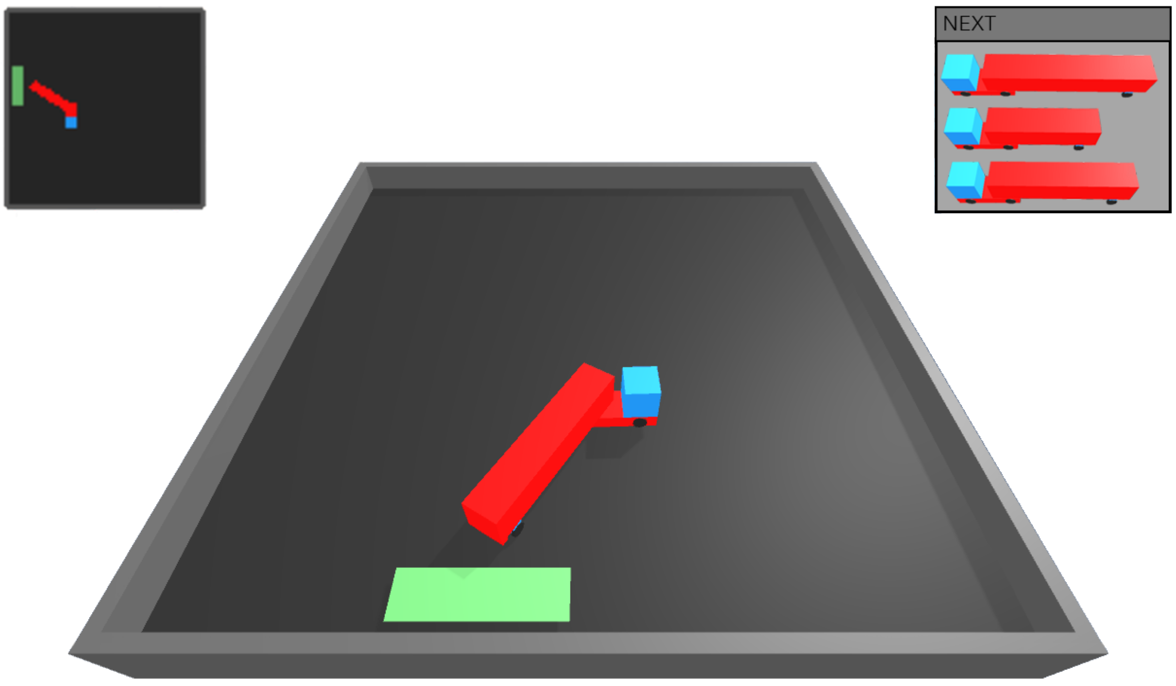
CHALMERS
UNIVERSITY OF TECHNOLOGY



Trailer docking using reinforcement learning with visual based input

Comparison of training complexity and generalisation performance

Master's thesis in Systems, Control and Mechatronics

ANTON ÖHAMMAR & SIMON MEDBO

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Master's thesis 2020

Trailer docking using reinforcement learning with vision based input

Comparison of training complexity and generalisation performance

ANTON ÖHAMMAR
SIMON MEDBO

Department of Electrical Engineering
Division of Signal processing and Biomedical engineering
Signal processing
Chalmers University of Technology
Gothenburg, Sweden 2020

Trailer docking using reinforcement learning with visual based input
Comparison of training complexity and generalisation performance
ANTON ÖHAMMAR, SIMON MEDBO

Supervisor: Andreas Andersson, Anton Öqvist, Aptiv
Examiner: Tomas McKelvey, Department of Electrical Engineering

Master's Thesis 2020
Department of Electrical Engineering
Division of Signal processing and Biomedical engineering
Signal processing
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Agent docking a trailer in Unity showing visual input and example of different trailer lengths.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2020

Trailer docking using reinforcement learning with visual based input
Comparison of training complexity and generalisation performance
ANTON ÖHAMMAR, SIMON MEDBO
Department of Electrical Engineering
Chalmers University of Technology

Abstract

An agent controlling a truck with an attached semi-trailer is trained to dock at a terminal using deep Reinforcement Learning (RL). The RL neural network input consists of pixel values coming from a top-down camera view and measured velocity of the truck. This thesis investigates how visual input combined with other observations affect the training, the ability to generalise to unseen scenarios and how training difficulty scales with increased complexity of the task. The RL network was successfully trained to control a semi-trailer combination to a set of chosen targets and to some extent also generalise to unseen scenarios in form of new semi-trailer lengths. It was shown how visual input benefits from being combined with certain non-visual measurements, e.g. angle between truck and semi-trailer, and how it can be degraded by using the wrong ones. Generalisation was primarily achieved by varying the simulation environment during training.

Keywords: Reinforcement Learning, pixel input, autonomous, docking, end-to-end, generalisation, complexity

Acknowledgements

The outcome of this thesis would not have been possible without the help of our supervisors, Andreas Andersson and Anton Öqvist at Aptiv, and our examiner Thomas McKelvey. Their expertise within control and experience in carrying out projects made sure that progress was always made and that results were reached in the end. Also the possibility to borrow high performance simulation hardware from Farid Kondori was essential to be able to finish within the given time frame.

ANTON ÖHAMMAR & SIMON MEDBO, Gothenburg, June 2020

Contents

List of Figures

List of Tables

Acronyms

ADAM	Adaptive Moment Estimation
ADAS	Advanced Driver-Assistance Systems
CNN	Convolutional Neural Network
DP	Dynamic Programming
DOF	Degrees of Freedom
GAIL	Generative Adversarial Imitation Learning
KL	Kullback Leibler
KPI	Key Performance Indicator
MC	Monte Carlo
MDP	Markov Decision Process
MPC	Model Predictive Control
ML	Machine Learning
MM	Minorise-Maximisation
POMDP	Partially Observable Markov Decision Process
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
SAC	Soft Actor-Critic
TD	Temporal Difference
TRPO	Trust Region Policy Optimization

Nomenclature

ϵ	Randomness factor
γ	Discount-rate
$\mathbb{E}[X]$	Expectation of a random variable
π	Policy
π^*	Optimal policy
π'	Updated policy
a	Action
G	Return (cumulative discounted reward)
J	Objective function
Q	Action-Value function
R_t	Reward at time (t)
s	State at time (t)
s'	State at time $(t+1)$
V	Value function

1
Introduction

1.1 Background

Computers are transforming our society and transportation is no exception. One apparent sign is the development of autonomous vehicles. This thesis is a collaboration with APTIV who are known for their development of Advanced Driver-Assistance Systems (ADAS) which assist drivers while driving or parking. To stay ahead of competitors they perform research and push towards finding better and more efficient solutions. ADAS are one step in the direction of a fully autonomous system, but further development is needed. In controlled delimited environments, such as trailer docking terminals, absence of civilians decreases safety risks and provides predictable environments for ADAS evaluation. Therefore it is a suitable application to try autonomous control, of a truck reversing to a docking terminal, based on unconventional methods such as RL with visual input.

With a known environment, including the dynamic vehicle model of the semi-trailer combination (truck and semi-trailer), traditional methods such as Model Predictive Control (MPC) can be applied to calculate a trajectory and control a truck. When required data for using these methods is unknown, one could instead use RL in a simulated environment, and get to know it by iteratively interacting with it [16] and thereby learn to provide nonlinear state feedback. Reinforcement learning is also well suited for using an image as input and therefore, instead of having one individual planner and controller in each truck, we propose a solution where a stand alone top-down camera view of a parking terminal and measured velocity is fed to a central network providing actions in form of steering and wheel torque to the truck in question.

1.2 Purpose

The overall purpose is to simplify autonomous terminal docking by training a policy capable of providing control to a truck with an attached semi-trailer. It will be done using methodology of RL, resulting in an understanding of the possibilities but also limitations of RL. Focus will also be placed on generalisation capabilities and complexity scaling of training in relation to the observation and task setup.

1.3 Objective

Throughout this thesis the following three questions will be investigated:

- How well does using visual based input work in comparison to also having non-visual measurements as input to a policy network?
  - E.g. what is the training progress difference between using only pixel data and using it in combination with measured position in two dimensions?

- How well can a trained policy network generalise and handle unseen scenarios when using visual observation?
  - E.g. To what extent can a policy network provide correct actions when interacting with scenarios that has not been presented during training?

- How does the complexity of training scale with increasing complexity of the agents task using visual observation?
  - E.g. controlling a truck with and without semi-trailer. How much more difficult it is for the training to converge, does it need additional techniques and how much longer does it take to learn correct behaviour?

The questions will be answered by applying RL on a simulated docking environment. Ensuring availability of data that can be utilised to answer our questions will be done by gradually increasing difficulty of the task that the RL agent will perform.

1.4 Scope

The outcome of this thesis will be an end-to-end RL solution, providing all necessary building blocks from an input top-down image of the terminal to steering and wheel torque input to the truck. Due to the given time frame, following limitations are specified.

The visual observation used when training the policy network will be a simplified version of what a real top-down image would look like, namely ideal homogeneous colours, no shadows and an orthographic projection. This implies that it will be less challenging for the policy network to make connections based on observations. Identification of a free terminal slot is assumed to be done on beforehand, by for example using deep learning [15]. The goal position will be marked as a patch in the observation image.

Overall parking terminal layout will stay the same throughout the whole thesis and the semi-trailer combination will always be initialised at the same position. There will be no attempt to train a policy that can provide correct actions for any arbitrary docking situation. Although attempts of generalising the policy network to handle changes in the environment will be carried out.

A real-world truck typically operates by steering, applying throttle, braking and changing gear. To reduce complexity of the problem propulsion control will consist solely of applying torque in forward and reverse direction on the rear wheels of the truck. Actions will be in form of a discrete vector action space with three values for steering and torque respectively, representing negative, zero and positive input.

Accuracy of the truck and semi-trailer models created in simulations will be of low importance, as the purpose is to analyse the usage of RL and not develop a controller capable of providing feedback to a specific semi-trailer combination. Although parameters and measurements such as wheelbase and weight will be in the correct order of magnitude to get a behaviour comparable to real-world scenarios.

Design of the network and reward model will not focus on generating a policy that is optimal when it comes to e.g. travelled distance, fuel consumption or smoothness. Focus will instead lie on reaching goal and if there is time for optimisation more complex parking scenarios will be prioritised.

2

Theory

2.1 Fundamental structure

RL is a growing research area in Machine Learning (ML). Although there are many different algorithms and techniques to apply, they all share the same basic theory and mathematical framework. That is, in a simulated environment, learning how to map state observations to actions by maximising a numerical reward leading to a goal as visualised in figure 2.1. What states that yield high reward is explored by an agent using a trial and error approach. What the agent has learnt is saved in a policy, that after completed training can be used to perform the simulated task. The policy can be compared to a controller that provides nonlinear state feedback.

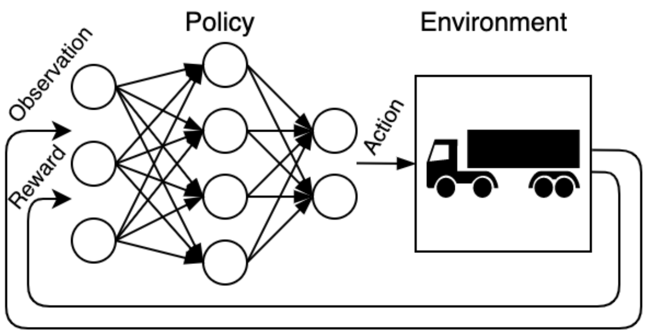


Figure 2.1: Simplified conceptual reinforcement learning

2.1.1 Markov Decision Process

Markov Decision Process (MDP) is a mathematical formalisation of sequential problems focused on modeling decision taking. Each state in a sequence is described by the symbol $\boldsymbol{s}$. A decision maker, within RL called agent, chooses action $\boldsymbol{a}$ to transition to the next state $\boldsymbol{s}'$ with probability $\boldsymbol{P}$. After performing an action and ending up in a state, the agent receives a predetermined reward $\boldsymbol{R}$ [21, pp. 47-68]. The model parameters can be summarised as:

- $\boldsymbol{S}$ - set of states describing the environment at different instances.
- $\boldsymbol{A}$ - set of possible actions that can be executed to transition between states.

- $\boldsymbol{P}$ - probability that action $\boldsymbol{a}$ leads to $\boldsymbol{s'}$ from state $\boldsymbol{s}$ in the next time step.
- $\boldsymbol{R}$ - reward that the agent receives after performing action $\boldsymbol{a}$ and ending up in a state $\boldsymbol{s}$.

Interaction between an agent and environment can for instance be the agent providing an action in terms of wheel torque and steering to an environment in which a vehicle moves. The environment responds with an observation of the state obtained, as well as a certain reward decided by a reward function. By interacting, the agent produces trajectories in the following form:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, ...R_n, S_n \tag{2.1}$$

where n is the final step in an episode, defined by the environment.

The sequential problem is characterised by the probability distribution $\boldsymbol{P}$, that given a state $\boldsymbol{s}$ and action $\boldsymbol{a}$ outputs a probability of the following state $\boldsymbol{s'}$ with associated reward $\boldsymbol{r}$. In a MDP formalisation of a driving truck, an action indicating positive wheel torque at time $t-1$ will generate a high probability of a state representing the truck located further ahead in the travel direction at time t.

$$p\left(s', r|s, a\right) = \Pr\left\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\right\} \tag{2.2}$$

Being able to formalise the problem as a MDP is crucial for RL. If the finite problem can be described by (2.2) anything in the environment can be computed, e.g. reward for a whole trajectory or transition probability in a certain state [21, pp. 47-68]. If the problem can be described by a MDP but the entire state is not directly observable for the agent, which is often the case in RL, it is called a Partially Observable Markov Decision Process (POMDP). The probability distribution then estimated by sampling experiences from the environment.

2.1.2 Reward

Reward is the incentive that directs an agent towards a specified goal or objective. The goal could be anything, e.g. keeping a pole upright by balancing it or reaching a certain position with a trailer. Reward is handed out to the agent every time step based on the state it reaches by taking an action. Reward is specified in the form of a value $R_t \in \mathbb{R}$.

The agent is trying to maximise reward by definition. This implies that the rewards have to be distributed in a way such that actions leading to maximum cumulative reward also leads to the goal or to completing some arbitrary mission [21, pp. 53-54]. Rewards are only used during training for the agent to learn. When training is done, it will know which states gave high reward and thereby know what action to take to get there.

When referring to a reward function, it implies multiple different conditions based on the current state that yield reward when met.

2.1.3 Observation

An observation describes the current state of the environment and is chosen differently depending on problem formulation. It is essentially what the agent bases decisions on and it is therefore vital that there is enough detail in the observation for a network to learn.

Notice that the agent takes actions using a policy, in a deep RL scenario where a neural network is used to estimate the policy, figure 2.2, the observation is what is used as input for the policy network which then outputs a distribution from which an action is sampled.

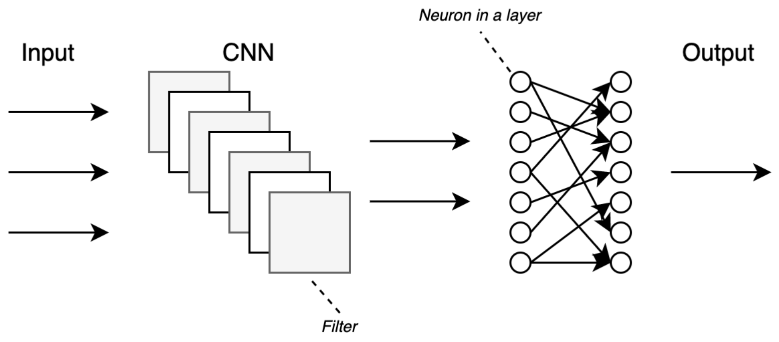


Figure 2.2: Simplified conceptual deep neural network with visual encoder

Common observations are e.g. position and rotation. When this type of data cannot be explicitly extracted one can instead use a visual observation, which in theory provides all above mentioned parameters but indirectly. Information is therefore difficult to extract. Using visual observation, a snapshot of the current observable environment state is taken, meaning that pixel intensity values make up the observation and is being fed to the neural network. As the number of input parameters are significantly larger compared to when using non-visual observations it becomes more difficult to train on. The policy network both has to learn how to extract important information hidden within the pixel data and then make use of it.

To extract information a Convolutional Neural Network (CNN) is applied to the observation as a first step. This technique is commonly used to analyse visual imagery within ML [3], where one applies a mathematical operation called convolution through multiple layers. Convolutional layers are trained simultaneously with the rest of the policy network to extract features in images on different levels. In layer one, CNN filters are trained to extract features such as blobs or edges and in later layers it can learn to extract whole objects. Feeding the CNN output instead of pixel intensity values to the hidden layers, building up the rest of the neural network, then drastically reduces complexity and enhances learning.

2.1.4 Policy

Choice of action is decided by a policy π. In a tabular case it can be a lookup table and in deep RL it is approximated by a function or neural network. In a more complex case, often continuous state space, it maps each observation to probabilities of choosing different actions (2.3) [21, pp. 58-59].

