UNIVERSITY OF OULU
FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

**Patrick Sunden**

# Q-LEARNING AND DEEP Q-LEARNING IN OPENAI GYM CARTPOLE CLASSIC CONTROL ENVIRONMENT

Bachelor's Thesis
Degree Programme in Computer Science and Engineering
March 2022

**Sunden P. (2022) Q-learning and Deep Q-learning in OpenAI Gym CartPole Classic Control Environment.** University of Oulu, Degree Programme in Computer Science and Engineering, 32p.

## ABSTRACT

**This thesis focuses on the basics of reinforcement learning and the implementation of Deep Q-learning, also referred to as Deep Q-network, to emphasize the artificial neural network, and Q-learning to the CartPole-v0 classic control learning environment. This work also presents the idea of a Markov Decision process, standard algorithms, and some basic information about the OpenAI Gym toolkit. DQN is a deep learning version of regular Q-learning, the crucial difference being the use of a neural network and experience replay. Cartpole-v0 can be considered an easy learning problem, especially for DQN, since the number of states and specific actions is relatively low. The learning results between Q-learning and DQN were examined by comparing the convergence and stability of rewards, the cumulative reward gain, and how quickly the Cartpole-v0 learning environment was solved. While it is tough to determine which implementation solved the CartPole-v0 problem better, it can be concluded that while DQN is often seen as the more advanced and complicated version of regular Q-learning, it did not perform better than Q-learning.**

**Keywords: Q-learning, deep Q-learning, DQN, DQL, OpenAI, reinforcement learning, machine learning, MDP, CartPole**

**Sunden P. (2022) Q-oppiminen ja syvä Q-oppiminen OpenAI Gym CartPole-säätöympäristössä.** Oulun Yliopisto, Tietotekniikan tutkinto-ohjelma, 32p.

# TIIVISTELMÄ

**Tämä työ keskittyy esittelemään vahvistusoppimisen perusteita, sekä vertailemaan oppimista Q-oppimisen ja syvän Q-oppimisen välillä CartPole-v0 säätöympäristössä. Työ käsittelee myös Markovin päätöksentekoprosessia ja niissä käytettäviä algoritmeja. Tärkein ero syvän Q-oppimisen ja Q-oppimisen välillä on se, että syvä Q-oppiminen käyttää neuroverkkoa ja muistista oppimista tavallisen Q-oppimisessa käytetyn Q-taulukon sijaan. CartPole-v0 oppimisympäristöä voidaan pitää helppona oppimisympäristönä erityisesti syvä Q-oppimiselle, sillä CartPole-oppimisympäristössä mahdollisten tilojen määrä on verrattain pieni. Oppimista implementaatioiden välillä vertailtiin tarkastelemalla palkintojen suppenemista ja vakautta, palkintojen kumulatiivista arvoa ja oppimisympäristön ratkaisunopeutta. Syvää Q-oppimista pidetään tavallisen Q-oppimisen monimutkaisempana muotona, ja se pärjääkin yleensä paremmin monimutkaisemmissa ympäristöissä, joissa tilojen määrä kasvaa erittäin suureksi. Etukäteen on mahdotonta sanoa, kumpi implementaatio oppii kohdeympäristön tehokkaammin. Syvä Q-oppiminen oppii vaikeita ympäristöjä paljon tehokkaammin kuin tavallinen Q-oppiminen, kun taas Q-oppiminen oppii vähätilaisia ympäristöjä tehokkaammin, koska sen ei tarvitse käyttää muistista oppimista, joka hidastaa harjoitusprosessia.**

**Avainsanat: Q-oppiminen, syvä Q-oppiminen, DQN, DQL, OpenAI, CartPole, vahvistusoppiminen, koneoppiminen, MDP**

# TABLE OF CONTENTS

# FOREWORD

I want to thank D.Sc. (Tech.) Jaakko Suutala for supervising and helping me compose this thesis.

Oulu, March 28th, 2022

Patrick Sunden

# LIST OF ABBREVIATIONS AND SYMBOLS

| | |
|---|---|
| AI | artificial intelligence |
| DDQN | double deep Q-network |
| DQL | deep Q-learning |
| DQN | deep Q-network |
| MDP | markov decision process |
| ML | machine learning |
| NLP | natural language processing |
| PER | prioritized experience replay |
| | |
| $\alpha$ | learning rate |
| $\varepsilon$ | decay coefficient |
| $\gamma$ | discount coefficient |
| $\pi$ | policy |
| $A$ | set of all actions |
| $a$ | an action |
| $P(S'\|S,A)$ | transition model |
| $r$ | a reward |
| $R(S)$ | reward function |
| $R_t$ | reward at timestep $t$ |
| $S$ | set of all states |
| $S_t$ | state at timestep $t$ |

# 1. INTRODUCTION

Artificial intelligence (AI) can be divided into multiple subfields and areas of application: machine learning (ML), expert systems, robotics, natural language processing (NLP), and speech recognition, ML and expert systems being the main subfields. ML, in general, is the study of algorithms used to solve complicated or time-consuming tasks that are hard for common programming languages to solve. Classic problems solved using ML are problems such as email spam filtering and product recommendations for customers based on what they have watched on a streaming service or bought at a specific store. ML can be implemented in multiple ways: supervised learning, unsupervised learning, and reinforcement learning (RL). The goal is to build a model that can make predictions by running training data through the algorithm. Supervised learning uses pre-labeled data so that the algorithm already knows what the input is, while unsupervised learning uses unlabeled data. [1]

RL is a problem-solving technique for machine learning where the agent explores the environment and learns from its mistakes. Sutton & Barto [2] summarize RL as follows: "Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal.". RL happens not only in virtual environments but also in our everyday lives. The rewards might come in various forms, but the principle remains the same. A toddler or a newborn animal uses RL when learning how to walk. Meanwhile, adults use RL when learning a new task, and no information about the new task is available. A positive reward is usually presented as positive numbers in a virtual environment and a negative reward as negative numbers. In real life, a negative reward could be a toddler falling on the ground as it is trying to stand on its feet, and a positive reward would be the toddler able to walk a certain distance without falling.

People have implemented RL into machines for decades. One of the most common RL algorithms is Q-learning, and its deep learning version, Deep Q-learning (DQL). A Deep Q-network (DQN) utilizes the DQL algorithm, and the Q-value evaluation used in Q-learning is replaced with a neural network, which takes a state as an input and predicts (action, Q-value) – pairs as outputs to determine the best action to take in each state [3]. The technical details of Q-learning and DQN are covered more in-depth in section 4.1.

Some important modern-day examples of RL utilization include industrial automation, finance, NLP, healthcare, and games. RL utilized in industry automation helps store and keep track of supplies in a warehouse. RL can also be used in trading and finance to determine what actions to take at specific stock or coin prices [4]. NLP is an AI area of application with the main idea to make machines able to understand text and speech as fluently as a human. NLP is used to summarize texts, dialogue generation, and question answering, all of which could be used in a general chatbot [5].

Reinforcement learning and AI are probably most known to the common public from AI bots developed into games. These games range from bots in online first-person shooters to strategic, tactical bots capable of mimicking human intuition in games like the Chinese GO. An example of a company that developed the AI for the game GO is DeepMind, founded in 2010 by a British company called DeepMind Technologies and was renamed in 2014 to Google DeepMind after being purchased by Google. DeepMind concentrates on deep learning and has developed AlphaGo, an AI specializing in the game called Go

[6]. Go is considered a challenging game for a program to learn since it requires intuitive and strategic thinking and has $10^{360}$ possible moves. AlphaGo was the first program to ever win a professional Go-player in 2016. According to [6], AlphaGo learned how to play by first observing 30 million positions from games played by humans, followed by RL, where it plays against itself repeatedly. This milestone was reached by combining Deep reinforcement learning with AlphaGo's ability to recognize successful board configurations by using a backtracking algorithm, where the algorithm checks what effects a specific move has and chooses the best action by ruling the bad actions out.

