Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Thesis for the Degree of Bachelor in Computer Science 15.0 credits
DVA331

A COMPARISON BETWEEN DEEP Q-LEARNING AND DEEP DETERMINISTIC POLICY GRADIENT FOR AN AUTONOMOUS DRONE IN A SIMULATED ENVIRONMENT

Dennis Tagesson
dtn18001@student.mdh.se

Examiner: Ning Xiong
Mälardalen University, Västerås, Sweden

Supervisor: Shaibal Barua
Mälardalen University, Västerås, Sweden

June 24, 2021

Abstract

This thesis investigates how the performance between Deep Q-Network (DQN) with a continuous and discrete state- and action space, respectively, and Deep Deterministic Policy Gradient (DDPG) with a continuous state- and action space compare when trained in an environment with a continuous state- and action space. The environment was a simulation where the task for the algorithms was to control a drone from the start position to the location of the goal. The purpose of this investigation is to gain insight into how important it is to consider the action space of the environment when choosing a reinforcement learning algorithm. The action space of the environment is discretized for Deep Q-Network by restricting the number of possible actions to six. A simulation experiment was conducted where the algorithms were trained in the environment. The experiments were divided into six tests, where each test had the algorithms trained for 5000, 10000, or 35000 number of steps and with two different goal locations. The experiment was followed by an exploratory analysis of the data that was collected. Four different metrics were used to determine the performance. My analysis showed that DQN needed less experience to learn a successful policy than DDPG. Also, DQN outperformed DDPG in all tests but one. These results show that when choosing a reinforcement learning algorithm for a task, an algorithm with the same type of state- and action space as the environment is not necessarily the most effective one.

Table of Contents

1. Introduction **1**
1.1. Motivation . . . 1
1.2. Problem Formulation . . . 1

2. Background **2**
2.1. Reinforcement learning . . . 2
2.2. Neural Network . . . 2
2.3. Deep Q-Network . . . 3
2.4. Deep Deterministic Policy Gradient . . . 4

3. Related Work **5**

4. Methodology **6**
4.1. Litterature review . . . 6
4.2. Materials . . . 6
4.3. Analysis . . . 6
4.4. Ethical and Societal Considerations . . . 7

5. Simulation Experiment **8**
5.1. Deep Q-Network . . . 9
5.2. Deep Deterministic Policy Gradient . . . 10

6. Exploratory Analysis **12**

7. Results **13**
7.1. 5000 steps . . . 13
7.2. 10000 steps . . . 14
7.3. 35000 steps . . . 14
7.4. Comparison . . . 16

8. Discussion **17**

9. Conclusions **18**

10.Future Work **19**

References **21**

1. Introduction

Reinforcement learning is part of the scientific area of machine learning. It has become an efficient problem solver for a wide selection of problems, one of such is autonomous navigation in a simulated environment. It can be problematic to train reinforcement learning in a live setting since it is essentially a trial by error, however, it is suitable in a simulated environment, where an error simply means a do-over. Unlike other methods in machine learning, like supervised learning, reinforcement learning does not require a labeled data set to learn from, which can be advantageous if such data sets are scarce. There exists a wide variety of reinforcement learning algorithms, each with its advantages and disadvantages. One important aspect to consider when choosing an algorithm is the state- and action space of the environment. The state- and action space can be either discrete or continuous. For example, if the action space is discrete then the number of actions is countable while if it is continuous the number of actions is, in theory, infinite. An algorithm with a discrete action space can only be applied to a continuous environment if the action space of the environment is discretized. This thesis investigates how the performance of one such algorithm differs from an algorithm with continuous action space that can be applied to the environment out of the box. This knowledge can then be used for future applications when deciding upon a reinforcement learning algorithm. In this thesis an algorithm with a discrete action space, Deep Q-Network (DQN) [1], is compared with an algorithm with a continuous action space, Deep Deterministic Policy Gradient (DDPG) [2]. They will be applied to a drone in a simulated environment. The goal of the algorithms is to learn this drone to go from point A to point B in as short a time as possible. The simulation features a continuous action- and state space. DQN and DDPG are both designed for a continuous state space so the big difference between them is the action space. Because of the action space, DQN cannot be applied to the environment out-of-the-box, the action space has to first be discretized. The purpose of this thesis is to examine how the performance of DQN and DDPG differs in a continuous environment and to test how the action space can be discretized for DQN to potentially achieve greater performance than DDPG.

1.1. Motivation

The aim of the thesis is to give insight into how important the choice of algorithm based on the action space is, which can be of aid to future applications where the choice of algorithm is unclear. An environment with a continuous state space has, in theory, infinite states. The state space could be discretized, similar to the action space, but not without removing possible states for the agent. This could have negative consequences since the agent can not interact with the majority of the environment anymore. The same issue does not present itself when the action space is discretized. Since only the movement of the agent gets limited no information from the environment is hidden behind the discretization. DQN and DDPG are both well-known and widely used in the field of reinforcement learning. The main reason these algorithms were chosen for this thesis is their similarities coupled with the difference in action space. The development of DDPG was motivated by the success of DQN and its lack of continuous control [2], their purpose was to create an algorithm, with insight from DQN, to handle continuous state spaces. This makes DQN and DDPG suitable candidates for investigating my research questions.

1.2. Problem Formulation

The choice of a reinforcement learning algorithm for a specific application is not a simple task. There are many different applications of reinforcement learning and the state-action space can look different in everyone. Intuitively a continuous action space algorithm would outperform a discrete one in a continuous environment, but this is not necessarily the case, which can be seen for discrete action spaces in [3].

The thesis will investigate the following research questions:

- How does the performance of an algorithm with discrete action space compare to one with continuous action space?
- How can the continuous action space be discretized for Deep Q-network to achieve better performance?

2. Background

2.1. Reinforcement learning

A standard reinforcement learning model [4] is based on two things, an agent and an environment. The environment has a set of states, its state space, and the agent has a set of possible actions called the action space. The interaction between the agent and environment works by having the agent take actions in the environment and based on those actions reward the agent. The reward is based on predetermined factors. The reward does not have to be positive. A negative reward can be used to punish the agent if an action resulted in an especially bad state. In this thesis, both positive and negative reinforcement will be used to incline the agent towards effective actions. A value function is used to give a value to actions in a specific state. The function calculates the expected reward for any action in a given state. The ultimate goal of the agent is to learn an optimal policy.

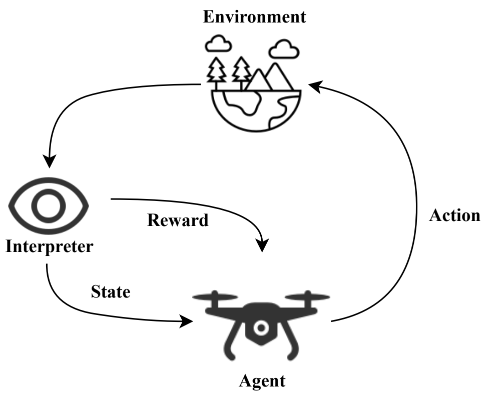


Figure 1: An illustration of the relationship between the agent and the environment in reinforcement learning.

A state- and action space can be either discrete or continuous. For a basic environment, for example, the game of Connect Four, a discrete state space is suitable because of its simplicity. The state can easily be represented by an array-like data structure. The same goes for the action space. Q-learning [5] is a reinforcement learning algorithm that can be applied to an agent with a discrete action- and state space. Q-learning uses a table to store the values of all actions for all states. Each iteration the agent lookup the highest-valued action for a given state in the table. A real example of continuous action is a gas pedal in a vehicle. The acceleration produced by pressing down the pedal can not be divided into “sections” where one section corresponds to an acceleration, which would be the case for discrete action. Similarly, a continuous state space is a set of states that can not be easily defined to a finite number of states.

