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

Sample-Efficient Deep Reinforcement Learning with Visual Domain Randomization

Author:

András Béres

Advisor:

Dr. Bálint Pál Gyires-Tóth
Róbert Moni

2020

Contents

Kivonat

Napjainkban az önvezető járművek témáját a tudományos közösség és a közvélemény részéről is nagy érdeklődés övezi. Mivel a mély tanulás területe eszközöket kínál a nagy mennyiségű szenzoradat feldolgozásához, a megerősítéses tanulás pedig a megfelelő döntések meghozatalában segíthet komplex, interaktív környezetek esetén, felhasználásuk az egyik módja lehetne annak, hogy az önvezetés feladatát megoldjuk. Vannak azonban nehézségek, amik hátráltatják ezeknek a módszereknek valós környezetben történő felhasználását.

Az egyik ilyen probléma a megerősítéses tanulás adatéhsége. Mivel a tanuláshoz felhasználható egyetlen jelzés az aktuális jutalom nagysága, rengeteg felfedezésre, kísérletezésre van szükség, hogy meghatározzuk a felhasznált háló paramétereinek megfelelő értékeit. Egy lehetséges megoldás a felügyelet nélküli tanulás, melynek során a bemeneti adatot úgy tanuljuk meg hatékonyabban reprezentálni, hogy olyan feladatokat oldunk meg, melyek megoldását már a nyers, címkézetlen adat is tartalmazza.

Mivel a mesterséges intelligenciát használó ágensek valós környezetben történő tanítása sokáig tart, ezért sokszor túl drága lenne, néha még veszélyes is, így előnyös lehet erre a célra szimulált környezeteket használni. Azonban a szimulátorok csak tökéletlen modelljei a valóságnak, ezért a valós alkalmazásukkor az ágensek teljesítménye általában jelentősen csökken. Ezt a jelenséget nevezzük a szimuláció és valóság közötti résnek. Egy lehetséges megoldás a véletlenszerű környezetek módszere, mellyel a tanítás során a szimuláció bizonyos paramétereit véletlenszerűen megváltoztatjuk, így kényszerítjük az ágenst arra, hogy robusztus legyen környezetének változásaival szemben. Ezáltal növelhetjük a valós alkalmazás sikerének esélyét, azonban így általában még több tanító adatra van szükség.

Dolgozatomban egyszerre alkalmazom a véletlenszerű környezetek módszerét a szimulátor kinézetének változtatására, és a felügyelet nélküli tanulást, melynek során a bemeneti képeket tömörítem majd rekonstruálom variációs autóenkóderek segítségével. Az így kapott tömörebb reprezentációt a felhasználva lehet a megerősítéses tanítást mintahatékonyabbá tenni.

Dolgozatom célja, hogy egy új módszert mutassak a két technika együttes alkalmazására, mellyel az előnyeik kölcsönösen megőrizhetők. Megvizsgálom, hogy miért lehet a naiv kombináció szuboptimális, és új megoldást javaslok a tapasztalatok alapján. Munkám fő ötlete az, hogy a módosított kinézetű bemeneti képek alapján ne önmagukat, hanem önmaguk módosítatlan (kanonikus) verzióját rekonstruáljuk.

A bemutatott megoldás lehetőséget nyújt a tanítást követő valós alkalmazás minőségének becslésére kvantitatív és vizuális módon, címkézetlen valós képek felhasználásával. Továbbá előzetes szakértői tudással is segíthetjük a rendszer tanulását úgy, hogy mi választjuk meg a kanonikus képek kinézetét.

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

Abstract

Recently the topic of self-driving cars has received great attention both from academia and the public. 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 problem is the sample-inefficiency of Reinforcement Learning. 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.

Since AI agents collecting large amounts of experience while interacting with the real world is usually too expensive and sometimes even dangerous, 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 can make a succesful transfer more probable, usually at the cost of requiring even more training samples.

I apply Visual Domain Randomization simultaneously with the Unsupervised Learning task of compressing input images to lower dimensional representations and reconstructing them with Variational Autoencoders. These compressed representations can be used as inputs to the reinforcement learning agent to make the training more sample-efficient.

The goal of this work is to propose a new technique to combine these methods while retaining their respective advantages. I inspect why the naive combination can be sub-optimal and propose a new method based on those findings. The main idea of this work is to reconstruct a non-randomized (canonical) version of the input image based on the visually randomized one.

The proposed technique provides a quantitative and a visual way of estimating the quality of a future sim-to-real transfer using only unlabeled real images. It can also provide a way of injecting priors into the system by letting the user define the visuals of canonical images.

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. Learning agents capture the public imagination by promising systems that learn from their errors and improve day by day. Systems powered with AI can provide huge benefits by outperforming humans in more and more tasks.

However there are still hurdles in they way of applying artificial intelligence and deep learning in a lot of real life application. These systems are generally not robust enough to changes in their environment, and can respond to these changes by behaving unexpectedly.

They also need much larger amounts of data to learn tasks that humans can master from a few examples. Overcoming both these problems would be an important step towards the application of deep learning systems more broadly.

In this work I explore ways to make the training of reinforcement learning agents more sample-efficient and to improve their robustness simultaneously, to prepare them for application in the real world. To do that, I combine two methods that are well known and used separately. I show a combination based on them which helps them to mutually retain their respective advantages.

In the following chapters I present the relevant background and results from the literature, then propose my method, comparing it to a more naive baseline solution. I will show possible extensions as well.

I also present an implementation, which I evaluate in the Duckietown self-driving environment. After that I present the experimental results and analyse them as well. I close my work with discussing the topic of sample-efficiency in the light of pretraining methods and determine 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. It recently has successfully been applied for image processing [4], natural language processing [5], and speech recognition [6], and is also capable of solving generative tasks [7].

Artificial **neural networks** are modular computation graphs, whose outputs are partially differentiable by 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 by 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 **backprogation** 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 proportionally with their partial derivatives, i.e. with the sensitivity of the loss on 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, and use it for an optimization step. This makes the optimization stochastic because the mean loss over the dataset is only estimated by a random subset of it, 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].

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 reason is 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 image processing, where the first layers of a network learn low-level features such as edges or textures, later ones recognize simple structures based on those, while the final ones can even perceive complex stuctures [21].

A great depth can also make the training process quite difficult, by the problem of vanishing or exploding gradients [20], so it 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 Deep Learning has three main subfields: **supervised learning**, **unsupervised learning** and **deep reinforcement learning**. In supervised learning, for each data sample we have a corresponding label, and for each input 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, can be expensive, and also sometimes unfeasible if the correct labels are not known.

Another subfield is unsupervised learning, which means learning from the raw input data without any labels, generally by solving tasks whose solutions are inherently present in it. 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

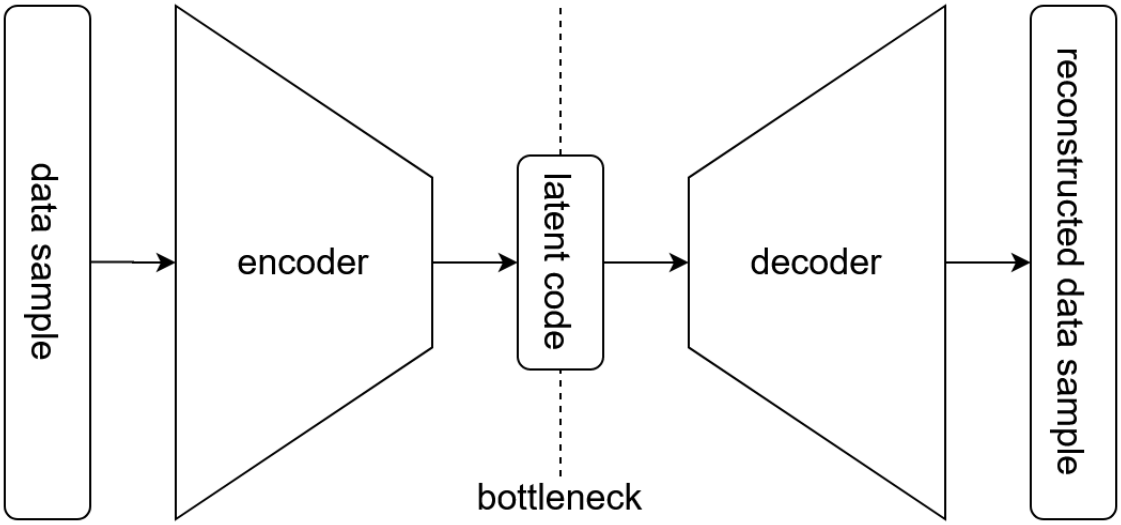

Figure 1: Architecture of an autoencoder

network can be used, and contrastive learning [22], where augmented versions of the same data sample have to be distinguished from augmented versions of other samples.