### 1.1. Research Questions and Contributions

This thesis aims to study how Q-learning and DQN perform when learning the CartPole-v0 problem provided by OpenAI Gym. The work will cover the mainline subfields of ML, including deep learning, reinforcement learning, deep reinforcement learning, and also gives insight into Markov decision processes, theory of reinforcement learning, and hyperparameter tuning. CartPole-v0 is a learning environment for training and testing different RL algorithms. The CartPole-environment includes a pole attached to a cart by an un-actuated joint. The idea is to balance the pole in an upright position by either moving the cart to the left or right. More details about the environment are presented in section 5.2.

Two programs were made for this thesis to figure out some differences in learning between Q-learning and DQN, presented in sections 4.1 and 4.2 (Algorithms 3 and 4). The first program is a simple Q-learning program that uses a table to store Q-values in (state, action) pairs. The agent uses this to determine the Q-value of an action in a particular state. The second program utilizes DQL and a neural network that takes a state as an input and predicts the (Q-value, action) pairs. More technical details about Q-learning and DQN can be found in chapter 4, and the implementations are made available in [7].

Some of the results that this thesis will be comparing between Q-learning and DQN are reward convergence, cumulative reward gain, and how quickly the algorithm solves the CartPole-v0 problem. Both Q-learning and DQN implementations should learn the CartPole-v0 environment with ease, but it is unclear whether DQN would yield better results with the help of a neural network and experience replay. In addition to these three metrics, the efficiency of the training phase will also be evaluated since it generally takes longer to train a neural network.

# 2. MARKOV DECISION PROCESS

A task that satisfies the Markov property in reinforcement learning is called a *Markov decision process (MDP).* Only finite MDPs will be dealt with in this work since they are more beneficial in RL than infinite MDPs. A finite MDP has a finite number of states and actions, whereas an infinite MDP has an infinite number of states and actions [8].

According to [2], MDPs are "a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards.". In Reinforcement learning and MDP, the agent is the decision maker that takes actions, receives rewards and interacts with the environment, which is comprised of everything the agent can interact with, such as all possible states and actions that can be taken in said states. Figure 1 shows the general interaction between the agent and the environment in an MDP. The agent starts at the initial state $S_0$ and takes an action *a,* receives a reward $R_t$ and lands in the state $S_t$, $S_t$ and $R_t$ being the state and reward at timestep t. The agent takes another action *a* while in state $S_t$, receives a reward $R_{t+1}$, and lands in the state $S_{t+1}$.

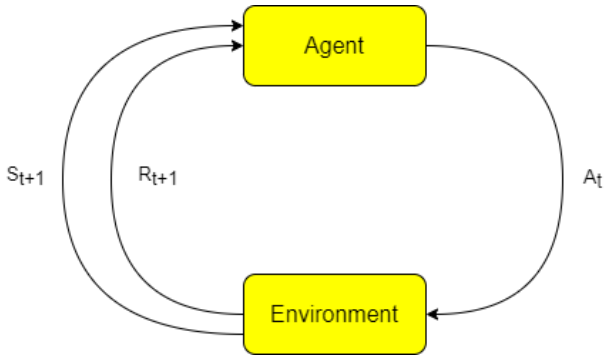


**Figure 1.** Environment-agent interaction in an MDP.

Russel and Norvig [1] define an MDP as follows:

- *S*, where $S_0$ is the initial state and *S'* is the next state if action *a* is done in state *S.*
- *A*, a finite set of actions in each state *S.*
- *P(S'|S, A),* a transition model, which describes the probability to end up in state *S'* from state *S* by taking action *A*. The transitions are Markovian if the probability of reaching *S'* depends only on state *S* and not on any of the previous states.
- *R(S),* a reward function, where the reward only depends on the current state. There are also other ways of defining the reward functions:
    - *R(S, A, S')*, early in the research of MDPs rewards were allocated to these 3-tuples, without having to know any information of different states,
    - *R(S, A),* information about rewards can also be stored on 2-tuples that have information about the current state and the action taken from the current state.

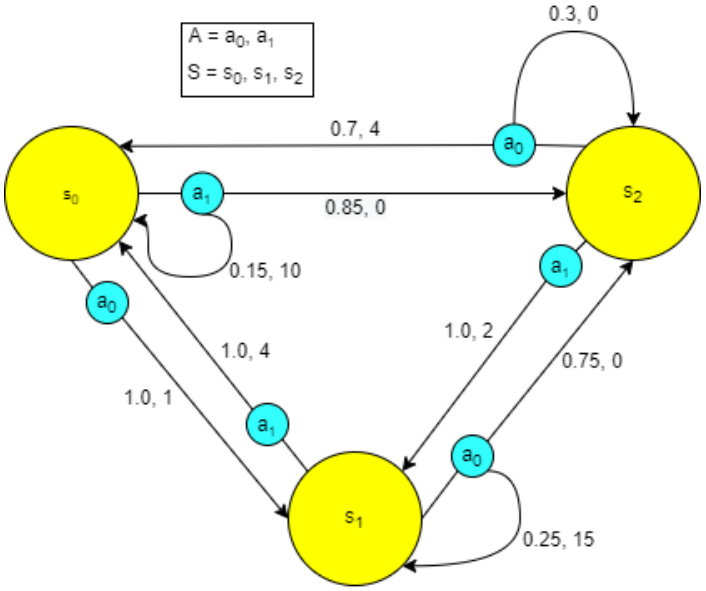


**Figure 2.** State transition graph of an MDP.

Figure 2 is an example of a state transition graph of an MDP, which shows the transition probabilities, rewards, and state transitions. The agent starts at state $S_0$ and has two possible actions $a_0$ and $a_1$. If the agent chose to take the action $a_0$, it would end up in state $S_1$ with a probability of 1.0 and receive a reward of 1. On the other hand, if the agent chose action $a_1$ in state $S_0$, there would be a 15% probability of ending up in the same state $S_0$ and receiving a reward of 10, and an 85% chance to land in state $S_2$ and receiving a reward of 0.

A solution to an MDP is called a policy and is traditionally denoted by the symbol π. A policy is simply a set of rules that the agent follows in the environment. When the agent learns enough information about every state in the environment, the policy becomes a complete policy, which means that the agent knows which action to take in every single state. The optimal solution to an MDP is called an *optimal policy* and is denoted with the symbol $\pi^*$. The difference between optimal policy and complete policy is that optimal policy is always complete, but a complete policy is not always optimal. A complete policy always leads the agent to the goal state, but might not yield the maximum reward available, whereas the optimal policy will.

A Markov decision process is an extension of a Markov chain, which will be explained in more detail in section 2.3. In addition to states and probabilistic transitions, which are critical factors in a Markov chain, MDPs include the choice of making actions without having to depend on probabilities, and rewards, which help in choosing the correct action [1].

### 2.1. Markov Property

The Markov property is named after a Russian mathematician Andrey Markov, known for his studies on stochastic processes, also known as random processes. Ibe [9] defines a stochastic process as a "family of random variables: { X(t, w) |t ∈ T, w ∈ Ω}, defined over

a given probability space and indexed by the time parameter *t*. Later in his career, Markov became known as the creator of Markov chains and Markov processes.

Markov property means that given the present state of the process, the future state is independent of the past. In all its simplicity, this means that the progress of the Markov process in the future only depends on the present state and does not depend on the past [1].

### 2.2. Markov Process

Generally, if a continuous stochastic process satisfies the Markov property, it is called a Markov process. The most straightforward Markov process is the first-order Markov process, in which the current state only depends on the previous state. There are also other Markov processes, such as second-, and third-order Markov processes. The difference between a first-order Markov process and a second-order Markov process is that the current state in the first-order Markov process only depends on the previous state, whereas in the second-order Markov process, it depends on the two previous states [1 p. 568].