2.2. Neural Network

For a more complicated environment, like the simulation in this thesis, a continuous state space is better suitable to give the agent a representation of the state. This is accomplished by incorporating neural networks into reinforcement learning. This is called deep reinforcement learning and it makes it possible to have a state that represents sensory data from the environment. Neural networks [6] is a structure built of multiple layers, each with a set of neurons connected to other neurons in bordering layers. There is no restriction on the number of layers in a neural network, however, there are some restrictions on the number of neurons in different categories of layers. The layers are divided into three categories, input layer, hidden layer, and output layer. The input layer receives data from external sources, and thus the size, i.e. number of neurons and shape must match the input data. For example, if an image is to be processed by a neural network the size of the input layer has to match the number of pixels in the image. The next set of layers are the hidden layers. Contrary to an input layer, the size of a hidden layer is not restricted by the data, instead, the size can be determined by the programmer. Finally, the output layer gives the result of the neural network. The size of the output layer is decided by the number of classes in the model, for example, in reinforcement learning, all actions an agent can perform. The output represents a prediction of what the best action is given a specific input. The neurons in neighboring layers are connected. Each neuron in a layer has a synapse to every neuron in the next layer. Each connection is given a weight that represents the relative importance of the connection. The weights are normally initialized to some arbitrary value and then updated during the training of the network. They are updated according to some loss function that computes if some prediction was good or bad. For an example of a simple neural network with one input layer, four hidden layers, and an output layer with a single output, see figure 2.

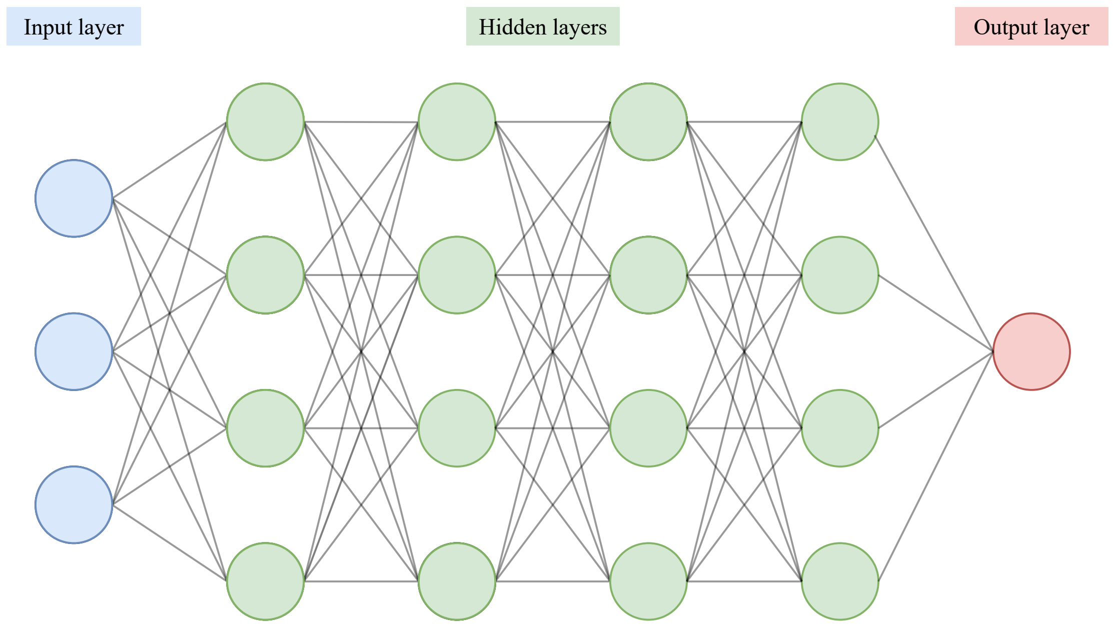


Figure 2: Visual representation of a neural network with an input layer with 3 neurons, 4 hidden layers with 4 neurons each, and finally an output layer with a single output neuron. The lines between neurons represent the weights.

2.3. Deep Q-Network

Following some research progress on sensory integration in deep learning [7], Q-learning and convolutional neural networks were combined to create Deep Q-Network (DQN) [1]. Q-learning uses value iteration to estimate the action value function, also called Q-function. It iterable updates the Q-value for every state-action pair by using the Bellman optimality equation [5],

$$q_*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} q_*(s',a')\right] \tag{1}$$

Where (s, a) is the state-action pair, R is the expected reward from taking action a in state s, t is the time-step of the current episode and γ is a variable that discounts future rewards. The purpose of γ is to give greater importance to immediate rewards relative to future rewards. These Q-values are stored in a table with the respective state-action pair. This table holds a Q-value for every possible action in each state. This is not viable for large state spaces. Deep Q-network solves this by, instead of using value iteration, training a function approximator, in this case, a neural network, to estimate the optimal Q-function.

The neural network is trained by minimizing the loss given by the following loss function, as defined in [1]:

$$L_i(\theta_i) = E_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a,\theta_i)\right)^2\right] \tag{2}$$

Where (s, a) is the state-action pair, θ is the weights of the network at iteration i, θ^- is the weights of the target network, r is the reward, γ is a variable that discounts future rewards and $E_{(s,a,r,s')\sim U(D)}$ is a set of random previous experiences from the replay buffer.

Before DQN, the consensus was that using a large non-linear function approximator, for example, a neural network, to estimate value functions was difficult and unstable [2]. To combat this two new features were introduced, a replay buffer and a target network [1]. The role of the replay buffer is to save experiences in the (s, a, r, s') format. That is when given a state s, what action a was predicted and what reward r and new state s' did this action result in. Random samples from the buffer are then a part of calculating the loss. It can be seen in the loss function presented above that two independent Q-values are calculated, one from the learning network and one from the target network. The target network lags behind the learning network and the weights of the target network are only updated every some fixed number of steps. The term $r + \gamma \max_{a'} Q(s', a'; \theta_i^-)$ from the loss function, equation (2), is called the target. The loss is minimized by making the Q-function closer to this target. If there was no target network with different parameters, the target, and the Q-function would be based on the same parameters which would make the learning unstable[1].

2.4. Deep Deterministic Policy Gradient

The development of deep deterministic policy gradient (DDPG) [2] was inspired by the success of DQN and is aimed to improve performance for tasks that requires a continuous action space. DDPG is based on the deterministic policy gradient (DPG) algorithm proposed in 2014 [8]. DPG is an algorithm that works with a discrete state space and a continuous action space. By combining DPG and neural networks, DDPG was designed for continuous state- and action spaces. DDPG incorporates an actor-critic approach based on DPG. This means the algorithm uses two neural networks, one for the actor and one for the critic. The critic is essentially a Q-function, it can learn the same way as the neural network in DQN, by minimizing the loss. The results from the critic are used in the actor policy, the gradients of the Q-value are computed with respect to the action input, which is returned by the critic. The actor-critic [8] approach is based on the policy gradient theorem. Thus, the actor policy is updated according to the sampled policy gradient, as was shown in [9]:

$$\nabla_{\theta^\mu} J = E_{s_t \sim \rho^\beta} \left[\nabla_\theta^\mu Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|s = s_t \right] \tag{3}$$

Where $\mu(s|\theta^\mu)$ is the actor function that maps states to some action, $Q(s, a|\theta^Q)$ is the critic, s is the state, a is the action and θ is the parameters of the network.

DDPG adopted the usage of target networks from DQN to improve stability [2]. However, the update of the target networks is modified. Instead of directly copying the parameters from the learning network at every fixed number of steps the target updates are “soft”. Initially, the actor and critic networks are copied to create the target networks. The target networks are then constrained to update the weights by slowly tracking the learning network. The weights of the target network are defined by $\theta' = \tau\theta + (1 - \tau)\theta'$, in which θ is the parameters of the learning network, θ' is the parameters of the target and τ is the share of weights that are included from the learning network. A replay buffer was also included, whose purpose and function are the same as in DQN.

For more details on DQN and DDPG see the paper [1] and [2] respectively.

