Evaluating the robustness of DQN and QR-DQN under domain randomization

Analyzing the effects of domain variation on value-based methods

Youri Zwetsloot[1]

Supervisor(s): Frans A. Oliehoek[1]**, Mustafa Mert Çelikok**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2025

Name of the student: Youri Zwetsloot
Final project course: CSE3000 Research Project
Thesis committee: Frans A. Oliehoek, Mustafa Mert Çelikok, Annibale Panichella

Abstract

Domain randomization (or DR) is a widely used technique in reinforcement learning to improve robustness and enable sim-to-real transfer. While prior work has focused extensively on DR in combination with algorithms such as PPO and SAC, its effects on value-based methods like DQN and QR-DQN remain underexplored. This paper investigates how varying degrees and types of DR affect the robustness and generalization capabilities of agents trained with DQN and QR-DQN. We identify clear differences in how DQN and QR-DQN respond to domain randomization, suggesting that naive application may hinder performance, whereas well-targeted distributions can enhance robustness and generalization. These findings underscore the importance of tailored DR strategies for different algorithms and contribute to a deeper understanding of DR's role in DQN-based methods.

1 Introduction

Due to the complexity, costs, and potential risks of training agents in real-life environments, engineers have often been forced to use simulations instead. It is exceedingly rare for agents trained in these simulations to perform nearly as well in the real-life environments in which they are eventually deployed. This problem is known as the sim-to-reality gap and is one of the most common problems in reinforcement learning (or RL). To combat this problem, engineers now commonly use domain randomization: varying domain properties during training. By randomly varying the environment properties that are expected to vary in real-life environments as well, the agent typically performs better in the latter. This technique is also commonly used to ensure robustness: the algorithm or agent property expressing how the agent performance is affected when environment properties are (slightly) altered.

Domain randomization (or DR) is a critical area in (deep) reinforcement learning, particularly for sim-to-real transfer. Much of the literature focuses on how to design effective DR strategies, often independent of the specific learning algorithm. Poorly designed DR can hinder rather than help robustness or generalizability [10]. To address this, techniques such as Active Domain Randomization aim to reduce the amount of required variation and data by identifying impactful property ranges [10]. Research has also explored how to tune DR parameters using sparse data from real world environments [17]. Zhao et al. [20] provide a good overview of sim-to-real transfer studies with an explicit focus on learning algorithms, highlighting PPO and SAC as the dominant choices in earlier DR work. SAC derives its popularity from its use in problems with constrained real-world data collection, such as robotics tasks [4] [1].

Despite the attention on PPO and SAC, the impact of DR on agents trained with RL methods DQN and QR-DQN remains underexplored. DQN, originally introduced by Mnih et al. in 2013 [11], and QR-DQN [6], a distributional variant, are still frequently used but are often superseded in favor of more recently developed methods. It is important to note that applying domain randomization techniques to a specific RL learning method like SAC may not generalize to other learning methods, such as DQN or QR-DQN. Bayram [3] evaluated DQN's robustness with domain randomization but was limited by computational constraints and did not provide a definitive analysis of DR's effect on performance. Bassani et al. [2] employed DQN and DDPG but focused on domain adaptation rather than domain randomization. The former involves aligning the environment property distribution with the target domain, while the latter focuses on introducing controlled variability in the source domain. As a result, the specific effects of domain randomization on the robustness of DQN and QR-DQN remain insufficiently explored.

In our work, we will be guided by the main research question: *how does domain randomization affect the robustness of DQN and QR-DQN?* This question can be subdivided into 3 (sub)questions:

- **RQ1**: How does the robustness of DQN compare to that of QR-DQN under varying degrees of DR?
- **RQ2**: How is the robustness of DQN (and QR-DQN) affected by different types of DR?
- **RQ3**: How well do DQN and QR-DQN, trained with domain randomization, generalize to unseen environments?

RQ3 deals specifically with the problem of sim-to-real transfer.

This work evaluates the effects of domain randomization on the performance and robustness of agents trained with DQN and QR-DQN. It provides the first focused comparison of these algorithms under varying degrees and types of domain randomization (as far as we can tell), demonstrating clear differences in their sensitivity to variation in environment properties. The study further explores how different approaches to DR influence agent behavior, suggesting that naive application of domain randomization may have limited or even negative effects, while more targeted approaches (particularly those emphasizing challenging conditions) can enhance both robustness and generalization. It also highlights how QR-DQN responds differently to DR compared to DQN. These findings offer insight into the broader class of DQN-based methods such as (Double DQN [16] and Dueling DQN [18]) and underscore the importance of careful domain randomization design.

We will proceed by giving an overview of the background required for our research in Section 2, before moving on to our methodology in Section 3. In Section 4, we will give an overview of the results of our experimentation, followed by a discussion of our results in Section 5. In Section 6 we present our conclusions and suggest possible future research topics. We will wrap up with a brief reflection on the ethical aspects of our research and the reproducibility of our methodology.

2 Background

2.1 Reinforcement learning

Reinforcement learning is a type of machine learning in which an agent learns by interacting with an environment, receiving rewards (or penalties) based on the actions it takes [14].

Single-agent reinforcement learning is typically modeled using a Markov decision process (or MDP). An MDP is a tuple $< S, A, T, R >$ containing a set of states S, set of actions A, transition function T and reward function R. At any point in time, the agent is in one of the states in the state space (the set of possible states) of the environment. By selecting an action from the action space (the set of possible actions), the agent initiates a transition. This results in a new state and yields a corresponding reward or penalty. This process represents a single interaction step. The objective in reinforcement learning is to learn a strategy that maximizes the total accumulated reward over time, also known as the return. This strategy is typically referred to as a policy $\pi(s)$ that maps a state s to an action [14].