### 2.3. Markov Chain

Serfozo [8] defines Markov chain as follows: A stochastic process $X = \{X_n : n \geq 0\}$ on a countable set S is a Markov chain if, for any $i, j \in S$ and $n \geq 0$,

$$P\{X_n + 1 = j | X_0, \dots, X_n\} = P\{X_n + 1 = j | X_n\}, \quad (1)$$

$$P\{X_n + 1 = j | X_n = i\} = p_{ij}. \quad (2)$$

The $p_{ij}$ is the probability that the Markov chain jumps from state i to state j. These transition probabilities satisfy $j \in S\ p_{ij} = 1$, $i \in S$, and the matrix $P = (p_{ij})$ is the transition matrix of the chain [10].

Equation (1) simply denotes the Markovian property, which was presented earlier in the thesis, and Equation (2) says that the transition probabilities do not depend on the time parameter n. One of the essential rules of a Markov chain is that the transition probabilities remain the same, and the probabilities sum up to 1. A Markov chain is a discrete-time process, which satisfies the Markov property, and it generally experiences transitions between states according to pre-set probabilistic rules [8].

## 2.4. Variants of solving MDPs

Value iteration and policy iteration are probably the most popular methods used to solve an MDP. Both algorithms are used to find the optimal policy for the environment.

### *2.4.1. Policy Iteration*

Policy iteration consists of policy evaluation and policy improvement:

**Algorithm 1.** Policy Evaluation and Policy Improvement

$Initialize\ V(s) \in R\ and\ \pi(s) \in A(s)\ arbitrarily\ for\ all\ s \in S$

*#Policy Evaluation*
Loop:
$\Delta \leftarrow 0$
Loop for each $s \in S$:
$v \leftarrow V(s)$
$V(s) \leftarrow \sum s', r\ p(s', r|s, \pi(s))[r(s, \pi(s), s') + \varphi V(s')]$
$\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta <$ $\Theta$(a small positive number determining the accuracy of estimation)

*#Policy Improvement*

$policy\_stable \leftarrow true$
For each $s \in S$:
$old\_action \leftarrow \pi(s)$
$\pi(s) \leftarrow argmax_a \sum s', r\ p(s', r|s, a)[r + \varphi V(s')]$
If $old_{action} \neq \pi(s)$, then $policy_{stable} \leftarrow false$
If $policy_{stable}$, then stop and return $V \approx v_* \leftarrow$ and $\pi \approx \pi_*$; else go to 2

Algorithm 1 repeats policy evaluation and policy improvement until convergence. Policy evaluation includes taking the policy $\pi$ and evaluating its value function, and once the value function of the policy is evaluated, the best action for the current value function is calculated using Eq. 3:

$$\pi(s) \leftarrow argmax_a \sum s', r\ p(s', r|s, a)[r + \varphi V(s')]. \quad (3)$$

Policy iteration undoubtedly converges since finite Markov Decision Processes have a finite number of states, actions, policies, and the policy is improved in every iteration of Policy Iteration. [2]

### *2.4.2. Value Iteration*

Value Iteration, presented in Algorithm 2, is a way to compute the optimal policy by iteratively computing the values of states using the Bellman update equation until convergence [2]. Initially, the values of the states are set to zero and are iteratively computed by taking actions and landing in terminal states with either a positive or a negative reward.

**Algorithm 2.** Value Iteration

Algorithm parameter: a small threshold $\Theta > 0$ determining accuracy of estimation
Initialize V (s) arbitrarily, except V (terminal);
V(terminal) = 0

Loop:
$\Delta \leftarrow 0$
Loop for each s $\in$ S:
$v \leftarrow V(s)$
$V(s) \leftarrow max_a \sum_{s',r} p(s', r|s, a)[r + \varphi V(s')]$
$\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \Theta$
*Output a deterministic policy* $\pi \approx \pi_*$, *such that*
$\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \varphi V(s')]$

According to [2], Value Iteration combines policy improvement, and the truncated policy evaluation step into a single equation:

$$V_{k+1}(s) \leftarrow max_a \sum_{s',r} p(s', r|s, a)[r + \varphi V(s')]. \quad (4)$$

Equation 4 is essentially Policy Iteration that is stopped after updating each state.

# 3. REINFORCEMENT LEARNING

According to Sutton & Barto [2], we came to know artificial intelligence as it is in late 1979. Reinforcement learning was described as "An idea of a learning system that wants something, that adapts its behavior in order to maximize a special signal from its environment." [2]. In the early 1980s, reinforcement learning was studied vastly in the psychology of animal learning. Both animal learning and RL share key attributes, such as receiving positive rewards when reaching the goal and receiving negative rewards when failing. A reward could be a positive integer in RL because the agent had been told to maximize the rewards. Then again, in animal learning, a reward would be a treat for successfully completing a task [2].

Sutton & Barto [2] identify four main elements of a reinforcement learning system: "a *policy*, a *reward signal*, a *value function*, and optionally a *model* of the environment.". As mentioned earlier in chapter 2, a policy is a solution to an MDP, and it "defines the learning agent's way of behaving at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states." [2].

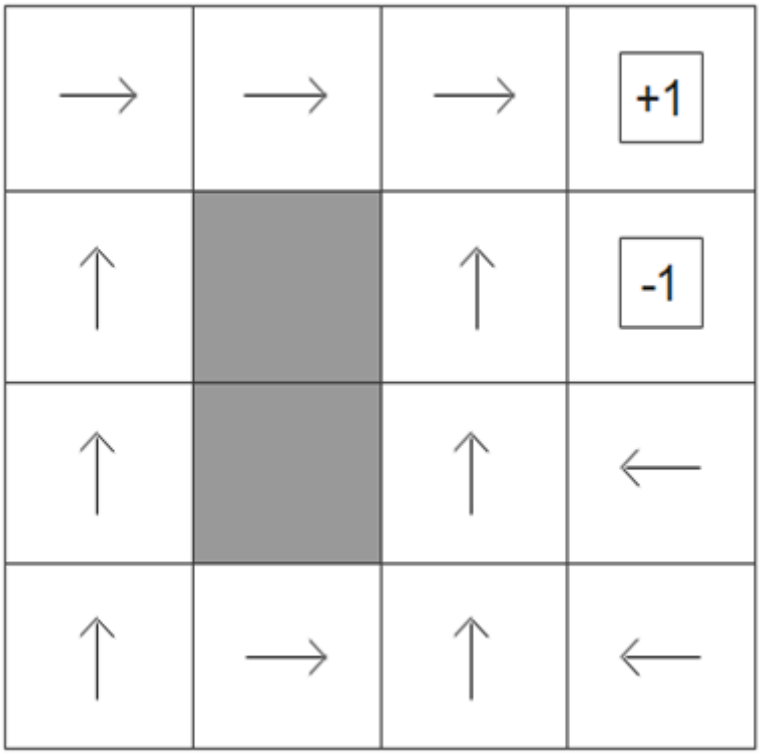


**Figure 3.** An example of a policy.

Figure 3 visualizes what a policy would essentially look like. The agent knows where to move in every state because it has already mapped the optimal path to the positive terminal state.

The difference between a reward signal and a value function is that reward is something the agent gets immediately from the environment after performing an action, while the value function specifies the rewards the agent might get in the long run. The reward tells the agent which states are good and bad. Most policy altering happens via reward signaling,

although if "a low reward follows the action selected by the policy, then the policy may be changed to select some other action in that situation in the future" [2]. The value function defines the value of a particular state, which is how much the agent could accumulate rewards in the future from that state. A good example was presented in [2 p. 6], where a state can have a low reward but still a very high value if the states followed by this state have high rewards, which means that the agent will receive higher rewards in the future.