3. Related Work

Many physical control tasks, like the movement of a body part or the motion of a vehicle, are continuous by nature, which can be troublesome for algorithms that operate on a discrete action space. Despite this, it was shown in [10] that Deep Q-Network has the potential to efficiently solve a problem with a continuous state space. They used DQN to learn multiple agents, i.e. cars, in a simulated environment to navigate occluded intersections. To do so they discretized the action space in three different ways. The simplest discretization was called Time-to-go, which only gave the agent two possible actions, wait or go. A more free action presentation was sequential actions, where the agent was given the ability to accelerate, decelerate or maintain current velocity. Lastly, creep-and-go was a combination of the previous two, the agent could wait, move forward and go. In all three of these, the path that the agent will take is predetermined. My initial discretization of the action space for DQN is similar to the sequential actions of [10], but with the difference that the path is not predetermined. Additionally, a drone is more free in its movements compared to a car, a drone can change direction both vertically and horizontally while the steering of a car only the direction horizontally. So in addition to discretization of the velocity at which the drone will move I will also discretize the direction of the movement.

Discrete versus continuous action spaces are widely discussed in the field of reinforcement learning. [11] showed that an algorithm designed for a continuous action space can outperform discrete algorithms on a game with a discrete action space. Their paper compared the Continuous Actor-Critic Learning Automaton (Cacla) with algorithms using a discrete action space, one of which was Q-learning, on two classical reinforcement learning problems, the mountain car, and the cart pole. The continuous action space was discretized by rounding the action value outputted by Cacla's actor to the nearest legal action in the action space. This is not directly applicable to my problem since I will discretize a continuous action space for a discrete algorithm. However, [11] together with [10] clearly shows that an algorithm can be applied to a different action space than what it was designed to and still show good performance, in this case even exceeding the performance of algorithms that was designed for this kind of spaces. The opposite, and more similar to my approach, discretizing a continuous action space, was done by Y. Tang and S. Agrawal in [12]. They demonstrated a significant performance increase for the on-policy algorithms Proximal Policy Optimization (PPO) and Trust Region Policy Optimization (TRPO). PPO and TRPO with a discrete policy were compared to multiple off-policy algorithms considered to be state-of-the-art, among these was DDPG. The algorithms were tested on games with humanoid tasks, like humanoid-v1 from OpenAI gym. Both PPO and TRPO demonstrated a greater average reward, after training for a number of steps than DDPG.

Similar results can be seen for off-policy algorithms in [3], by J. Pazis and M. G. Lagoudakis. They proposed a new method called Binary Action Search allows a reinforcement learning algorithm with a discrete action space to learn a continuous action space policy. They applied their method to two reinforcement learning algorithms, Least-Squares Policy Iteration and Fitted Q-Iteration which showed promising results in three different environments with a continuous action space. They state that the method can be applied to any reinforcement learning algorithm supporting discrete actions and continuous states. Instead of having a policy that predicts an action when given a state, the Binary Action Search also considers the current action and will decide how to modify it.

In the original paper of DDPG [2] their experiments showed that their algorithm needed less experience before reaching a solution than DQN. My research complements theirs by doing a more in-depth comparison between DDPG and DQN where specific improvements from the DDPG research were incorporated in the DQN implementation. For example, having DQN use "soft" target updates as described in [2] and using the same hyperparameters and the same basic structure of the neural networks.

4. Methodology

The data necessary to investigate the research questions will be gathered by doing a simulation experiment. According to [13] experiment can be used to examine how a system behaves under certain conditions, where several parameters can affect the result. Experiment is a useful method for this thesis since the first research question I aim to answer is about comparing the two deep reinforcement learning algorithms, Deep Q-Network, and Deep deterministic policy gradient. [13] also states that experiments can be used to determine which variables and parameters have the greatest influence on the result. This makes it a suitable research method for the second research question as well, where I examined how the action space can be discretized to improve performance for DQN.

4.1. Litterature review

It is important to familiarize myself with the current state of knowledge in the field [13]. This was done by performing a literature review before the experiment. The purpose of this is to get an understanding of what is already published by scientists in the field of deep reinforcement learning and more specifically DQN and DDPG. To find relevant and useful papers I used a method called snowballing [14]. It is done in three steps, first I find an initial set of papers, then the reference list of those papers is used to find additional papers, and so on until no relevant papers can longer be found. The third step is to use a database to search for papers that cite one of my chosen papers. By doing that, relevant papers published after the initial set can also be found. I used the search engines Google Scholar and Primo to find the initial sets of papers. Some of the keywords that are used in the search are reinforcement learning, deep reinforcement learning, q-learning, deep q-network, deterministic policy gradient, and deep deterministic policy gradient.

4.2. Materials

The next step, after the literature review, is the implementation of the deep reinforcement learning algorithms, DQN and DDPG. The agent in the simulation is a drone and since a drone's movement is not constricted at all its action space is continuous. DQN operates on a discrete action space so the algorithm's action space has to be discretized. Instead of allowing the drone to move in any direction it will be restricted to move in only 6 directions, up, down, left, right, forwards, and backward. DDPG works with a continuous action space so this restriction only applies to DQN. I used two libraries for the implementation, keras-rl[15] to implement the algorithms, DQN and DDPG, and the reinforcement learning environment will be created with OpenAI Gym [16]. When the implementation is done, the reinforcement learning of the autonomous drone can begin. The simulated environment that was used is AirSim [17]. DQN and DDPG were able to control and receive information about the drone from the AirSim API. The API allows the algorithms to programmatically control a drone as well as receive the current state of the vehicle. To apply the two reinforcement learning algorithms on the agent and get the state, the API was used to extract information such as current speed, current height, images from cameras, and collision detection.

Since reinforcement learning is all about learning from past experiences, data were collected during each episode to build a knowledge base that the agent will use for its decision-making in the next episode.

4.3. Analysis

I conducted an exploratory analysis of the data collected during the simulation experiments to answer the research question. The data relevant to the comparison of DQN and DDPG are the loss, average Q-value, the episode reward, number of steps, and duration of the episode. In some cases, the computation of the loss and Q-value would result in an overflow or division by 0, which did not return a value. So before the data is used all the "not a number" values were changed to 0. Then the data was used to create graphs to visualize the performance of DQN and DDPG. The graphs were the basis for the exploratory analysis.

4.4. Ethical and Societal Considerations

This thesis does not concern any real or confidential data. The simulation used in this thesis, AirSim[18], and the python libraries keras-rl[15] and Open AI gym [19] are all open-source software and released under the MIT License which gives no restriction to usage.

5. Simulation Experiment

The experiment is conducted in an environment built in Unreal engine provided by AirSim [17]. The environment is an urban neighborhood where the entities are mostly static, i.e. there are no moving cars or other large objects that can collide with the drone, except for some rarely seen small animals running around the environment. The environment is built to scale, meaning one distance unit is considered to be one meter. The algorithms are trained one at a time in this environment. The algorithms are trained for a fixed number of steps and then they will be evaluated based on the logs kept during the training. Data is recorded each step and then logged at the end of an episode. To prevent that the training got halted because the drone got stuck in a loop the episode is terminated and reset after 200 steps, regardless of the current state of the agent.

The reward given to the agent is based on multiple different factors. There is one reward associated with a non-terminal action, the current distance to the goal. The distance d is the Euclidean distance, which, in three dimensions, are calculated by

$$d(p1, p2) = \sqrt{(p1_x - p2_x)^2 + (p1_y - p2_y)^2 + (p1_z - p2_z)^2} \tag{4}$$

Where $p1$ and $p2$ are three-dimensional points in the environment. The distance between the agent's current position and the goal is calculated every step that is not a terminal state. The reward is the previous distance to the goal subtracted by the current distance to the goal multiplied by 5. It is multiplied by 5 to give it greater importance relative to the other reward factors. Additionally, there are four different terminal conditions that affect the reward:

- Collision
- Agent is stuck
- Distance to goal is too great
- The goal is reached

If one of these becomes true then the episode is terminated and the agent is reset. The AirSim API provides collision detection, so whenever the agent collides with an object the reward is adjusted accordingly. The agent receives a -10 reward after a collision. If the agent is stuck it also gets a -10 reward. The agent is considered stuck if it stays in place for more than five steps. The final negative terminal state is if the distance to the goal is greater than 20 meters. If this occurs the agent receives -10 reward that step. There is only one positive terminal state, that is if the agent reaches the goal. Whenever that happens it receives 100 reward. The environment considers the agent to have reached the goal if it is closer than 3 meters of it.

