Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Domain Randomization for Deep Reinforcement Learning in Self-Driving Environments

Master's Thesis

Author:
András Béres

Supervisors:
Dr. Bálint Pál Gyires-Tóth
Róbert Moni

December, 2021

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Béres András*, szigorló hallgató kijelentem, hogy ezt a dolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2021. december 19.

Béres András
hallgató

Kivonat

A megerősítéses tanulás és a mély tanulás két olyan tudományterület, melyek együttesen az önvezető járművek kihívásának megoldására kínálnak lehetőséget, azonban van két megoldásra váró probléma a valós alkalmazásuk útjában, ezekkel foglalkozom dolgozatomban.

Az egyik ilyen probléma a megerősítéses tanulás adatéhsége, aminek egy lehetséges megoldása szimulátorok használata, és az ezekben történő tanítás utáni az alkalmazás a valóságban. A másik említett probléma pedig az, hogy ezek az algoritmusok általában nem robusztusak környezetük változásaival szemben, ami a valóságban történő alkalmazásukat megnehezíti.

Dolgozatomban a véletlenszerű kinézetű és dinamikájú környezetek módszereit vizsgálom, ezzel párhuzamosan reprezentáció tanulást alkalmazva felügyelt vagy felügyelet nélküli módon, hogy a bemeneti képeket tömörebben reprezentálva a megerősítéses tanulást hatékonyabbá tegyem. Az ilyen módon feldolgozott képeket használom aztán fel megerősítéses tanulásra, hogy ezáltal az ágenseket robosztusabbá tegyem. Megoldásom célja hogy a valóságban történő alkalmazást segítsem, ezzel párhuzamosan a megerősítéses tanulást megkönnyítve.

Munkám célja a véletlenszerű környezetek módszereinek implementációja és összehasonlítása a szakirodalmat követve. Dolgozatomban öt véletlenszerű kinézeti és egy véletlenszerű dinamikájú megoldást vizsgálok meg, azokat rugalmas moduláris megoldásban implementálva.

Megvizsgálom a sávkövetés problémáját szabályozástechnikai szempontból is, és megvalósítok egy új szimulációs környezetet is, ami pusztán a szabályozástechnikai feladatot modellezi, elválasztva azt a képfeldolgozástól.

A megvalósított algoritmusokat nem csak véletlenszerű és nem véletlenszerű környezetekben, hanem a valóságban is kiértékelem, és bemutatok gyakorlati megoldásokat a sikeres valós alkalmazás elősegítésére.

A bemutatott módszert a Duckietown önvezető környezetben alkalmazom, ahol a sávkövetés a megoldandó feladat egy differenciális meghajtású jármű irányításával, mindössze egy kamera képe alapján. Az algoritmusokat a PyTorch nyílt forráskódú mélytanuló programkönyvtár segítségével valósítom meg.

Abstract

Reinforcement Learning and Deep Learning are paradigms that offer the possibility of solving the self-driving problem, however there are two important hurdles in the way that I tackle in this thesis.

The first is that these algorithms need large amounts of data, which can be mitigated by the usage of a simulator for training, and then transferring to the real world. The second issue is that algorithms are usually not robust to changes in their environment, which makes sim-to-real transfer difficult.

In this thesis I apply Visual Domain Randomization and Dynamics Randomization simultaneously with either Supervised or Self-Supervised State Representation Learning to compress the input images to a useful lower dimensional representation. These are then used as inputs to the reinforcement learning agent to make it more robust both to visual and dynamical changes in their environment. This makes sim-to-real transfer easier while also making the reinforcement learning task lighter.

The goal of this work is to implement and benchmark multiple domain randomization methods from the literature and to compare their effectiveness. In this work I investigate five main types of visual domain randomization and a single dynamics randomization method, implementing them using a flexible modular architecture.

I also investigate the task of lane following from a control systems perspective, implementing a novel simulation environment which separates the control task from the perception task.

The proposed techniques are not only evaluated in randomized and non-randomized simulated environments, but in reality as well, with practical considerations helping with sim-to-real transfer.

The method is evaluated in the Duckietown self-driving car environment, where the task is to follow lanes by controlling a differential drive vehicle based only on camera images. The algorithms are implemented using the PyTorch open-source deep learning library.

Chapter 1

Introduction

Achievements of deep learning methods are followed with greater and greater popular interest nowadays as systems powered with artificial intelligence (AI) outperform humans in a wider and wider variety of tasks. Learning agents show the promise of systems that learn from their errors and improve day by day.

Recently the topic of self-driving cars has received great attention from academia and the public as well. While advances in the field of Deep Learning provide us tools for processing vast amounts of sensor data, Reinforcement Learning methods promise us the ability to take the right actions in complex interactive environments. Using these tools could be one way to solve the self-driving task, however some problems make the real world application of these techniques difficult.

One such issue is that these systems are generally not robust enough to changes in their environment, and can produce unexpected behaviours as a response to them.

Another problem is the sample inefficiency of Reinforcement Learning. These agents usually require large amounts of data to learn tasks what humans can master from a few examples using the prior knowledge.

Since the only source of information for learning is a reward signal, it can take several exploratory steps to find the right parameters for the network. A proposed solution is to use Unsupervised Learning on the input data, which means learning from the raw data without any labels, generally by solving tasks whose solutions are inherently present in it.

AI agents collecting large amounts of experience while interacting with the real world would usually be too expensive and sometimes even dangerous, so it is beneficial to train the agents in a simulator, and then transfer them to the real world. However, our simulators can only be imperfect models of reality, so the performance of agents is usually reduced after the transfer. This problem is called the Sim-to-Real Gap. A proposed solution is Domain Randomization, which perturbs certain parts of the simulation during training, therefore forcing the agent to be robust against changes in its environment. This increases

the probability of a successful transfer, usually at the cost of requiring even more training samples.

Overcoming both problems would be an important step towards the application of deep learning systems on a broader scale.

In this work I explore ways to improve the robustness of reinforcement learning agents and to make training them more sample-efficient simultaneously, to prepare them for application in the real world. I evaluate and test multiple domain randomization algorithms and also summarize a novel one.

The rest of this thesis is organized as follows: in Chapter 2 I present the relevant background from the literature, categorizing and summarizing methods for domain randomization, and also giving an overview of autoencoding-based algorithms. Then in Chapter 3 I present the modular system I designed for tackling the task, and introduce the types of algorithms that I implemented. Following that, I describe in detail how I implemented my solutions in Chapter 4, showing how the system was built. Following that, I present the used evaluation procedures and issues I found during real testing in Chapter 5 and also detail the steps I took to tackle them. After that, in Chapter 6 I present my experimental results in both the simulator and the real world, across algorithms, and compare the findings. I close my work discussing the achieved results in Chapter 7, proposing possible improvements and determining future research directions.

Chapter 2

Theoretical Background

2.1 Deep Learning

Deep Learning [1] is a subfield of Machine Learning and the broader field of Artificial Intelligence, in which we train artificial neural networks [2][3] for solving tasks. Recently, it has successfully been applied to Computer Vision [4][5], Natural Language Processing [6], and Audio Processing [7].

Artificial **neural networks** are modular computation graphs, whose outputs are partially differentiable with respect to their parameters (weights), and their layers contain nonlinear activation functions. If the graph is acyclic, i.e. it does not contain any directed circles, it is called a feedforward neural network, otherwise it is a recurrent [8][9] one.

By using several input-output data examples, the weights can be adjusted such that the network will produce outputs that are on average closer to our desired outputs, with the aim that it will be able to generalize to unseen data samples as well. That way the networks can be used as general nonlinear function estimators, and are useful tools in settings where we have enough data samples to train a network with sufficient capacity to solve the task at hand.

During training, we iterate over the items of the training dataset, and based on the discrepancy between the network outputs and the desired outputs, we calculate a **loss** value using a loss function. Since the partial derivatives of the output with respect to the weights can be calculated, the parameters of the network are adjusted in the negative gradient direction in order to decrease the average loss.

Partial derivatives can be computed using the **backpropagation** of the error [10][11], by applying the chain rule from calculus from the output of the network towards the input. The weights are changed in proportion to their partial derivatives, i.e. with the sensitivity of the loss to them. This algorithm is called **gradient descent** [12], and the proportionality constant is called **learning rate**.

As calculating the partial derivatives of the loss over the whole training dataset before every optimization step is infeasible and inefficient, we generally take a subset of the dataset, called a **minibatch** (sometimes simply called batch, which used to refer to the whole dataset), and use it for an optimization step. This makes the optimization stochastic, because the mean loss over the dataset is only estimated using its random subset, therefore we call the algorithm stochastic gradient descent (SGD) [13]. Nowadays we generally use improved versions of this optimization procedure, such as the Adam algorithm [14], which utilizes estimates of first- and second-order moments of the gradients using exponential moving averages to help optimization.

The *deep* adjective in the field's name refers to the fact that the networks we tend to use are deep in the graph-theoretic sense, i.e. the amount of layers between the input and the output is numerous, in practice it spans a range from a few layers to even a thousand [15]. The main reason being that for the representation ability of the networks, depth is more useful than width [16].

From a theoretic point of view, the universal approximation theorem [17] states that a network with two layers and an appropriate activation function can approximate any Borel measurable function with a desired nonzero error, given enough width. This function space is broad enough, since any continuous function on a closed and bounded subset of $\mathbb{R}^n$ is Borel measurable [18]. It has been shown that for a shallow network to have the same representation ability as a deeper network, it has to be exponentially wider [16][19].

This effect can also be seen in practice, deeper networks are generally more powerful, but also harder to optimize [18][20]. An intuitive example for the advantages of depth is computer vision, where the first layers of a Convolutional Neural Network (CNN) [21] learn low-level features such as edges or textures, later ones recognize simple structures based on those, while the final ones can perceive complex structures [22].

A great depth can also make the training process quite difficult by the problem of vanishing or exploding gradients [20], so depth cannot be chosen to be arbitrarily large. The maximum depth achievable in practice, however, grows year by year, thanks to architectural improvements [15].

The field of Machine Learning has three main subfields: **Supervised Learning**, **Unsupervised Learning** and **Reinforcement Learning**, and Deep Neural Networks can be applied in each of these.

In supervised learning, for each data sample we have a corresponding label, and for each input sample the network is trained to predict the its label. This can be optimized in a straightforward way, similar to what has been described above. A drawback of these family of methods is that they require the labeling of the datasets, which usually requires human effort, and is also sometimes unfeasible if the correct labels are not known. While the cost of labeling scales linearly with dataset size (labeling each example takes a constant time), model performance only scales sublinearly [23] with it. This means that labeling more

and more samples becomes less and less cost-efficient, while gathering unlabeled data is simpler and cheaper.

Another subfield is unsupervised learning, which means learning from the raw input data without any labels. This can be done by solving tasks whose solutions are inherently present in the data, which is called self-supervised learning (though sometimes the names of these two concepts are used interchangeably in the literature). Examples of such tasks are corruption-restoration processes, where either the class of corruption has to be predicted or the original input has to be restored, generative modeling, where the intermediate features of a similar-but-novel data sample generator network can be used, and contrastive learning [24], where augmented versions of the same data sample have to be distinguished from augmented versions of other samples.

The last one is reinforcement learning. In this setting an agent makes decisions in an interactive environment, and it receives a scalar reward in every timestep based on its performance. Here the task is to optimize the behaviour in this time-dependent, stateful, non-differentiable environment, to receive as much reward as possible. This topic will be further discussed in Section 2.2.

2.1.1 Representation Learning

Representation learning [25] is the task of learning to extract features from the input data, or learning to transform it in a way, which makes it more useful for downstream tasks such as classification or prediction.

Since we generally have large quantities of training data in computer vision (images), and the classical image processing algorithms have only been useful on a narrow range of tasks, it provided a fertile ground for data-driven algorithms, such as neural networks and deep learning. With the help of convolutional neural networks [21], a family of networks specializing in image processing, remarkable achievements have been made in image representation learning tasks such as image classification [4], image segmentation [26], and object detection [27].

While deep learning is not the only way to learn representations, in this work I employ deep learning and convolutional neural networks to learn image representations it two ways: I use regression from frames to physical track parameters as a supervised representation learning method, and I also use autoencoders discussed in the next subsection as an unsupervised one. The downstream task in my case is the control of a self-driving vehicle in simulated and real environments.

2.1.2 Autoencoders

An autoencoder [28][18] (AE, classically called autoassociator) is a neural network whose task is to represent the identity function, i.e. to map its input to its output. Since this task would be trivial, it always contains a bottleneck layer, which in some way forces it

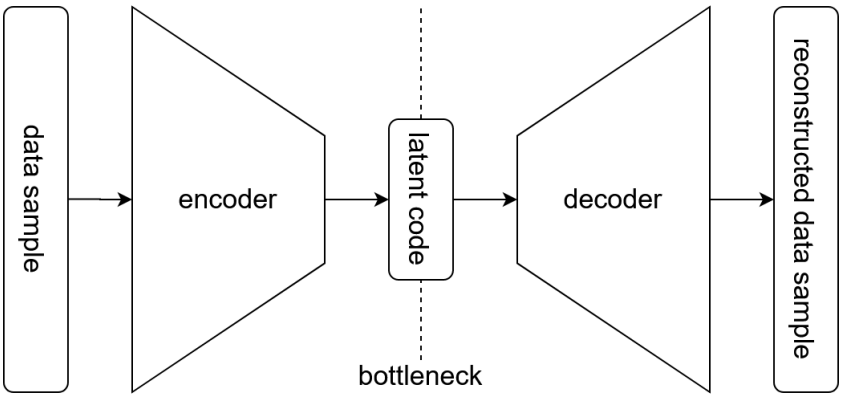

Figure 1: Architecture of an autoencoder

to compress its input, making the task non-trivial. We can then map our data to these learned compressed representations and use them in other tasks.

As training an autoencoder does not require any labels, only the data samples, autoencoders are an unsupervised (and also self-supervised) method for representation learning.

We call the part of the network before the bottleneck layer **encoder**, and the part after it the **decoder**. The output of the bottleneck layer is called the **latent code**, and usually it is considered as a learned representation of the data. The structure of an autoencoder is shown on Figure 1.

$$\begin{aligned} z &= encoder(x) \\ \hat{x} &= decoder(z) \\ \hat{x} &= decoder(encoder(x)) \\ L_{AE} &= d(\hat{x}, x) \end{aligned} \tag{2.1}$$

$$\begin{aligned} &p_{data}(X) \\ &p_{encoder}(Z|X=x) \\ &p_{decoder}(\hat{X}|Z=z) \\ &p_{autoencoder}(\hat{X}|X=x) \\ &L_{AE} = -\log p_{autoencoder}(\hat{X}=x|X=x) \end{aligned} \tag{2.2}$$

In Equation 2.1 we can see the components of a deterministic autoencoder, with x being the input data sample, h the latent code, $\hat{x}$ the reconstructed data sample on the output, L_{AE} the value of the loss, and $d(x, \hat{x})$ is a distance metric, e.g. L1- or L2-norm of their difference. As we will see in Sections 2.1.3 and 2.1.4, one can generalize the concept to probabilistic mappings as well, with the notations shown in Equation 2.2, with the upper

case letters being random variables and the lower case letters being their corresponding samples. In these settings the loss is usually the negative log-likelihood of the input sample.

To simplify notation, here and in further equations I will only write the value of the loss for a single data sample. One has to keep in mind that the loss is actually the expectation over all loss values for all examples in the training dataset, and is estimated as a mean over the data samples in a minibatch during training in the batched setting.

Bottlenecks are generally realized in two ways. We can make the latent code lower dimensional than the input, in which case we talk about **undercomplete autoencoders** [28], or we can regularize some part of the network in the case of **regularized autoencoders**.

By regularizing the latent code we can force sparsity in the latent codes (sparse autoencoders) [29], small gradients of latent representations at the data points (contractive autoencoders) [30], or limited information content and a simple latent manifold structure (variational autoencoders) [31][32]. By corrupting the input data and trying to reconstruct the uncorrupted version (denoising autoencoders) [33], we can also force the network to implicitly learn the structure of the data distribution [34][18].

Undercomplete autoencoders, i.e. when the latent code is lower dimensional than the output, are so common, that when someone uses simply the term "autoencoder" they usually refer to undercomplete autoencoders.

Since the dimensionality of latent codes and corruption of the input is an architectural choice, and the regularization of latent codes is usually implemented with adding penalty terms to the loss function, most these autoencoding methods can be applied simultaneously as well.

2.1.3 Denoising Autoencoders

Denoising autoencoders (DAE) [33] use a usually stochastic corruption process, which introduces noise to the input data. Then the task of the network is to reconstruct the original version of the input data sample, as can be seen on Figure 2.

$$
\begin{aligned}
&p_{data}(X) \\
&p_{corruption}(\tilde{X}|X = x) \\
&p_{encoder}(Z|\tilde{X} = \tilde{x}) \\
&p_{decoder}(\hat{X}|Z = z) \\
&p_{autoencoder}(\hat{X}|\tilde{X} = \tilde{x}) \\
&L_{DAE} = -\log p_{autoencoder}(\hat{X} = x|\tilde{X} = \tilde{x})
\end{aligned}
\tag{2.3}
$$

In Equation 2.3 we can see the components and the loss of a denoising autoencoder, with the new notations $\tilde{x}$ being the corrupted data sample, and $p_{corruption}()$ being a stochastic

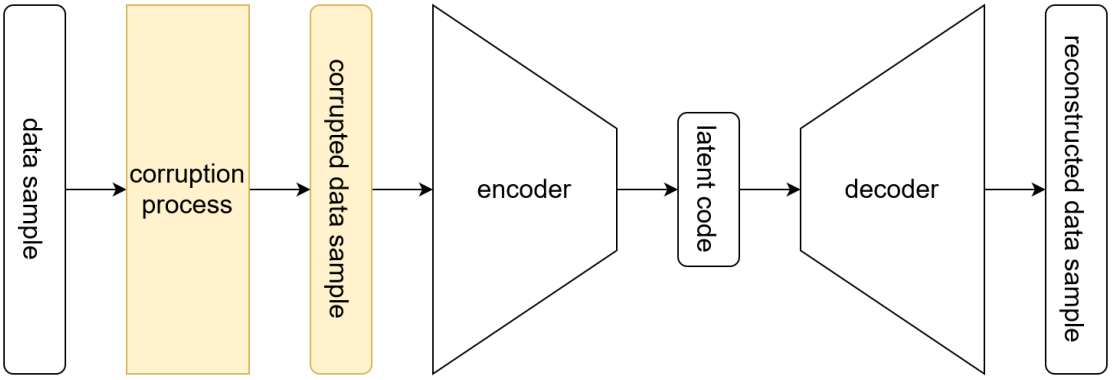

Figure 2: Architecture of a denoising autoencoder (differences with a regular autoencoder indicated in yellow)

corruption process. The loss is the negative log-likelihood of the input sample, given the corrupted sample.

The theoretically most understood corruption process is the additive uncorrelated Gaussian noise, however in practice salt-and-pepper noise [35], uncorrelated masking noise [35], and for images correlated masking noise (occlusion) [36] has been used as well. In this work I will show that using different renderings of the same image with different textures is also an effective corruption process for images when using a simulator.

It has also been shown that solving the denoising task implicitly forces the network to learn to represent the probability distribution of the data [34][18]. This property has been utilized in the recent denoising diffusion models [37], to use overcomplete denoising autoencoders for generating images by denoising an image of complete Gaussian noise.

Recent trends show an increasing popularity of denoising autoencoders. Besides generating images they have also been used for natural language processing in BERT [38], which was trained to predict masked tokens (words) from its input text. They have also been successfully applied for self-supervised image representation learning [39] by training a denoising autoencoder to predict masked image-patches of its input.

2.1.4 Variational Autoencoders

A variational autoencoder (VAE) [31][32] is a type of autoencoder that employs a specific latent code regularization scheme, which makes it capable to be used as a generative model. It has other interesting properties too, which we will discuss in this section.

As it is shown on Figure 3, the encoders of variational autoencoders produce a parametrization of a latent code distribution, from which the input of the decoder is sampled. That means, that the model is stochastic and the latent code samples will be noisy, which makes it necessary for the decoder to be robust to latent code perturbations.