Learning implies updating the policy such that actions leading to the goal has high probability. How it is updated is specified by chosen RL algorithm.

$$\pi : S_t \rightarrow A_t \tag{2.3}$$

2.1.5 On- vs off-policy

The policy being optimised is referred to as the target policy. Training using an on-policy method implies that the target policy network is used to collect experiences while simultaneously being updated. I.e. exploration is only performed on states that the target policy leads to. This can shorten training times if the policy is in the process of converging to the optimal solution, but also result in local convergence as a strong bias is introduced [21, pp. 103-116]. In off-policy algorithms the network is optimising its target policy using experiences collected with a second behaviour policy. It gives the advantage that even if the target policy is deterministic the behaviour policy could be partly random and continue to explore the environment, which counteracts bias.

2.1.6 Objective function and optimiser

RL problems have a main objective of completing tasks set up by a user. In order to succeed, the agent will try to maximise accumulated reward during an episode. Therefore, functions are constructed to mathematically describe what is important in order to complete the task.

These are called objective functions and by optimising these, optimal behaviour will be learnt. E.g. if an objective function simply describes discounted cumulative reward, optimisation is done by maximising it throughout training. RL algorithms can also have multiple or combined objective functions. For instance minimising policy entropy to ensure stability by adding a separate objective function or inserting negative entropy to the previously mentioned reward objective.

How to perform optimisation has been a subject outside of RL for many years. One of the more popular iterative methods to optimise an objective function is stochastic gradient ascent. In later years, to improve performance in RL, alternative methods have been establish. As of 2014 an optimiser called Adaptive Moment Estimation (ADAM) was introduced for deep learning [1]. It is a combination of RMSprop (a gradient size adjustment technique) and stochastic gradient ascent with momentum. Since the release it has gained a lot of traction and it is common within machine learning optimisation since it in most applications drastically increases training performance.

The main difference is that ADAM, compared to other methods estimates first and second moments of the gradient to be able to adapt the learning rate separately for each weight in a neural network. The first moment is the mean and the second is non-centered variance.

2.1.7 Value functions and Bellman equation

A fundamental part of learning is to be able to evaluate a policy and the rewards it result in, since this provides a performance measurement. One especially has to take into account that a single action in an early stage will affect the following sequence of states. It is therefore favorable to be able to evaluate reward for future states based on being in a certain state. In RL one commonly uses a value function $V^{\pi}(s)$ to do that. It estimates how good it is to be in a specific state s at time t by accounting for the return G when following a policy π [21, pp. 58-59]:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[G_t | S_t = s\right] \tag{2.4}$$

where the return is the future discounted reward in an episode according to

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.5}$$

γ is a discount factor, stating what weight future rewards should have [21, pp. 58-59]. From (2.5) the return at time t can be derived as:

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{2.6}$$

To realise the value function and make it usable, it has to be expressed as value of a state and its successor states. This enables optimisation of subproblems which is one of the cornerstones in RL. For this the principle of Bellman equation is applied. It implies that the value is calculated by estimating expected future cumulative discounted reward. Combining (2.4) and (2.6) results in the bellman equation [21, pp. 58-59]:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[R_{t+1} + \gamma V^{\pi}\left(s_{t+1}\right) | S_t = s\right] \tag{2.7}$$

An alternative to the value function is the action-value function $Q^{\pi}(s, a)$.

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}\left[G_t | s_t = s, a_t = a\right] \tag{2.8}$$

It indicates the overall expected discounted reward when taking action a in state s and then following policy π until the end of an episode [21, pp. 58-59]. If the action value function is multiplied with the policy distribution and summed over all possible actions it translates to the value function:

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi_{\theta}(a|s) Q^{\pi}(s, a) \tag{2.9}$$

2.1.8 Dynamic programming

Dynamic Programming (DP) is based on breaking down a complex problem into smaller parts, optimise those and use the result to recursively solve the greater problem. In RL a value function, section 2.1.7, is used as a tool for an organised and structured search among sub problems in order to find the global optimal policy π^*, solving the problem.

DP can be used when the complete MDP is known including rewards, transition probabilities and therefore values of all states. Commonly this information is not known in RL applications, making DP not applicable in many cases. The theory is of high importance though as most RL methods are partially based on DP and the theory that one can split a complete trajectory into pieces where each can be optimised individually leading to a complete optimised trajectory [21, pp. 73-74].

2.1.9 Monte Carlo

In DP the complete MDP is known, but when it is not one has to find a way to learn without all necessary information available. Monte Carlo (MC) methods are a family of algorithms built on randomness. In RL they utilise an agent policy to explore an environment by sampling trajectories to fill the gap in lack of information [21, pp. 91-92]. Depending on what method is used to learn, the experience can either be stored in a buffer for later usage or immediately applied as a policy update. MC sampling suffers from high variance as episodes can be completely different from each other. A solution to this is to implement temporal difference learning.

2.1.10 Temporal Difference learning

While the idea of MC is to run complete episodes and sample these to recursively calculate values of each state visited, Temporal Difference (TD) enables updates after each time step. It is applying a form of bootstrapping by combining MC and DP. In a simple MC method the value estimate is updated using the return G_t while following a fixed policy π, which is first known at the end of an episode:

$$\mathrm{V}'(\mathrm{S_t}) \leftarrow \mathrm{V}(\mathrm{S_t}) + \alpha\left(G_t - \mathrm{V}(\mathrm{S_t})\right) \tag{2.10}$$

TD learning does instead make use of the reward received after each time step R_{t+1} and the value estimate $V(S_{t+1})$:

$$\mathrm{V}'(\mathrm{S_t}) \leftarrow \mathrm{V}(\mathrm{S_t}) + \alpha\left(\mathrm{R_{t+1}} + \gamma \mathrm{V}(\mathrm{S_{t+1}}) - \mathrm{V}(\mathrm{S_t})\right) \tag{2.11}$$

Using an estimate to update the value is more sample efficient as it doesn't require running a full episode before performing the update [21, pp. 119-120]. It also reduces the high variance that usually occurs when using MC. Estimated values will not differ from each other between episodes to the same extent, resulting in more stable training. However during initial phases of the training, value estimation can induce bias. This is because the value function estimation will be initialised randomly or to zero, resulting in incipient inaccuracies.

2.1.11 Importance sampling

Estimating the value function for a policy requires experience collection and is therefore an expensive process. Importance sampling is a method that can be applied in order to reduce computational complexity and increase sample efficiency. Instead of recollecting experiences after each policy update the current estimated value is mapped to the new policy using a probability ratio. It is calculated from the ratio between the probability distribution of the new policy π'_θ and current policy π_θ [13]:

$$V'_\theta = V_\theta \frac{\pi'_\theta}{\pi_\theta} \tag{2.12}$$

The policy probability distribution is then stored in a neural network. Keeping this copy in the memory is an inexpensive task compared to resampling the value function. In practice this means that the value function is sampled using π_θ and then mapped to π'_θ.

If the updated and old policy differ too much the estimation accuracy decreases. This flaw introduces a need to sync policies and resample the value function, how often is a parameter that has to be tuned. Compared to collecting new experiences after every policy update this is still a considerable improvement in sampling efficiency.

Importance sampling is a major building block in more advanced policy gradient methods such as Trust Region Policy Optimization (TRPO) and Proximal Policy Optimization (PPO).

2.1.12 Experience buffer

Complex problems yield many states and actions to choose from, especially if the state space is continuous. In such case one uses deep reinforcement learning with a neural network to approximate a solution to the problem. Training a network can be challenging with the possibility of premature local convergence. This is partly because data used during updates can be sampled from closely related experiences and introduce a strong bias.

Using an experience buffer one creates a buffer in which multiple episodes with state, action and rewards are stored [5]. By sampling a small batch, called mini-batch, from the buffer when updating the policy network, the data in use will be uncorrelated and training becomes more stable. Having multiple uncorrelated experiences sampled will reflect the actual problem to a greater extent, making it easier for the network to converge to the global optimum. Since multiple samples have to be collected before an update is performed, a drawback of the method is that it will slow down training, although it will most likely reach good performance faster.

Depending on if the algorithm uses on- or off-policy learning there are different ways of implementing an experience buffer, on- or off-policy is covered in section 2.1.5. If the policy is updated in a on-policy manner it is based on experiences only from the the current policy, therefore the buffer has to be completely emptied and refilled after each policy update, as visualised in figure 2.3.

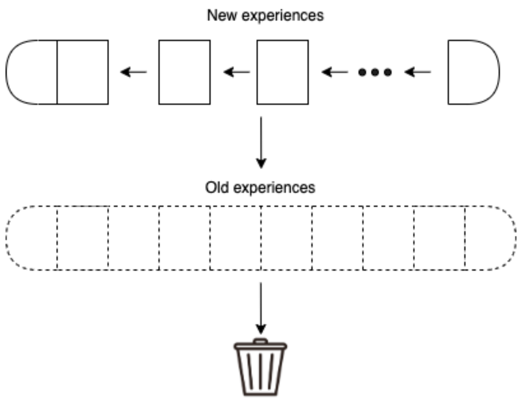


Figure 2.3: On-policy buffer example

In an off-policy method experiences are collected using an arbitrary policy and not specifically the most recent one. Therefore one instead e.g. continuously stores a buffer of fixed size by using the "one-in-one-out" principle. This is often referred to as using experience replay. Visualised in figure 2.4.

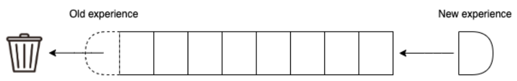


Figure 2.4: Off-policy buffer example

The size of this buffer is a tuneable parameter. If one has a complex problem it is important to make the buffer large enough to generate a distribution capable of describing the problem, at the cost of using RAM.

2.1.13 Curriculum Learning

Improving of a policy is generally done by updating it in a direction that leads to high reward. If the agents task is complicated, there is a low probability of discovering reward that leads to the goal when initially using a random policy, as one most often starts with. It would require a lot of training before the agent by chance performs the correct sequence of actions, if it happens at all.

By training on a simple problem to start with and learn a basic behavior, this behaviour can be used to increase the probability of discovering more inaccessible rewards when making a task more difficult. Suddenly the initial problem that at first was almost impossible is manageable. Performing changes in the problem formulation and stepwise slowly increasing complexity is called curriculum learning [4, p. 324]. This can both reduce training time considerably and create new opportunities of what an agent can learn.

2.1.14 Imitation Learning

Imitation learning is partially dealing with the same problem as curriculum learning, section 2.1.13, i.e. complex problems with rewards that are hard to find by random exploration. Instead of letting the agent struggle with learning from sparse rewards, a demonstration with recorded trajectories leading to the goal is used during a chosen number of steps, to initially help the learning process. It both enables the agent to find a trajectory towards goal and speeds up training. An operator demonstrates by performing the task multiple times, simultaneously state action pairs are recorded. The agent is then partially imitating this behaviour and learns from it by observing corresponding rewards. This is what is called imitation learning [6].

An alternative to imitation and curriculum learning would be to design a reward function that provide rewards to the agent in all possible states, guiding the agent when it is about to get lost, but also providing meticulous reward close to the goal in order to reach desired precision. Developing such a competent reward function is challenging and there are usually lots of ways for the agent to exploit it. Therefore the two alternative methods described are very versatile and efficient.

2.2 Algorithms

2.2.1 Value-based methods

Value-based RL methods learn by estimating how good it is to be in certain states, i.e. estimating values. With this, the policy generally selects actions transitioning to the states with largest values. Figure 2.5 illustrates how S_3 is chosen over S_2 and S_4 as it has a larger value. The same concept applies for S_7 and S_{10}.

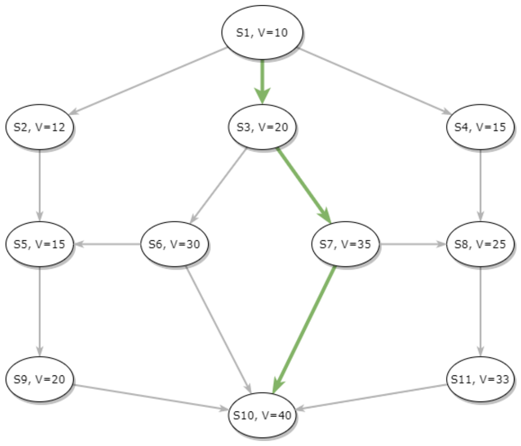


Figure 2.5: Value based learning

Using a value based method, the objective is to optimise the value function, which a policy then follows [21, pp. 9-11]. The policy is implicitly updated by improving value estimates of states in the environment, since this will alter decisions. It can e.g. act greedily or epsilon-greedily upon state value estimations. Epsilon-greedy implies that the agent explores the environment by taking partially random actions, chosen by a parameter ϵ. Estimating state values can be done with various methods, but they all suffer from one drawback. In environments with large state spaces, it is very computationally expensive and nearly impossible to estimate values of all states, making value based RL not suitable to solve all types of problem formulations.

2.2.2 Policy gradient methods

Instead of estimating values of all states and indirectly utilising that as decisive, policy gradient methods are based on directly mapping state observations to actions. The policy network is updated by following gradients with respect to the policy itself, which gives the advantage of not having to store resource heavy value estimations of irrelevant states. Only values useful for policy evaluation are estimated. This increases efficiency and make continuous state spaces manageable.

Also, the policy gradient method is by default well suited to manage continuous state and action space as it generates a probability distribution mapped to actions, from which one samples. Finally the need for additional exploration method such as acting epsilon-greedily in order to explore the environment vanishes as actions, either discrete or continuous, can be stochastic through sampling and thereby enhance exploration naturally. This leads to policy gradient methods being favorable in more advanced, higher dimensional problem formulations. There are multiple methods to perform the actual update of policy network weights, all attempting to generate a policy that results in a trajectory giving maximum reward to the agent.

2.2.2.1 Fundamental policy gradient - REINFORCE

The policy π_θ in a deep RL problem, is made up of a neural network containing weights and biases, making up the model parameters θ. Improvement of π_θ is done by, during training, altering the parameters θ such that the objective function (2.13) is maximised, more about objective functions in section 2.1.6. Using gradient ascent, the parameters can be changed in direction of the objective function gradient with respect to the policy $\nabla_\theta J(\theta)$, thereby finding optimum of the objective.

The basic objective function in policy gradient methods is defined as:

$$J(\theta) = \sum_{s \in \mathcal{S}} d^{\pi_\theta}(s) V^{\pi_\theta}(s) = \sum_{s \in \mathcal{S}} d^{\pi_\theta}(s) \sum_{a \in \mathcal{A}} \pi_\theta(a|s) Q^{\pi_\theta}(s,a) \tag{2.13}$$

where $d^{\pi_\theta}(s)$ is the stationary distribution of policy π_θ, thus probability of reaching state s when starting from state s_0 and following policy π_θ. $V^{\pi_\theta}(s)$ is the value of state s when following policy π_θ [18], which can also be expressed in terms of the action value function, see (2.9).

Calculating $\nabla_\theta J(\theta)$ is tricky since it depends on the policy through both the stationary policy distribution and the value estimation. d^{π_θ} would have to be updated each time the policy is updated, but with an unknown environment it is not possible without collecting an unreasonable large number of samples. In order to solve this problem, the policy gradient theorem comes in handy. It enables us to remove the derivative of d^{π_θ} when calculating the objective function gradient (2.13) using proportionality. The theorem states that the gradient can be rewritten as:

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_{s \in \mathcal{S}} d^{\pi_\theta}(s) \sum_{a \in \mathcal{A}} Q^{\pi_\theta}(s,a) \pi_\theta(a|s) \propto \sum_{s \in \mathcal{S}} d^{\pi_\theta}(s) \sum_{a \in \mathcal{A}} Q^{\pi_\theta}(s,a) \nabla_\theta \pi_\theta(a|s) \tag{2.14}$$

Derivation of the policy gradient theorem can be found in [21, p. 324].

Multiplying (2.14) with $\frac{\pi_\theta(a|s)}{\pi_\theta(a|s)}$ and using $\nabla \ln f(x) = \frac{\nabla f(x)}{f(x)}$ the objective gradient can be further simplified:

$$\nabla_\theta J(\theta) = \sum_{s \in \mathcal{S}} d^{\pi}(s) \sum_{a \in \mathcal{A}} \pi_\theta(a|s) Q^{\pi}(s,a) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \tag{2.15}$$

$$= \mathbb{E}_\pi \left[Q^\pi(s,a) \nabla_\theta \ln \pi_\theta(a|s) \right] \tag{2.16}$$

where $\mathbb{E}_\pi$ is the expected value following policy π_θ, corresponding to $s \sim d_\pi$, $a \sim \pi_\theta$.

With an expression of the objective function gradient in place, gradient ascent and thereby θ updates can be performed. The most basic policy gradient algorithm is called REINFORCE. It uses the entity that the expectation of a sampled gradient is equal to the true gradient [20]. By applying (2.8) on (2.16) the gradient becomes:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[Q^\pi(s,a)\nabla_\theta \ln \pi_\theta(a|s)\right] = \mathbb{E}_\pi \left[G_t \nabla_\theta \ln \pi_\theta \left(A_t|S_t\right)\right] \tag{2.17}$$

G_t can be estimated by exploring and sampling trajectories. The update is performed according to:

Algorithm 1 REINFORCE

```
1: Initialise the policy parameter $\theta$ randomly
2: Collect trajectory in on-policy fashion: $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, \ldots R_n, S_t$
3: for $t = 1, 2, 3, ..., T$ do
4:     Estimate return $G_t$
5:     Update the policy parameters $\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta \ln \pi_\theta \left(A_t|S_t\right)$
```

Since exploration is done using MC method, section 2.1.9, the gradient can suffer from high variance. This can make it difficult for the gradient ascent to converge. The problem originates from having a stochastic policy which may take completely different actions and thus trajectories in each episode. By using an experience buffer, explained in section 2.1.12, when performing one step of gradient ascent the variance can be reduced.