The state representation that is given to the agent are a one dimensional array with seven elements, $[v_x, v_y, v_z, p_x, p_y, p_z, dist\_goal]$ where v is the current velocity of the drone, p is the current position and $dist\_goal$ is the distance to the goal. This input data is the input to the neural networks of DQN and DDPG. Both implementations of DQN and DDPG used the stochastic gradient descent method Adam [20] for updating their weights. The same learning rate was used for the algorithms, 0.001. The weights for the respective network were initialized the same way. The final layer was initialized to a random uniform value between -0.003 and 0.003 and the hidden layers used Glorot uniform initialization [21]. The range, $[-range, range]$, of which the sampled values are distributed from are given by

$$range = \sqrt{\frac{6}{(fan\_in + fan\_out)}} \tag{5}$$

Where fan_in is the number of input weights and fan_out is the number of outputs weights in a layer. All hidden layers for the neural network in DQN, the actor network, and the critic in DDPG, uses Rectified Linear Unit (ReLU) [22] activation. Unlike the original paper on DQN [1], my implementation of DQN uses "soft" target updates to make it similar to DDPG. Both DQN and DDPG sets the parameter τ to 0.001. So the weights θ' of the target networks are defined by $\theta' = 0.001 \cdot \theta + 0.999 \cdot \theta'$.

5.1. Deep Q-Network

The neural network for DQN is built with four layers, the input layer, two hidden layers, and one output layer. First is the input layer with seven input neurons, one for each element in the array with data of the velocity, position, and distance to the goal of the drone. The first hidden layer has 300 neurons and the second has 400 neurons, which is the same they used in the original paper for DDPG [2] for low-dimensional inputs. The structure of the neural network of DQN can be seen in figure 3. A linear activation function is used in the output layer since the output layer predicts a Q-value for each action. The Q-value for every neuron in the output layer, i.e. each possible action, is calculated by the Bellman optimality equation, which can be seen in equation (1). The Q-values returned by the network are predictions on how good the different actions are in a specific state. Each output neuron corresponds to an action in the range $[0, 5]$, and the action with the highest valued Q-value is chosen for the agent. The neurons $d_1, d_2, ..., d_6$, in figure 3, corresponds to the actions $0, 1, ..., 5$ respectively. This action determines in what direction the drone will move. The movement is with a velocity of $5m/s$ and the duration of it is 0.5 seconds. So after $0.5s$ a new action can be given to the agent. The velocity vector that each action corresponds to can be seen in table 1 The drone executes every command for 0.5 seconds.

Action	0	1	2	3	4	5
Velocity (v_x, v_y, v_z)	(5,0,0)	(-5,0,0)	(0,5,0)	(0,-5,0)	(0,0,5)	(0,0,-5)

Table 1: Table with the corresponding velocity associated with a given action. The velocities are represented by a three dimensional vector.

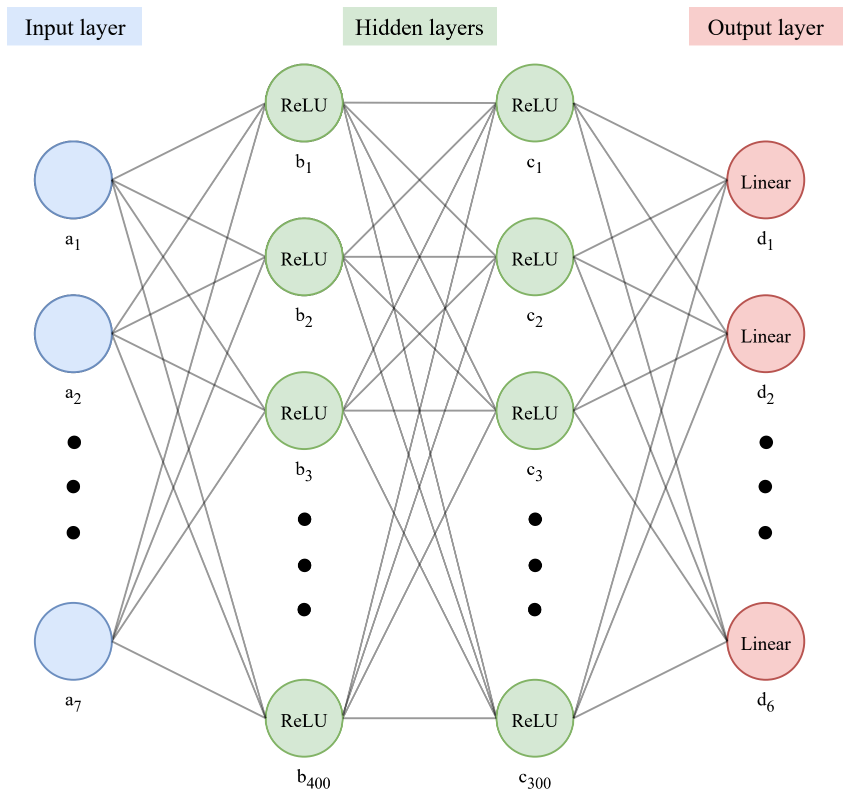


Figure 3: Visual representation of the neural network used for DQN. The are seven neurons in the input layer, 400 in the first hidden layer, 300 in the second hidden layer and six neurons in the output layer. All hidden layers use the Rectified Linear Unit (ReLU) activation function and the output used no activation function, i.e. linear activation.

5.2. Deep Deterministic Policy Gradient

The neural networks used in DDPG were structured as similar as possible to DQN but since DDPG is an actor-critic-based algorithm, some layers and activation functions differ. Unlike DQN, the action given by the actor network in the DDPG implementation is continuous and is in the form of a three-dimensional vector, with each value representing a velocity in some axis. Since the hyperbolic tangent (TanH) function is used as the activation function in my final layer of the actor, it will predict a value between -1 and 1. The noise is then added to the predicted value, which is the resulting action given to my reinforcement environment. Since this value is small it would not give the drone a significant velocity, so the action is multiplied by 5 and clipped to -5 and 5. So if the multiplication would go below -5 or over 5 it is simply set to -5 or 5, respectively. The action is multiplied by 5 and clipped to make it comparable to the velocity of the DQN counterpart, which always moves the drone with a velocity of 5 m/s.

