DEEP LEARNING FOR MAP-LESS NAVIGATION: SMOOTH VELOCITY CHANGE

Submitted by

WALTER TAY ANN LEE

Department of
Mechanical Engineering

In partial fulfilment of the
requirements for the Degree of
Bachelor of Engineering
National University of Singapore

Session 2019/2020

Supervisor:
Associate Professor ZHANG Yunfeng

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

郑安利

Walter TAY Ann Lee

23 March 2020

To my friends from ME and USP

Acknowledgments

All of the work described in this thesis was done under the supervision of my advisor, Dr. Zhang Yunfeng. I would like to thank him for giving me the flexibility of exploring my thesis in a different direction, which allowed me to make many useful insights and connections in the field of deep reinforcement learning.

I would also like to thank Zhang Wei, who helped guide my learning process and gave insightful answers to the problems that I had. The work on a learning-based approach to motion profile smoothing grew out of discussions with my friends from computer science, He Yuchen and Khaw Yew Onn. Thanks to Raivat Shah for collaborating with me on projects not included in this thesis document, which helped develop my understanding of deep learning. I am also immensely grateful to Eddie Lim Jia Lok, who helped design the illustrations for the neural network architectures featured in this thesis.

Finally, this thesis is dedicated to my parents, Tay Boon Chit and Laura Lim Swee Yin, for all the years of love and support.

Contents

Summary

Deep learning for map-less navigation: Smooth Velocity Change

by

Walter TAY Ann Lee

Bachelor of Engineering in Mechanical Engineering

National University of Singapore

This work compares a learning-based approach with a hard coding approach to smoothing the motion profile of a robot trained to perform mapless navigation.

It is necessary to smooth the motion profile to meet the physical limits of the robot as well as to reduce unwanted vibrations. These vibrations occur for robots trained in deep reinforcement learning (DRL) because conventional DRL models for map-less navigation implicitly assume that arbitrary motion in the configuration space is allowed as long as obstacles are avoided.

We design a reward feature which punishes the robot for performing jerky actions. We compare the results of training the robot using this reward feature with the results of training the robot with a velocity smoother instead.

Our results show that the robot fails to learn a trapezoidal or S-curve motion profile to reduce jerk. The robot also underperforms our velocity smoother in terms of jerk reduction. These results highlight the difficulty of tuning the appropriate reward function for the robot to learn a desired behaviour and shows instead how hard coding the limitations of the robot can be very effective. Further research may consider how to simultaneously learn optimal behaviour which is known (e.g. velocity smoothing) with optimal behaviour which is not known (obstacle avoidance and target-driven navigation). For example, designing a training schema which incorporates both inverse reinforcement learning and reinforcement learning may achive this.

List of Figures

List of Symbols

v_t	Linear Velocity at time t
ω_t	Angular Velocity at time t
$accel_t$	Linear Acceleration at time t
α_t	Angular Acceleration at time t
j_t	Linear Jerk at time t
ζ_t	Angular Jerk at time t
y_t	270-degree sparse laser scan at time t
α_{EWMA}	Smoothing factor
MDP	Markov Decision Process
S	State space: a set of states of the environment
A	Action space: a set of actions which the agent selects from each timestep
$P(r, s'\|s, a)$	Transition probability distribution
π	Policy
θ	Parameters of the policy
$R(s_t, a_t, s_{t+1})$	Reward function
s_t	Current state at time t
a_t	Action taken at time t
r_t	Reward received at time t
γ	Discount factor
d_t	Distance between robot and its target at time t
δt	Time for each step
C	Constant used as a time penalty
$P(\tau\|\pi)$	Probability of a T-step trajectory
$J(\pi)$	Expected return
π^*	Optimal policy

$V^\pi(s)$	On-Policy Value Function
$Q^\pi(s,a)$	On-Policy Action-Value Function
$V^*(s)$	Optimal Value Function
$Q^*(s,a)$	Optimal Action-Value Function
$A^\pi(s,a)$	Advantage Function
ϕ	Parameters of the Deep Q-Network
ϕ_{targ}	Parameters of the target network
$L(\phi,\mathcal{D})$	Mean-Squared Bellman Error Loss Function

Chapter 1

Introduction

1.1 Brief Description of Project

This work presents a case study of a learning-based approach to smoothing the movement of a real robotic platform. The robotic platform is operated by an end-to-end neural network which is trained using deep reinforcement learning (DRL) to perform target driven map-less navigation. Previous works [7], [9], [11] have focused more on improving the path-finding or trajectory-generating aspect of map-less navigation while this work focuses on the motion profile planning of map-less navigation. That is, the main idea of this work is to test a learning-based approach to motion smoothing and compare it with an approach which hard codes the motion limitations (a non-learning-based approach). A smooth motion profile is taken here to mean a motion profile that has minimal jerk (time derivative of acceleration).

1.2 Problem Definition

Consider the following histogram of the robot's angular velocity:

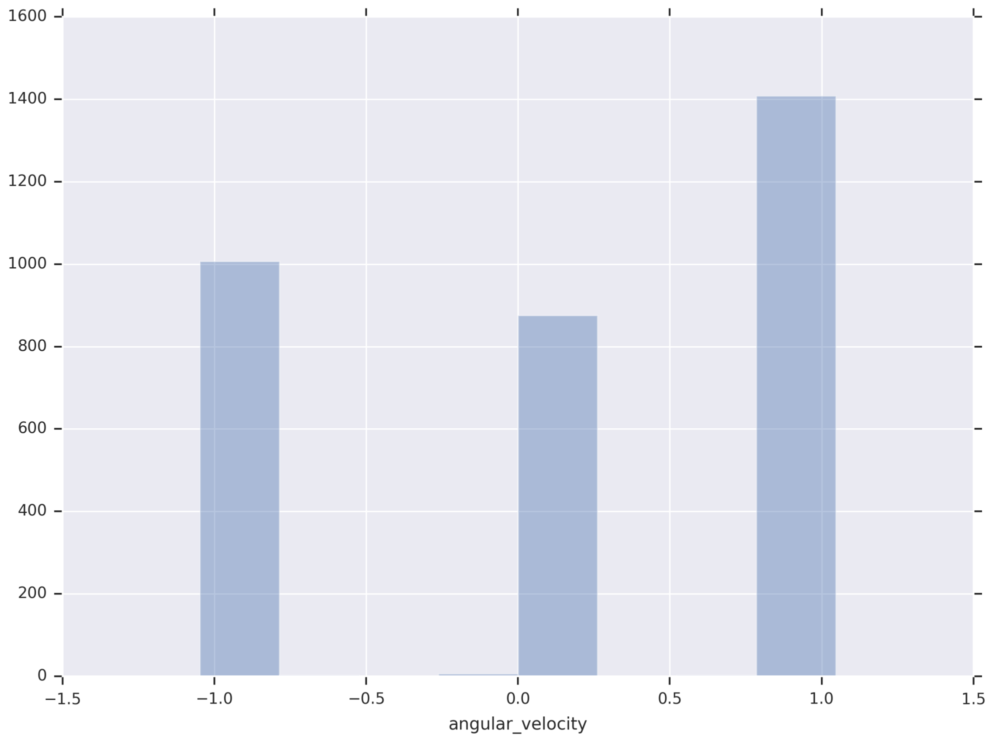


Figure 1.1: Histogram shows the angular velocity of a well-trained robot in 100 episodes of simulated map-less navigation in Complex World § A.2

From Figure 1.1, we observe that after sufficient training, the DRL model only chooses extreme actions for the robot to take (i.e. maximum velocity at 1.0 and -1.0, and minimum velocity at 0). Furthermore, to perform these actions, the robot has to accelerate instantaneously from 0 to 1.0 and vice versa. In practice, this is not possible for the robot due to acceleration limits; however, conventional DRL models for map-less navigation *implicitly assume* that arbitrary motion in the configuration space is allowed as long as obstacles are avoided. Moreover, even if this motion profile were possible it would not be ideal as it creates unnecessary jerk and jerk-induced vibrations for the robot.

Figure 1.2 shows that the robot does indeed experience jerk (and at specific magnitudes):

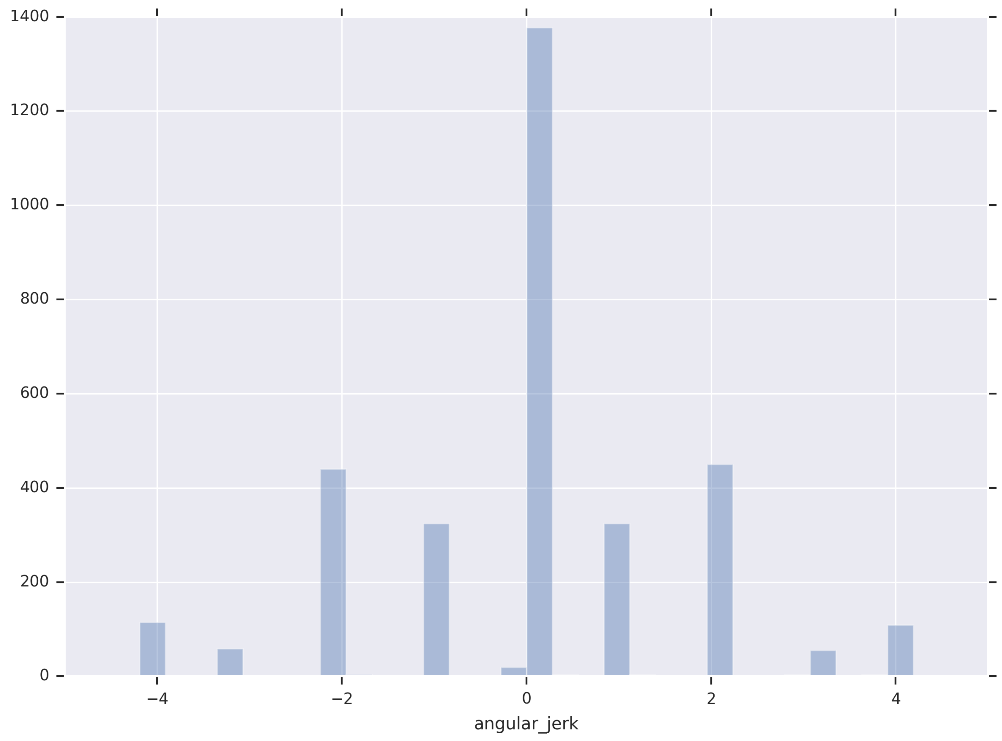


Figure 1.2: Histogram shows the angular jerk of a well-trained robot in 100 episodes of simulated map-less navigation in Complex World § A.2

Motion control methods that achieve fast motions without residual vibrations are already well-known [6]. Thus, applying an trapezoidal velocity profile to velocities generated by the DRL model should be able to reduce jerk and vibrations.