In Q-learning, one of the most widely used reinforcement learning algorithms [8], the Q-function $Q^\pi(s, a)$ represents the expected total return of taking action a in state s, and then following a policy π thereafter. The Q-function is defined as:

$$Q^\pi(s,a) = E_\pi \left[\sum_{k=0}^{H-1} \gamma^k r_{k+1} \middle| s_0 = s, a_0 = a \right] \quad (1)$$

Here, H represents the horizon, or the number of steps in the MDP. The discount factor $\gamma \in [0, 1]$ controls how an agent values rewards over time. Low values for γ result in the agent prioritizing short-term rewards, whereas high values result in an agent maximizing long-term rewards [8].

The Q-function assigns a numerical value to each possible action in each possible state, reflecting the long-term benefit of that action. The mapping from state to expected total reward, when following policy π, is called the value function $V_\pi(s)$. In Q-learning the value function corresponds to $V_\pi(s) = \max_a(Q_\pi(s, a))$. The objective of Q-learning is to learn or approximate the Q-function (and corresponding value function) [14].

This paper focuses on value-based methods, which learn a state-action value function to guide policy decisions, in contrast to policy-gradient methods that learn a policy directly.

2.2 DQN

Traditional reinforcement learning techniques rely on linear function approximation. Due to the complexity and high-dimensionality of some state representations (such as raw pixel observations), linear approximations of value functions no longer suffice. To resolve this issue, Mnih et al. [11] introduced the use of (deep) neural networks to create nonlinear approximations of the Q-function. This approach is now known as DQN or Deep Q-Networks.

DQN introduced two key mechanisms to improve stability and efficiency. Firstly, experience replay, which stores past transitions in a buffer to learn more efficiently from a limited number of samples, and randomly selects from it to break the correlation between successive samples. Secondly, two networks are used: an online network and a target network. The parameters of the target network are periodically updated by copying the weights of the online network (whose weights are updated every step). This reduces oscillations during learning [8].

Additionally, DQN typically utilizes an epsilon-greedy policy to select actions during training. With probability ϵ, the agent chooses a random action (exploration), and with probability 1 - ϵ, it selects the action that maximizes its current Q-function estimates (exploitation). Over time, ϵ is gradually reduced to allow the policy to become more deterministic as it gains confidence in its Q-value estimates. The epsilon-greedy policy is a simple and effective way to balance exploration and exploitation in DQN and many other value-based methods [8].

2.3 QR-DQN

Building on DQN, newer methods have addressed some of its limitations and/or further improved performance and robustness. Knowing just the expected return for a particular action (such as in DQN) may be misleading, as two actions could have the same expected return but differ greatly in variance or risk. Instead of predicting just the expected return, QR-DQN estimates a full distribution of possible returns by learning several quantiles of that distribution. The more quantiles we use, the better QR-DQN can estimate the return distribution. This distributional approach captures uncertainty more effectively and can lead to more robust policies. These differences suggest that QR-DQN may respond differently to domain randomization than DQN, motivating a comparative study [6].

2.4 Robustness and the sim-to-reality gap

Robustness is an algorithm or agent property that reflects how well a trained model can deal with (slight) changes in an environment. Improving the robustness of reinforcement learning methods has been an active area of research since the inception of RL.

The simulation-reality gap is a problem closely related to robustness. Due to the costs, dangers, and relative difficulty of training algorithms in the real world, researchers are often forced to use simulations instead. Models trained in such simulations typically fail to achieve the same performance level in real-world environments. This begs the question: How can we get a policy that works well in a simulated environment to work well in a real-life environment?

2.5 Domain randomization

Domain randomization is the most widely adopted approach to close the simulation-reality gap (or to achieve Sim-to-Real Transfer) [20]. Instead of training agents in environments with a single fixed set of properties (or environment configuration), training takes place in environments whose properties are randomly varied [5]. By training an agent in this way, it learns to handle a large set of distinct environments, some of which may resemble the target environment. Domain ran-

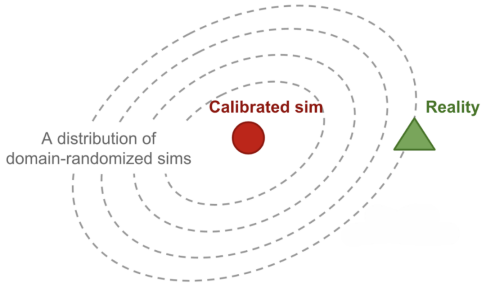


Figure 1: Visual representation of domain randomization. The calibrated sim represents the default environment. Adapted from [19].

domization can improve robustness and aid in Sim-to-Real transfer, but is not guaranteed to [20] [10].

Implementing domain randomization can be a difficult problem. The most simple (and naive) approach typically involves drawing property values from a uniform distribution bounded using knowledge of the target domain. However, using uniform distributions has drawbacks. It likely involves training models in unrealistic environments that lack any resemblance to a real-world scenario, leading to unnecessary resource usage [19]. In addition, training on overly broad randomization distributions can lead to infeasible solutions [19]. This is because excessive variation creates too many distinct environments, some of which may be unsolvable or irrelevant. As a result, the model struggles to extract consistent patterns, making it difficult to learn effective policies.

Other techniques exist: for example, Muratore et al. [12] propose an algorithm that uses (limited) data from the real-world target domain to construct a more realistic distribution to draw from during training.

Mehta et al. [10] show that training models more in difficult environments can lead to improved generalization: the model learns to handle edge cases and subsequently performs well under simpler conditions. Constraining the randomization distribution to emphasize such challenging scenarios may enhance overall robustness. This is a technique that we will employ in our training of DQN and QR-DQN.

3 Methodology

3.1 Environment

As explained in Section 2, reinforcement learning agents learn by interacting with their environment. OpenAI developed an API for single agent reinforcement learning environments in Python, called Gym (which has since been forked and is now known as Gymnasium) [15]. This framework allows developers to create environments of their choosing which, by adhering to the Gym API, provide easy training access for agents. By making an RL method implementation compatible with the Gym framework, it can be trained in any of the Gym-compatible environments.

The highway-env library provides a set of Gymnasium-compatible environments in the domain of autonomous driving [9]. Each of these environments represents a (common) autonomous driving task, among them: driving on a highway, merging on a highway, crossing a roundabout, and parking in a specific parking spot. For our purposes, we have made use of the first: the 'Highway' task. Figure 2 shows a still of the environment of this task.

