ON THE SCALABILITY OF DEEP INVERSE REINFORCEMENT LEARNING IN HIGH-DIMENSIONAL STATE SPACES

HENRY MAATHUIS

Supervisors:
DR. M.A. WIERING
PROF. DR. L.R.B. SCHOMAKER

Master's Thesis Artificial Intelligence
University of Groningen

September 28, 2020

ABSTRACT

Inferring intentions behind observed behaviours is arguably an important aspect when learning new tasks. Inverse Reinforcement Learning (IRL) is a relatively small and recent area which studies techniques which allow for deriving reward functions on the basis of observed behaviour. Such a reward function can be seen as a blueprint and stipulates how an agent should behave and what it should achieve. Once a reward function is in place, it is possible to learn a policy. A policy provides an actual implementation what an agent does given the state that the agent is in. Normally in a Reinforcement Learning task, the reward function is handcrafted by a human. IRL therefore requires an extra abstraction to first derive the reward function before a policy can be learnt.

One of the big problems in IRL is that the algorithms do not scale well to high-dimensional environments due to overfitting of the reward function during the reconstruction process. As such, this thesis contributes to the field of IRL by accessing the state-of-the-art Adversarial Inverse Reinforcement Learning (AIRL) algorithm in combination with dimensionality reduction techniques, specifically autoencoders. Three different autoencoders are considered including one that forms a discrete latent distribution over the data. In the experiments, the algorithms are evaluated on several high-dimensional environments: Catcher, Pong and Freeway. The results indicate that autoencoders could be useful in cases where the state space is relatively complex. Although, one should be careful when introducing autoencoders as they can also thwart the process of learning reward functions.

Additionally, this thesis is meant as an overview of the current state of knowledge of IRL. Hopefully this thesis piques interest amongst new researchers to further explore the field of IRL.

CONTENTS

1 INTRODUCTION 1
1.1 Deep Reinforcement Learning 2
1.2 Deep Inverse Reinforcement Learning 2
1.3 Autoencoders 2
1.4 Research Questions and Contributions 3

I BACKGROUND INFORMATION

2 FUNCTION APPROXIMATION 7
2.1 McCulloch-Pitts Model 7
2.2 Rosenblatt Perceptron 7
2.3 Multi-Layer Perceptron 8
2.4 Neural Network Optimisation 9
2.5 Convolutional Neural Network 9
2.6 Generative Adversarial Networks 11

3 DIMENSIONALITY REDUCTION TECHNIQUES 15
3.1 Standard Downsampling 15
3.2 Autoencoders 15

4 REINFORCEMENT LEARNING 21
4.1 Key Concepts 21
4.2 Reinforcement Learning Algorithms 23

5 STATE-OF-THE-ART DEEP REINFORCEMENT LEARNING 27
5.1 DQN 27
5.2 Policy Gradient Algorithms 28

6 INVERSE REINFORCEMENT LEARNING 35
6.1 Imitation Learning 35
6.2 Inverse Reinforcement Learning 35
6.3 Maximum Margin Optimisation 37
6.4 Guided Cost Learning 39
6.5 Adversarial Inverse Reinforcement Learning 41
6.6 Convolutional Neural Network AIRL 42

II METHODOLOGY

7 GENERAL IMPLEMENTATION 47
7.1 Environments 47
7.2 Architecture Proximal Policy Optimization 48
7.3 Autoencoders 50
7.4 Adversarial Inverse Reinforcement Learning 54

8 RESULTS 57
8.1 Sampling Expert Demonstrations 57
8.2 Learning a Low-Dimensional Embedding 57
8.3 AIRL 65

9 DISCUSSION AND CONCLUDING REMARKS 69

9.1 AIRL Behaviours 69
9.2 Research Questions 71
9.3 Concluding Remarks 71

BIBLIOGRAPHY 73

1 INTRODUCTION

Machine Learning is a widely studied field that concerns itself with learning specific tasks without the need of step-by-step instructions. The earliest form of Machine Learning described in literature concerns itself with pattern classification. Nowadays the field of Machine Learning is often separated into three fields: Supervised Learning, Unsupervised Learning and Reinforcement Learning. These categories are explained before diving into Inverse Reinforcement Learning. At the end of this chapter, the research questions and the contributions are reported.

In Supervised Learning (SL), the learners are asked to make a prediction on the basis of an input pattern. In such algorithms the learner has access to what the desired output prediction should be. By allowing the learner to adapt to the input patterns, it is able to learn how to make predictions consistent with the desired output. In general, SL algorithms come in two flavours: classification and regression. An example of a classification task is to distinguish images of cats and dogs [36]. Predicting the future stock prices is an example of regression [2].

In Unsupervised Learning (UL) there are no labels. These types of algorithms have to find the underlying structure in the data itself or learn something about the distribution of the data. Many of these algorithms work by grouping data together that share similar properties. This process is also called clustering. An example of a clustering is to group similar stars in astronomical datasets together [58]. Another class of UL algorithms learn by association. Association Rule Mining concerns itself with learning specific rules in the data [29]. An example are systems that look at consumer purchase behaviour. How likely is a person to buy product B when product A is bought?

Reinforcement Learning (RL) concerns itself with learning how to interact with an environment. A reward function specifies how well the agent is behaving in such a system. Retrieving feedback from the reward function differs from SL since it does not indicate what the correct action of the agent should have been. Instead it provides a sparser signal that indicates how well the agent performed. RL systems therefore use trial and error in order to learn how to behave optimally in an environment. An example of Reinforcement Learning is to learn to grasp objects using a robotic arm [34]. RL is further discussed in section 4.

1.1 DEEP REINFORCEMENT LEARNING

Standard Reinforcement Learning techniques are limited in applicability since they do not scale well to more complicated tasks. When combining Reinforcement Learning techniques with Deep Neural Networks (DNN), it is possible to learn more complicated tasks which are impossible to achieve with shallow neural networks or tabular approaches [32]. Such DNNs are able to automatically learn features from the input data forming a representation that can be used efficiently with RL techniques without the need of handcrafting input features. Convolutional Neural Networks (CNNs) [24] are a special type of DNNs which can extract relevant information from pixel data. This research makes use of CNNs in combination with DRL techniques. CNNs are described in more detail in section 2.5.1. Details on DRL techniques are found in section 5.

1.2 DEEP INVERSE REINFORCEMENT LEARNING

Although Reinforcement Learning is able to learn behaviour by clever consultation of a reward function, it is sometimes not feasible to use this approach. In Inverse Reinforcement Learning (IRL) the problem of RL is reversed [1, 35]. Instead, the goal of IRL is to derive a reward function automatically given observed behaviour. In cases where it is hard to come up with a reward function, IRL can be considered. Generally speaking, IRL consists of two steps. First it attempts to reconstruct a reward function. Second it learns a policy given the newly reconstructed reward function. For simple tasks, IRL is able to retrieve a reward function that reflects the observed behaviour well. However there is a lot of work to do to improve the performance on more complicated Deep IRL tasks. The field of IRL is still relatively young and has many problems to overcome. Chapter 6 describes the aspects of IRL in more detail.

1.3 AUTOENCODERS

Current Deep IRL algorithms have problems when faced with high-dimensional data. To alleviate the problem, autoencoders could be considered to reduce the dimensionality of the data [27]. This research utilises several autoencoders to evaluate the performance of Adversarial Inverse Reinforcement Learning [15], the state-of-the-art IRL algorithm, on high-dimensional pixel environments. Most autoencoders in this research yield a low-dimensional continuous feature vector. The most promising autoencoder uses the concept of Vector Quantisation algorithms to deliver a discrete low-dimensional representation [54].

1.4 RESEARCH QUESTIONS AND CONTRIBUTIONS

This primary goal of this thesis is to give an answer to the research questions which are stated in this section. Additionally in this section the contributions of the thesis as a whole are highlighted.

1.4.1 *Investigating Deep Inverse Reinforcement*

Many problems with current Deep Inverse Reinforcement Learning (IRL) algorithms are tied to the curse of dimensionality (COD). COD is a problem widely known in the field of Machine Learning in which the data is embedded sparsely in a large volume of space. Although some research has been done on autoencoders to reduce the dimensionality of the data on IRL problems, much research has to be done to properly scale to high-dimensional problems. In order to explore the benefits of autoencoders on deep IRL problems, the following research questions come to mind:

1. Can vanilla and variational autoencoders help improve the reconstruction of reward functions on high-dimensional data?
2. How do these results compare to instances where raw input of the original data is considered?
3. Vanilla and variational autoencoders learn a low-dimensional continuous representation. Would autoencoders that learn a binary or discrete representation instead allow for more robust reward function reconstructions?

Note that the reward functions themselves are not evaluated. Instead, the policy obtained directly from the reward function is used to evaluate the expected return of rewards.

1.4.2 *Additional Contributions*

Additionally this thesis is meant as a review of the current state of knowledge that is relevant for the field of Deep Inverse Reinforcement Learning (DIRL). Considering that the field of DIRL is relatively small, it aims to provide researchers with a better understanding of the problems and limitations currently faced in this field.

Part I

BACKGROUND INFORMATION

2 FUNCTION APPROXIMATION

2.1 MCCULLOCH-PITTS MODEL

Function approximation is widely studied in the domain of Machine Learning. In the early 40s, a theoretical biological model for function approximation has been coined; The McCulloch-Pitts model [30]. This model is loosely based on the working of a biological neuron and is able to partition the input space in two subspaces by means of a straight line. One important restriction of the McCulloch-Pitts neuron is that each of the inputs and the output are binary. Another important restriction is that the weights are fixed in this system, therefore this system was not able to learn on the basis of examples. The output of the McCulloch-Pitts neuron is simply the weighted sum of the inputs as can be seen in the equation below:

$$y_j = f(\mathbf{w} \cdot \mathbf{x_j}). \tag{2.1}$$

Note that w is the weight vector and x_j is the input vector. $f(\cdot)$ is the activation function that transforms the output y_j. Since the output is binary, $f(\cdot)$ can simply use a threshold value to determine whether the output should map to 0 or 1.

2.2 ROSENBLATT PERCEPTRON

It took many years after the introduction of the McCulloch-Pitts model before there was a breakthrough in the field of Neural Networks. In 1958, Rosenblatt developed the Perceptron based on the McCulloch-Pitts model [41]. While the previous model used fixed weights, the Rosenblatt Perceptron is able to adapt its weights when presented with inputs. Similar to the McCulloch-Pitts model, the output is computed as the weighted sum of its inputs.

Also the concept of bias was introduced which allows for offsetting the decision boundary from the origin of the input space. Another difference is that the inputs and output are also no longer restricted to be binary. This allows the activation function $f(\cdot)$ to squash the weighted sum of inputs to a non-binary value. Examples of other activation functions are: sigmoid, tanh, linear and ReLu. These changes cause the output computation to take the following form:

$$y_j(t) = f(\mathbf{w(t)} \cdot \mathbf{x_j} + b(t)), \tag{2.2}$$

where $w(t)$ are the weights, and $b(t)$ is the bias at time t.

The Perceptron is able to update its weights by making use of the discrepancy between the output of the neuron and the target output. An often used method to optimise a neural network is called Gradient Descent. This computation is given below:

$$\mathbf{w(t+1)} = \mathbf{w(t)} + \eta \cdot (d_j - y_j(t)) \cdot \mathbf{x_j}, \tag{2.3}$$

where η is the learning rate, d_j is the desired output. $x_{j,i}$ denotes the input i of example j. $w_i(t+1)$ is the ith weight value at time $t+1$. In Section 2.4, there is a longer discussion on neural network optimisation.

2.3 MULTI-LAYER PERCEPTRON

In the case of the McCulloch-Pitts model and the Perceptron, only one neuron is used. A single neuron only allows for linear separation of the data. Combining multiple Perceptrons in a network allows us to learn more complex non-linear decision boundaries. An example of such a network is called a Multi-Layer Perceptron (MLP) and the structure is depicted in Figure 2.1. The weights are represented as the connections between the neurons. Such a network is also called a *feedforward neural network* since the activation flows forward to the direction of the output. An MLP can be used for problems such as regression and classification. In each of these tasks, the goal is to learn the underlying rule in the data. Similar to the Perceptrons, MLPs can be optimised by training on a set of examples.

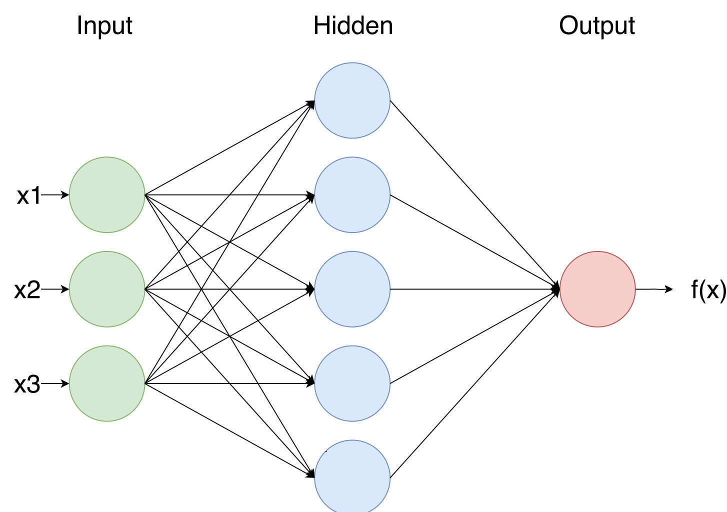


Figure 2.1: An example of an MLP. It consists of 3 input units, 1 hidden layer holding 5 units and 1 output unit.