However, the main contribution of this paper is to demonstrate a case study of a learning-based approach to the same problem. We look at the potential performance impacts, impact to training time, and motion smoothing capabilities that such an approach entails, as well as how it compares to a model with a velocity smoother.

Chapter 2

Background

This section provides the background required to understand this work. Readers who already have an understanding of Deep Reinforcement Learning and Deep Deterministic Policy Gradient (DDPG) may skip this section.

We would like to note that a large portion of this section has been reproduced from OpenAI's Spinning Up in Deep RL [1]. What is different in this reproduction, however, is that the background information on DRL has been adapted to fit the ideas and concepts related to map-less navigation found in this work.

2.1 Deep Reinforcement Learning

Deep reinforcement learning (DRL) is the combination of reinforcement learning and deep learning. This idea of using neural networks for reinforcement learning, however, is not new and can be dated all the way back to Teasauro's TD-Gammon [10]. In the early 2010s, however, the field of deep learning began to find groundbreaking success, particularly in speech recognition [2] and computer vision [4]. This success, combined with the advances in computing power, allowed the revival in interest of using deep neural networks as universal function approximators for reinforcement learning - leading to deep reinforcement learning.

Deep reinforcement learning is useful when [1]:

- we have a sequential decision-making problem (which we can represent as a Markov Decision Process)
- we do not know the optimal behaviour (e.g. multi-modal problem)

- but we can still evaluate whether behaviours are good or bad

2.1.1 Markov Decision Process

To train our agent with deep reinforcement learning, we must be able to model the relationship between the agent and its environment. A Markov Decision Process (MDP) is a mathematical object that describes our agent interacting with a stochastic environment. It is defined by the following components:

- S: **state space**, a set of states of the environment. This is the set of inputs for our robot which contain a view of the world (i.e. a stack of laser scans, the current speed of the robot, and the target position with respect to the robot's local frame)
- A: **action space**, a set of actions, which the agent selects from each timestep. The actions taken by our robot to reach its target are its linear velocity and angular velocity, both of which are continuous-valued variables
- $P(r, s'|s, a)$: **transition probability distribution**. For each state s and action a, P specifies the probability that the environment will emit reward r and transition to state s'. This transition function describes the relationship between states, actions, next states, and rewards in an environment.

2.1.2 Policies

The end goal is to find a policy π, which tells the agent what actions to take given a state. In DRL, we use parameterized policies: policies whose outputs are computable functions that depend on a set of parameters (e.g. the weights and biases of a neural network) which we can adjust to change the behavior via some optimization algorithm [1].

We denote the parameters of such a policy by θ, and then write this as a subscript on the policy symbol to highlight the connection:

$$a_t \sim \pi_\theta(\cdot|s_t) \tag{2.1}$$

2.1.3 Reward and Return

The reward function R is critically important in reinforcement learning (and specifically for this work). It depends on the current state of the world, the action just taken, and the next state of the world:

$$r_t = R(s_t, a_t, s_{t+1}) \tag{2.2}$$

The goal of an agent is to maximise the cumulative reward over a trajectory. In this work, the type of return used is called an **infinite-horizon discounted return**, which is the sum of all rewards ever obtained by the agent, but discounted by how far off in the future they're obtained. This formulation of reward includes a discount factor $\gamma \in (0, 1)$

Specifically (prior to adding the reward feature proposed in this work), this work uses the reward function for map-less navigation proposed in Xie et. al's AsDDPG network [11]:

$$Reward, r_t = \begin{cases} R_{crash} & \text{if robot crashes,} \\ R_{reach} & \text{if robot reaches the goal,} \\ \gamma((d_{t-1} - d_t)\Delta t - C) & \text{otherwise.} \end{cases}$$

2.1.4 The RL Problem

The goal of reinforcement learning is to select a policy which maximises **expected return** when the agent acts according to it. To talk about expected return, we first have to talk about probability distributions over trajectories.

Let's suppose that both the environment transitions and the policy are stochastic. In this case, the probability of a T-step trajectory is:

$$P(\tau|\pi) = p_0(s_0) \prod_{t=0}^{T-1} P(s_{t-1}|s_t, a_t)\pi(a_t|s_t) \tag{2.3}$$

The expected return, denoted by $J(\pi)$, is then:

$$J(\pi) = \int_\tau P(\tau|\pi)R(\tau) = \tau \sim \pi R(\tau) \tag{2.4}$$

The central optimization problem in RL can then be expressed by:

$$\pi^* = arg\underset{\pi}{max} J(\pi) \tag{2.5}$$

with π being the optimal policy.

2.1.5 Value Functions

It's often useful to know the **value** of a state, or state-action pair. By value, we mean the expected return if you start in that state or state-action pair, and then act according to a particular policy forever after. Value functions are used, one way or another, in almost every RL algorithm.

There are four main functions of note:

1. The **On-Policy Value Function**, $V^{\pi}(s)$, which gives the expected return if you start in a state s and always act according to policy π:

$$V^{\pi}(s) = \tau \sim \pi R(\tau)|s_0 = s \tag{2.6}$$

2. The **On-Policy Action-Value Function**, $Q^{\pi}(s, a)$, which gives the gives the expected return if you start in state s, take an arbitrary action a (which may not have come from the policy), and then forever after act according to policy π:

$$Q^{\pi}(s, a) = \tau \sim \pi R(\tau)|s_0 = s, a_0 = a \tag{2.7}$$

3. The **Optimal Value Function**, $V^{*}(s)$, which gives the expected return if you start in state s and always act according to the *optimal* policy in the environment

$$V^{*}(s) = \underset{\pi}{max} \tau \sim \pi R(\tau)|s_0 = s \tag{2.8}$$

4. The **Optimal Action-Value Function**, $Q^{*}(s, a)$, which gives the expected return if you start in state s, take an arbitrary action a, and then forever after act according to the *optimal* policy in the environment:

$$Q^{*}(s, a) = \underset{\pi}{max} \tau \sim \pi R(\tau)|s_0 = s, a_0 = a \tag{2.9}$$

2.1.6 The Optimal Q-Function and the Optimal Action

There is an important connection between the optimal action-value function $Q^{*}(s, a)$ and the action selected by the optimal policy. By definition, $Q^{*}(s, a)$ gives the expected return for starting in state s, taking (arbitrary) action a, and then acting according to the optimal policy forever after.

The optimal policy in s will select whichever action maximises the expected return from starting in s. As a result, if we have Q^*, we can directly obtain the optimal action, $a^*(s)$, via:

$$a^*(s) = \arg\max_a Q^*(s,a) \tag{2.10}$$

This connection is important; some RL algorithms learn by improving their policy (thereby directly obtaining "good" actions), whereas others learn by improving their Q-Function (which allows them to determine the best action from a set of actions, indirectly obtaining "good" actions).

2.1.7 Bellman Equations

All four of the value functions obey special self-consistency equations called **Bellman equations**. The basic idea behind the Bellman equations is this:

The value of your starting point is the reward you expect to get from being there, plus the value of wherever you land next.

The Bellman equations for the on-policy value functions are:

$$V^\pi(s) = \underset{\substack{a\sim\pi \\ s'\sim P}}{E}[r(s,a) + \gamma V^\pi(s')] Q^\pi(s,a) = \underset{s'\sim P}{E}[r(s,a) + \gamma \underset{a'\sim\pi}{E}[Q^\pi(s',a')]] \tag{2.11}$$

where $s' \sim P$ is shorthand for $s' \sim P(\cdot|s,a)$, indicating that the next state s' is sampled from the environment's transition rules; $a \sim \pi$ is shorthand for $a \sim \pi(\cdot|s)$; and $a' \sim \pi$ is shorthand for $a' \sim \pi(\cdot|s')$.

The Bellman equations for the optimal value functions are:

$$V^*(s) = \underset{a}{max} \underset{s'\sim P}{E}[r(s,a)+\gamma V^*(s')] Q^*(s,a) = \underset{s'\sim P}{E}[r(s,a)+\gamma \underset{a'}{max}[Q^*(s',a')]] \tag{2.12}$$

The crucial difference between the Bellman equations for the on-policy value functions and the optimal value functions, is the absence or presence of the *max* over actions. Its inclusion reflects the fact that whenever the agent gets to choose its action, in order to act optimally, it has to pick whichever action leads to the highest value.

2.1.8 Advantage Functions

Sometimes in RL, we don't need to describe how good an action is in an absolute sense, but only how much better it is than others on average. That is to say, we want to know the relative **advantage** of that action. We make this concept precise with the **advantage function**.

The advantage function $A^{\pi}(s,a)$ corresponding to a policy π describes how much better it is to take a specific action a in state s, over randomly selecting an action according to $\pi(\cdot|s)$, assuming you act according to π forever after. Mathematically, the advantage function is defined by

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \tag{2.13}$$

2.2 Deep Deterministic Policy Gradient (DDPG)

The network presented in this work is a variant of DDPG [11]. As such, it is crucial that we also provide a background for what DDPG is, and how it works.

Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy.

This approach is closely connected to Q-learning, and is motivated the same way: if you know the optimal action-value function $Q^*(s,a)$, then in any given state, the optimal action $a^*(s)$ can be found by solving:

$$a^*(s) = \arg\max_a Q^*(s,a) \tag{2.14}$$

DDPG interleaves learning an approximator to $Q^*(s,a)$ with learning an approximator to $a^*(s)$, and it does so in a way which is specifically adapted for environments with continuous action spaces. But what does it mean that DDPG is adapted specifically for environments with continuous action spaces? It relates to how we compute the max over actions in $\max_a Q^*(s,a)$.