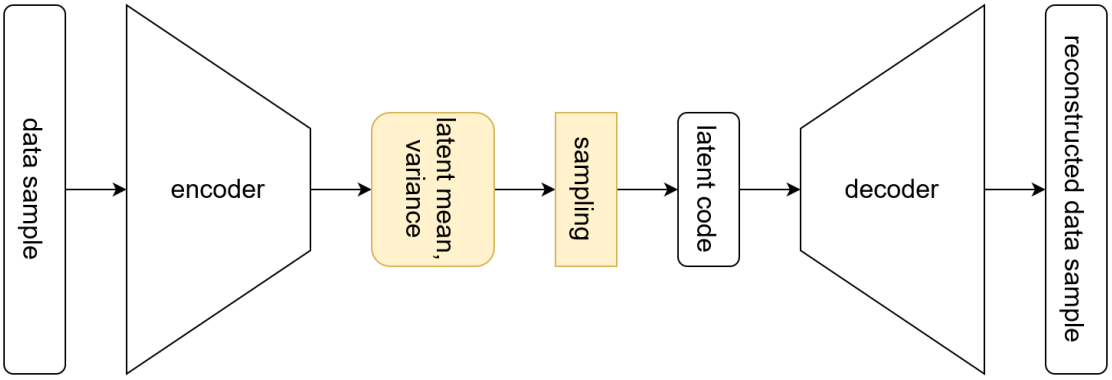

Figure 3: Architecture of a variational autoencoder (differences with a regular autoencoder indicated in yellow)

$$
\begin{aligned}
&p_{latent}(Z) \\
&p_{data}(X) \\
&p_{encoder}(Z|X=x) \\
&p_{decoder}(\hat{X}|Z=z) \\
&p_{autoencoder}(\hat{X}|X=x) \\
&L_{VAE} = L_{AE} + D_{KL}(p_{encoder}(Z|X=x)||p_{latent}(Z))
\end{aligned}
\tag{2.4}
$$

In Equation 2.4 we can see the components and the loss of a variational autoencoder, with the new notations p_{latent} being the prior latent distribution, and $D_{KL}()$ being the KL-divergence of two distributions. The loss is the negative log-likelihood of the input sample plus the KL-divergence of the posterior latent distribution from the prior one.

Generally the prior $p_{latent}(Z)$ is chosen to be a multivariate Gaussian distribution with an identity covariance matrix.

Since the naive implementation of the sampling operation from the posterior latent distribution $p_{encoder}(Z|X=x)$ would not be differentiable, we have to use a method called the "reparametrization trick". First we sample a noise vector from a unit Gaussian, then we multiply that with the standard deviation of the posterior, and add the mean. That way the computations based on our parameters are completely deterministic, beacuse the noise vector is treated as a separate input, and therefore gradients can be propagated through the operation.

Variational autoencoders have proved to be powerful generative models, and they and their extensions have been successfully applied to natural image generation [40], high-resolution human portrait generation [41][42], video generation [43] and even language modeling [44]. Its training procedure is generally considered more stable than the other popular method, Generative Adversarial Networks (GAN) [5], however it is also known to produce blurrier samples [45].

A possible explanation for the blurriness of the generated images is the choice of $p_{decoder}(\hat{X}|Z)$ since this distribution is not known, it is generally assumed to be a multivariate normal distribution with a constant valued diagonal covariance matrix. As generally pixels that are close to each other are correlated, this choice is actually a limiting factor.

Concurrent work [46] has shown that the variance values can be calibrated based on the training dataset either by estimating variance on the fly during training from each minibatch, or by iterating over the dataset before training. That way no hyperparameter is needed to set the strength of the KL-divergence term in the loss function. In my work I used the latter method for estimating the variance for every pixel of the output image separately, so that tuning the strength of the KL-divergence term [47] was not required.

In the following I will present 3 different views of variational autoencoders to help the reader build intuition for them.

Deep Learning View

From a practical deep learning point of view, the variational autoencoder is a regularized autoencoder.

The KL-divergence term in the loss function is actually a regularization term, that penalizes deviation of the latent representations of the input samples from the unit Gaussian distribution. And since the KL-divergence term can be arbitrarily large for arbitrarily low variance values, it prevents the network to collapse its latent distributions to very low variance Gaussians (Dirac delta distributions in the limit). Therefore the network has to be robust to the noisiness of the latent codes.

Since the latent representations cannot differ too much from unit Gaussians, the stochastic latent sampling ensures that the decoder generates plausible samples from all latent codes near the prior distribution, ensuring that it will produce plausible samples during inference for latent codes sampled from the prior distribution.

These two properties together ensure that the structure of the latent space is relatively simple, since the noisy sampling procedure prevents sharp boundaries between close regions of the latent space.

An extension of variational autoencoders has been proposed based on this view: the Beta-VAE [47]. It modifies the loss function by adding a hyperparameter β, which is a scaling factor of the KL-divergence term, as can be seen in Equation 2.5.

$$L_{\beta-VAE} = L_{AE} + \beta D_{KL}(p_{encoder}(Z|X = x)||p_{latent}(Z)) \tag{2.5}$$

From this view, it controls the strength of the regularization, and is widely used in practice. By tuning it, one can find a good balance between regularization and representation capability. It has also been shown [46] that tuning it is equivalent to tuning the variance of the decoder distributions and using a constant β.

In this work, since I used calibrated decoder distributions [46], I will only use the original VAE loss function (i.e. $\beta = 1$) since it is theoretically more grounded.

Variational View

From the variational point of view it is only a coincidence, that the variational autoencoder is an autoencoder, since it is a generative model by design [48]. Being an autoencoder is only a side-product of efficiency considerations.

From this view, the main part of the model is the decoder, which could also be called the generator. During inference, it receives a sample from the prior latent distribution, and generates a sample according to it.

To optimize such a model we would like to make the samples more probable, i.e. increase $p_{decoder}(\hat{X} = x)$. This can be achieved by simply using the negative log probability of the data samples as a loss function, as can be seen in Equation 2.6. However since we do not know which latent codes produce which data example, optimizing this quantity would not be efficient, since $p_{decoder}(\hat{X} = x|Z = z)$ would be zero for most latent codes, and we would have to integrate over all z-s.

We can however derive a lower bound for the log probability, called evidence lower bound. In this case since we optimize the negative log probability, it is an upper bound, which becomes the loss function of the variational autoencoder.

$$
\begin{aligned}
&L_{desired} = -\log(p_{decoder}(\hat{X} = x)) \\
&L_{desired} \leq -L_{ELBO} = L_{VAE} \\
&p_{encoder}(Z|X = x) \rightarrow z \\
&L_{VAE} = -\log p_{decoder}(\hat{X} = x|Z = z) + D_{KL}(p_{encoder}(Z|X = x)||p_{latent}(Z))
\end{aligned}
\tag{2.6}
$$

The main idea from that view is that we need to have an inference network (an encoder), that is only used during training and estimates the latent code for each data example that could have produced it. So we only optimize the log probability for data generated from those latent codes that are considered to be good candidates.

This is actually the way that both papers [31][32] introducing variational encoders have proposed them.

Information-Theoretic View

One could also look at variational autoencoders from an information-theoretic perspective.

The term $-\log p_{decoder}(\hat{X} = x)$ can be seen as the amount of information need in nats (Euler's-number-based bits), to construct the data sample from our decoder [48].

What we do instead is, we first estimate a latent code from the image, whose extra information content is given by $D_{KL}(p_{encoder}(Z|X=x)||p_{latent}(Z))$ compared to that, if we would simply have sampled from $p_{latent}(Z)$. Then we need to add $-\log p_{decoder}(\hat{X} = x|Z=z)$ which is the information needed to reconstruct the sample from the latent code. So the sum of these two terms is a good estimation of the amount of information that is needed to construct the data sample. It is only an upper bound however, because our encoder is not ideal, so we "waste" some amount of nats.

2.1.5 Autoencoders with Image Augmentation

In this section I present two works combining autoencoders with image augmentations, that are relevant for this thesis.

The first of them introduces Augmented Autoencoders [49] which are essentially denoising autoencoders with the corruption process implemented as image augmentations. An interesting side-effect of this objective that some augmentations cannot be used, with the simplest example being random crop, since the cropping process is random and its parameters cannot be inferred from the image, the encoder would not be able to reconstruct it, which would lead to blurry reconstructions. In line with that the author of that work only use Gaussian blur, brightness, contrast and color transformations.

The second one proposes consistency regularization for variational autoencoders with respect to image augmentations [50]. The main idea is to measure the distance (the KL-divergence in this case) between the latent distributions of augmented and non-augmented images, and use that as an auxiliary loss, to enforce consistency between the two images, or in other words, invariance to augmentations.

I implement and test similar methods to both of these in this work.

2.2 Reinforcement Learning

Reinforcement learning (RL) [51] is a subfield of machine learning, in which the task is to train an active agent of an environment to maximize its cumulative future rewards by learning to make the right actions. The main components of reinforcement learning are shown on Figure 4. I will introduce the most important concepts in the next paragraphs, following the lectures of David Silver for the course on reinforcement learning at *University College London* [52].

An **agent** is an active actor, who receives an **observation** from its environment, and based on it and its **policy** (strategy), takes an **action**. During training it tries to improve its behaviour in order to get more **reward**.

The environment can be modelled as a Markov Decision Process (MDP) [53]. *"MDPs are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future*

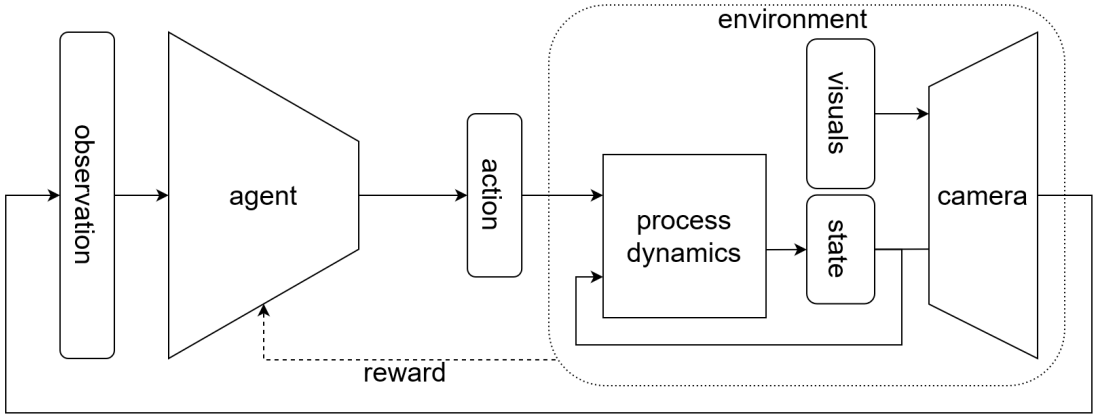

Figure 4: The main components of reinforcement learning

rewards. Thus MDPs involve delayed reward and the need to tradeoff immediate and delayed reward." [51, p. 37]

A finite MDP is defined by the tuple of its finite set of possible states (S) and actions (A), its state-transition probability matrix (P), its reward function (R), and its discount factor (γ).

State consists of those parameters of the environment, that, together with the actions, provide sufficient information to determine the future events. In a simulated environment all of the internal variables of the system can be considered a part of its state, however they could also contain implementation-specific irrelevant information as well. Therefore, we generally only consider those internal variables a part of the state, that contain meaningful information about the process and its future.

In case of an MDP the state can be observed completely, therefore it is called fully observable, and the states and observations can be considered the same. If this condition is not met, then it is a Partially Observable Markov Decision Process (POMDP) [54].

The dynamics, behaviour of an environment can be described using its **state-transition probability matrix**. As shown in Equation 2.7, the matrix determines for every possible action in every state the probability of getting into a given next state, for all possible next states. $P^a_{ss'}$ is an element of the matrix, whose value is the probability of s' corresponding to a current s state and a action.

$$P^a_{ss'} = \mathbb{P}(S_{t+1} = s' | S_t = s, A_t = a) \tag{2.7}$$

The state-transition matrix can be used to directly solve the MDP, however its size is huge even for simple problems, for more complex ones it is unfeasible to even be determined. In practice we generally tend to try to omit using it.

The **reward** is a scalar feedback signal in every timestep, which shows the agent how well it is doing in the current task.

The reward corresponding to a given timestep is determined by the **reward function**. Equation 2.8 shows that it outputs for all state-action pairs, how much reward they are worth. R_s^a is an element of the reward function, whose value is the expected value of reward in the next step R_{t+1}, for the current s state and a reward.

$$R_s^a = \mathbb{E}(R_{t+1}|S_t = s, A_t = a) \tag{2.8}$$

In theory we tend to assume that it is a part of the environment, however in practice one usually has to find a correct way to give rewards to the agent. We have to find a reward function for which the expected behaviour is actually the optimal one, i.e. it cannot be easily exploited, and which helps the learning enough, i.e. by not being to sparse. This process is called reward shaping [55].

Equation 2.9 shows **return** G_t, which is for each timestep the sum of all future rewards discounted by the **discount factor** γ. This is analogous to the present value in economics, if we consider the future rewards analogous to future cash flows, and our interest rate is constant. The discount factor represents that we consider future rewards closer in time more valuable, since farther ones are more uncertain.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+1+k} \tag{2.9}$$

A recurring element of reinforcement learning algorithms is that we try to estimate the value of either the states, or the state-action pairs for a given policy based on the returns. Another frequently used method is to estimate the optimal policy as a function, that maps actions to observations. And in model-based algorithms we can estimate a dynamics model of the environment, which maps next states to a current state-action pair, which can then be used for planning. If we use deep learning for these function estimation tasks, then it is called Deep Reinforcement Learning.

Recently deep reinforcement learning has been successfully applied to robotics [56][57], recommender systems [58], and chip design [59]. It has also been used to create professional- or even superhuman-level agents in classical games such as go [60], chess [61], or Atari [62], and in modern computer games such as Dota 2 [63] and Starcraft 2 [64].

Challenges

In this subsection I will discuss important challenges in the reinforcement learning setting. The first one of these is the exploration vs. exploitation trade-off. To find new and better behaviours in an environment, an agent has to explore and try behaviours with unknown outcomes. However, simply sticking to its best known behaviour can lead to a better and less risky performance in the short term. An effective reinforcement learning algorithm has to find a good balance between these two types of behaviours.

Another challenge is the task of credit assignment and sparse or delayed rewards. To be able to effectively learn, an agent has to decide retrospectively, which of its past actions led to its current state and therefore current reward. The more sparse or more delayed these rewards are, the harder the credit assignment gets.

Finally, the last one is data-efficiency. Deep reinforcement learning algorithms tend to require much more data to learn to solve a task than humans. This can be partially eliminated by using simulators for training, as shown in the next section, but it produces its own set of new challenges.

In this work I am using representation learning separately from reinforcement learning, to make training the agents more data-efficient.

2.3 Sim-to-Real Transfer

It can generally be stated that model-free reinforcement learning algorithms are not using gathered experience efficiently, so they need several interactions with their environment to learn to complete certain tasks. That makes training in the real world slow, and since it also usually needs human supervision, it is generally too expensive and sometimes even dangerous to train in the real world. A common solution to this problem is that the agents learn in simulated environments and are then transferred to the real world.

Sim-to-real transfer has already been successfully applied in robot arm manipulation [57], robot locomotion [56] and simple self-driving tasks [65].

However, our simulators can only be imperfect models of reality, so the performance of agents is usually reduced after the transfer, sometimes they are unable to complete the same task that they have already completed in the simulated environment. This is called the **sim-to-real gap**, and one has to take it into account if they want to apply agents trained in simulation to the real world.

The sim-to-real gap can be decreased using the following techniques:

- More realistic simulator environments
  - More realistic rendering and textures [66]
  - System identification and calibration [57]: more accurate dynamics parameters based on measurements
- Domain adaptation [67]: performance difference between the simulated and real environment can be decreased by fitting certain statistics to be more similar, by using auxiliary loss functions, or with transfer learning
- Domain randomization [68][69]: random perturbation of some parameters (e.g. visuals) of the simulated environment in every training episode, to broaden the range of environment, in which the agent performs properly (will be further discussed in Section 2.3.2)

- Regularization:
  - Observation-noise [57]: can make the agent more robust to discrepancies in its observations
  - Action-noise [57]: can force the agent to plan more robustly or behave more conservatively
  - Network regularization [70]: application of techniques typically used against overfitting in deep learning, such as L2 regularization [71] and droupout [72]

2.3.1 Domain Randomization

Domain randomization is a technique where in every training episode we randomly perturb some selected parameters of the simulator which we use. The aim of this technique is that by training a reinforcement learning agent in a diverse set of virtual environments, the range of environments in which it performs well gets broader, with the goal being that reality will fall within this range as well, increasing the probability of a successful sim-to-real transfer.

The two main methods of domain randomization are visual domain randomization [68][69], where visual parameters are perturbed, such as textures, lightning, background, and dynamics randomization [73], where the parameters of the process dynamics are changed. Their respective scopes are illustrated on Figure 5.

In the case of visual domain randomization and image observations, one could also use image augmentation methods instead of rerendering the images. These however should not distort the perceived state of the simulator, which is observed by the agent. In first-person view environments such as car driving, random cropping the image would distort the agent's perception of its own position on the track, so I consider it observation-noise instead of visual domain randomization, this however was also shown recently [74][75] to be effective in regularizing the networks to improve training performance. A simple non-distorting image augmentation example is Gaussian pixel noise, but one could change the brightness, contrast or the saturation of the images as well.

These two methods are quite different, and promote generalization in different aspects. While a large diversity can be helpful in the case of visual domain randomization, it is usually detrimental for dynamics randomization. This means that the randomization ranges are important hyperparameters for both methods. In this work I investigate both of them, and will introduce the most relevant works in the following sections.

2.3.2 Visual Domain Randomization

The technique of visual domain randomization is usually used for high-dimensional sensors, it is applied mainly for camera-based tasks, however it can easily be generalized for LIDARs as well.

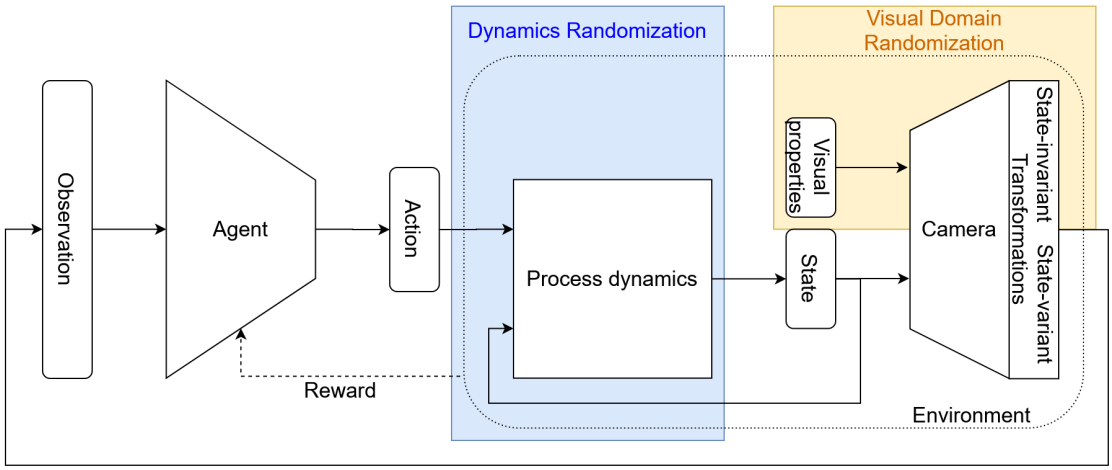

Figure 5: The scopes of visual domain randomization and dynamics randomization

Figure 6: Visual domain randomization: A [68], B [69], C [76]

Using cameras as sensors has the advantage that they are cheap, easy to acquire and can be used for a broad range of tasks. Their main difficulty however is, that they are high dimensional, their images contain a lot of data, and it is algorithmically not easy to get meaningful information out of that.

An agent that uses a camera sensor can be trained in simulator by rendering an image of a simulated camera and using that as input. The difficulty of transferring to the real world stems from the fact that the diversity of images in a simulator is much lower than in the real world. There is a danger that the model learns some specific properties of the simulator (like the colors and textures of some objects), that will not be the same in reality, or will be much more diverse. In that case since these inputs are different from anything the network has seen, its outputs become unpredictable.

Examples of visual domain randomization can be seen on Figure 6, other applications will be shown in the following sections.

Direct methods

In this subsection I will present applications of visual domain randomization from the literature, in which the randomized environments are used in the same way as the original one: as a training environment. This is the most straightforward way to apply visual domain randomization.

One of the first applications of the technique, in which the goal was to ensure generalization by visual diversity and not to make it visually more realistic, was for the task of indoor camera-based drone control [68]. The authors carried out the training in simulated indoor environments, in which they placed lightsources, furniture, closed and open doors in random positions and directions. They also randomly chose realistic wall textures. Though their network was pretrained on realistic images, they did not use any further real images during training, and their algorithm was capable of flying in the reality as well, with approximately one crash every minute.

The technique was also successfully applied in robotics for object localization [69]. The task of the network was to determine the positions of objects on a table, with other distracting objects present, based on camera images. The training was carried out without any real images, with a random amount of objects with randomized shape, texture and position, and with a random amount of lightsources with randomized direction, temperature and position. They also perturbed the position and direction of the camera, and the parameters of noise added to the images. They used multiple thousand non-realistic textures with randomized colors. Based on the ablation study, the randomized texture and camera positions had the highest impact, which is a finding that I have seen in multiple robotics applications.

The method has also been applied to object detection [76] as well, where the authors used it along with a wide range of image augmentation techniques. Findings show that their model has an accuracy similar to as if it has been trained on a highly realistic virtual dataset. In their case the randomized light sources and textures had the greatest impact on the result.