The final element presented in [2] is a *model* of the environment. Not every RL system has a model; this leads to model-free and model-based systems. The key distinction between model-free and model-based algorithms is that while model-free algorithms do not make any predictions regarding future states, model-based algorithms such as policy iteration and value iteration use the model's prediction of the next state and reward to predict the next optimal action. Model-free learning can often be considered trial-and-error learning because the agent must try every state to know whether it yields positive or negative rewards. As presented in chapter 2, an MDP has four key elements: a set of states $S$, a set of actions $A$, a transition model $P(S'|S,A)$, and a reward function, which is most commonly denoted as *R(S)* or *R(S, A)*. If an RL agent knew all the previously mentioned four key elements of an RL problem, there would not be a problem since the agent already would know the solution to the problem. A model-free algorithm, such as Q-learning, which is presented in algorithm 3, does not try to learn the optimal policy by using or learning the transition model or the reward function regarding the environment but instead derives the optimal policy by estimating Q-values for each action in each state. Model-based algorithms such as policy iteration and value iteration, presented in algorithms 1 and 2, strive to learn a model of the environment by gathering information about the environment through observations and then estimate the MDP. Once enough information about the environment has been gathered, the model-based algorithm can begin to make predictions about optimal actions in next states by using the estimated transition model and reward function.

# 4. Q-LEARNING AND ITS VARIANTS

## 4.1. Q-learning

Q-learning is an off-policy temporal difference control algorithm that updates a simple table of Q-values for each state-action pair [2]. An off-policy learning method improves a policy that is different from the policy used for action selection. Agents generally have a target policy and a behavior policy. Target policy is the policy that the agent uses to learn from the rewards received when taking specific actions, while behavior policy is the policy that the agent uses to determine the following action in a state. Because Q-learning is an off-policy algorithm, which means it does not improve the same policy used for action selection, it can proceed to explore continuously while learning optimal policy.

**Algorithm 3.** Q-learning

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

In Algorithm 3, we can observe the Q-learning algorithm, where S, A, S' are the same as presented in chapter 2. We can also observe some new variables, such as learning rate ($\alpha$), discount factor ($\gamma$), and $max_a$, which is the maximum reward that the agent can receive after taking the optimal action in the current state. Learning rate is set between 0 and 1, where zero means that the Q-values are never updated, and nothing is learned. On the contrary, it being close to one means that the agent can learn quickly. However, the learning rate shouldn't be too high since this will result in unstable training. The discount factor is also a value between 0 and 1, where it being zero means that the agent only cares about the near future rewards, and it being close to one, it cares about long-term rewards.

The Q-learning algorithm can be simplified into six steps:

1. Initialize the Q-table containing Q-values for state-action pairs.
2. Initialize the current state S.
3. Choose a legal action for the current state S according to an action selection policy ($\varepsilon - greedy, \varepsilon - soft,\ softmax$).
4. Take the action A according to the action selection policy, observe the reward R received from the transition and observe the new state S'.
5. Update the Q-value in the Q-table using the formula shown in Algorithm 3.
6. Update the current state to the new state and repeat until the current state is the terminal state.

### 4.2. DQN

DQN is a reinforcement learning method that uses the Q-learning algorithm and a neural network to learn about the environment. Deep Q-learning can be seen as a more complicated version of Q-learning since Q-learning uses a Q-table to link the actual Q-values to state-action pairs. Simultaneously, deep Q-learning takes a state as an input, passes this through a neural network with a set of hidden layers, and gives an (action, Q-value) pair as output. Deep learning and neural networks are generally used in problems and learning projects where the environment has a large discrete or continuous state space and multiple actions in each state. An excellent example of such a learning environment is chess, which has a state-space of approximately $10^{120}$ different states, which is too much for a Q-table to handle. Instead of taking an exact value from the state-action pairs in the Q-table, a neural network is trained to estimate the value on the Q-table by changing weights between artificial neurons. The Deep Q-learning algorithm with experience replay is presented in [22] as follows:

**Algorithm 4.** Deep Q-learning with Experience Replay

$Initialize\ replay\ memory\ D\ to\ capacity\ N$
$Initialize\ action-value\ function\ Q\ with\ random\ weights$
**for** $episode = 1, M$ **do**
  $Initialise\ state\ s_t$
  **for** $t = 1, T$ **do**
    $With\ probability\ \varepsilon\ select\ a\ random\ action\ a_t$
    $otherwise\ select\ a_t = max_a Q^*(s_t, a; \theta)$
    $Execute\ action\ a_t\ and\ observe\ reward\ r_t\ and\ state\ s_{t+1}$
    $Store\ transition\ (s_t, a_t, r_t, s_{t+1})$
    $Set\ s_{t+1} = s_t$
    $Sample\ random\ minibatch\ of\ transitions\ (s_t, a_t, r_t, s_{t+1}) from\ D$
    $Set \begin{cases} r_j & , for\ terminal\ s_{t+1} \\ r_j + \gamma\ max_{a'} Q(s_{t+1}, a'; \theta), & for\ non-terminal\ s_{t+1} \end{cases}$
    $Perform\ a\ gradient\ descent\ step\ on\ (y_j - Q(s_{t+1}, a'; \theta))^2$
  **end for**
**end for**

The neural network is used to approximate the Q-value function in deep Q-learning. Steps 1-4 are somewhat like regular Q-learning, except for step 1, where instead of a Q-table being initialized, the DQL-algorithm uses a neural network to represent the action-value function. The first step of the DQL-algorithm would be to initialize the neural network and the replay memory with a capacity of N, which is used in experience replay. This is followed by initializing a state at the beginning of every episode, choosing an action according to the initialized state and epsilon, executing upon this action, and observing the reward, next state, and the Boolean variable "done." The next steps differ from the standard Q-learning algorithm. After the action has been executed, a five-tuple: (state, action, reward, next state, done) is stored into the replay memory to be used in the next step. After a required number of samples have been stored, the algorithm can start performing

experience replay, where the weights in the neural network are updated in batches. In Algorithm 4, this is referred to as gradient descent. The DQL algorithm used in this thesis uses mini-batch gradient descent, which is a combination of batch gradient descent and stochastic gradient descent. A pre-set number of samples, a batch, are chosen randomly from the replay memory to further train the neural network. This process combines the randomness of the stochastic gradient descent and the computation efficiency of the batch gradient descent. After the experience replay has finished training the network, the algorithm returns to step 2, resetting the environment and initializing a state.

### 4.3. Neural network

Artificial neural networks are biologically inspired computational devices that use a network of nodes that are somewhat based on the human neuron, to translate the input data into the desired output [11]. A basic neural network consists of layers that consist of neurons, activation function, weights, and bias. There are three different types of layers in a neural network: an input layer, hidden layer, and an output layer. As the name suggests, the input layer takes data, such as an image, audio, or text file, as an input and proceeds them to the neurons on the hidden layer. The hidden layer applies weights to the inputs and passes the inputs through an activation function that gives an output to the output layer neurons or the subsequent hidden layer neurons [2]. The output can be defined with a simple equation:

$$Output = activation\_function(weighted\ sum\ of\ inputs) \tag{5}$$

There are linear and non-linear activation functions. An excellent example of a non-linear activation function would be the sigmoid function:

$$S(x) = \frac{e^x}{e^{x+1}}. \tag{6}$$

The sigmoid function outputs a value between 0 and 1, if the input is very negative, it returns a value close to zero, and if the value is very positive, it returns a value close to one [21]. The most popular example of a linear activation function would be the Rectified Linear Unit-function (ReLU):

$$f(x) = \begin{cases} 0, & for\ x < 0 \\ x, & for\ x \geq 0 \end{cases}, \tag{7}$$

which returns a value of zero if the input is negative, or the input itself if it is positive [21]. An activation function calculates using the bias and the weighted sum of inputs whether a specific artificial neuron should be fired and if so, at what intensity.

As seen in Figure 4, in a fully connected neural network every input point is connected to every neuron in the next hidden layer, and every neuron in the first hidden layer is connected to the second hidden layer. The output of the first hidden layer becomes the input for the second hidden layer. The lines connecting these neurons in Figure 4 represent the weights that determine the strength of the connection between the neurons.