When there are a finite number of discrete actions, the max poses no problem, because we can just compute the Q-values for each action separately and directly compare them. (This also immediately gives us the action which maximizes the

Q-value.) But when the action space is continuous, we can't exhaustively evaluate the space, and solving the optimization problem is highly non-trivial. Using a normal optimization algorithm would make calculating $\max_a Q^*(s,a)$ a painfully expensive subroutine. And since it would need to be run every time the agent wants to take an action in the environment, this is unacceptable.

Because the action space is continuous, the function $Q^*(s,a)$ is presumed to be differentiable with respect to the action argument. This allows us to set up an efficient, gradient-based learning rule for a policy $\mu(s)$ which exploits that fact. Then, instead of running an expensive optimization subroutine each time we wish to compute $\max_a Q(s,a)$, we can approximate it with $\max_a Q(s,a) \approx Q(s,\mu(s))$.

2.2.1 Quick Facts

- DDPG is an off-policy algorithm
- DDPG can only be used for environments with continuous action spaces
- DDPG can be thought of as being deep Q-learning for continuous action spaces

Next, we'll explain the math behind the two parts of DDPG: learning a Q function, and learning a policy.

2.2.2 The Q-Learning Side of DDPG

First, let's recap the Bellman equation describing the optimal action-value function, $Q^*(s,a)$. It's given by

$$Q^*(s,a) = \underset{s' \sim P}{E}[r(s,a) + \gamma \max_{a'} Q^*(s',a')] \tag{2.15}$$

where $s' \sim P$ is shorthand for saying that the next state, s', is sampled by the environment from a distribution $P(\cdot|s,a)$.

This Bellman equation is the starting point for learning an approximator to $Q^*(s,a)$. Suppose the approximator is a neural network $Q_\phi(s,a)$, with parameters ϕ, and that we have collected a set $\mathcal{D}$ of transitions (s,a,r,s',d) (where d indicates whether state s' is terminal). We can set up a mean-squared Bellman error (MSBE) function, which tells us roughly how closely Q_ϕ comes to satisfying the Bellman equation:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}}\left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d)\max_{a'} Q_\phi(s',a')\right)\right)^2\right] \tag{2.16}$$

Here, in evaluating $(1-d)$, we've used a Python convention of evaluating True to 1 and False to 0. Thus, when $d ==$ True—which is to say, when s' is a terminal state—the Q-function should show that the agent gets no additional rewards after the current state.

Q-learning algorithms for function approximators, such as DQN (and all its variants) and DDPG, are largely based on minimizing this MSBE loss function. There are two main tricks employed by all of them which are worth describing, and then a specific detail for DDPG.

Trick One: Replay Buffers. All standard algorithms for training a deep neural network to approximate $Q^*(s,a)$ make use of an experience replay buffer. This is the set $\mathcal{D}$ of previous experiences. In order for the algorithm to have stable behavior, the replay buffer should be large enough to contain a wide range of experiences, but it may not always be good to keep everything. If you only use the very-most recent data, you will overfit to that and things will break; if you use too much experience, you may slow down your learning. This may take some tuning to get right.

Trick Two: Target Networks. Q-learning algorithms make use of target networks. The term

$$r + \gamma(1-d)\max_{a'} Q_\phi(s',a') \tag{2.17}$$

is called the target, because when we minimize the MSBE loss, we are trying to make the Q-function be more like this target. Problematically, the target depends on the same parameters we are trying to train: ϕ. This makes MSBE minimization unstable. The solution is to use a set of parameters which comes close to ϕ, but with a time delay—that is to say, a second network, called the target network, which lags the first. The parameters of the target network are denoted ϕ_{targ}.

In DQN-based algorithms, the target network is just copied over from the main network every some-fixed-number of steps. In DDPG-style algorithms, the target network is updated once per main network update by polyak averaging:

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi, \tag{2.18}$$

where ρ is a hyperparameter between 0 and 1 (usually close to 1).

DDPG Detail: Calculating the Max Over Actions in the Target. As mentioned earlier: computing the maximum over actions in the target is a challenge in continuous action spaces. DDPG deals with this by using a target policy network to compute an action which approximately maximizes $Q_{\phi_{\text{targ}}}$. The target policy network is found the same way as the target Q-function: by polyak averaging the policy parameters over the course of training.

Putting it all together, Q-learning in DDPG is performed by minimizing the following MSBE loss with stochastic gradient descent:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}}\left[\left(Q_{\phi}(s,a) - \Big(r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))\Big)\right)^2\right], \quad (2.19)$$

where $\mu_{\theta_{\text{targ}}}$ is the target policy.

2.2.3 The Policy Learning Side of DDPG

Policy learning in DDPG is fairly simple. We want to learn a deterministic policy $\mu_{\theta}(s)$ which gives the action that maximizes $Q_{\phi}(s,a)$. Because the action space is continuous, and we assume the Q-function is differentiable with respect to action, we can just perform gradient ascent (with respect to policy parameters only) to solve

$$\max_{\theta} \underset{s\sim\mathcal{D}}{\mathrm{E}}\left[Q_{\phi}(s, \mu_{\theta}(s))\right]. \quad (2.20)$$

Note that the Q-function parameters are treated as constants here. Exploration vs. Exploitation

DDPG trains a deterministic policy in an off-policy way. Because the policy is deterministic, if the agent were to explore on-policy, in the beginning it would probably not try a wide enough variety of actions to find useful learning signals. To make DDPG policies explore better, we add noise to their actions at training time.

At test time, to see how well the policy exploits what it has learned, we do not add noise to the actions.

Chapter 3
Main Text

The main contribution of this work is to test a learning-based approach to smoothing the motion profile of a DRL model. We do this in the following sequence:

1. Introduce smooth motion profiles
2. Present the network architectures used to apply this learning-based approach
3. Discuss the empirical findings of this learning-based approach as well as compare it with a non-learning based approach

3.1 Smooth Motion Profile

In map-less navigation, we deal with point-to-point motion. That is, the goal of the robot is to move from its starting position and reach some target end position while avoiding obstacles. In terms of motion, point-to-point means that from a stop (zero velocity), the load is accelerated to a constant velocity, and then decelerated such that the final acceleration, and velocity, are zero at the moment the load arrives at the programmed destination. However, achieveing fast motions without residual vibration is a challenge that pervades many applications of point-to-point motion [6], including deep reinforcement learning for map-less navigation.

To perform point-to-point motion with minimal vibration, previous works have focused on optimization of the motion profile itself [6]. For example, Figure 3.1 shows a comparison between a typical trapezoidal motion profile and an (improved) S-curve motion profile.

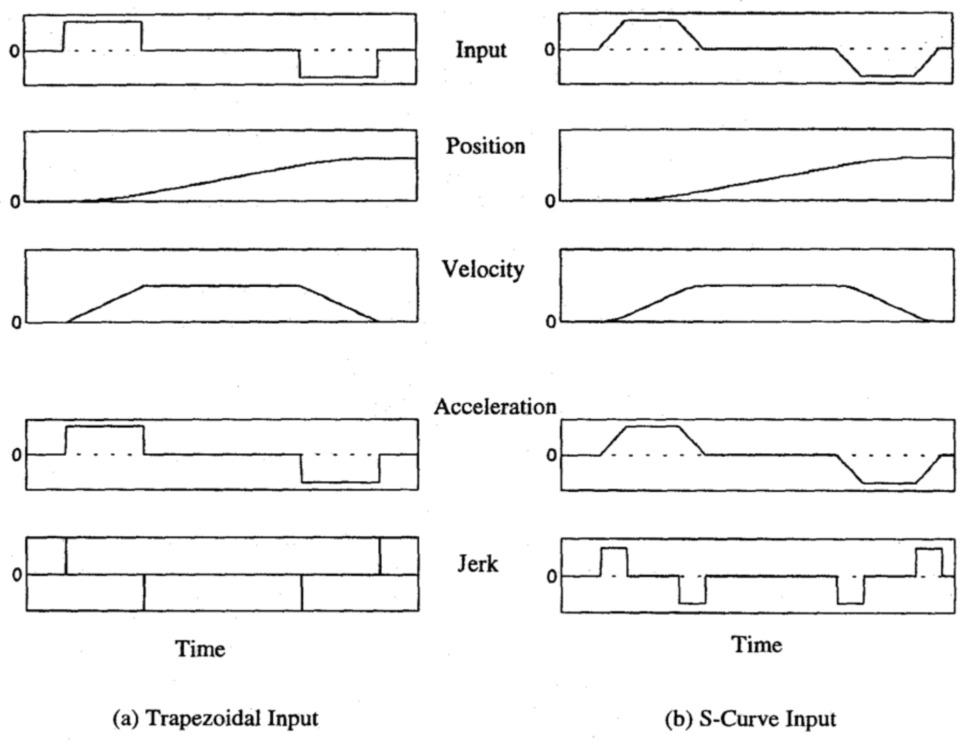


Figure 3.1: Comparison of Trapezoidal and S-Curve Motion Profiles. Image from Meckl and Arestides, 1998 [6]

As can be seen from Figure 3.1, smoothing a motion profile involves jerk reduction (or some manipulation of the jerk profile). Jerk is the time-derivative of acceleration, that is:

$$j(t) = \frac{da(t)}{dt} = \frac{d^2v(t)}{dt^2} \tag{3.1}$$

3.2 Proposed Approach

3.2.1 Three Different Models

In this work, we compare the motion profiles of the following three models:

1. **Normal Model**: The default end-to-end neural network trained with deep reinforcement learning to perform map-less navigation

2. **Model with Velocity Smoother**: The same network as Normal Model but trained with a simple velocity smoother

3. **Model with Jerk-Learning**: A network very similar to Normal Model but with an added reward function that penalizes the robot for the jerk generated. The inputs of this model are also slightly different (for the reward function to work properly).

3.2.2 State

Normal Model and *Model with Jerk-Learning* have different features included in their inputs (states) but they both have the same output (action).