Further improvements of basic REINFORCE is to calculate an advantage instead of the action value Q^π, in the gradient ascent update. If the average action from a state results in a cumulative discounted reward of e.g. 500 and a specific action results in 502, the majority of weight updates does not contribute to determining if the specific action is better than the average. By subtracting a baseline the advantage becomes 2. This has a larger signal to noise ratio and is easier to make use of. Advantage, A^π, is calculated by subtracting the state value from the action value:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \tag{2.18}$$

as $V^\pi(s)$ does not depend on a, using advantage does not affect the functionality of REINFORCE, updates just become more significant [20].

Many more improvements can be done but those are significant enough to be categorised as new algorithms. Before covering other algorithms, tuneable hyperparameters that are used in most policy gradient algorithms need to be introduced.

2.2.2.2 Hyperparameters

Most policy gradient algorithms are similar. The following hyperparameters are commonly tuned to improve learning.

- Experience sample collection related:
  - **Buffer Size**, how many experiences that are collected before a policy update is performed, in an on-policy method. In an off-policy method it decides how many samples to continuously store in the buffer.

  - **Mini-batch/Batch Size**, how many samples from the experience buffer that are drawn for the gradient ascent.
  - **Horison**, Utilising a buffer, the horison decides how many steps of experience to collect each episode before adding it to the experience buffer. The future expected return outside of this horison will be estimated using the value function. It has similar meaning as TD, section 2.1.10. This parameter can be of different importance depending on what algorithm is used.
  - **Epoch**, how many passes that are run through the experience buffer during gradient descent.

- Reward related:
  - **Reward discount factor** γ, a weight factor of future rewards used when estimating values and calculating return.
  - **Generalised advantage estimation coefficient** λ, decides how much the agent relies on current value estimate versus actual received reward when updating the value estimate.

- Policy update related:
  - **Clip parameter** ε, a limit of how much a policy probability distribution is allowed to change during update.
  - **KL divergence coefficient** β, a measurement of how much a policy probability distribution is allowed to differ from the old one.
  - **Learning rate**, the strength of each gradient descent update.

- General:
  - **Max Steps / Ending Condition**, what triggers the end of an episode. Can be set to a fixed number of steps or any other condition such as reaching a certain reward or state.
  - **Network Structure**, structure of the neural network including number of convolutional layers, hidden layers and neurons in each hidden layer.

2.2.2.3 Trust Region Policy Optimisation

Ordinary policy gradient ascent does not constrain the policy updates, which can lead to large unstable updates. Another disadvantage is that since it is an on-policy method it has to resample the value function after each policy update. This leads to poor sample efficiency. See section 2.1.5 for explanation of on-policy. These issues are addressed by an algorithm called TRPO [24].

Regular gradient ascent takes update steps in direction of the objective function to find an optimum. In TRPO a trust region that limits the step size is introduced. It prevents updates from overshooting the optimum and potentially deteriorating the policy. If the divergence between old and new policy is large, the trust region is reduced and vice versa. This assures a stable but still fast convergence. Enforcing the actual trust region is done by including Kullback Leibler (KL) divergence in the optimisation problem. KL is a measure of the difference between two distributions P and Q, in this case between the new and old policy:

$$D_{KL}(P\|Q) = \sum_{x=1}^{N} P(x) \log \frac{P(x)}{Q(x)} \tag{2.19}$$

Sample efficiency is also improved by introducing importance sampling which is explained in section 2.1.11. In order to optimise the policy for maximum reward the trust region and importance sampling is combined in a Minorise-Maximisation (MM) algorithm.

The main objective is to maximise expected discounted future reward for a complete episode. Using equation (2.5) and summing over all time steps in an episode results in:

$$\max_{\theta} J\left(\pi_{\theta}\right) = \mathrm{E}_{\pi_{\theta}}\left[\sum_{t=0}^{\infty} \gamma^{t} R_{t}\right] \tag{2.20}$$

From this stage equations become more complex. To make them more compact and readable the policy π_θ that is dependent on parameters θ will be denoted by π. Maximising the objective $J(\pi)$ can also be rephrased to maximising improvement between each policy update, the updated policy is referred to as π':

$$\max_{\pi'} J\left(\pi'\right) = \max_{\pi'}\left(J\left(\pi'\right) - J(\pi)\right) \tag{2.21}$$

The MM algorithm uses a lower bound surrogate function $M(\pi)$ to optimise the main objective $J\left(\pi'\right) - J(\pi)$. This ensures stable improvements each iteration of the algorithm. Since $M(\pi)$ is a lower bound function it would have to break the lower bound constraint in order to decrease the value of $J\left(\pi'\right) - J(\pi)$. Each iteration the policy is updated according to the optimum of $M(\pi)$ until it converges to a local or global optimum.

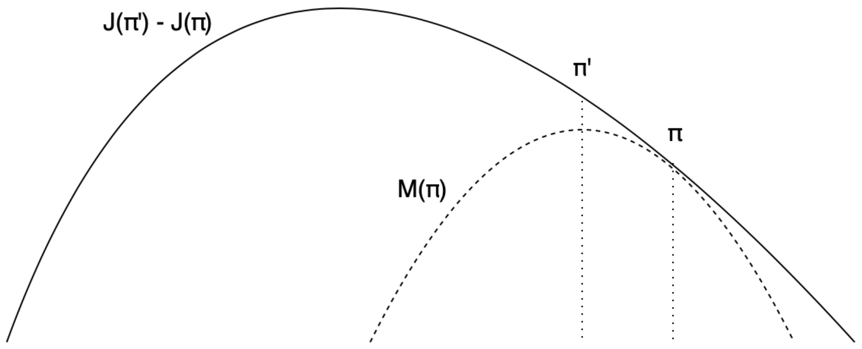


Figure 2.6: Example of Minorise-Maximisation algorithm optimisation

Next step is to construct the lower bound function $M(\pi)$. Instead of using future discounted reward as in the objective, future discounted advantage is utilised. The advantage, now being estimated using importance sampling, is calculated. It is referred to as $\mathcal{L}$:

$$\mathcal{L}_\pi(\pi') = \mathrm{E}\left[\sum_{t=0}^{\infty} \gamma^t \frac{\pi'(a_t|s_t)}{\pi(a_t|s_t)} A^\pi(s_t, a_t)\right] \tag{2.22}$$

Lower bound property is achieved by introducing KL divergence between the new and old policy and setting up the following equation:

$$|J(\pi') - (J(\pi) + \mathcal{L}_\pi(\pi'))| \leq C\sqrt{\underset{s\sim d^\pi}{\mathrm{E}}\left[D_{KL}(\pi'\|\pi)[\mathrm{s}]\right]} \tag{2.23}$$

Where D_{KL} is the KL divergence between the new and old policy, C is a tuneable parameter and d_π is the stationary policy distribution. Derivation proving that (2.23) assures lower-bound property in relation to the main objective can be found in the original TRPO paper [24].

Solving for $J(\pi') - J(\pi)$ results in:

$$J(\pi') - J(\pi) \geq \underbrace{\mathcal{L}_\pi(\pi') - C\sqrt{\underset{s\sim d^\pi}{\mathrm{E}}\left[D_{KL}(\pi'\|\pi)[s]\right]}}_{M} \tag{2.24}$$

M is then the RHS of (2.24). By iteratively optimising M the maximum of $J(\pi') - J(\pi)$ can be found. The optimisation is referred to as being KL penalised. By tweaking the optimisation problem one can get something that is instead referred to as KL constrained:

$$\max_{\pi'} \mathcal{L}_\pi(\pi') \text{ s.t. } \underset{s\sim d^\pi}{E}\left[D_{KL}(\pi'\|\pi)[s]\right] \leq \delta \tag{2.25}$$

mathematically the objective is the same, but instead of tuning C, δ is tuned which is a hard constraint and therefore easier to adjust.

2.2.2.4 Proximal Policy Optimisation

Similar to TRPO, PPO aims to increase stability of the policy update by avoiding too large parameter updates. However PPO tries to simplify the constraint. It is usually done in two different ways, either limiting the objective gradient by clipping it or penalising the KL divergence of the objective function [17].

PPO starts with an estimate of the advantage using importance sampling, in the same way as TRPO. The most common strategy to reduce parameter updates is using clipping.

$$J(\theta) = \mathbb{E}\left[r(\theta)A(s,a)\right] \tag{2.26}$$

where $r(\theta)$ is the probability ratio between old and current policy.

$$r(\theta) = \frac{\pi'(a|s)}{\pi(a|s)} \tag{2.27}$$

A large policy change will result in equivalent increase or decrease of the probability ratio (2.27). A function that clips the probability ratio if it deviates more then ε from equilibrium is introduced:

$$\mathrm{clip}(r(\theta), 1-\varepsilon, 1+\varepsilon) \tag{2.28}$$

By changing the objective into taking the minimum of (2.26) and (2.28) the new objective becomes a lower bound to the original one. The incentive to make large parameter changes has then been removed:

$$J^{\mathrm{CLIP}}(\theta) = \mathbb{E}\left[\min\left(r(\theta)A(s,a), \mathrm{clip}(r(\theta), 1-\varepsilon, 1+\varepsilon)A(s,a)\right)\right] \tag{2.29}$$

It is not trivial how (2.29) ensures that policy changes are kept small, but by doing some simplifications it is easier to get a better understanding:

$$\begin{aligned} J^{\mathrm{CLIP}}(\theta) = \mathbb{E}\left[\min\left(r(\theta)A(s,a), \mathrm{clip}(r(\theta), 1-\varepsilon, 1+\varepsilon)A(s,a)\right)\right] \rightarrow \\ \mathbb{E}\left[\min\left(r(\theta)A(s,a), g\left(\varepsilon, A(s,a)\right)\right)\right] \end{aligned} \tag{2.30}$$

where:

$$g(\varepsilon, A) = \begin{cases} (1+\varepsilon)A & A \geq 0 \\ (1-\varepsilon)A & A < 0 \end{cases} \tag{2.31}$$

for derivation of (2.30)-(2.31) see [23].

If the advantage A is positive the simplified objective (2.30) becomes:

$$\mathbb{E}\left[\min\left(r(\theta), (1+\varepsilon)\right)A(s,a)\right] \tag{2.32}$$

With a positive advantage the probability ratio $r(\theta)$ increases. If it becomes larger than $1+\varepsilon$ the minimise function limits the objective function value to $1+\varepsilon$. Therefore the updated policy does not benefit from large changes. If the advantage instead is negative the minimisation problem becomes a maximisation problem and reduces to:

$$\mathbb{E}\left[\max\left(r(\theta), (1-\varepsilon)\right)A(s,a)\right] \tag{2.33}$$

again the same mechanism limits the policy update. When both the policy and value function are estimated with a common network, the value function loss has to be added to the objective in order to improve value estimation which is critical when estimating advantage. Also an entropy term is added which serves as a regulariser. When the entropy is large all actions are equally likely and when it is low one action is dominant. By adding an entropy term to the objective it prevents one action from becoming too dominant, encouraging exploration. The final clipped PPO objective becomes:

$$J^{\mathrm{CLIP}'}(\theta) = \mathbb{E}\left[J^{\mathrm{CLIP}}(\theta) - c_1\left(V(s) - V_{\mathrm{target}}\right)^2 + c_2 H\right)\Big] \tag{2.34}$$

where c_1 and c_2 are tuning parameters for value function loss and entropy.

2.3 Generalisation techniques

Generalisation in RL is commonly defined as training a network in such a way that it performs well on data that has not been presented during training, e.g. colour or geometrical changes in the environment [11]. Intuitively it comes down to learning something larger that is useful beyond the exact details shown by the training MDPs. Training on MDPs that are representative of the real problem and learning underlying functions one can try to learn how to perform well in scenarios that has not been presented to the agent.

In supervised learning, something called overfitting is a well-known problem, as it is often clearly pronounced when testing a trained neural network. It occurs when the network fits too well to the training data and therefore does not fit the actual data distribution. This is exemplified by the black linear trend and the overfitted red curve in figure 2.7. Until recently, RL has mostly been applied on tasks that are not changing and with extensive training, the state space has been covered to a great extent. I.e. the training environment has been similar to the one used for evaluation, such that overfitting has not been a problem. If the goal instead is to train a policy with the possibility to generalise and it is shown to only perform well on one specific setting of the environment, it can be seen as overfitting. A policy network with as little overfitting as possible, while still succeeding to complete the task, could possibly handle unseen changes in the environment. Generalisation is therefore an overfitting problem.

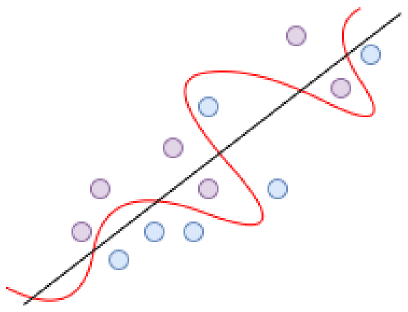


Figure 2.7: Overfitting example. A good data point split represented by the black line. The red curve is overfitted to the samples.

While learning how to dock a trailer, instead of being taught to always turn left when e.g. observing walls in a certain position, the agent should learn a more meaningful behaviour based on what is actually important, i.e. not crashing and reversing towards the goal. A simplified interpretation could be that one of the data points to which it overfits to in figure 2.7 corresponds to the wall in question. The appearance of a wall close to the goal might just be a coincidence. If the goal is moved in the next instance, the agent will not know how to find it because features of the wall has stronger learning signals then the actual goal. Therefore it is of great importance to try to decouple learning signals from irrelevant features in the observation and make sure features that are connected to reaching the goal are heavily weighted. E.g.

observing the goal should result in actions leading to that area. Other features that happen to be in frame should not be the main ones leading to the correct actions.

Learning such a policy and not overfitting to irrelevant features is complex since it can not be explicitly imprinted. There are multiple methods to obtain more general training, which can result in a more usable and robust policy. Some of them will be covered in the following sections.

2.3.1 Size of neural network

To solve an arbitrary RL problem, a neural network architecture has to be large enough to allow learning, i.e. have enough Degrees of Freedom (DOF). Too few neurons will make it impossible to build consistent connections from observations to values or actions. As can be seen in figure 2.7 more DOF also gives a network the possibility to overfit data, although it could also allow for learning of more complex behaviours which could benefit unseen states and scenarios. A possible approach can therefore be to create a network with sufficient capacity, then successively decrease network size and train, to possibly encourage generalisation as overfitting will be less prone [10].

2.3.2 Regularisation

Different techniques to counteract overfitting are commonly gathered under the expression regularisation. In supervised learning, performance is normally measured on a data set different from the training data. Developing methods to avoid overfitting to the training data have therefore been a central subject within supervised learning. Some methods can successfully be deployed on RL problems.

2.3.2.1 Dropout

All internal connections in a policy network result in co-dependencies between neurons. It can obstruct the functionality of each individual neuron and possibly cause overfitting. By dropping (disregarding) a subset of all neurons during each parameter update strong dependencies could be prevented and also the risk of overfitting [4, pp. 224-270]. Which neurons that are dropped is decided by a dropout probability.

2.3.2.2 L1 & L2

L2 parameter regularisation, also known as weight decay is a strategy where the L_2 norm of network parameters is added to the loss function. Large weights will be penalised reducing the risk of overfitting certain features. Another method is L_1 regularisation. It adds the absolute value of network parameters to the loss function [4, pp. 227-233]. Principally both methods are the same and somewhat similar to the idea of dropout, section 2.3.2.1.

2.3.2.3 Batch Normalisation

Batch normalisation could both have an effect on training performance and reduce the risk of overfitting. Before feeding input to a network one usually wants to normalise the input. This ensures that all data is the same order of magnitude which counteracts imbalanced gradients and increases learning. The same advantages can be gained between each layer in a neural network and similarly to dropout it will allow independent learning [7] between each layer, which speeds up learning. Batch normalisation of the output H_{out} from a hidden layer is done according to:

$$H'_{out} = \frac{H_{out} - \mu}{\sigma} \tag{2.35}$$

where μ is batch mean and σ standard deviation.

2.3.2.4 Data augmentation

By augmenting visual observation data a greater variety in the policy network input is obtained, which can generate a more robust policy [9]. There are many different ways of augmenting an observation. One suggestion is to mask rectangles of varying size and colours in the observation, according to [8] it was shown to achieve great performance.

In another attempt to improve generalisation using visual observation it is suggested to augment input data using an extra convolutional layer between observation and policy network [12]. Each episode the convolutional layer is randomly initialised. Weights are normalised in order to not disturb the observation excessively and risk loss of important information. This method automatically creates an increased variance in training data, which makes the policy less prone to overfitting irrelevant visual features in observations. It is also referred to as domain randomisation and is commonly used to bridge the gap between simulation and real-world applications.

The method has one major downside. It produces a policy that generally has a high variance in performance. In order to compensate for this, it is suggested that a feature matching term is added to the objective function. It is calculated by feeding both the original observation and an augmented version to the network separately and from this calculate mean square error between outputs from the final hidden layers [12]. This will encourage attention to the same visual features for both the augmented and original input and therefore reduce variance that is otherwise added to the policy.