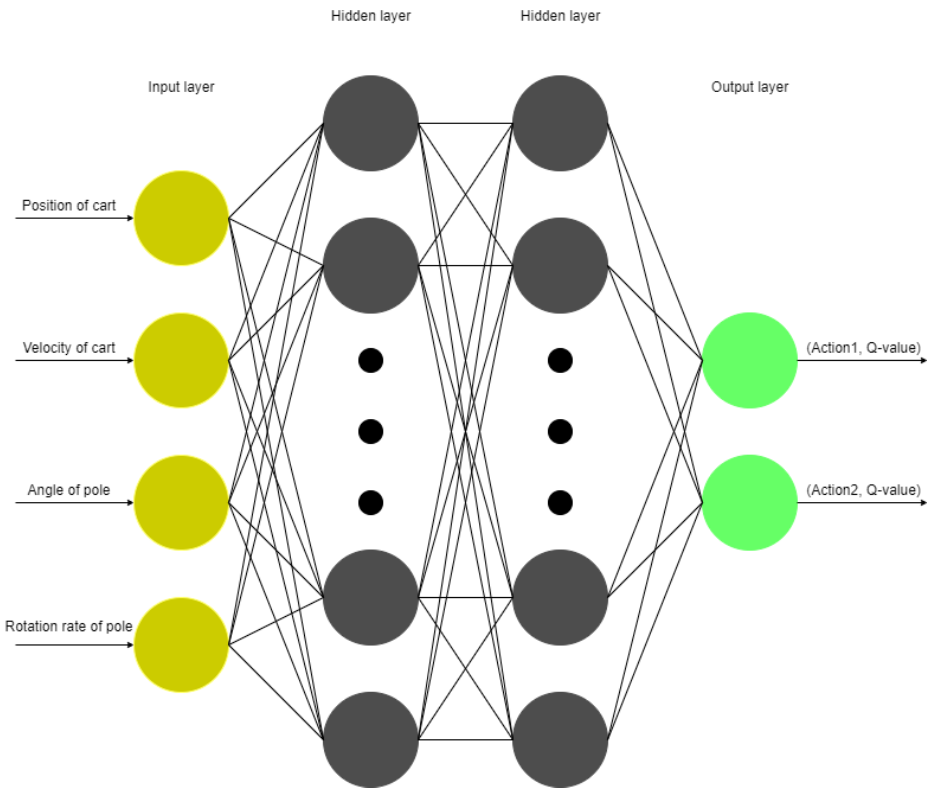


**Figure 4.** Model of the neural network used in this thesis.

The simplest way of teaching a neural network is by using supervised learning, which means processing data with known answers. This way, the neural network will learn what details are needed to construct the correct output. A picture of a rabbit can be used as an example. The neural network can be shown hundreds of pictures containing rabbits; the more variance between rabbits that are shown to the network, the better it is at figuring out whether a picture contains a rabbit or not. Feeding the network with pictures of rabbits that have different appearances helps the neural network recognize new images. Whereas if the training data is small and unvarying, there lies a risk that the network might overfit, meaning that the neural network wouldn't be able to recognize untrained data since it has become too good at recognizing the training data.

# 5. IMPLEMENTATION

## 5.1. OpenAI Gym

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching everything from walking to playing games like Pong or Pinball [12]. These games, such as Pinball, are called environments and they provide the user with the observation-space and the action-space of the environment, which are like rules that the agent must follow while learning how to perform in the selected environment. OpenAI Gym provides environments of varying difficulties. The primary purpose of OpenAI Gym is to provide a collection of different environments that share a similar interface that allows for comparison using the same algorithms [13].

Easiest and probably most used are "Classic control"-environments; this includes the CartPole-v0 – environment used in this thesis. The classic-control environments are considered easy since the action space is limited to only two actions, left and right, while the state space is relatively limited as well. Some more complicated environments are Atari, Box2D, and MuJoCo environments. Box2D-environments are like classic-control environments, but they have larger state- and action spaces.

Before the environment is available for use, the toolkit must be installed on the machine. Python version 3.5 or newer must also be installed for the gym-toolkit to work correctly. The official OpenAI Gym documentation has got simple instructions on installing the toolkit and how to create an environment. The toolkit can be installed by using the package installer for Python [14]:

```
pip install gym
```

A simple example of getting an environment running:

```
import gym
env = gym.make('the_environment_you_want_to_make')
env.reset()
for k in range(100)
   env.render()
   env.step(env.action_space.sample())
env.close()
```

This is a highly straightforward example of setting up an environment. This example does not include boundary checking or learning of any kind. Before we can do anything, we must import the gym-toolkit. After this, we create the environment itself and assign it to a variable using the "gym.make()" command. The environment can be chosen by passing the name of the environment as a parameter of the command. The rest of the commands are relatively simple, "env.reset()" resets the environment to its initial state, "env.render()" displays a popup window that renders the environment for every new action that is provided as a parameter for the function "env.step()". The "env.step()"-function takes an

action at each step and returns four parameters; observation, reward, done, and info. Observation is an object representing info about the environment, the reward is the amount of reward received after performing the action, done is a Boolean value that contains the information whether the environment should be reset, and info is used for debugging. [13]

### 5.2. CartPole-v0

CartPole-v0 is an environment with an un-actuated joint on a cart that moves along a frictionless track. At the start of an episode, the pole is pointing upwards, as seen in Figure 5, and the main goal for the agent is to get the pole to stay upright by either moving to the left or the right. The episode ends when the pole is more than 15 degrees from the vertical axis, or the cart moves more than 2.4 units from the center to either left or right [12]. The CartPole-v0 problem can be considered solved if the agent gets an average score of 195 points over 100 consecutive episodes. The CartPole-v0 problem is possibly one of the easiest reinforcement learning environments that OpenAI has to offer since there are only two possible discrete actions and a state-space that holds information about the position of the cart, velocity of the cart, angular speed of the pole and the angle of the pole. There is also a newer version of the Cartpole-problem called Cartpole-v1, where the problem is considered solved when the agent scores an average of 475 out of 500 points over 100 consecutive episodes. Cartpole-v0 was chosen as the learning environment for this thesis due to its smaller computational encumbrance.

**Figure 5.** Cartpole-v0 starting position and failure position.

### 5.3. TensorFlow

TensorFlow is an open-source deep learning software library released in November 2015 by Google. TensorFlow can be used to model and train artificial neural networks. TensorFlow allows dataflow graph and structure building to show how data moves through a graph. The name TensorFlow can be split into two parts, “tensor” and “flow.” The first part, “tensor” represents the tensors used as input, and the second part, “flow” represents the flowchart of operations performed on the input.

The neural network used in this thesis was created by mainly using the Keras module, which is an easy way to implement neural networks and deep learning and provided by TensorFlow. The following function is used to create a simple neural network:

```
def layers(state_space, action_space, learning_rate):
      model = Sequential()
      model.add(Dense(24,input_dim=state_space,activation
      ="relu"))

      model.add(Dense(24,activation="relu"))
      model.add(Dense(action_space, activation = 'linear'))
      model.compile(loss='mse',optimizer=Adam(learning_rate=
      learning_rate))
      model.summary()
      return model
```

First a sequential model is created, and some dense layers added to the model. A dense layer is the most common deeply connected network layer. The first dense layer is added by using model.add()-method and the specifications of the layer are defined in the Dense()-function. These specifications include the number of neurons, the possible size of the input layer, and the activation function. After defining the layers of the network, the network is configured for training by using the compile-function [15].

The neural network used in this thesis consists of an input layer of four nodes, two hidden layers with 24 artificial neurons each, and an output layer with two output nodes. The neural network takes a state as input; in this case, the inputs are shown in Figure 3, and the state space of the CartPole-v0 problem is represented by four continuous variables, which is why the network has four input nodes. The network has two output nodes since the neural network takes the four states as inputs, processes them, and gives the outputs as action - Q-value pairs. The reasoning behind having two output nodes is that the cart can either move to the left or the right (Action1 and Action2), and the action with the highest Q-value will be executed. Both of the hidden layers use ‘ReLU’ as the activation function. The output layer uses a linear activation function because in the output layer, the node either needs to be activated or not.