The Highway environment simulates a multilane road for our agent to drive on. Our agent's (represented by the green 'car' in figure 2) objective is to get as high an overall reward as possible. Rewards are given for driving in the rightmost lane, changing lanes, and driving at high speed. In addition, it receives a penalty for colliding with other cars. In each environment step, the agent can choose its next action: slowing down, speeding up, changing lanes, or doing nothing.

The sequence of steps before the environment ends is called an episode. Episode length is determined by two things: environment duration and collisions. The `duration` environment property controls how many actions (corresponding to a single step each) an agent can take before the environment ends. Furthermore, an episode can be cut short if our agent collides with another car.

It is important to note that collisions can result in a (much) lower overall reward by curtailing the number of actions an agent can perform. However, driving recklessly (meaning: at high speed) prior to a collision can occasionally make up for an early crash. This balancing act was featured in some of our agent's behavior.

The Highway environment is highly customizable. Some of the properties that we can change are the lane count, vehicle count, vehicle density, and vehicle behavior. We can also customize the rewards the agent receives based on its actions. However, we will not engage in any reward shaping, instead relying on highway-env's defaults because our focus is on evaluating robustness, regardless of reward settings. Lastly, we can also change the duration of an environment, which will become important for our model evaluation.

As a final comment, the Highway environment includes multiple different observation types. These represent the way in which information about the state of the environment is observed at each step by the ego vehicle (the vehicle controlled by the agent). The model uses these observations to learn from. The `KinematicObservation` is a nested array containing the velocities and relative distances (to the ego vehicle) of the ego vehicle and all other vehicles. Another observation type, the `GrayscaleObservation`, is a nested array that represents a grayscale image of the scene. During some preliminary tests, we found that changing observation types had a negligible impact on performance, so we used the default: `KinematicObservation`.

3.2 Domain randomized Highway environment

For the purposes of evaluating domain randomized DQN and QR-DQN, we will look specifically at four Highway environment properties:

1. lane count
2. vehicle count
3. vehicle density
4. vehicle politeness

Vehicle density is a property that controls how vehicles are clustered around the ego vehicle. Vehicle politeness, meanwhile, is an individual vehicle's property (not an environment property) that controls whether it is triggered to change lanes when a car behind it is driving at a higher velocity. Its value can range between 0 and 1, with 1 leading to maximum willingness to change lanes for a speeding vehicle, versus minimum willingness when the value is set to 0. The lane-changing behavior is informed by the MOBIL model put forward by Kesting et al. [7].

We chose these four properties during some preliminary evaluation and testing. These were picked because their inclusion in domain randomization seemed to affect performance.

To properly evaluate the effect of domain randomization on the robustness of both DQN and QR-DQN (and answer the research questions we posed in Section 1), we focus on 3 different DR implementations. Specifically:

1. **Single property (6 - 9 vehicles per lane)**: this implementation (or configuration) corresponds to a naive approach to DR: simply take a range of possible values and draw values from it uniformly. Importantly, all other properties are kept to their default values, not randomized. Recall that the default value for vehicle count per lane is 7.
2. **Single property (8 or 9 vehicles per lane)**: using these values corresponds to training models in 'hard' environments, in the hope that they learn to generalize to easier environments. As discussed in Section 2.5, training on difficult ranges can lead to an agent learning to handle edge cases and being able to generalize well to easier conditions. Again, all other environment properties are set to their default values.
3. **Multiple properties**: in this configuration we apply domain randomization to the four properties mentioned above. In fact, we randomize according to the values in Table 1. The 'Default' column shows the default values in the Highway environment we used.

We chose these three training configurations in accordance with our research goals and questions. Comparing configurations (1) and (2) allows us to say something about the effects of naive domain randomization versus a more sophisticated approach which focuses on hard environments (as described in Section 2.5), which corresponds to our 2nd research question. In addition, comparing these configurations with configuration (3) allows us to say something about the effects of different degrees of domain randomization (single property versus multiple properties), which tackles our 1st research question. The specific values for each of these configurations were picked during some initial testing and evaluation, keeping these approaches in mind.

Another example of training under difficult conditions is the case of training on 2 and 3 lanes. The 2 lane case for the Highway is significantly harder to deal with than environments with more lanes. The ego vehicle must learn to deal with slowing down, and not changing lanes. Especially if there are two vehicles ahead, besides each other, forcing the ego vehicle to slow down. This scenario is depicted in Figure 3 below. The 1 lane case was deemed 'unsolvable', at least given the training configuration we used, and was excluded as a result.

To be able to evaluate generalization and robustness (recall our 3rd research question), we use extended randomization ranges in test environments. In other words, we test our model (in part) on out-of-distribution data. If we train our model to handle 2 or 3 lanes, will it also be able to handle 4, 5 or even 6 lanes? Similarly, in the case of vehicle count, we train using 6 to 9, 7 to 9, and 8 or 9 vehicles per lane, but *test* or *evaluate* for 5 to 10 vehicles per lane.

The politeness property is perhaps the most interesting in this regard. The 3rd training configuration used a politeness value between 0 and 0.5 (see Table 1), which means other vehicles are typically not inclined to change lanes if the ego vehicle speeds behind them. Additionally, we include politeness values between 0.5 and 1.0 in the test environments. These correspond to 'easier' environments, but if the vehicle ahead of the ego vehicle is more inclined to get out of the way, the ego vehicle could take advantage by driving at higher velocities to receive higher rewards per step.

	Default	**Training**	**Testing**
Vehicle Count (per lane)	7	7 - 9	5 - 10
Lane Count	3	2 - 3	2 - 6
Density	1.0	1 - 1.2	0.7 - 1.3
Politeness	0	0 - 0.5	0 - 1.0

Table 1: Values set for 4 properties in the default Highway environment, training environment and testing environment respectively. The training and testing environment use domain randomization using the values given in the 'Training' and 'Testing' columns (ranges are inclusive).