2.3.2.5 Varying Environment

In previous research on RL algorithms specifically designed to generalise well, it was shown that regular RL algorithms such as PPO could outperform dedicated generalisation algorithms. Instead, introducing variations in environments during training gave better results [11].

Variation implies that environment parameters are changed during training, e.g. goal position or kinematics, with an ambition to train a policy that can interpolate between these and in that way generalise.

3
Method

Analysing effects of using visual input to a policy network as well as generalisation capabilities required a thoughtful experiment setup. A lot of resources were also needed to perform these RL experiments, a simulation environment, a RL algorithm, tuning of the algorithm and lastly an approach to evaluate results.

The game engine Unity, was used to develop an environment containing a docking terminal as well as a semi-trailer combination with close to realistic dynamics. A Unity developed plugin called ML-agents was used along with the environment to utilise a RL algorithm and train a RL agent. Training implied running the simulation, interacting with the environment and learning from gained experience, as described by figure 3.1. Different iterations of training setups were carried out to collect a basis for analysis.

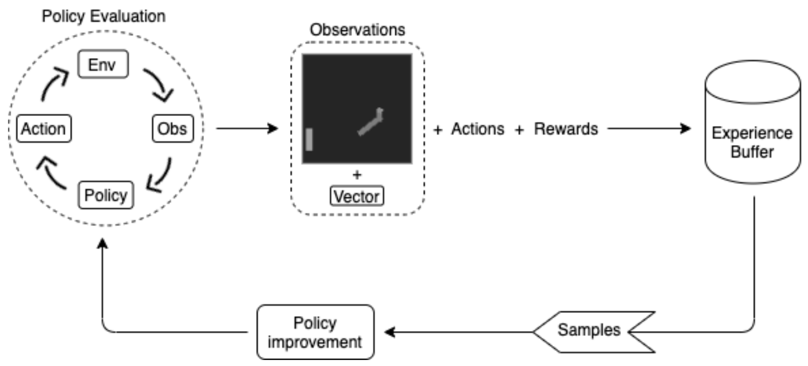


Figure 3.1: Conceptual schematics of reinforcement learning training

3.1 Simulation environment

Many well-established research platforms in RL are based on games or game engines. In fact, one of the most famous examples is a neural network learning to solve different classic ATARI games [2], which gained a lot of traction when it was published in 2013. According to some, this was a discovery that started the exposure for deep RL in recent years. Games are well suited for developing RL algorithms since all of them in one way or another has an interactive environment and can give rewards based on performance. Unity engine is a well-known platform with multiple famous game titles released. In 2017 they released a RL plugin called ML-Agents which has gotten many updates since then and there exists a large community where lots of information can be shared among users.

The engine supports builds with extremely detailed graphics and advanced physics. Even though this could help train a network with the possibility of also working in a real-world application, at an experimental stage, it would only make training more difficult. Therefore a three-dimensional environment with a continuous state space was designed with as simple shapes and environmental dynamics as possible. Both truck and semi-trailer were created out of cuboids onto which wheels, steering and propulsion capabilities were attached, as shown in figure 3.2.

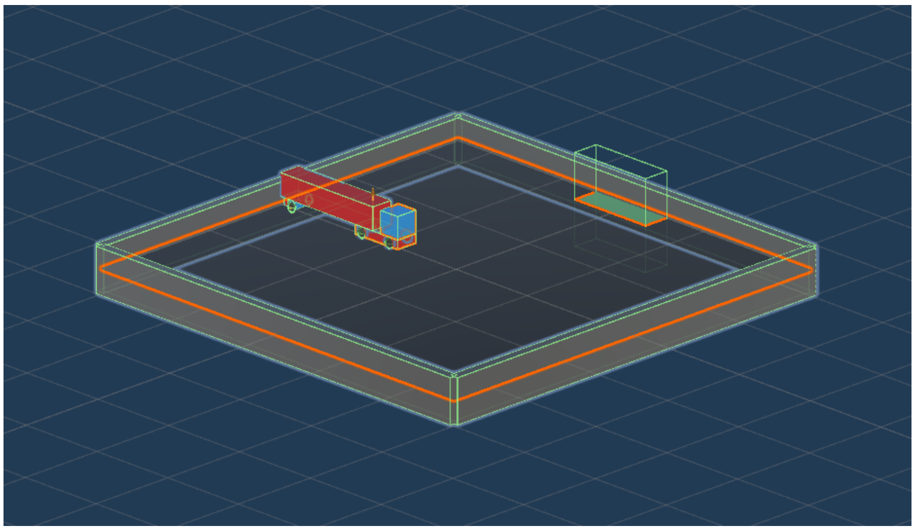


Figure 3.2: Building blocks in the environment

Dimensions of the semi-trailer combination were not critical as it would not contribute to the outcome of the thesis. A Volvo FH16 was used as a guideline for proportions from which a simple truck with only one rear wheel axle, as well as a semi-trailer with a single axle was created. Initially the semi-trailer length was set to 13 meters but in order to perform generalisation experiments, it was changed according to specific cases. While switching between semi-trailer lengths, the link point was always placed 1.25m from the front of the semi-trailer and the wheel axle 1.5m from the rear, see figure 3.3.

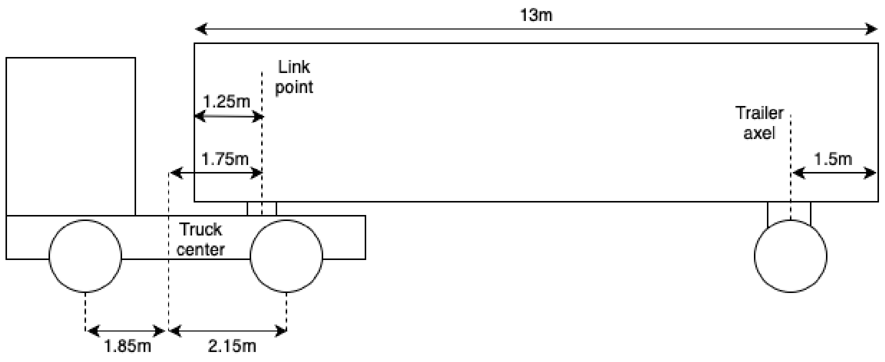


Figure 3.3: Drawing of semi-trailer combination from the left side

A simplified vehicle model of the semi-trailer combination is shown in figure 3.4. The torque τ was limited to $\pm 800 Nm$ and the steering angle α_1 to $\pm 40°$.

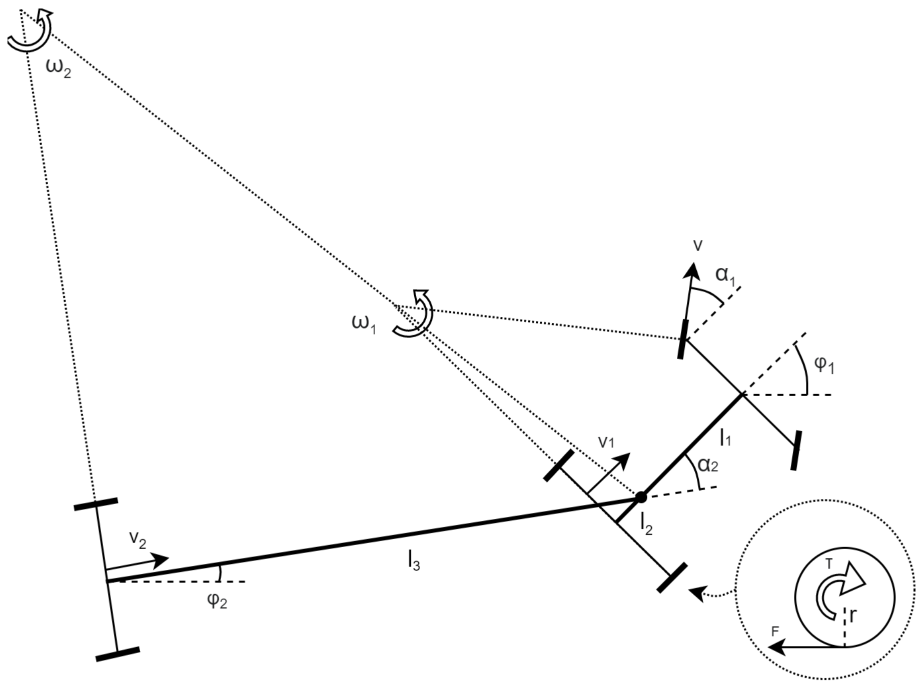


Figure 3.4: Simplified drawing of semi-trailer combination dynamic vehicle model

The dynamic vehicle motion model applied in Unity was highly complex, but it can be simplified as in equation 3.1 to 3.3 according to [14]. These equations show how, in principle, the applied inputs were converted to motion to the designed semi-trailer combination where $r = 0.4m$ and lengths were set according to figure 3.3.

Equation 3.1 shows acceleration as a result of propulsion where inertia and slip is disregarded. Reaction force from torque applied on the wheel is transformed into acceleration $\dot{v}_1$:

$$\dot{v}_1 = \frac{F}{m} = \frac{T}{mr} \tag{3.1}$$

Equation 3.2 shows motion of the truck:

$$\begin{bmatrix} v_1 \\ \omega_1 \end{bmatrix} = \begin{bmatrix} v \cos \alpha_1 \\ (v \sin \alpha_1)/l_1 \end{bmatrix} \tag{3.2}$$

Equation 3.3 shows motion of the semi-trailer:

$$\begin{bmatrix} v_2 \\ \omega_2 \end{bmatrix} = \begin{bmatrix} \left(\cos \alpha_2 \cos \alpha_1 + \frac{l_2}{l_1+l_2} \sin \alpha_2 \sin \alpha_1\right) v \\ \frac{1}{l_3} \left(\sin \alpha_2 \cos \alpha_1 - \frac{l_2}{l_1+l_2} \cos \alpha_2 \sin \alpha_1\right) v \end{bmatrix} \tag{3.3}$$

The docking terminal was based on a $50m$ x $50m$ plane with an initially $4m$ wide rectangle marking the goal and walls acting as outer bounds, as can be seen in figure 3.5. The semi-trailer combination had a fixed starting position and the goal could have different widths and be placed at 33 different positions, evenly spaced along the $50m$ wall.

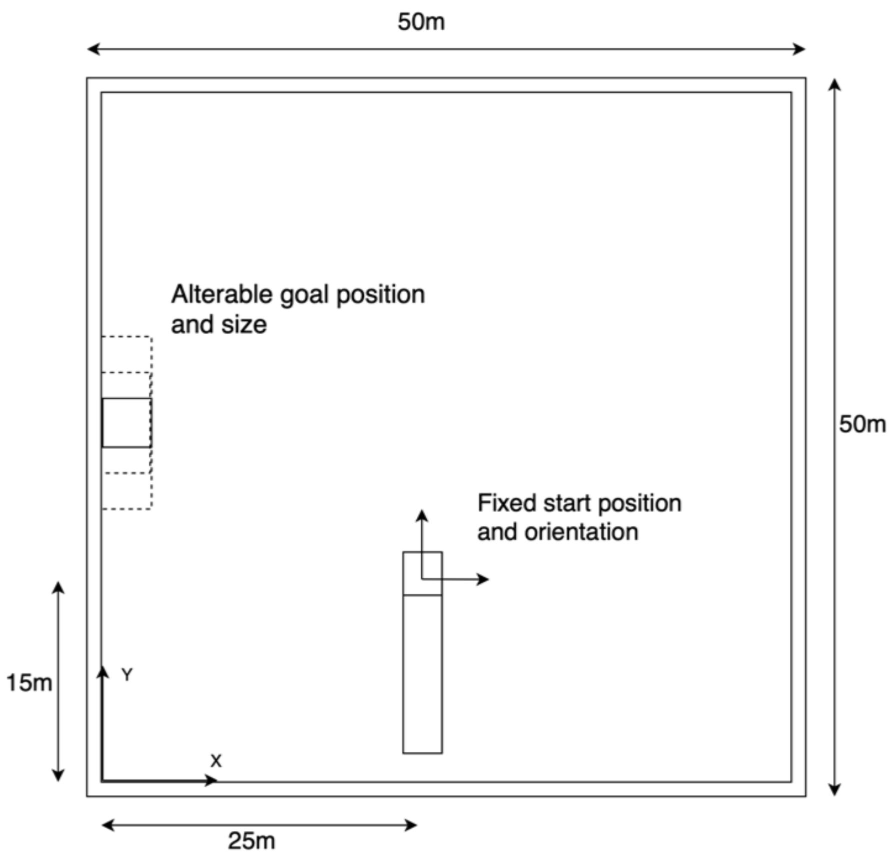

50m
Alterable goal position and size
50m
Fixed start position and orientation
15m
Y
X
25m

Figure 3.5: Drawing of the environment

As long as the simulation environment was adapted to the ML-Agents Python API, there was native support for Unity Core Platform. Adaptation implied specifying onto what object the agent should act, what actions it could take, what observation it could use, what event triggered which reward and other parameters needed for training such as episode length etc. Environmental interactions withing Unity was then bridged through the API to python scripts utilising TensorFlow, an open-source machine learning software library where the neural network was trained. This way one could observe agent behaviour in the Unity environment while simultaneously training the policy using TensorFlow, which made training progress easy to monitor.

Providing this data to TensorFlow, the environment supplied detailed MDPs according to

$$Observation, Action, Reward, ... \rightarrow S_0, A_0, R_1, S_1, A_1, ... \tag{3.4}$$

which is necessary to train a neural network as described in section 2.1.1.

Unity and ML-Agents run at 50Hz update rate in real time. The environment in this thesis is relatively slow and changing input fifty times a second would result in an unnatural and jerky behaviour. To counteract this a decision interval of fifty was introduced, meaning that a new action was only taken every fifty steps. This would translate to an action being taken every second in real time, which also would translate well to a real world application.

3.1.1 Dynamics

As Unity Engine tries to mimic true physical properties, dynamics are applied to rigid bodies by default. According to delimitations of this thesis, accuracy of the semi-trailer combination model had low priority in order to simplify the task. With vehicle model dynamics already in place in Unity, the impact of this was slightly minimised by modifying tuneable parameters, e.g. reducing tire slip and vehicle roll. Also since the truck was moving at slow speed at all time, some dynamics did not have a substantial impact on the semi-trailer combination motion between time steps, e.g. tire slip was naturally small.

Adding vehicle dynamics to the environment would not affect the final outcome of the thesis as a proof of concept, but it would increase the possible state space and stochasticity which would make the network more difficult to train. Regardless, it was kept since a successful result with dynamics applied would show the advantage of using RL over traditional methods.

3.1.2 Observations

Implementing visual observation in ML-Agents was intuitively done by placing a camera in the environment positioned such that it, from a top-down perspective, captured the docking terminal area. The resolution was set to 84x84 pixels to allow enough detail to be captured, but still not contain too much data as the number of trainable parameters in a CNN quickly grows with a larger input, which increases computational load. In addition to that, the image feed was converted from RGB to greyscale, which saved both GPU time and RAM. As long as the objects had different colours and became separable in different shades of grey, the visual input would still provide enough information to not complicate learning. Also, greyscale conversion could rather be seen as a measure to make the policy more robust to slightly different coloured semi-trailers.

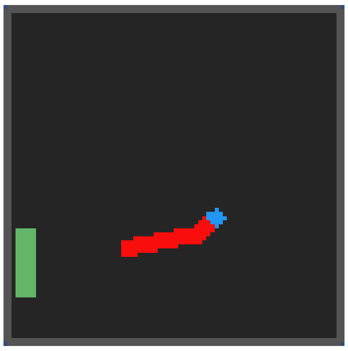


Figure 3.6: Observation in RGB

Table 3.1: Colours converted to greyscale values

Object	Greyscale value (0-255)
Semi-trailer (Red)	125
Goal (Green)	163
Truck (Blue)	146
Walls (Light grey)	84
Plane (Dark grey)	37

In the environment, different colours represented different objects, a green goal, red semi-trailer and blue/red truck as can be seen in figure 3.6. Red and blue colouration of the semi-trailer combination was chosen in order to make front and rear separable. This set of colours made them translated into well distinguishable shades of grey. The basis for learning was to make connections based on the observation, thus different coloured pixels. A possible successful outcome from this setup could have been e.g. learning a dependency between the greyscale versions of red and green, leading to high reward. Separable colourations could be found in a real-world scenario except for the green target marking. For the image input to be useful the goal needed to be coloured, otherwise the neural network would not be able to perceive what visually correlated to a large reward. A green marking, showing where an available docking spot exists could be realised with e.g. a software drawing an overlay on the observation image based on information from a another neural network trained to detect free spots [15] or simply external sensors. Goal identification was considered to be out of scope and the green area was therefore manually added to the observation.

To be able to compare performance of different observation setups, a number of normalised non-visual observations, chosen to represent the environment well, were also introduced as input. Combining the visual input with these in different ways during training experiments gave a basis to analyse training performance.

The non-visual observations were:

- Global position of semi-trailer rear in two dimensions
- Global position of truck in two dimensions
- Local position of semi-trailer rear in two dimensions relative to target
- Global angle of truck in degrees
- Angle between truck and semi-trailer in degrees
- Velocity of truck in two dimensions

Normalisation was done to simplify training, explained in section 2.3.2.3. If the observed value is represented by x, the normalised value x' is calculated according to:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{3.5}$$

3.1.3 Environment scripts

Functionality in Unity and ML-Agents was specified by different scripts that were written in C# and linked to objects in the environment. The main script was attached to the truck itself, on which the agent acted.