At a certain time step t, the state of the agent (robot) trained with *Normal Model* is described by:

- a stack of three 270-degree sparse laser scans y (y_t, y_{t-1}, y_{t-2})
- the current linear and angular velocity of the robot (v_t, ω_t)
- x and y components of the robot's displacement from its target position $(d_{x,t}, d_{y,t})$

In the *Model with Jerk-Learning*, the acceleration and jerk of the robot are included in the input layer. The idea is to allow the robot to use information about its motion profile to reduce the jerk generated (in combination with the reward function to incentivize this behaviour). The inputs for this model are:

- a stack of three 270-degree sparse laser scans y (y_t, y_{t-1}, y_{t-2})
- a stack of linear and angular velocities of the robot $(v_t, \omega_t, v_{t-1}, \omega_{t-1}, v_{t-2}, \omega_{t-2})$
- a stack of linear and angular accelerations of the robot $(accel_t, \alpha_t, accel_{t-1}, \alpha_{t-1}, accel_{t-2}, \alpha_{t-2})$
- a stack of linear and angular jerks of the robot $(j_t, \zeta_t, j_{t-1}, \zeta_{t-1}, j_{t-2}, \zeta_{t-2})$
- x and y components of the robot's displacement from its target position $(d_{x,t}, d_{y,t})$

3.2.3 Action

For all 3 models, the output (action) is a linear velocity (v_t) and an angular velocity (ω_t). However, the action for the *Model with Velocity Smoother* is slightly different as it is constrained by a simple velocity smoother. In this work, we use the following algorithm for simple velocity smoothing:

Algorithm 1: Simple Velocity Smoother

```
1: if Target Velocity < Control Velocity then
2:     Control Velocity = min(Target Velocity, Control Velocity + $v_c$)
3: else if Target Velocity > Control Velocity then
4:     Control Velocity = max(Target Velocity, Control Velocity - $v_c$)
5: else
6:     Control Velocity = Target Velocity
```

Where:

- v_c is some constant velocity value

Essentially, Algorithm 1 is an acceleration limiting approach to motion control where a maximum acceleration is assumed. This generates a trapezoidal motion profile similar to the one found in Figure 3.1.

3.2.4 Simulation Setup

To enable the DRL agent to learn from trial and error, we use a simple robot simulation environment called Stage. Stage provides a virtual world populated by mobile robots and sensors, along with various objects for the robots to sense and manipulate. We use Stage in ROS (Robot Operating System) to train our DRL agent [3].

3.2.5 Reward Function

The reward function, $r(s, a)$, serves as a training signal to encourage or discourage behaviours in the context of a desired task [12]. Thus, the features chosen for the reward function must capture the relevant objectives of the task. In this work, the following reward function was used for the *Normal Model*:

$$Reward, r_t = \begin{cases} -5 & \text{if robot crashes,} \\ 5 & \text{if robot reaches the goal,} \\ \gamma\Big((d_{t-1} - d_t)\Delta t - C\Big) & \text{otherwise.} \end{cases} \tag{3.2}$$

Where:

- γ acts as a discount factor (for infinite-horizon discounted return)
- d_{t-1} and d_t indicate the distance between the robot and its target at two consecutive time stamps
- Δt represents the time for each step
- C is a constant used as a time penalty.

Note that the policy learnt with this reward function may be *sub*-optimal. This is due to the fact that the agent is driven by something else other than reaching the target in the shortest time possible. Ideally, a sparse reward function would be better as it would only reward the robot when it actually reaches the target; however, a sparse reward function is challenging for random exploration.

For the *Model with Jerk-Learning*, a new feature which penalizes the robot for jerky movements is added to the reward function. We have chosen to use a dense penalty function for the new feature as it would alleviate the challenges of learning the appropriate motion profile to minimize jerk:

$$Reward, r_t = \begin{cases} -5 & \text{if robot crashes,} \\ 5 & \text{if robot reaches the goal,} \\ \gamma\Big((d_{t-1} - d_t)\Delta t - C\Big) - & \\ \frac{\gamma}{25}\left(\left(\frac{j_t}{\max j_t}\right)^2 + \left(\frac{\zeta_t}{\max \zeta_t}\right)^2\right) & \text{otherwise.} \end{cases} \tag{3.3}$$

Where:

- j_t is linear jerk at time t
- $\max j_t$ is the theoretical maximum linear jerk for each state transition
- ζ_t is angular jerk at time t
- $\max \zeta_t$ is the theoretical maximum angular jerk for each state transition

3.2.6 Network Architecture

The network architecture we use is largely similar to Xie et. al's Assisted DDPG network [11]. The difference, however, is that we have discarded the convolutional neural network (CNN) layers and used a sparser laser beam to allow the agent to learn the policy faster. We reiterate: the goal here is not to produce a state-of-the-art model for map-less navigation, but to empirically study a learning-based approach to jerk reduction. Figure 3.2 shows the network architecture of the *Normal Model*.

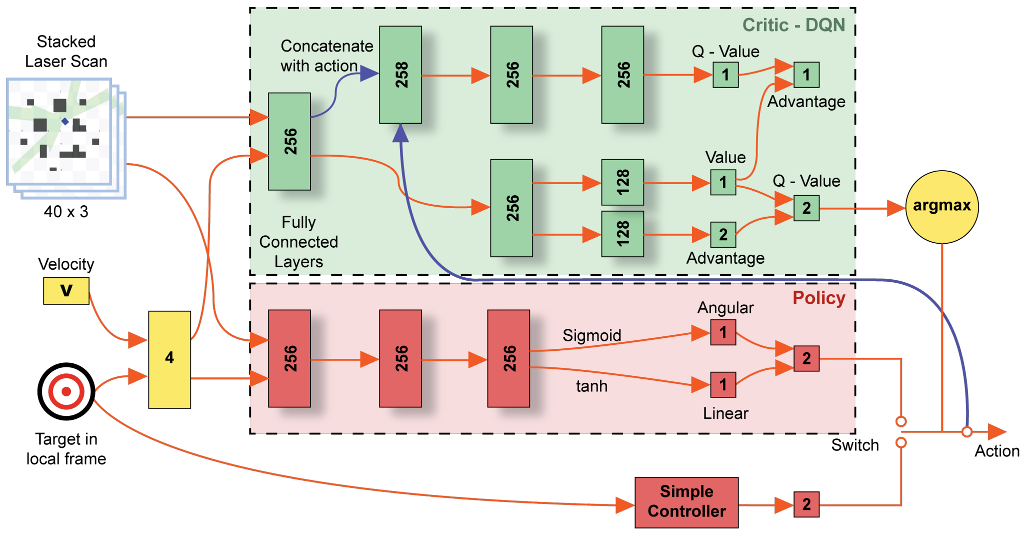


Figure 3.2: Network architecture. Network layers are demonstrated by rectangles. Orange arrows indicate the connectivity between network layers and some other components, e.g. input state and the output of the simple controller. Blue arrows indicate the operation of concatenation (i.e. tf.concat). The final action is selected based on the Q-value predicted by the critic-DQN.

The network architecture of Figure 3.2 consists of two parts:

1. **Policy network and assistive controller (red)**: The policy network consists of a fully-connected layers which estimates the optimal linear and angular velocities for the robot based only on the input state (i.e., laser scans, current speed, and target position in the local frame). This is a very important feature of DDPG as sampling actions from a policy is computationally less expensive than exhaustively evaluating as many discrete actions as possible (since our action space is continuous). Note that the activation functions of the out-

puts of the policy are sigmoid (linear velocity) and tanh (angular velocity), respectively.

2. **Augmented critic network (green)**: The critic-DQN is also constructed with fully-connected layers. It has two branches: one is the critic branch where the action is concatenated into the second layer; the other is the DQN branch where (similar to Xie et. al [11]) we apply dueling and double network architecture to speed up training and avoid overestimation. Note that there is no nonlinear activation for its output layers.

The network architecture of the *Model with Velocity Smoother* is exactly the same as the *Normal Model* but with a velocity smoother at the end. It is shown in Figure 3.4:

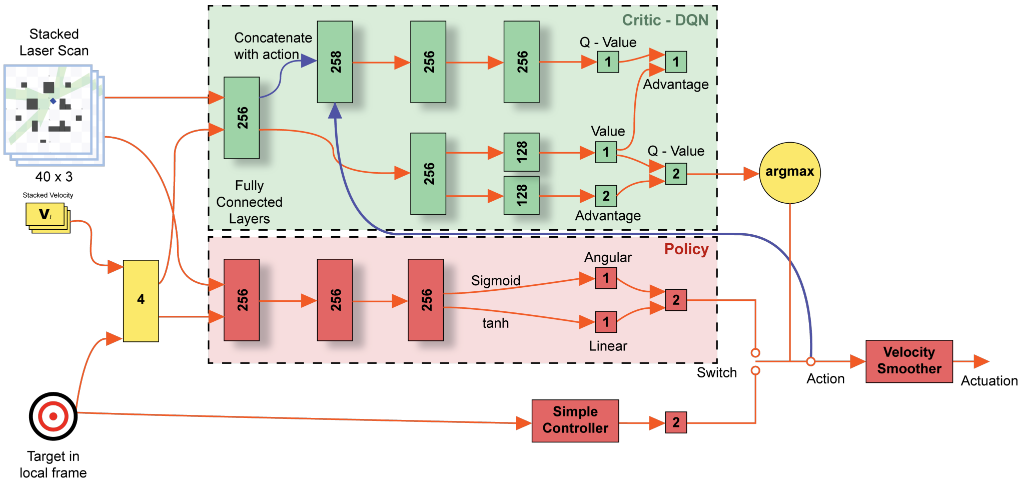


Figure 3.3: Network architecture of Model with Velocity Smoother. Output layer has a velocity smoother.