3.3 Training

highway-env provides two types of Highway environment. A default version and a faster variant. The faster variant (`highway-fast-v0`) includes fewer vehicles by default (20) and a duration of only 20 steps (versus 40 in the slower version). The 'Default' column in Table 1 corresponds to the default configuration of this faster environment. To speed up training, we made use of the faster variant.

We use implementations of DQN and QR-DQN from Stable Baselines 3, a Python library that provides ready-made implementations of popular RL methods [13]. The QR-DQN implementation can be found in the corresponding `s3-contrib` package.

Because our focus is on evaluating the robustness and generalizability of DQN and QR-DQN, and comparing the two, our aim is to use an (almost) identical configuration for both. As such, both models were trained with a learning rate of 5×10^{-4}, a discount factor $\gamma = 0.8$, and an ϵ-greedy exploration schedule with final $\epsilon = 0.05$. In addition, 50 quantiles were used for QR-DQN (recall from Section 2, the more quantiles we use the better the estimation of the return distribution). A complete configuration of DQN and QR-DQN can be found in Appendix A. These configurations were largely

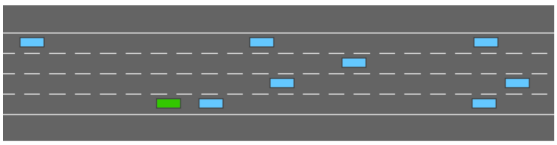


Figure 2: Still of our simulated highway environment. The green 'car' is operated by our agent.

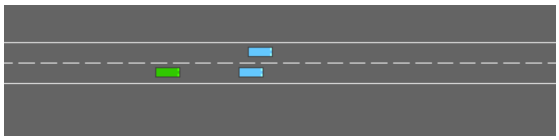


Figure 3: Still of our simulated highway with just 2 lanes. The car is forced to slow down because it cannot overtake the car ahead.

derived from the defaults used by highway-env and Stable Baselines 3, with some small adjustments made to improve training efficiency, and to bring the two configurations in line.

DQN and QR-DQN were both trained in each of the three training configurations mentioned above. In addition, both DQN and QR-DQN were trained using a baseline training configuration which did not use any domain randomization (using the values corresponding to those in the 'Default' column in Table 1). In total, then, DQN and QR-DQN were trained using 4 training configurations each, for a total of 8 configurations. Each of these training configurations was carried out using five different seeds, resulting in 40 training runs overall.

3.4 Evaluation

During testing, we tracked 3 performance metrics: episode length, episode return (or overall reward), and crash count[1].

We should note that the crash count and episode length are obviously closely related. If a car crashes just a few steps into the environment, it reduces the overall mean episode length considerably. This is also why we report the reward/step, instead of the total reward or return. By reporting the episode return instead, we would effectively be reporting the crash rate through 3 different metrics.

Models trained with training configurations (1) and (2) were tested on domain randomized environments for which the vehicle count per lane was uniformly drawn from the interval [5, 10], all other properties were set to highway-env defaults (see 'Defaults' in Table 1). Models using training configuration (3) were tested/evaluated on domain randomized environments using the property values shown in the 'Testing' column in Table 1. Each model was evaluated in 10 differently-seeded environments. We averaged the episode length and return and summed the number of crashes at the end.

Since we have trained 5 models per training configuration, and each of these models is trained on 10 different seeds (see above) and subsequently averaged, we take these 5 return means, episode length means, and crash counts and average them again. In this last step, we also compute the standard error.

To induce crashes/collisions we extended the duration during evaluation/testing to 100 steps (instead of the 30 used during training). Given more steps, the ego vehicle is more likely to crash. The differences between our models become more obvious as well, allowing for easier comparisons.

[1]We use the `tensorboard` package, part of the TensorFlow framework, for training visualization and logging https://www.tensorflow.org/tensorboard.

4 Results

4.1 Single property (vehicle count)

The results of training and testing/evaluating models in environments based on randomizing just the number of vehicles in the environments are shown in Tables 2 and 3. Specifically, the tables contain results for 3 different setups. The first two rows correspond to the training/testing of models trained without domain randomization. The 3rd and 4th rows contain the results for the models trained on environments for which the vehicle count per lane was randomly drawn from between 6 and 9 (inclusively). The last two rows correspond to the results for models trained on environments for which the vehicle count per lane was randomly drawn from the set $\{8, 9\}$.

4.2 Multiple properties

The results of the environments based on the property values of Table 1 are presented in Table 4 and 5. These results correspond to varying the values for all four properties.

Config	Reward / step	Length	Crashes
DQN	0.806 ± 0.0129	60.4 ± 9.02	6.2 ± 1.16
QR-DQN	0.782 ± 0.0086	81.1 ± 6.52	2.8 ± 0.73
DQN (6 - 9)	0.770 ± 0.0077	85.3 ± 6.07	3.6 ± 1.33
QR-DQN (6 - 9)	0.772 ± 0.0086	79.9 ± 6.69	3.4 ± 1.17
DQN (8 - 9)	0.756 ± 0.0081	79.1 ± 3.30	3.4 ± 0.51
QR-DQN (8 - 9)	0.736 ± 0.0075	90.1 ± 8.01	1.6 ± 1.12

Table 2: **Training performance (single property)**: performance metrics (mean and standard error) as measured/evaluated in environments with an equivalent configuration to that in which the models were trained. The episode length is limited to 100 steps. The crash count was compiled over 10 runs. The numbers between brackets indicate the range from which the vehicle count per lane was drawn in the configuration. The first 2 rows (and configurations) show results for models trained without domain randomization.

5 Discussion

5.1 Single property (vehicle count)

At first glance, **DQN** trained in the default environments achieves the highest reward/step of any training configuration in both the training and testing evaluation contexts. This comes at the cost of the highest crash rates in both scenarios. The lowest reward/step, meanwhile, is achieved by the **QR-DQN** model trained in environments with 8 or 9 vehicles per lane, shown in the 6th row. This setup achieves the lowest crash rate by far (and correspondingly the highest episode lengths).