2.4 NEURAL NETWORK OPTIMISATION

Ultimately the goal of Neural Networks is to derive the underlying structure of the data. When a neural network is presented with an example, it will make a prediction y. This prediction y can be compared with the actual target t. In general, we want to minimise the difference for each example between the prediction y that the neural network makes and the actual target t. In Machine Learning, this is often referred to as minimising a loss function.

2.4.1 *Gradient Descent*

Gradient Descent (GD) is an iterative optimisation algorithm which attempts to minimise such a loss function in order to update the parameters in machine learning models [8, 9]. Specifically, GD can be used to update the weights in a Neural Network. Since this is an iterative algorithm, the idea is to perform steps in the direction of the steepest decent of the weight space.

Since GD takes steps in the direction of the minimum, it is important to define how big those steps are. In literature this step size is often referred to as the learning rate η. If η is too large, it is possible that we overshoot the minimum. This allows the system to osculate around the minimum or even diverge from it. If η is too small, it can take a long time for the system to converge to a local minima.

2.4.2 *Other optimisation algorithms*

There are also other optimisation algorithms such as Stochastic Gradient Descent (SGD) [40], RMSProp [51] and Adam [22]. SGD differs from GD since it uses an approximation of the error gradient by computing the loss over a random subset of data. An advantage of this method is that it can be computationally much more efficient if a relatively small subset is chosen. The downside is that the error landscape is also approximated, which can lead to undesirable steps into the error landscape. RMSProp and Adam make some small changes which restrict how one should travel in the error landscape. RMSProp divides the learning rate by an exponentially decaying average of squared gradients. Adam is similar to RMSProp, but also makes use of exponentially decaying average of gradients .

2.5 CONVOLUTIONAL NEURAL NETWORK

In some cases the Multi-Layer Perceptron (MLP) is unsuitable for a specific regression or classification task. When the prediction depends on local spatial coherence of the data, MLPs generally perform poor. It is important to note that it still possible for an MLP to learn a

specific regression or classification task when the input images are of sufficient low-dimensional space. Alternatively, Convolutional Neural Networks (CNN) can take local spatial coherence of the input into account [26]. This is achieved by Convolutional Layers and optionally Pooling Layers. A discussion on these layers are found in Section 2.5.1 and Section 2.5.2.

2.5.1 *Convolutional Layer*

Convolutions in a Neural Network are performed to detect specific features in a set of data. A convolution is defined as performing an element-wise product with a filter (or kernel) after which a sum is taken to obtain a single value. The output of a convolution provides an indication whether a specific feature is present in that image. The higher the value, the more likely the feature is located in that region. The filter is shifted over the entire image and results in a convolved feature.

The computation for a discrete 2D convolution is given by:

$$[f * g](m, n) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f(i, j) g(m - i, n - j), \tag{2.4}$$

where f is the input image, g is the kernel. m and n denote the location of the convolved value in the resulting convolved feature. An example of a 2D convolution is shown in Figure 2.2.

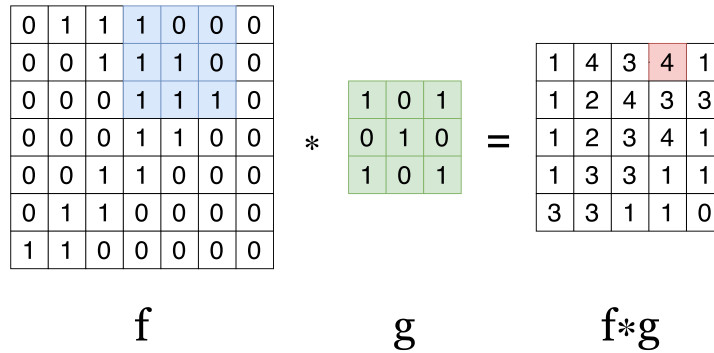


Figure 2.2: Convolving a 7x7 image f with a 3x3 kernel g results in a 5x5 convolved feature. Kernel g is shifted over the entire image and computes the element-wise product at each location in the image.

Convolutional Layers simply perform a convolution to the provided input and pass the output to the next layer.

2.5.2 *Pooling Layer*

Another layer that is often used in CNNs are pooling layers. Pooling Layers are often applied after using one or more convolutional layers

to prevent over-fitting by providing an abstracted form of the output representation. Pooling layers reduce the size of the resulting convolved feature. This process is also called downsampling. Many forms of pooling exists, such as maximum pooling and average pooling.

Pooling Layers use a sliding window over an input. In maximum pooling, the maximum value of the region of the input that coincides with the sliding window is considered the output for that location. Strides can be used to indicate how many positions the sliding window moves at a time. This allows for reducing the representation even more. Similarly, average pooling follows the same procedure, except that it takes the average of the input region that coincides with the sliding window. An example of maximum pooling and average pooling can be found in Figure 2.3.

Input

1	7	3	3
6	2	6	8
6	3	3	2
3	4	3	0

Max

7	8
6	3

Average

4	5
4	2

Figure 2.3: Performing maximum pooling and average pooling with a 2x2 stride.

2.5.3 *Transposed Convolutional Layer*

The convolutional layers and pooling layers in CNNs cause the input of the data to be reduced (downscaling) or kept the same. Transposed Convolutional Layers are used to upsample an input feature map to obtain a feature map of higher spatial dimensionality. An example where transposed convolutional layers are used are decoders in autoencoder systems. In such systems the goal is to reconstruct the original input image from a smaller representation. A more detailed explanation on Transposed Convolution Layers is described in [12].

2.6 GENERATIVE ADVERSARIAL NETWORKS

In generative models the aim is to learn a probability distribution over some data. This specification allows for generating new samples from the learnt distribution. It is possible to test the performance of these models by observing what the samples from such a distribution look like. One example of a generative model are variational autoencoders as discussed in section 3.2.2. These variational autoencoders learn a la-

tent variable model from which samples can be generated. Generative Adversarial Networks (GANs), a relatively new class of generative models, have been coined by Goodfellow et al, [17]. GANs consists of two parts, a generator and a discriminator network.

2.6.1 *Discriminator*

A discriminator can be seen as a simple classifier. It estimates the probability that the data came from the training set consisting of real data rather than from the fake generated data. The task of the discriminator is to correctly distinguish between the real and the fake samples.

2.6.2 *Generator*

The goal of the generator is to capture the overall data distribution. By doing so, it could generate samples which fool the discriminator by stating that they came from the original data distribution. New samples are formed by constructing a noise vector from which the generator attempts to reconstruct the data.

2.6.3 *Combining the Discriminator and Generator*

The discriminator and generator compete with each other in order to improve their individual task. The discriminator stimulates the generator to adapt based on how well the generator fools the discriminator. In turn the discriminator is adapted to improve its classification capabilities concerning real and fake samples. In each iteration both the generator and discriminator are updated to ensure that they are continuously improving. An overview of the GAN architecture is depicted in 2.4.

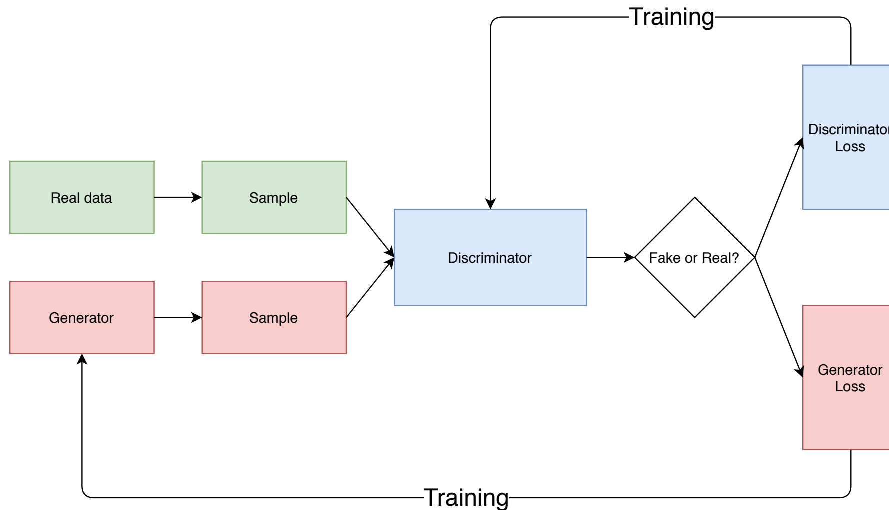


Figure 2.4: Overview of the Generative Adversarial Network.

Limitations

Although the concept of GANs are rather intuitive, they are notoriously hard to train. GANs are susceptible to problems including vanishing gradients, mode collapse, overfitting, sensitivity to hyperparameter initialisation and non-convergence issues [4]. There is a lot of ongoing research in the field of generative modelling that aims to stabilise GAN training. Two versions of GANs and their implications are discussed in section 2.6.4.

2.6.4 *Training a GAN*

The vanilla version of GANs utilises the Jensen-Shannon (JS) Divergence to measure the similarity between the training data and generated samples [17]. This version could be considered as a minimax game in which the generator tries to fool the discriminator and the discriminator classifies the data as fake or real. In each training step, both the generator and discriminator are updated. The discriminator is adapted to maximise its accuracy while the generator is adapted to minimise the accuracy of the discriminator. The minimax objective that the discriminator D and generator G collectively want to reach is:

$$\min_G \max_D \mathop{\mathbb{E}}_{x\sim\mathbb{P}_r}[\log(D(x))] + \mathop{\mathbb{E}}_{\tilde{x}\sim\mathbb{P}_g}[\log(1 - D(G(z)))], \tag{2.5}$$

where $\mathbb{P}_r$ is the distribution of the real training data and $\mathbb{P}_g$ is the distribution of the generated fake data.

Another version of a GAN is the Wasserstein GAN (WGAN) [3]. WGANs differ from vanilla GANs by changing the role of the discriminator. The discriminator no longer decides whether the sample came from the original distribution, but instead acts as a critic. Generally

speaking high values for the discriminator now indicate that the data is real and small values indicate fake data.

WGANs aim to solve the problem of non-continuity in the parameters of the generator caused by the JS-divergence. As an alternative, Earth-Mover distance is used which looks at how much mass should be transported from one distribution P to the other Q to transform P to Q. The benefit of this approach is that it is fully differentiable. WGANs still make use of the minimax objective although it can be simplified to:

$$\min_G \max_D \mathop{\mathbb{E}}_{\boldsymbol{x}\sim\mathbb{P}_r}[D(\boldsymbol{x})] - \mathop{\mathbb{E}}_{\tilde{\boldsymbol{x}}\sim\mathbb{P}_g}[D(G(z))]. \tag{2.6}$$

Research has shown that WGANs are much more robust against problems such as mode collapse and it allows the generator to still learn properly even when the critic is powerful and well-trained.

3 DIMENSIONALITY REDUCTION TECHNIQUES

The aim of this chapter is to explain several dimensionality reduction (DR) techniques. DR concerns itself with finding or constructing a new subset of features that can describe the data with fewer dimensions. A large portion of this chapter is dedicated to a specific class of DR techniques, namely autoencoders. There exist many flavors of autoencoders each specialised to invoke constraints on the learned data encoding. The context for DR techniques and therefore the use of autoencoders in this research is to prevent the overfitting problem that is present in high-dimensional inverse reinforcement learning problems. However many other reasons to use autoencoders exist including data visualization, speeding up computation time in algorithms or data storage.

3.1 STANDARD DOWNSAMPLING

One of the easiest ways to reduce the dimensionality of data is to perform downsampling. In statistical signal processing, downsampling is a process of compression in which an approximation is obtained by lowering the sampling rate of a signal. Any type of data can be considered to be a signal, hence downsampling can be applied to a variety of different data types including images. In the case of images, downsampling simply lowers the resolution by throwing away a number of pixels. For example, downsampling an image by two means that every other pixel in both the width and the height of the image is thrown away. The new obtained image is therefore an approximation of the original image. Although it is farily easy to apply, the downside of this approach is that the lost data cannot be recovered.

3.2 AUTOENCODERS

Another way to reduce the dimensionality of the data is through the use of autoencoders [42]. Autoencoders are neural networks which are trained to learn a low dimensional representation of the original data. Normally an autoencoder consists of two parts: an encoder and a decoder. Both the encoder and the decoder make use of a separate neural network, for example a Multi-Layer Perceptron or a Convolutional Neural Network. The goal of the encoder is to embed the original input data to a lower dimension *latent* space. In turn, the decoder attempts to reconstruct the latent embedding back to

the original data. The reconstructions are useful since it allows for validating the effectiveness of the autoencoder by comparing it to the original data. A metric is then used to minimise the discrepancy between the original and the reconstructed data.