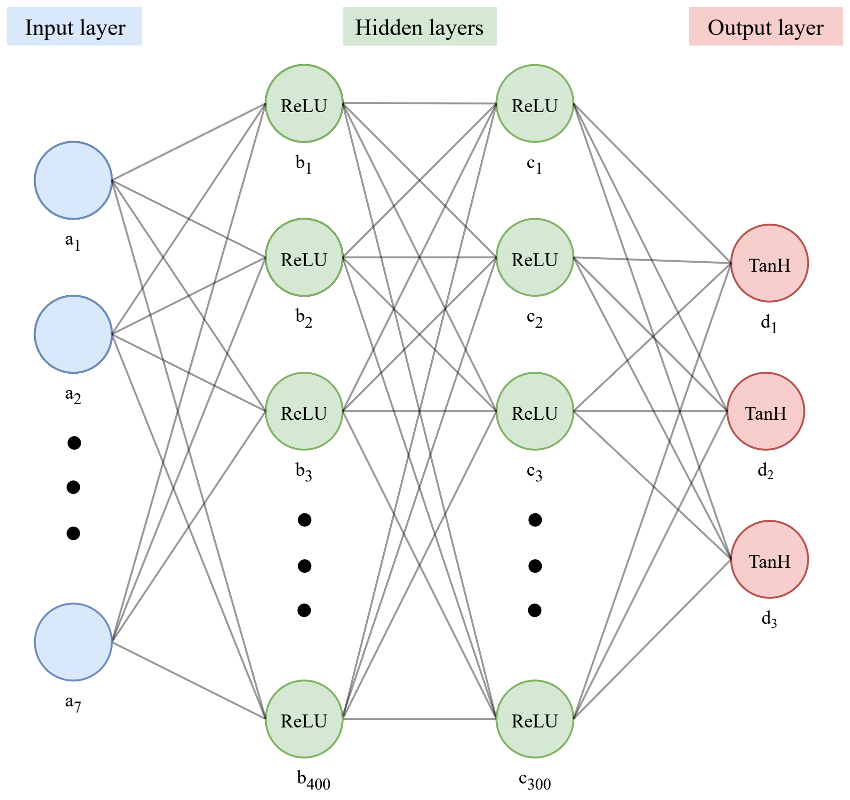


Figure 4: The actor neural network used in the DDPG implementation. The are seven neurons in the input layer, 400 in the first hidden layer, 300 in the second hidden layer and 3 neurons in the output layer. All hidden layers use the Rectified Linear Unit (ReLU) activation function and the output uses the hyperbolic tangent (TanH) activation function.

There are some significant differences between the actor and the critic neural network, both use 400 and 300 for each hidden layer respectively, but the differences lie in the input and output. Both networks have the array with the state representation as input in the input layer, however, the critic network also has the action as an input. The actor network, see figure 4, is very much similar to the network of DQN, seen in figure 3, except for the output layer. While the critic network is more distinct, see figure 5. Initially, the critic network is structured the same as the actor, with the state representation as input followed by a hidden layer with 400 neurons. After the hidden layer, the action predicted by the actor network is given to the critic as input. The input consists of a vector (v_1, v_2, v_3) which corresponds to the velocity of the agent in the x-,y- and z-axis respectively. The action input is concatenated with the result from the first hidden layer and connects with the second hidden layer. The output layer of the critic network only consists of one neuron. This neuron holds the Q-value for the state-action pair given as input to the critic network. The state is the state representation of the environment and the action being the output of the actor network. Every step during the learning the agent interprets the current state of the environment and gives the state to the actor network, which then predicts an action. This action is then given to the critic network together with the state to predict the Q-value.

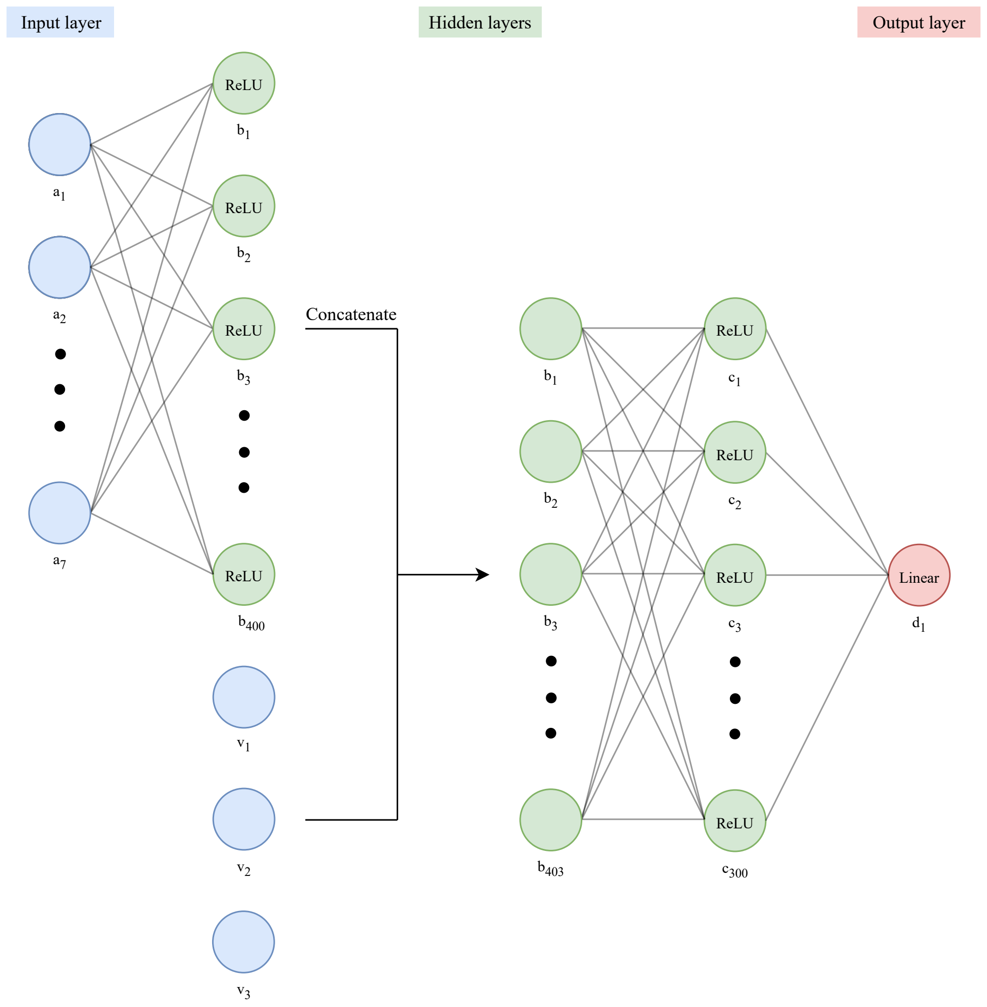


Figure 5: The critic network used in the DDPG implementation. The are seven neurons in the input layer, 400 in the first hidden layer, 300 in the second hidden layer and one neuron in the output layer. All hidden layers use the Rectified Linear Unit (ReLU) activation function and the output uses linear activation.

Since the policy of DDPG is deterministic it is more prone to exploitation rather than exploration. So to make the agent explore more noise is added to the action predicted by the actor. The noise used in my experiment is the Ornstein-Uhlenbeck process [23], which is the method recommended in the original paper of DDPG [2]. I used it with parameters $\theta = 0.15$ and $\sigma = 0.3$.

6. Exploratory Analysis

The data gathered from the simulation experiments are used to compare the performance of DQN and DDPG. During the simulations, logs are kept for every episode. Each episode has several categories with data associated with it. The following data are used for evaluating the performance of the algorithms:

- Average Q-value
- Episode reward
- Duration

The average Q-value is simply the average of all Q-values computed in that episode. Episode reward is the reward given to the agent some episode, number of steps is the total number of steps until it reached a terminal state and the duration is simply the duration of the episode in seconds.

The mean Q-value indicates how good the actions in a certain episode were. The evolution of the Q-value can be used to get an overview of how well the network was trained. If the Q-values are increasing it is an indication that the predictions are improving. The scale of the returning Q-values are different for DQN and DDPG, so to be able to give a compare them, the Q-values are normalized to $[-10, 10]$. The reward per episode is an obvious metric to use when comparing reinforcement learning algorithms since it shows how fast the respective policy adapt to maximize the reward. However, the episode reward does not indicate how fast the agent reached the goal. In theory, the agent could move up and down in the same spot in quite some time and then go to the goal and get the same amount of reward as if it went straight for the goal. The metric duration is used to determine how efficiently an agent reached the goal. I will refer to the speed of completion for an episode as the "time to goal". The best time to goal for a test is the duration of the episode with the least time spent before successfully completing it. An episode is considered successful if the episode reward is greater than 70. Additionally, the average duration of all successful episodes in training was used to get an overview of how efficiently the agent complete the objective. Simply using the duration to compare the best time to goal of the respective algorithm is not enough to get an indication of which algorithm finds the most efficient policy, since one could have a single efficient episode with the rest of them being slow. Also, the number of successful episodes is likely to be different between the results of DQN and DDPG. So to get a metric that also takes into account the number of successful episodes the number of successful episodes is divided by the average duration of those episodes. This will prevent that one algorithm gets a very good score when it has very few successful episodes with a low time to goal. This metric will be referred to as "time to goal score".