The network architecture of the *Model with Jerk-Learning* is slightly different in terms of its inputs (as mentioned in § 3.2.2). It is shown in Figure 3.4:

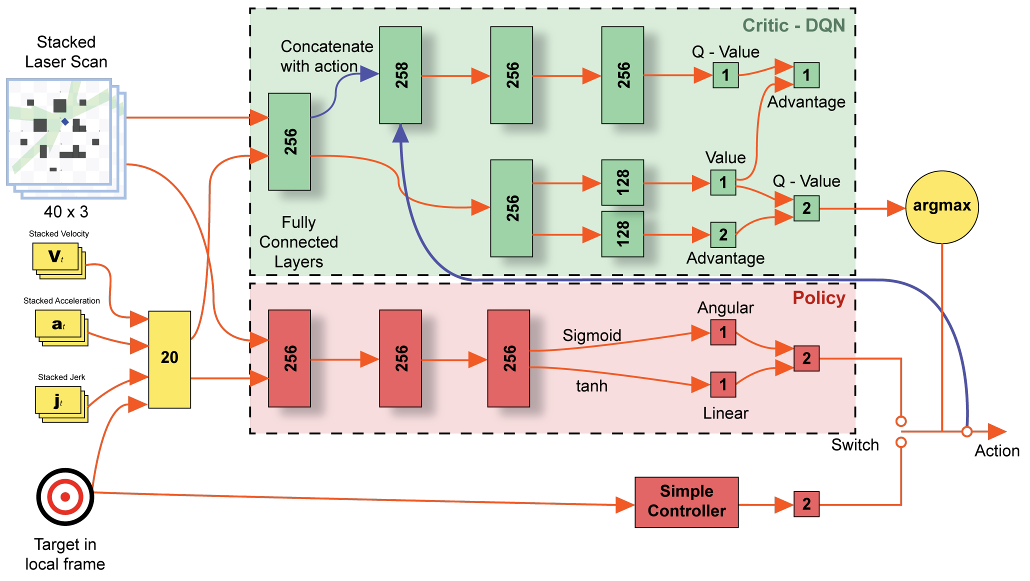


Figure 3.4: Network architecture of Model with Jerk-Learning. Input layer now includes stacks of velocities, accelerations, and jerks of the robot.

3.2.7 Training Methodology

The training procedure of both networks is according to Algorithm 2 [11]:

Algorithm 2: Assisted DDPG

```
1:  Initialize $A(s, a|\theta^A), Q(s, \sigma|\theta^Q)$, and $\pi(s, \theta^\pi)$;
2:  Initialize target network $\theta^{A'}, \theta^{Q'}$, and $\theta^{\pi'}$;
3:  Initialize replay buffer $R$ and exploration noise $\epsilon$;
4:  while Episode is not terminal do
5:      Reset the environment;
6:      Obtain the initial observation;
7:      for step = 1 do
8:          Infer switching $[Q_{policy}, Q_P] = Q(s_t, \sigma|\theta^Q)$
            $\sigma_t = \arg\max([Q_{policy}, Q_P])$ if $\sigma_t == 1$ then
9:              Sample policy action $a_t = \pi(s_t|\theta^\pi) + \epsilon$
10:         else
11:             Sample action $a_t$ from external controller
12:         Execute $a_t$ and obtain $r_t, s_{t+1}$;
13:         Store transition $(s_t, a_t, r_t, s_{t+1}, \sigma_t)$ in R;
14:         Sample N transitions $(s_i, a_i, r_i, s_{i+1}, \sigma_i)$ in R;
15:         Optimize critic-DQN by minimizing loss function;
16:         Update policy;
17:         Update target networks;
```

All 3 models were trained in 2 different Stage simulation worlds § A.1, § A.2. In each simulation world, the networks were trained until the average performance reached a plateau (i.e. when gradient of average performance with respect to time is approximately zero).

3.3 Experiment Results

In this section, we discuss and compare the results of training the 3 different models. Visualizing these results, however, remains a difficulty. Consider Figure 3.5:

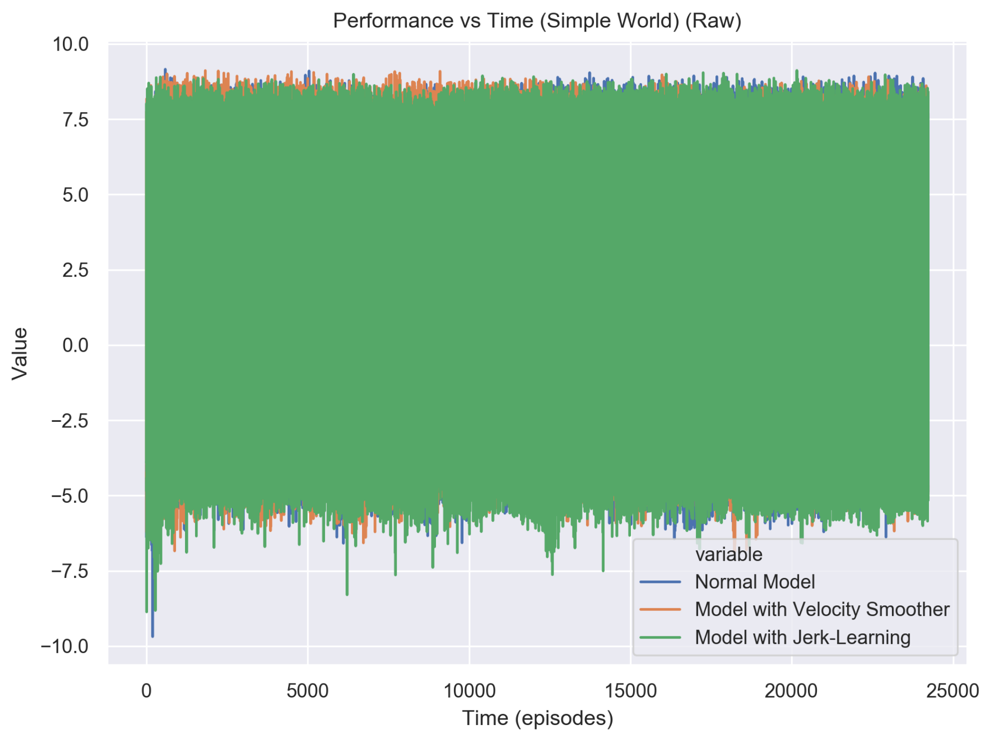


Figure 3.5: Graph of Performance vs Time (Raw).

It is difficult to see how well the model is performing over time from Figure 3.5 purely based on the raw results. Therefore, we have chosen to use an exponentially weighted moving average (EWMA) to better visualize how the agent's performance is improving over time (as well as other variables which we will look at in this work). EWMA is calculated as follows:

$$y(t) = \frac{x_t + (1-\alpha_{EWMA})x_{t-1} + (1-\alpha_{EWMA})^2 x_{t-2} + ... + (1-\alpha_{EWMA})^t x_0}{1 + (1-\alpha_{EWMA}) + (1-\alpha_{EWMA}^2) + ... + (1-\alpha_{EWMA})^t} \tag{3.4}$$

Where:

- x_t is the input
- y_t is the result
- α_{EWMA} is the smoothing factor.

While it is possible to pass α_{EWMA} directly, it's often easier to think about the span of an EW moment. In this work, unless otherwise specified, we use a span of 1000 for our smoothed results. This is also known as an "N-day EW moving average". That is:

$$\alpha_{EWMA} = \frac{2}{s+1} \text{ for span s} \geq 1 \tag{3.5}$$

Using EWMA, our graph of performance vs time is now smoothened and we are able to see the long-term trends:

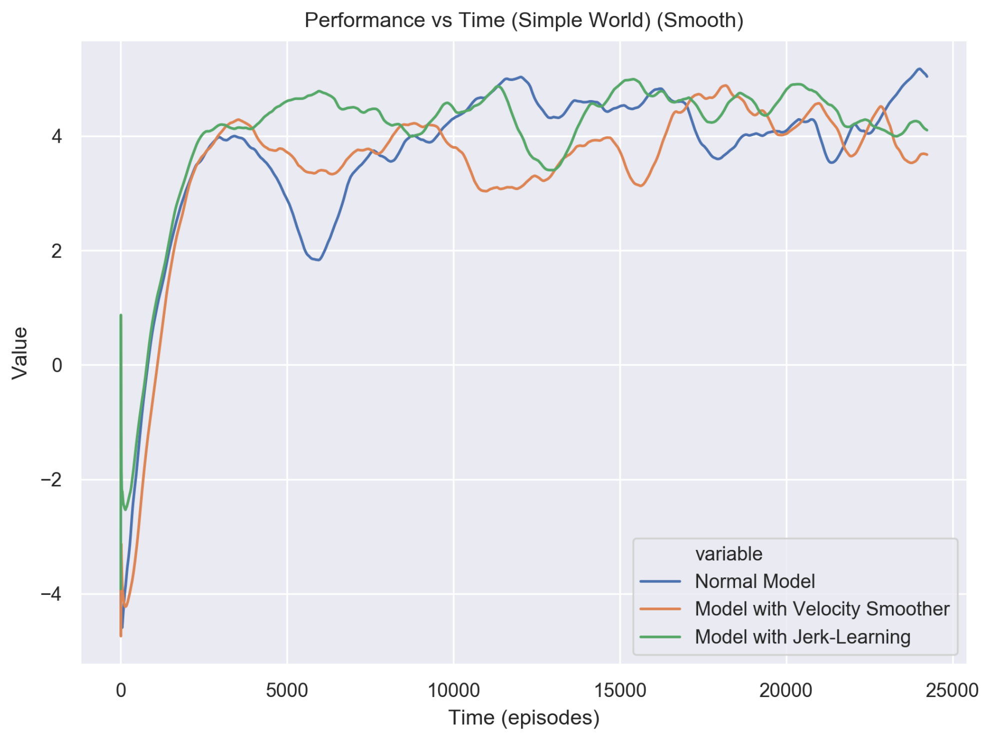


Figure 3.6: Graph of Performance vs Time (Smooth).

3.3.1 Performance of Models

In this section, we compare the performance of the 3 models. We measure performance as the return of a trajectory (which is a measure of cumulative reward along the trajectory). The way this is computed is according to Equation 3.6:

$$\text{Performance} = R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t \tag{3.6}$$

Equation 3.6 (infinite-horizon discounted return) is explained in § 2.1.3.