3.2.1 *Vanilla autoencoder*

One of the most simple autoencoders is the vanilla autoencoder. This autoencoder consists of three layers: one input layer, one output layer and one hidden layer. The input and output layer have the same amount of nodes and should match the dimensionality of the data. An embedding is learnt by training the system end-to-end by optimising a loss function. In many cases this means the minimization of the Mean Squared Error (MSE). Figure 3.1 demonstrates two autoencoders. The autoencoder on the left implements a multilayer perceptron and the autoencoder on the right makes use of convolutional neural network in order to deal with input images. Note that convolutional layers are used to obtain the lower dimensional latent space. By means of transposed convolutional layers a reconstructed image is obtained from the latent space representation.

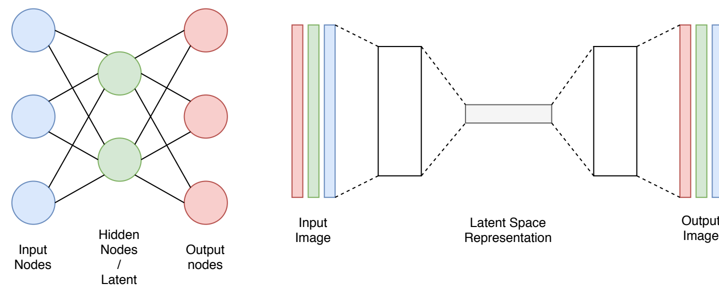


Figure 3.1: Left: An example of an autoencoder utilising a multi-layered perceptron. Right: An example of an autoencoder utilising a convolutional neural network.

3.2.2 *Variational Autoencoder*

Although the autoencoders as described above have proven to be useful for learning a more compact representation, there are also shortcomings to the vanilla autoencoders. One of the fundamental problems with these types of autoencoders is that there is no constraint on the learnt embedding space. This means that the embedding space might contain gaps between the datapoints. By sampling from a space that contains discontinuities, an unrealistic output can be obtained. Variational AutoEncoders (VAEs) are designed to construct latent spaces which are continuous [23]. One of the powerful features of VAEs is that they approximate the latent vectors z as probability distributions. VAEs generally model the latent space to be a centered normally

distributed Gaussian. A new latent sample can be constructed by sampling from the standard deviation vector and adding the mean to it.

The goal is to maximise the **E**vidence **L**ower **BO**und (ELBO) which is needed to approximate the posterior inference. The loss of a VAE is computed as:

$$\mathcal{L}(\theta, \phi; x, z) = E_{q_{\phi}(z|x)}[log p_{\theta}(x|z)] - D_{KL}(q_{\phi}(z|x)||p(z)), \quad (3.1)$$

where the first part is the reconstruction or generative loss and the second part is the latent loss which uses the KL-divergence to enforce normality on the latent vectors with mean 0 and standard deviation 1. Furthermore, x denotes the original data and z denotes the latent embedding. θ and ϕ are learnable parameters required to form a reconstruction.

Since VAEs provide more control over the latent embedding space they can be preferred over the use of vanilla autoencoders.

3.2.3 *Vector Quantised-Variational AutoEncoder (VQ-VAE)*

More recent research [54] combines concepts of Vector Quantisation (VQ) [18] with Variational Autoencoders. These types of autoencoders make use of discrete latent spaces. According to the authors many things in the real world are discrete such as phonemes or classes of animals. It would not make sense to interpolate between these types of categorical data. Another benefit is that categorical distributions, generally speaking, are easier to model. Before going into the details of VQ-VAE, it is worth explaining how Vector Quantisation works.

Vector Quantisation

The goal of VQ is to partition the input space into L distinct regions. Each cell in L is called a Voronoi region and is represented by its respective codevector. The idea is to map n-dimensional vectors x into a finite set of n-dimensional codevectors. The finite set of n-dimensional codevectors is often referred to as a codebook. To find the codevector to which an input pattern relates, a search metric such as Euclidean distance can be used. The input pattern is assigned to the closest codevector in the codebook.

Learning happens by updating the location of the codevectors. Initially the codevectors are distributed at random or according to some metric which ensures that the codevectors are spread evenly. For each iteration, the input patterns are assigned to the closest codevector. The update step is implemented by moving each codevector to the average location of the input patterns corresponding to that codevector. This step is repeated until a stopping criterion is met. For example, one could stop whenever the change of codevectors is smaller than a specific threshold.

Combining VQ with Autoencoders

The novelty in this type of autoencoder is that it utilises a quantisation layer to build a discrete latent space alongside the standard autoencoder algorithm. The overall architecture is shown in Figure 3.2.

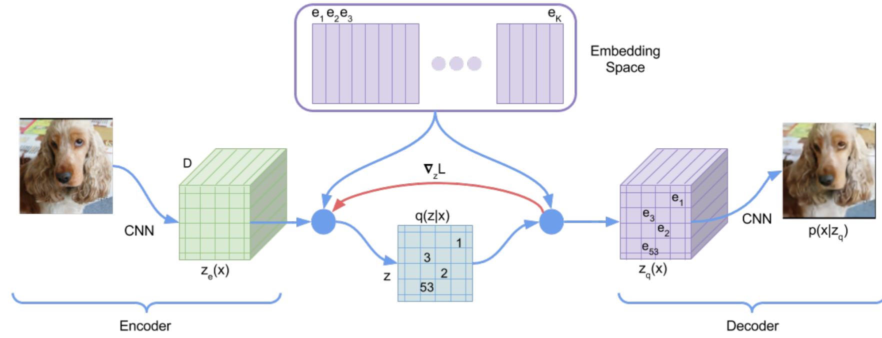


Figure 3.2: A visualization of VQ-VAE acting on image data. Reprinted from [54].

OBTAINING THE LATENT REPRESENTATION The latent embedding space is defined by $e \in R^{KxD}$, where K represents the size of the latent space that follows from the encoder and D is the dimensionality of the latent codevector. An input pattern x is mapped to this space by the encoder resulting in $z_e(x)$. The quantisation layer in turn maps the output of the encoder to R^K by committing to the closest codevector e_j in the codebook given by:

$$q(z = k \mid x) = \begin{cases} 1 & \text{for k } = \text{argmin}_j \left\| z_e(x) - e_j \right\|_2 \\ 0 & \text{otherwise} \end{cases} . \tag{3.2}$$

This shows the strength of the approach by projecting the embedding space to a much lower dimensional space in which only the indices of each codevector are considered.

TRAINING THE SYSTEM Since the quantisation layer makes use of the argmin operation it is not possible to propagate the gradients back through the system directly. One way to deal with this is to copy over the gradients from z_q back to z_e. The stop gradient operator $sg[\cdot]$ is used in the computation of the loss function to restrict the flow of gradients where they cannot be computed.

The authors use three loss functions in order to train the system end-to-end. First, similar to the vanilla autoencoder, the reconstruction loss (or MSE) is used to optimise the encoder and the decoder:

$$\mathcal{L}_{\text{reconstruction}} = -\log\left(p\left(x \mid z_q(x)\right)\right) . \tag{3.3}$$

Second, a codebook loss is implemented to minimise the difference between the embedding vectors and the encoder output using $l2$ regularisation:

$$\mathcal{L}_{\text{codebook}} = \|sg\left[z_e(x)\right] - e\|^2 . \tag{3.4}$$

This is basically how the vanilla VQ algorithm works.

Last is the commitment loss. The commitment loss causes the output of the encoder to be relatively close to the embedding space. It also helps with committing to a specific codevector by minimising fluctations between multiple codevectors in the codebook. The commitment loss is given by:

$$\mathcal{L}_{\text{commitment}} = \beta \|z_e(x) - sg[e]\|^2 , \tag{3.5}$$

where β is a parameter that indicates how important the commitment loss is compared to the other two losses. The authors chose $\beta = 0.25$ and have shown robust results when varying β between 0.1 and 2.0.

Combining the loss functions yields:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{reconstruction}} + \mathcal{L}_{\text{codebook}} + \mathcal{L}_{\text{commitment}}. \tag{3.6}$$

Since this involves a minimisation process, it is impossible to propagate the gradients back through the quantisation layer.

STATE-OF-THE-ART In further research, VQ-VAE has proven to be suitable for large scale image generation [39]. Other applications include music generation and compressing speech while maintaining high reconstruction quality [11, 16].

4 REINFORCEMENT LEARNING

Neural Networks are able to approximate an arbitrary decision boundary. As such they can also be combined with Reinforcement Learning (RL) algorithms. In a standard RL setting, an agent is situated in an unknown environment and has no apriori knowledge on how to behave in the environment. The goal of the RL agent is to derive a policy or strategy that allows the agent to maximise its future rewards. By interacting with the environment an agent receives rewards which can be used to adapt to the environment by altering its strategy. Figure 4.1 illustrates the typical RL cycle. The purpose of this section is to provide an overview of the basics of RL and the parts that are relevant for this thesis. A more complete overview can be found in [56], [49] and [25].

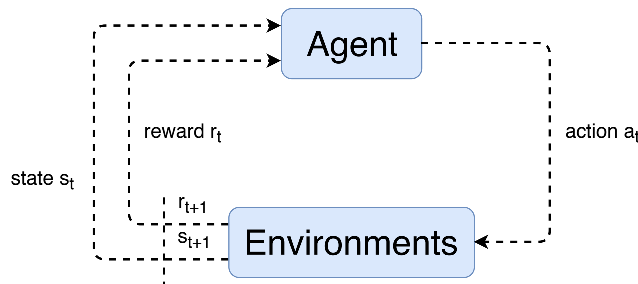


Figure 4.1: The typical Reinforcement Learning cycle. An agent performs an action a_t in state s_t. Upon taking that action it receives a reward r_{t+1} and ends up in a new state s_{t+1}.

The next section provides a formal description of the key concepts in RL.

4.1 KEY CONCEPTS

ENVIRONMENT The agent is situated in an environment. In this environment, the agent can act and observe what the consequences are. Formally it defines the transitions from one state to another upon taking an arbitrary action a.

STATE SPACE The state space S is the set of states in an environment that an agent can observe. These states can take many forms. States can be discrete and continuous. An example of a continuous state are the joint positions of a robotic arm or the pixel information of a video-game. An example of a discrete state might be the configuration of a chess board.

ACTION SPACE The action space A is the set of possible actions that the agent can take. An example related to the robotic arm is the ability to control the robotic arm by utilising the stiffness of the muscles. The stiffness could be represented by a continuous value. Another instance are the actions one could perform on an Atari console. Some Atari games use a joystick which is able to move to any of the 8 cardinal directions such as represented on a compass. Additionally the joystick has a neutral position when no direction was indicated. This results in a discrete action space of length 9.

REWARD FUNCTION Reward functions are required to provide stimuli to agents which allows them to be aware of the consequences of their actions. A reward function R can be associated with both a state S ($R : S \to \mathbb{R}$) or with a state-action pair ($R : S \times A \to \mathbb{R}$). A reward function can provide positive and negative rewards. Positive rewards can be provided when the agent performed desirable behaviour. On the other hand, negative reward can be provided when the agent acts undesirably in the environment. In some cases negative rewards are distributed to encourage an agent to reach a terminal state as quickly as possible. The agent attempts to minimise these accumulating penalties obtained from the negative rewards by finding a solution that requires the least amount of steps.

The sum of discounted future rewards, also called return, is defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{4.1}$$

where R_{t+1} is the reward at time $t+1$ and γ is the discount factor.

DISCOUNT FACTOR The discount factor γ ($0 \leq \gamma \leq 1$) is a meta-parameter that denotes how important future rewards are. A value close to 0 leads to choosing actions that only maximise the immediate reward. A discount factor close to 1 is preferred when rewards in the distant future are preferred. In general, the closer the discount factor γ is to 1, the further the rewards will propagate through time. As seen in Equation 4.1, the discount factor is used to discount all the rewards that are accumulated over time. The further the reward is away, the less influence it has on the overall return due to the exponentiation of the discount factor.

MARKOV DECISION PROCESS Markov Decision Processes (MDPs) allow systems to act optimally when faced with uncertainty. Formally an MDP is defined as a 5-tuple consisting of states, actions, transition probabilities, rewards and a discount factor. A process is Markovian if the transition probability distribution of the next state only depends on the current state and action. Formally this means that $\mathrm{P}(s_{t+1}|s_t, a_t)$.

POLICY The policy defines the behaviour of an agent. Formally it maps each observed state to a particular action $\pi : S \rightarrow A$. In RL the aim is to iteratively adjust the policy of the agent such that it maximises the sum of discounted rewards.

OPTIMAL VALUE FUNCTION There are several ways to compute an optimal policy. A policy is optimal if there does not exist another policy that returns a higher expected sum of discounted rewards. There can exist multiple optimal policies if the expected sum of discounted rewards is the same.

A state value function $V_\pi(s)$ roughly indicates how good it is to be in a specific state. $V_\pi(s)$ effectively returns the discounted sum of rewards when starting in state s following policy π. The computation for this value function is given in Equation 4.2.

$$V_\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| s_0 = s, \pi \right] \tag{4.2}$$

The state-action value function as given in Equation 4.3 computes the discounted sum of rewards when performing action a in state s after which policy π is followed.

$$Q_\pi(s,a) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| s_0 = s, a_0 = a, \pi \right] \tag{4.3}$$

The goal is to maximise both value functions. This is achieved by considering the policy π that yields the highest sum of discounted rewards. The computation for the *optimal state value function* is given in 4.4.

$$V^*(s) = \max_\pi V_\pi(s) \tag{4.4}$$

In Equation 4.5, the *optimal state-action value function* is computed by considering the policy that yields the highest sum of discounted rewards.

$$Q^*(s,a) = \max_\pi Q_\pi(s,a) \tag{4.5}$$

4.2 REINFORCEMENT LEARNING ALGORITHMS

A wide variety of methods exist to find an optimal policy. Two approaches are value iteration [6] and policy iteration [19].