The last one is deep reinforcement learning, which is also a subfield of reinforcement learning. In this setting an agent can make 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 2.3.

2.2 Autoencoders

2.2.1 Overview

An autoencoder [23][18] (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 to compress its input, making the task nontrivial. We can then map our data to these learned compressed representations and use them in other tasks.

We call the part of the network until the bottleneck layer **encoder**, and the other part the **decoder**. The output of the bottleneck layer is called the **latent code**, and generally it is used for the representation of the data. The structure of an autencoder is shown on Figure 1.

As training an autoencoder does not require any labels only the data samples, autoencoders are trained in an unsupervised manner.

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

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

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 2.2.2 and 2.2.3, one can generalize the concept to stochastic 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 calculated 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** [23], 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) [24], small gradients of latent representations at the data points (contractive autoencoders) [25], or limited information content and a simple latent manifold structure (variational autoencoders) [26][27]. By corrupting the input data and trying to reconstruct the uncorrupted version (denoising autoencoders) [28], we can also force the network to implicitly learn the structure of the data distribution [29][18].

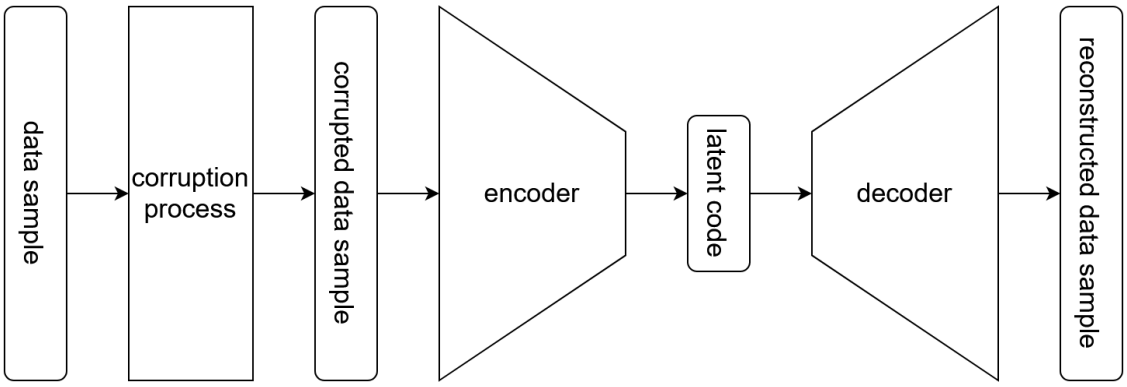

Figure 2: Architecture of a denoising autoencoder

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 the 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, these methods can be applied simultaneously as well.

2.2.2 Denoising Autoencoders

Denoising autoencoders (DAE) [28] 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 corruption process. The loss is the negative log-likelihood of the input sample, given the corrupted sample.

It has also been shown that solving this task implicitly forces the network to learn to represent the probability distribution of the data [29][18].

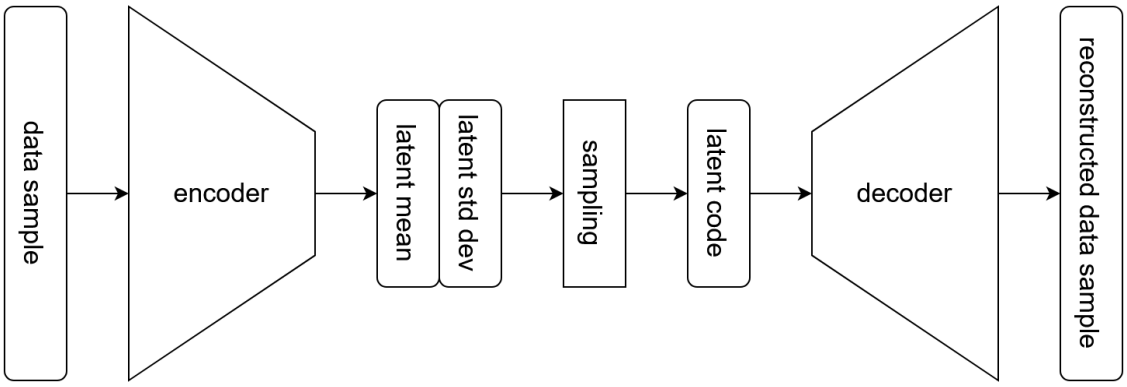

Figure 3: Architecture of a variational autoencoder

The theoretically most understood corruption process is the additive uncorrelated gaussian noise, however in practice salt-and-pepper noise [30], uncorrelated masking noise [30], and for images correlated masking noise (occlusion) [31] has been used as well.

In this work I will show that for image-type input data it is also feasible to use:

- different renderings of the same image with different textures
- augmented versions of the original image

2.2.3 Variational Autoencoders

A variational autoencoder (VAE) [26][27] 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.

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

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 [32], high-resolution human portrait generation [33][34], video generation [35] and even language modeling [36]. Its training procedure is generally considered more stable than the other popular method, Generative Adversarial Networks (GAN) [7], however it is also known to produce blurrier samples [37].

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 [38] 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 will use the latter method for estimating the variance for every pixel of the output image separately.

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

2.2.3.1 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 distibution.

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.

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

An extension of variational autoencoders has been proposed based on this view: the Beta-VAE [39]. 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.

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 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 following [38], I will only use the original VAE loss function (i.e. $\beta = 1$) since it is theoretically more grounded.

2.2.3.2 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 [40]. 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 [26][27] introducing variational encoders have proposed them.

2.2.3.3 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 [40].

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.

2.3 Reinforcement Learning

2.3.1 Overview

Reinforcement learning [41] is a subfield of machine learning, in which the task is to train an active agent of an environment to be able to act more optimally in the sense of maximizing its cumulative future rewards. The main components reinforcement learning are shown on Figure 4. I will introduce the most important concepts in the next paragraphs, following

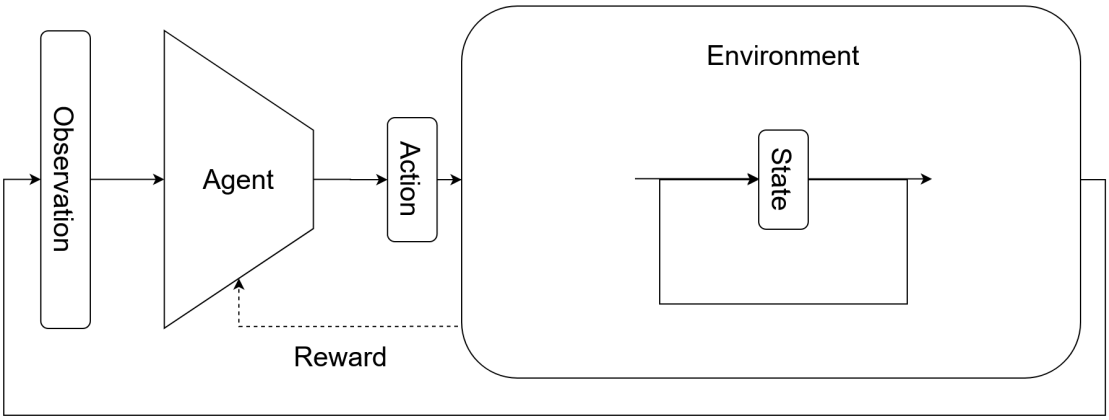

Figure 4: The main components of reinforcement learning

the lectures of David Silver for the course on reinforcement learning at *University College London* [42].

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 az a Markov Decision Process (MDP) [43]. *"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 rewards. Thus MDPs involve delayed reward and the need to tradeoff immediate and delayed reward."* [41, 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 [44].

The dynamics, behaviour of an environment can be described using its **state-transition probability matrix**. The matrix determines for every possible action in every state the probability of getting into a given next state, for all possible next states. 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 the state transition probability matrix.

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

$P^a_{ss'}$ is an element of the matric, whose value is the probability of s' corresponding to a current s state and a action.

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**. It outputs for all state-action pairs, how much reward they are worth. 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.

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

R^a_s 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.

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

Return G_t is for a 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.

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 can be called deep reinforcement learning.

Recently reinforcement learning has been successfully applied to create professional- or even superhuman-level agents in classical games such as go [45], chess [46], or Atari [47], and in modern computer games such as Dota 2 [48] and Starcraft 2 [49].

2.3.2 Sim-to-Real Gap

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 transfered to the real world.

Sim-to-real transfer has already been successfully applied in robot arm manipulation [50], robot locomotion [51] and simple self-driving tasks [52].