Config	Reward / step	Length	Crashes
DQN	0.776 ± 0.0068	74.2 ± 10.6	4.8 ± 1.69
QR-DQN	0.754 ± 0.0068	75.9 ± 7.89	3.8 ± 1.07
DQN (6 - 9)	0.758 ± 0.0066	84.1 ± 8.04	3.4 ± 1.44
QR-DQN (6 - 9)	0.766 ± 0.0068	80.5 ± 5.92	3.4 ± 1.03
DQN (8 - 9)	0.752 ± 0.0097	78.3 ± 4.65	3.0 ± 0.71
QR-DQN (8 - 9)	0.746 ± 0.0129	91.3 ± 3.71	1.6 ± 0.68

Table 3: **Testing performance (single property)**: performance metrics (mean and standard error) as evaluated in domain randomized test environments (vehicle count per lane uniformly drawn from interval [5, 10]). See Table 2 for training results for these configurations.

Config	Reward / step	Length	Crashes
DQN	0.806 ± 0.0129	60.4 ± 9.02	6.2 ± 1.16
QR-DQN	0.782 ± 0.0086	81.1 ± 6.52	2.8 ± 0.73
DQN (DR)	0.794 ± 0.0153	56.5 ± 6.04	5.8 ± 0.92
QR-DQN (DR)	0.760 ± 0.0105	66.3 ± 9.60	4.6 ± 1.03

Table 4: **Training performance (multiple properties)**: performance metrics (mean and standard error) as measured/evaluated in environments with an equivalent configuration to that in which the models were trained. DR indicates that models were trained on domain randomized environments (according to the values in Table 1).

Furthermore, a clear difference between **DQN** and **QR-DQN** can be observed. Models trained using **DQN** suffer from a higher mean crash count than **QR-DQN** (except for the DR configuration with 6 - 9 vehicles per lane, as shown in row 3 and 4 in Table 3). This lower crash count does not come at any large expense in terms of reward/step. These results should come as no surprise. As stated in Section 2, **QR-DQN** attempts to estimate the return distribution, incorporating risk into the determination of what action the agent should take next, plausibly causing fewer collisions.

Perhaps the most interesting results of these experiments, as alluded to above, are the crash counts achieved by models trained in domain randomized environments with 8 or 9 vehicles per lane (shown in rows 5 and 6 of Table 3). The crash count achieved for **QR-DQN** using this configuration is the lowest crash count of all training configurations (including those achieved by applying DR to multiple properties). These results are in line with our expectations based on the idea that the model learns to generalize from difficult conditions to easier ones, as supported by the related work in Section 2.5. Training in more difficult environments only forces the model to learn to deal with these more effectively than if it is also trained under easier conditions. Recall that the default value for vehicle count per lane is 7. Including environments with 6 or 7 vehicles per lane (rows 3 and 4 in Tables 2 and 3) during training seems to underperform training on only 8 or 9 vehicles instead.

Overall, for the single property case, it seems that domain randomization reduces the crash count (and increases the corresponding episode length), at no or minimal cost to the mean reward/step; training in more difficult DR environments yields better results and improved generalization/robustness; and **QR-DQN** outperforms **DQN** in terms of crash count (and episode length) at minimal cost to the reward/step.

Config	Reward / step	Length	Crashes
DQN	0.756 ± 0.0075	81.8 ± 5.32	3.4 ± 0.98
QR-DQN	0.758 ± 0.0073	86.2 ± 4.96	2.4 ± 0.93
DQN (DR)	0.780 ± 0.0058	75.5 ± 11.9	3.8 ± 1.46
QR-DQN (DR)	0.748 ± 0.0086	88.0 ± 7.39	2.2 ± 0.97

Table 5: **Testing performance (multiple properties)**: performance metrics (mean and standard error) as evaluated in domain randomized test environments (according to the values in the 'Testing' column from Table 1) See Table 4 for training results for these configurations.

5.2 Multiple properties

Upon initial examination, two training configurations appear to show the most contrasting results in Tables 4 and 5. These mimic similar results in the case of single property domain randomization.

Firstly, the non-domain randomized **DQN** configuration (see 1st row of Table 4), when evaluated under its training conditions, achieves the highest reward/step. Again, this comes at a cost: It suffers from the highest crash rate (crashing in roughly 6 out of 10 runs).

Secondly, the **QR-DQN** configuration trained in a domain randomized environment achieves the lowest crash rate and associated high episode length (as seen in the last row of Table 5) of all training configurations. As we have explained above, this should come as no surprise: **QR-DQN**'s risk-accounting approach pays off. In addition, it supports our suspicion that domain randomization improves robustness (recall that we are evaluating the models on out-of-distribution data).

However, if we compare the performance of domain randomized **QR-DQN** (4th row in Table 5), with that of the non-domain randomized **QR-DQN** (2nd row in Table 5), the impact of domain randomization becomes less clear. Though the DR version outperforms the non-DR version slightly in terms of crash rate (and again, with corresponding episode length), the reward/step is slightly lower for the DR version. Overall, DR seems to have a very limited impact, if any impact at all. This is in stark contrast to the effects of domain randomization in the single property case.

The picture becomes even more complicated when we look at the results for DR and non-DR **DQN**. The 1st and 3rd rows in Table 5 show these results. While the reward/step goes up, so does the crash count (and accordingly the episode length goes down).

We can discuss the differences between **DQN** and **QR-DQN** more confidently, if we ignore domain randomization. In both the training and evaluation/testing context, there are significant differences between the crash rate: **QR-DQN** giving lower crash counts compared to **DQN**. In addition, the reward/step seems to be slightly higher for **DQN**, compared to **QR-DQN**. This is most clearly presented in the 1st and 2nd row of Table 4, and the 3rd and 4th rows of Table 5. These results align with our observations discussed in the previous subsection on single property DR.