Value iteration starts with a random value function and makes use of the Bellman equation to find the optimal value function. To achieve

this, an iterative process is performed to find new value functions. In Equation 4.6, the update for the state value function is given.

$$V_{i+1}^{\pi}(s) = \max_{a} \left[R(s,a) + \gamma \sum_{s' \in S} p\left(s' \mid s,a\right) V_i(s') \right] \tag{4.6}$$

Equation 4.7 shows the update for the state-action value function.

$$Q_{i+1}^{\pi}(s,a) = R(s,a) + \gamma \sum_{s' \in S} p\left(s' \mid s,a\right) Q_i(s',a') \tag{4.7}$$

Once the optimal value function is determined, it is trivial to obtain the optimal policy. Equation 4.8 shows how to recover the policy using the obtained state value function.

$$\pi(s) = \arg\max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} p\left(s' \mid s,a\right) V(s') \right] \tag{4.8}$$

In Equation 4.9 the policy is recovered using the obtained state-action value function.

$$\pi(s) = \arg\max_{a \in A} Q(s,a) \tag{4.9}$$

Policy Iteration differs from Value Iteration by also updating the policy at each step. This is achieved in two steps. The first step recovers the value function given a policy (policy evaluation). The second step is to find a new policy based on this previous value function. These steps are repeatedly executed to obtain an optimal policy.

Value Iteration and Policy Iteration are often not a viable algorithm to derive the optimal policy. One limitation is that both algorithms require the model to be fully observable. This means that the transition probabilities (dynamics of the environment) and rewards should be known. Both Value Iteration and Policy Iteration also iterate over all the states in the state space S. If the state space is large it is not possible to find an optimal policy in reasonable time.

4.2.1 *Monte Carlo Methods*

Monte Carlo methods provide an alternative to Value Iteration and Policy iteration in order to learn value functions and optimal policies. These kinds of methods learn value functions on the basis of experiences (state-action pairs) sampled from the environment at hand. This means that the dynamics of the environment are not required to learn an optimal policy. Since learning is based on individual examples an

approximated value function is obtained as a result. A Monte Carlo update can be expressed as:

$$V(S_t) = V(S_t) + \alpha(G_t - V(S_t)), \tag{4.10}$$

where G_t is the sum of discounted future rewards and $\alpha \in [0, 1]$ is the learning rate. The return G_t is only known after the entire episode (once all the state-action pairs are presented). The result is a more practical approach to computing the value function.

4.2.2 *Temporal Difference Learning*

TD(0) Temporal Difference (TD) Learning differs from Monte Carlo methods by using a single state-action pair to approximate the value function [48]. The update equation is adapted to take the following form:

$$V(S_t) = V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t))\right]. \tag{4.11}$$

This form of TD learning is called TD(0) or one-step TD since only one state-action pair is used to update the value function. Note that TD(0) is similar to Monte Carlo, except that the update takes place after every step.

Q-LEARNING An off-policy TD control algorithm is Q-Learning [55]. Q-Learning is used to obtain the Q-function which denotes the utility of performing action a in state s. The Q-function approximates the optimal state-action value function independent of the policy that is followed. The Q-function is updated according to the following Equation:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)) \tag{4.12}$$

SARSA SARSA [43] is an on-policy TD control algorithm. It uses experiences of the form $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ to update the state-action value function $Q(s, a)$ rather than using state-action pairs. Equation 4.13 shows how the Q-function is updated.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{4.13}$$

A more elaborate explanation on TD learning and other forms of TD learning are discussed in [49].

5 STATE-OF-THE-ART DEEP REINFORCEMENT LEARNING

Humans are able to adapt well to various situations they encounter. In normal situations, complex problems such as grabbing an object with your hands or driving a car without causing an accident can be tackled rather easily by a human. Control tasks or high-level cognitive tasks as such are not trivial to solve using an artificial construct. The goal of this chapter is to describe the recent advances of Deep Reinforcement Learning (DRL). DRL utilises deep learning to solve complex RL problems.

The first part focuses on the Deep Q-Network (DQN), an extension on Q-learning. The second part describes several policy gradient methods to obtain policies. Specifically, Deep Deterministic Policy Gradient (DDPG), Trust Region Policy Optimisation (TRPO) and Proximal Policy Optimisation (PPO) are discussed.

5.1 DQN

In 2013, researchers successfully implemented a Deep Neural Network that is able to learn control policies directly from pixel data using RL [32]. Their approach, DQN, was able to play Atari 2600 games reasonably well, outperforming human players in some games using only pixel information.

DQN differs from online Q-Learning by employing an Experience Replay (ER) buffer in combination with Convolutional Neural Networks and optionally a target network. An ER buffer accumulates the experiences (s_t, a, r, s_{t+1}) of the agent at each time step. At each step samples are taken uniformly from the ER buffer, yielding a list of experiences. On each of these experiences a Q-Learning update step is performed. Finally the algorithm chooses an action based on the ϵ-greedy strategy similar to Q-Learning.

There are several advantages with this approach over the standard Q-Learning approach. Firstly, it allows the algorithm to be more data efficient as the experiences in the ER buffer are reused multiple times to adjust the weights of the network. Secondly, there are less correlations between the experiences that the network is trained on. In Q-Learning, the experience that the network is trained on depends on the previous experience. DQN takes random experiences and therefore there is less correlation between subsequent experiences. This leads to a decrease in variance in the Q-Learning updates.

5.2 POLICY GRADIENT ALGORITHMS

Another approach to solving Reinforcement Learning problems is Policy Gradient (PG) [50]. Policies in PG algorithms are represented by a parametric distribution $\pi_\theta(a|s) = [\mathbb{P}][a|s;\theta)$ and actions are stochastically selected according to the policy parameters θ. Many approaches, including DQN, are value-based methods which use a value-function to both optimise the policy and perform action selection. Similar to DQN, PG algorithms use value functions to model and optimise a policy. However, PG algorithms do not require a value function to perform action selection. By interacting with the environment, one is able to continuously update the parameters of the model θ until the network converges to the optimal policy π^*. Updating the model using PG methods is done by ascending the gradient of the policy with regards to a local maximum $J(\theta)$:

$$\Delta\theta = \alpha \nabla_\theta J(\theta), \tag{5.1}$$

where $\nabla_\theta J(\theta)$ is the policy gradient. There are no limitations on the parameterisation of the policy as long as the policy is differentiable.

PG has several advantages over value function approaches. Dependent on the task, sometimes it is easier to solve the problem by learning a policy directly rather than solving it by using a value function. In general PG approaches have better converging properties. Value functions are known to oscillate or diverge. PG approaches perform small updates to the policy directly which leads to a much smoother learning trajectory. PG is more effective in continuous and high dimensional spaces. Another advantage is that PG allows for learning stochastic policies, which is useful to learn a set of tasks which can otherwise not be solved optimally by maximising a value function. As PG methods are stochastic, it is possible to learn probabilities for taking actions rather than learning a deterministic action by inheriting randomness in the model.

One downside from vanilla PG algorithms is that the variance of the gradients is high. Other PG algorithms attempt to reduce the variance.

5.2.1 *REINFORCE*

REINFORCE is one of the most basic policy gradient based approaches [57]. The algorithm uses episode samples to update the policy Θ in an online manner. This is achieved by estimating Q by using an unbiased return sample. Using this information, an update to the parameters is performed in the direction of the stochastic gradient:

$$\Delta\theta_t = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) G_t. \tag{5.2}$$

A baseline can be used to reduce the variance in a PG system. No bias is introduced when adding a baseline. This means that a value function is used in the computation of the gradient. With a baseline, REINFORCE can converge to a local minimum but is slow and produces high variance estimates. This is due to the nature of Monte Carlo methods.

5.2.2 *Actor-Critic Methods*

Actor Critic methods utilise two approximators to learn a value function and a policy. Actor-Critic methods combine value function approximation with policy methods. While the vanilla PG approach uses the return to estimate the action value function, now a critic is used to estimate the value function. To estimate this function, a policy evaluation algorithm should be considered such as temporal-difference learning.

In an Actor-Critic system there are two components. The critic is responsible for evaluating the decisions of the actor. The evaluation determines how to adapt the policy parameters (actor) in the direction suggested by the critic. The actor is the component that acts in the environment by choosing the actions.

Actor-Critic methods share some similarities with the REINFORCE algorithm. Both approaches use value functions as a baseline. However Actor-Critic methods use bootstrapping while REINFORCE does not [49].

5.2.3 *Off-Policy Actor Critic methods*

Actor-Critic Systems can also be modelled off-policy. This means that the policy gradient is estimated using a separate behaviour policy $\beta(a|s)$ rather than the using the target policy $\pi_\theta(a|s)$ itself. The behaviour policy is also used to generate trajectories. The critic in turn evaluates the state value function from these sampled trajectories in an off-policy manner. Gradient temporal difference learning is used for this. The actor also makes use of these trajectories off-policy by updating the policy parameters θ by means of stochastic gradient ascent. Importance sampling is used to correct for the mismatch between the behavioural policy and the target policy $\frac{\pi_\theta(a|s)}{\beta_\theta(a|s)}$.

5.2.4 *DDPG*

Deep Deterministic Policy Gradient (DDPG) [28] is a model-free off-policy Actor-Critic system that is able to learn policies in high-dimensional and continuous action spaces and is based on Deterministic Policy Gradient (DPG) [47]. Unlike the previous discussed PG

algorithms which are all stochastic, DPG is a deterministic algorithm. One important difference between stochastic and deterministic PG algorithms is that deterministic algorithms only integrate over the state space. Stochastic PG algorithms integrate over both the state and action space. Stochastic policies require more examples to estimate the policy gradient since integration is done over states and actions. Especially with high dimensional action spaces, computing the gradient becomes less efficient.

Deterministic policies have more difficulty to fully explore the state and action spaces when compared to stochastic policies. Deterministic Policy Gradient introduces an off-policy learning algorithm that chooses actions according to a stochastic exploration policy while learning is done with a deterministic target policy.

DDPG combines the advantages from DPG and DQN. DQN stabilises learning by introducing experience replay buffer. This buffer allows for training the function approximator off-policy while minimising the correlation between samples. Note however that DQN only works well in discrete spaces due to the maximisation step needed to obtain the highest Q. DDPG extends DQN to be able to work with continuous action spaces while learning a deterministic policy.

5.2.5 *TRPO*

A scalable model-free algorithm for optimising policies is Trust Region Policy Optimisation (TRPO) [45]. TRPO guarantees monotonic improvements when optimising a policy. As of writing, TRPO is considered one of the strongest baseline algorithms to test when implementing new policy gradient algorithms. TRPO relies on the use of Natural Gradient Descent. A conjugate gradient together with a line search is computed to obtain the gradient. Trust Region methods are used to constrain the update step to improve the policy. TRPO specifically considers the KL divergence, which computes the difference between the old and new policy parameters, as a measurement of trust. The objective function that is maximised given the constraint is:

$$\underset{\theta}{\text{maximise}} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right], \tag{5.3}$$

which is subject to

$$\hat{\mathbb{E}}_t \left[\text{KL} \left[\pi_{\theta_{\text{old}}} (\cdot \mid s_t), \pi_\theta (\cdot \mid s_t) \right] \right] \leq \delta. \tag{5.4}$$

Natural Gradient Descent

▷ Performance

when solving optimisation problems using standard gradient descent approaches is heavily dependent on the parameterisation of the model.

Gradient descent is an optimization strategy that minimizes some objective in an iterative fashion. Each iteration, the algorithm takes a step in the direction of the steepest descent. There is no guarantee that the network parameters before and after a gradient descent update are similar. In order to become invariant to model parameterisation, Natural Gradient Descent [20] can be considered. Using a similarity measure such as KL divergence, Natural Gradient Descent computes the similarity between the distribution of network parameters before and after the update and limits how different our distribution becomes when following the gradient. This ensures that the behaviour of the policy is similar before and after the update step which results in more stable learning.

Performance when solving optimisation problems using standard gradient descent approaches is heavily dependent on the parameterisation of the model. Gradient descent is an optimization strategy that minimizes some objective in an iterative fashion. Each iteration, the algorithm takes a step in the direction of the steepest descent. There is no guarantee that the distribution for action selection before and after a gradient descent update are similar. In order to become invariant to model parameterisation, Natural Gradient Descent [20] can be considered. Using a similarity measure such as KL divergence, Natural Gradient Descent computes the similarity between the distribution for action selection before and after the update and limits how different the distribution becomes when following the gradient. This ensures that the behaviour of the policy is similar before and after the update step which results in more stable learning.

Fisher Information

KL divergence provides a measure of similarity between two probability distributions. In order to compute the similarity between two distributions, The Fisher Information measure is used to compute the second derivative or Hessian of the KL divergence.