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 [53]
  - System identification and calibration [50]: more accurate dynamics parameters based on measurements
- Domain adaptation [54]: performance difference between the simulated and real environment can be decreased by fitting certain statistics to be more similar, by using auxiliary lost functions, or with transfer learning
- Domain randomization [55][56]: random perturbation of some parameters (e.g. visuals) of the simulated environment in every training episode, to broaden the range of environments, in which the agent performs properly (will be further discussed in Section 2.3.3)
- Regularization:
  - Observation-noise [50]: can make the agent more robust to discrepancies in its observations
  - Action-noise [50]: can force the agent to plan more robustly or behave more conservatively
  - Network reguralization [57]: application of techniques typically used against overfitting in deep learning, such as L2 regularization [58] and droupout [59]

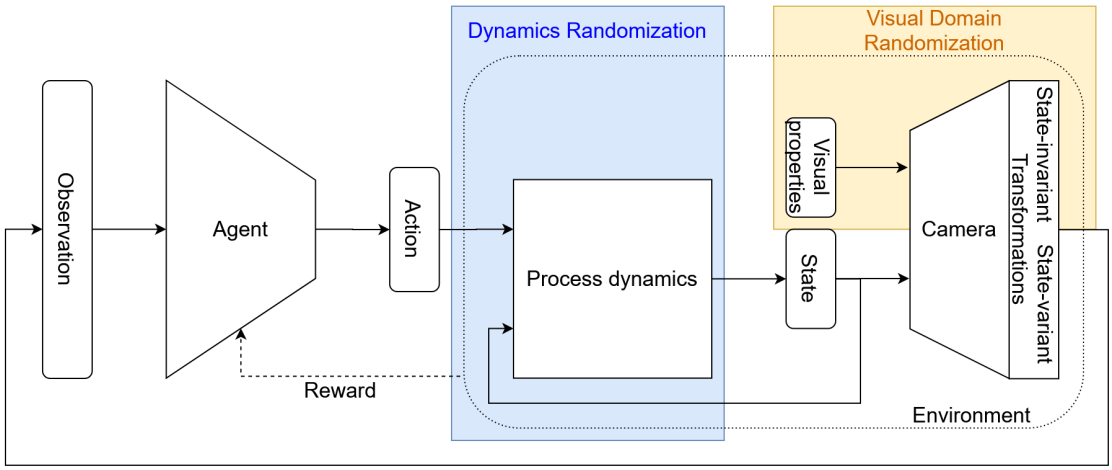

Figure 5: The scopes of visual domain randomization and dynamics randomization

2.3.3 Visual Domain Randomization

Domain randomization is a technique where we randomly perturb some selected parameters of the simulator we use, in every training episode. The aim of this technique is that by training a reinforcement agent in a diverse set of virtual environments, the range of environments in which it performs well is broadened, which increases the probability of a successful sim-to-real transfer.

The two main methods of domain randomization are visual domain randomization [55][56], where visual parameters are perturbed, such as textures, lightning, background, and dynamics randomization [60], 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 underlying state of the simulator, as observed by the agent. A simple example is Gaussian noise, but one could change the brightness, contrast or the saturation of the image as well.

These two methods are quite different, and promote generalization in different aspects. In this work I will only consider the method of visual domain randomization, and will introduce the most relevant works in the following sections.

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

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

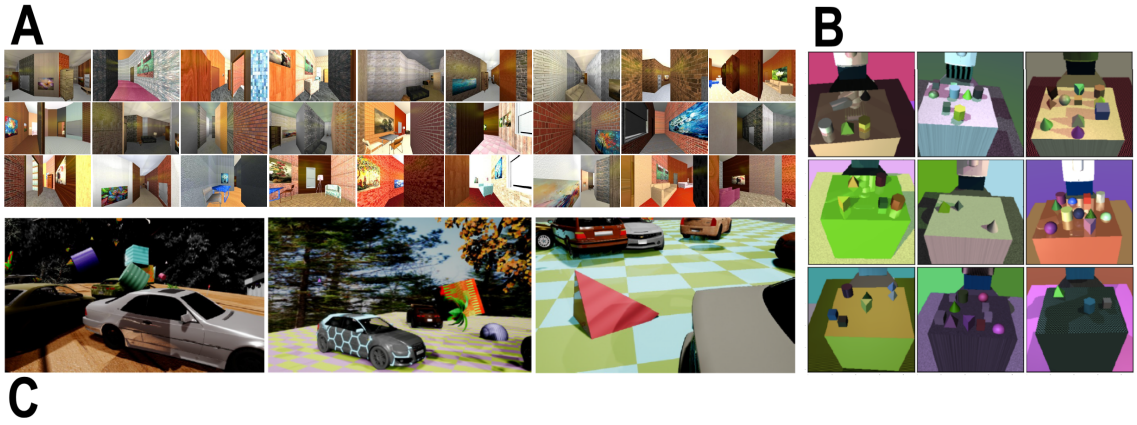

Figure 6: Visual domain randomization: A [55], B [56], C [64]

dimensional, their images contain a lot of data, and it is not easy algorithmically to get meaningful information out of that.

Since we generally have large quanitities of training data in image processing (images), and the classical algorithms have only been useful on a narrow range of tasks, it provided a fertile ground for neural networks and deep learning. With the help of convolutional neural networks [61], a family of networks specializing in image processing, remarkable achievements have been made in image processing tasks such as image classification [4], image segmentation [62], and object detection [63].

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 less 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.

2.3.3.1 Direct methods

In this section I will present such 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.

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 [55]. The authors carried out the training in simulated indoor environments, in which they placed lightsources, furniture, closed and open doors in random pozitions and directions. They also randomly choosed 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 [56]. 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 nonrealistic 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 for object detection [64], where the authors also used 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 lightsources and textures had the greatest impact on the result.

2.3.3.2 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 [65] 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 originial counterpart. So we can just simply add this term to the loss function, and use it for optimization, thereby ensuring robustness of the learned strategies.

A similar solution is the one, where we do not calculate the distances between the outputs, but between the activations of the last hidden layers instead [66], 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 in performance drop. Another work proposes this same method [67], however an interesting detail is that they use a randomly initialized convolutional layer for data augmentation.

Visual domain randomization can also be used for domain adaptation [68]. In this case a network is trained to generate a canonical observation (an observation that is similar to the observations of the original environtment) based on the randomized observation. We

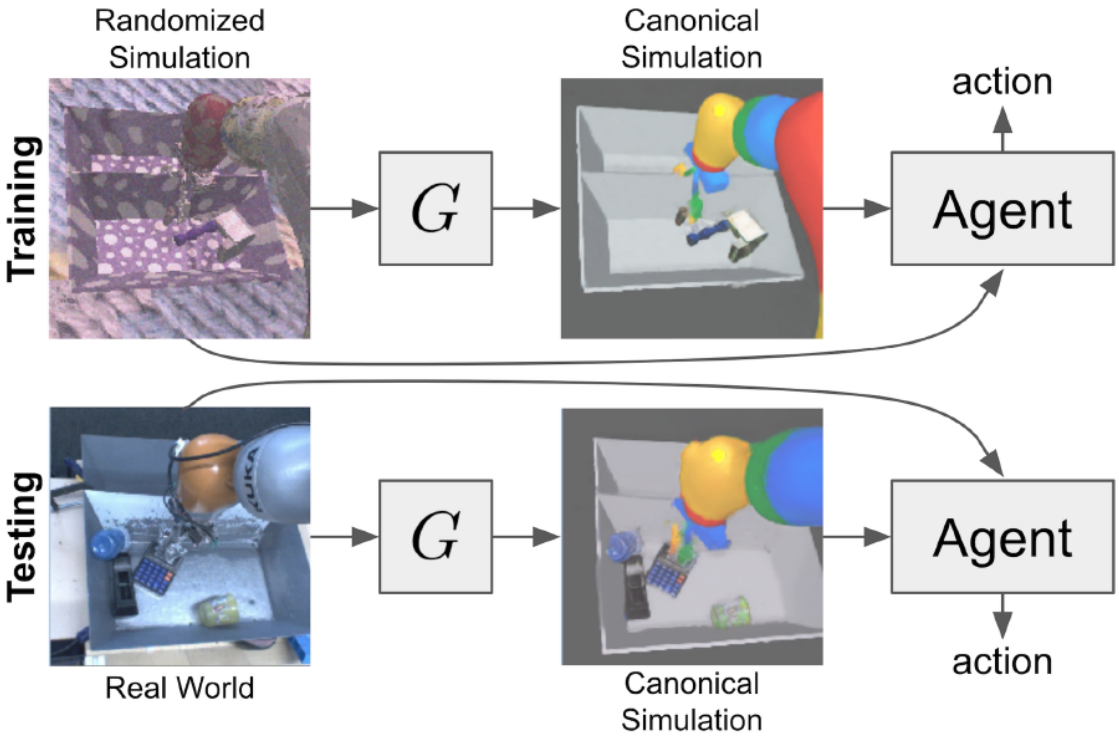

Figure 7: Domain adaptation based on domain randomization [68]

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.

This work can be seen as the closest to my work, and I will also use the same idea of randomized and canonical obervations.

