbb{A}} Q(S, a, \theta_i, \alpha, \beta)$.

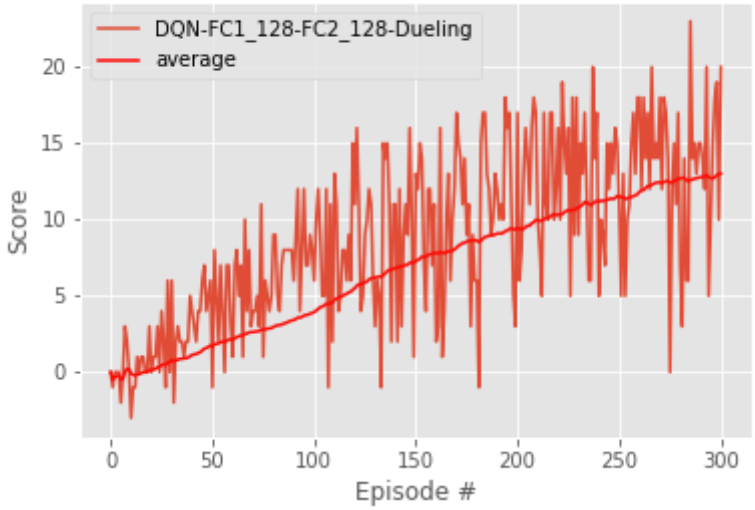
Part II - Practice

In this project, agents of varying complexity were trained and evaluated in the supplied environment. The performance of the agents is measured by the fewest number of episodes required to solve the environment.

The best model in this environment turned out to be a model with 128 units in both the FC1 and FC2 layer and using Dueling Q-Learning. In this context it is interesting that the supplied environment is similar to the one mentioned in the paper Dueling Network Architectures for Deep Reinforcement Learning, the environment of the Atari game Enduro, in which the dueling architecture also showed promise.

FC1 Units	FC2 Units	Use Double Q-Learning	Use Dueling Q-Learning	Number of Episodes	Average Score
64	64	TRUE	TRUE	360	13.04
64	64	TRUE	FALSE	252	13.05
64	64	FALSE	TRUE	279	13.05
64	64	FALSE	FALSE	249	13.02
64	128	TRUE	TRUE	313	13.04
64	128	TRUE	FALSE	315	13.01
64	128	FALSE	TRUE	300	13.02
64	128	FALSE	FALSE	343	13.01
128	64	TRUE	TRUE	231	13.02
128	64	TRUE	FALSE	346	13.01
128	64	FALSE	TRUE	303	13.03
128	64	FALSE	FALSE	314	13.06
128	128	TRUE	TRUE	233	13.07
128	128	TRUE	FALSE	298	13.04
128	128	FALSE	TRUE	201	13.01
128	128	FALSE	FALSE	310	13.01

Below we show the trajectory of the best model, the trajectories for other models can be found in the sub-directory images of the project folder. The checkpoints can be found in the sub-directory pth_checkpoints.



Part III - Improvements

- **Improve model evaluation**: All model configurations were evaluated on a single run of each model. To ensure a more stringent evaluation, each model configuration should be run multiple times and evaluated on the basis of these samples.
- **Add prioritized experience replay**: Prioritized experience replay (see Prioritized Experience Replay) does not sample experience transitions uniformly from replay memory, but prioritizes important transitions by sampling them more frequently.
- **Implement Noisy DQN**: Implement the method described in Noisy Networks for Exploration, which adds parametric noise to the weights to induce stochasticity to the agent's policy, yielding better performing exploration.