Lastly, like in the case of the vehicle count DR, it seems that the domain randomized test environment is significantly easier than any of the other environments. Non-DR **DQN** (1st row in Table 4 and 5) achieves a lower reward/step, but considerably lower crash count (3.4 instead of 6.2) during evaluation. The same holds for the other training configurations. This is easy to explain for the DR models: we specifically trained on hard distributions to evaluate generalization and robustness. For the non-DR models, it seems to imply that the default environment settings of the highway-env Highway environment (as shown in Table 1) represent a relatively difficult environment. In fact, we know specifically that the vehicle density values below 1.0 result in easier conditions for the ego vehicle (column 'Testing' in Table 1), because it results in fewer vehicles close-by to potentially collide with. In addition, the default value of 0.0 for vehicle politeness represents the 'hardest' condition for the ego vehicle, as other vehicles are least inclined to move out of the way, forcing the ego vehicle to change lanes and potentially collide.

Overall, the results for the case of applying DR over multiple properties gives similar results to those for the single property case. Firstly, in both cases, the non-DR **DQN** configuration achieved the highest performance at the expense of the highest crash count, with the DR **QR-DQN** configuration achieving the lowest crash count. Secondly, in both cases, **QR-DQN** got lower crash counts than **DQN**, regardless of whether DR was used or not. Third, in both instances, the environments in which the models were evaluated were comparatively 'easier' compared to the training environments. In contrast, the impact of domain randomization was considerably less clear in the case of multiple property DR, compared to that of the single property case, where domain randomization ostensibly reduced crash count and improved robustness.

6 Conclusions and Future Work

6.1 Conclusions

This study investigates how domain randomization (DR) affects the robustness of agents trained using DQN and QR-DQN. We assess robustness through return, episode length, and crash rate (or count) in both standard training and domain randomized testing environments.

We will recall our research question here: *How does domain randomization affect the robustness of DQN and QR-DQN?* This question consists of 3 subquestions:

1. **RQ1**: How does the robustness of DQN compare to that of QR-DQN under varying degrees of DR?
2. **RQ2**: How is the robustness of DQN (and QR-DQN) affected by different types of DR?
3. **RQ3**: How well do DQN and QR-DQN, trained with domain randomization, generalize to unseen environments?

To answer **RQ1**, the differences between the varying degrees of DR are quite clear. When DR is implemented as varying a single environment property (vehicle count per lane), DR has a clear positive impact on crash count (and episode length) with minimal to no cost in terms of reward/step, implying improved robustness and generalization. However, DR implemented as varying four properties in the environment (lane count, vehicle count, vehicle density, and vehicle politeness), has a more limited or even adverse impact on these metrics.

In addition, QR-DQN consistently outperforms DQN in terms of crash count, both in DR and non-DR settings. However, DQN typically achieves marginally higher reward/step, which suggests a trade-off between risk aversion and return.

To answer **RQ2**, in Section 2.5 we discuss the difficulty of implementing domain randomization, as shown in related literature. Not all approaches improve robustness and/or generalization. In our research, we focused specifically on two common approaches, or types of DR: (1) a simple approach: uniformly draw values from a naive range of values and (2) a more sophisticated approach: draw property values from a range such that the resulting environments feature more difficult conditions. Approach (2) relies on the belief that training in such hard environments leads to better generalization (and robustness). To compare these approaches, we focus on a single property (vehicle count per lane, which has a default value of 7), using a 'naive' range of 6 - 9 vehicles per lane to mimic approach (1), and using 8 or 9 vehicles per lane for approach (2). Although both approaches show improved robustness and generalization, approach (2) outperforms approach (1). Specifically, QR-DQN trained using approach (2) achieves the lowest crash count and comparable reward/step.

Lastly, to answer **RQ3**, all models perform better in the DR test environments than in their respective training environments when varying vehicle count only, suggesting that DR improves robustness and generalization to unseen environments and/or conditions. In particular, the domain randomized QR-DQN model achieves the lowest crash rate and longest episode duration of all in the test environment (with out-of-distribution data), suggesting a capacity for sim-to-real transfer. However, as we alluded to above, the impact of DR in the case of applying DR to four environment properties is limited, or even adverse.

In conclusion, the findings suggest that QR-DQN may offer greater robustness than DQN, particularly in terms of crash counts, although DQN tends to achieve slightly higher rewards per step. Domain randomization appears to improve robustness when applied selectively. For example, by varying only the vehicle count. More extensive randomization across multiple environment properties resulted in limited or even adverse effects, suggesting the need for a more sophisticated approach. Furthermore, the results indicate that emphasizing challenging scenarios could lead to better generalization compared to a naive uniform approach, in accordance with earlier work on achieving sim-to-real transfer through domain randomization. Lastly, the improved performance of some DR-trained models in unseen environments (particularly QR-DQN) suggests enhanced sim-to-real transfer in addition to improved robustness, though further investigation would be valuable.

6.2 Future work

In Section 2.5 (Domain Randomization), we have tried to emphasize the difficulty of using domain randomization effectively. The application of domain randomization to achieve higher performance, robustness, and sim-to-real transfer is an active research area. Succeeding typically means finding the right distributions to randomize on, often manually.

In our research, we have used this manual approach to determine potentially effective distributions to randomize on, performing some preliminary testing to see the effects on performance and robustness. Any of our results could be improved upon by applying more competent search efforts to find more effective (or impactful) distributions.

Not just the choice of distribution is relevant in this regard. The properties of the environments to randomize presented an additional choice in our investigation. To implement DR using multiple properties, we decided on the four properties given in Table 1 because preliminary testing showed that performance was affected after changing their values, but other properties could have had more of an impact (or even an adverse one). More importantly, to focus on the problem of sim-to-real transfer, other properties *should* perhaps have been included. If we want to apply our research to the problem of sim-to-real transfer, we typically have no real choice in picking properties to vary. We have to look at the real environment (in the context of autonomous driving: real highway environments) and deduce which properties to vary based on which properties *actually* vary. We believe that a closer look at these properties and distributions presents an opportunity for future research.