Chapter 3

Proposed Method

3.1 Baseline

The goal of this work is to propose a novel method which combines visual domain randomization and unsupervised learning for increased sample efficiency using autoencoders. In order to do that we have to establish a baseline solution, which combines both these techniques in a straightforward naive way.

The architecture of the baseline solution is the following (see Figure 8):

A dataset is generated by gathering visually randomized observations from the simulator. Then I train a variational autoencoder on these images to be able to create a compressed representation of them.

Using the encoder of the network, I implement an observation wrapper for the simulator, which compresses the observations during the training of the reinforcement learning agent. I use the means of the latent code distributions as the latent representation. That way

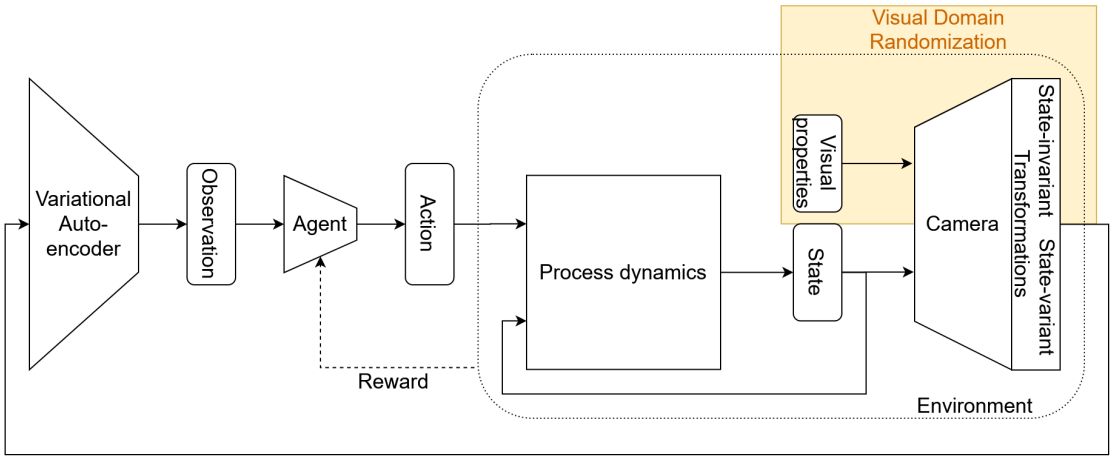

Figure 8: Overview of the baseline architecture

the agent can learn more sample-efficiently using the compressed representations of the visually randomized image observations.

3.2 Drawbacks of the baseline

Although this baseline method already improves sample-efficiency, it actually is not completely efficient, and also does not make a successful transfer the easiest 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 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 unsupervised 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 the visual domain randomization.

3.3 The Method

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.

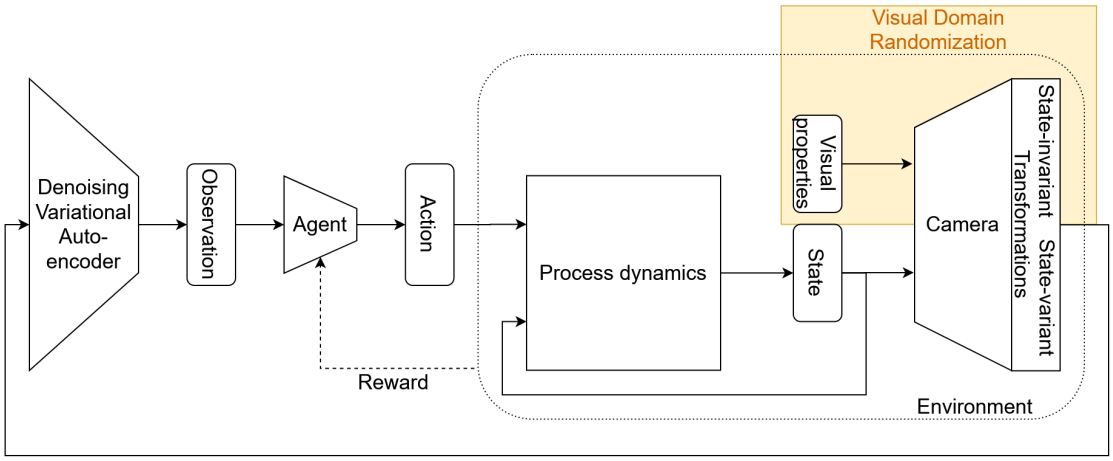

Figure 9: Overview of the proposed architecture

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 9.

3.3.1 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 Applications

The general application of the method is of course in settings where one would like to increase the sample efficiency of the reinforcement learning training procedure, while also using visual domain randomization simultaneously.

This application has been implemented in the Duckietown self-driving car environment, and the results are discussed in Chapter 5.

In this section I would like to propose further use-cases of the method, that are made possible by the design choices discussed above.

3.4.1 Visualization of the Latent Space

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, I have used this method to evaluate progress during training.

Another possibility is that when using only 2 latent dimensions (i.e. encoding a full input image into only 2 noisy variables), one can visualize the whole latent space by plotting generated images based on a grid of latent codes in the latent space to a grid of images.

3.4.2 Evaluation of Sim-to-Real Transfer Capability

3.4.2.1 Quantitative

As we have seen from the information-theoretic view of variational autoencoders (Section 2.2.3.3) 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 on them, we can monitor the KL-divergence term of the loss function. If it is considerably higher than what was seen during training on simulated images, that means that a successful sim-to-real transfer is not too likely. However if it does not differ to much, that signals that the encoder does not use considerably more information for encoding them, which means that they fit into the model's latent structure well.

An advantage of this application is that is does not require any label for the real images, which makes the real world data collection simple.

3.4.2.2 Visual

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.

One can perform a forward pass on the real world images and evaluate their "canonical" versions visually. If they have an acceptable quality and seem to be feasible visually, that means that the complete network was capable of encoding and decoding them, which means that the autoencoder is likely to work well on these types of real images even after a sim-to-real transfer.

3.4.3 Regularization with Latent Augmentation

Even though regularization is not as widely used yet in deep reinforcement learning as in supervised learning, recently it has been shown in [69], that reinforcement learning encoders overfit too if given enough capacity, which hinders their performance.

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.

3.5 Extensions

By interpreting the notions of "randomized observation" and "canonical observation" broadly, one can use the methods with the following two extensions as well.

3.5.1 Segmented Image as Canonical Observation

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 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.

3.5.2 Augmented Image as Randomized Observation

A limiting factor of the method is that is 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 here I propose a solution to these problems. 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 those augmentations, for which it makes sense to be interpreted as noise. E.g. random shifts are not useful augmentations in a self-driving setting, since they change the underlying observed state (the position of the car), and can not be denoised without memory.

Chapter 4

Implementation

4.1 Reinforcement Learning Task

I have implemented and evaluated the proposed method in the Duckietown self-driving car environment [70], using the Stable-Baselines3 [71] reinforcement learning library which uses the PyTorch [72] open-source deep learning library as its backend. I have used the PPO algorithm [73] for the training of the agents.

4.1.1 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 the **Duckietown**, which is a small scale 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 folloving (sometimes with other vehicles), navigation in junctions and pedestrian (duck) and obstacle 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 lightsource
- Camera position, angle, and field of view
- Color of the sky