In summary, the metrics that were used to compare the algorithms were:

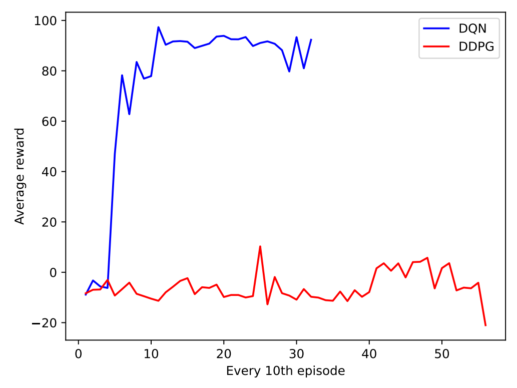
- Average Q-value
- Episode reward
- Best time to goal
- Time to goal score

7. Results

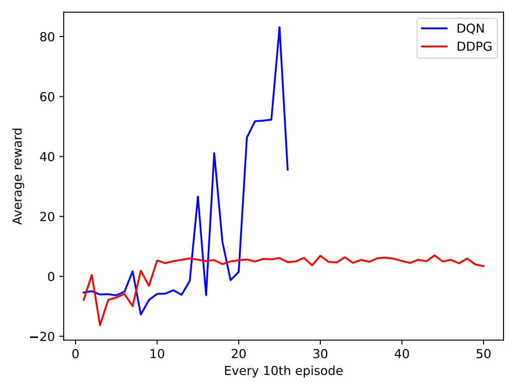
Two different locations were used for the goal in the environment and the agents were tested on those two goals for three different amounts of steps, 5000, 10000, and 35000. The easiest goal location was located 15 meters in front of the start position of the drone and there were no obstacles in the way. This goal is referred to as “goal 1”. So all the agent had to learn was to go forward until it reached the goal. The other goal location was positioned closer to the start but had some obstacles in the way. It was 12 meters diagonally behind the start with a small fence and street signs in the way of a direct line to the goal. This goal is referred to as “goal 2”.

7.1. 5000 steps

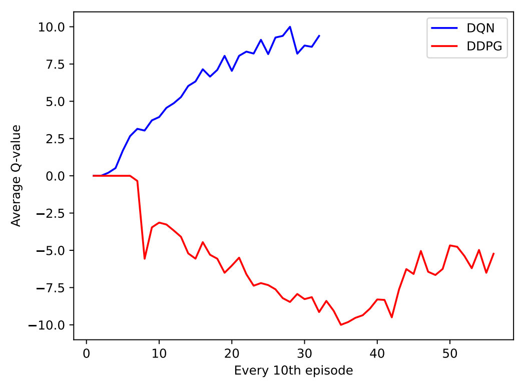
The average episodic reward for the shortest training phase with 5000 steps can be seen in figure 6. Figure 6a depicts the result of the training for DQN and DDPG with the goal 1 and 6b shows the result for goal 2.



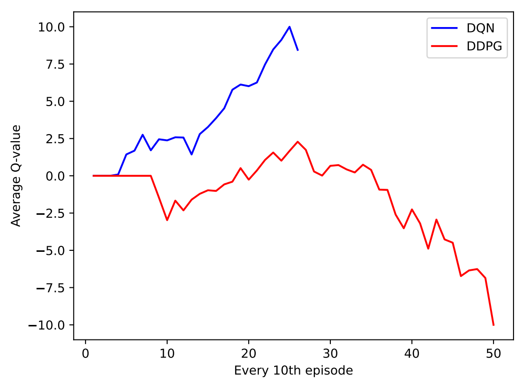
(a)



(b)



(c)



(d)

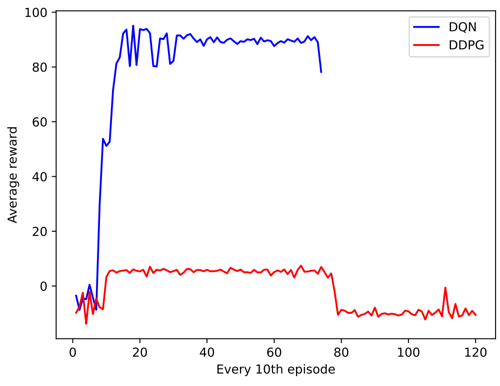
Figure 6: Graphs showing the average episodic reward and the average Q-values for DQN and DDPG over 5000 steps. (a) and (c) is the average reward and Q-values, respectively, for the goal located directly in front of the start position. (b) and (d) is the average reward and Q-values, respectively, for the goal that is diagonally behind.

The results of the reward show that DQN learns a good policy after 10 episodes for goal 1 and remains quite stable from then on. DDPG does not seem to be able to improve at all in these 5000 steps. For the test with goal 2, DQN did not converge and remain stable. The average reward improved but not with the same stability shown for goal 1 in figure 6a. Unlike DQN, DDPG converges after about 10 episodes however, it does so prematurely. An optimal policy would achieve above 100 reward per episode, which DDPG does not come close to.

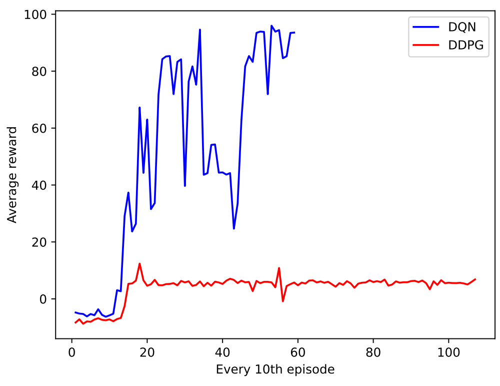
The average Q-value for the 5000 step training can be seen in figure 6c and 6d. It shows similar results as the average episodic reward for 5000 steps, DQN is continuously improving over the course of the training for both goal locations. Whereas the Q-values for DDPG are not improving.

7.2. 10000 steps

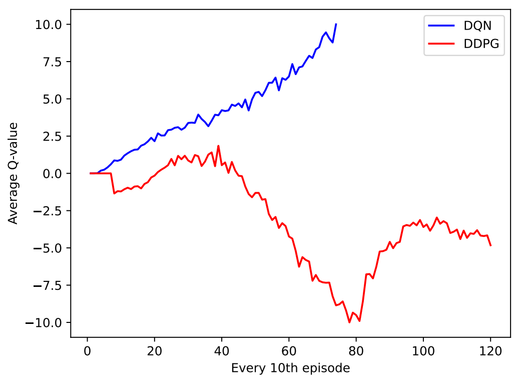
The graphs with the average reward and Q-values can be seen figure 7, 7a and 7b shows the average reward for goal 1 and 2 respectively. 7c and 7d shows the average Q-values for both goals. The results for training DQN and DDPG for 10000 steps are similar to that of the results for 5000 steps. The same patterns can be seen in the curves below as the curves in 6, for goal 1 DQN quickly converges to an close-to-optimal policy while DDPG does not improve significantly in these 10000 steps. Also the curves for the Q-values show the same pattern, where the Q-values for DQN are continuously improving while Q-values for DDPG worsens for the most part.



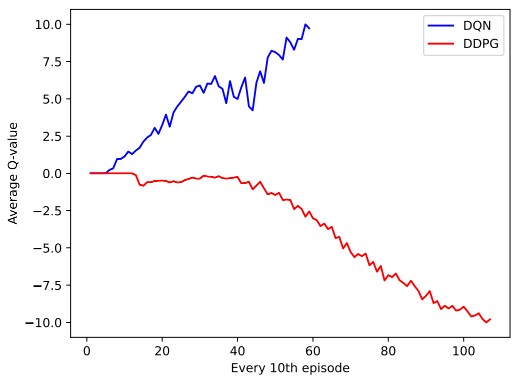
(a)



(b)



(c)



(d)

Figure 7: Graphs showing the average episodic reward and the average Q-values for DQN and DDPG over 10000 steps. (a) and (c) is the average reward and Q-values, respectively, for goal 1 (b) and (d) is the average reward and Q-values, respectively, for goal 2.