ML-Agents built-in key functions simplified scripting and enabled full focus on RL implementation design. The following predefined functions were utilised (as of version 0.13.1 of ML-Agents).

- Done() - Marks the end of a episode, calls the reset function and starts a new episode.
- AddReward() - Adds reward every time the function is called, which is every step.
- SetReward() - Overwrites the accumulated reward for the past decision interval, 50 time steps in this thesis, and sets a reward for the whole interval.

The main script had three principal functions: taking actions, collecting reward and resetting the environment when desired. To establish these functions, transformation of objects, boolean triggers and global variables used between physical objects were utilised. Action taking was implemented as a simple discrete vector action-space, as continuous action greatly increases complexity. The agent was given three outputs for wheel torque (forward, neutral or backward) and three for steering angle (left, straight or right). Both steering and throttle action was sampled in parallel, meaning that the agent could apply them simultaneously.

Implementation of the reward distribution was mainly based on comparing the truck position and rotation with specifications of the reward function at every time step, the exact reward function is in described in a later section. Comparisons were done using both continuous measurements but also booleans that were triggered if the semi-trailer entered a specified area or collided with another object. Specifications of the reward function can be seen in figure 3.8 to 3.10. Resetting the environment involved moving the semi-trailer combination to initial position, resetting velocities, forces, actions and all booleans. Booleans triggered when entering areas or crashing into objects were handled by task-specific scripts that were linked to the objects in question. This enabled realisation of different reward functions with great control.

3.2 Training setup

To succeed with training, focus was mainly on tuning hyperparameters and reward functions. As there is no general rule to succeed with RL, existing implementations and guidelines from ML-Agents were therefore used as an initial starting point.

3.2.1 Choice of algorithm

As part of ML-Agents 0.13.1 there were two different state of the art algorithms implemented to work with their API. These were Soft Actor-Critic (SAC) and PPO, both policy gradient methods that used ADAM optimiser for fast and stable parameter updates, see section 2.1.6.

Whereas SAC has shown to outperform [22] the older algorithm PPO in multiple tasks, it is also known to not always be consistent in performance. To increase the chances of stable learning, instead of learning as fast, or sample efficient, as possible, PPO was chosen. See section 2.2.2.4 for explanation of PPO.

3.2.1.1 Tuning hyperparameters

In the ML-Agents 0.13.1 documentation there were recommendations on what order of magnitude parameters should be set depending on chosen algorithm and if the state and action space was continuous or discrete [25]. There were also some hints on how one could tune them depending on certain outcomes of training, especially when looking at learning and loss graphs. However, since the provided parameters did not account for visual observation this had to be taken into account and tuning had to be performed to get results from the RL algorithm.

As multiple parameters can affect the same type of behaviours during training, it was crucial to only tune one at a time as it was otherwise hard to draw conclusions. The course of action was to start as safe as possible when changing parameters. This meant starting with standard values and move towards what was suppose to result in slow and stable learning. E.g. large buffer and batch size as described section 2.1.12. Starting out too aggressive made it hard to understand what the source of failure was.

Recall algorithm 1, REINFORCE, row 5:

$$\theta \leftarrow \theta + \alpha \gamma^t G_t \nabla_\theta \ln \pi_\theta (A_t | S_t) \tag{3.6}$$

Each policy update is scaled by a learning rate factor α that when reduced makes learning slow and steady. Learning rate is used in most algorithms including PPO. It was therefore a good starting point to first decrease the learning rate and then slowly increasing it throughout the tuning process.

Another parameter which affected training stability was the time horison. It determined from which step in each episode one should bootstrap the value estimate, see theory section 2.1.10. Bootstrapping from an early state lowers variance and increases stability, but having it set too short introduces a strong bias making it harder for the agent to see the goal, which can lead to local convergence. Therefore it had to be increased to cover a complete successful episode. When performing an experiment and manually providing actions to reach the goal, an episode turned out to be approximately 1600 steps. With a decision interval of 50 it resulted in 32 experiences. While slowly increasing the time horison to possibly increase detection of important information in successful trajectories, the value 32 acted as a guideline of what could be a working parameter.

Value estimation with a time horison set to 32 would correspond to values being updated as:

$$\mathrm{V}'(\mathrm{S_t}) \leftarrow \mathrm{V}(\mathrm{S_t}) + \alpha \left(\sum_{k=0}^{32} \gamma^k R_{t+k+1} + \gamma^{33} \mathrm{V}(\mathrm{S}_{33}) - \mathrm{V}(\mathrm{S_t}) \right) \tag{3.7}$$

With the main feature of PPO being to clip objective gradients to ensure stable updates, this parameter was therefore essential to tune in order to arrive at a functional algorithm. Recall the clip objective of PPO

$$J^{\mathrm{CLIP}}(\theta) = \mathbb{E}\left[\min\left(r(\theta)A(s,a), \mathrm{clip}(r(\theta), 1-\varepsilon, 1+\varepsilon)A(s,a)\right)\right] \tag{3.8}$$

where ε clips the update if the new and old policy differ too much. Epsilon was therefore drastically decreased and from there slowly increased until training instability was observed. One could then find out what the limit of ε was.

3.2.2 Training a baseline agent

The first step was to get a simple environment working by using close to default parameters provided by ML-Agents. This meant having solely a truck driving forward to a static goal. Once the neural network successfully learned to provide state feedback which took the truck from start to goal, the next step was to try to make it reverse into the goal. This initially turned out to be a lot more difficult as rewards had to be carefully chosen for it to understand when it needed to switch from driving forward. With a more complex manoeuvre, additional techniques were introduced to help the agent comprehend what it had to do. Using a curriculum, theory section 2.1.13, the agent could get an easy task such as a wider goal and learn that first until meeting performance criterion. After this succeeded one could make the task more difficult by decreasing the size of the goal as visualised in figure 3.7 and then continue training.

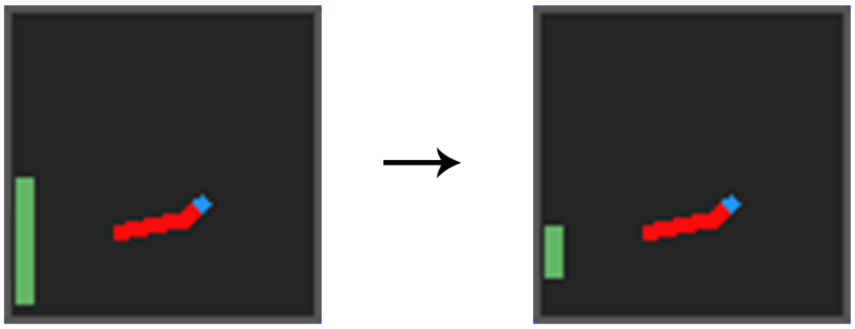


Figure 3.7: Training using a curriculum

Using behaviour cloning also helped give the agent hints of what type of actions it should take in a certain states to get a large reward, theory section 2.1.14. Starting out with imitation learning and then letting the agent continue on its own after e.g. 15 million steps made it discover reward easier and learn faster. This way one could also make the rewards more sparse avoiding unpredictable local minima, since it will through these techniques still be directed towards goal.

Adding a semi-trailer complicated training even more since it introduced a new unwanted behaviour that needed to be taken into account. Avoiding a so called jackknife position was crucial and done by modifying the reward function.

Lastly, in order to assure that the policy was learning a meaningful behaviour and not a sequence of actions, the goal position was moved between every episode by sampling from the 33 discrete positions spanning the left side of the docking terminal. It made the task more complex and therefore more training time was required to reach previously obtained levels of accumulated episode reward.

To shorten training time while having more complex tasks, a ML-Agents tool called "Asynchronous Environments" was implemented. It made it possible to run multiple training environment instances asynchronously, decreasing training time and increasing sample efficiency. The functionality was based on making the neural network capable of performing an action in one instance and continue with another while the first take its step, to not have to wait for the environment to return observations needed for the next action.

Training the baseline agent resulted in the following parameters:

Table 3.2: Hyperparameters for baseline agent.

Buffer size	51200 experiences
Batch size	256 experiences
Time horison	32 experiences
Behavioural cloning	10^6 experiences
Epsilon	0.1
Beta	5^{-3}
Lambda	0.95
Curiosity strength	0.02
Learning rate	1^{-4}
Hidden units	256
Num hidden layers	2
CNN layers	2

3.2.2.1 Reward function - Complexity scaling analysis

Starting out using a complex reward function in a continuous state space can lead to local minima convergence with unwanted behaviours. Therefore the reward started out simple and was gradually made more complex to get the wanted behaviour. Initially, rewards were set according to what is a succeeded task and what is a failed one. That meant giving positive reward once the goal was reached and negative if the truck crashed as can be seen in figure 3.8. The reward given when reaching the goal was constrained to also having an angle of 90 ± 10 deg.

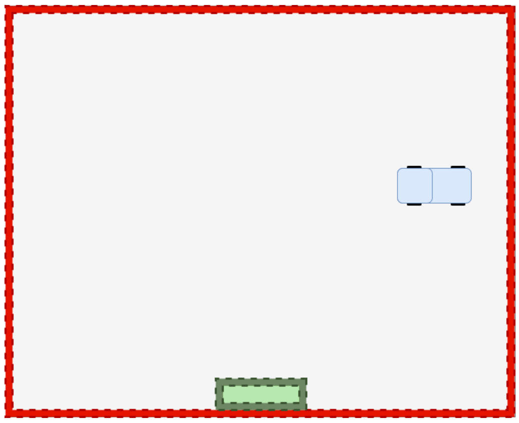


Figure 3.8: Sparse reward function. Red dashed line gives -0.01 and green dashed line $+1$

As sparse rewards require the agent to unintentionally hit the goal for it to learn and then "see" that reward in earlier states through the discount factor, it could easily lead to an agent learning nothing. The reward function was therefore enlarged with a reward which increased linearly with decreasing distance to the goal as in figure

3.9. That way the agent would more often see how moving towards the goal was a good behaviour.

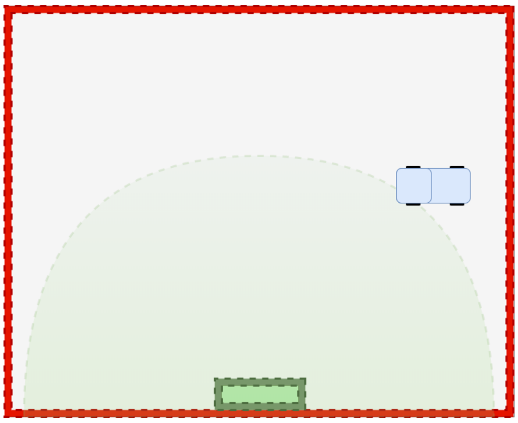


Figure 3.9: Red dashed line gives -0.01 in reward, green dashed line $+1$ and dashed light green half circle $\frac{1}{\text{distance}}$.

Training RL problems are known to be unstable and with both continuous state space and visual based input the training becomes even more difficult. It was therefore easy for the agent to end up in local minimum during reward improvement iterations. To avoid that, the environment was run manually after each change to check for unwanted possible exploitation.

Once the reward function was designed in a way that made it natural for the agent to drive towards the goal, other extensions of the reward function had to be formed to encourage reversing towards the goal. A natural condition for reversing is to have a certain angle of the truck. In global coordinates it meant that an angle of 90 ± 45 degrees towards the goal needed to be met, as shown in figure 3.10.

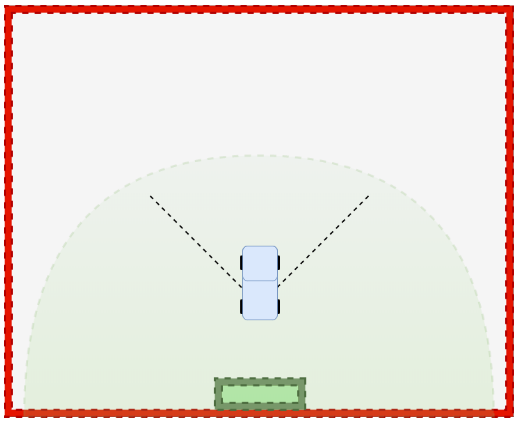


Figure 3.10: Red dashed line gives -0.01 in reward, green dashed line $+1$ and dashed light green half circle $\frac{1}{\text{distance}}$ if angle of truck is between the black dashed lines.

Adding an extra condition in form of an angle constraint made the reward function more sparse, making the agent see positive reward less frequent thereby needing more samples. To avoid having the agent exploit the function by alternating between going backwards and forwards piling up on rewards, a negative addition at each time step was introduced. Negative reward for each time step also gave a incentive to complete the task faster.

With the semi-trailer attached, reward functions were mostly kept although some values needed to be re-tuned again. Changing the angle and position to be measured at the semi-trailer instead of the truck meant that the same conditions were applicable for it to reverse into goal. Avoiding jackknife was done by measuring the angle between truck and semi-trailer and adding a negative reward every time the angle got above 85 degrees, penalising such states regardless of position as in figure 3.11. Setting this reward too large could confuse the agent as it would learn to not turn sharp at all, making it hard to maneuver the semi-trailer combination. It was therefore iterated until a suitable value was found.

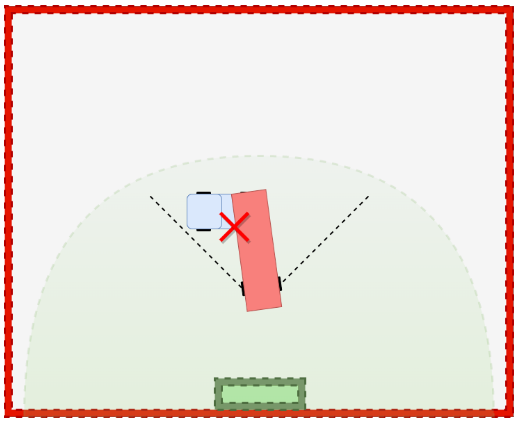


Figure 3.11: If the angle between truck and semi-trailer was above 85 degrees, a negative reward of -0.1 was given.

The final reward function is specified in pseudo code, algorithm 2.

Algorithm 2 Reward function - Complexity experiments

```
SetReward(-0.0005f);
if Closer to target & $Angle = 90 \pm 45\,\text{deg}$ then
    SetReward(1f / distanceToTarget);
if Goal Reached & $Angle = 90 \pm 10\,\text{deg}$ then
    SetReward(1.0f);
if Collision then
    SetReward(-0.01f);
if AngleTruckTrailer $> 85$ then
    SetReward(-0.1f);
```

3.2.2.2 Reward function - Generalisation analysis

For the generalisation training, a couple of modifications were done to the reward function. In order to be able to clearly state if the goal was reached and tell difference between variations in the environment, the reward function was simplified by removing the linearly increasing distance reward. It was a crucial part of the reward function and as stated in section 3.2.2.1 a sparse reward function can make it difficult to learn, but with additional imitation learning and increased training time the agent managed to learn. Other rewards were kept unchanged, resulting in the following reward function:

Algorithm 3 Reward function - Generalisation experiments

```
SetReward(-0.0005f);
if Goal Reached & $Angle = 90 \pm 10\,\mathrm{deg}$ then
    SetReward(1.0f);
if Collision then
    SetReward(-0.01f);
if AngleTruckTrailer $> 85$ then
    SetReward(-0.1f);
```

3.3 Evaluation tools

Visual inspection provided a good measurement on current performance during training as well as performance of the final policy. A simple visual snapshot during training did however not tell what the trend was, e.g. distinguish if the agent was still learning or if it had converged.

Therefore, a number of Key Performance Indicator (KPI) were identified to give structure to the analysis of complexity scaling and generalisation. Monitoring the progress of training and analysing results was thereby mainly done with the following parameters that were plotted:

- In order to get an indication of learning, the mean **cumulative reward** received during episodes spanning over 1000 experiences was plotted. An over time steady positive gradient indicated ongoing learning while a negative trend meant forgetting. Observing an unchanged value meant convergence. If the reward variance was significant, the probability of converging to an acceptable solution was fairly low and the experiment could be aborted.

- The maximum episode length was a fixed value set in the environment. As an episode was also reset if the agent reached the goal, the logged **episode length** was affected. Plotting this helped to identify when the goal was reached for the first time and it also gave information of how frequent it managed to do it.

- In order to tell how certain the agent was on its decisions the **policy entropy** was plotted. It viewed uncertainty in the discrete action probability distribution for all states. Preferably it should have a high value initially and decrease slowly as the agent learns the correct behaviour and the need for randomness vanishes.

- During training the agent estimates state values, according to section 2.1.7. In order to tell if the agent had explored rewards the **value estimate** was plotted. It displayed the mean value of all estimated states and increased as the agent discovered new reward. If negative reward was given for poor states it could also decrease.

- In order to more precisely tell if the agent was still learning, the **policy loss** was plotted as it described to what extent the policy was changing. The need for policy changes decreased when the agent captured a correct behaviour. Therefore the magnitude of policy loss should decrease during successful training.