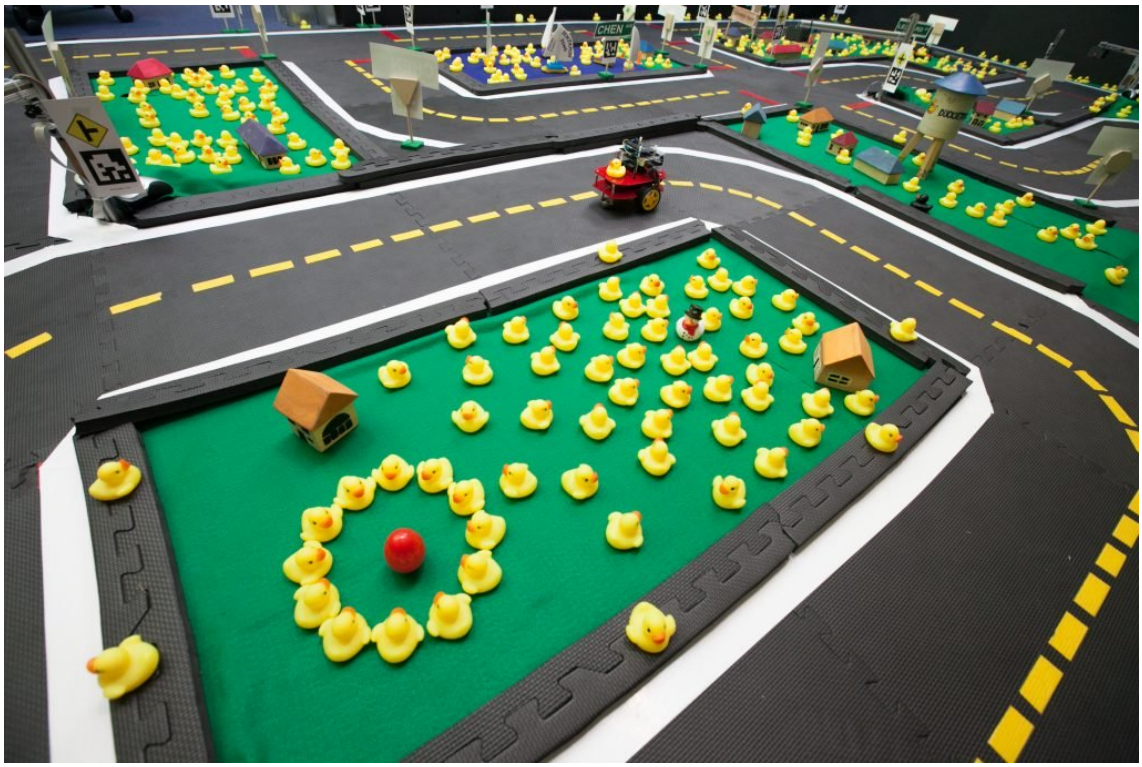

Figure 10: The Duckietown platform [74]

- 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 slightly modified 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.

4.1.2 Observation space

Following the work of [75], I have downscaled the 640x480 input image by a factor of 4 by both its sides to a size of 160x120 then I cropped 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.

It has been shown in [75], that 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 this prior work, and it has also been shown that stacking more than 3 frames does not yield considerable benefits, therefore I have stacked 3 past frames together for every observation. Since I used the images as colored images, the final size of the input image observations became 9x160x80.

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

4.1.3 Action space

A differential robot is usually controlled by diriving 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 much more slowly than it is capable of.

As my initial experiments have shown that 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 [75], 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 4.1, 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 minimimal 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.

$$
\begin{aligned}
u_{nom} &= 1.0 \\
u_{avg} &= \min(u_{nom}, \frac{1}{1+|\phi|}) \\
u_{left} &= clip(u_{avg}(1+\phi), -1, 1) \\
u_{right} &= clip(u_{avg}(1-\phi), -1, 1)
\end{aligned}
\tag{4.1}
$$

4.1.4 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. A more accurate description is that the reward is the speed of a virtual 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 4.2, 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.

$$
R = v_{progress} = v \cdot cos(\delta) \cdot \frac{r}{r+p} = v \cdot cos(\delta) \cdot \frac{1}{1+cp} \tag{4.2}
$$

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 [75].

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.

4.1.5 Evaluation metrics

I used the following metrics to monitor the performance of lane following agent 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 to the mean progression speed ($\bar{R}$)
- The average length of the evaluation episodes ($\bar{t}$)

The means have been taken over the whole process of the evaluation, which generally consisted of multiple 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 means have been taken over the whole process of the evaluation, which generally consisted of multiple episodes.

4.2 Dataset

I have generated and saved 200.000 images during the training of a simple convolutional reinforcement learning agent, and I have saved 3 different renderings for each image: a visually randomized, a canonical, and a segmented (Section 3.5.1) one. For saving and loading, the *pickle* 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: *4way, loop_empty, udem1 and zigzag_dists*, as can be seen in Figure 11.

I have also implemented a fourth type of observation, the augmented observation, which is always generated on the fly by applying a random convolution [67] to the canonical observation, and then min-max scaling it to be between 0 and 1. This has been used to implement the extension to my proposed method, discussed in Section 3.5.2.

Examples of the four types of observations can be seen in Figure 12.

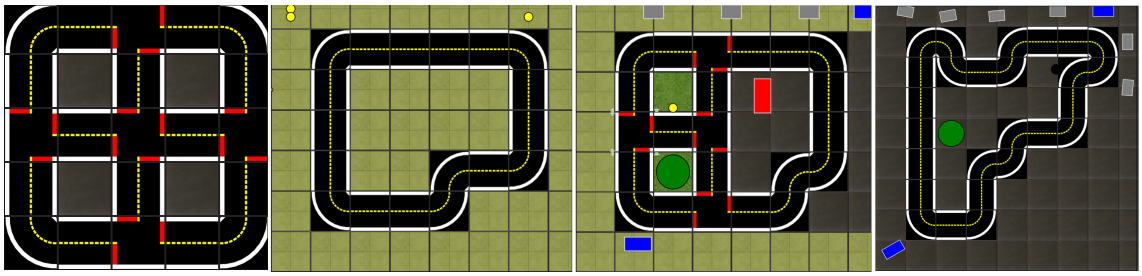

Figure 11: The maps that were used: 4way, loop_empty, udem1 and zigzag_dists

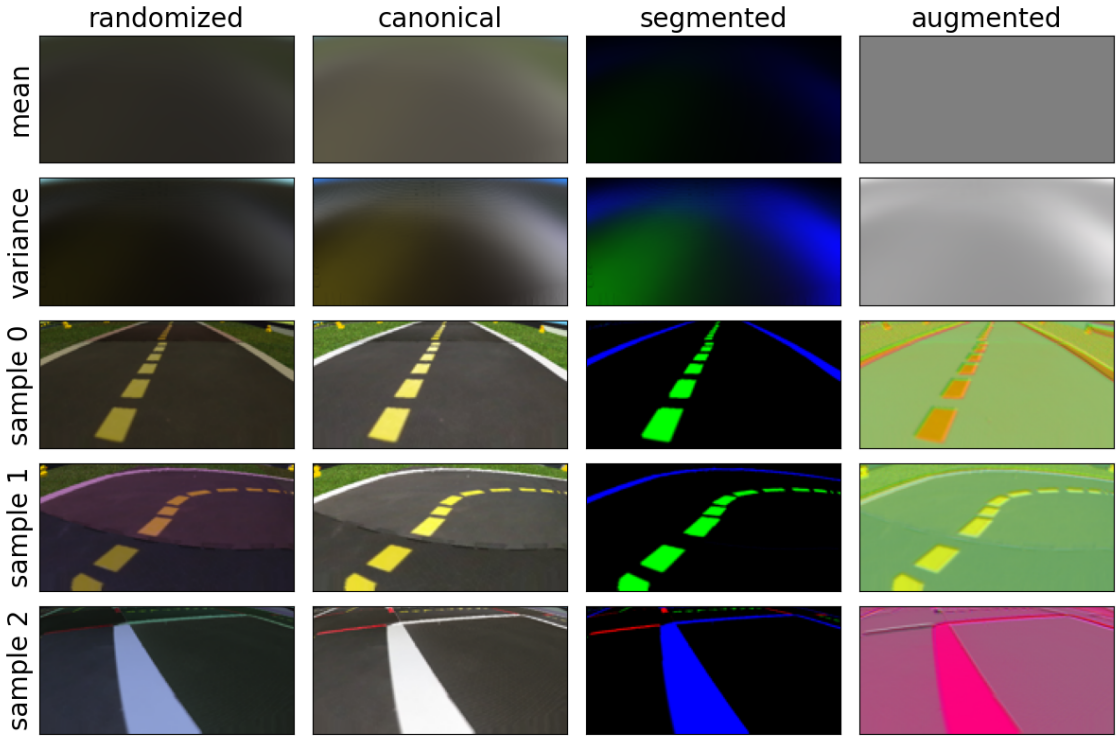

Figure 12: The mean and variance images corresponding to each observation type, with 3-3 corresponding samples

Using the dataset, the mean and variance observations for each image type have been determined before the training, to enable to use calibrated decoders in the variational autoencoder.

I have created another dataset as well, which contains 19.000 real images, downloaded from online logs of Duckiebots.

4.3 Variational Autoencoder

I have implemented the denoising variational autoencoder in the PyTorch deep learning framework, ith the usage of the PyTorch Lightning library, which is lightweight PyTorch wrapper that removes boilerplate code and makes the remaining code more organized, therefore more readable.

I have used pixelvise calibrated decoder distributions following the work of [38], but I have 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 distibution of the decoder is considered Gaussian as well, with its mean outputted by the decoder based on the latent code z, and its variance calculcated 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 distibutions.

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

4.4 Hyperparameters, architecture

During my work, I tried to use default hyperparameters wherever possible. I have extensively tuned manually the hyperparameters of the reinforcement learning agent, however

could not achieve meaningful improvements. The final hyperparameters I used can be found in Table 1.

For consistency, the feature extractor of the reinforcement learning agent has been replaced with a custom encoder, that is quite similar to the encoder of the variational autoencoder, with the exception that is has a 64 wide feature space in the last layer.

The value and policy networks have been kept at their default values, i.e. as separate 2 layered fully connected networks with Tanh activations.