Indirect methods

In this section I will present some indirect applications of visual domain randomization, that do not use the randomized environments for training, but for network regularization or domain adaptation instead.

One of the techniques [77] is based on the following idea: the robustness of a policy can be measured by calculating the average distance of the policy outputs in the case of a randomized observation and its original counterpart. So we can just simply add this as a regularization term to the loss function, and use it for optimization, thereby ensuring robustness of the learned strategies.

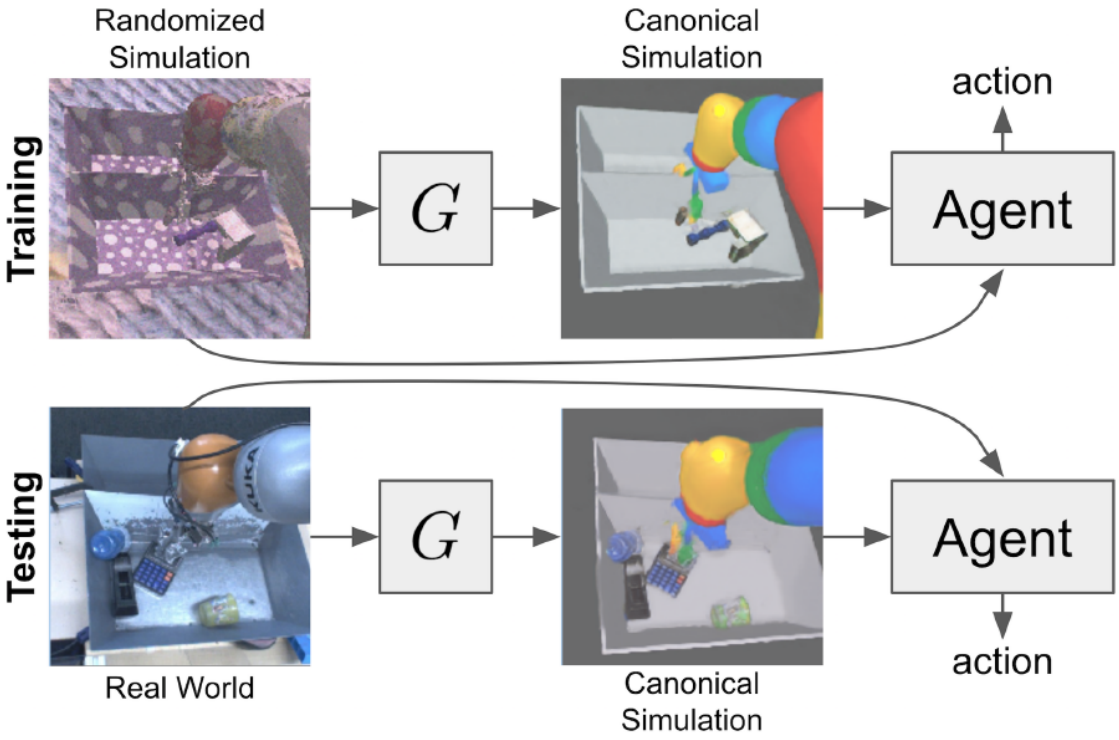

Figure 7: Domain adaptation based on domain randomization [80]

A similar solution is the following, where the distances are not calculated between the outputs, but between the activations of the last hidden layers instead [78], which can be seen as a high level representation of the input. This helps to avoid the situation where the two parts of the loss function have opposite effects, therefore in this case increasing the strength of the regularization parameter does not cause a performance drop when comparing with [77], as it is shown in the appendix. Another work proposes this same method [79], however an interesting detail is that they use a randomly initialized convolutional layers for data augmentation.

Visual domain randomization can also be used for domain adaptation [80]. In this case a network is trained to generate a canonical observation (an observation that is similar to the observations of the original environment) based on the randomized observation. We then train our agent based on images adapted by this network, and we can also use this network to adapt the real observations. The method is shown in Figure 7.

I also experiment with using pairs of randomized and canonical observations in my work.

2.3.3 Dynamics Randomization

The goal dynamics randomization [73] (Figure 5) is to reduce the sim-to-real gap by randomly perturbing the dynamics parameters of the simulator in each episode. This should result in a policy which is capable of functioning well on a wider range of dynamics parameters, making it more robust to changes to them. For specific architectures (e.g.

recurrent neural networks, discussed in dynamics estimation subsection) it is also possible to estimate the current environment dynamics and adapt to it on-the-fly [73].

Uniform Dynamics Randomization

When using uniform dynamics randomization, one has to select valid ranges for the selected dynamics parameters, and before each episode, sample a value from these and use it for setting the dynamics parameters of the environment. This is the simplest way to implement dynamics randomization, and in my experience, this is the most commonly used method, it however requires careful selection of ranges for each parameter.

It was successfully applied for object pushing with robot arms [73]. Apart from the dynamics parameters they also randomized the gains of lower level controllers, the sample time, and they also employed observation noise.

In another application four-legged robots were taught to walk [56], but they also used system identification along with dynamics randomization.

One of the most complex applied results was achieved with an algorithm, whose task was to learn to single-handedly solve Rubik's cube [57]. They not only used dynamics randomization to achieve that, but visual domain randomization, observation-noise, action-noise and system identification as well.

Adaptive Dynamics Randomization

In adaptive dynamics randomization methods either the ranges of dynamics parameters or the values themselves are changed during training, usually by another neural network. These techniques generally require more samples and more computing power compared with uniform dynamics randomization.

It is important to note here, that in some sense, even uniform randomization methods can seen as adaptive, if we consider that one usually tunes its parameter ranges manually across training runs, if the real performance of the agent requires it [81].

The first class of these techniques views the selection of dynamics parameters as a higher level problem, and considers it as an outer reinforcement learning problem, independent from the inner one, where the reward is the performance of the inner model in the real environment [81]. That way it proposes the automate the process of real testing and manual tuning of the parameters. Unfortunately this solution has the drawback, that it requires a considerable amount of experience from the real world, which makes the implementation it difficult in most cases.

A similar method that does not require real world testing at its core is active domain randomization [82], in which the higher level algorithm uses a discriminator networks error as reward, which is trained to discriminate between trajectories under randomized and reference environment dynamics, making the assumption, that the trajectories that

are easier to differentiate are more useful for the generalization of the inner agent. A drawback of this method is that the dynamics ranges have to be selected very carefully not to be too wide, as the outer network will try to make the dynamics and reference environments as different as possible over time.

Finally, it is also possible to apply adversarial networks for this problem [83]. In this setting the agent has an adversary, which is capable of applying disturbing forces inside the environment in a way which decreases its reward. That way the adversary can learn to exploit the weak points of the agent, and choosing these dynamics more frequently it can help to make it more robust.

Dynamics Estimation

In environments with dynamics randomization the neural network architecture has a significant role in enabling the trained agent to be able to adapt to the changing dynamics.

Model-based reinforcement learning algorithms might be able to explicitly estimate the current dynamics parameters and use them for planning [84].

Using recurrent neural networks enable the agents to use its memory to estimate the current dynamics parameters implicitly [73][57], which makes them a good fit for algorithms using dynamics randomization.

Feedforward networks are unable to estimate dynamics by default, however it can still be useful to randomize the dynamics during training as it forces them to learn more robust control strategies. However if we concatenate past observations to the input of the agents (e.g. frame stacking), this enables them to learn to estimate at least some dynamics parameters implicitly (e.g. estimating acceleration and motor strength based on it) which makes them capable to adapt to changing dynamics [73].

2.4 State Representation Learning

State representation learning [85], shown on Figure 8, is the task of learning representations that are useful for a downstream reinforcement learning task. When one does not try to explicitly learn a representation, just lets the network learn to process images minimizing solely its reinforcement learning loss, it is called end-to-end reinforcement learning, which can be considered the contrary paradigm to state representation learning.

For learning representations, one may or may not use the fact that the data used for reinforcement learning is sequential and is conditioned on past states and actions.

This fact can be used in the following ways: one can learn a model, which predicts the next observation based on the last observation and action. One can also learn an inverse model, which given the current and previous observation, predicts the previous action that was taken. It is also possible to use the reward signal for representation learning by trying

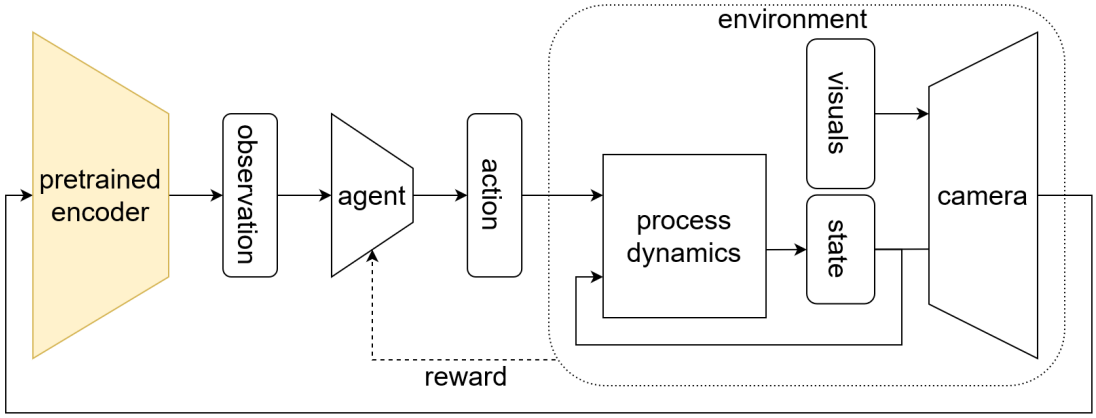

Figure 8: Block diagram of state representation learning (used for training the encoder) applied in the reinforcement learning setting (used for training the agent). Difference with end-to-end reinforcement learning highlighted in yellow

to predict a reward given an observation and optionally an action corresponding to the same timestep, which corresponds to estimating the reward function.

Model-based reinforcement learning methods usually use some combination of the above mentioned and even additional auxiliary objectives for state representation learning, with a recent and complete example being DreamerV2 [86].

For this thesis however those works are the most relevant that do not rely on the fact that the data is generated by a Markov decision process. The main advantage of these methods is their simplicity and that general image representation learning methods can simply be applied in those cases.

There are three groups of these methods, the first being supervised regression to quantities that based on expert knowledge are known to be useful for the reinforcement learning task at hand. For supervised regression I follow the Bottleneck method [87] where the main idea is to separate the network used for reinforcement learning into a perception and control module, and train the perception module in a supervised way to predict the required physical quantities (which are the physical parameters of the track in my case: angle, distance and curvature).

The second is self-supervised representation learning via undercomplete autoencoders [88]. These methods learn to compress images in a self-supervised manner, providing a denser representation for the reinforcement learning process. A drawback of these methods is if an object that is important to the task is visually insignificant (such as the ball in a ping-pong game), it might get lost during compression. For the self-driving use-case however, which is what this thesis investigates, this issue was not found to be relevant. I will present other autoencoder-based state representation learning methods in Section 2.5.

The other group applies contrastive representation learning methods for learning representations [89]. The main idea of contrastive learning is to try to discriminate differently

augmented versions of the same images from differently augmented versions of different images. Data augmentation, especially random crops are crucial [24] for these methods to work well, however in the self-driving use-case most image augmentations cannot be used because they distort the agents' perceived state as I discussed in Section 2.3.1. Therefore I do not use contrastive representation learning methods in this work.

2.5 Self-Driving Applications

In the following section I present algorithms from the autonomous driving literature that are the most relevant to my use-case.

World Models [90] uses a variational autoencoder in combination with a recurrent neural network for state representation learning. The autoencoder encodes the images into a latent space, while the recurrent neural network acts as a learned model, mapping latent states and actions to next latent states. Finally a small controller neural network, getting latent states as inputs, is trained with evolution strategies [91]. The algorithm is simple, self-supervised (apart from environment rewards) and it was the first to solve the CarRacing-v0 OpenAI Gym environment.

The authors of the next algorithm [92] propose to use the soft actor-critic (SAC) [93] reinforcement learning algorithm with continuous action space on top of the features of a variational autencoder trained in the Donkey Car environment.

The last work [94] shows that it is possible to learn to drive a real car in under 24 hours in the real world (with the help of a human safety driver), by simultaneously training a variational autoencoder and using its compressed representation as reinforcement learning inputs. Interestingly, they find that the autoencoder does not improve performance in simulation, but in the real helps the training converge using 3-4 times less data. They use the deep deterministic policy gradient (DDPG) [95] algorithm with a continuous action space.

2.6 Most Relevant Pieces of Literature

In this section I list those works from the chapter that are most closely related to my work, and I also summarize the connections to help the reader easily find the most relevant references.

- Denoising autoencoders w.r.t. image augmentation: [49]
- Invariance regularization of autoencoders w.r.t. image augmentation: [50]
- Invariance regularization w.r.t. domain randomization: [77][78][79]
- Visually randomized and canonical observation pairs: [80]

- Calibrated decoder distributions to avoid tuning the β hyperparameter of variational autoencoders: [46]
- Reinforcement learning with supervised state representation learning using regression: [87]
- Reinforcement learning for autonomous driving with self-supervised state representation learning using variational autoencoders: [92][94]

Chapter 3

System Design

3.1 System Architecture

The central task of my thesis was to investigate, implement and compare different domain randomization techniques, so my goal was to find a system architecture which supports me in this investigation.

As I showed in Sections 2.3.2 and 2.3.3, one can change the appearance of an environment on the one hand, and can also change its dynamics on the other. The two techniques have considerable differences: while the appearance can be changed to a wide range of looks, the dynamics range has to be carefully selected not to limit the maximal achievable performance of the agent. The randomized dynamics effect the state transition probability matrix of the Markov decision process, while the appearance change only affects the observations.

The main design decision of my thesis was to separate perception and control, and to select a system architecture to support that. I decided to apply state representation learning with reinforcement learning instead of testing the algorithms via end-to-end reinforcement learning.

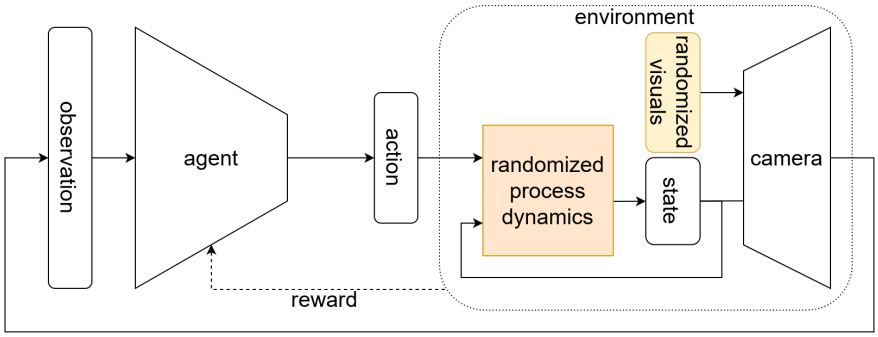

Figure 9: End-to-end reinforcement learning

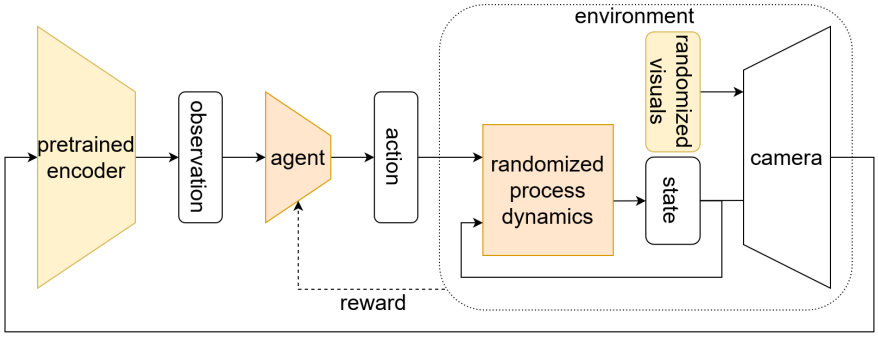

Figure 10: The chosen system architecture. Interacting modules illustrated with the same colors

As it is shown on Figure 9, when using both types of domain randomization, both of them affects the weights of the neural network contained in the agent.

In contrast, as I illustrate on Figure 10, my design decision instead was to pretrain an encoder, which was implemented as a convolutional neural network, separately, that way it would be the only module affected by visual domain randomization (both indicated with yellow in the figure). This also enabled me to experiment more flexibly with a range of visual domain randomization algorithms in the simpler and more consistent supervised and self-supervised settings.

In line with that, agents, implemented as fully connected multilayer perceptrons without convolutional layers, with and without dynamics randomization (both indicated with orange in the figure) could be trained using the same encoder network, saving computation, time, and more importantly leading to better comparability.

Advantages of the chosen architecture:

- The modules can be trained separately, the encoder (perception module) in a supervised or self-supervised way, the agent (control module) based on true state or encoder outputs in a reinforcement learning way. This also makes the results more consistent, as only a subnetwork is trained with reinforcement learning, whose performance usually has higher variance.
- Since randomized visuals only effect the encoder (perception module), while randomized dynamics only effect the agent (control module), the two methods of domain randomization can be investigated independently from each other.
- Using the same encoder with different agents or different encoders with the same agent helps with comparability.
- The modular architecture supports experimentation and makes debugging easier.

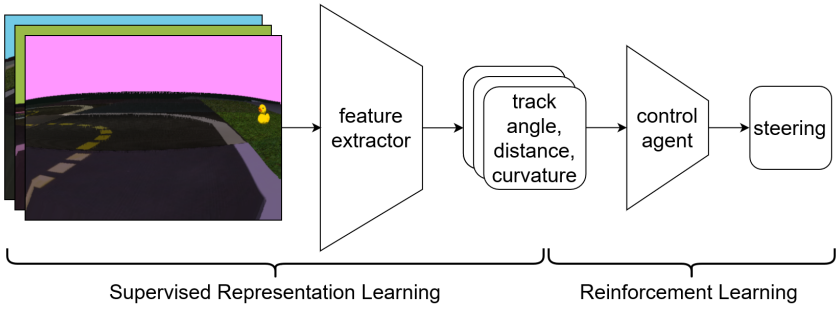

Figure 11: Supervised encoder with direct visual domain randomization

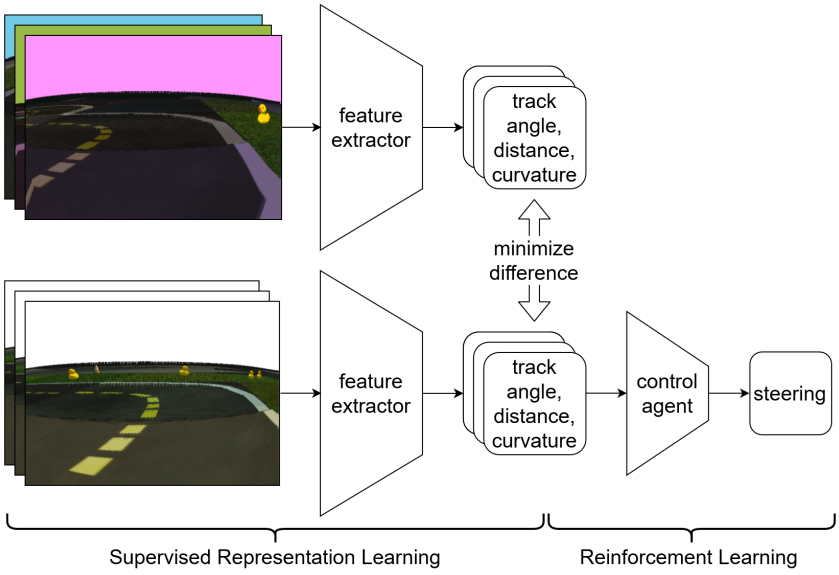

Figure 12: Supervised encoder with indirect visual domain randomization

3.2 Visual Domain Randomization Algorithms

Note: On the following figures a stack of images with colorful sky symbolizes "randomized" observations, while another one with only white sky symbolizes "canonical" observations. The definition of "randomized" and "canonical" observations can be different across algorithms.

Direct Visual Domain Randomization Algorithms

My solutions employing direct visual domain randomization (Section 2.3.2) are shown on Figure 11 and Figure 13. The encoder (perception module) receives directly images with randomized appearance, and has to learn to complete its task using those.