Figure 3.7 and Figure 3.8 show that all 3 models display performance improvements at approximately equal rates. All 3 models also plateau at roughly the same performance level. We also observe that the *Model with Jerk-Learning* generally outperforms the *Model with Velocity Smoother*.

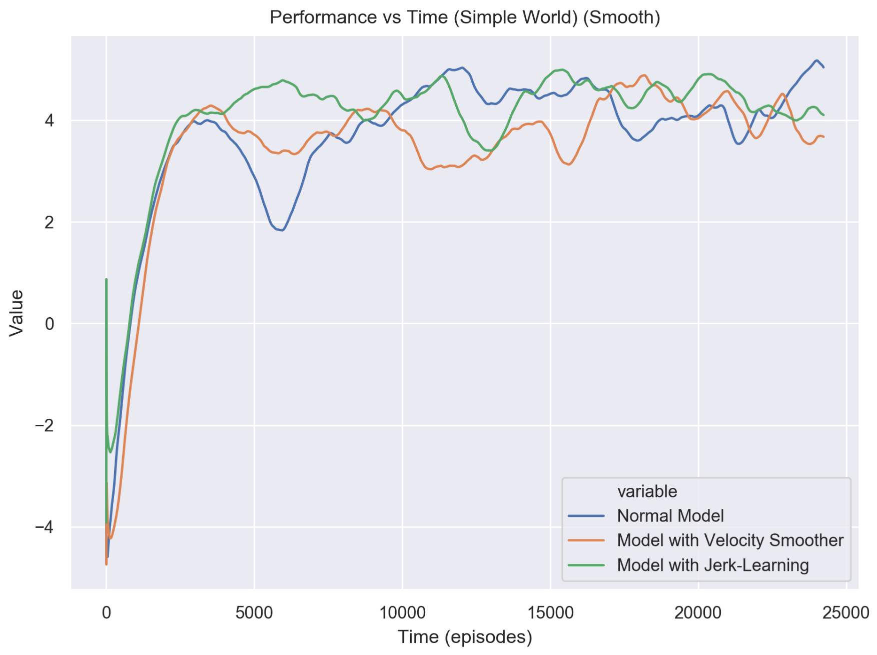


Figure 3.7: Performance vs Time (Simple World)

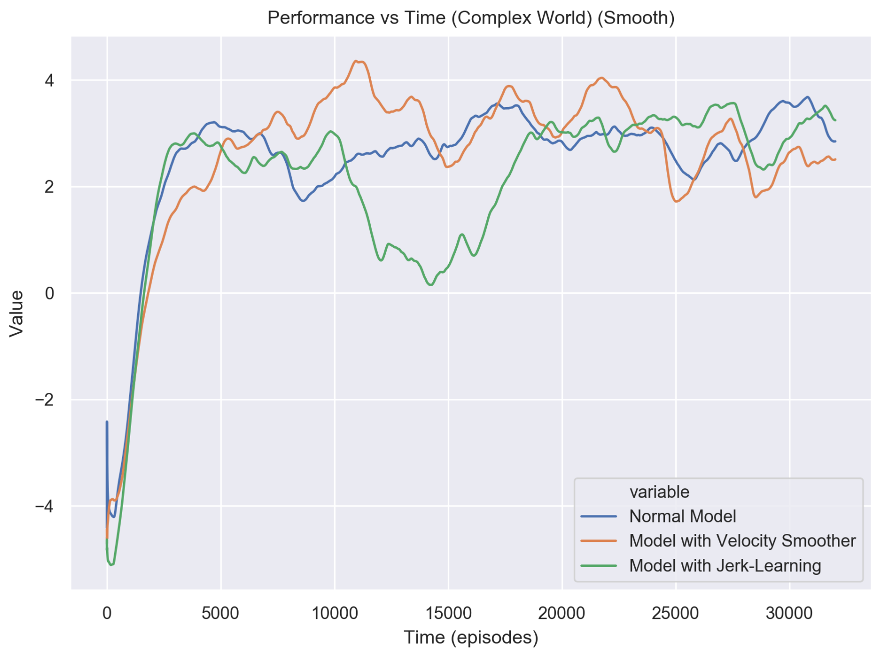


Figure 3.8: Performance vs Time (Complex World)

Figure 3.9 and Figure 3.10 show that all 3 models display improvements in success rate at approximately equal rates. All 3 models also end up plateuing at the same success rate. Again, we observe that the *Model with Jerk-Learning* generally outperforms the *Model with Velocity Smoother*.

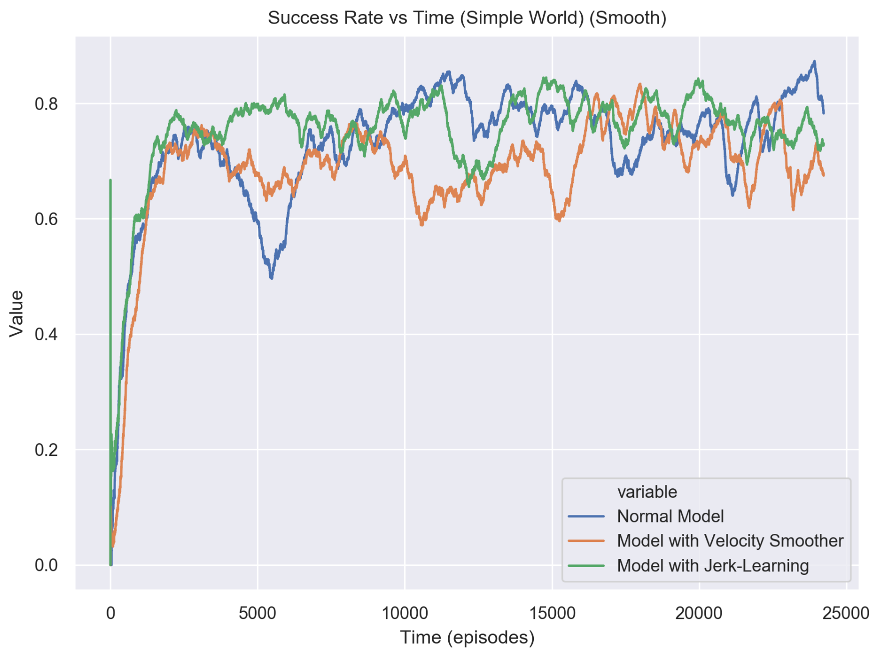


Figure 3.9: Success Rate vs Time (Simple World)

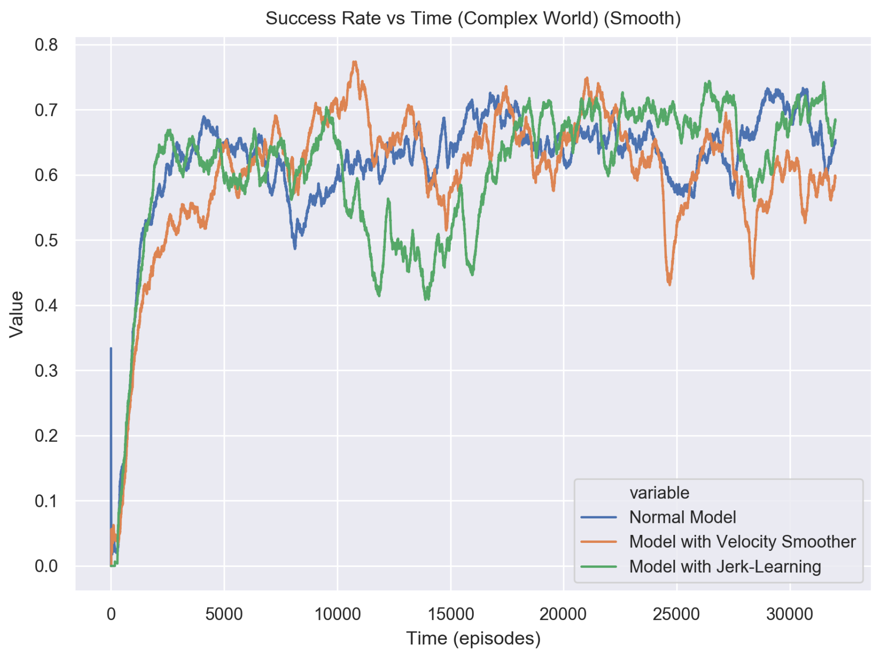


Figure 3.10: Success Rate vs Time (Complex World)

3.3.2 Motion Profile of Models

In this section, we compare the motion profiles of the 3 models. We do this by looking at histograms of linear and angular jerk for each model in each world.

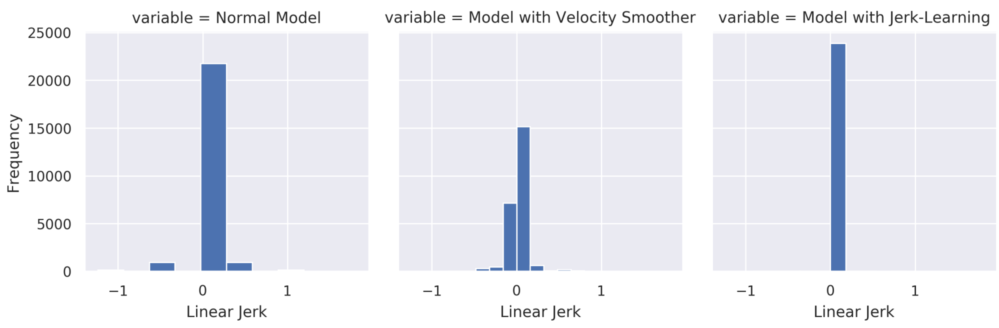


Figure 3.11: Histograms of linear jerk for 500 episodes across all 3 models in Simple World § A.1. Model with Jerk-Learning only slightly outperforms the other models here in terms of jerk reduction.