7.3. 35000 steps

DDPG starts to show some more competitive results when trained for 35000 steps, as seen in the graphs in figure 8. Its performance is still inferior to that of DQN for goal 1 but it does show some more competitive results for goal 2. For goal 1 DQN quickly converges to a close-to-optimal policy after only 10 episodes, as seen in figure 8a, while DDPG takes more than 200 episodes to reach a similar average reward as DQN.

For goal 2, DDPG did show performance similar to that of DQN, although at a slower rate. Initially, DDPG showed a greater average episodic reward than DQN but after 50 episodes the performance of exceeded DDPG. It took almost 100 episodes before DDPG started to show significant improvement, while DQN started to show a considerable increase in the average reward after only 30 episodes.

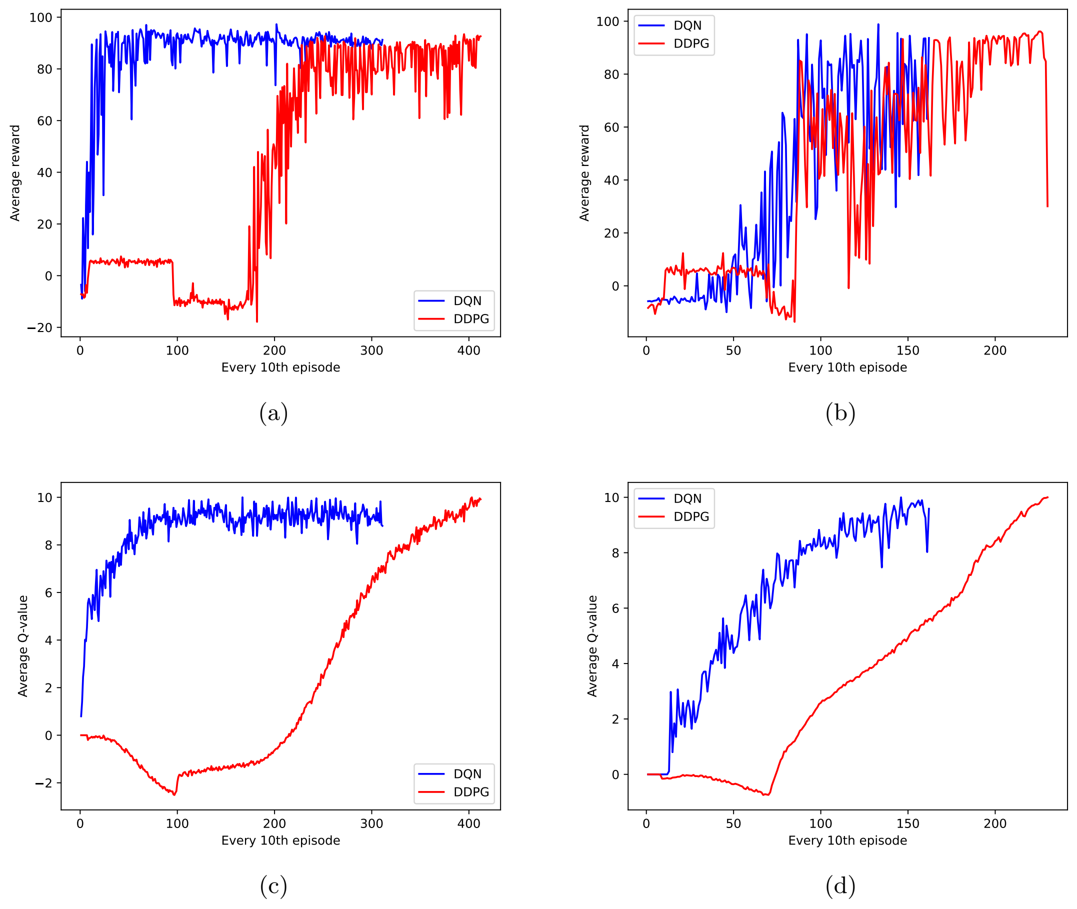


Figure 8: Graphs showing the average episodic reward and the average Q-values for DQN and DDPG over 35000 steps. (a) and (c) is the average reward and Q-values, respectively, for goal 1. (b) and (d) is the average reward and Q-values, respectively, for goal 2.

The Q-values for DDPG shows an initial decline eventually followed by a steady climb. It took about 100 and 70 episodes before it the average Q-value started to increase for goal 1 and 2, respectively.

For goal 2 DDPG showed a similar speed of increase of the Q-values as DQN. DQN was more than 50 episodes faster than DDPG to consistently improve, but looking at the start of the incline for both algorithms 8d show that difference of the rate at which the Q-values increase for the respective algorithm is minimal. If I compare the average Q-value at the start of incline for DQN and DDPG respectively with the Q-value 150 episodes later it shows that DQN was slightly in favor. The average Q-value for DQN increased by 9.587 in these 150 episodes while it increased by 9.258 for DDPG. So even though DQN was significantly faster initially to improve DDPG did increase at approximately the same rate once the Q-values started to increase.

7.4. Comparison

Overall DQN outperformed DDPG in all but one test, where they trained for 35000 steps on goal 2. Looking at the graph, figure 8b, it appears DQN and DDPG have a similar rate of improvement. However, DDPG did have a higher average episodic reward, 44.3 compared to that of 37.3 for DQN. As seen in table 2, DDPG achieved better performance according to all metrics except for the average Q-value for that test. DDPG also got a better time to goal duration in two more tests, the tests with 10000 and 35000 steps with goals 2 and 1, respectively. But showed a worse performance in all other metrics in those tests.

In the figures 6 and 7 it is clear that a training length of 5000- and 10000 steps was not enough for DDPG to learn a good enough policy. For the training lasting 35000 steps, DDPG only began to significantly improve after about 100 episodes for goal 1 and about 70 episodes for goal 2. Where DQN showed considerable progress in the first few episodes for goal 1 and after about 30 episodes for goal 2. The 35000 steps tests were the only ones where DDPG appeared to be converging to a close-to-optimal policy. Especially noticeable in tests with 5000- and 10000 steps, DQN trained faster and did not require as much experience as DDPG. DDPG did not reach an average episodic reward over 20 for all tests with 5000- and 10000 steps, while DQN almost reached an episodic reward of 100.

	Time to goal score		Best time to goal		Average episode reward		Average Q-value	
Test	DQN	DDPG	DQN	DDPG	DQN	DDPG	DQN	DDPG
5000 goal 1	33.9	0.204	4.64s	4.91s	74.8	-5.6	4.42	-3.98
5000 goal 2	3.96	0.273	7.19s	9.36s	11.8	3.2	3.37	-0.806
10000 goal 1	92.9	0.103	4.63s	9.70s	77.6	-0.882	2.99	-2.26
10000 goal 2	33.2	0.57	6.68s	4.73s	50.4	3.97	3.73	-3.08
35000 goal 1	447	372	4.60s	4.25s	86.7	41.6	6.81	2.16
35000 goal 2	50.8	124	7.21s	4.74s	37.3	44.3	5.5	3.44

Table 2: Table with the results for respective test and algorithm. The Test column specifies the number of steps and the goal for the training.

8. Discussion

The results show that DQN was able to solve the task successfully for all tests, even for the tests where the algorithms were only trained for 5000 steps. DDPG did only find a successful policy in the tests with 35000 steps. This indicates that DDPG needs more experience than DQN to solve the task of moving the drone from the start position to the goal. The experiments in [2] demonstrated, contrary to my results, that DDPG needed less experience than DQN to find solutions. A plausible explanation for the different results in [2] and those in this thesis are that the specific task that the algorithms are meant to solve can be more or less suitable for a specific algorithm.