- As mentioned, states were estimated with a value. Mean loss of the value function update, **value loss**, could be utilised to further monitor learning progress. It gave an understanding of how well the value of each state was predicted i.e. how much it was changing. An increasing value loss was expected during training as reward increased. Once the accumulated reward stabilised it lead to a decrease in value loss.

3.3.1 Convolutional neural network evaluation script

To analyse how well the visual encoder managed to extract important features from the observation one could either look at learned filter weights, as described in theory section 2.1.3, or a feature activation map. This map was constructed by feeding an observation image to the trained CNN and taking the output from a chosen layer with a chosen filter being applied. As filters in each layer activate on different features, the output was a heat map showing what features were activated and where, providing an important basis for analysis. Example of a feature activation map can be seen in figure 3.12.

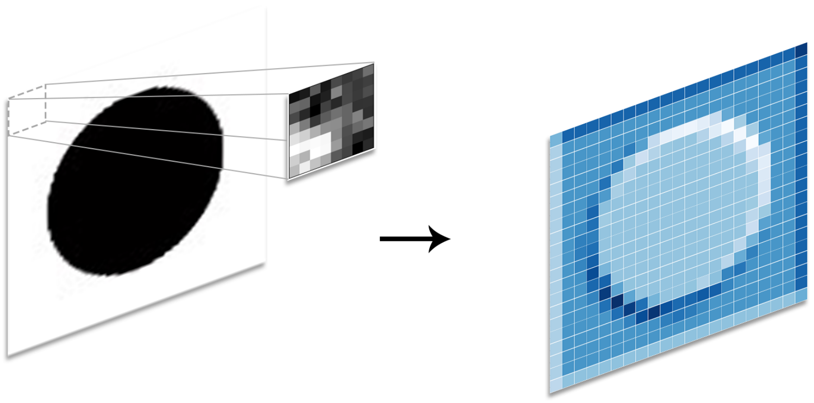


Figure 3.12: Example filter weights applied to a image, resulting in a feature activation map

3.4 Experiments

Many iterations, a lot of computations and many hours spent on research was needed to get a feeling for what settings made learning better. The main research questions were treated according to experiment plans to get as relevant results as possible.

3.4.1 Complexity scaling - Observation

Even though one can never be sure of certain results in RL, non-visual observations are known to be more stable during training and usually more prone to converge than using visual input. To get data from where one could draw conclusions, a list of scenarios to test was created, see section 3.1.2 for details on observations. It started with a scenario that was thought to yield the best result and then it was made more difficult by removing non-visual observations. A scenario with only non-visual observations was also used as a reference.

Not supplying a perception of velocity would make observations insufficient leading to the theory of MDP, section 2.1.1, not being applicable since present observations will not be enough to describe the state independent of the past. This experiment was therefore not incorporated.

Table 3.3: Complexity scaling observation, experiment cases

Case	Visual	Velocity	Trailer rear pos	Truck pos	Rel Trailer rear pos	Angle truck	Angle truck & trailer
1	x	x	x	x	x	x	x
2		x	x	x	x	x	x
3	x	x		x	x	x	x
4	x	x			x	x	x
5	x	x				x	x
6	x	x					x
7	x	x					

With the goal of comparing learning performance, stability and convergence, all experiments specified in table 3.3 were run until approximate convergence as time was a limiting factor. Even though more time could yield a result where the agent learned a better behaviour, one could after approximately 100 million steps get enough result of training to compare the experiments. All reward graphs were at that point either starting to level out or already flat as shown in figure 4.1.

3.4.2 Complexity scaling - Task

To analyse training differences when changing task in the environment, three cases were used as specified in table 3.4. All parameters were kept unchanged, except for how many steps behavioural cloning was used, for a fair comparison without other influencing factors. Using behavioural cloning for too many steps could make performance worse as it induced overfitting and thereby distorted results. Therefore, the easier tasks used it for a smaller number of steps. As input visual observation and velocity in two dimensions were given.

Table 3.4: Complexity scaling tasks, experiment cases

Case	Task
1	Truck and semi-trailer docking reverse
2	Truck docking reverse
3	Truck docking forward

All experiments were trained until guaranteed convergence, visually validated by observing unchanged reward for five million steps and all KPIs from section 3.3 describing stability and convergence e.g. losses, entropy and variance were examined.

3.4.3 Generalisation

Regardless of what environment RL is applied on, either the reward function or algorithm is likely to differ between implementations. Same was true for this thesis and made it questionable to what degree potential generalisation discoveries was going to be applicable on RL in general. Therefore, experiments were focused on measurements that were not directly dependent on the algorithm or reward function.

As described in section 2.3.2.5, varying parameters in the environment can have great impact on generalisation capabilities and it is also possible to implement on any RL setup. To investigate how well a policy network can generalise using visual input, analysis on how variations in the environment affected the generalisation was performed.

In the semi-trailer docking case, a valuable generalisation property would be for the policy to handle different semi-trailer lengths. Changing length also implied a new distance between the semi-trailer wheel axle and link point. It created entirely new kinematics that the policy had to generalise to. If small changes are performed in a simulation environment a policy can normally interpolate between solutions to some extent and manage to solve scenarios that has not been presented during training, as illustrated in figure 3.13

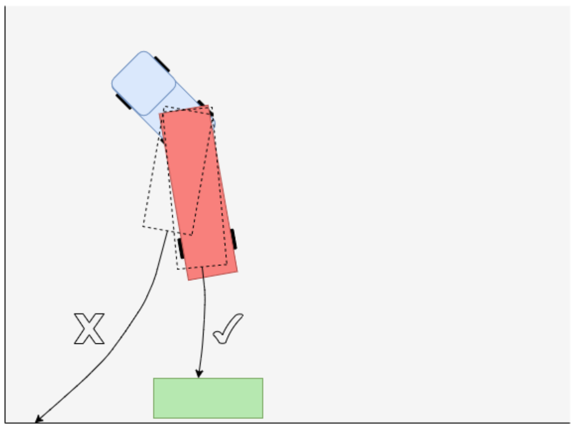


Figure 3.13: Visualisation of generalisation on different semi-trailer lengths

Functionality was added to randomly change the semi-trailer length each episode during training. The length was sampled from a predefined discrete range, table 3.5.

Table 3.5: Generalisation experiment cases

Case	**10m**	**11m**	**12m**	**13m**
1				x
2			x	x
3		x	x	x
4	x	x	x	x

The experiments were carried out by first training using one fixed semi-trailer length, followed by adding more lengths to sample from according to table 3.5. To make the four policies comparable, a reference maximum reward was elaborated. During development of experiment methods, training of case one in the generalisation environment had an agent reaching a maximum accumulated discounted reward of 4.2 after convergence. This value was therefore chosen as a baseline. A Tolerance and slight reduction of the maximum reward was introduced and used as a reference for training completion. This meant that training of each policy was interrupted when it reached a accumulated discounted reward of 4.1 ± 0.1 for five million steps. Also convergence was authenticated by visually observing a number of plotted parameters, decreasing entropy close to zero, constant value estimate, decrease in policy loss magnitude and value loss approaching zero. The interpretation of each parameter is explained in section 3.3.

The four policies were then evaluated on all possible lengths in a range between 11.5-13 meters with a resolution of 0.1 meter. Each semi-trailer length setup ran one million steps and the mean reward over all steps was calculated and used for evaluation. All as described in figure 3.14.

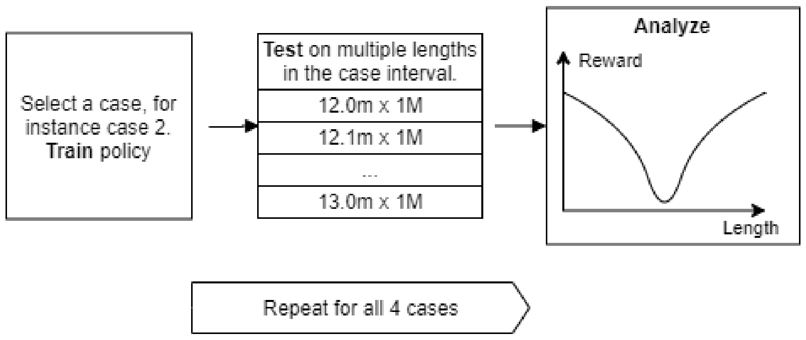

Select a case, for instance case 2. **Train policy**
Test on multiple lengths in the case interval.
12.0m x 1M
12.1m x 1M
...
13.0m x 1M
Analyze
Reward
Length
Repeat for all 4 cases

Figure 3.14: Generalisation experiment, flow chart

To put the reward into context, the success rate was also measured in the neighbourhood of where the best and worst performance was expected. Success rate gave an exact measurement of how frequently the agent completed the task but it did not tell how well it did it. It could have crashed multiple times on its way to the goal or possibly ended up in a jackknife situation. However, that was something the reward revealed. Reward was therefore chosen as the main basis for analysis with the success rate as a complement to give an understanding of what certain reward levels correspond to in regards to completing the task.

4

Results

4.1 Observation complexity comparison

Experiments were performed according to table 3.3 with all parameters set to fixed values for a fair comparison. All cases were run approximately 100-130 million steps until, or close to, convergence. Different observation setups learnt differently as can be seen in figure 4.1. Some were more unstable, but learned faster, while some were smoother and slower. A reward curve displays better behaviour if a large value is gained faster, while still remaining stable.

The observation complexity experiments showed that case 4, 5, 6 and 7 overall were learning faster and more stable compared to the other cases, see figure 4.1. Characteristics for these cases were that they had fewer non-visual observations available. Between case 3 and 4 the truck position was removed, this was therefore the single observation that had the largest impact on the experiment. Case 4-7 also showed rapid increase in value loss, see figure 4.4, further justifying fast learning. Explicit results follow below.

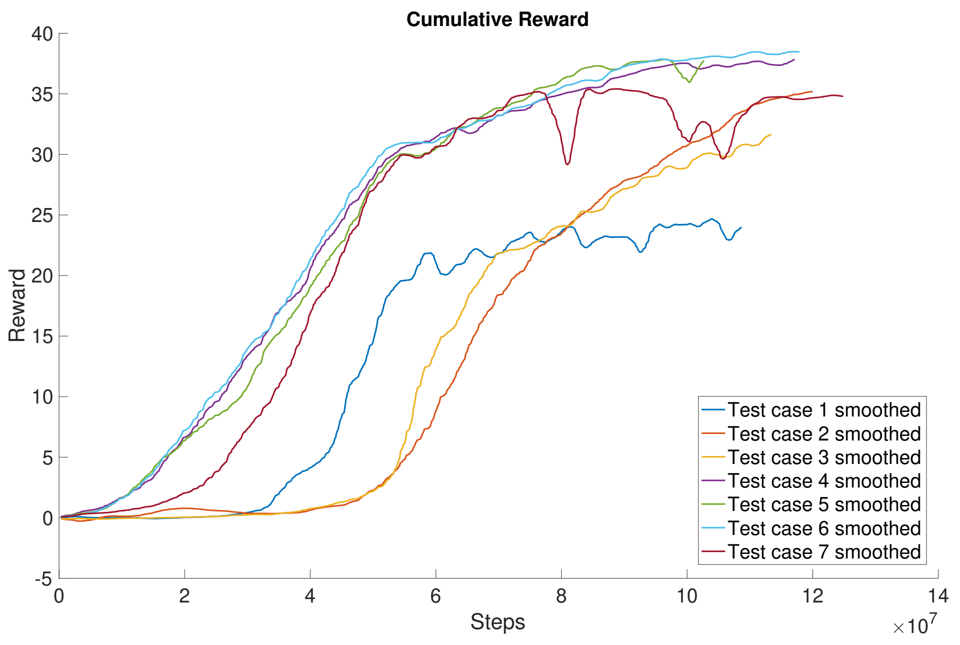


Figure 4.1: Observation complexity experiment, case 1-7 - Smoothed average cumulative reward

The curves in figure 4.2 show variance corresponding to fluctuations in reward, visible in figure 4.1. A lower variance indicates a more stable training. It can be large initially, but should decrease as training starts to converge. Large peaks show momentarily forgetting.

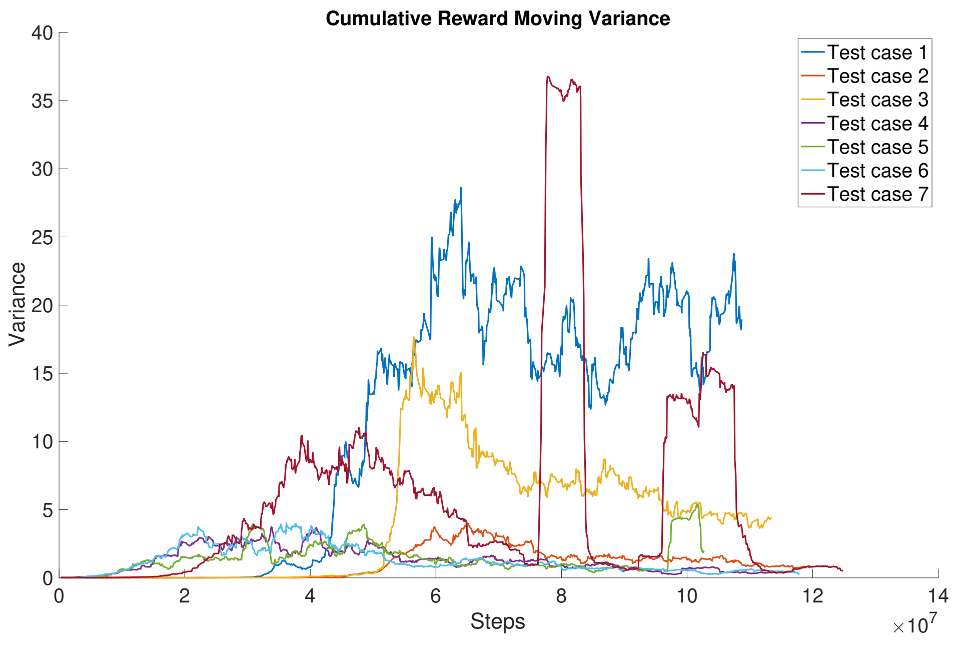


Figure 4.2: Observation complexity experiment, case 1-7 - Variance of cumulative reward

Policy entropy shows how random actions are. As an agent learns, entropy should decrease until converging close to zero. It may fluctuate during initial phases of learning while exploring randomly.

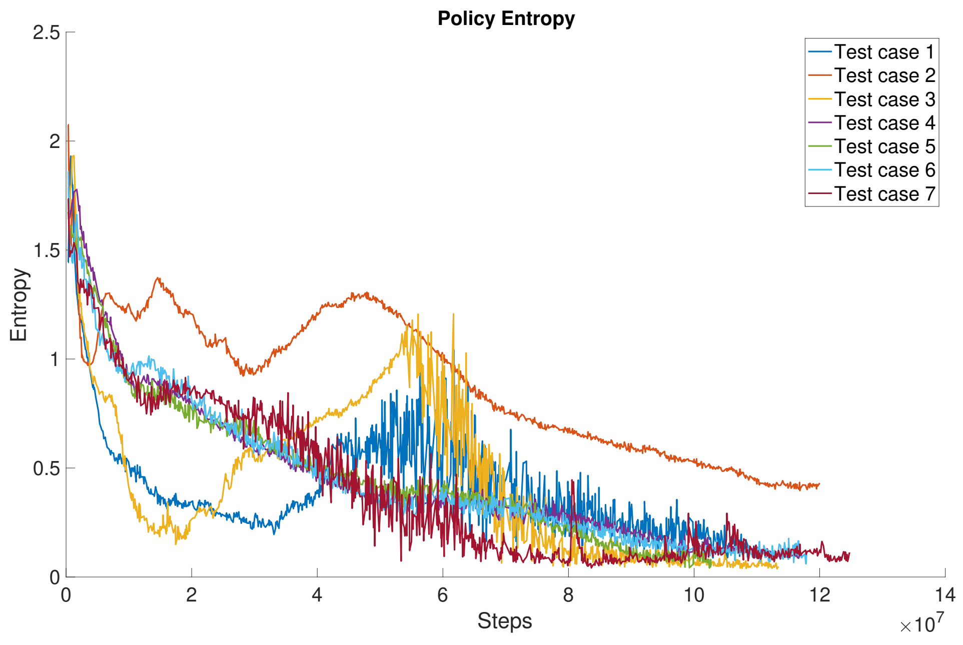


Figure 4.3: Observation complexity experiment, case 1-7 - Policy entropy

The value loss shown in figure 4.4 displays how well the network is able to predict values of each state. During training these are learnt and the loss should therefore increase. When converging, estimation will not differ that much and the loss should decrease. Instability leads to a high loss which will not decrease.