### 5.4. Q-learning and Deep Q-learning Programs

The programs for this work were coded in python using algorithms 3 and 4 and can be found at [7]. Q-learning and DQN algorithms are generally straightforward to make, but there are some changes the user can make to make the implementation suited for the environment. The most important changes are bucketing for Q-learning and hyperparameter changes for both algorithms. The state of the cartpole can be determined by four different characteristics: angle of the pole, velocity of the pole velocity of the cart and position of the cart, all of which are continuous values. Handling of continuous values would be too much for Q-learning, instead they need to be bucketed, which means that they are converted into discrete values of chosen size.

```
def get_discrete_state(state):

discretized = list()
for i in range(len(state)):
    scaling=((state[i]+abs(lower_bounds[i]))/(upper_bounds[i]
      -lower_bounds[i]))
     new_obs = int(round((buckets[i] -1) * scaling))
     new_obs = min(buckets[i] - 1, max(0, new_obs))
     discretized.append(new_obs)

return tuple(discretized)
```

The above function is used to figure out what bucket the continuous state belongs in. After the correct bucket has been identified, the bucket is returned and used to determine the correct action by using the following function:

```
def epsilon_action(state_value, epsilon):
    if np.random.random() < epsilon:
         #do exploration = either move left or right
          action = np.random.randint(0, env.action_space.n)
    else:
         #take the best action according to the q table
         action = np.argmax(Q_table[discrete_state])
    return action
```

The action taken depends on the exploration hyperparameter epsilon. If the randomly generated number between 0 and 1 is lower than the epsilon, a random action is taken, and accordingly if it is greater than epsilon, an action is taken according to the optimal policy. The algorithm is run for a set number of times, as presented in algorithm 3 until the terminal state has been reached.

The core idea of Q-learning remains the same in DQN, with the addition of experience replay and a neural network. The initialization of the neural network was presented in section 5.3.

```
def experience_replay(self, iteration):

    mini_batch = random.sample(self.memory, self.batch_size)
    current_state=np.zeros((self.batch_size,self.state_space))
    next_state = np.zeros((self.batch_size, self.state_space))
    action, reward, done = [], [], []

    for i in range(self.batch_size):
        current_state[i] = mini_batch[i][0]
        action.append(mini_batch[i][1])
        reward.append(mini_batch[i][2])
        next_state[i] = mini_batch[i][3]
        done.append(mini_batch[i][4])
    target = self.model.predict(current_state)
    Qvalue_ns = self.model.predict(next_state)

    for i in range(self.batch_size):
        if done[i]:
            target[i][action[i]] = reward[i]
        else:
            target[i][action[i]]=reward[i]+self.discount_factor*(
            np.amax(Qvalue_ns[i]))

    self.model.fit(current_state, target, epochs = 1, verbose = 0,
    batch_size=self.batch_size)

    self.epsilon = self.epsilon_update(iteration)
```

The experience replay-function randomly chooses a number of memories defined by batch size, of previous experiences that are stored in a buffer memory, where the oldest additions to the memory are removed out of the way of new experiences. In the case of the DQN program used in this thesis, the replay function samples 64 random experiences and re-learns them to adjust the weights in the neural network. The experience replay-function is called at every timestep, which means that every time an action is taken, the neural network is trained by 64 randomly chosen past experiences. Batch size is one of the hyperparameters that can be defined by the user; the higher the batch size is, the longer it takes to train the neural network.

### 5.5. Hyperparameter Optimization

Hyperparameters in reinforcement learning are parameters used to control and optimize the learning process. Three critical hyperparameters are used in this thesis: alpha (α), epsilon (ε), and gamma (γ). Alpha is known as the learning rate, and its values range from 0 to 1. A learning rate of 1 would mean that the agent would learn quickly and update the Q-values often, and a value of 0 would therefore mean that it doesn't update the Q-values much, and the learning would be very time-consuming. Alpha is often decayed the further the agent is in the learning progress since, in the beginning, the learning should be fast, so alpha should be closer to one. Still, the learning rate should be low once the learning approaches its end to avoid jumping up and down between the convergence point.

Epsilon is the hyperparameter for exploration; the higher epsilon is, the higher chance that a random action would be taken. Like alpha, epsilon always has a value between 0 and 1 and is decayed. The reason for decaying epsilon is that a lot of exploration is needed to

chart all the possible actions in different states at the start of the learning process so that the optimal actions can be learned. As more state-action pairs get explored, the need for exploration gets lower, and the need for exploitation grows. An epsilon value of 0.1 would mean that there is a 10% chance in a specific state to take a completely random action instead of taking the optimal action.

Gamma is the discount factor that determines the importance of long-term rewards. The discount factor also gets a value between 0 and 1, where one would mean that the agent values future rewards with much greater weight instead of immediate rewards.

There are multiple ways to optimize hyperparameters in RL. The most common ways are grid search, random search, and manual tuning. This work uses manual tuning, which is the simplest method from a programming and implementation perspective. The general value for the hyperparameter is usually known, but fine tuning the parameters to get optimal results is difficult. With manual tuning, the user chooses the hyperparameters by iteratively running the algorithm multiple times and figuring out the optimal hyperparameter values. Table 1 contains the hyperparameters that yielded the best results. The disadvantage of this method is that it requires some time and knowledge to figure out the hyperparameters, especially when working with neural networks, but the advantage is that the user gets to know the effects of hyperparameter changes, which can be helpful in upcoming projects.

Grid search is the simplest and most brute-force way of figuring out optimal hyperparameters. Grid search works by choosing the wanted parameters and setting a range for each parameter, after which the parameters are placed in a grid, and the algorithm is tested with all possible combinations of the hyperparameters. Because grid search goes through all possible combinations of hyperparameters, more parameters equal longer optimization times. In his book, T. Agrawal [16] states that it would take about a week to optimize five parameters for ten algorithms with four possible values for each parameter and a dataset that takes one minute to train an algorithm on one set of hyperparameters.

A better method for training with large datasets and many hyperparameters is called random search. As the name suggests, random search randomly picks hyperparameters from the grid and trains the model. This might not be an optimal method for models that use a low amount of hyperparameters, still, it is a better approach than grid search for models that have too many hyperparameters for grid search to handle in a reasonable time. There is no guarantee that random search will yield optimal hyperparameters, but there are higher chances of finding near-optimal parameters when training models with multiple algorithms and parameters.

**Table 1.** Hyperparameters

| Hyperparameter | DQN | Q-learning |
|---|---|---|
| Learning-rate (α) | 0.001 | 1 |
| $\alpha_{min}$ | 0.001 | 0.001 |
| Exploration rate (ε) | 1 | 1 |
| $\varepsilon_{min}$ | 0.01 | 0.001 |
| Discount factor (γ) | 0.99 | 0.99 |
| Batch size | 64 | N/A |
| Memory size | 2000 | N/A |

There were some differences between the hyperparameters of regular Q-learning and DQL. Exploration rate and discount factor remained the same across the two implementations. The exploration rate was decayed logarithmically using the following formula:

$$\max\left(\varepsilon_{min,}\min\left(1, 1 - log_{10}\left(\frac{iteration+1}{5}\right)\right)\right). \quad (8)$$

The resulting chart of this epsilon decay can be seen in Figure 6, where the epsilon reaches its minimum value after 48 episodes. The decay formula presented in Equation 8 was also used in learning-rate decay for the Q-learning implementation, but not for DQN since it made the rewards converge later and resulted in more instability during the first 100 episodes of training. DQN uses two additional hyperparameters, which are not applicable to regular Q-learning: batch size and memory size. Batch training was presented in more detail in section 4.2.