Surrogate Loss

TRPO utilises a surrogate loss function to optimise the policy. First the likelihood ratio between the new and old distribution parameters is computed. Then the likelihood is multiplied by the estimated advantage. The loss is obtained by taking the negative mean of the previous result.

Conjugate Gradient

Optimisation of the policy is done via the conjugate gradient method and line search. The value function is updated by means of stochastic gradient descent. Conjugate Gradient aims to approximately solve $x = A^{-1}b$, where A is a symmetric positive-definite matrix. This

matrix is given by the Fisher information matrix. An iterative process is started to minimise $\frac{1}{2}x^T Ax - bx$.

Trust Region Methods

Optimising objectives can be done using line search methods and trust region methods. Line search finds a direction to improve the policy by means of gradient descent using a selected step size. In contrast, trust region methods first select a trust region. A trust region caps the maximum step length. Then the goal is to search the region around our estimate to find a point which optimises the object. Trust region methods approximate the objective f with a simpler objective f, and only the region is considered that approximates the original objective f close enough. This simpler objective is achieved by not using the new state distribution frequency, but maintaining the old state visitation frequency.

5.2.6 *Proximal Policy Optimisation*

Proximal Policy Optimisation (PPO) [46] aims to solve some of the limitations of other RL algorithms. Vanilla PG approaches fail to use data efficiently and learning is not robust. The data inefficiency is caused by only training on the data that is generated by the policy. After a single policy gradient update the old data is no longer used. Learning is not robust since there is no constraint on the gradient step size. A step size too big leads to poor policies, while a small step size causes a slow learning process. Instead, PPO attempts to improve on the work of TRPO by making the algorithm more scalable, robust and data efficient. To ensure reliability and stability, trust region methods are used. In PPO, policy optimisation is achieved in two steps. The first step is to sample data from the policy. The second step is to optimise the policy by multiple epochs of stochastic gradient ascent. As discussed, TRPO maximises a surrogate objective function and is hard constrained by the size of the update step. Instead of a hard constraint, PPO reformulates the objective function such that the problem is no longer a hard constraint optimisation problem. The authors of PPO propose two alternatives to the constraint. One alternative is to have an adaptive KL penalty coefficient. The other alternative is to use a clipped surrogate objective function as described below. Results indicate that using a clipped surrogate objective function yields better performance than using an adaptive KL penalty coefficient. In this research only the clipped surrogate objective function is considered. [46] describes the adaptive KL penalty in more detail.

Clipped Surrogate Objective

Consider the ratio between the new and the old policy $r_t(\theta)$. The next step is then to approximate the long-term reward η by optimising the surrogate objective function:

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t] = \hat{\mathbb{E}}_t[r_t(\theta)\hat{A}_t]. \quad (5.5)$$

Maximising this objective without a constraint causes the update step to be very large. Note that CPI stands for conservative policy optimisation. PPO proposes to optimise the following function instead which penalises updates which are too large:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)], \quad (5.6)$$

where ϵ is a hyperparameter used to clip the probability ratio $r_t(\theta)$. Note that $L^{CLIP}(\theta)$ is a lower bound of $L^{CPI}(\theta)$. Optimising the objective function for PPO is less computationally expensive compared to the objective function for TRPO.

6 INVERSE REINFORCEMENT LEARNING

Reinforcement Learning (RL) provides powerful set of tools to learn how to operate in an environment through trial and error.

As discussed in Chapter 4, the goal of RL is to derive an optimal policy which specifies what action is to be performed given an arbitrary state. Deriving such a policy is achieved by allowing an agent to carry out a set of actions in an environment. By providing rewards or punishments to the agent, it is possible to reinforce its behaviour or policy. These rewards and punishments are captured in a reward function, which provides a specification of how well an agent is performing given the state that the agent is in.

This section discusses several methods to obtain a policy without the need of reward engineering. The main purpose is to provide an overview of different Inverse Reinforcement Learning (IRL) algorithms. Before going into the several IRL algorithms, a short discussion on Imitation Learning is given to bridge the gap between Imitation Learning and IRL.

6.1 IMITATION LEARNING

One way to learn behaviour is to clone the behaviour of an expert directly. One way to achieve this is through Behavioural Cloning (BC) [31, 38]. BC attempts to learn for each state what action has to be performed in a supervised learning manner. This is achieved by sampling state-action pairs from trajectories or another source of demonstrations and training a supervised learner to minimise the discrepancy between the output of the learner and the desired action. There are several drawbacks using this approach. First, there is no reasoning about the outcome and dynamics in behavioural cloning. The actions are simply copied from the demonstrations of the expert. This can lead to poor generalisation when faced with an unseen state. Second, BC has only proven to be effective in applications where no long-term planning is required. Considering the limitations, it can be desirable to reason what the goal of the expert is rather than copying the actions blindly. The following section is dedicated to Inverse Reinforcement Learning which aims to alleviate these limitations.

6.2 INVERSE REINFORCEMENT LEARNING

Inverse Reinforcement Learning (IRL) attempts to solve the problem of deriving reward functions automatically when only expert demon-

strations are readily available [35, 44]. Generating reward functions using IRL methods is generally more useful than Behavioural Cloning, as discussed in the previous section, since it allows for better generalisation in environments. An agent trained using Behavioural Cloning is able to store examples from the training demonstrations, but is likely to fail in situations that are not captured in the training data. IRL methods attempt to derive what the goal of the environment is given the expert demonstrations.

There are several reasons to study the field of Inverse Reinforcement Learning.

- To understand that IRL inverts the RL problem. RL derives a policy given a reward function while IRL tries to reconstruct a reward function learning from expert demonstration. In some cases, expert examples are easier to capture compared to providing a sensible reward function which leads to desirable behaviour. Manually defining a reward function limits the applicability of the model.
- It is possible to model the rewards more closely to the expert demonstration in contrast to manually specifying such a reward function.
- A reward function is a representation that describes the preferences of an agent in a well-defined manner. In [44], the authors claim that a reward function is more transferable than a learnt policy. A small change in the environment causes the policy to collapse, however a reward function might be able to generalise to different environments.
- In some tasks it is beneficial to capture the preference of an expert in the system. In autonomous vehicle control tasks, it is not possible to start with tabula rasa initialisation and learn a policy by exploration. This would be a dangerous procedure if there are no measures to prevent collisions.

6.2.1 *Definition IRL*

Consider a MDP (S, A, T, γ) without a reward function. Given a finite state space S, a set of action A, the transition probabilities $T_{sa}(s'|s, a)$ describe the probability of ending up in s' when performing action a in state s, a discount factor γ, and a policy π; the goal is to find the set of reward functions R which lead to an optimal policy π^* in the specified MDP.

6.2.2 *Limitations of IRL*

Before going into the specific IRL algorithms it is important to describe some of the problems associated with IRL algorithms. One of the problems is that most IRL algorithms are susceptible to reward shaping. This means that the recovered reward function contains a shaped component. As this component is dependent on the dynamics of the environment it is impossible to generalise to unseen environments. It is inevitable, without taking further measures, to obtain expert examples that do not contain any information about the environment that was used to extract them. This means that the IRL algorithm should take this into consideration when deriving a reward function. Reward shaping causes IRL to be an under-defined problem. There are multiple reward functions that can yield an optimal policy by means of standard RL for a specific environment.

Another challenge is that multiple policies can be found which explain the same set of expert examples. It is important to know that IRL algorithms assume that the policy of the expert is optimal with respect to the unknown reward function [37]. The policy is directly evaluated on the basis of the unknown reconstructed reward function.

6.3 MAXIMUM MARGIN OPTIMISATION

Many IRL algorithms make use of maximum margin optimisation (MMO) to find a reward function that yields the best possible policy by a margin. The problem of maximum margin can be formulated as:

$$\begin{aligned} \underset{w}{\text{minimize}} \quad & ||w||_2^2 \\ \text{subject to} \quad & w^T \mu(\pi^*) \geq w^T \mu(\pi) \forall \pi, \end{aligned} \tag{6.1}$$

where π^* is the optimal policy and $\mu(\cdot)$ is the expected cumulative discounted sum of feature expectations. This optimisation problem leads to a large number of constraints. Iterative Constraint Generation is used to solve each constraint. Maximum Margin approaches are known to introduce bias into the learned reward function. Entropy approaches attempt to avoid any bias.

6.3.1 *Entropy Optimisation*

Entropy Optimisation [21] methods make use of the maximum entropy principle. In an IRL setting, these methods try to obtain a distribution over possible reward functions and avoid bias.

Maximum Entropy IRL (MaxEnt IRL) [59] attempts to recover a distribution over all trajectories which maximises the entropy. The distributions that are considered are subject to the feature expectations of the policy and should match the expert data. By utilising a

probabilistic model of behaviour, MaxEnt IRL resolves the ambiguity that multiple reward functions map to the optimal policy and that multiple policies can lead to the same feature count.

Consider a trajectory $\tau = \{s_1, a_1, ..., s_t, a_t, ..., s_T, a_T\}$ drawn from the expert policy $D : \{\tau_i\} \sim \pi^*$. The reward of the trajectory is $R_\psi(\tau) = \sum_t r_\psi(s_t, a_t)$.

The probability of the trajectory under the expert is defined as an energy-based model for behaviour:

$$p(\tau) = \frac{1}{Z} exp(R_\psi(\tau)). \tag{6.2}$$

This indicates that the trajectories associated with a high reward are more likely to be sampled from our expert. Evaluating the partition function Z is relatively easy in low dimensional state spaces or when the dynamics are known. In high dimensional state spaces or if the dynamics are unknown, evaluating Z is hard. The reward function is in turn reconstructed by maximising the log likelihood of the set of demonstrations:

$$\begin{aligned}
\max_\psi \mathcal{L}(\psi) &= \sum_{\tau \in D} log\ p_{r_\psi}(\tau) \\
&= \sum_{\tau \in D} log\ \frac{1}{Z}\ exp(R_\psi(\tau)) \\
&= \sum_{\tau \in D} R_\psi - M\ log\ Z \\
&= \sum_{\tau \in D} R_\psi - M\ log\ \sum_\tau exp(R_\psi(\tau)),
\end{aligned} \tag{6.3}$$

where M is the number of expert trajectories. Gradient Descent is applied to optimise the objective. This results in:

$$\begin{aligned}
\nabla_\psi \mathcal{L}(\psi) &= \sum_{\tau \in D} \frac{dR_\psi(\tau)}{d\psi} - M \frac{1}{\sum_\tau exp(R_\psi(\tau))} \sum_\tau exp(R_\psi(\tau)) \frac{dR_\psi(\tau)}{d\psi} \\
&= \sum_{\tau \in D} \frac{dR_\psi(\tau)}{d\psi} - M \sum_\tau p(\tau|\psi) \frac{dR_\psi(\tau)}{d\psi} \\
&= \sum_{\tau \in D} \frac{dR_\psi(\tau)}{d\psi} - M \sum_\tau p(\mathbf{s}|\psi) \frac{dr_\psi(\mathbf{s})}{d\psi}.
\end{aligned} \tag{6.4}$$

Note that $p(s|\psi)$ is the state visitation frequency. This is the probability of visiting the state at any given time under the demonstration policy for a reward function parameterised by ψ. In [59], the authors describe an efficient way to compute the state visitation frequencies by approximating the state frequencies using a large fixed time horizon based on the value iteration algorithm.

The Maximum Entropy IRL algorithm has several steps:

1. Initialise the policy parameters ψ
2. Sample expert demonstration D
3. Derive the optimal policy $\pi(a|s)$ considering the current reward function r_ψ
4. Compute the state visitation frequencies $p(s|\psi)$
5. Compute the gradient $\nabla_\psi \mathcal{L}(\psi)$ of the objective as shown in 6.4
6. Update reward parameters using the gradient $\nabla_\psi \mathcal{L}(\psi)$
7. Repeat steps 3 to 6

6.4 GUIDED COST LEARNING

Guided Cost Learning (GCL) [13] attempts to overcome some of the issues of MaxEnt IRL by making the algorithm more robust towards larger state spaces. Additionally, GCL is able to deal with systems in which the dynamics are unknown, unlike MaxEnt IRL. This is achieved by making two changes.

- One change is to estimate the partition function rather than computing the partition function analytically. This is achieved by *importance sampling* which allows for sampling from a distribution whose probabilities are proportional to the absolute value of the exponential of the reward function. Since the optimal reward function is not known, adaptive sampling is performed to estimate the partition function. This means that the distribution changes when optimising the objective function.
- The second change is to remove the need for optimising the policy fully every time a new reward is computed. Instead, only one policy optimisation step is performed after each modification of the reward function.

The following section describes how Generative Adversarial Networks (GANs) are related to IRL and how they can be combined.

6.4.1 *GANs and IRL*

The strength of Generative Adversarial Networks (GAN) [17] is to learn an objective for generating data. The generator could be exploited as such that it attempts to generate data similar to data from expert demonstrations. In turn, the goal of the reward function is to associate high rewards to samples of the expert demonstrations and low rewards to samples from the generator. There are some noticeable differences between GANs and IRL on how they operate. Section 2.6 describes GANs in more detail.