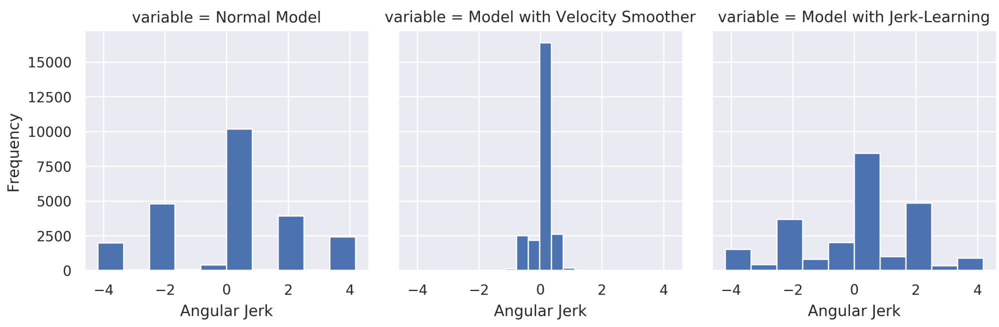


Figure 3.12: Histograms of angular jerk for 500 episodes across all 3 models in Simple World § A.1. Model with Velocity Smoother outperforms the other models here in terms of jerk reduction.

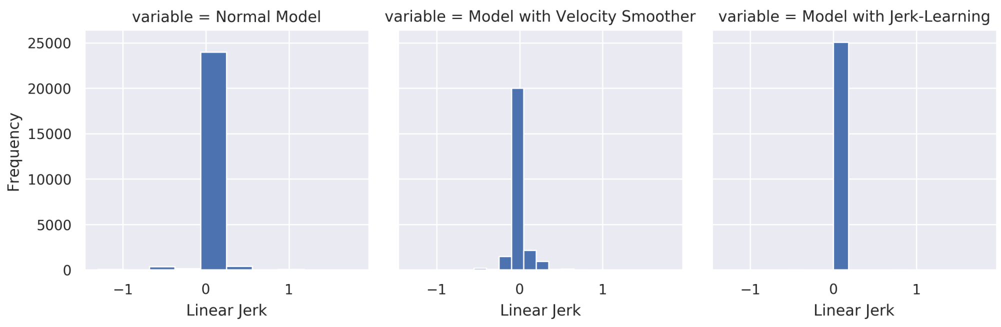


Figure 3.13: Histograms of linear jerk for 500 episodes across all 3 models in Complex World § A.2. Model with Jerk-Learning only slightly outperforms the other models here in terms of jerk reduction.

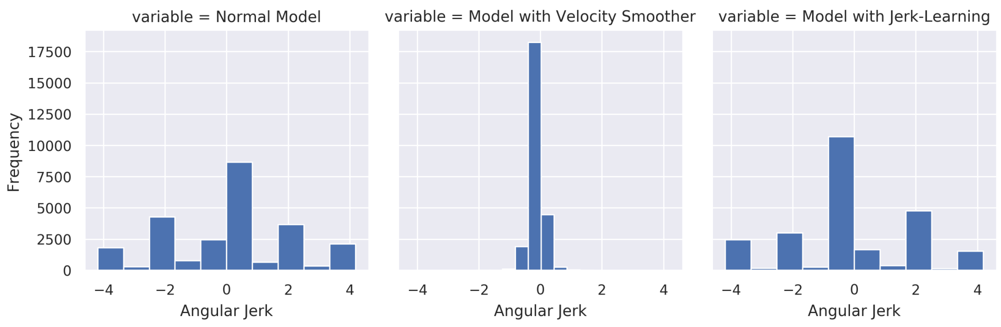


Figure 3.14: Histograms of angular jerk for 500 episodes across all 3 models in Complex World § A.2. Model with Velocity Smoother outperforms the other models here in terms of jerk reduction.

3.4 Discussion

From § 3.3.1, we observe that neither the velocity smoother nor the jerk-learning reward feature seemed to have a significant impact on the training time required.

§ 3.3.2 showed that the *Model with Velocity Smoother* outperformed the other 2 models in terms of jerk reduction. The only cases where it seemed to (very) slightly underperform the *Model with Jerk-Learning* were for linear jerk. We can understand why this is case when we look at the histograms of linear velocities:

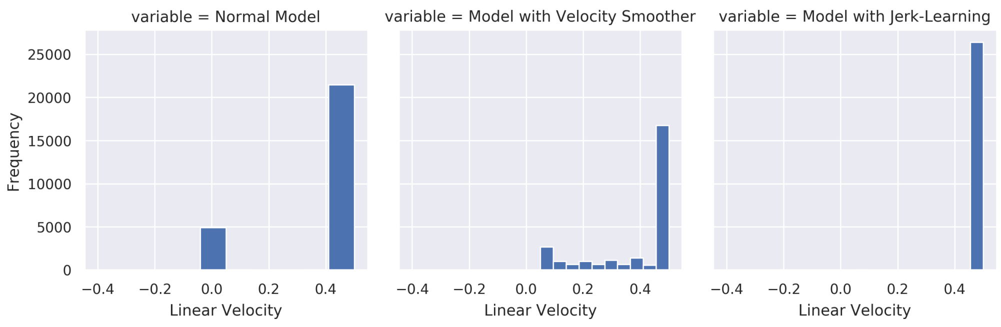


Figure 3.15: Histograms of linear velocity for 500 episodes across all 3 models in Simple World § A.1

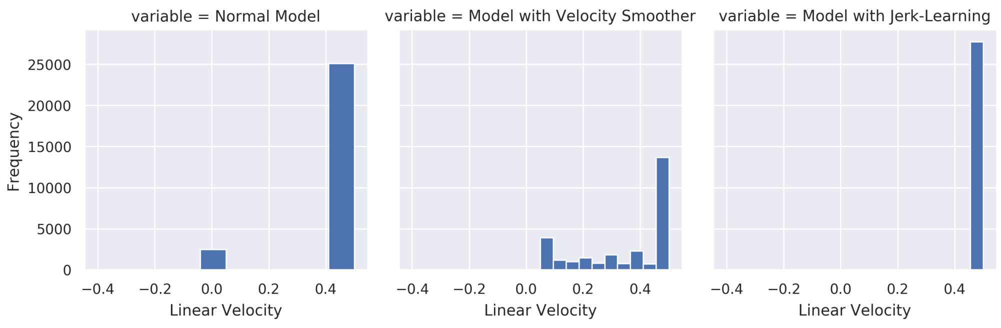


Figure 3.16: Histograms of linear velocity for 500 episodes across all 3 models in Complex World § A.2

From Figure 3.15 and Figure 3.16 we see that to avoid generating any linear jerk the *Model with Jerk-Learning* only produces the highest linear velocity possible in every state. That is, it avoids changes in velocities to avoid generating jerk. This is also the case for angular velocities as can be seen from Figure 3.17 and Figure 3.18

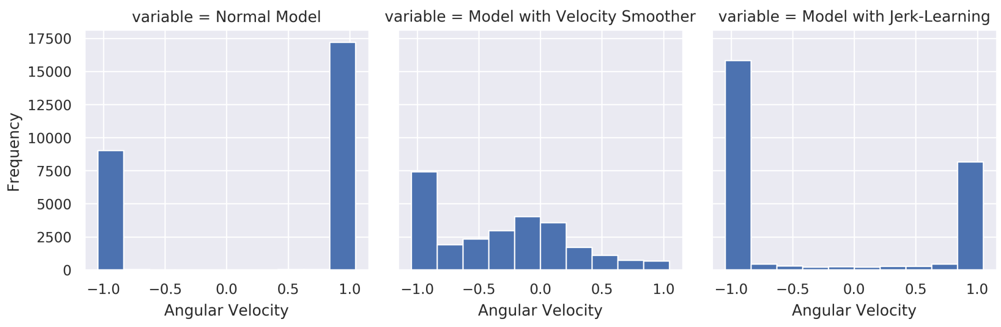


Figure 3.17: Histograms of angular velocity for 500 episodes across all 3 models in Simple World § A.1

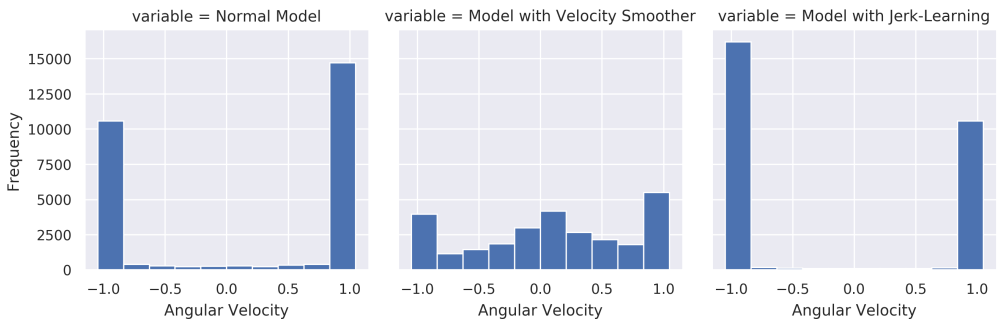


Figure 3.18: Histograms of angular velocity for 500 episodes across all 3 models in Complex World § A.2

Furthermore, Figure 3.17 and Figure 3.18 also show why the *Model with Velocity Smoother* outperforms the other 2 models in terms of angular jerk: the angular velocities generated by *Model with Velocity Smoother* is more spread out.

Chapter 4

Conclusion and Future Work

4.1 Conclusion

In this work, we proposed an approach to smoothing the motion profile of a robot without explicitly coding for the motion profile (i.e. through a learning-based approach). This approach has been shown to not been able to outperform a model with a velocity smoother in terms of jerk reduction. What we have shown instead is that it may be prudent to hard code the mechanical limitations of the robot as this approach may be simpler, and does not impact the training time or success rate of the agent (at least in terms of mapless navigation).

While hard coding the mechanical limitations of the robot may improve the smoothness of the motion profile of the robot, it does have a number of limitations. First, this assumes that the programmer understands what those limitations are in the first place. Secondly, the effort to generate and test the limitations of the robot may be non-trivial. Thirdly, the hard coding the limitations of the robot may not necessarily solve the problem we intend to solve in every scenario (however, in this scenario it happens to work).

4.2 Future Research Directions

Many open problems remain, which relate to and could build on this thesis work. Below, we describe some of the frontiers (which we consider) to be the most exciting.