For both the supervised and self-supervised encoder types a baseline is run without domain randomization, and another one with it. While in the case of the self-supervised encoder

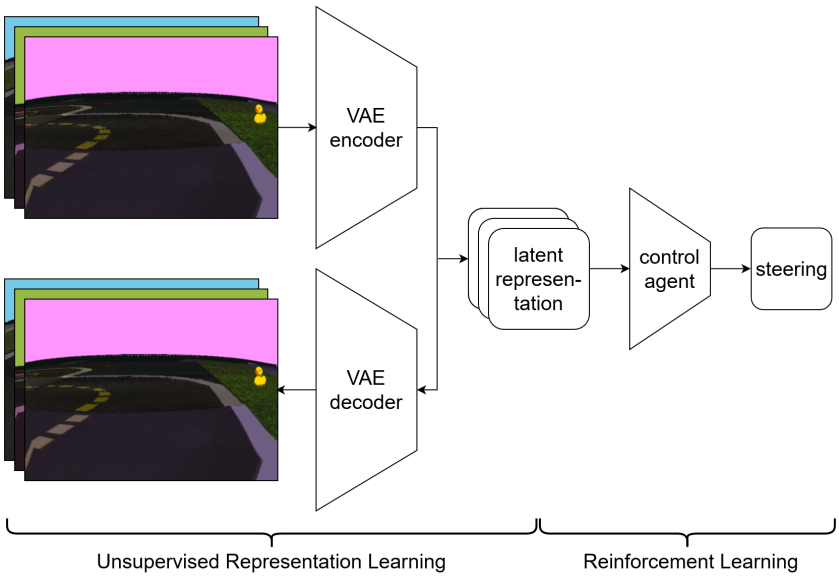

Figure 13: VAE with direct visual domain randomization

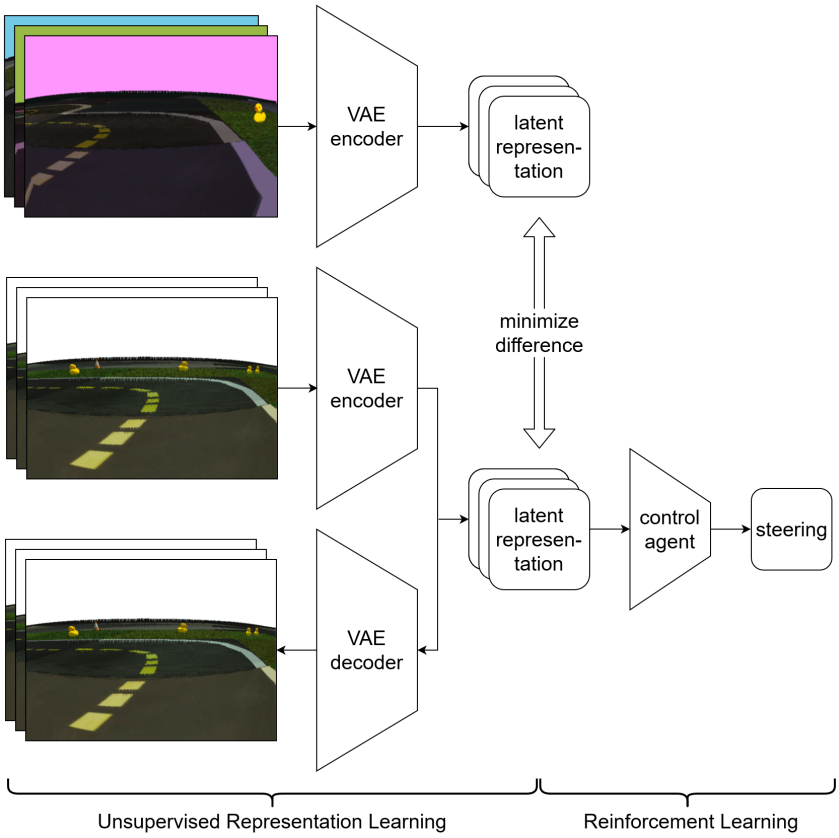

Figure 14: VAE with indirect visual domain randomization

a third training is run in which augmented real images are mixed into the randomized images.

Indirect Visual Domain Randomization Algorithms

My solutions that employ indirect visual domain randomization (Section 2.3.2) are shown on Figure 12 and Figure 14. Since my implementation regularizes the outputs of the encoder and not the output of the agent, it follows those methods [78][79] that do not regularize the outputs of the controller network, but one of its intermediate layers instead.

A distance measure is defined on the encoder outputs for both the supervised (mean squared error) and self-supervised (KL-divergence) cases, carefully, to be comparable with the main loss. This distance measure used to calculate an auxiliary loss based on the difference of representations between "randomized" and "canonical" images, and it is added to the main loss. That way the encoders are trained to become invariant to visual domain randomization over the course of training.

For these algorithms one can freely define what is considered a "canonical" image. For both the supervised and self-supervised encoder types two trainings are run: one where "canonical" is a non-randomized image, and another one where it is a segmented image (see Section 4.1.1 for more details).

3.3 State Representation Learning Algorithms

Supervised State Representation Learning Algorithms

My solutions employing supervised state representation learning are shown on Figure 11 and Figure 12. I follow the Bottleneck method [87] and regress to the track's physical parameters, such as angle, distance and curvature. For indirect visual domain randomization, I normalize the physical parameters to have zero mean and unit variance, and use mean squared error as the distance measure.

Self-supervised State Representation Learning Algorithms

My solutions employing variational autoencoders for self-supervised state representation learning are shown on Figure 13 and Figure 14. I follow For indirect visual domain randomization, I normalize the physical parameters to have zero mean and unit variance, and use mean squared error as the distance measure.

3.4 Domain Derandomization

In this section I present a summarized description of a novel domain randomization method that I proposed on the 2020 Student Research Conference [96].

3.4.1 Baseline

The motivation for proposing novel method was to combine visual domain randomization and unsupervised learning for increased sample efficiency using autoencoders. In order to do that we have to select a baseline solution, which combines both these techniques in a straightforward naive way.

The architecture of the baseline solution is shown on Figure 13, it is the variational autoencoder-based state representation learning using direct visual domain randomization. It can be trained on a dataset generated by gathering visually randomized observations from the simulator.

Using the encoder of the network, one can implement an observation wrapper for the simulator, which compresses the observations during the training of the reinforcement learning agent. I used the means of the latent code distributions as the latent representation. This baselines can already help the agent learn more sample-efficiently using the compressed representations of the visually randomized image observations.

Analysis of the baseline

Although this baseline method already improves sample-efficiency, it actually is not completely efficient, it still has room for improvement to help a transferring to reality as much as possible.

The main reason is the following: when applying visual domain randomization, we purposefully change parts of the input image observation, that are irrelevant to the task, i.e. the visuals of the environment. By changing these parameters stochastically in every training episode, we force the reinforcement learning agent to be invariant to these perturbations.

However when we use variational autoencoders to compress and reconstruct the observations, the visuals will become relevant, they will actually become quite important, since they inflict large changes in pixel-space. Therefore the variational encoder will have to encode the current values of these visual parameters to its latent space, to be able to reconstruct them accurately.

This has two drawbacks. The first is that the variational autoencoder will waste its latent capacity to encode information that we know is irrelevant, therefore it will only have less capacity to encode useful information for the reinforcement learning task. One could counteract that by increasing the dimensionality of the latent space, but this makes the

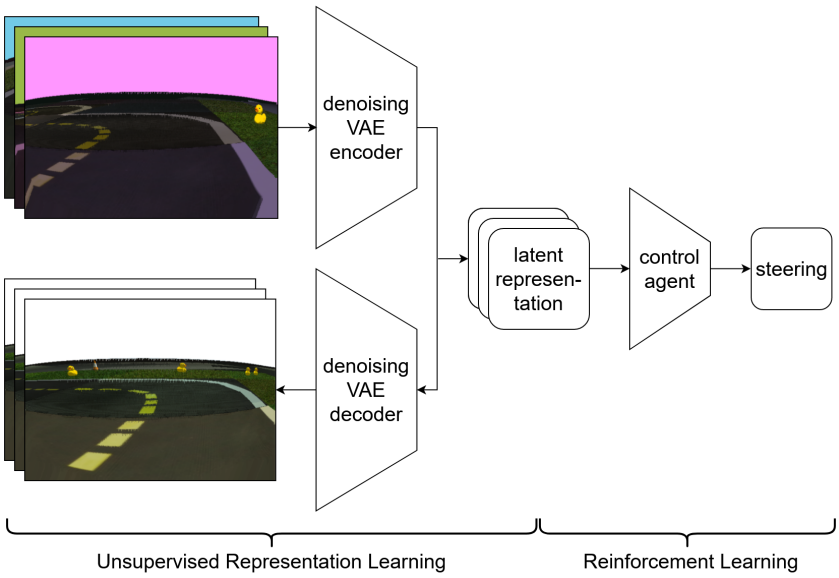

Figure 15: Domain derandomizing variational autoencoder

reinforcement learning procedure less sample-efficient by increasing the dimensionality of the observation space.

The second is that by encoding information about visual features to the latent space, we risk the possibility that if the agent does not become completely invariant to them, it can hinder its performance in the real world.

The conclusion is that we should modify the learning objective of the variational autoencoder in a way, that prevents, or at least does not promote the encoding of features that are perturbed by visual domain randomization.

3.4.2 Proposed Solution

The main idea of this work is that in order to prevent the encoding of the values of randomized visual features, we should reconstruct the non-randomized (canonical) image observations.

One can recognize that this changes the variational autoencoder's objective to a denoising one, by interpreting the visual domain randomization as a corruption process over the canonical image, which makes the perturbed visuals to be considered noise.

This lines up exactly with the consideration, that we only perturb visuals that are irrelevant to the task. This will encourage the autoencoder to ignore these perturbed features, and utilize its latent capacity in the most effective way. Based on these considerations I would argue that this is a more natural and effective way of combining domain randomization with unsupervised representation learning, than the naive baseline solution introduced above.

The architecture of the proposed method is shown on Figure 15.

Motivations of Design Choices

Because of the fact that the method proposes the usage of a denoising variational autoencoder in conjunction with a reinforcement learning agent and visual domain randomization, which can be seen as quite convoluted, I would like to discuss the motivations behind these design choices. I would also like to give intuition and clarify the role of each element of the proposed technique. The theoretical background and further explanation of these statements are provided in Chapter 2.

Motivations for each element of the method:

- Visual domain randomization: to decrease the sim-to-real gap and increase the chances of a successful sim-to-real transfer
- Undercomplete autoencoder: to increase sample efficiency by compressing the observations to a lower dimensional space
- Variational autoencoder: to regularize the latent space and make the model more explainable by being able to draw unconditional generated samples from it
- Denoising autoencoder: to be able to utilize full capacity when combined with visual domain randomization and to encourage meaningful representation learning

3.4.3 Applications and Extensions

The general application of the method is naturally in settings where one would like to increase the sample efficiency of the reinforcement learning training procedure, while also using visual domain randomization simultaneously. In this section I summarize further use-cases of the method, that are made possible by the design choices discussed above.

Since the variational autoencoder is a generative model, it provides an opportunity to make the model more explainable. One can observe the quality of unconditional generated samples during and after training, simply by drawing samples from the latent prior (unit Gaussian) and performing a forward pass on the decoder. Using this method, one can evaluate the performance of the decoder visually.

As we have seen from the information-theoretic view of variational autoencoders (Section 2.1.4) the KL-divergence term of the loss function can be interpreted as the extra information in the posterior latent distribution conditioned on an image over the prior one.

This can be used as a tool to evaluate the amount of information the model needs to encode a specific image. We can gather images from the real world and interpret them as visually randomized observations. Then by performing a forward pass using them, we can monitor the KL-divergence term of the loss function. An advantage of this application is that it does not require any labels for the real images, which makes real world data collection simple.

Another possibility is, somewhat similarly to the method mentioned above, to evaluate the reconstructions of real world images. Since these images are interpreted as visually randomized observations, the network will try to produce their canonical versions, i.e. as if these scenes would have been rendered in the simulator with canonical textures.

This method provides an elegant way to augment the latent codes to regularize the network of the reinforcement learning agent, thereby preventing it from overfitting. The idea is that we should not use mean of the posterior latent code distribution, but instead sample from it. That way we can achieve a higher diversity in the input observations of the agent, thereby regularizing it and making it more robust, with possibly some loss in the final performance. One could also define a sampling temperature, which can be used to control the regularization strength.

An advantage of the method is its flexibility in terms of what we consider as a canonical observation. In the standard setting it was an observation that has been rendered with the original, non-randomized textures of the simulator. But what if we consider other textures canonical? We could use this as a tool to inject prior knowledge into the system, e.g. by changing the texture of all objects such that we render a segmented image. This can be achieved by using black textures for everything unimportant, and only using single colors for important objects.

A limiting factor of the method is that it needs randomized and canonical rendering of the same scenes. This could be hard to implement if the code of the used simulator is either highly complex or unavailable. Therefore one could also use image augmentations as a corruption process. That way the only thing needed are the canonical observations, which can simply be the standard observations of the simulator, while the randomized observations will be augmented versions of them. One should however take great care when designing the augmentation pipeline, and should only use such augmentations that do not distort the perceived state of the agent.

Chapter 4

Implementation

4.1 State Representation Learning Implementation

I have implemented the state representation learning algorithms in the PyTorch deep learning framework, with the usage of the PyTorch Lightning library, which is a lightweight PyTorch wrapper that removes boilerplate code and makes the remaining code more concise and organized, therefore more readable.

The implemented state representation learning algorithms are summarized in Table 1.

4.1.1 Dataset Collection

To be able to apply state representation learning efficiently, I needed to generate a dataset of observations and corresponding physical parameters. Though one could use a streaming-type dataset, which is generated by the simulator, this would not only bottleneck the training speed, but the samples would not be statistically independent and identically distributed.

Based on these considerations, I have generated and saved images of 200.000 scenes during the training of a simple convolutional reinforcement learning agent, and stored 3 different renderings for each image: a visually randomized, a canonical (non-randomized), and a segmented one. For saving and loading, the *pickle* python package has been used. I have also saved the corresponding speeds, angular velocities, lane angles, lane distances, and lane curvatures for future work. The generated dataset has a 22.6 GB storage size.

Four different maps were used, all being a part of the official Duckietown simulator (Duckietown Gym [97]): *4way, loop_empty, udem1 and zigzag_dists* can be seen in Figure 16.

Note that there are intersections on two out of these four maps, which I included on one hand to increase diversity, but also to help future efforts dealing with intersections.

I have also implemented a fourth type of observation, the augmented observation, which is always generated on the fly by applying a random convolution [79] to the canonical

Table 1: A summary of all of the encoder variants trained for state representation learning. For each non-end-to-end algorithm a control agent was trained as well using its input observation type for reinforcement learning in the Duckietown Gym.

Algorithm	**Note**	**Input observation**	**Output**	**Invariance observation**
E2E	End-to-end	Canonical image	Actions	-
E2E RND	End-to-end	Randomized image	Actions	-
SUP	Supervised	Canonical image	Physical parameters	-
SUP RND	Supervised	Randomized image	Physical parameters	-
SUP CAN	Supervised	Canonical image	Physical parameters	Randomized image
SUP SEG	Supervised	Segmented image	Physical parameters	Randomized image
VAE	Autoencoder	Canonical image	Canonical image	-
VAE RND	Autoencoder	Randomized image	Randomized image	-
VAE CAN	Autoencoder	Canonical image	Canonical image	Randomized image
VAE SEG	Autoencoder	Segmented image	Canonical image	Randomized image
VAE REAL	Autoencoder	Randomized, real images	Randomized, real images	-
DVAE CAN	Derandomize	Randomized image	Canonical image	-
DVAE SEG	Derandomize	Randomized image	Segmented image	-

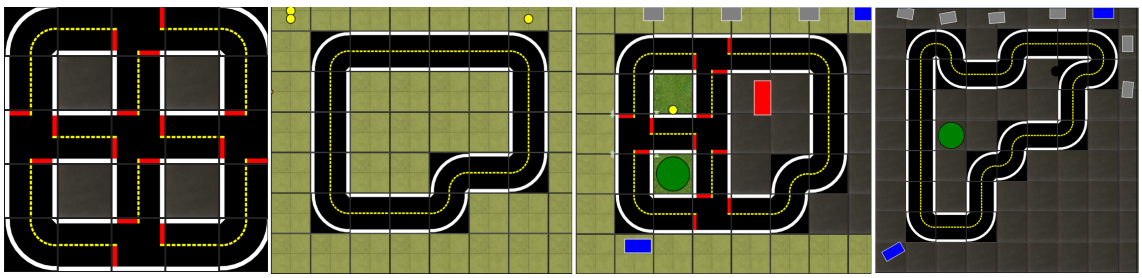

Figure 16: The maps that were used: 4way, loop_empty, udem1 and zigzag_dists

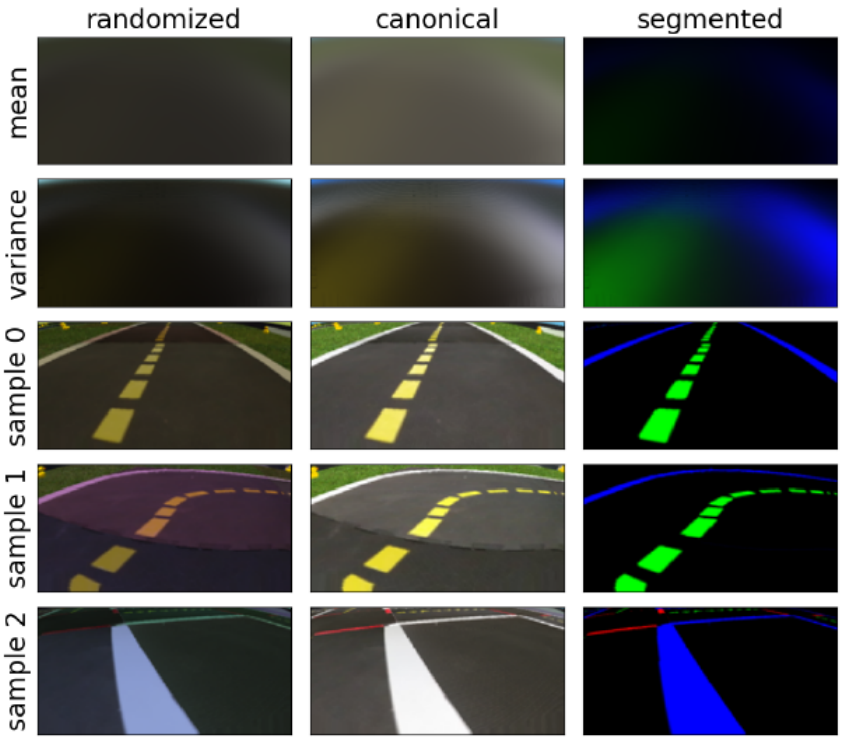

Figure 17: The mean and variance images corresponding to each observation type, with 3-3 corresponding samples

observation, and then min-max scaling it to be between 0 and 1. This has been used to implement the extension to the proposed method, domain derandomization [96], and is not detailed further in this work.

Examples of the three types of observations can be seen in Figure 17.

Using the dataset, the mean and variance observations for each image type have been determined on the training set before the training, to enable the usage of calibrated decoders [46] in the variational autoencoder, and to avoid tuning the KL-divergence regularization strength, which would have made the experiments computationally much more demanding.

I have created another dataset as well, which contains 19.000 real images, downloaded from online logs of Duckiebots from the Duckietown website [97]. I handpicked 3 videos of agents with reasonable performance and diverse lighting conditions, extracted and saved the frames from them. These images have been saved under the *randomized images* category, with no corresponding canonical or segmented frames, nor physical values.

4.1.2 Supervised Feature Extractor Implementation

Since we want to learn to predict multiple physical outputs simultaneously and accurately, I wanted to apply a method which works out-of-the box, and doesn't require further hyperparameter tuning, so I decided to normalize all input value parameters to have zero mean and unit variance, as shown in Equation 4.1, and then applied a simple mean squared

error as the loss function, with batch size N, x being an input image, y being its true physical label, y_{norm} being its normalized label and $\hat{y}$ being its predicted normalized label by the network.

$$\begin{aligned} & y \to \mu_y, \sigma_y^2 \\ & y_{norm} = (y - \mu_y)/\sigma_y \\ & encoder(x) \to \hat{y} \\ & L_{SUP} = \frac{1}{N} \sum_x (y_{norm} - \hat{y})^2 \end{aligned} \tag{4.1}$$

Equation 4.2 shows, how the invariance regularization loss was calculated for the indirect visual domain randomization. Similarly mean squared error was calculated but between the predicted representations of the input image and the invariance-input image, whose transformation we want to be invariant to. This auxiliary loss is added to the main loss directly, without reweighting, which is motivated by the need not to introduce a new hyperparameter, and also the result from a related work [78] (Appendix B), which shows that the network's performance on the reference domain does not depend heavily on the weight of the invariance loss, if it is not the controller's output which is regularized, but an earlier layer.

$$\begin{aligned} & x \to x_{inv} \\ & encoder(x_{inv}) \to \hat{y}_{inv} \\ & L_{INV} = \frac{1}{N} \sum_x (\hat{y} - \hat{y}_{inv})^2 \\ & L = L_{SUP} + L_{INV} \end{aligned} \tag{4.2}$$

4.1.3 Variational Autoencoder Implementation

I have used pixelwise calibrated decoder distributions following the work of [46], but I iterated once over the whole training dataset instead of estimating the variances of the distributions based on the minibatches.

I have used a unit Gaussian as a latent prior, and another Gaussian as the latent posterior whose parameters are produced by the encoder based on the input image x. The distribution of the decoder is considered Gaussian as well, with its mean output by the decoder based on the latent code z, and its variance calculated beforehand.