- GANs operate on single samples while IRL algorithms operate on entire trajectories. Consequently, GANs generate single samples while IRL learns a policy that generates trajectories.
- IRL uses a reward function whereas GANs use a discriminator to discriminate between policy data and demonstration data

The next section described a method that unifies GANs and GCL.

6.4.2 *GAN-GCL*

GAN-GCL [14] optimises the following discriminator or reward function:

$$D^*(\tau) = \frac{p(\tau)}{p(\tau) + q(\tau)}, \tag{6.5}$$

where $q(\tau)$ is the probability of the trajectory coming from the policy and $p(\tau)$ is the probability of the trajectory coming from the expert distribution. Note that this could be rewritten to:

$$D^*(\tau) = \frac{\frac{1}{Z}exp(R_\psi(\tau))}{\frac{1}{Z}exp(R_\psi(\tau)) + q(\tau)}, \tag{6.6}$$

where $R_\psi(\tau) = \log \pi^*(\tau)$, and R_ψ is the approximation of the reward. A justification for this formulation is given in [15]. This means that updating the discriminator is directly linked to updating the reward function. The discriminator cross-entropy loss is given by:

$$\mathcal{L}_{discriminator}(\psi) = \mathbb{E}_{\tau\sim p}[-logD_\psi(\tau)] + \mathbb{E}_{\tau\sim p}[-log(1 - D_\psi(\tau))]. \tag{6.7}$$

The generator attempts to learn a distribution close to the expert data. The idea is that discriminator should not able to distinguish between whether the data is sampled from the policy or whether it came from the expert demonstrations. The loss of the generator or policy is computed as:

$$\begin{aligned} \mathcal{L}_{generator}(\theta) &= \mathbb{E}_{\tau\sim q}[log(1 - D_\psi(\tau)) - logD_\psi(\tau)] \\ &= \mathbb{E}_{\tau\sim q}[logq(\tau) + logZ - R_\psi(\tau)]. \end{aligned} \tag{6.8}$$

This is also called Entropy-Regularised Reinforcement Learning as both the expected reward and the entropy are maximised.

Although these algorithms have better scaling capabilities and can deal with unknown dynamics, there is still a reasonable amount of drawbacks. One drawback is that GANs are notoriously hard to optimise. Another drawback is that GAN-GCL and other IRL algorithms have difficulty scaling to high dimensional raw pixel data. This is due to the high variances occurring when approximating the reward functions since full trajectories are used in the optimisation process.

6.5 ADVERSARIAL INVERSE REINFORCEMENT LEARNING

As shown GAN-GCL operates on full trajectories. This can lead to high variance estimates. Adversarial Inverse Reinforcement Learning (AIRL) [15] reduces the variance estimates by considering single state, action pairs. By adapting Equation 6.5, the discriminator takes the form:

$$D^*(s, a) = \frac{p(s, a)}{p(s, a) + q(s, a)}. \tag{6.9}$$

The authors argue that the use of this form is suitable for imitation learning but not for learning a reward function. This formulation causes an entangled reward function which is not robust to changes in environment dynamics. This is also called the reward ambiguity problem.

6.5.1 *Reward Ambiguity Problem*

Consider the reward transformation,

$$\hat{r}(s, a, s') = r(s, a, s') + \gamma\Phi(s') - \Phi(s), \tag{6.10}$$

for any function $\Phi : \mathcal{S} \to \mathbb{R}$, the policy is optimal. Since IRL algorithms learn on the basis of expert demonstrations, it is hard to disambiguate between the reward functions inside this set of transformations. By restricting how reward functions are learned, it is possible to find reward functions that are robust to changes in the environment dynamics. By restricting the discriminator to operate only on the current state it is possible to remove reward shaping. A shaping term can be introduced which acts similar to how the advantage function works in Reinforcement Learning. This shaping term is only dependent on the state and prevents any unwanted reward shaping. This state-only version of AIRL prevents imitation learning from happening and allows for learning a representative unshaped reward function.

6.5.2 *Generalisation performance*

The authors have shown that AIRL can properly reconstruct reward functions when training on a set of of control tasks by using the techniques described above. To validate whether the reward function is unshaped, the authors mirrored the environment during testing. The agent still managed to do its task successfully indicating that the reward functions are robust to changes in dynamics.

6.6 CONVOLUTIONAL NEURAL NETWORK AIRL

Although AIRL was able to reconstruct unshaped reward functions for control tasks, these are considered to be relatively low-dimensional in state. Learning reward functions in high-dimensional state spaces such as the Atari video game environments is substantially harder.

6.6.1 *Arcade Learning Environment*

The Arcade Learning Environment (ALE) [5] is often used to benchmark algorithms on pixel data as it provides an interface to hundreds of Atari environments. In many of these environments, the state space is 210x160x3 resulting in 100,800 dimensions when considering that each dimension ranges from 0 to 255. Comparatively, Humanoid Mujoco, an environment in which the goal is to control a 3D robot to walk as fast as possible, only has 376 dimensions [52].

6.6.2 *Scaling to Pixel Data*

In [53], an attempt has been made to scale Inverse Reinforcement Learning techniques to higher dimensional pixel data. In order to achieve this, their work modifies the existing AIRL architecture in two ways. First by adapting the AIRL algorithm to support CNNs rather than MLPs for both the discriminator and generator architecture. Second by performing Proximal Policy Optimisation rather than Trust Region Policy Optimisation. This baseline is called CNN-AIRL. To stabilise training they propose several extensions to the CNN-AIRL baseline.

- The authors of [53] introduce a novel autoencoder which is tuned to work for video game environments. An autoencoder learns a low-dimensional embedding of the data. The embedded representation is fed to the reward network, allowing that the reward architecture is able to learn from fewer expert demonstrations. As only the discriminator is susceptible to the curse of dimensionality, the policy does not have to make use of the embedded representation, but can use the original pixel data instead.

- Another extension is to compute the mean reward and standard deviation over the last sample of trajectories that are used for training the discriminator. In that way the rewards can be centered and scaled. This makes sure that the order of the rewards are stationary upon updating the rewards. This is important as Policy Gradient techniques take gradient steps.

- Dataset expansion is required the prevent overfitting of the discriminator. To expand the dataset, the authors use the last k

rollouts of the forward RL step. This is done for each discriminator training step.

6.6.3 *Difficulties in Training*

- The authors note that the reward function is often discontinuous in video games. Hitting or missing a target with a gun is binary, even though a shot could have been off only by a few centimetres. Control tasks are in general more continuous as the reward function is often a function of speed (forward velocity in the case of Humanoid Mujoco) or distance from goal. It is therefore important that reward functions with sharp decision boundaries can be reconstructed.
- Training the discriminator is rather difficult. The training of the discriminator should be powerful; negative rewards are to be generated whenever the trajectories of the sampled policy deviate sufficiently from the demonstrations of the expert. However it should also be noted that the discriminator should not be too powerful. In that case the discriminator always discriminates the expert trajectories from the sampled policy trajectories. In literature this is referred to as the *discrimination-rewarding trade-off* [7].
- In order to find a good balanced discriminator one has to solve the curse of dimensionality problem. In high dimensional state spaces it is impossible to train on sufficiently many expert trajectories causing the discriminator to overfit. To solve this problem, dimensionality reduction techniques can be used.

6.6.4 *Limitations of AIRL*

- One of the limitations of AIRL is that it utilises Policy Optimisation algorithms such as TRPO and PPO although they do not provide the state-of-the-art results on many Atari environments. Instead, Q-Learning based approaches such as DQN often outperform Policy Gradient methods. However, Q-Learning based methods are sensitive to changes in reward functions after each discriminator update. This is undesirable for learning a stable policy. Therefore it is important to train these algorithms on environments which are known to be solvable by Policy Gradient approaches.
- Although autoencoders provide a way to reduce the dimensionality of the input data, they are trained to reconstruct the whole input data. One of the limitations is that a trained latent embedding also captures information that is irrelevant for constructing

the proper reward function. For example in Pong only the locations of the ball and the paddles are relevant. For future research the authors propose to learn an embedding using the Causal InfoGAN method as this takes into consideration the sequential nature of the data.

Part II

METHODOLOGY

7 GENERAL IMPLEMENTATION

This chapter describes the overall pipeline for the experiments. The goal is to describe the design decisions and how these will answer the research questions as stated in Section 1.4. First, a description of the data environment is given. Second, the three steps required for training the overall architecture are described consisting of collecting expert demonstrations, learning a low-dimensional representation of the data and training the Adversarial Inverse Reinforcement Learning (AIRL) algorithm. As discussed in Section 6.6, many IRL algorithms suffer from the curse of dimensionality in which the discriminator overfits on the high-dimensional data. In order to alleviate the problem of overfitting, the representation obtained from autoencoders can be used.

The goal of the experiments is to test whether AIRL can recover reasonable reward functions when challenged with high-dimensional environments. Specifically, low-dimensional representations from three different autoencoders are fed to the AIRL algorithm. One of the autoencoders constructs a discrete latent vector which might proof useful in solving the Atari environments.

7.1 ENVIRONMENTS

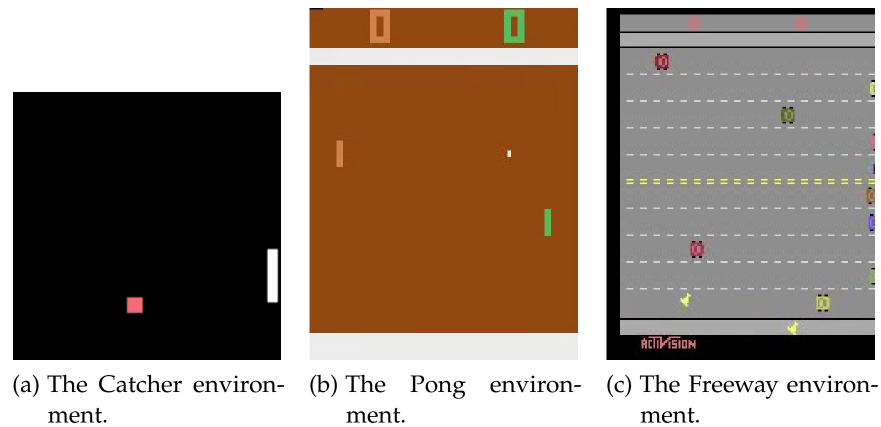
ARCADE LEARNING ENVIRONMENT The larger part of the experiments are performed on two Atari 2600 games from the Arcade Learning Environment (ALE) [5]. ALE is commonly used by researchers and provides access to a range of challenging high-dimensional pixel environments. In the experiments, the performance of the algorithms are evaluated on Pong and Freeway. In Figure 7.1 an example of the environments is displayed. Pong is visually relatively simple while Freeway is visually more complex. Both environments generate colour images of size 210x160x3 which are downscaled to 80x80 grayscale images. In order to train on these environments efficiently, an action is repeated for 4 consecutive frames. Due to the nature of Pong and Freeway, there is no need for framestacking. The expert can minimise the distance between the ball and the bat independent of the direction the ball is moving in. This means that the state at any given time is simply one 80x80 grayscale image.

PYGAME LEARNING ENVIRONMENT A small number of experiments are performed on the Catcher environment. Catcher is part of PyGame Learning Environment (PLE) and offers environments which

are visually less complex compared to the environments found in ALE. A frame of the Catcher environment is displayed in Figure 7.1. The generated images are of size 80x80 and are converted to grayscale such that frame representation is identical to the ones used for Pong and Freeway.

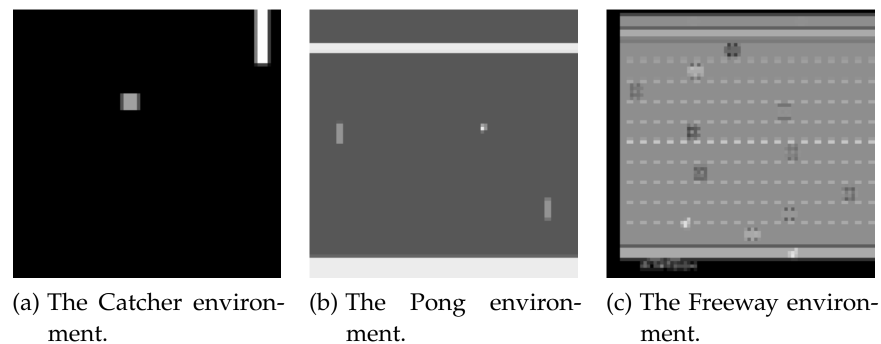
7.1.1 *Additional changes to Pong and Freeway*

There is no need to be aware of the score in order to play Pong and Freeway effectively. In order to accelerate the training process, the scores are replaced by the background colour of their environment. Figure 7.2 shows some example frames after the preprocessing steps.



(a) The Catcher environment. (b) The Pong environment. (c) The Freeway environment.

Figure 7.1: The Catcher Pong and Freeway environments.



(a) The Catcher environment. (b) The Pong environment. (c) The Freeway environment.

Figure 7.2: The processed frames used further in the pipeline.