Lastly, QR-DQN is not the only learning method built on DQN. Other variations of DQN are in common use today. Among them, Double DQN [16] and Dueling DQN [18]. The questions surrounding the impact of domain randomization on *their* robustness and the sim-to-reality gap have (as far as we know) not been addressed yet.

7 Responsible Research

7.1 Ethical concerns

Our research has been conducted with the ethical aspects of research in mind. Our use of a simulated training environment and corresponding data precludes most ethical concerns about the misuse and improper handling of data, whether personal or dangerous.

Our use of a simulated autonomous driving environment to evaluate the robustness of DQN and QR-DQN could lead to irresponsible use of our conclusions. For example, someone could apply conclusions from our research to the real-world domain of autonomous driving, where lives are obviously at stake. The primitive nature of the highway-env simulator should hopefully dissuade anyone from generalizing our results, as does this section.

7.2 Reproducibility

To aid in the reproducibility of our results, we have taken the utmost care to describe our entire methodology in Section 3, including any relevant concepts, setup instructions, and other variables. In addition, we have included the configuration of the hyperparameters and other relevant configuration parameters for DQN and QR-DQN in Appendix A. Lastly, a complete overview of our training and test setup and any remaining code is available on GitHub[2]. Using the seeds included in Appendix B and the code mentioned above, anyone should be able to reproduce our results reliably.

7.3 Use of LLMs

In the preparation of this research paper, a large language model (ChatGPT) was used to aid in various aspects of the writing and formatting process. Specifically, ChatGPT was used to generate well-structured tables for some of the results. Additionally, the model assisted in generating proper LaTex code to align figures and insert them accurately within the document. Furthermore, ChatGPT was a valuable tool in ensuring adherence to the stylistic norms and conventions of computer science research, helping to produce a clear, consistent, and polished final manuscript.

A Hyperparameters

A.1 DQN

```
DQN(
    "MlpPolicy",
    env,
    policy_kwargs=dict(net_arch=[256, 256]),
    learning_rate=5e-4,
    buffer_size=1_000_000,
    learning_starts=1000,
    batch_size=64,
    gamma=0.8,
    train_freq=1,
    gradient_steps=1,
    target_update_interval=1000,
    verbose=1,
    seed=seed,
    ...
)
```

A.2 QR-DQN

```
policy_kwargs = dict(
    n_quantiles=50, net_arch=[256, 256]
)

QRDQN(
    "MlpPolicy",
    env,
    policy_kwargs=policy_kwargs,
    learning_rate=5e-4,
    buffer_size=1_000_000,
    learning_starts=1000,
    batch_size=64,
    gamma=0.8,
    train_freq=1,
    gradient_steps=1,
    target_update_interval=1000,
    verbose=1,
```

[2]https://github.com/yzwetsloot/research-project

```
    seed=seed,
    # default DQN configuration
    exploration_fraction=0.1,
    exploration_final_eps=0.05,
    ...
)
```

B Seeds

Recall from Section 3 (Methodology) that we trained 5 models per training configuration. Each of these 5 models was trained using a different seed. In addition, each of the 5 models per training configuration was tested on 10 differently-seeded environments. Below we present all of these seeds.

To speed up learning, we made use of Stable Baselines3's [13] `SubprocVecEnv`. This feature introduces the notion of vectorized environments across multiple CPU cores. We used 6 cores during training. To enable deterministic learning, we set 6 seeds, one for each initial environment (one environment per core). For a better overview of how these seeds were used, see the GitHub repository[3].

- Model seeds during training (5): 1, 11, 21, 31, 41.
- Environment seeds during training (6): 1, 2, 3, 4, 5, 6.
- Environment seeds during testing (10): 1024, 1025, 1026, 1027, 1028, 1029, 1030, 1031, 1032, 1033.

C Training plots

See Figures 4, 5, 6 and 7 below for plots of return (or overall reward) over 100K training steps, for different training configurations. Recall that each training configuration was trained on 5 different seeds, explaining the 5 distinct plots per subfigure.

References

[1] Oladayo S. Ajani, Sung-ho Hur, and Rammohan Mallipeddi. Evaluating Domain Randomization in Deep Reinforcement Learning Locomotion Tasks. *Mathematics*, 11(23):4744, January 2023. Number: 23 Publisher: Multidisciplinary Digital Publishing Institute.

[2] Hansenclever F. Bassani, Renie A. Delgado, Jose Nilton de O. Lima Junior, Heitor R. Medeiros, Pedro H. M. Braga, and Alain Tapp. Learning to play soccer by reinforcement and applying sim-to-real to compete in the real world, 2020.

[3] Ege Bayram. Evaluating robustness of deep reinforcement learning for autonomous driving: Effects of domain randomization on training and robustness, 2023.

[4] Bahador Beigomi and Zheng H. Zhu. Towards real-world efficiency: Domain randomization in reinforcement learning for pre-capture of free-floating moving targets by autonomous robots, 2024.

[5] Xiaoyu Chen, Jiachen Hu, Chi Jin, Lihong Li, and Liwei Wang. Understanding domain randomization for sim-to-real transfer, 2022.

[6] Will Dabney, Mark Rowland, Marc G. Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression, 2017.

[7] Arne Kesting, Martin Treiber, and Dirk Helbing. General lane-changing model mobil for car-following models. *Transportation Research Record*, 1999(1):86–94, 2007.

[8] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey, 2021.

[9] Edouard Leurent. An environment for autonomous driving decision-making. https://github.com/eleurent/highway-env, 2018.

[10] Bhairav Mehta, Manfred Diaz, Florian Golemo, Christopher J. Pal, and Liam Paull. Active domain randomization, 2019.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[12] Fabio Muratore, Christian Eilers, Michael Gienger, and Jan Peters. Data-efficient domain randomization with bayesian optimization. *IEEE Robotics and Automation Letters*, 6(2):911–918, April 2021.

[13] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[14] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2 edition, 2018.

[15] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.