Making these assumptions, the calculations needed for the autoencoder are simplified to those shown in Equation 4.3, with $\mathcal{N}$ being the normal distribution, and μ and σ^2 being the means are variances of their corresponding distributions, using the fact that the KL-divergence can be analytically calculated for two normal distributions.

$$
\begin{aligned}
&p_{latent}(Z) = \mathcal{N}(0, I) \\
&p_{data} \rightarrow \sigma_x^2 \\
&encoder(x) \rightarrow \mu_z, \sigma_z^2 \\
&p_{encoder}(Z|x) = \mathcal{N}(\mu_z, \sigma_z^2) \rightarrow z_{sample} \\
&decoder(z_{sample}) \rightarrow \mu_{\hat{x}} \\
&p_{decoder}(\hat{X}|z_{sample}) = \mathcal{N}(\mu_{\hat{x}}, \sigma_x^2) \\
&L_{VAE} = -\log p_{decoder}(\hat{X} = x|z_{sample}) + D_{KL}(p_{encoder}(Z|x)||p_{latent}(Z)) \\
&L_{VAE} = \frac{1}{2}\sum_x((\mu_{\hat{x}} - x)^2/\sigma_x^2) + \frac{1}{2}\sum_z(\mu_z^2 + \sigma_z^2 - \log(\sigma_z^2) - 1)
\end{aligned}
\tag{4.3}
$$

$$
\begin{aligned}
&x \rightarrow x_{inv} \\
&encoder(x_{inv}) \rightarrow p_{encoder}(Z|x_{inv}) \\
&L_{INV} = D_{KL}(p_{encoder}(Z|x_{inv})||p_{encoder}(Z|x) \\
&L_{INV} = \frac{1}{2}\sum_z(\frac{(\mu_{z_{inv}} - \mu_z)^2 + \sigma_{z_{inv}}^2}{\sigma_z^2} - \log(\sigma_{z_{inv}}^2) + \log(\sigma_z^2) - 1) \\
&L = L_{VAE} + L_{INV}
\end{aligned}
\tag{4.4}
$$

Equation 4.4 shows, how the invariance regularization loss was calculated for the indirect visual domain randomization. Similarly KL-divergence was calculated but between the predicted latent distributions of the input image and the invariance-input image, whose transformation we want to be invariant to. This auxiliary loss is added to the main loss directly, without reweighting, which is motivated similarly as in the supervised case. Please note that by following this line of reasoning, we get a similar loss formulation to the work on VAEs with consistency regularization [50], with regularization strength being 1.0 and without trying to reconstruct the invariance-input images.

Decoder Initialization

An important note is that with default initialization the decoder for segmented image generation task usually converged to a completely black local optimum. This is caused by the bias of the last layer being initialized to 0, which causes the network to output grey images after the sigmoid output activation, which leads to too large gradient updates initially using the mostly black images, which destroys initalization, and the network jumps to a local optimum generating only black images.

$$mean\_output = \frac{1}{N * H * W} \sum_{n,i,j} x_{n,i,j}$$
$$bias = \log(\frac{mean\_output}{1 - mean\_output}) \tag{4.5}$$

I solved this issue by initializing the bias of the last layers of the decoder networks to be the logit (inverse sigmoid value) of the mean pixel brightness of the current class of images for each channel separately, as shown in Equation 4.5, with n, i, j being the image index, row index and column index respectively, and N, H, W being the dataset size, image height and image weight respectively. This meant that after initalization, the network already generates well calibrated images on average, which solved the mentioned issue while also accelerating convergence in the early iterations of training.

Bounding the Range of Autoencoded Representations

An implementation detail is that the Stable-Baseline3 RL library [98], which I used for reinforcement learning trainings, requires that all observation spaces should have limits, i.e. minimal and maximal values.

Though the latent distributions of VAEs are centered around zero, we cannot know for sure there range in advance. My solution to this issue was to L2-normalize the representations, bounding their range between -1 and 1, while keeping their directions unchanged. Empirically this did not decrease performance.

Utilizing Real Images

An advantage for the standard VAE is that it not only does not require physical labels (which we do not have for real images), but it does not require image pairs either (which we again do not have for real images).

This means that a standard variational autoencoder can be trained on randomized images, and we can enhance its training set by adding real images to it to help with generalization to reality. The standard VAE is the only investigated algorithm for which this is possible.

I applied color augmentation and random resized crops on the real and randomized images and mixed them in a 50-50% ratio for training this model, aiming for the highest robustness possible. The algorithm is called VAE REAL.

4.2 Reinforcement Learning Implementation

I have implemented and evaluated the proposed method in the Duckietown self-driving car environment [97], using the Stable-Baselines3 [98] reinforcement learning library which

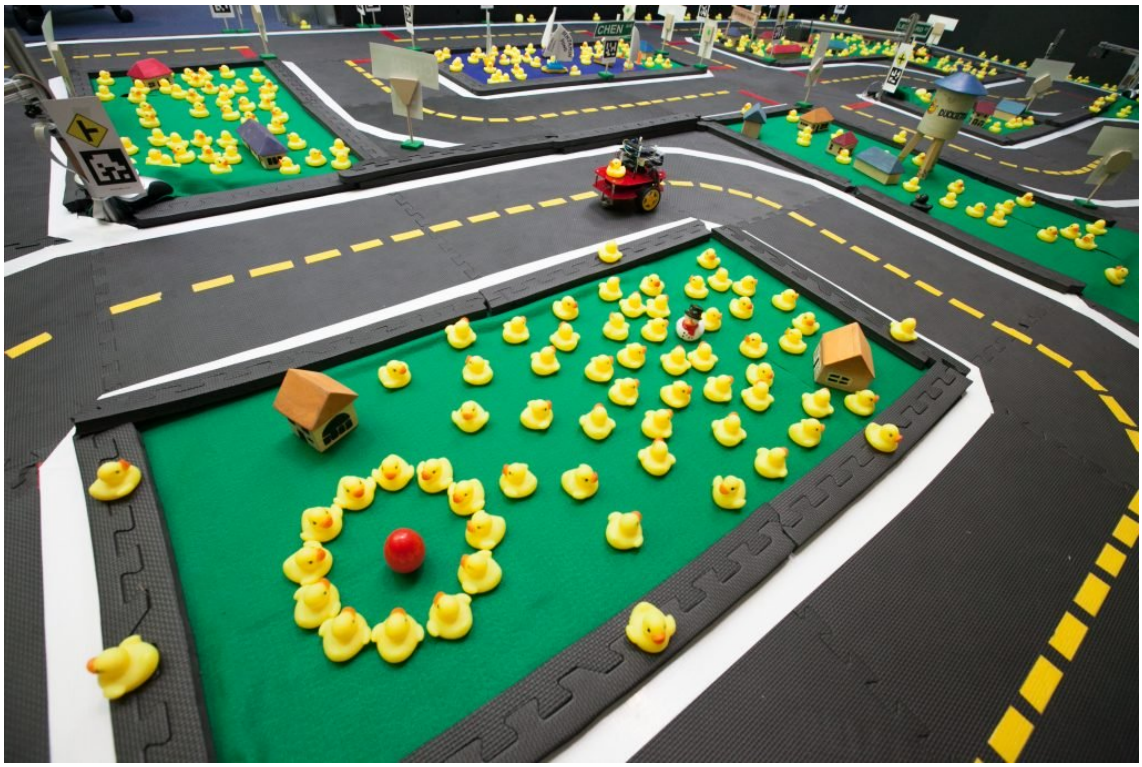

Figure 18: The Duckietown platform [102]

uses the PyTorch [99] open-source deep learning library as its backend. I have used the PPO algorithm [100] for the training most of the RL agents with a continuous action space. I would like to note here, that during the first semester of my work on this thesis, I used the original Stable-Baselines library, which was Tensorflow-based [101] and offered slightly different functionality.

4.2.1 The Duckietown Platform

The Duckietown self-driving platform consists of multiple main parts, one of which are the **Duckiebots**, which are small-sized autonomy-capable vehicles, that are controlled by a Raspberry Pi, and are equipped with a single camera. They are differential drive vehicles, which means that they do not use a servo motor for steering, instead their motors are independent on their sides, and they can turn by driving their motors at different speeds.

Another part of the system is **Duckietown**, which is a small scale well-specified, real, physical driving environment, which can be used by the Duckiebots for driving, therefore their performance can be evaluated in a real environment.

The last main part is the **Duckietown Gym**, which is a self-driving car simulator, implementing the OpenAI Gym interface. The simulator contains multiple maps that provide tasks such as lane following, navigation in intersections, and pedestrian- (duckie-) and vehicle- (duckiebot-) avoidance.

An important feature of the simulator is that it implements visual domain randomization by optionally perturbing the following components:

- Position and color of the light source
- Camera position, angle, and field of view
- Color of the sky
- Texture and color of road tiles
- Amount, type, position and color of environment objects

I have perturbed all of these components when training the reinforcement learning agent with visual domain randomization.

One can create wrappers for the simulator, which can be "wrapped around" it, meaning that they generally override or modify some feature of the simulator, creating a new simulator with a customized set of features.

Wrappers enable the user to customize the observation space, the action space, the reward function, and also to add new custom functionality. I have used this feature extensively, it enabled me to create a codebase that does not require any modification of the original simulator codebase, while also enabling me to run the same agents using a unified interface for the Duckietown Gym simulator, my custom lane following simulator (introduced in the following section) and for reality too.

4.2.2 Custom Lane Following Environment

One of a drawbacks of the Duckietown Gym simulator is that is has very limited support for dynamics randomization. It has a flag implemented for it, which, when active, randomizes the trim of the vehicle (i.e. the balance of the strength of the two motors), causing a slow natural rotation during movement when the trim is different from 0.

One can also set a *frame rate* (default 30) which corresponds to the frame rate of the physics, and a *frame skip* (default 1), which corresponds to the simulations steps taken per controller action (controller frame rate = frame rate / frame skip). These however will stay constant during training, so are still very limited in helping in testing dynamics randomization.

Since implementing a fully featured dynamics randomization into Duckietown Gym would have been not only cumbersome but error-prone as well, I decided to implement a custom lane following simulator, which only simulates the physical process of lane following, without rendering images or dealing with image perception at all. The block design of this custom environment can be seen on Figure 19.

This gave me the flexibility to implement a lightweight simulation environment which only deals with pure physical quantities, making it an order of magnitude faster, which enabled

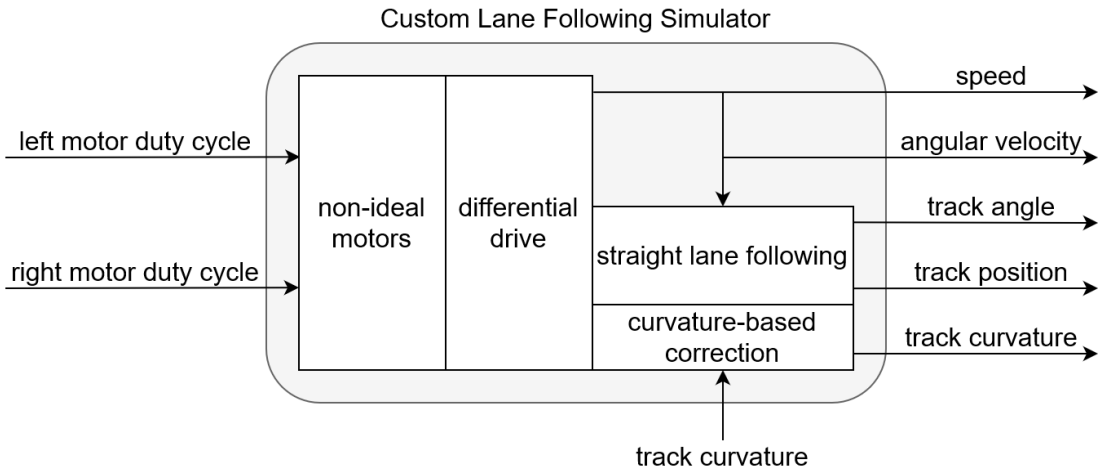

Figure 19: Block diagram of the custom lane following simulator

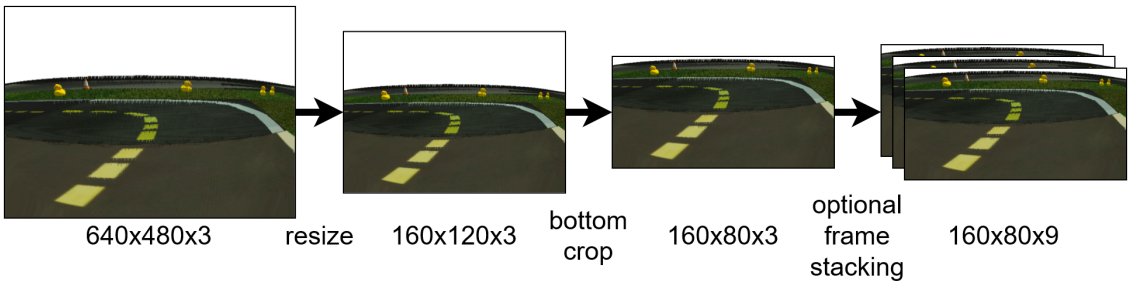

Figure 20: An illustration of the preprocessing pipeline

me replace the tile-based track system with a randomly generated one to eliminate another possible source of overfitting.

I implemented this environment to be compatible in software interface with the Duckietown Gym and also well aligned with it in dynamics, so that transferring control policies between to two is possible.

This gave me a tool which enabled faster and easier prototyping of control policies and also a complete evaluation opportunity for randomized dynamics. The details of the custom simulator are presented in Section 4.3.

4.2.3 Observation Space

Following the work of [103], I have downscaled the 640x480 input image by a factor of 4 on both sides to a resolution of 160x120 then I cropped out the upper third of the image, which generally only contained information about the background objects and the sky, which yielded an observation of size 160x120.

Theoretically, stacking multiple past frames can be useful, as this enables the network to infer information about its speed and angular velocity (which need at least 2 frames), and its acceleration and angular acceleration (which need at least 3 frames). These theoretical

considerations have been reinforced by prior work [103], and it has also been experimentally shown that stacking more than 3 frames does not yield considerable benefits, therefore I stacked 3 past frames together for every observation for the RL agents when not using rotary encoders. Since I used colored images, the final size of the input image observations became 160x80x9, as shown on Figure 20.

I did not apply any other preprocessing on the input images, such as tresholding certain colors or filtering, I let the neural network to discover what features are useful on its own.

The alternative solution for sensing speed and angular velocity is to use the rotary encoders that the latest edition of Duckiebots are equipped with. This option brings its own set of challenges, and I explore both options in this work.

4.2.4 Action Space

A differential robot is usually controlled by driving its motors on its sides at different speeds. In the case of the Duckiebot, one can control the duty cycles of the PWM (pulse width modulated) signals that drive its DC motors.

That means that the space of possible actions is two-dimensional and by each dimension it spans the range of $[-1.0; 1.0]$. This action space is rather large, and it also contains some actions that are not too useful, for example we do not want drive the vehicle drive backwards or to drive it much more slowly than what it is capable of.

As my initial experiments have shown, using a discrete action space is suboptimal, and since the task at hand is inherently continuous, I have chosen to use a continuous action space. I have followed prior work [103], and have defined a 1-dimensional action space. The only thing the agent can directly influence is its steering angle, which is then mapped to two target speeds of its two motors. These speeds are chosen to be as high as possible while still having a difference that is proportional to the steering angle. This has the effect that when taking sharp turns the car has to slow down to be able to provide the needed difference between the wheel speeds.

The exact derivation is described in Equation 4.6, where u_{nom} is the desired maximal duty cycle (nominal duty cycle), u_{avg} is the average duty cycle of the two motors (this depends on the desired steering angle), u_{left} and u_{right} are the duty cycles of the corresponding motors, and ϕ is the desired steering angle, while $clip(value, min, max)$ is a function that clips its input to be between a minimal and maximal value.

For small values ϕ can actually be interpreted as a steering angle in radians, however for larger values it should be interpreted as a scalar value that is proportional to the angular velocity of the vehicle, with $|\phi| = 1$ meaning that either one of the motors stops completely while the other one runs at full speed, meaning that the vehicle goes at half of its maximal speed.

$$
\begin{aligned}
u_{nom} &= 1.0 \\
u_{avg} &= \min(u_{nom}, \frac{1}{1+|\phi|}) \\
u_{left} &= clip(u_{avg}(1+\phi), -1, 1) \\
u_{right} &= clip(u_{avg}(1-\phi), -1, 1)
\end{aligned}
\tag{4.6}
$$

4.2.5 Reward Function

I have chosen to use a reward function that is physically motivated. In each timestep the reward of the agent is the speed at which it is progressing in its lane (with which speed it is completing the track). A more accurate description is that the reward is the speed of a virtual twin vehicle that moves exactly in the middle of the lane, is exactly parallel to it, and completes its route at the same rate as the actual car.

The exact formula of the reward function is shown in Equation 4.7, where v is the physical speed of the car, δ is the signed angle of the car and lane, r is the signed radius of the turn, c is the signed curvature of the turn ($c = 1/r$), and p is the signed distance of the car from the lane.

$$
R = v_{progress} = v \cdot cos(\delta) \cdot \frac{r}{r+p} = v \cdot cos(\delta) \cdot \frac{1}{1+cp} \tag{4.7}
$$

The formula can be understood in the following way: $v \cdot cos(\delta)$ is the component of the vehicle's speed that is parallel to the lane, $v \cdot cos(\delta)/(r+p)$ is the angular velocity of the vehicle in a turn, and $v \cdot cos(\delta)/(r+p) \cdot r$ is the circumferential velocity of the equivalent virtual vehicle in the turn, that is moving exactly on the middle of the lane.

This reward function has the advantage that it penalizes high angles and turns taken in the outer regions of the road, while it promotes high speeds and turns taken on the inner regions of the road. In practice the value is generally quite close to simply the speed of the vehicle, which has also been shown to be a plausible reward function [103].

When using this reward function however, care has to be taken to limit how much the car can leave its lane, otherwise it will tend to take left turns by going over to the other lane.

4.3 Dynamics Randomization in the Custom Simulator

I created the first prototype of the custom lane following environment in Matlab Simulink. During implementation I took care to only use simple mathematical functions and integrators based on physical considerations, to make the subsequent Python-based implementation simpler.

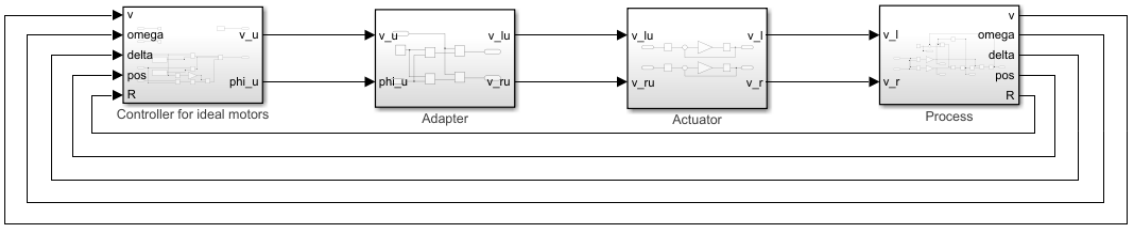

Figure 21: The high level block diagram of the Simulink-based simulation (after removing monitor objects)

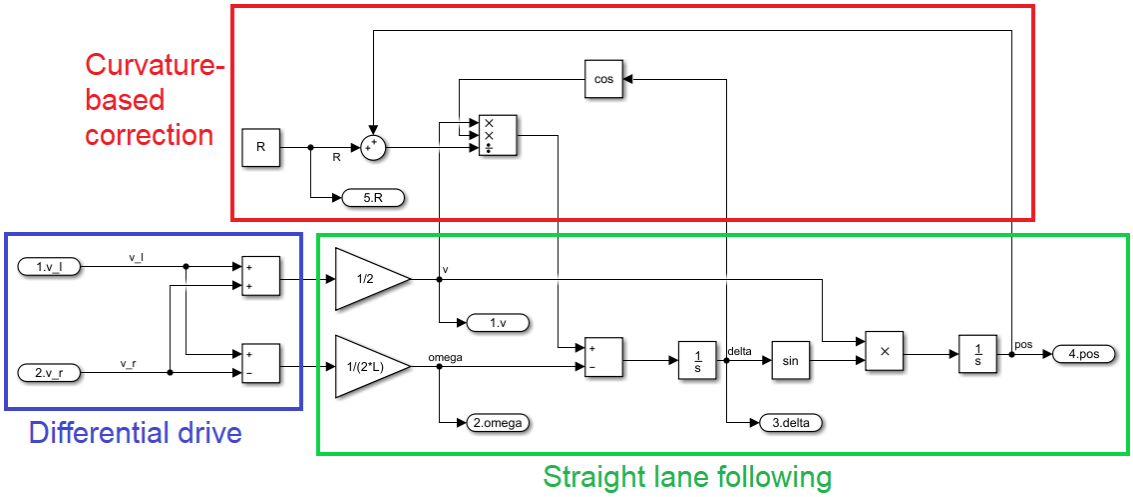

Figure 22: The process block of the Simulink-based simulation

The high level Simulink block diagram can be seen on Figure 21, and its core component is presented on Figure 22.