7.2 ARCHITECTURE PROXIMAL POLICY OPTIMIZATION

Inverse Reinforcement Learning techniques make use of expert data in order to reconstruct a reward function. One way to gather expert data is to learn an environment by means of standard Reinforcement Learning. Once an expert has learned to play an environment effectively, it

Parameter	Value
Parallel Runs	8
Value Function Coefficient	0.5
Entropy Coefficient	0.01
Horizon	128
# of epochs when optimising the surrogate	4
$ of training minibatches per update	4
Discount Factor γ	0.99
GAE Parameter λ	0.95
Learning Rate η	alpha * 2.5e-4
Cliprange	0.1
# of Environments	8
# of Workers	8
# of Timesteps	10e6

Table 7.1: Shared settings for all experts.

is possible to collect high quality expert trajectories. Proximal Policy Optimisation (PPO) is the state-of-the-art policy gradient algorithm to effectively learn high-dimensional pixel environments. In Chapter 5, PPO is explained in detail. The chosen parameters for this architecture are displayed in Table 7.1 and reflect the default parameters for the Atari environment in the OpenAI baselines package [10]. In the experiments, PPO utilises Convolutional Neural Networks (CNNs) as function approximators for the Actor and the Critic. Chapter 2 describes CNNs in greater detail. Both the Actor and the Critic make use of the vanilla CNN architecture as described in [33]. The PPO architecture is trained using the stochastic optimisation algorithm Adam [22] and is described in Section 2.4.2.

7.2.1 *Collecting expert data*

After reaching a proper policy through PPO, it is possible to sample demonstrations from it. The Inverse Reinforcement Learning algorithms as discussed in this thesis assume that the demonstrations are sampled from an optimal policy. Therefore it is important that the samples are taken from a policy which result in high reward scores. A perfect game of Pong yields a maximum reward of +21. The current human highscore for Freeway is +38 [1]. The authors of PPO [46] ob-

1 As consulted from http://highscore.com/games/Atari2600/Freeway/10875 (16-07-2020)

Parameter	Value
Weight initialization	He
Optimization	Adam
Learning rate η	1e-4
Encoding regularization term (if used)	0.01
# of Environments	8
Batchsize per Environment	32
Size Replay Buffer	100 batches of size 8 x 32
Maximum amount of Epochs	2500

Table 7.2: Shared settings for all autoencoders

tained a reward of +32.5 in Freeway for PPO which is close to the level of human experts. In the Catcher environment there is no maximum reward value. The game simply ends when the ball passes the paddle. Training continues until the reward exceeds a score of 100 meaning that it successfully catches 100 balls before dropping one.

7.3 AUTOENCODERS

One of the biggest challenges in Inverse Reinforcement Learning (IRL) algorithms is to deal with high-dimension state spaces. Autoencoders are a class of dimensionality reduction techniques in which the goal is to compress the original data as much as possible while maintaining all relevant information. In the experiments, three different autoencoders will be tested: vanilla autoencoder, variational autoencoder and vector-quantised variation autoencoder. All of these autoencoders are discussed in more detail in Chapter 3.

7.3.1 *General Implementation Autoencoders*

The vanilla, variational and vector-quantised variational autoencoders share a set of settings and hyperparameters across the autoencoders. Table 7.2 provides these settings and parameters.

7.3.2 *Vanilla Autoencoder*

The vanilla autoencoder makes use of Batch Normalisation (BN), Convolutional (Conv) Layers and fully connected (FC) layers to learn a compressed representation of the data. To obtain the reconstructed image, the decoder makes use of Deconvolutional (Deconv) Layers.

Layer	Type	Kernel	Depth	Stride
1	Conv + BN + LeakyRelu	8	32	4
2	Conv + BN + LeakyRelu	4	64	2
3	Conv + BN + LeakyRelu	3	64	1
4	FC + sigmoid	-	80	-

Table 7.3: Encoder architecture for the Vanilla Autoencoder and VAE. The only difference for VAE is that the depth for the last layer is split in two parts since the mean and log-variance are encoded separately.

ENCODER The input consists of the 80x80 preprocessed frames. It follows the standard Nature CNN architecture as described in Table 7.3. The end result is a continuous feature vector of size 80.

DECODER The decoder is fed the output of the encoder and attempts to reconstruct the input image. The architecture for the decoder is listed in Table 7.4.

OPTIMISATION Adam is used as the optimisation algorithm. A regularisation term of 0.01 is chosen to prefer values close to 0 in the latent vector. Furthermore, the encoding is optimised by minimising the mean squared error between the original and the reconstructed image.

7.3.3 *Variational Autoencoder*

Another way to encode data is to utilise Variational Autoencoders (VAEs). Unlike vanilla autoencoders, VAEs are generative models and make assumptions regarding the distribution of the latent variables defined by a mean and standard deviation. Section 3.2.2 describes VAEs in more detail. In the experiments, the VAEs are parameterised by convolutional neural networks.

ENCODER The architecture for the VAE encoder identical to the Vanilla Autoencoder is listed in Table 7.3.

DECODER The architecture for the decoder is listed in Table 7.4. It is identical to the decoder of the Vanilla Autoencoder except that the depth is split in two parts since the mean and log-variance are encoded separately.

OPTIMISATION Similar to the vanilla autoencoder, VAE computes the difference between the reconstructed and the original image using the mean squared error. Additionally a latent loss is computed which

Layer	Type	Kernel	Depth	Stride
1	Deconv + Relu	3	64	1
2	Deconv + Relu	4	64	2
3	Deconv + Relu	8	32	4

Table 7.4: Decoder architecture for the Vanilla Autoencoder and VAE.

forces the output distribution of the encoder to be close a standard Gaussian. Adam is again the optimisation algorithm.

7.3.4 *Vector Quantised Variational Autoencoders*

Recent work combines Vector Quantisation [18] with Variational Autoencoders [23]. In Section 3.2.3 the overall architecture of Vector Quantised Variational Autoencoders (VQ-VAE) is discussed.

AUTOENCODER PARAMETERS Table 7.5 lists all the relevant parameters to instantiate the encoder, vector quantisation layer and the decoder. These parameters are discussed in Section 3.2.3. Both the encoder and the decoder have Residual Blocks as part of their architecture. The architecture for these blocks is shown in Table 7.7.

Parameter	Value
Commitment Cost	0.25
Embedding Dimensionality	16
# of Embeddings	4
Size of image	80x80

Table 7.5: Decoder architecture for the VAE.

ENCODER The encoder constructs a low-dimensional image map representing a compressed version of the original data. In the experiments, the parameters are carefully chosen on the basis of previous research. Similar to the previous autoencoders, the encoder in VQ-VAE operates on 80x80 images. The encoder architecture is displayed in Table 7.6. The resulting output is a 20x20x16 map which needs to be processed by the vector quantisation layer.

Layer	Type	Kernel	Depth	Stride
1	Conv + ReLu	4	8	2
2	Conv + ReLu	4	16	2
3	Conv + ReLu	3	16	1
4	Residual Block (see Table 7.7)			
5	Residual Block (see Table 7.7)			

Table 7.6: Encoder architecture of VQ-VAE.

Layer	Type	Kernel	Depth	Stride
1	Input + ReLu	-	-	-
2	Conv + ReLu	3	32	2
3	Conv	1	128	2
4	Layer 1 + Layer 3	-	-	-

Table 7.7: Residual Block used in the encoder and decoder of VQ-VAE.

VECTOR QUANTISATION A vector quantisation step is performed on the latent representation obtained from the encoder. Recall that the number of embeddings is 4 which can be represented with 2 bits. This results in a 20x20 map holding values $n \in \{1,2,3,4\}$. All relevant parameters for the Vector Quantisation (VQ) step are listed in Table 7.5. Consult 3.2.3 or [54] for a more detailed explanation on the Vector Quantisation step.

REPRESENTATION FOR AIRL The map has to be transformed to a representation suitable for the AIRL algorithm. This is done by first flattening the vector after which a binarisation step is performed to yield a binary vector of size 800.

DECODER After the Vector Quantisation step, the quantised latent embedding map is consumed by the decoder. Table 7.8 lists the layers necessary to create a reconstruction of the original image.

OPTIMISATION The optimisation process is explained in detail in Section 3.2.3. The commitment cost is set to 0.25. Again Adam is used as the optimisation strategy.

Layer	Type	Kernel	Depth	Stride
1	Conv	3	16	1
2	Residual Block (see Table 7.7)			
3	Residual Block + ReLu (see Table 7.7)			
2	Deconv + ReLu	4	8	2
3	Conv + (Sigmoid - 0.5)	4	16	2

Table 7.8: Decoder architecture for the VQ-VAE.

7.4 ADVERSARIAL INVERSE REINFORCEMENT LEARNING

Inverse Reinforcement Learning deals with the reconstruction of a reward function by learning from expert examples. The cornerstone of this research is the Adversarial Inverse Reinforcement Learning (AIRL) algorithm that is extensively described in Chapter 6. AIRL allows for reconstructing reward functions on the basis of adversarial examples and expert data. However research has shown that AIRL scales poorly to high-dimension problems in which the dynamics are unknown. The pseudocode for AIRL is described in Algorithm 1 and is reprinted from [15]. Although several solutions are proposed to alleviate the overfitting problem, there is still a lot of improvement possible.

Algorithm 1 Adversarial Inverse Reinforcement Learning

1: Obtain expert trajectories τ_i^E
2: Initialise policy π and discriminator $D_{\theta,\phi}$.
3: **for** step t in {1, ..., N} **do**
4: Collect trajectories $\tau_i = (s_0, a_0, \ldots, s_T, a_T)$ by executing π.
5: Train $D_{\theta,\phi}$ via binary logistic regression to classify expert data τ_i^E from samples τ_i.
6: Update reward $r_{\theta,\phi}(s,a,s') \leftarrow \log D_{\theta,\phi}(s,a,s') - \log\left(1 - D_{\theta,\phi}(s,a,s')\right)$.
7: Update π w.r.t. $r_{\theta,\phi}$ using any policy optimisation method.
8: **end for**

7.4.1 *Architecture AIRL*

In this research several configurations for AIRL are considered to solve the Catcher, Pong and Freeway environments as described in 7.1. This research aims to investigate the performance of AIRL on high-dimensional pixel data. In order to alleviate the problem of discriminator overfitting, autoencoders are used to form a low-dimensional latent

Parameter	Value
# of Expert Demonstrations	20
# of Training Epochs	200
Batch Size	256
Entropy Weight	0.01
Autoencoders	None, Vanilla Autoencoder, VAE, VQ-VAE
# of Discriminator Iterations	100
# of PPO steps in batch	4
Discount Factor γ	0.99
Reward Architecture	MLP (CNN if no encoding is used)
Reward Shaping Architecture	MLP (CNN if no encoding is used)
RL Algorithm	PPO with CNN for Actor and Critic

Table 7.9: Training parameters for the AIRL architecture. Different autoencoders are tested on their performance. Several batch sizes and number of expert trajectories are considered to reconstruct a proper reward function.

on which the discriminator is trained. Finding a reasonable number of expert trajectories is important to reduce the chance of overfitting. The same holds for the batch size. Since the iterations are computationally very expensive, the batch size and number of expert trajectories are based on previous research. [53]. The implementation is based on [53].

In all experiment the batch size is set to 256 and the number of expert trajectories is 20. The amount of training epochs is set to 100, surpassing the 40 training epochs used in previous research on Atari environments [53]. The other parameters have less influence on the optimisation process and are chosen based on existing literature [15, 53]. All parameters relevant to the AIRL training process are listed in Table 7.9.

In order to evaluate the performance of the autoencoders in the AIRL system, it is important to have a baseline. This baseline is the AIRL architecture and does not use any autoencoder. The input of the system are the raw 80x80 images and the reward architecture makes use of standard Nature CNNs. The results are collected and depicted in Chapter 8.

8 RESULTS

This chapter is structured by evaluating the results of each step in the Inverse Reinforcement Learning (IRL) pipeline as described in Chapter 7. IRL requires expert demonstrations in order to learn the underlying rewards. The *first* step is to train a policy to optimality from which expert demonstrations can be sampled from. To prevent the discriminator from overfitting in the IRL process, it is important to work with low-dimensional representations of the game states. The *second* step is therefore to train autoencoders to reduce the dimensionality of the original data. The *last* step is to evaluate the IRL algorithm as a whole by feeding it the low-dimensional representation of the sampled expert demonstrations. The next sections cover these three steps and provide the reader with the outcomes of the experiments.

8.1 SAMPLING EXPERT DEMONSTRATIONS

Gathering high quality expert demonstrations is important to properly train IRL algorithms as they assume that demonstrations are approximately optimal. The behaviour should be optimal under the reward function that IRL attempts to recover. One way to obtain expert demonstrations is to first learn a policy using standard Reinforcement Learning from which the demonstrations can be sampled. As discussed in 7, PPO is used throughout the experiments to obtain a policy. PPO is trained on three different environments: Catcher, Pong and Freeway.

The reward curves for Catcher, Pong and Freeway are displayed in Figure 8.1 and indicate that the agents are able to properly learn the environments yielding high rewards for Catcher, a maximum reward of 21.0 for Pong and a high reward of 32.6 for Freeway. Each update step corresponds to 1024 frames by training on 8 different environments of the same game with a batch size of 128 in parallel.

After training, the policy epoch that yielded the highest rewards is used to sample expert demonstrations from. This ensures that the expert demonstrations are of sufficient quality. The next step is to learn a low-dimensional embedding of the states encountered in Catcher, Pong and Freeway.