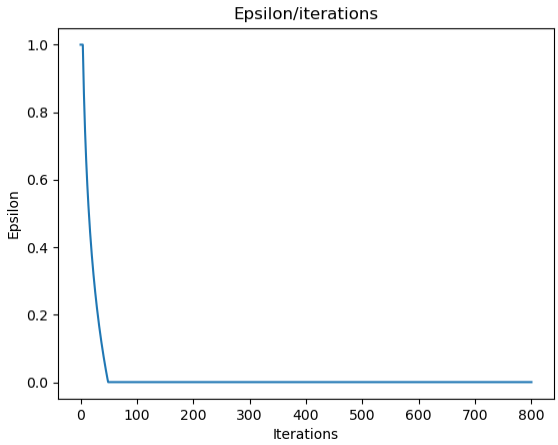


**Figure 6.** Epsilon decay chart using the formula in Eq. 8.

### 5.6. Metrics

As stated in section 1.1, the learning results were evaluated using three key metrics, algorithm convergence, how quickly the environment is solved, and cumulative reward gain. The first two metrics, reward convergence and how quickly the problem is solved, can be seen in Figures 7, 8 and 9. Convergence in Q-learning means that the rewards received from particular states get closer to a certain point and do not fluctuate, in this case the Q-values can be said to be converged to their optimal values since there is no change in the rewards in learning curve after 31 episodes. The rewards converge to a value of 200,

which is the maximum possible reward in CartPole-v0. More specifically, when the threshold is reached where the rewards no longer increase or decrease, the algorithm can be said to converge. The third metric, cumulative reward, is a critical metric where the learning performance can be monitored during the whole learning period. Cumulative reward is calculated by taking the sum of all the rewards received since the learning was started.

### 5.7. Results

The results were obtained with the hyperparameters shown in Table 1 and with the programs found in [7]. The results received did not differ considerably, but the learning process did. The Q-learning program finished 800 iterations of training in 5 seconds, while the DQN program took 1 hour and 25 minutes with a GTX 1060 with 6GB of GDDR5 RAM and an intel i7-7700k @ 4.2Ghz processor. The longer training time can be explained with experience replay since it trains the neural network with saved memories. If the maximum score of 200 is reached, the neural network will be trained 200 times by 64 past experiences, explaining the time-consuming training compared to regular Q-learning.

As shown in Figure 7, the Q-learning agent reached a score of 200 after only 31 episodes and solved it later at 131 episodes. Once exploitation becomes more prevalent than exploration, the agent chooses actions with higher Q-values instead of going for random actions, and the rewards received increase drastically, which can be seen by comparing Figures 6 and 7. The chance to take a random action after 31 episodes is 0.19 according to Eq. 8, and it continues to decrease drastically, since the minimum value of 0.001 will be reached at 48 episodes. The cumulative rewards chart in Figure 8 for Q-learning proves this fact even further; a linear growth in cumulative rewards indicates that the rewards remain steady and do not fluctuate.

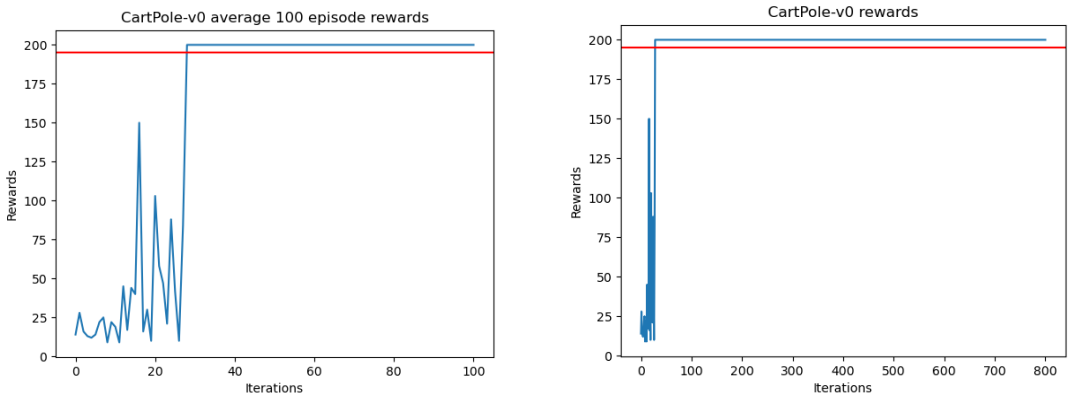


**Figure 7.** Q-learning Cartpole-v0 rewards.

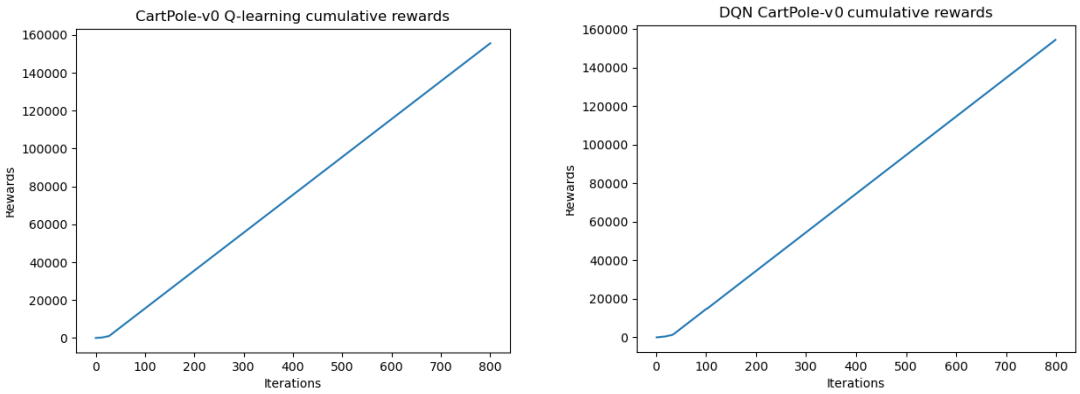


**Figure 8.** Q-learning and DQN Cartpole-v0 cumulative reward in 800 iterations.

These results show that CartPole-v0 is an easy learning environment for regular Q-learning to solve mainly because the Q-table, presented in section 4.1, is small enough to be efficient.

Figure 9 represents the rewards received by the DQN program as a function of iterations. DQN reached the maximum reward of 200 after 31 episodes, and the CartPole-v0 problem was solved in approximately 137 episodes, a bit later than Q-learning since there was a drop in rewards after 100 episodes of training. The neural network training was stopped after the average score of 100 consequent iterations reached 195 because it prevents catastrophic forgetting, which is quite common in DQN. The stopping of the training when reaching a desired score or error is known as early stopping, the error is the difference between the optimal score and the current score [18]. Early stopping works because as soon as the neural network learns the CartPole learning environment, there are no more tasks to be learned, which means further training would turn out to be useless or even harmful. After all, it could lead to catastrophic forgetting [17].

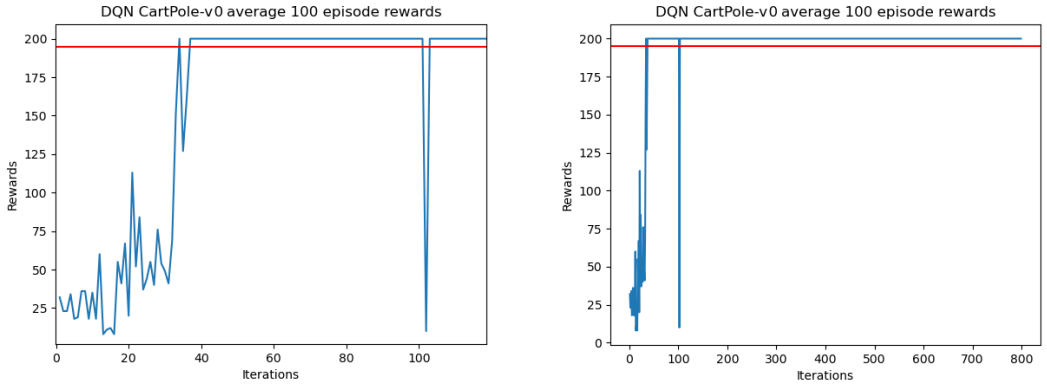


**Figure 9.** DQN CartPole-v0 average rewards over 800 iterations.

It is safe to say both implementations solve the CartPole-v0 environment with ease. However, there is no clear winner since the maximum points were reached in 31 episodes by both agents and solved only six episodes faster by the regular Q-learning. Solving the learning environment six episodes faster does not make Q-learning a better method to solve CartPole-v0 since actions have some randomness involved in the first 50 episodes. In addition, it is not easy to find optimal parameters for DQN by using manual tuning since DQN has more hyperparameters than regular Q-learning and a neural network that needs to be configured.