An important note is that for the segmented image generation task the final activation of the decoder had to be changed to Tanh + ReLU, that way the network starts after initialization from a black image and therefore does not converge to completely black local optimum.

Table 1: Hyperparameter values used for the PPO algorithm

Name	Value
Optimization steps	50.000 - 400.000
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 2: 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

Table 3: 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 4: Autoencoder hyperparameters

Name	Value
Learning rate	0.001
Number of epochs	5
Width	64
Latent dimensions	8 (2 for latent visualization)
Standard deviation parametrization	as the log of the standard deviation

Chapter 5

Results

5.1 Analysis of the Autoencoders

During my work I have implemented 5 different types of autoencoder-based models who will be indicated in the following ways:

- VAE: the baseline solution (ordinary variational autoencoder, maps from randomized to randomized)
- DVAE: the standard implementation of the proposed methoed (denoising variational autoencoder, maps from randomized to canonical)
- DVAE-SEG: a denoising variational autoencoder that maps from randomized to segmented observations
- DVAE-AUG: a denoising variational autoencoder that maps from augmented to canonical observations
- DVAE-REG: the standard model, however when used for observation compression, the latent distributions are sampled instead of just taking the mean

5.1.1 Evaluation

In this section I evaluate all the variants of the proposed method based on two main components. The first is the visual quality of the reconstructions, which can be seen in Figure 13, the second is the final values of their loss functions, that can be found in Table 5.

Visually we can see that the assumption that the baseline solution would not be using its latent capacity most effective can be true, since it is the only method whose reconstruction of the red line from the sample 2 is barely visible. Generally it can be sad that all methods exhibit an acceptable visual quality for these images, possibly DVAE-AUG distort the most

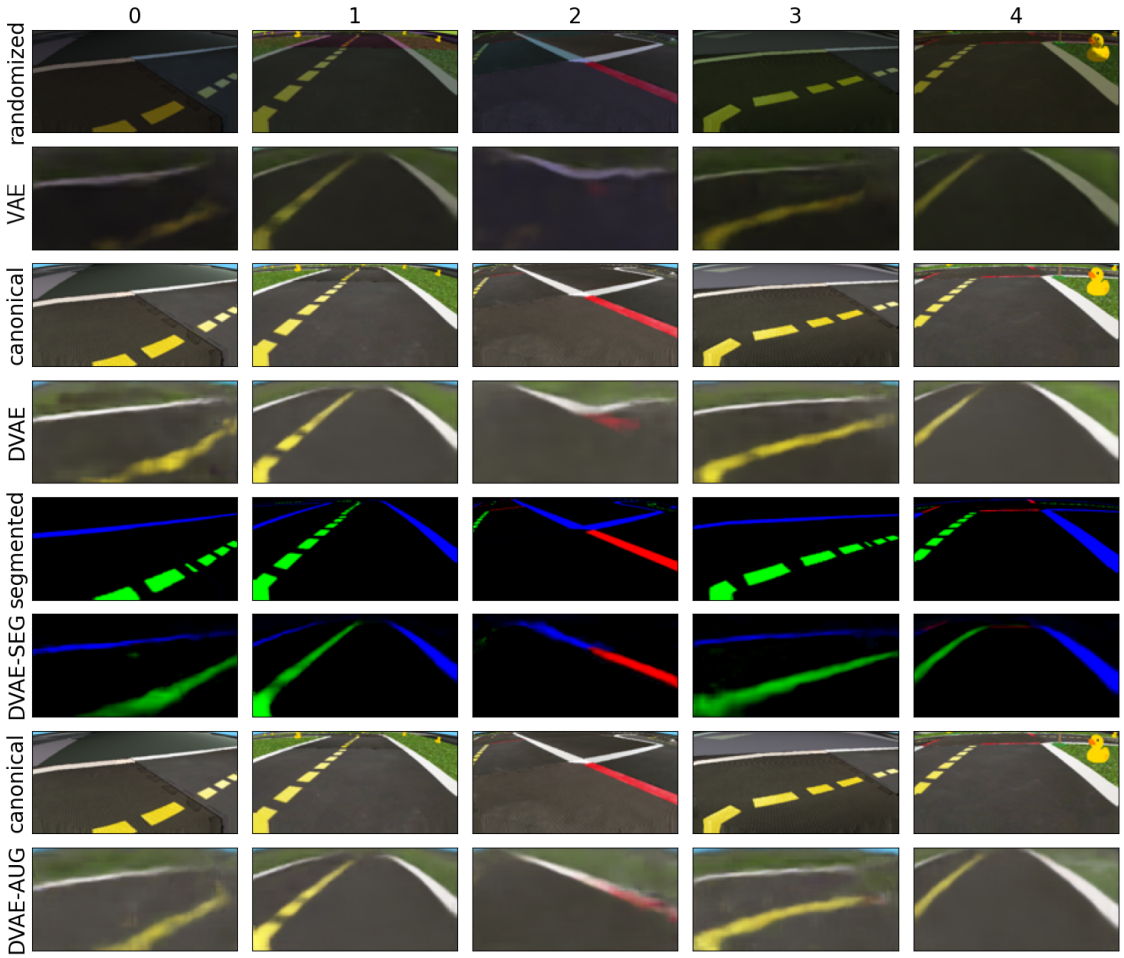

Figure 13: Target and reconstructed image pairs for each method

features among them. A interesting note that neither of them reconstructs the yyellow rubber duck from ample 4. Maybe the dataset does not contain enough of them?

The loss table shows similar values for the KL-divergence terms, with somewhat different reconstruction errors. Interestingly, the reconsturction error of DVAE is lower than DVAE-SEG's eventhough it has to reconstruct a more complex image (a canonical one instead if the segmented). DVAE-AUG shows the highest reconstruction error which lines up with our visual observations.

Table 5: Loss components for the autoencoder variants

	Loss [nat]	**Reconstruction error [nat]**	**KL-divergence [nat]**
VAE	6651	6608	42.6
DVAE	**5720**	**5678**	42.2
DVAE-SEG	6453	6412	40.6
DVAE-AUG	7596	7556	**39.8**

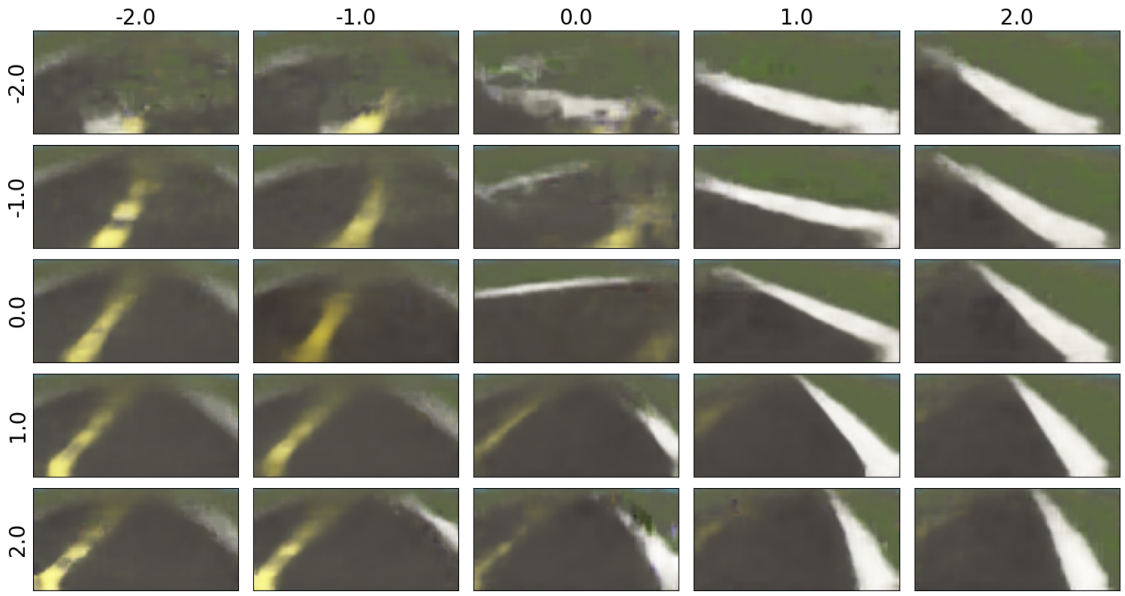

Figure 14: The latent space of a 2-dimensional denoising variational autoencoder, sampled from a grid

5.1.2 Visualization of the Latent Space

I have trained an autoencoder with a two-dimensional latent space as well, to be able to visualize it on a 2D-grid with a grid of generated images from the correstponding latent codes. The visualization can be seen on Figure 14.

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. Maybe these are exactly inbetween two regions, or maybe the network was forced to make these representation because of the highly reduced latent capacity.