This helped me debug and correct my theoretical mistakes in the prototyping phase, and helped me to test the environment with baseline classical controllers. Only after having a working prototype did I start working on the Python-based implementation.

The Python-based implementation followed the Simulink model. I took care to ensure that the interface architecture of the environment follows the Duckietown Gym's one, and the default dynamics parameters were set to follow the Duckietown Gym in behaviour as much as possible.

The custom environment supports the method of uniform dynamics randomization, introduced in Section 2.3.3, dynamical parameters can have ranges and reference (nominal) values, to support training both with and without randomized dynamics.

The dynamics parameters and their default ranges and reference value are listed in Table 2. During dynamics randomization these are sampled at the start of every episode, except track parameters (track curvature and segment length), which are sampled more frequently: at the start of every new segment, which roughly corresponds to a tile: a track section with constant curvature.

Table 2: Ranges and reference values of dynamics parameters

Parameter	Reference value	Minimum	Maximum
Controller sample time [ms]	33	10	50
Wheel distance [cm]	16.0	14.0	18.0
Maximal motor speed [m/s]	0.6	0.5	0.7
Motor delay time [s]	0.1	0.08	0.16
Motor time constant [s]	0.233	0.2	0.3
Motor symmetry	True	False	True
Track curvature [1/m]	-	-8.0	8.0
Segment length [m]	-	0.25	0.75
Initial motor duty cycle [%]	-	0.0	100.0
Initial track angle [rad]	-	-0.15	0.15
Initial track position [cm]	-	-5.0	5.0

Table 3: A summary of all control agents trained in the custom environment, investigating dynamics randomization

Algorithm	**Input observation**	**Dynamics randomization**
STACK	3x stacked physical parameters	False
STACK DR	3x stacked physical parameters	True
ENC	1x physical parameters, angular velocity from encoder	False
ENC DR	1x physical parameters, angular velocity from encoder	True

The curvature of the track is the reciprocal of the radius of the arc at each point. This has the advantage, that straight segments can be represented with a curvature of 0 instead of a radius of infinity.

The algorithms trained in the custom environment, investigating dynamics randomization are summarized in Table 3.

4.4 Hyperparameters, Neural Architecture

During my work, I tried to use default hyperparameters wherever possible. I have extensively tuned manually the hyperparameters of the reinforcement learning algorithm (Proximal Policy Optimization [100]), however could not achieve meaningful improvements. The final hyperparameters I used can be found in Table 4, with unmentioned hyperparameters being left on library defaults.

For consistency, the supervised feature extractor is quite similar to the encoder of the variational autoencoder, with the only difference being the output dimensionality of their last layers.

The value and policy networks of the reinforcement learning agents have been kept default, i.e. as separate 2 layered fully connected networks with Tanh activations.

The used hyperparameters are listed in tables 4, 5, 6 and 7.

Table 4: Hyperparameter values used for the PPO algorithm

Name	Value
Optimization steps	64 * 2048 = 131 072
Learning rate	0.0003
Number of steps between updates	2048
Batch size	64
Optimization epochs	10
Time horizon (discount factor)	0.8 s (0.96)
Gradient clip range	0.2
Entropy coefficient	0.0 (std dev has been kept constant instead)
Initial log standard deviation	-1.2 (and kept constant)

Table 5: Encoder architecture

Layer	Output dimensions
Conv(kernel=3, stride=2, padding=1) + ReLU	width x 40 x 80
Conv(kernel=3, stride=2, padding=1) + ReLU	width x 20 x 40
Conv(kernel=3, stride=2, padding=1) + ReLU	width x 10 x 20
Conv(kernel=3, stride=2, padding=1) + ReLU	width x 5 x 10
Linear()	2 x latent dim or output dim

Table 6: Decoder architecture

Layer	Output dimensions
Linear() + ReLU	width x 5 x 10
Upsampling(scale=2) + Conv(kernel=3, padding=1) + ReLU	width x 10 x 20
Upsampling(scale=2) + Conv(kernel=3, padding=1) + ReLU	width x 20 x 40
Upsampling(scale=2) + Conv(kernel=3, padding=1) + ReLU	width x 40 x 80
Upsampling(scale=2) + Conv(kernel=3, padding=1) + Sigmoid	width x 80 x 160

Table 7: State representation learning hyperparameters

Name	Value
Learning rate	0.001
Number of epochs	20
Width	64
VAE latent dimensions	8 (2 for latent visualization)
VAE standard deviation parametrization	as the log of the standard deviation

Chapter 5

Evaluation

5.1 Evaluation in the Simulator

5.1.1 Evaluation Metrics

I used the following metrics to monitor the performance of lane following agents during and after the training:

- The mean absolute angle between vehicle and the lane ($\bar{\delta}$)
- The mean absolute distance between vehicle and the center of the lane ($\bar{p}$)
- The mean speed of the vehicle ($\bar{v}$)
- The mean magnitude of the component of the speed that is parallel to the lane ($\bar{v}_f$)
- The mean reward per timestep, which is equivalent with the mean completion speed ($\bar{R}$)
- The average length of the evaluation episodes ($\bar{t}$) in timesteps which can be used to calculate the average completion rate by dividing it with the maximal length of the episodes

The following metrics were only used in the single episode evaluation setting:

- The maximal absolute angle between vehicle and the lane
- The maximal absolute distance between vehicle and the center of the lane
- The median signed distance between vehicle and the center of the lane (to detect asymmetries)

The averages have been calculated over the whole process of the evaluation, which generally consisted of multiple episodes: 10-20 during training, 50 during final evaluation.

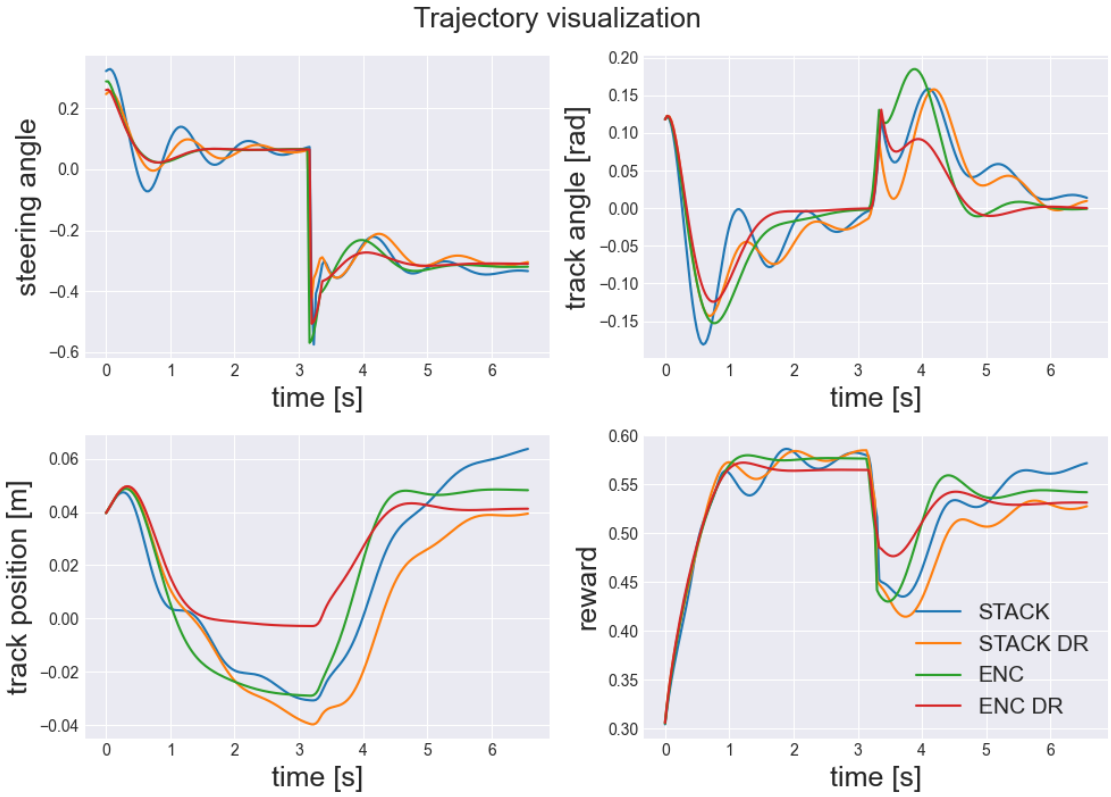

Figure 23: Policy visualization using one of its evaluation trajectories. At 3 seconds the policy reaches a new track segment, this is what causes the sudden jump in the visualized values.

5.1.2 Visualization of Controllers

Being able to understand and reason about the behaviours of trained lane following agents was crucial to find mistakes and improve both the system and the algorithms. Visualization was a useful tool to achieve that, which I employed in three distinct ways.

Visualization via Trajectories

The most straightforward solution for visualization is to store the states, actions and rewards during an evaluation episode, and plot them as a time-series. This gives us a picture about the general behaviour of the agents, parts of which would be much more difficult to infer from the metrics mentioned in the previous section.

The exacts behaviour however is hard to understand from a single trajectory, and it is also gets tough to understand at a glance after an episode length.

A trajectory visualization example is shown on Figure 23, with 4 different agents trained with reinforcement learning. We can see that in this case, all agent were able reduce their initial position- and angle errors, and were also able to stay on track after reaching a new

track segment. Segments with different curvatures require different steering angle offsets for steady driving, which has to be readjusted for each new segment.

Though experimented for a short while with agents using discrete action spaces, they are omitted by default from these visualizations as they decrease the clarity of the plots, because of the discrete nature of their actions.

Policy Visualization by Probing the Observation Space

The idea of this visualization method is to select a reference point in the observation space (let it be in our case at the speed being the maximal value and all other physical parameters at zero). Let us probe all observation dimensions separately from this point to between their minimal and maximal value, and lets plot the corresponding actions.

Mathematically, for each dimension of the observation space we choose the straight line that is parallel to the dimension's unit vector and intersects the reference observation, and we plot the values of actions corresponding to the points of this line. As an extension, we can plot the normalized action histograms as well.

The technique can be used for classical linear agents as well, for their case the observation-action functions will be linear up to action saturation.

Please note that the method only works for visualizing non-recurrent agents, for example agents with stack observation would be affected by the order of probing, so these agents are only visualized by their normalized actions histograms.

A trajectory visualization example is shown on Figure 24, which shows that in this case, the learned non-recurrent controller are approximately linear, and are most sensitive to angle error, as this is the only case in which their observation-action function saturates.

In my experience this method can characterize control policies quite well, seeing irregular functions in the policy visualization usually means that the agent is not performing well yet.

This technique could also be used to calibrate classical agents to mimic converged reinforcement learning agents (provided that their action functions stay approximately linear), which could give further insights into how these agents work.

Visualization via Training Curves

Another, commonly used method is to plot the evaluation performance of agents over the course of their training, which can be used for example to estimate at a glance, whether a training has converged or it needs more training steps. I present such an example on Figure 25 with ± 1 standard deviation range colored on the background.

This example shows the results of one of my preliminary experiments where I compared RL algorithms to see which I should use. There are two takeaways one could make from this

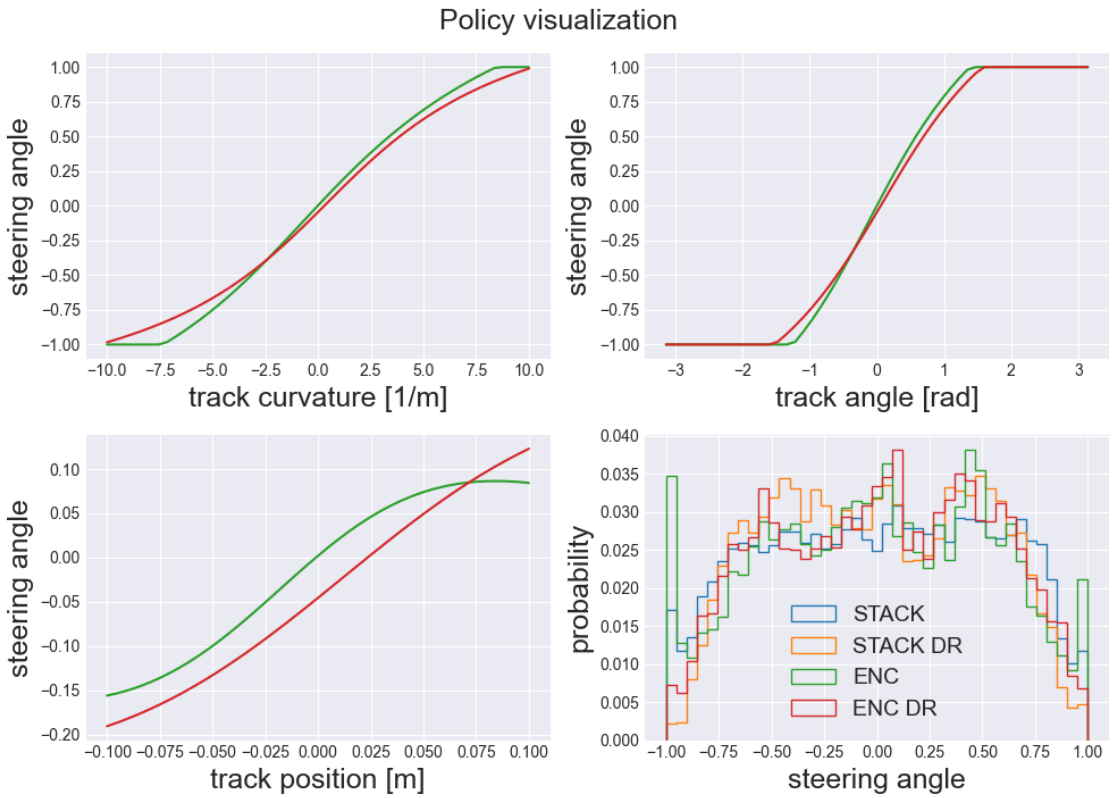

Figure 24: Policy visualization by observation space probing

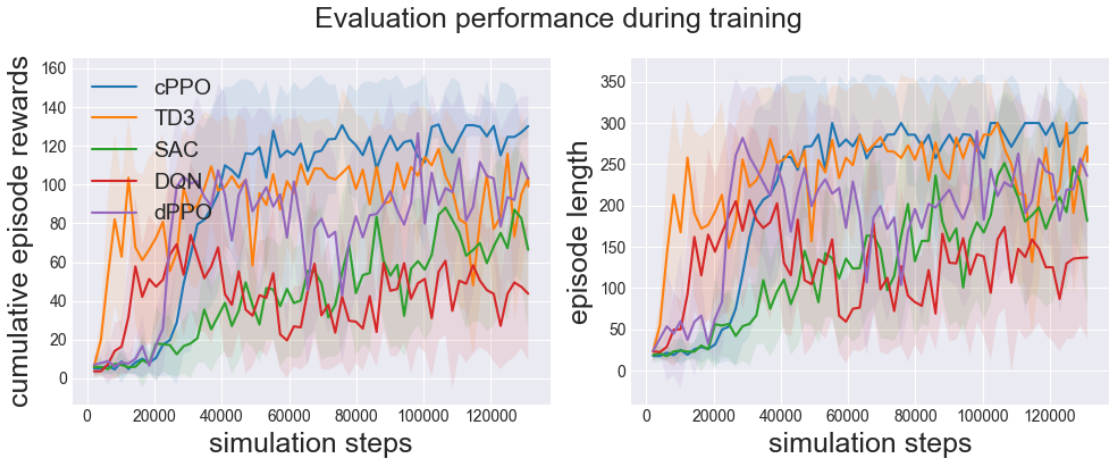

Figure 25: Comparison of RL algorithms in the custom lane following environment. cPPO and dPPO mean PPO algorithms with continuous and discrete action spaces, respectively.

visualization: on one hand, RL is noisy, there is no algorithm that consistently converges to a high performance with low variance.

On the other hand, PPO with continuous action space seemed to outperform the others and also seemed to be the most stable, while DQN performed the worst in this experiment.

5.2 Evaluation in Reality

5.2.1 Implementing Real Inference

Real world evaluation requires implementing an input preprocessing and output postprocessing pipeline that exactly mirrors the one that the model was trained on. One could manually implement it but that can be error-prone, so I went with a different solution instead.

I used a "dummy" empty environment, which represented reality. If new observations arrived, I would manually copy them into this environment. I also added all fields to it that its wrappers required. I could then wrap all the observation and action wrappers around it that I used for training, guaranteeing that the data pipeline is identical across simulation and reality.

5.2.2 Measurement of Robot Characteristics

To increase the probability of sim-to-real transfer, I had to make sure that the dynamics parameters used in training are close enough to reality so that they do not hinder real performance.

Measuring some characteristics of the robot was actually challenge, and required engineering. The first step was to to read and calibrate the parameters of the encoders (rolling it for a given distance and monitoring encoder ticks) to be able to measure speed at both wheels, which could be used to calculate vehicle speed and angular velocity.

I could then measure the vehicle's duty cycle - speed and duty cycle - angular velocity characteristics by holding the wheels in the air, but these were values without torque, which is not true in reality.

I then implemented an agent that uses the encoders to be able to drive in a straight line using proportional control on the orientation calculated from the traveled distances by each wheel.

I used then this straight going agent to measure the above mentioned characteristic under realistic load. The results are shown on Figure 26. For the measurements I calculated exponential moving averages over 1 second.

The ratio of speeds and angular velocities should be equal to the wheel distance, but in reality this is not the case because the motors have a higher torque load when turning.

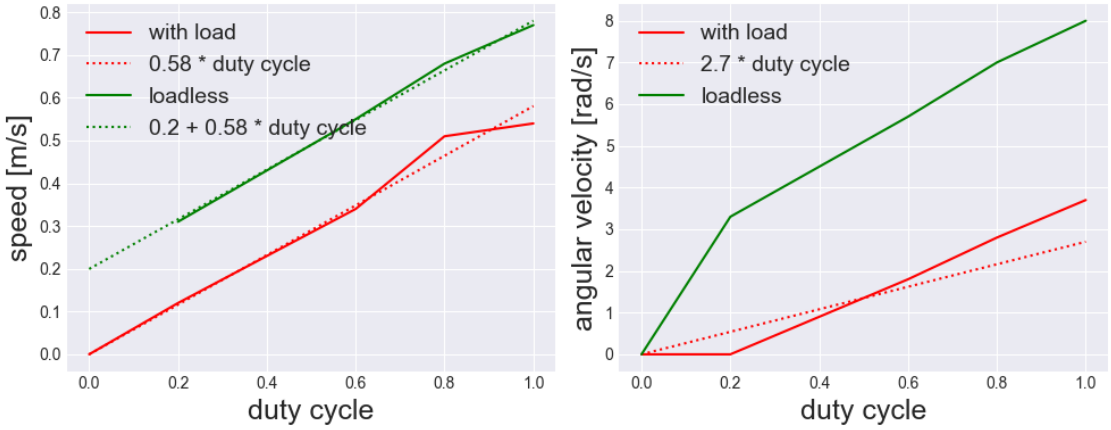

Figure 26: Measured and modeled vehicle dynamics

Table 8: Measured vehicle dynamics vs simulator defaults

Quantity	**Simulator**	**Measured**
maximum speed	0.6 m/s	0.58 m/s
virtual wheel distance	16.0 cm	21.5 cm
wheel radius	3.2 cm	3.3 cm
motor delay time	0.15 s	0.07 s
motor time constant	0.233 s	0.3 s

To model this effect, we can calculate a "virtual wheel distance" by calculating the ratio of these to quantities.

I also measured the motor delay time and the motor time constant, the latter one was acquired by measuring how much time it takes for it to reach the $1 - \frac{1}{e} = 0.632$-times its final velocity (without load), taking into account the motor delay time.

Comparing the values used by the simulator with what I measured we can see that though there are differences, the sum of motor delay time and motor time constant are actually quite close, and the other large difference, the virtual wheel distance might be caused by the fact that I measured it at a small turning radius, causing me to slightly overestimate it because of the high torque load.

Based on these considerations, and to keep my results comparable, I decided to trust the simulator defaults, and used them for my trainings.

5.2.3 Sim-to-Real Differences

In this subsection I list the most important differences that are neglected in the simulator and can cause issues in reality.

The vehicle is assumed to be weightless and frictionless (or that friction is compensated by an offset voltage on the motors, which seems to be true). This assumption causes

troubles at low duty cycles (<40%), where the vehicle is much slower to turn because of the constant friction.

The frames per second (FPS) is constant, which is very far from the truth: it not only fluctuates heavily, but it also depends on the host computer and size the neural network used. I recommend using a cable for networking on the host computer. This is actually an issue as there is no way of knowing and replicating FPS-s of other locations.

The wheels do not slip. This assumption causes issues at high duty cycles (>60%) where the slipping can be severe, but it can also happen at the 50% duty cycle that used for testing.

The track tiles are not textured (they actually are) and have a non-glossy surface (mostly true).