As seen in the results, DDPG did way worse initially in every test with respect to average episodic reward and Q-value. One reason for this is insufficient exploration. The movement of DQN was more random and thus explored more. The agent of DDPG appeared to choose a direction and then continue going that direction for a number of episodes. The noise did affect the movement but it still went in the same general direction. To combat this more noise could be added to the action predicted by the actor network of DDPG. Choosing another noise function could also be beneficial. Ornstein–Uhlenbeck (OU) noise was recommended by the original paper but more recent research [24][25] found no benefit to using OU noise and instead used Gaussian noise. Gaussian is, unlike OU, uncorrelated noise, which means the values generated are not dependent on each other. This would lead to more diverse noise values which would result in more drastic velocity changes. Another alternative would be to use a fairly new concept termed parameter noise [26]. The traditional way of applying noise is to add it to the predicted action value, parameter noise instead applies it to the parameters of the network. Insufficient exploration could also explain why DDPG did not show competitive performance in the tests with low step count, see figures 6 and 7. If the agent is more prone to exploitation when it lacks experience it is less likely to randomly reach the goal since the agent will be more inclined to do what it already knows.

During the implementation phase some I found that DDPG was more sensitive to instability caused by inefficient hyperparameters compared to DQN. The parameters that were ultimately decided upon were the ones that simply seemed to work, i.e. those that allowed both DQN and DDPG to actually improve during training. More time would be necessary to thoroughly tune the different parameters, like learning rate, weight initializing, and variables related to the noise, to optimize the algorithms.

The time to goal metric that I used to compare the algorithms was based on the episode reward and the duration of the episode. This does give an indication of what algorithm found the quickest solution but since it is done post-training, it is not something that the algorithms can use to improve their time to goal results. An improvement would be to add another reward to the reward function that was based on the time it took for the agent to reach the goal.

These experiments were made using low-dimensional networks where the input was a one-dimensional array. It is possible that the results would differ for high-dimensional networks using, for example, sensory input. So the conclusions drawn from this research are only applicable to deep reinforcement learning algorithms with low-dimensional networks.

The action space was discretized simply by limiting the velocity in six directions. No further discretization was necessary since DQN, with this simple discretization, showed better performance in five of the six tests than DDPG.

9. Conclusions

This research aimed to investigate how the performance of two reinforcement learning algorithms with different action spaces, namely discrete and continuous, compare to each other when applied to an environment with continuous state- and action space.

The research questions were investigated by first conducting a simulation experiment where data was recorded each episode of the training of the algorithms. The data that was used for the comparison of DQN and DDPG was the average Q-value, episode reward, time to goal score, and the best time to goal. The data was then analyzed using the research method exploratory analysis.

The results showed that DQN, with a discrete action space, outperformed DDPG, with a continuous action space, when applied to an environment with a continuous state- and action space. Intuitively, the algorithm with a continuous action space would be more suitable for a continuous environment but as shown in this thesis this is not the case. DQN needed less experience than DDPG to find solutions for the tasks and it also appeared to be more stable. The only metric where DDPG showed competitive results was the best time to goal, where DDPG got a better time in three of the six tests.

It is clear from the results that the task of navigating a drone from A to B does not require a high resolution of actions to successfully tackle the problem. However, that is not the case for all tasks. Other problems may require a wide range of actions that are essential to the task and thus, can not be discretized without hindering the learning. For such problems, DDPG would be superior because of its continuous action space.

Based on these findings, the choice of reinforcement learning algorithm is not as simple as only looking at the state- and action space of the algorithm and environment. An algorithm with a continuous action space is not necessarily better than one with a discrete action space for a continuous environment, as shown in this thesis. The opposite is not always true either, which can be seen in [2]. To be entirely certain when choosing the most efficient algorithm it is necessary to test them on the task they are meant to solve. And choose the algorithm based on the results.

10. Future Work

This research was done using low-dimensional networks, an interesting contribution to this work would be to use sensory input from the drone together with high-dimensional neural networks. To see if similar conclusions can be drawn. Another interesting expansion of the work would be to expand the reward function to include a reward that is based on the time it took the agent to reach the goal. Then the algorithms get a chance to improve the time during training, which could lead to another interesting metric to base the comparison on:

- What algorithm was better at improving their time to goal?

By implementing this the work could be expanded to not only having different algorithms be compared but also having it compete against humans. Allowing the researcher to explore the research question:

- Can the reinforcement learning algorithm learn a policy that flies the drone to the goal faster than a human?

Future work should spend a significant amount of time tuning the network of the reinforcement learning algorithms and test a variety of different noise functions so that the algorithms can demonstrate their full potential.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, “Human-level control through deep reinforcement learning,” *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” *arXiv preprint arXiv:1509.02971*, 2015.

[3] J. Pazis and M. G. Lagoudakis, “Binary action search for learning continuous-action control policies,” in *Proceedings of the 26th Annual International Conference on Machine Learning*, 2009, pp. 793–800.

[4] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[5] C. J. Watkins and P. Dayan, “Q-learning,” *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[6] S. Russell and P. Norvig, “Artificial intelligence: a modern approach,” 2002.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[8] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, “Deterministic policy gradient algorithms,” in *International conference on machine learning*. PMLR, 2014, pp. 387–395.

[9] R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour *et al.*, “Policy gradient methods for reinforcement learning with function approximation.” in *NIPs*, vol. 99. Citeseer, 1999, pp. 1057–1063.

[10] D. Isele, R. Rahimi, A. Cosgun, K. Subramanian, and K. Fujimura, “Navigating occluded intersections with autonomous vehicles using deep reinforcement learning,” in *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 2034–2039.

[11] H. Van Hasselt and M. Wiering, “Using continuous action spaces to solve discrete problems,” 06 2009, pp. 1149–1156.

[12] Y. Tang and S. Agrawal, “Discretizing continuous action space for on-policy optimization,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, 2020, pp. 5981–5988.

[13] K. Säfsten and M. Gustavsson, *Forskningsmetodik: för ingenjörer och andra problemlösare*. Studentlitteratur AB, 2019.

[14] C. Wohlin, “Guidelines for snowballing in systematic literature studies and a replication in software engineering,” in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE ’14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi-org.ep.bib.mdh.se/10.1145/2601248.2601268

[15] M. Plappert, “keras-rl,” https://github.com/keras-rl/keras-rl, 2016.

[16] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” *arXiv preprint arXiv:1606.01540*, 2016.

[17] S. Shah, D. Dey, C. Lovett, and A. Kapoor, “Airsim: High-fidelity visual and physical simulation for autonomous vehicles,” in *Field and Service Robotics*, 2017. [Online]. Available: https://arxiv.org/abs/1705.05065

[18] Microsoft Research, “Welcome to airsim,” https://microsoft.github.io/AirSim/, 2018.

[19] OpenAI, “Openai gym,” https://github.com/openai/gym, 2016.

[20] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” 2017.

[21] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the thirteenth international conference on artificial intelligence and statistics.* JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

[22] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks,” in *Proceedings of the fourteenth international conference on artificial intelligence and statistics.* JMLR Workshop and Conference Proceedings, 2011, pp. 315–323.

[23] G. E. Uhlenbeck and L. S. Ornstein, “On the theory of the brownian motion,” *Physical review*, vol. 36, no. 5, p. 823, 1930.

[24] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. Tb, A. Muldal, N. Heess, and T. Lillicrap, “Distributed distributional deterministic policy gradients,” *arXiv preprint arXiv:1804.08617*, 2018.

[25] S. Fujimoto, H. Hoof, and D. Meger, “Addressing function approximation error in actor-critic methods,” in *International Conference on Machine Learning.* PMLR, 2018, pp. 1587–1596.

[26] M. Plappert, R. Houthooft, P. Dhariwal, S. Sidor, R. Y. Chen, X. Chen, T. Asfour, P. Abbeel, and M. Andrychowicz, “Parameter space noise for exploration,” 2018.