5.1.3 Sim-to-Real Transfer Capability

I have discussed it as a possible application of the method, that we can also use real world images as the network's input. In this section I show the two ways of evaluating sim-to-real transfer capability based on real images.

5.1.3.1 Quantitative

The quantitative measurements show that the difference between the KL-divergence terms of simulated and real images is actually not that high. The VAE model has the highest discrepancy, which could signal a less probable sim-to-real transfer.

Table 6: KL-divergence terms for simulated and real images

	Validation KL-divergence [nat]	**KL-divergence on real images [nat]**
VAE	42.6	51.8
DVAE	42.2	42.7
DVAE-SEG	40.6	**34.6**
DVAE-AUG	**39.8**	41.0

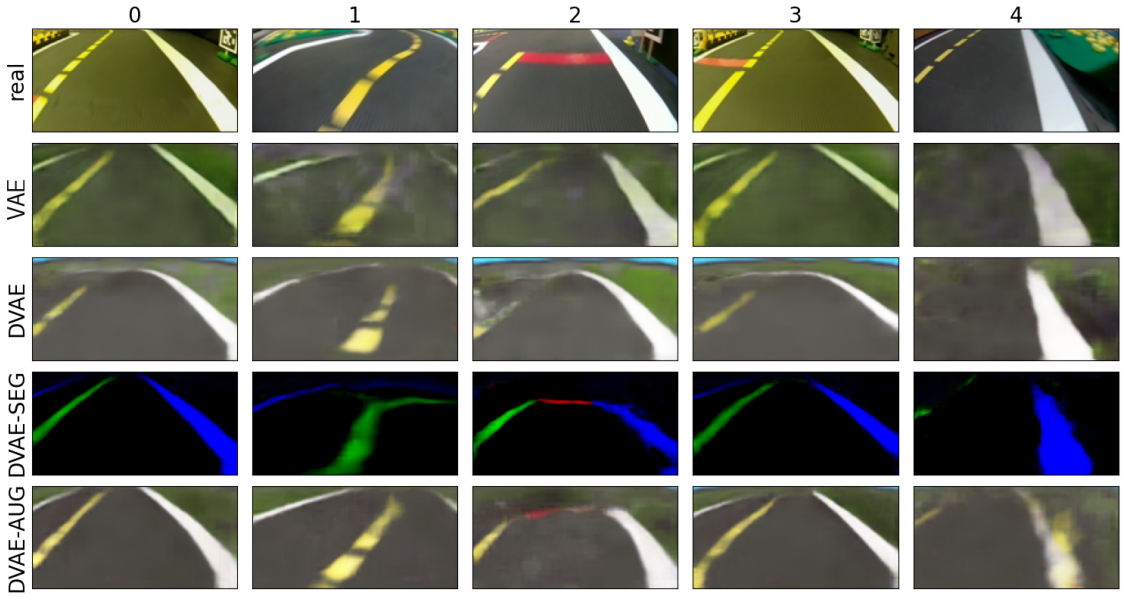

Figure 15: Reconstructed versions of real images with each method

Interestingly the DVAE-SEG model produced lower KL-divergence latent codes for the unseen real images, than on the ones that is has been trained on. This phenomenon might require further investigation.

The DVAE and DVAE-AUG models show a relatively low KL-divergence discrepancy which makes them good candidates for sim-to-real transfer.

5.1.3.2 Visual

Another option is to inspect the outputs of the networks visually, and draw conlcusions from that. The general quality of reconstructions, that I have seen from the models when they have a real image as input, is lower. The blurry spot-like sturctures 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, so they should not fail completely in these new environments.

5.2 Comparisons with other Methods

To provide proper comparisons, the following end-to-end reinforcement learning models will be used as baselines to the proposed models:

- CNN-DR: a purely reinforcement learning agent with a convolutional feature extractor, that tries to learn from the randomized observations
- CNN-CAN: a purely reinforcement learning agent with a convolutional feature extractor, that tries to learn from the canonical observations. It can be interpreted as the method of [68], with the adaptation network being a perfect oracle
- STATE: a purely reinforcement learning agent that is fully connected and learns directly from the physical states (lane-angle, lane-distance, lane-curvature, speed, angular velocity)

All of these baseline methods are end-to-end, which means that they do not utilize any pretraining, and learn representations based only on the reward signal.

The agents that utilize representations learned by unsupervised pretraining will be indicated by the name of the pretraining method used: VAE, DVAE, DVAE-SEG, DVAE-AUG and DVAE-REG. A more detailed description is provided in Section 5.1.

5.2.1 Domain Randomization Methods

When comparing the proposed method with other methods, that deal with visual domain randomization in a different way, we can see that the proposed method is on one hand more sample-efficient since it reaches higher performance with less training examples. This can be attributed to its unsupervised pretraining objective.

For fair comparison, the end-to-end methods CNN-DR and CNN-CAN were given twice the amount of training examples during training to offset the advantages in sample-efficiency caused by the pretraining.

On the other hand we can also see that it achieves higher performance, then the other methods. Looking at Table 7, we can se that it's average episode length is significantly faster, along with its average speeds and reward.

This show that the proposed method is a capable tool for solving tasks in the visual domain randomization settings.

5.2.2 Sample-Efficient Methods

If we compare the model to other sample efficient methods, one thing can be seen clearly: the method is still far away from both the sample-efficiency, and the pure performance

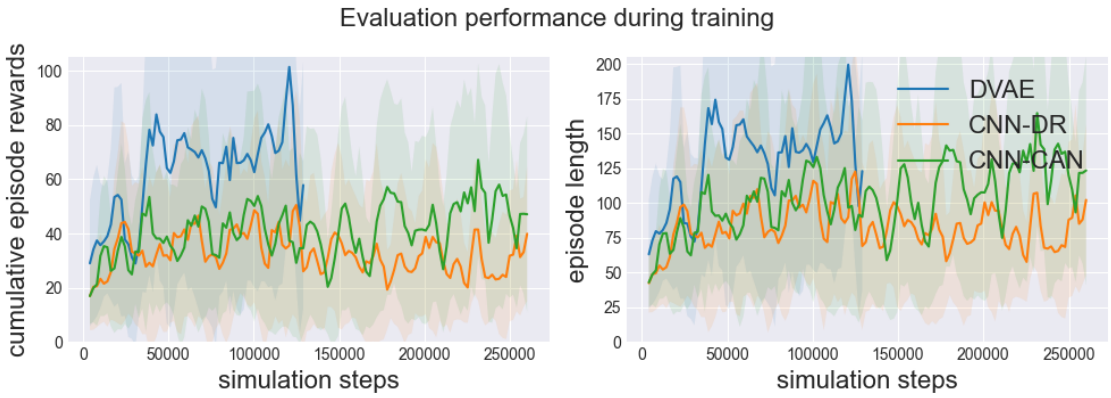

Figure 16: Comparison with other domain randomization methods

Table 7: Comparison with other domain randomization methods

	$\bar{\delta}[rad]$	$\bar{p}[m]$	$\bar{v}[m/s]$	$\bar{v}_f[m/s]$	$\bar{R}[m/s]$	$\bar{t}[1]$
DVAE	**0.116**	**0.032**	**0.494**	**0.485**	**0.491**	**166.3**
CNN-DR	0.187	**0.032**	0.439	0.424	0.414	110.8
CNN-CAN	0.225	**0.031**	0.424	0.406	0.399	139.7

of such an agent, that can access the true physical state of the environment. This means that there is still room for improvement on the representation learning side of things.

An other interesting observation is that the proposed technique is only slightly better than its baseline, the simple VAE model. It achieves only slightly higher speeds and episode lengths.