# 6. DISCUSSION

CartPole turned out to be an easy learning problem since both the DQN and Q-learning implementation solved it effortlessly, which is likely due to the relatively small number of possible states available in CartPole-v0. As mentioned in section 5.1, OpenAi Gym offers much more complex learning environments, which could be a problem for regular Q-learning because of the Q-table not being able to handle a large number of states. DQN, on the other hand, could even be improved further with Double Deep Q-learning (DDQN) and Prioritized Experience Replay (PER), which would render problems such as catastrophic forgetting useless. Early stopping was also a crucial implementation of the DQN program since it would prevent catastrophic forgetting without the need to implement PER or DDQN. The DQN program would learn the environment in under 150 iterations, after which further training could lead to catastrophic forgetting since some of the crucial experiences would be deleted from the memory buffer. DDQN utilizes the same principle as DQN, but instead of just one artificial neural network, DDQN utilizes a target neural network in addition to the active neural network. According to [19], in regular DQN, action selection and action evaluation are performed by the same network model, resulting in "overoptimistic value estimates". DDQN uses the active network to update the weights of the target network by using Polyak averaging: $\theta' = \tau\theta + (1 - \tau)\theta'$, where $\theta'$ is the target network, $\theta$ is the active network, and $\tau$ determines how significant the network weight update is [20]. The update multiplier $\tau$ receives values from 0 to 1, where one means that the active network weights are copied to the target network, which is called a "hard copy," and zero means that no update has been done to the target network. PER is a form of Experience Replay, where the idea is that some experiences are more important than others, meaning that these experiences with higher priority would have a more significant impact on the learning than the lower priority experiences. Further studies and implementations regarding DDQN and PER were left for the future work.

It must be noted that the hyperparameter optimization for the implementations were done manually, meaning that there is no guarantee that the hyperparameters chosen were optimal. In the future, it would be beneficial to compare Q-learning and DQN implementations with more complex environments and use grid search for hyperparameter optimization.

# 7. CONCLUSION

This thesis focused on investigating the difference in learning between regular Q-learning and DQN in the CartPole-v0 learning environment. The first four chapters presented the basic theory of reinforcement learning and the algorithms used, after which the Q-learning and DQN implementations alongside the CartPole-v0 classic control environment were presented.

In general, the model-free Q-learning algorithm performed well, even when compared to the results in [20], from where parts of the DQN-program were adapted. The manual tuning of the hyperparameter turned out to be a lot trickier than initially expected. The tuning took hundreds of iterations and was exceedingly time-consuming while tuning DQN since it introduced two new hyperparameters, batch size and memory size, to the program. In addition to the new hyperparameters, the training lasted for 10 hours before adding early stopping, which reduced the training time to 1 hour and 25 minutes.

As mentioned in the previous chapter, there is no clear winner when comparing the learning results between Q-learning and DQN. There were no differences in results when comparing the key metrics presented in 5.5 and taking randomness in learning into account. However, the Q-learning algorithm was significantly faster at training the agent than DQN. With this being said, Q-learning is better at learning an easy environment since it is more efficient in terms of time. The difference in learning between Q-learning and DQN could be further tested by introducing them to more challenging environments since for easy environments, such as CartPole, the regular Q-learning performs equally well compared to DQN in terms of the metrics used.

# 8. REFERENCES


[1] Russell S. J. & Norvig P. (2010) *Artificial Intelligence: A Modern Approach (3rd Edition),* Harlow, UK: Pearson Education Limited.

[2] Sutton R. S. & Barto A. G. (2018) *Reinforcement Learning: An Introduction (2nd Edition),* Cambridge, USA: MIT Press.

[3] Dong, H., Ding Z., Zhang S. (2020) *Deep Reinforcement Learning: Fundamentals, Research and Applications,* Singapore, Springer Nature Singapore Pte Ltd.

[4] Hambly B., Xu R., Yang H. (2021) *Recent Advances in Reinforcement Learning in Finance,* arXiv: arxiv:2112.04553.

[5] Kamath U., Liu J., Whitaker J. (2019) *Deep Learning for NLP and Speech Recognition,* Switzerland, Springer Nature Switzerland AG.

[6] Chang H. S., Fu M. C., Hu J. & Marcus S. I. *Google DeepMind's AlphaGo, (*Retrieved Nov. 21, 2021) Available: https://www.informs.org/ORMS-Today/Public-Articles/October-Volume-43-Number-5/Google-DeepMind-s-AlphaGo.

[7] Q-learning and Deep Q-learning, (Retrieved Feb 22, 2022) Available: https://github.com/patricksunden/Q-learning-and-Deep-Q-learning.

[8] Serfozo R. (2008) *Basics of applied stochastic processes,* Georgia Institute of Technology, Atlanta, GA, USA, Berlin: Springer.

[9] Ibe O. C. (2013) *Markov Processes for Stochastic Modeling.* University of Massachusetts, Lowell, MA, USA: Elsevier.

[10] Russell J. & Santos E. Jr. (2019) *Explaining Reward Functions in Markov Decision Processes* in: The thirty second international FLAIRS, Lido beach resort, Sarasota, Florida, USA, 19.-22.5.2019, AAAI Publications. S. 56-61. (DOI: 10.1609/aimag.v40i4.5199).

[11] Gurney K. (1997) *An introduction to neural networks,* UCL Press, London.

[12] OpenAI Gym, (Retrieved June 15, 2021) Available: https://gym.openai.com/.

[13] OpenAI Gym Documentation, (Retrieved June 15, 2021) Available: https://gym.openai.com/docs/.

[14] Pip project description, (Retrieved Nov. 30, 2021) Available: https://pypi.org/project/pip/.

[15] TensorFlow Keras documentation, (Retrieved Feb 15, 2022) Available: https://www.tensorflow.org/api_docs/python/tf/keras/.

[16] Agrawal T. (2021) *Hyperparameter Optimization in Machine Learning,* Apress.

[17] Goodfellow I. J., Mirza M., Xiao D., Courville A., Bengio Y. (2013) *An Empirical Investigation of Catastrophic Forgetting in Gradient-Based Neural Networks,* arXiv preprint arXiv:1312.6211.

[18] Cao J., Ma J., Huang D., Yu P. (2022) *Finding the optimal multilayer network structure through reinforcement learning in fault diagnosis,* Measurement (vol 188), DOI: 10.1016/j.measurement.2021.110377.

[19] H. Van Hasselt, A. Guez, and D. Silver. (2016) *Deep reinforcement learning with double q-learning*. In Thirtieth AAAI conference on artificial intelligence, arXiv:1509.06461v3.

[20] S. Kumar, (2020) *Balancing a CartPole System with Reinforcement Learning – A Tutorial* arXiv:2006.04938v2 [cs.RO].

[21] Athaiya A., Sharma S. & Sharma S. (2020) *Activation Functions in Neural Networks*, International Journal of Engineering Applied Sciences and Technology (Vol 4. Issue 12, pp. 310-316).

[22] Jafari R. & Javidi M. M. (2020) *Solving the protein folding problem in hydrophobic-polar model using deep reinforcement learning,* Springer Nature Switzerland AG.