5.2.4 Sim-to-Real Adaptation

In simulator, agents are usually trained at maximum speed (100% duty cycle, 0.6 m/s), however in reality this speed is far from achievable under realistic driving conditions, mainly because of the slipping of the wheels. A popular heuristic among prior works [103] in our team is to multiply all duty cycles output by the network with a constant, 0.5 for example. Though this is a simple thing to do, it is not at all obvious from a controls perspective what its effects actually are, and whether doing this is theoretically sound. I would like to discuss that here.

Left and right motor duty cycles (u) transfer to a speed (v) and angular velocity (ω) the following way using the distance of wheels (L) (assuming either no friction and torque load, or that these are compensated using an offset voltage):

$$\begin{aligned} v_{left} &= u_{left} * v_{max} \\ v_{right} &= u_{right} * v_{max} \end{aligned} \tag{5.1}$$

$$\begin{aligned} v &= \frac{v_{left} + v_{right}}{2} = v_{max}\frac{u_{left} + u_{right}}{2} \\ \omega &= \frac{v_{left} - v_{right}}{L} = v_{max}\frac{u_{left} - u_{right}}{L} \end{aligned} \tag{5.2}$$

This then transfers to a turning radius (r) in the following way:

$$r = \frac{v}{\omega} = \frac{L}{2}\frac{u_{left} + u_{right}}{u_{left} - u_{right}} \tag{5.3}$$

This means that by multiplying all duty cycles with a constant value, the turning radius will remain unchanged, and the vehicle will turn on the same path as before (assuming negligible time delays), just more slowly. If the time delays are not negligible, then this makes the task easier, as the vehicle moves at a slower speed, while having the same action delay time.

I used these considerations for transferring all my models to the real world, which were trained at 100% duty cycle and tested at 50%.

I also experimented with training at lower speeds, but without too much success. My intuition is that the higher the speed, the tougher dealing with the actuation delays get, meaning that the task is the hardest on high speeds. For sim-to-real transfer we want to force the agent to solve the toughest possible task, if it is slightly too conservative in the real world, it is usually not a problem.

Chapter 6

Results

In this chapter I will present numerous results measured under different conditions both in simulators and in reality, I will also summarize the results and draw attention to the most important takeaways.

6.1 Visual Domain Randomization Results

6.1.1 State Representation Learning Results

Representation Learning Metrics

On tables 9 and 10 I present the quantitative results of the perception module trainings.

Reconstructions of Simulator Images

In this section I present visualizations from the some of the VAE- and DVAE-based models to show the visual quality of the reconstructions in specific circumstances.

Figure 27 shows target and reconstructed image pairs for VAE RND, DVAE CAN and DVAE SEG. Based on these reconstructions all models seem to capture those parts of the images that are relevant to driving.

Table 9: Mean absolute validation errors of the supervised representation learning algorithms

Algorithm	**Angle error [deg]**	**Position error [cm]**	**Curvature error [1/m]**
SUP	3.3	1.54	0.44
SUP RND	4.1	1.69	0.54
SUP CAN	3.6	1.47	0.49
SUP SEG	3.6	1.58	0.51

Table 10: Validation metrics of the VAE-based representation learning algorithms

Algorithm	**Mean squared error [nats/dim]**	**KL-divergence [nats/dim]**	**Invariance KL-divergence [nats/dim]**
VAE	0.125	5.5	-
VAE RND	0.160	5.4	-
VAE CAN	0.126	3.9	8.3
VAE SEG	0.115	3.9	12.3
VAE REAL	0.087	4.9	-
DVAE CAN	0.132	5.2	-
DVAE SEG	0.149	5.3	-

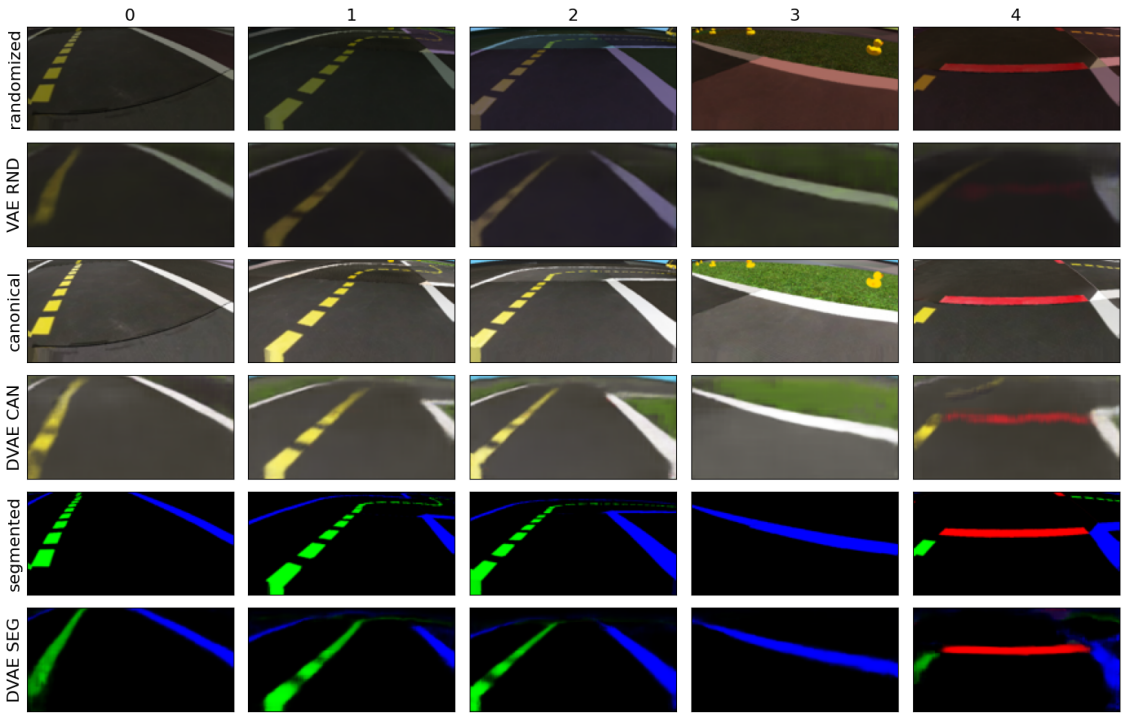

Figure 27: Reconstructions of simulator images with VAE-based methods

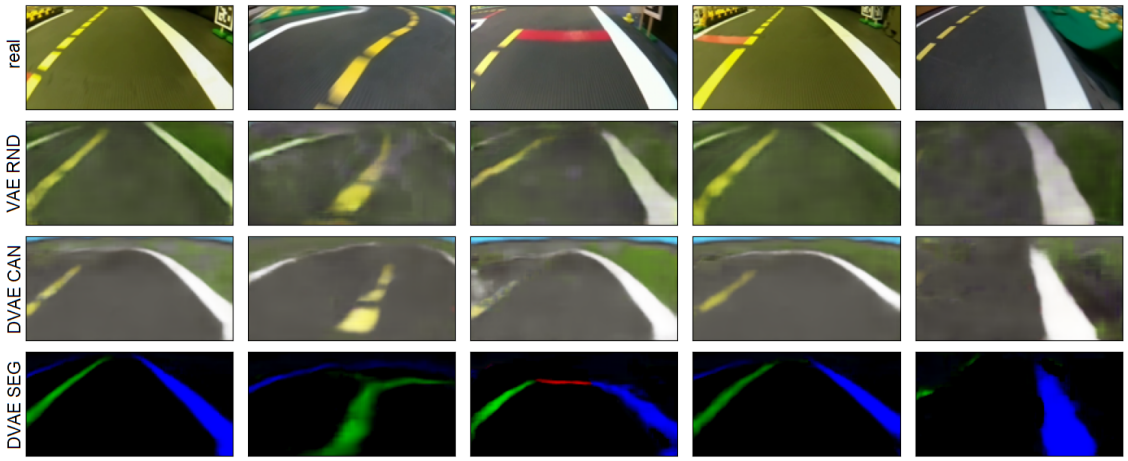

Figure 28: Reconstructions of real images with VAE-based methods

Reconstructions of Real Images

One could also inspect the outputs of the networks visually when using the networks on real images, and draw conclusions from that. The general quality of reconstructions that I have seen from the models when they have a real image as input, is lower. With the blurry spot-like structures are more prevalent.

However, the models still manage to give meaningful reconstructions of these real images, which means at least that they can extract some information, which could be useful for the downstream RL agent when applying them in the real world.

Visualization of the Latent Space

I have trained a DVAE CAN with a two-dimensional latent space, to be able to visualize it on a 2D-grid with a grid of generated images from the corresponding latent codes. The visualization can be seen on Figure 29.

We can generally see straight road sections, with differing lane angles and positions, which supports the assumption that these representations can be useful for solving the lane following task. Towards the top left corner we can see generated images with unclear structures. My explanation is that because of the highly reduced latent capacity, the network was forced there to interpolate between very different images.

6.1.2 Performance in Simulation

Table 11 shows the results of state representation learning algorithms with their default control modules (trained using their representations of their input image type in the Duckietown Gym), evaluated in simulation.

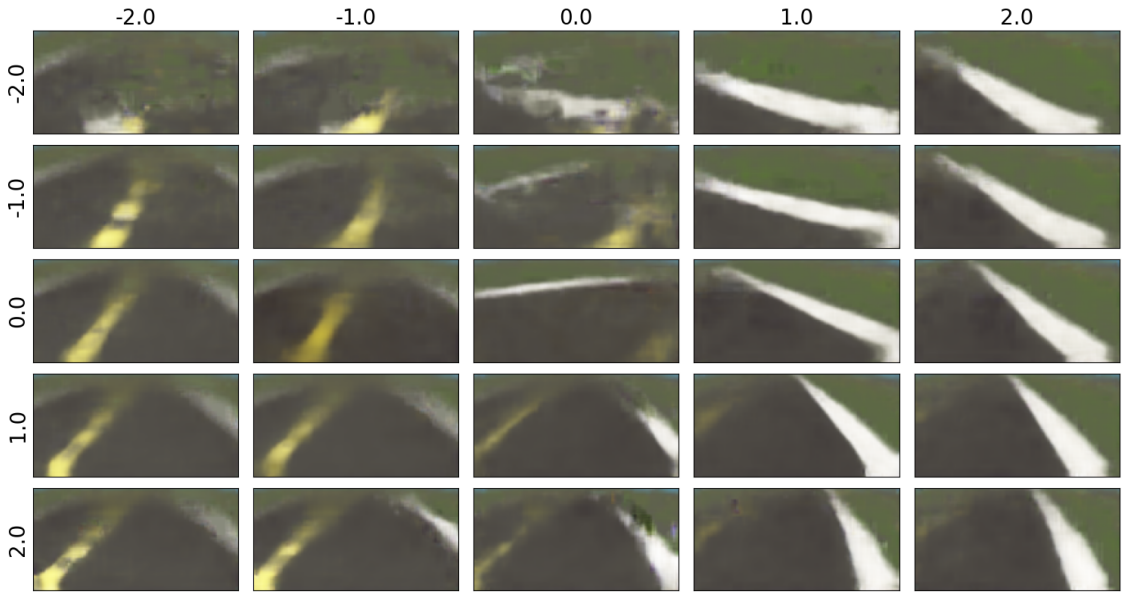

Figure 29: The latent space of a 2-dimensional denoising variational autoencoder, sampled from a grid

The most interesting finding from this line of experiments is that algorithms that did not use canonical (non-randomized) images for their training (algorithms not highlighted in bold), do not perform well in this setting, with the positive exception being VAE REAL, which might have been able to generalize better because of the real images and the extra data augmentation.

This result shows that visual domain randomization has its limits, it increases the coverage of visuals of the simulator, but does not guarantee generalization. My opinion is that the visual domain randomization in the Duckietown Gym might be too extreme, visual inspection of the images show that the images are usually much darker in comparison to the canonical images.

A strong negative exception is E2E. As one can see on tables 11, 12 and 13, end-to-end reinforcement learning does not manage to perform well in any of the evaluations. I believe this might not be caused by the paradigm itself, because others were able to achieve strong results using purely end-to-end RL [103], but the choice of RL library. I chose Stable-Baselines3 because of its ease of use, but my general experience with it is that while it can train fully connected policies to high performance, it has issues with training convolutional neural networks, which is something that prior work noticed as well [103]. This could explain the weaker end-to-end RL results.

Table 12 shows the results of state representation learning algorithms with their default control modules (trained using their representations of their input image type in the Duckietown Gym), evaluated in simulation.

This line of experiments show that all investigated state representation methods, that are trained with some sort of visual domain randomization, are capable of solving the self-

Table 11: Evaluation results in the **simulator without visual domain randomization**. Algorithms with perception modules trained using non-randomized images are highlighted in bold, completion ratios above 70% are underlined. See Table 1 for an overview of these algorithms.

Algorithm	**Absolute angle error [deg]**	**Absolute position error [cm]**	**Completion speed [m/s]**	**Completion ratio [%]**
E2E	11.5	3.9	0.40	30.0
E2E RND	12.9	3.2	0.35	29.3
SUP	6.0	2.5	0.47	<u>88.6</u>
SUP RND	9.0	3.4	0.44	39.2
SUP CAN	6.6	2.0	0.48	<u>90.8</u>
SUP SEG	6.8	2.0	0.47	52.1
VAE	6.4	2.5	0.48	54.4
VAE RND	8.7	2.4	0.46	52.9
VAE CAN	7.8	2.5	0.51	<u>83.4</u>
VAE SEG	8.1	2.6	0.47	45.1
VAE REAL	7.6	2.8	0.47	<u>73.9</u>
DVAE CAN	15.4	3.4	0.32	14.0
DVAE SEG	10.0	3.5	0.43	30.7

driving task in the simulator with randomized visuals, as all of them (with the exception of DVAE SEG) gets an 80% or higher completion ratio.

6.1.3 Performance in Reality

Table 13 shows the results of state representation learning algorithms with their default control modules evaluated in reality.

The most important learning from the real tests was the following: none of those methods transfered to reality that did not use some form of visual domain randomization. This shows that visual domain randomization is a useful tool and an be crucial for sim-to-real transfer.

The real testing also showed that a good portion of the randomized algorithms (SUP CAN, VAE RND, VAE CAN, VAE REAL) are capable of generalizing to the real world, and to drive almost without errors (max. 2 resets over 60 seconds in both directions). On average, VAE-based approaches seemed to outperform supervised representation learning approaches in these tests.

Both quantitatively and visually, VAE REAL outperformed the others, which suggests that being able to use real images for training might be more important than the promised advantages of the more sophisticated methods.

Interestingly, feature extractors using segmented images for training (SUP SEG, VAE SEG, DVAE SEG) seem to have a low performance across the real tests. This might mean

Table 12: Evaluation results in **simulator with visual domain randomization**. Algorithms with perception modules trained using randomized images are highlighted in bold, completion ratios above 70% are underlined. See Table 1 for an overview of these algorithms.

Algorithm	**Absolute angle error [rad]**	**Absolute position error [m]**	**Completion speed [m/s]**	**Completion ratio [%]**
E2E	10.1	3.8	0.42	23.0
E2E RND	12.4	3.3	0.38	27.0
SUP	17.8	4.0	0.28	12.5
SUP RND	7.3	2.8	0.47	79.9
SUP CAN	6.6	2.0	0.48	84.0
SUP SEG	4.5	1.7	0.50	89.4
VAE	11.0	3.9	0.41	18.2
VAE RND	6.9	2.9	0.49	82.7
VAE CAN	7.7	2.5	0.50	82.3
VAE SEG	7.2	2.7	0.49	86.9
VAE REAL	6.0	2.5	0.50	88.1
DVAE CAN	6.8	2.9	0.50	91.3
DVAE SEG	6.7	3.1	0.50	60.9

that though segmentation is a useful prior, it might decrease the robustness of agents to the real world.

Evaluation of robustness in reality

Thank to the modularity of the solution, we can evaluate the robustness of agents to certain changes in their environments.

Table 14 shows the results of the two best methods from reality, when we increase their speed, and when we turn off the lights, simulating night driving, in which case the only sources of light will be the Duckiebots' LED lamps.

The speed increase is implemented by raising the base duty cycle of the motors from 50% to 60% raising the speed from 0.35 to 0.42 m/s. One can already see some slipping of the wheels at 50% duty cycle (which is not modeled in the simulator), but at 60% the robot still remains drivable. Above that the slipping is too severe, and I would not recommend using higher speeds than these.

Though we see a decreased performance in both cases, VAE CAN seemed to come out ahead of VAE REAL in terms of robustness in both cases.

For supervised feature extractors it is actually reasonable to think about swapping perception and control modules across algorithms, as all are trained to achieve the same results: estimating physical track parameters from images.

For these experiments I swapped the perception and control modules between the two best supervised state representation learning methods (SUP RND and SUP CAN). Table

Table 13: Evaluation results in **reality**. Algorithms with perception modules trained using real images are highlighted in bold, survival times greater than or equal to 20 are underlined. See Table 1 for an overview of these algorithms.

Algorithm	**Traveled tiles (outer)**	**Traveled tiles (inner)**	**Survival time (outer) [s]**	**Survival time (inner) [s]**
E2E	30	30	5.5	4.6
E2E RND	30	29	7.5	6.7
SUP	0	0	0.0	0.0
SUP RND	35	41	10.0	20.0
SUP CAN	41	44	60.0	30.0
SUP SEG	29	31	6.0	5.0
VAE	22	26	5.0	4.6
VAE RND	40	49	30.0	no resets
VAE CAN	43	48	no resets	no resets
VAE SEG	36	35	12.0	5.5
VAE REAL	44	51	no resets	no resets
DVAE CAN	0	0	0.0	0.0
DVAE SEG	37	37	8.6	12.0

Table 14: **Robustness** evaluation results in **reality**. Top half corresponds to agents driving at increased speeds (0.6 m/s instead of 0.5 m/s), bottom half corresponds to agents driving with room lights turned off. Survival times greater than or equal to 20 are underlined. See Table 1 for an overview of these algorithms.

Algorithm	**Traveled tiles (outer)**	**Traveled tiles (inner)**	**Survival time (outer) [s]**	**Survival time (inner) [s]**
VAE CAN	44	56	20.0	no resets
VAE REAL	41	50	20.0	15.0
VAE CAN	41	38	30.0	15.0
VAE REAL	34	30	8.6	5.5

Table 15: Robustness evaluation results in **reality**. Top half corresponds to agents with swapped perception and control modules between each other. Survival times greater than or equal to 20 are underlined. See Table 1 for an overview of these algorithms.

Perception	**Control**	**Traveled tiles (outer)**	**Traveled tiles (inner)**	**Survival time (outer) [s]**	**Survival time (inner) [s]**
SUP RND	SUP CAN	44	48	no resets	60.0
SUP CAN	SUP RND	44	43	no resets	60.0
SUP RND	SUP RND	35	41	10.0	20.0
SUP CAN	SUP CAN	41	44	60.0	30.0

15 shows the results. Interestingly, the swapped agents not only work well, but they work even better in the real environment. Tests in the simulator show a similar performance compare to the original counterparts.

This means that either the controllers do not overfit to the way these networks make mistakes, or that they make mistakes in the same ways which makes their controller modules compatible with each other.

6.2 Dynamics Randomization Results

6.2.1 Performance in Simulation

I also trained control modules in the custom lane following environment with and without dynamics randomization (see Table 3 for an overview of these algorithms), and transferred them to the Duckietown Gym using an ideal sensor for physical state.

The experiments shown in Table 16 led me to the following observations: dynamics randomization led to similar or slightly worse performance both in environments with and without dynamics randomization. This is an unexpected result, and I will discuss it further in Chapter 7.

One can also see that controllers trained with dynamics randomization led to lower absolute angle errors in all cases, which signals a driving behaviour with less oscillations, which means that it can still have a positive effect by forcing the agent to drive more conservatively.

Another noteworthy result is that encoder-based agents seem to be working better, while also learning faster (Figure 30) in this case. This is in line with what I saw with my tests with classical controllers, i.e. that using encoders is better than frame stacking, while all the RL agents trained in the Duckietown Gym worked better with frame stacking than with encoders in contrast.