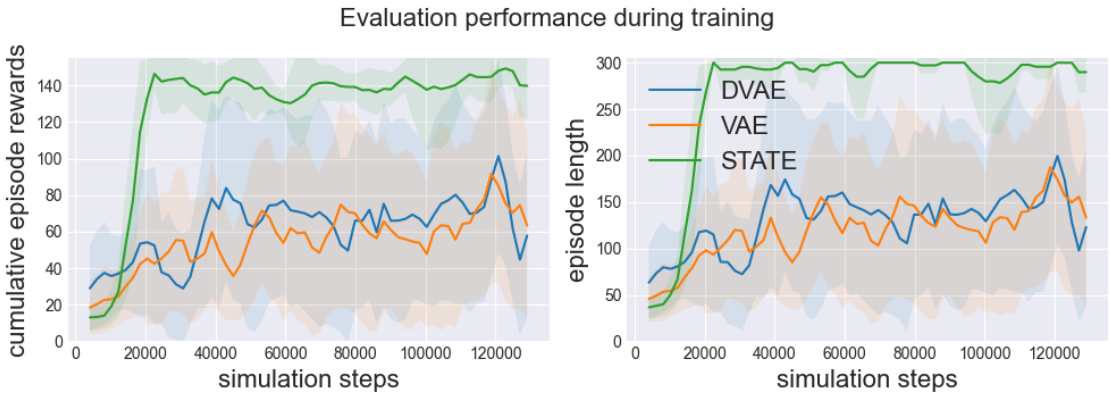

Figure 17: Comparison with other sample-efficient methods

Table 8: Comparison with other sample-efficient methods

	$\bar{\delta}[rad]$	$\bar{p}[m]$	$\bar{v}[m/s]$	$\bar{v}_f[m/s]$	$\bar{R}[m/s]$	$\bar{t}[1]$
DVAE	0.116	0.032	**0.494**	**0.485**	0.491	166.3
VAE	0.127	**0.024**	0.485	0.476	0.473	152.7
STATE	**0.071**	0.031	**0.497**	**0.495**	**0.494**	**278.6**

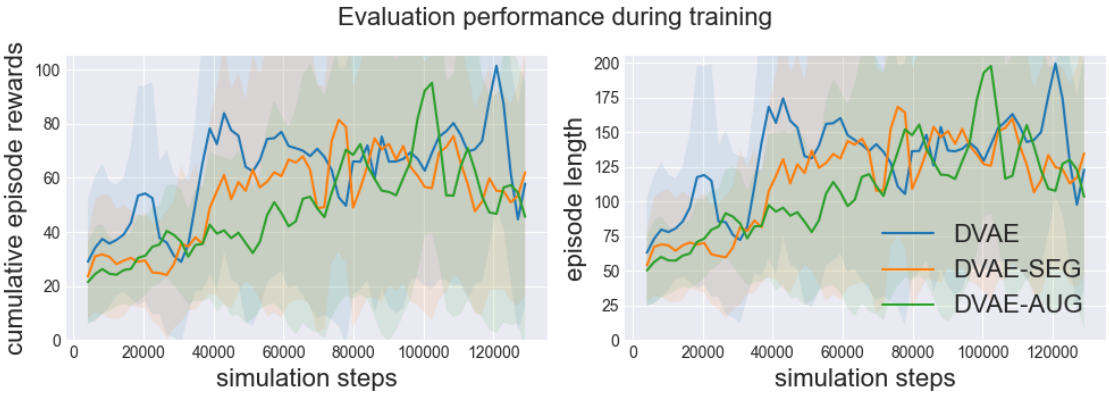

Figure 18: Comparison with the extensions

5.2.3 Extensions

When comparing the method to its extensions, we can see that they provide a really similar performance. This can tell us, that even though the target images differ greately between these methods, what the network learns is still similarly useful as a latent representation.

5.2.4 Latent Augmentation

Lastly, if we take a look at the version of the method, where the reinforcement learning network is regularized by only getting access to the noisy distributions of the latent space from which it can sample latent codes, we see that the performance of this regularized method is slightly lower.

This means that we were not overfitting, therefore the regularization only made the problem a bit harder. Even though it did not help in this setting, it can still be a useful tool in tasks where our reinforcement learning encoder is overparametrized.

Table 9: Comparison with the extensions

	$\bar{\delta}[rad]$	$\bar{p}[m]$	$\bar{v}[m/s]$	$\bar{v}_f[m/s]$	$\bar{R}[m/s]$	$\bar{t}[1]$
DVAE	**0.115**	0.031	0.494	0.485	**0.490**	**165.0**
DVAE-SEG	0.120	**0.025**	**0.501**	**0.491**	0.480	**165.9**
DVAE-AUG	0.136	**0.026**	0.478	0.468	0.464	150.9

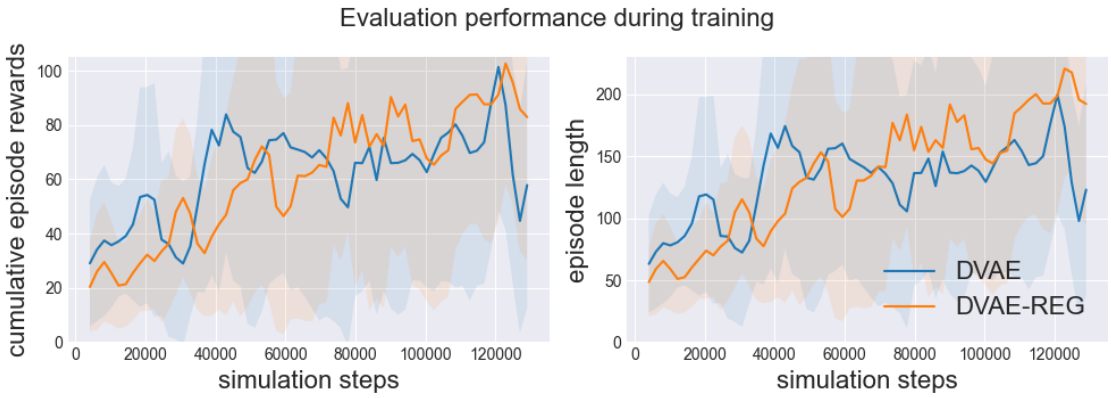

Figure 19: Comparison with the use of latent augmentation

Table 10: Comparison with the use of latent augmentation

	$\bar{\delta}[rad]$	$\bar{p}[m]$	$\bar{v}[m/s]$	$\bar{v}_f[m/s]$	$\bar{R}[m/s]$	$\bar{t}[1]$
DVAE	**0.116**	0.032	**0.494**	**0.485**	**0.491**	**166.3**
DVAE-REG	0.163	**0.028**	0.471	0.458	0.472	133.2

5.3 Discussion

I would like to discuss the term "sample-efficiency", when used in settings with unsupervised pretraining. When I evaluated the other methods that have not been pretrained, I did not take into account that the pretrained autoencoders have already seen quite a few data samples.

Though from a theoretical point of view this argument is right, these samples have different cost associated with them from a practical viewpoint.

Those samples that were needed for the pretraining can always be reused, such as I did not have to generate a new dataset every time I had to retrain the autoencoders. I can also add, that if a feature extractor is pretrained properly, it can be used for multiple reinforcement learning experiments consecutively.

One has to take into account that applying deep learning is always an iterative process, and it is useful if we can save on retraining. On- and even off-policy reinforcement learning agents will always have to gather new data when being retrained. This however can change in the future with the help of offline reinforcement learning techniques.

Chapter 6

Conclusion

In this work I proposed a novel method that combines visual domain randomization and unsupervised autoencoding pretraining in an effective way, by utilizing denoising autoencoders.

I have summarized the related literature, and have given intuitive explanations of denoising and variational autoencoders, that play an important role in the proposed method. I also presented the problem of the sim-to-real gap, which is an important obstacle in the way of applying reinforcement learning in the real world, and have shown possible countermeasures from the literature.

I proposed my method by comparing it to a naive baseline, considered its inefficiencies, and argued that this new technique is even more natural to apply, since it promotes robustness against visual perturbations, just like visual domain randomization does. I have given motivations for all the design choices behind the new method, and showed multiple extensions and further applications.

I have implemented and evaluated this method in the Duckietown self-driving environment. I have gathered a dataset for pretraining tasks, which has been built with further applications in mind, as it contains pieces of information about the physical state of the simulator that is not used in this work.

Since the method provides multiple ways of evaluating it, I have tested and analysed its variants in numerous settings, the fact that it is a generative model, helped with its interpretabilty and explainability.

The results show that the proposed method outperforms the baseline, and without knowing the complete physical state, it narrows the performance gap with the the ideal setup, where the complete state of the system is known. The extensions achieved a similar performance while extending the possible use-cases of the method. With the help of image augmentations the technique can be used even in settings where domain randomization is not implemented at all in the simulator.

Future work could explore whether extensions of variational autoencoders such as hierarchical or vector-quantized ones can be useful in this setting by learning higher level representations of the input data. It would also be useful to investigate how these different image reconstruction objectives shape the latent space and why do they lead to solutions with similar performance.

I hope that the proposed technique makes a small step on the long way towards the goal of applying deep learning and reinforcement learning models in the real world effectively and safely.

Acknowledgement

The research presented in this work 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] 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.

[6] 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.

[7] 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.

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

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

[23] 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.

[24] 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.

[25] 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.

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

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

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

[34] 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.

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

[36] 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.

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

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

[39] 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.

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

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

[42] D. Silver, "Ucl course on reinforcement learning." `https://www.davidsilver.uk/teaching/`. Access date: 2020.10.28.

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

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

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

[56] 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.

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

[58] 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.

[59] 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.

[60] 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.

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

[62] 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.

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

[64] 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.

[65] 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.

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

[67] 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.

[68] 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.

[69] 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.

[70] 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: 2020.10.28.

[71] 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: 2020.10.28.

[72] 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.

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

[74] D. Foundation, "About the duckietown platform." `https://www.duckietown.org/about/platform`. Access date: 2020.10.28.

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