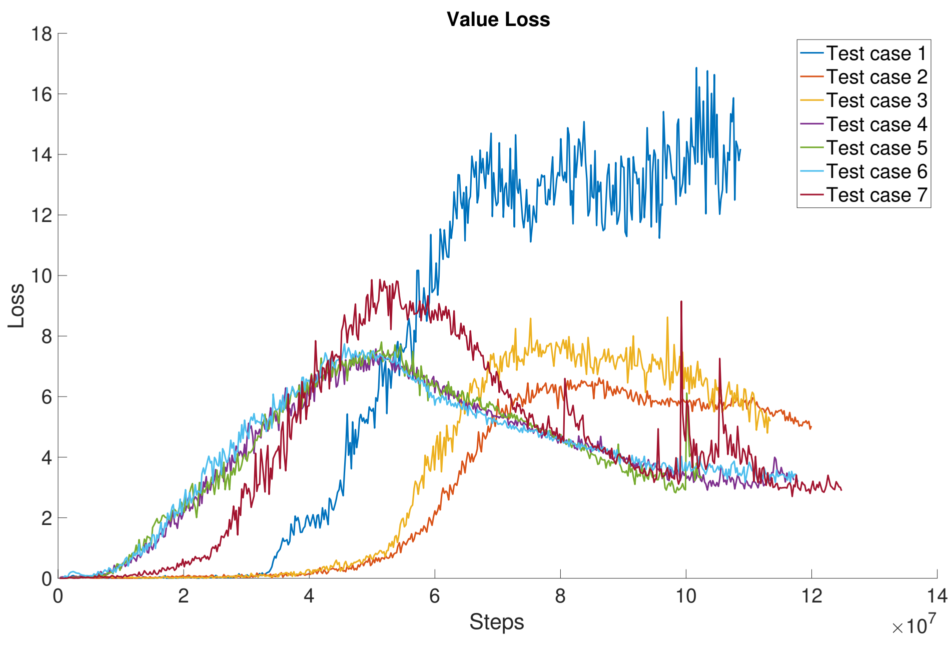


Figure 4.4: Observation complexity experiment, case 1-7 - Value loss

Figure 4.5 and 4.6 show the difference between convolutional filter weights with all measured non-visual observations (case 1) and only velocity (case 7). Gradients are more distinct in case 7 where only one non-visual observation is used.

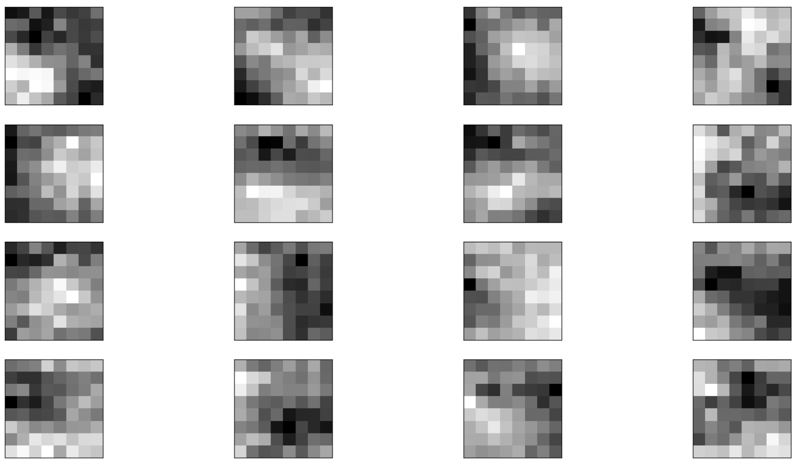


Figure 4.5: Observation complexity experiment, case 1 (visual & all measured) - Convolutional weights, all filters first layer

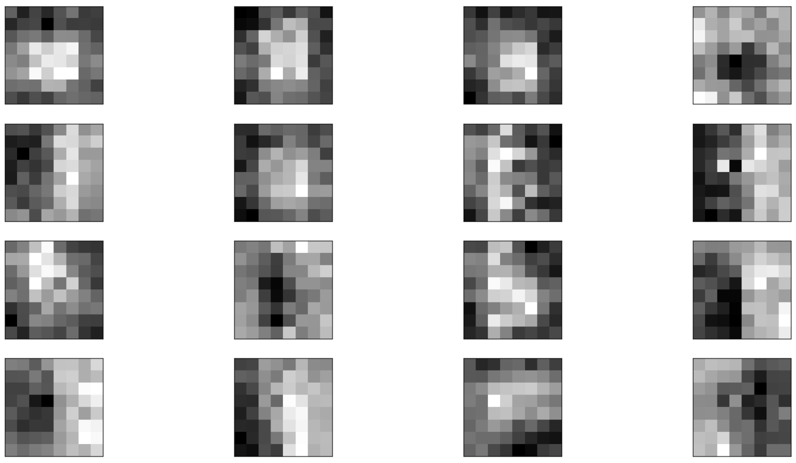


Figure 4.6: Observation complexity experiment, case 7 (visual & velocity) - Convolutional weights, all filters first layer

Figure 4.7 and 4.8 show the difference between convolutional feature activation maps again for case 1 and 7. Using fewer non-visual observations result in activation maps that overall distinguish objects more clearly.

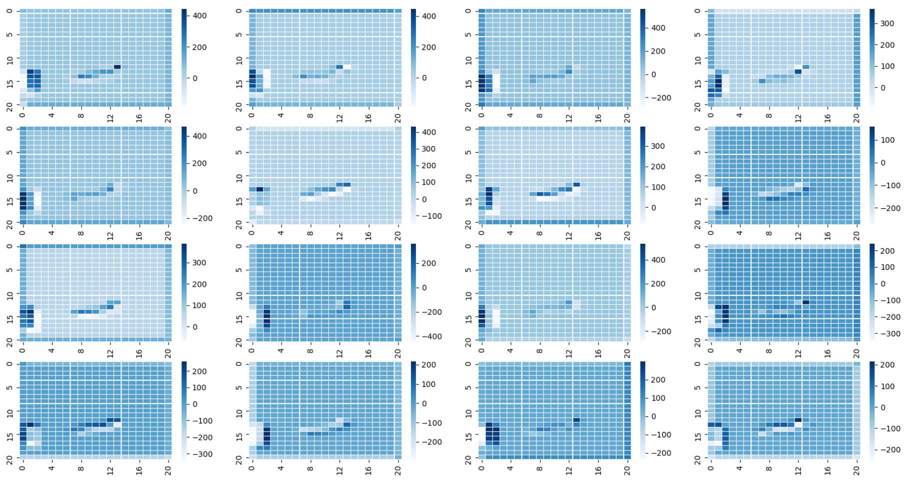


Figure 4.7: Observation complexity experiment, case 1 (visual & all measured) - Activation map, all filters first layer

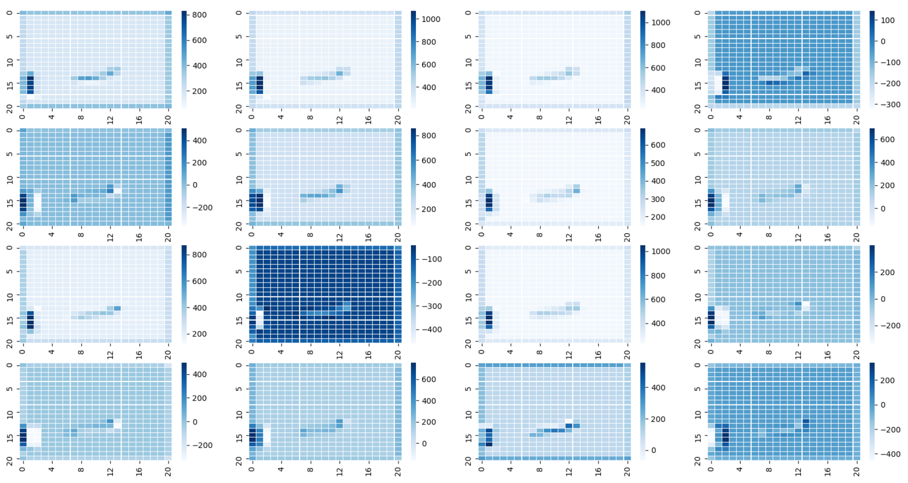


Figure 4.8: Observation complexity experiment, case 7 (visual & velocity) - Activation map, all filters first layer

4.2 Task complexity scaling

Testing complexity scaling with changing task setups was performed according to table 3.4. Differences in final peak reward can be disregarded as it is expected for a simpler model to reach the goal faster and thereby accumulate a larger reward. A steeper rise in average cumulative reward and a quicker dropping variance, without momentary peaks, indicates better training performance.

However figure 4.9 indicates that a more complex task (case 1), both results in decreased stability and learning speed.

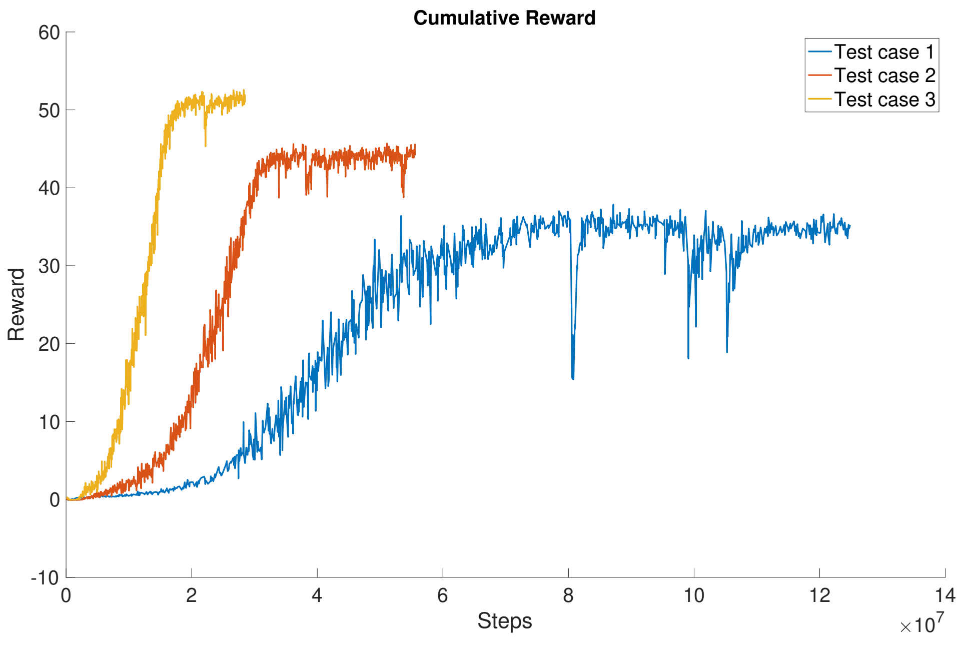


Figure 4.9: Task complexity experiment, case 1-3 - Average cumulative reward

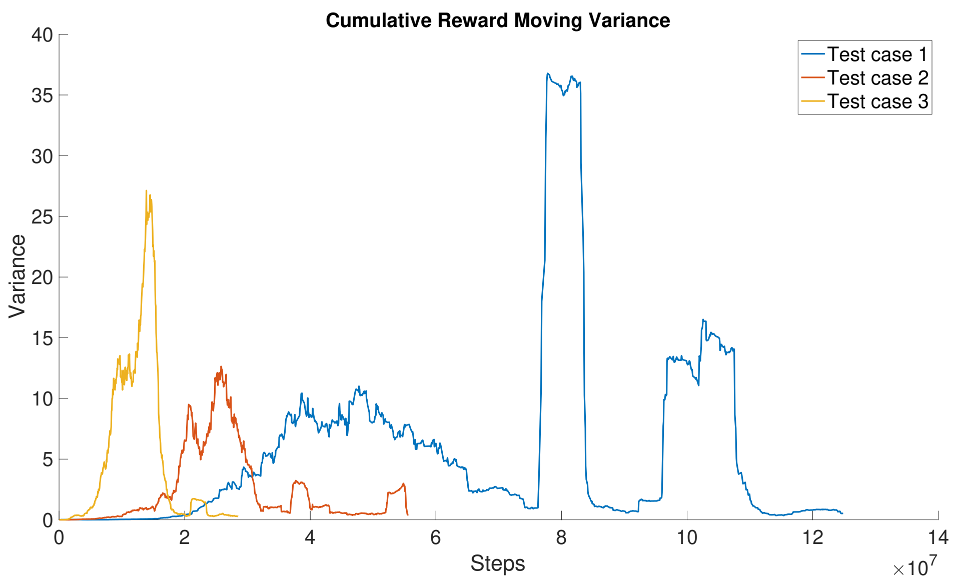


Figure 4.10: Task complexity experiment, case 1-3 - Variance of cumulative reward

4.3 Generalisation results

Results from generalisation experiments, described in section 3.4.3, table 3.5, consists of four data sets, one for each policy. To make the data transparent, mean of the one million evaluation steps on each semi-trailer length was calculated. It resulted in curves with 16 data points, spanning 11.5 to 13 meter, each describing how well the policy in question generalises to different semi-trailer lengths.

Figure 4.11 displays all four curves making it easy to tell which policy generated highest reward on intermediate semi-trailer lengths, that was not explicitly trained on. High reward on intermediate lengths indicates good generalisation capability. Experiment case three, trained on semi-trailer lengths 11, 12 and 13 meter and case four, trained on 10, 11, 12 and 13 meter, generated higher reward on intermediate semi-trailer lengths compared to the other two policies and most likely generalize better.

The success rate for the policy generated by experiment 4, tested on 12, 12.1, 12.2 and 12.5 meter semi-trailer are also marked in figure 4.11, the relationship to reward is not linear but it gives an indication of what different levels of reward means in terms of completing the task. Policy four, tested with a 12.5 meter semi-trailer has almost 60% success rate but the reward has dropped significantly, indicating that the docking has been completed in a non-satisfactory way, e.g. crashing or taking too long time.

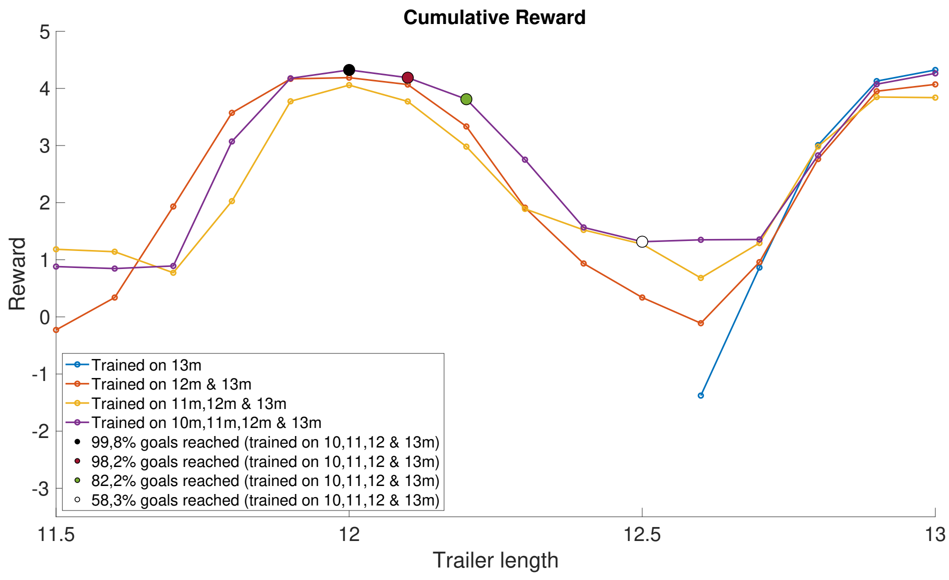


Figure 4.11: Generalisation experiment, case 1-4 - Average cumulative reward of policies evaluated on different lengths for 1M steps

4.4 Best achieved performance

The following list characterises achieved performance of the most versatile fully trained policy network:

- Successfully provide nonlinear state feedback to a dynamic vehicle model in an environment with a continuous state space.
- Converge to a solution using only visual input and velocity of truck in 2D.
- Can manage semi-trailer lengths $10m$, $11m$, $12m$ & $13m$ with some generalisation in between according to figure 4.11.
- Able to dock at 33 different terminals with an angle of $\pm 6°$ and a success rate of 99.8%.
- Can dock between parked trailers being separated by $9.5m$ at a terminal $3m$ wide with a semi-trailer being $2.5m$ wide, as visualised in figure 4.12.

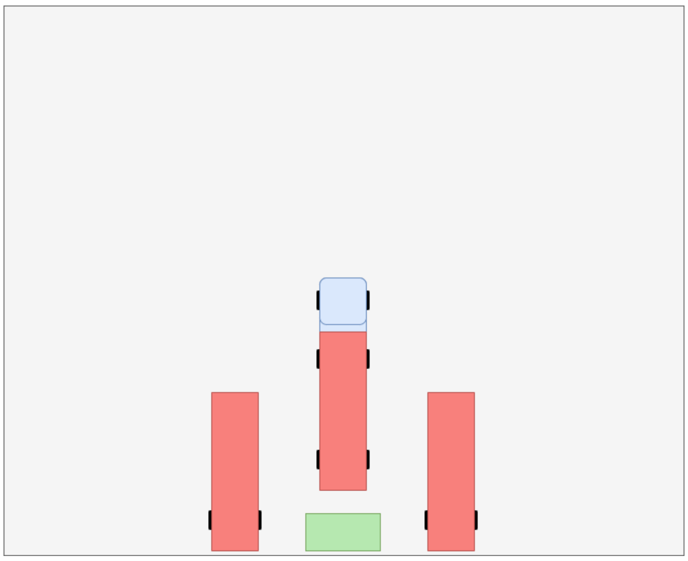


Figure 4.12: Semi-trailer docking between parked trailers

5
Discussion

5.1 Observation complexity scaling

Training a neural network using visual input is difficult and in some cases not even possible if the data presented does not provide details from which the network can interpret useful information. With the concept idea of this thesis being a top-down view of a docking terminal using a stationary camera, the problem becomes hard to solve as actions lead to less pronounced changes in the environment observation.