[16] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

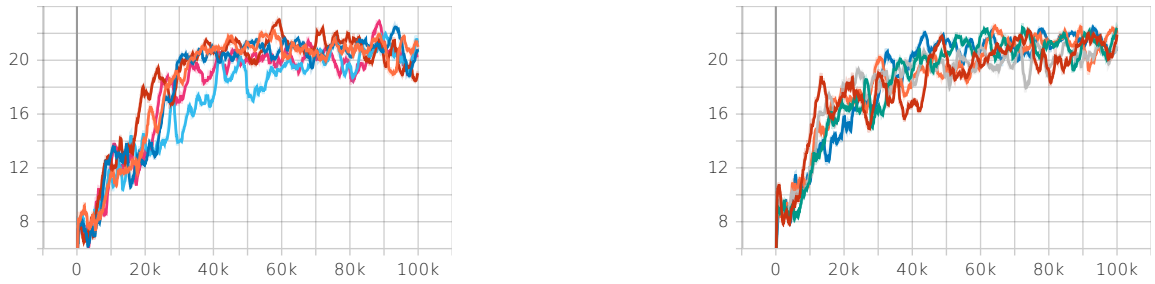
[17] Quan Vuong, Sharad Vikram, Hao Su, Sicun Gao, and Henrik I. Christensen. How to pick the domain randomization parameters for sim-to-real transfer of reinforcement learning policies?, 2019.

[18] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2016.

[19] Lilian Weng. Domain randomization for sim2real transfer. *lilianweng.github.io*, 2019.

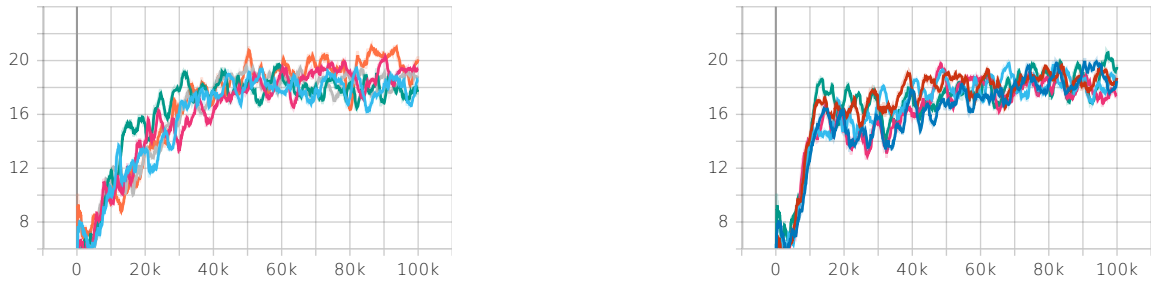
[20] Wenshuai Zhao, Jorge Peña Queralta, and Tomi Westerlund. Sim-to-real transfer in deep reinforcement learning for robotics: A survey. *arXiv preprint arXiv:2009.13303*, 2020.

[3] https://github.com/yzwetsloot/research-project



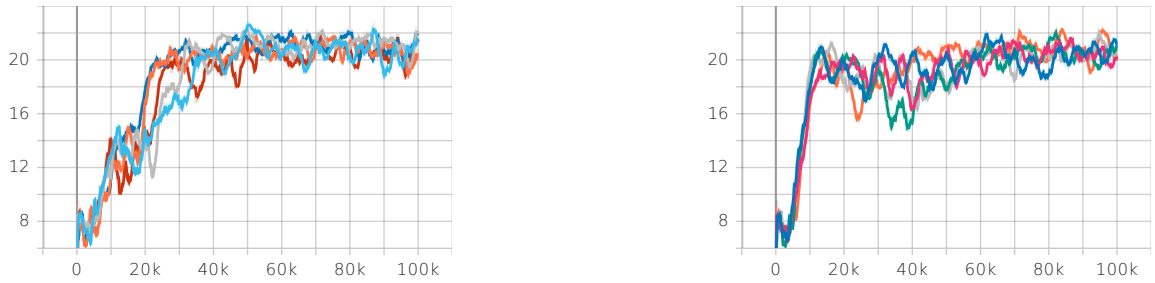
(a) DQN (b) QR-DQN

Figure 4: Return for (a) DQN and (b) QR-DQN plotted over 100K timesteps, without domain randomization (and using 5 distinct seeds).



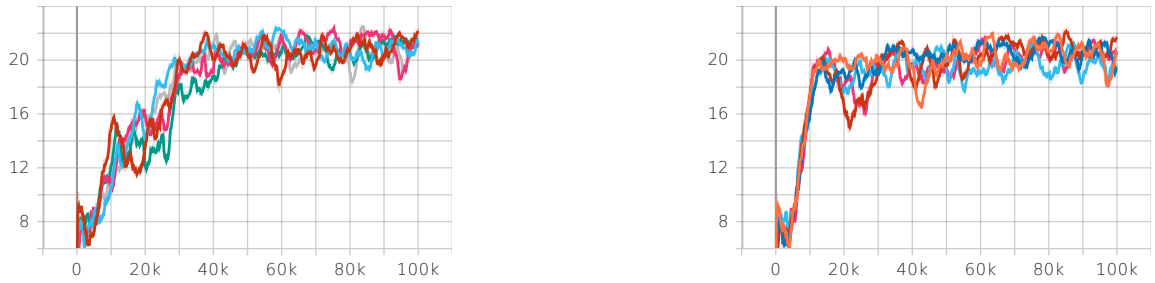
(a) DQN (b) QR-DQN

Figure 5: Return plotted for (a) DQN and (b) QR-DQN over 100K timesteps and 5 distinct seeds, trained in domain randomized environments according to Table 1.



(a) DQN (b) QR-DQN

Figure 6: Return plotted for (a) DQN and (b) QR-DQN over 100K timesteps, and 5 distinct seeds. Models were trained in environments for which the vehicle count per lane was drawn uniformly from the interval [6, 9].



(a) DQN (b) QR-DQN

Figure 7: Return plotted for (a) DQN and (b) QR-DQN over 100K timesteps, and 5 distinct seeds. Models were trained in environments for which the vehicle count per lane was drawn uniformly from the interval [8, 9].