4.2.1 Inverse Reinforcement Learning

Instead of developing a reward feature with trial and error, inverse reinforcement learning (IRL) allows us to learn what the appropriate reward feature may be by observing the behaviour of an agent that is already capable of performing smooth mapless navigation. We are able to do this because autonomous navigation in environments where global knowledge of the map is available is already well understood (e.g. computing the shortest feasible path with the Dijkstra algorithm) [5], and motion control methods that achieve fast motions without residual vibrations are also well known [6]. We can use this to extract the optimal reward function that allows the agent to learn smooth mapless navigation.

4.2.2 Exploration

The idea behind exploration is to allow the agent to experience unfamiliar parts of the state space and avoid converging to a suboptimal policy. Policy gradient methods are prone to converging to suboptimal policies, as we observed many times during this thesis. Improving exploration may allow the agent to learn policy that can perform smoother mapless navigation.

4.2.3 Prioritized Experience Replay

Experience replay lets DRL agents remember and reuse trajectories from the past. In this work, experience transitions were uniformly sampled from a replay buffer without considering their significance. Prioritizing specific experiences can be utilized to replay important transitions more frequently and learn more efficiently [8]

We believe the above (but not limited to) future research directions will advance the technology presented in this thesis and contribute to academia and industry.

Bibliography

[1] J. Achiam, “Spinning Up in Deep Reinforcement Learning”,, 2018. [Online]. Available: `https://github.com/openai/spinningup`.

[2] G. E. Dahl, Dong Yu, Li Deng, and A. Acero, “Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition”, *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 20, no. 1, pp. 30–42, Jan. 2012, ISSN: 1558-7916, 1558-7924. [Online]. Available: `http://ieeexplore.ieee.org/document/5740583/` (visited on 02/22/2020).

[3] B. Gerkey, R. T. Vaughan, and A. Howard, “The player/stage project: Tools for multi-robot and distributed sensor systems”,, vol. 1, pp. 317–323, 2003.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks”, *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, May 24, 2017, ISSN: 00010782. [Online]. Available: `http://dl.acm.org/citation.cfm?doid=3098997.3065386` (visited on 02/22/2020).

[5] S. M. LaValle, “Planning Algorithms”, Cambridge: Cambridge University Press, 2006, ISBN: 978-0-511-54687-7 978-0-521-86205-9. [Online]. Available: `https://www.cambridge.org/core/product/identifier/9780511546877/type/book` (visited on 12/29/2019).

[6] P. Meckl and P. Arestides, “Optimized s-curve motion profiles for minimum residual vibration”, in *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No.98CH36207)*, Philadelphia, PA, USA: IEEE, 1998, 2627–2631 vol.5, ISBN: 978-0-7803-4530-0. [Online]. Available: `http://ieeexplore.ieee.org/document/688324/` (visited on 01/06/2020).

[7] M. Pfeiffer, S. Shukla, M. Turchetta, C. Cadena, A. Krause, R. Siegwart, and J. Nieto, “Reinforced imitation: Sample efficient deep reinforcement learning for map-less navigation by leveraging prior demonstrations”, *arXiv:1805.07095*

[cs], Aug. 31, 2018. arXiv: `1805.07095`. [Online]. Available: `http://arxiv.org/abs/1805.07095` (visited on 12/23/2019).

[8] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized experience replay”, *arXiv:1511.05952 [cs]*, Feb. 25, 2016. arXiv: `1511.05952`. [Online]. Available: `http://arxiv.org/abs/1511.05952` (visited on 03/22/2020).

[9] L. Tai, G. Paolo, and M. Liu, “Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation”, *arXiv:1703.00420 [cs]*, Jul. 21, 2017. arXiv: `1703.00420`. [Online]. Available: `http://arxiv.org/abs/1703.00420` (visited on 01/01/2020).

[10] G. Tesauro, “Td-gammon, a self-teaching backgammon program, achieves master-level play”, *Neural computation*, vol. 6, no. 2, pp. 215–219, 1994.

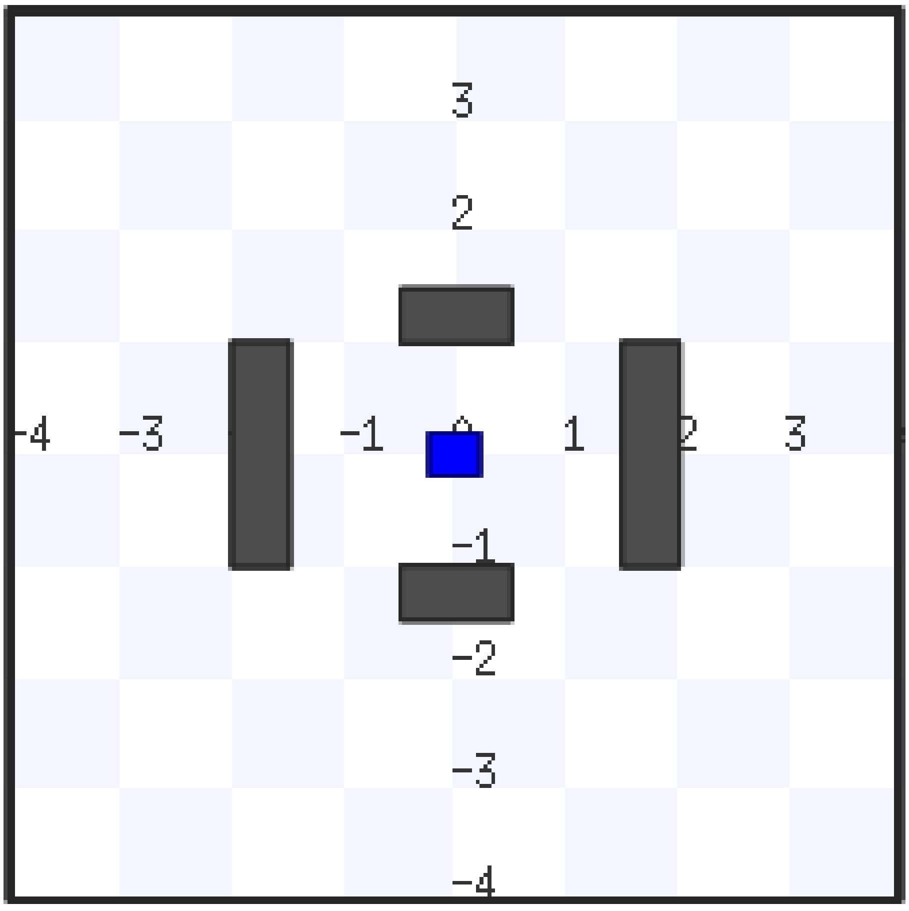
[11] L. Xie, S. Wang, S. Rosa, A. Markham, and N. Trigoni, “Learning with training wheels: Speeding up training with a simple controller for deep reinforcement learning”, in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, Brisbane, QLD: IEEE, May 2018, pp. 6276–6283, ISBN: 978-1-5386-3081-5. [Online]. Available: `https://ieeexplore.ieee.org/document/8461203/` (visited on 12/30/2019).

[12] M. Zhu, Y. Wang, Z. Pu, J. Hu, X. Wang, and R. Ke, “Safe, efficient, and comfortable velocity control based on reinforcement learning for autonomous driving”, *arXiv:1902.00089 [cs, stat]*, Oct. 31, 2019. arXiv: `1902.00089`. [Online]. Available: `http://arxiv.org/abs/1902.00089` (visited on 12/23/2019).

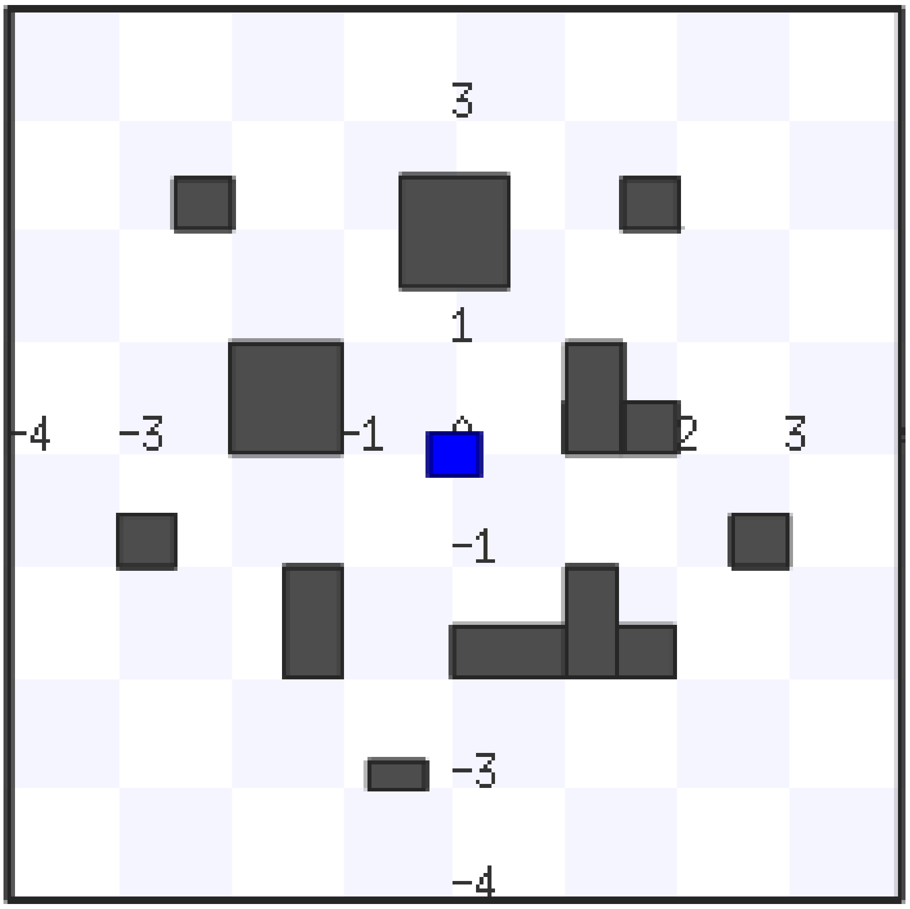
Appendix A

Stage Simulation Worlds

A.1 Simple World



A.2 Complex World


3
1
-4
-3
-1
0
2
3
-1
-3
-4