As a non preprocessed greyscale image is only seen as a matrix of values by the policy, it can not make use of the information in the same way as a human can. Convolutional layers is then a great advantage and a necessity to get any sort of meaning out of the image. Regardless, it is still a lot less straight forward than e.g. getting the measured position as input.

Looking at figure 4.5 and 4.6 one can see what trained weights of the first convolutional layers in the two extreme cases look like. With figure 4.6 showing the case with only pixel and velocity input, one can see how sharper features with more contrast are formed. Since filter weights are initialised randomly, this shows how providing less measured input forms the network into learning how to make better use of visual input.

The feature activation maps found in figure 4.7 and 4.8 follows the same pattern as the weights, with case 7 showing how the CNN more clearly has learned to distinguish the goal and walls, which can be seen by the greater contrast in the heat map. This is of great importance to succeed with the task and shows how the CNN extracts useful information from the visual input.

To evaluate the training process and understand which measured inputs that could be useful in combination with pixel data, when providing control for a truck and semi-trailer, figure 4.1 to 4.4 shows learning performance progress during the seven experiment cases. In figure 4.1 one can see how removing measurements initially improves learning performance. With different measured positions being removed in the first experiments, it is clear that these observations does not cooperate well with the visuals. At case 4 only the semi-trailer rear relative coordinates are used as a positional input. This gives a lot better results, but one can also see how removing this and only keeping angle and velocity measurements retains the same performance.

It is not until removing measured angle between truck and semi-trailer, and only using velocity as measurement as in case 7, that the training performance slightly drops again. This can possibly be solved by increasing resolution of the visual observation to provide more details, but there are then other training difficulties that arise. Training for a longer time is also possible, which allow performance to get similar to that of case 6.

Training with visual observations is known to be unstable, as can also be seen in figure 4.2 by looking at the reward variance for most experiment cases. This variance should be small when learning is stable. Compared to when solely using measured observations as in case 2 one can see that only case 5 and 6 using visual, velocity and angles as input, retain similar stability while performing better. This shows how critical it is to pair the visual input with some sort of velocity perception and strictly selected measurements if high detail observation is needed.

5.2 Task complexity scaling

The neural network is trained to provide nonlinear state feedback on a task in a specified environment. When the complexity of it increases it will be harder for the network to learn connections and find combination of neurons to perform the task. The example in this thesis is somewhat similar to what a human would experience. Just because one can easily back a car does not mean that it is as easy to do the same with a semi-trailer as the kinematics is completely different and also has more DOF, which makes it harder to control. In a simplified kinematic model comparison one can see that a truck has three DOF, where the whole semi-trailer combination has four, as shown in figure 5.1.

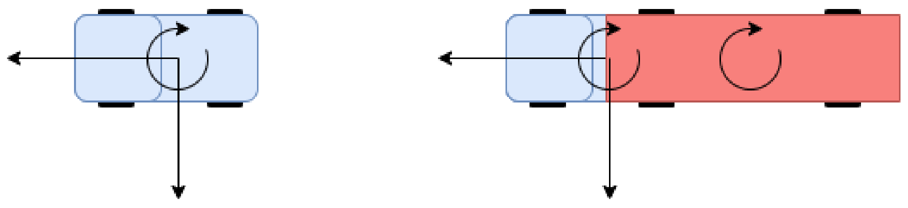


Figure 5.1: Degrees of freedom - simple kinematic model comparison

Adding properties leading to a dynamic vehicle model even if they are small, as was done with Unity Engine, drastically increases DOF as e.g. inertia on rigid bodies are added. This does not only make both models harder to control, but it also increases the gap in complexity between adding a semi-trailer and only controlling a truck.

As can be seen in plot 4.9 the network is able to learn how to dock the front of a truck in approximately 20 million steps. When adding the constraint of having to dock with the rear, the agent instead needs around 35 million steps to converge. This increases complexity of the task as it now needs to turn around somewhere. With rewards being given only once the vehicle moves toward the goal, orientated with the rear facing the goal, the agent will not see rewards as frequently since it starts out forward, facing the goal.

There is no noticeable difficulty difference between driving forward and reversing for the agent, with only a truck to control. One should therefore be able to learn this task at a similar amount of steps with a highly detailed reward function. As discussed in section 3.2.2.1 this can induce unwanted behaviours though and may not be a feasible solution.

By also adding a semi-trailer to the truck, steps needed for convergence increases to approximately 85 million. As the agent with this setup also needs to turn around somewhere, the truck trajectory is somewhat similar to when not having a semi-trailer. The difficulties in training are therefore attributed to the changes in kinematics, i.e. number of DOF in the vehicle model. One can also, in figure 4.10, see how instability increases with increased task complexity, as the reward variance settles slower. It is also visible how the agent undergoes the same phases of learning, as moving variance patterns are similar only stretched out over a different number of steps.

To measure pure training performance with different setups, all parameters were kept the same. However in the experiments involving a more simple task, one could probably e.g. decrease size of the neural network, making it converge even faster. In such a case the comparison would have been between optimally trained networks, which was not the idea. This would still have shown similar results though with possibly even larger differences.

5.3 Generalisation

True generalisation means that the policy can handle unseen states, i.e. new scenarios. Due to margins in what is seen as an acceptable solution, the network can to some extent often generalise to scenarios that are close to the ones presented during training. All experiments related to generalisation in this thesis are based on varying the semi-trailer length according to a number of discrete lengths and in that way create new scenarios. If the policy would have been trained on a large number of semi-trailer lengths covering a range almost continuously, the policy network would most likely during an evaluation perform well on any length in the range it was trained on. However training on such a large amount of scenarios is irrational and highly inefficient. If instead a reasonable number of training scenarios are used, one can investigate what generalisation performance that can be achieved. There is a long list of methods tackling the generalisation challenge but as a first step knowing what varied training scenarios can result in is essential to achieve desired generalisation performance. Good performance in the generalisation evaluation equals a cumulative episode reward of at least four. In a generalisation perspective, high reward is achieved by performing optimal actions even though the scenario and thereby observation input differ from what has been presented during training.

According to the generalisation experiments described in section 3.4.3 and the result presented in figure 4.11 all policies perform relatively poor on semi-trailer lengths that it has not specifically been trained on, i.e. intermediate scenarios. In a visual observation setup, an intermediate scenario is a questionable concept. When referring to this, it means a scenario in-between others made up of shorter and longer semi-trailers in a sequence. When looking at the marked goal reached rates in figure 4.11 these indicate that policy four manages to complete the task in 60% of the episodes when evaluated on a 12.5 meter semi-trailer. Considering that the closest scenario that the policy has been trained on is a semi-trailer that is half a meter longer or shorter, one could argue that 60% success rate is quite good. However, when visually examining the docking simulation it is clear that the drastically reduced reward is justified. The agent fails to dock correctly by crashing and then repeatedly tries again until it either succeeds or fails within the episode time. Therefore the success rate does not drop down to zero, since it only takes into account if the goal was reached. In a generalization perspective this is not only bad though, as it clearly shows that the policy is making use of experiences from other scenarios by taking half-decent actions instead of being completely unable to act. With more training scenarios there is therefore great potential for further generalization.

When further analysing figure 4.11 one clear trend is distinguishable. The two policies trained on the largest amount of different semi-trailer lengths perform better on unseen intermediate scenarios. A larger number of lengths meant an increased range between the longest and shortest while the intervals stayed unchanged. Training a policy to perform well on a wide range of semi-trailer lengths implies that it has to handle different kinematics. It is there therefore a more complex task to solve, but still, performance improved over the whole range and not only for the specific lengths that were added to the training scenario. This is most likely because it forces the policy to understand more of the actual kinematics and dynamics and thereby what a meaningful behaviour is, in order to be able to complete the task. With unchanged kinematics during training, the policy could instead e.g. have associated position of the truck with when it was time to steer and any slight change to the environment or task would then have led to failure. Even if the policies fail to fully generalise to an extent that makes them unusable on intermediate scenarios, most certainly they still become more robust to small changes or imperfections in the environment when increasing the number of training scenarios. As mentioned, the result in figure 4.11 shows average cumulative rewards that in this environment indicate bad performance on several intermediate semi-trailer lengths. The trend that generalisation is improved when increasing the range of lengths is not proven to increase performance all the way to an acceptable reward level in the whole range. Most likely there is a limit but due to time limitations, the limit was not explored. A possible bottleneck could be that the network simply reaches its capacity limit.

6

Conclusion

6.1 Observation complexity scaling

It is difficult to determine exactly why visual and measured position does not collaborate well. It is not safe to say that this is always the case in all RL implementations. However, if the environment setup and visual observation resolution are similar to what is used in this thesis, it is likely.

One explanation could be that the visual input works well to estimate position, to therefore provide position measurements could clash with the neural networks understanding of position according to pixel values, making it harder to learn patterns. One can also conclude that the neural network struggles to extract more detailed information such as angles from the top-down image. However, with both velocity and angle being easy to measure, not requiring any advanced sensors, in a real-world implementation it can be reasonable to base a RL learned network on measurements as in case 6 if using only velocity does not provide sufficient performance.

6.2 Task complexity scaling

How to measure complexity in an environment is highly individual, but one can from the results conclude that the largest negative impact on training performance is found when increasing the kinematic complexity. The negative impact can not be counteracted by simply increasing training time, since having more complex kinematics introduces instabilities in training and puts greater demand on the neural network. When instead changing task to something more difficult such as reversing, but still using the same dynamic vehicle model, training stability remains similar and convergence is reached by allowing for more training steps. As mentioned above this can also be treated by modifying the reward function, which is not as effective on a more complex vehicle model. To counteract instability one can instead provide additional detailed measurements, as discussed in section 5.1.

6.3 Generalisation

By varying the simulation environment during training the trained policy can increase its ability to handle unseen scenarios close to the ones presented during training. Performing changes that "stretches" and challenge existing kinematics in the dynamic environment especially appear to affect generalisation in a positive way.

Even if the goal is to achieve good performance and generalisation capability in a specific range of scenarios. Adding training scenarios outside of that range challenges the policy and in order to succeed it is constrained to really learn mechanisms in the environment. Generalise and learning a meaningful behaviour therefore benefits greatly from varying the environment.

7
Future Work

RL is an endless topic and that is why testing and further improvements can be done on all parts of the implementation in this thesis. There are a couple of areas that would be especially interesting to look into further. The main limiting factor is time since each experiment often requires days of training. Therefore, future work should mainly be focused on expanding and continuing investigations that have already been started to increase reliability of results. Examples could be to:

- Try other combinations of observations
- Feed stacked visual inputs, as a perception of velocity, and train without any measurements
- Investigate generalisation performance when training on more semi-trailer lengths

In regards to performance of the policy network it is possible to develop the network architecture, testing other training algorithms and other reward functions. It could lead to improvement in training time as well as quality of the actual parking job e.g. smoothness, accuracy and docking time. One could try to adapt performance to what is needed for a real world implementation e.g. smaller maximum deviation in docking angle and distance to dock once stopped.

The theory section 2.3 brings up multiple methods for improving generalisation that was not tested in practice. It would be valuable to have deeper knowledge on what effects they have, leaving a demand for further experiments. One could try:

- Implement other algorithms
- Investigate impact of neural network size
- Other regularisation techniques

When the environment used in this thesis was developed it was done to roughly mimic a real world scenario with a vision to make findings useful in real applications. It would therefore be interesting to investigate how the outcome of this thesis could be transferred to reality. e.g. applying lifelike textures to the observation, train until convergence, and then testing the policy on a real scale model of a semi-trailer combination.

Bibliography

[1] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization", arXiv.org, 2017. [Online]. Available: https://arxiv.org/abs/1412.6980. [Accessed: 23- Mar- 2020].

[2] V. Mnih et al., "Playing Atari with Deep Reinforcement Learning", arXiv.org, 2013. [Online]. Available: https://arxiv.org/abs/1312.5602. [Accessed: 14- May- 2020].

[3] A. Krizhevsky, I. Sutskever and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", Dl.acm.org, 2017. [Online]. Available: https://doi.org/10.1145/3065386. [Accessed: 23- Mar- 2020].

[4] I. Goodfellow, Y. Bengio and A. Courville, Deep Learning. MIT Press, 2016.

[5] S. Zhang and R. Sutton, "A Deeper Look at Experience Replay", arXiv.org, 2018. [Online]. Available: https://arxiv.org/abs/1712.01275. [Accessed: 23- Mar- 2020].

[6] J. Ho and S. Ermon, "Generative Adversarial Imitation Learning", arXiv.org, 2016. [Online]. Available: https://arxiv.org/abs/1606.03476. [Accessed: 23- Mar- 2020].

[7] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", arXiv.org, 2015. [Online]. Available: https://arxiv.org/abs/1502.03167. [Accessed: 21- May- 2020].

[8] T. DeVries and G. Taylor, "Improved Regularization of Convolutional Neural Networks with Cutout", arXiv.org, 2017. [Online]. Available: https://arxiv.org/abs/1708.04552. [Accessed: 21- May- 2020].

[9] E. Cubuk, B. Zoph, D. Mane, V. Vasudevan and Q. Le, "AutoAugment: Learning Augmentation Policies from Data", arXiv.org, 2019. [Online]. Available: https://arxiv.org/abs/1805.09501. [Accessed: 21- May- 2020].

[10] S. Lawrence, C. Giles and A. Tsoi, What Size Neural Network Gives Optimal Generalization? Convergence Properties of Backpropagation, 2nd ed. Queensland: Department of Electrical and Computer Engineering University of Queensland, 1996.

[Online]. Available: https://clgiles.ist.psu.edu/papers/UMD-CS-TR-3617.what.size.neural.net.to.use.pdf. [Accessed: 21- May- 2020].

[11] C. Packer, K. Gao, J. Kos, P. Krähenbühl, V. Koltun and D. Song, "Assessing Generalization in Deep Reinforcement Learning", arXiv.org, 2020. [Online]. Available: https://arxiv.org/abs/1810.12282. [Accessed: 13- Apr- 2020].

[12] K. Lee, K. Lee, J. Shin and H. Lee, "Network Randomization: A Simple Technique for Generalization in Deep Reinforcement Learning", arXiv.org, 2020. [Online]. Available: https://arxiv.org/abs/1910.05396. [Accessed: 13- Apr- 2020].

[13] J. Hui, "RL-Importance Sampling", Medium, 2020. [Online]. Available: https://medium.com/@jonathan_hui/rl-importance-sampling-ebfb28b4a8c6. [Accessed: 21- May- 2020].

[14] U. Larsson, C. Zell, K. Hyyppa and Å. Wernersson, Navigating an articulated vehicle and reversing with a trailer. San Diego: Proceedings of the 1994 IEEE International Conference on Robotics and Automation, 1994.

[15] D. Acharya, W. Yan and K. Khoshelham, "Real-time image-based parking occupancy detection using deep learning", Ceur-ws.org, 2018. [Online]. Available: http://ceur-ws.org/Vol-2087/paper5.pdf. [Accessed: 21- May- 2020].

[16] T. Jaakkola, S. Singh and M. Jordan, "Reinforcement Learning Algorithm for Partially Observable Markov Decision Problems", ResearchGate, 1999. [Online]. Available: https://www.researchgate.net/publication/2457557_Reinforcement_Learning_Algorithm_for_
[Accessed: 08- Jun- 2020].

[17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, "Proximal Policy Optimization Algorithms", arXiv.org, 2017. [Online]. Available: https://arxiv.org/abs/1707.06347. [Accessed: 23- Mar- 2020].

[18] L. Weng, "Policy Gradient Algorithms", Lil'Log, 2020. [Online]. Available: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html#policy-gradient. [Accessed: 21- May- 2020].

[19] K. Cobbe, O. Klimov, C. Hesse, T. Kim and J. Schulman, "Quantifying Generalization in Reinforcement Learning", arXiv.org, 2019. [Online]. Available: https://arxiv.org/abs/1812.02341. [Accessed: 07- May- 2020].

[20] A. Ecoffet, "An Intuitive Explanation of Policy Gradient", Medium, 2020. [Online]. Available: https://towardsdatascience.com/an-intuitive-explanation-of-policy-gradient-part-1-reinforce-aa4392cbfd3c. [Accessed: 21- May- 2020].

[21] R. Sutton and A. Barto, Reinforcement Learning: An Introduction, 2nd ed. London: MIT Press, 2018.

[22] T. Haarnoja et al., "Soft Actor-Critic Algorithms and Applications", arXiv.org, 2019. [Online]. Available: https://arxiv.org/abs/1812.05905. [Accessed: 20-Mar- 2020].

[23] J. Achiam, Simplified PPO-Clip Objective. OpenAI, 2018. Available: https://drive.google.com/file/d/1PDzn9RPvaXjJFZkGeapMHbHGiWWW20Ey/view. [Accessed 7- May- 2020].

[24] J. Schulman, S. Levine, P. Moritz, M. Jordan and P. Abbeel, "Trust Region Policy Optimization", arXiv.org, 2017. [Online]. Available: https://arxiv.org/abs/1502.05477. [Accessed: 23- Mar- 2020].

[25] M. Mattar, A. Juliani, V. Berges, D. Pang, J. Ward and J. Shih, "Training with Proximal Policy Optimization", GitHub, 2018. [Online]. Available: https://github.com/gzrjzcx/ML-agents/blob/master/docs/Training-PPO.md. [Accessed: 28- May- 2020].