My last observation is based on the learning curves (Figure 30: using dynamics randomization slowed down learning in both cases, probably by making the gradient estimates

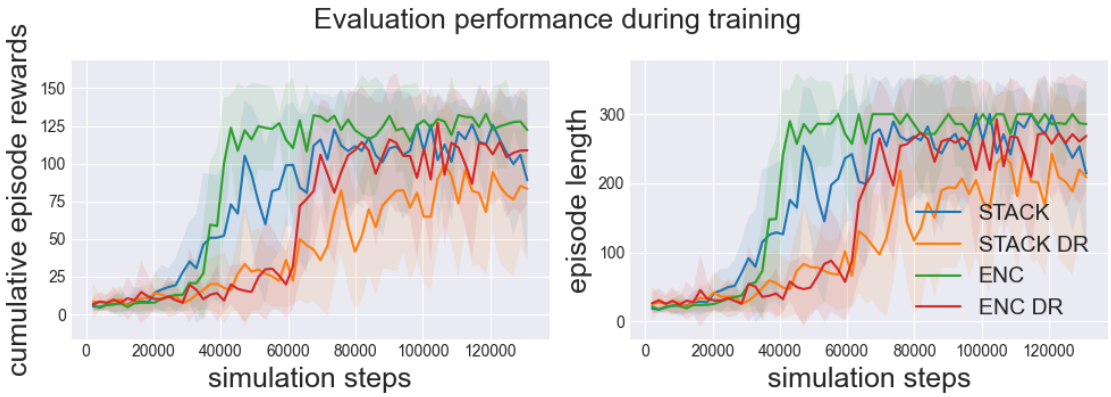

Figure 30: Evaluation performance over the course of the training

Table 16: Evaluation results in the **simulators**. Top third: custom line following simulator without dynamics randomization, center: with dynamics randomization, bottom third: Duckietown Gym with an ideal state sensor. Completion ratios above 70% are underlined. See Table 3 for an overview of these algorithms.

Control	**Absolute angle error [deg]**	**Absolute position error [cm]**	**Completion speed [m/s]**	**Completion ratio [%]**
STACK	11.4	3.0	0.43	87.9
STACK DR	9.7	3.2	0.42	76.4
ENC	7.5	2.9	0.44	94.2
ENC DR	6.8	2.4	0.44	90.4
STACK	13.6	3.3	0.39	71.8
STACK DR	10.0	3.1	0.41	72.4
ENC	9.0	3.3	0.42	93.8
ENC DR	8.8	2.9	0.42	86.0
STACK	12.7	2.8	0.43	98.5
STACK DR	10.5	2.8	0.45	96.1
ENC	6.3	1.9	0.49	98.1
ENC DR	5.2	2.2	0.49	99.9

have higher variance, and through that reducing sample efficiency. These are the tradeoff-s of dynamics randomization that I saw.

6.2.2 Performance in Reality

The experiments shown in Table 17 show that these controllers can also perform robustly in the real world if attached on top of well working supervised perception modules (SUP RND, SUP CAN). That way we get models whose perception modules have been trained on an offline dataset, while their control modules have been trained in a custom lane following environment, meaning that they never explicitly interacted with the Duckietown Gym and still were able to learn to drive reasonably in the real world.

Table 17: Evaluation results in **reality**. Survival times greater than or equal to 20 are underlined. See Table 3 for an overview of these algorithms.

Perception	**Control**	**Traveled tiles (outer)**	**Traveled tiles (inner)**	**Survival time (outer) [s]**	**Survival time (inner) [s]**
SUP RND	STACK	30	34	7.5	<u>60.0</u>
SUP RND	STACK DR	34	44	8.6	<u>60.0</u>
SUP RND	ENC	45	40	<u>no resets</u>	15.0
SUP RND	ENC DR	41	42	<u>30.0</u>	20.0
SUP CAN	STACK	31	41	8.6	<u>60.0</u>
SUP CAN	STACK DR	35	38	10.0	12.0
SUP CAN	ENC	42	39	<u>no resets</u>	12.0
SUP CAN	ENC DR	43	36	<u>60.0</u>	8.6

I would also like to note here the interesting finding, that while the STACK and STACK DR control modules worked well in the inner loops in reality (mostly just 1 reset per 60 seconds), ENC and ENC DR agents worked only on outer loops (more smooth left turns), while being much worse on inner loops (more sharp right turns).

6.3 Results on the AI Driving Olympics

During my work on this thesis I have participated twice in the AI Driving Olimpics (AIDO), Urban League [104]: first on AIDO 5 in 2020, second on AIDO 6 in early december of 2021, both organized as a competition track on the NeurIPS conference.

On AIDO 5 I managed to achieve second place in the simulator-based Lane Following challenge, and first place in the real world Lane Following challenge. That year I submitted preliminary versions of some of those algorithms that I described here. That year all solutions of our team were somewhat unstable due to the introduction of a new version of the Duckiebot, which we did not have available at that time for real testing.

This year, on AIDO 6, I submitted imitation learning-based solutions to the finals that I experimented with earlier, due to software issues in my reinforcement learning stack at that time, which made my reinforcement learning-based agents less reliable during real testing. This year I got third place in the Lane Following with other Vehicles, and fourth place in the Lane Following with Intersections challenge, both evaluated in the real world.

Working on these tasks was a challenging, but very rewarding experience. Based on the final results I achieved by the completion of this thesis, I am looking forward to future editions of the AI Driving Olympics.

Chapter 7

Conclusion

7.1 Conclusion on Domain Randomization

Based on the overview of real testing results, presented in Chapter 6, my conclusions are the following on the topic of domain randomization.

In this thesis I implemented and tested 11 state representation learning + 2 end-to-end reinforcement learning methods for the task of autonomous driving. Of the 8 methods trained with some form of visual domain randomization, 4 transferred successfully to the real world and achieved a performance of maximum 2 resets per 60 seconds of driving in both directions. None of the agents using perception modules without visual domain randomization succeeded in that same task. I have also showed that some of these agents are robust to higher speeds, driving in the dark, and swapped controller modules.

I believe this evidence is sufficient to support the claim that visual domain randomization is crucial for sim-to-real transfer, and is a powerful tool for helping reinforcement agents generalize better to be able to work in the real world.

I also evaluated 4 control modules in the real world in 8 different combinations all-in-all, and found that dynamics randomization has a more subtle effect on the agents' performance: it can help with smoother driving, however it did not affect performance much while it made the training procedure less sample-efficient.

In my opinion, dynamics randomization can be a useful tool as well, however I would recommend careful consideration of whether it is worthwhile to use it. For the lane following task, my conclusion is that it is enough to train in a simulator that is a slightly pessimistic model of reality, which can force the model to drive conservatively. I believe that this should be enough for sim-to-real transfer from a dynamics point of view.

7.2 Future Work

Improved dataset coverage

During the real world tests I noticed a peculiar behaviour: some of the models tended to drift towards the center line, sometimes even drifting to the other line, especially during turns. After a point, maybe when losing sight of the center line, they started spinning to the left, even though there still was enough information available from the left lane to know that a small right steering could correct the situation.

I believe these behaviours were caused by a poor coverage of these cases in the offline dataset I gathered. Since the dataset was gathered over the course of training an (end-to-end) reinforcement learning agent, it was unable to drive in the other lane, because in that case the episode would terminate.

Rebuilding the dataset improving the coverage of these edge cases could lead to the agents behaving reasonably in these situations, improving real world performance essentially for free.

Using real images for state representation learning

During real evaluation I was pleasantly surprised with how well models using invariance regularization for indirect visual domain randomization worked. Based on these results I believe it could be an interesting future avenue to explore further unsupervised uses of real images during training.

Further investigation of agents using wheel encoders

I believe that stacking frames introduces a sample-rate dependence to the system, combined with a higher computational demand. In my opinion wheel encoders provide useful information, and further utilizing them should be a promising future direction.

7.3 Summary

The main task of this work was investigate domain randomization methods for deep reinforcement learning, with a focus on self-driving environments.

During my work, I overviewed and categorized the related literature, and identified methods that could be implemented into a common framework. I chose a modular approach to the task, separating perception from control, allowing me to investigate visual domain randomization and dynamics randomization independently.

I implemented 4 visual domain randomization algorithms from the literature with an additional novel one, and tested them using state representation learning and reinforcement

learning. To achieve that, I gathered a dataset of 200.000 simulated data points, each containing 3 different renderings of the same scene and also physical labels for supervised representation learning, along with 19.000 real data points only containing images without labels.

To be able to evaluate the effects of dynamics randomization, I designed and built a lightweight lane following simulator to be able to investigate the control task without the perception task. During implementation I kept the two simulator environments compatible and aligned in dynamics.

I have trained 2 reinforcement learning agents using end-to-end reinforcement learning, 11 using state representation learning, and 4 more control agents using reinforcement learning in the custom simulator. I evaluated the agents using both simulators and in the real world as well. I presented results of 25 real measurements about the behaviours of agents in the real world.

I identified possible bottlenecks and proposed promising directions for future improvements. Finally I documented and summarized my results and conclusions.

Acknowledgement

The work presented in this thesis has been supported by Continental Automotive Hungary Ltd.

Bibliography

[1] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[2] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[3] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

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

[5] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, pp. 2672–2680, 2014.

[6] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of machine learning research*, vol. 3, pp. 1137–1155, 2003.

[7] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 6645–6649, IEEE, 2013.

[8] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[9] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[10] S. Linnainmaa, "Taylor expansion of the accumulated rounding error," *BIT Numerical Mathematics*, vol. 16, no. 2, pp. 146–160, 1976.

[11] P. J. Werbos, "Applications of advances in nonlinear sensitivity analysis," in *System modeling and optimization*, pp. 762–770, Springer, 1982.

[12] A. Cauchy, "Méthode générale pour la résolution des systemes d'équations simultanées," *Comptes rendus de l'Académie des Sciences*, vol. 25, no. 1847, pp. 536–538, 1847.

[13] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[14] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *European conference on computer vision*, pp. 630–645, Springer, 2016.

[16] M. Telgarsky, "Benefits of depth in neural networks," *arXiv preprint arXiv:1602.04485*, 2016.

[17] K. Hornik, M. Stinchcombe, H. White, *et al.*, "Multilayer feedforward networks are universal approximators.," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[18] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*, vol. 1. MIT press Cambridge, 2016.

[19] G. F. Montufar, R. Pascanu, K. Cho, and Y. Bengio, "On the number of linear regions of deep neural networks," in *Advances in neural information processing systems*, pp. 2924–2932, 2014.

[20] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen," Master's thesis, Technische Universität München, 1991.

[21] Y. LeCun, Y. Bengio, *et al.*, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, 1995.

[22] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*, pp. 818–833, Springer, 2014.

[23] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[24] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," *arXiv preprint arXiv:2002.05709*, 2020.

[25] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[26] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241, Springer, 2015.

[27] C. Szegedy, A. Toshev, and D. Erhan, "Deep neural networks for object detection," in *Advances in neural information processing systems*, pp. 2553–2561, 2013.

[28] P. Baldi and K. Hornik, "Neural networks and principal component analysis: Learning from examples without local minima," *Neural networks*, vol. 2, no. 1, pp. 53–58, 1989.

[29] M. Ranzato, C. Poultney, S. Chopra, and Y. L. Cun, "Efficient learning of sparse representations with an energy-based model," in *Advances in neural information processing systems*, pp. 1137–1144, 2007.

[30] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio, "Contractive auto-encoders: Explicit invariance during feature extraction," in *Proceedings of the 28th International Conference on Machine Learning, ICML 2011*, 2011.

[31] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[32] D. J. Rezende, S. Mohamed, and D. Wierstra, "Stochastic backpropagation and approximate inference in deep generative models," *arXiv preprint arXiv:1401.4082*, 2014.

[33] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proceedings of the 25th International Conference on Machine Learning*, pp. 1096–1103, 2008.

[34] G. Alain and Y. Bengio, "What regularized auto-encoders learn from the data-generating distribution," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3563–3593, 2014.

[35] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, P.-A. Manzagol, and L. Bottou, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion.," *Journal of machine learning research*, vol. 11, no. 12, 2010.

[36] D. Pathak, P. Krahenbuhl, J. Donahue, T. Darrell, and A. A. Efros, "Context encoders: Feature learning by inpainting," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2536–2544, 2016.

[37] J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," *arXiv preprint arxiv:2006.11239*, 2020.

[38] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[39] K. He, X. Chen, S. Xie, Y. Li, P. Dollár, and R. Girshick, "Masked autoencoders are scalable vision learners," *arXiv preprint arXiv:2111.06377*, 2021.

[40] A. Van Den Oord, O. Vinyals, *et al.*, "Neural discrete representation learning," in *Advances in Neural Information Processing Systems*, pp. 6306–6315, 2017.

[41] A. Vahdat and J. Kautz, "Nvae: A deep hierarchical variational autoencoder," *arXiv preprint arXiv:2007.03898*, 2020.

[42] Anonymous, "Very deep {vae}s generalize autoregressive models and can outperform them on images," in *Submitted to International Conference on Learning Representations*, 2021. under review.

[43] S. Gur, S. Benaim, and L. Wolf, "Hierarchical patch vae-gan: Generating diverse videos from a single sample," 2020.

[44] C. Li, X. Gao, Y. Li, X. Li, B. Peng, Y. Zhang, and J. Gao, "Optimus: Organizing sentences via pre-trained modeling of a latent space," *arXiv preprint arXiv:2004.04092*, 2020.

[45] S. Zhao, J. Song, and S. Ermon, "Towards deeper understanding of variational autoencoding models," *arXiv preprint arXiv:1702.08658*, 2017.

[46] O. Rybkin, K. Daniilidis, and S. Levine, "Simple and effective vae training with calibrated decoders," *arXiv preprint arXiv:2006.13202*, 2020.

[47] I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinick, S. Mohamed, and A. Lerchner, "beta-vae: Learning basic visual concepts with a constrained variational framework," *International Conference on Learning Representations*, 2017.

[48] C. Doersch, "Tutorial on variational autoencoders," *arXiv preprint arXiv:1606.05908*, 2016.

[49] M. Sundermeyer, Z.-C. Marton, M. Durner, M. Brucker, and R. Triebel, "Implicit 3d orientation learning for 6d object detection from rgb images," in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 699–715, 2018.

[50] S. Sinha and A. B. Dieng, "Consistency regularization for variational auto-encoders," *arXiv preprint arXiv:2105.14859*, 2021.

[51] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[52] D. Silver, "Ucl course on reinforcement learning." `https://www.davidsilver.uk/teaching/`. Access date: 2021.12.19.

[53] R. Bellman, "A markovian decision process," *Journal of mathematics and mechanics*, pp. 679–684, 1957.

[54] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial intelligence*, vol. 101, no. 1-2, pp. 99–134, 1998.

[55] A. Y. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *Icml*, vol. 99, pp. 278–287, 1999.

[56] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke, "Sim-to-real: Learning agile locomotion for quadruped robots," *arXiv preprint arXiv:1804.10332*, 2018.

[57] O. M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, *et al.*, "Learning dexterous in-hand manipulation," *The International Journal of Robotics Research*, vol. 39, no. 1, pp. 3–20, 2020.

[58] Y. Lin, Y. Liu, F. Lin, P. Wu, W. Zeng, and C. Miao, "A survey on reinforcement learning for recommender systems," *arXiv preprint arXiv:2109.10665*, 2021.

[59] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nazi, *et al.*, "A graph placement methodology for fast chip design," *Nature*, vol. 594, no. 7862, pp. 207–212, 2021.

[60] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[61] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[62] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[63] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, *et al.*, "Dota 2 with large scale deep reinforcement learning," *arXiv preprint arXiv:1912.06680*, 2019.

[64] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[65] A. Bewley, J. Rigley, Y. Liu, J. Hawke, R. Shen, V.-D. Lam, and A. Kendall, "Learning to drive from simulation without real world labels," in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 4818–4824, IEEE, 2019.

[66] S. James and E. Johns, "3d simulation for robot arm control with deep q-learning," *arXiv preprint arXiv:1609.03759*, 2016.

[67] M. Wang and W. Deng, "Deep visual domain adaptation: A survey," *Neurocomputing*, pp. 135–153, 2018.

[68] F. Sadeghi and S. Levine, "Cad2rl: Real single-image flight without a single real image," *arXiv preprint arXiv:1611.04201*, 2016.

[69] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 23–30, IEEE, 2017.

[70] Z. Liu, X. Li, B. Kang, and T. Darrell, "Regularization matters in policy optimization," *arXiv preprint arXiv:1910.09191*, 2019.

[71] A. Y. Ng, "Feature selection, l 1 vs. l 2 regularization, and rotational invariance," in *Proceedings of the 21st International Conference on Machine Learning*, p. 78, 2004.

[72] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[73] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-real transfer of robotic control with dynamics randomization," in *2018 IEEE international conference on robotics and automation (ICRA)*, pp. 1–8, IEEE, 2018.

[74] I. Kostrikov, D. Yarats, and R. Fergus, "Image augmentation is all you need: Regularizing deep reinforcement learning from pixels," *arXiv preprint arXiv:2004.13649*, 2020.

[75] M. Laskin, K. Lee, A. Stooke, L. Pinto, P. Abbeel, and A. Srinivas, "Reinforcement learning with augmented data," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[76] J. Tremblay, A. Prakash, D. Acuna, M. Brophy, V. Jampani, C. Anil, T. To, E. Cameracci, S. Boochoon, and S. Birchfield, "Training deep networks with synthetic data: Bridging the reality gap by domain randomization," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 969–977, 2018.

[77] M. Aractingi, C. Dance, J. Perez, and T. Silander, "Improving the generalization of visual navigation policies using invariance regularization," *36th International Conference on Machine Learning, Workshop RL4RealLife*, 2019.

[78] R. B. Slaoui, W. R. Clements, J. N. Foerster, and S. Toth, "Robust domain randomization for reinforcement learning," *arXiv preprint arXiv:1910.10537*, 2019.

[79] K. Lee, K. Lee, J. Shin, and H. Lee, "Network randomization: A simple technique for generalization in deep reinforcement learning," in *8th International Conference on Learning Representations*, 2020.

[80] S. James, P. Wohlhart, M. Kalakrishnan, D. Kalashnikov, A. Irpan, J. Ibarz, S. Levine, R. Hadsell, and K. Bousmalis, "Sim-to-real via sim-to-sim: Data-efficient robotic grasping via randomized-to-canonical adaptation networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[81] Q. Vuong, S. Vikram, H. Su, S. Gao, and H. I. Christensen, "How to pick the domain randomization parameters for sim-to-real transfer of reinforcement learning policies?," *arXiv preprint arXiv:1903.11774*, 2019.

[82] B. Mehta, M. Diaz, F. Golemo, C. J. Pal, and L. Paull, "Active domain randomization," *arXiv preprint arXiv:1904.04762*, 2019.

[83] L. Pinto, J. Davidson, R. Sukthankar, and A. Gupta, "Robust adversarial reinforcement learning," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 2817–2826, JMLR. org, 2017.

[84] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson, "Learning latent dynamics for planning from pixels," *arXiv preprint arXiv:1811.04551*, 2018.

[85] T. Lesort, N. Díaz-Rodríguez, J.-F. Goudou, and D. Filliat, "State representation learning for control: An overview," *Neural Networks*, vol. 108, pp. 379–392, 2018.

[86] D. Hafner, T. P. Lillicrap, M. Norouzi, and J. Ba, "Mastering atari with discrete world models," in *International Conference on Learning Representations*, 2020.

[87] F. Zhang, J. Leitner, B. Upcroft, and P. Corke, "Vision-based reaching using modular deep networks: from simulation to the real world," *arXiv preprint arXiv:1610.06781*, 2016.

[88] D. Yarats, A. Zhang, I. Kostrikov, B. Amos, J. Pineau, and R. Fergus, "Improving sample efficiency in model-free reinforcement learning from images," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, pp. 10674–10681, 2021.

[89] M. Laskin, A. Srinivas, and P. Abbeel, "Curl: Contrastive unsupervised representations for reinforcement learning," in *International Conference on Machine Learning*, pp. 5639–5650, PMLR, 2020.

[90] D. Ha and J. Schmidhuber, "World models," *arXiv preprint arXiv:1803.10122*, 2018.

[91] N. Hansen, "The cma evolution strategy: A tutorial," *arXiv preprint arXiv:1604.00772*, 2016.

[92] B. Prakash, M. Horton, N. R. Waytowich, W. D. Hairston, T. Oates, and T. Mohsenin, "On the use of deep autoencoders for efficient embedded reinforcement learning," in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pp. 507–512, 2019.

[93] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *arXiv preprint arXiv:1801.01290*, 2018.

[94] A. Kendall, J. Hawke, D. Janz, P. Mazur, D. Reda, J.-M. Allen, V.-D. Lam, A. Bewley, and A. Shah, "Learning to drive in a day," in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 8248–8254, IEEE, 2019.

[95] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning.," in *ICLR (Poster)*, 2016.

[96] A. Beres, "Sample-efficient deep reinforcement learning with visual domain randomization," *Student Research Conference of Budapest University of Technology and Economics*, 2020. Access date: 2021.12.19.

[97] M. Chevalier-Boisvert, F. Golemo, Y. Cao, B. Mehta, and L. Paull, "Duckietown environments for openai gym." `https://github.com/duckietown/gym-duckietown`, 2018. Access date: 2021.12.19.

[98] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, "Stable baselines3." `https://github.com/DLR-RM/stable-baselines3`, 2019. Access date: 2021.12.19.

[99] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[100] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[101] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale

machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[102] D. Foundation, "About the duckietown platform." https://www.duckietown.org/about/platform. Access date: 2021.12.19.

[103] A. Kalapos, "Applying transfer learning to autonomous driving task," Master's thesis, Budapest University of Technology and Economics, 2020.

[104] D. Foundation, "Ai driving olympics." https://driving-olympics.ai/. Access date: 2021.12.19.