8.2 LEARNING A LOW-DIMENSIONAL EMBEDDING

To prevent the overfitting problem of the discriminator in AIRL, autoencoders are implemented to reduce the dimensionality of the

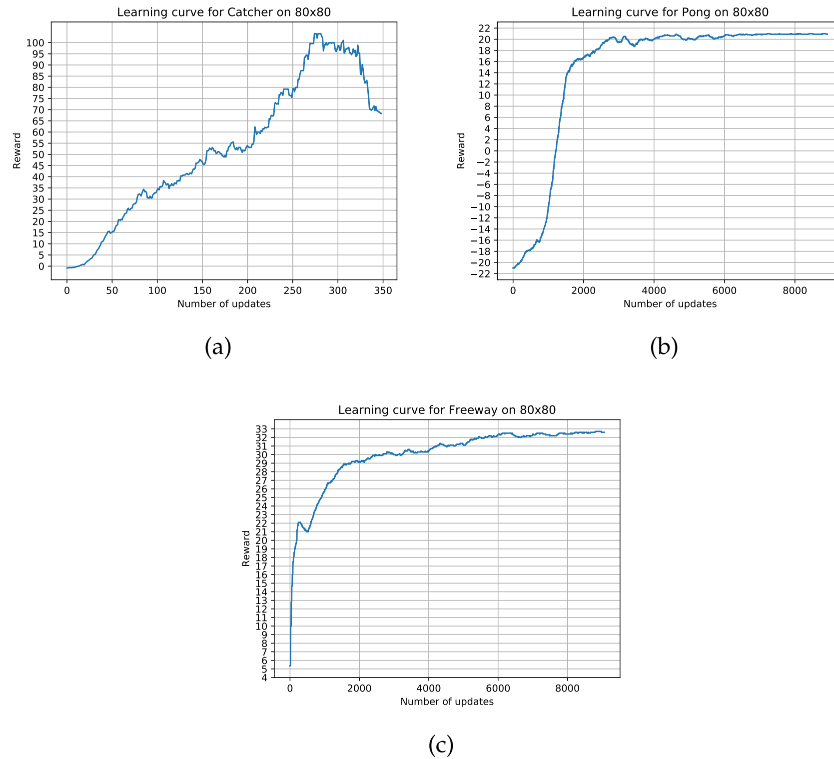


Figure 8.1: Reward curves for the two Atari environments. The curves represent the rewards obtained by a single agent in one simulation.

data. This section illustrates the results from the three different autoencoders: vanilla autoencoder, variational autoencoder and vector-quantised variation autoencoder. All autoencoders are trained separately for 2500 training steps for all environments. To ensure the best results, only the model of the autoencoders which yields the lowest loss is stored.

8.2.1 *Vanilla Autoencoder*

The simplest way to encode the data is through a vanilla autoencoder. This autoencoder enforces no constraints on the latent embedding and only requires that the difference between the original image and its reconstruction is minimised. Figure 8.2, 8.3 and 8.4 illustrate the quality of the vanilla autoencoder on the different environments. The first column displays the original images. The second column displays the reconstructed images. The third column displays the difference between the original and the reconstructed frames. A uniform black image in the last column would indicate that the original image and its reconstruction are identical.

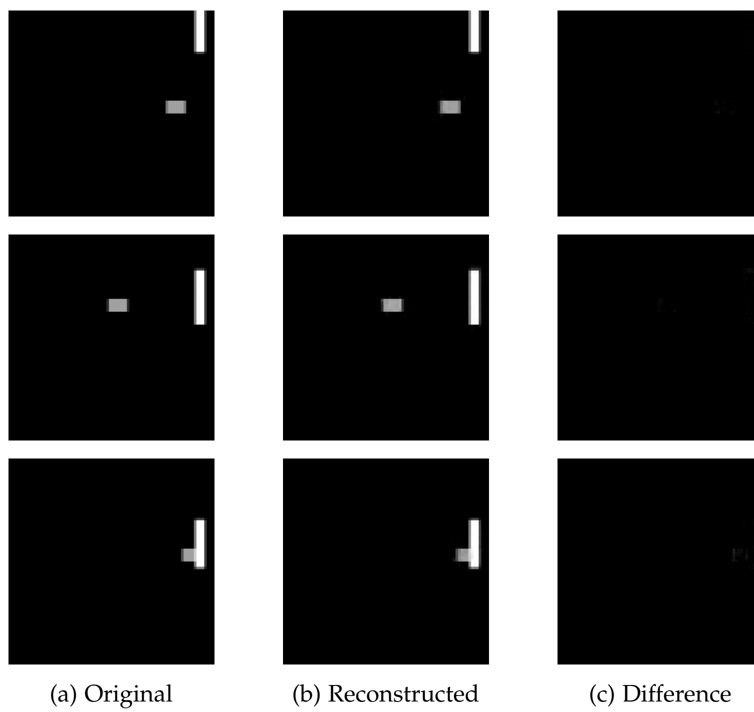


Figure 8.2: Reconstructions with a standard autoencoder on Catcher. The difference images indicate that all the details are properly captured in the reconstruction.

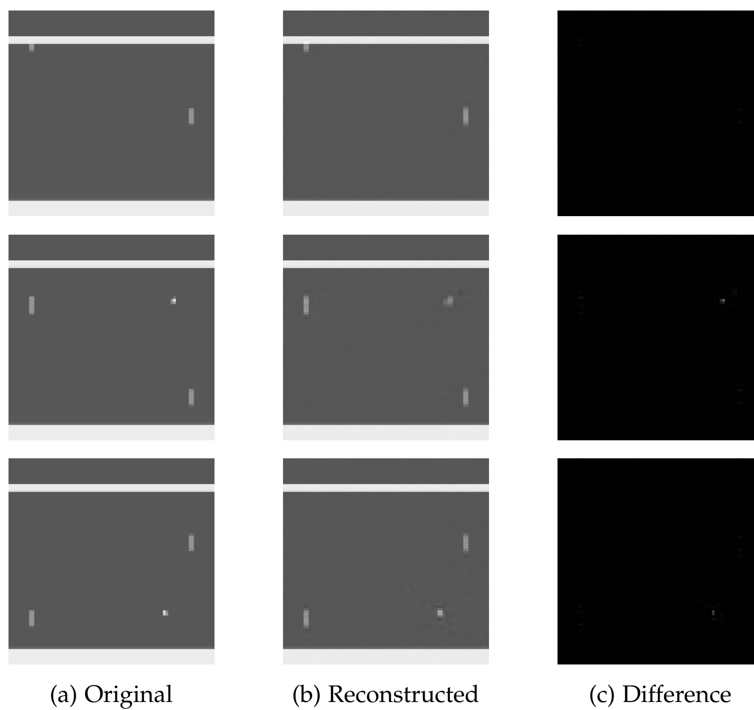
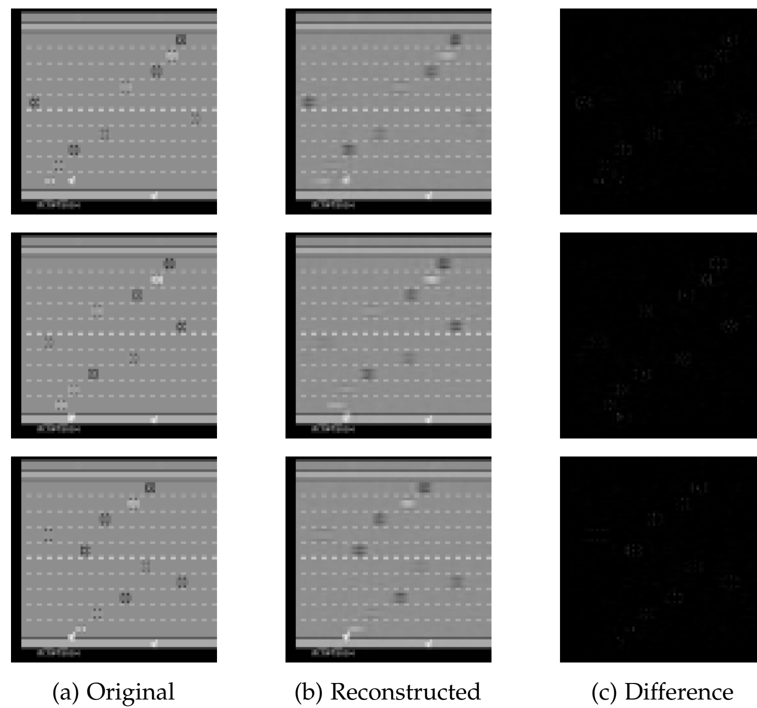


Figure 8.3: Reconstructions with a standard autoencoder on Pong. The difference images indicate that the overall structure of the frames is captured. However, fine details are missing in the reconstructions.



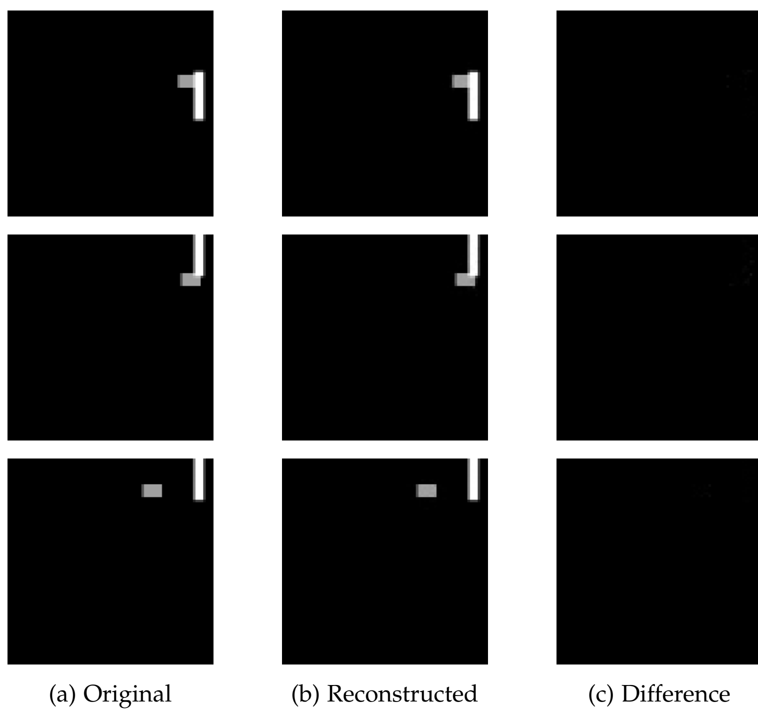
(a) Original (b) Reconstructed (c) Difference

Figure 8.4: Reconstructions with a standard autoencoder on Freeway. The difference images indicate that the overall structure of the frames is captured. However, fine details are missing in the reconstructions.

Although the decoded image reflects the original encoded images rather well, there are some important issues. First, in some cases the position of the objects is not properly represented. Second, sometimes the objects disappeared in the decoded image. This could arguably lead to a decrease in performance further in the pipeline. Third, the contrast between the objects and the background is often reduced.

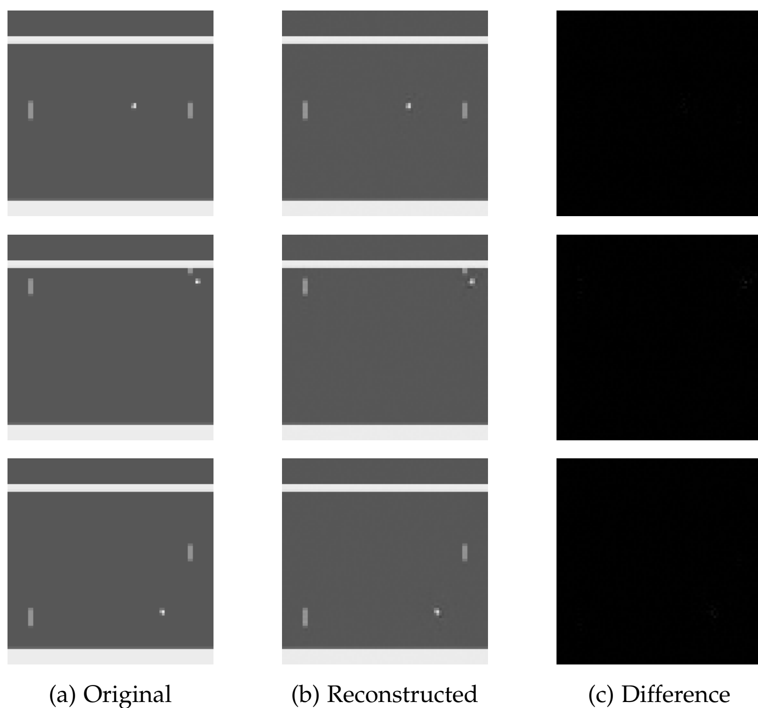
8.2.2 *Variational Autoencoder*

In order to obtain more control over our latent distribution, Variational Autoencoders (VAEs) could be considered. Figure 8.5, 8.6 and 8.7 illustrate the quality of the VAE. Similar to results from the vanilla autoencoder, the original, the reconstruction and the difference images are represented in the first, second and third column accordingly.



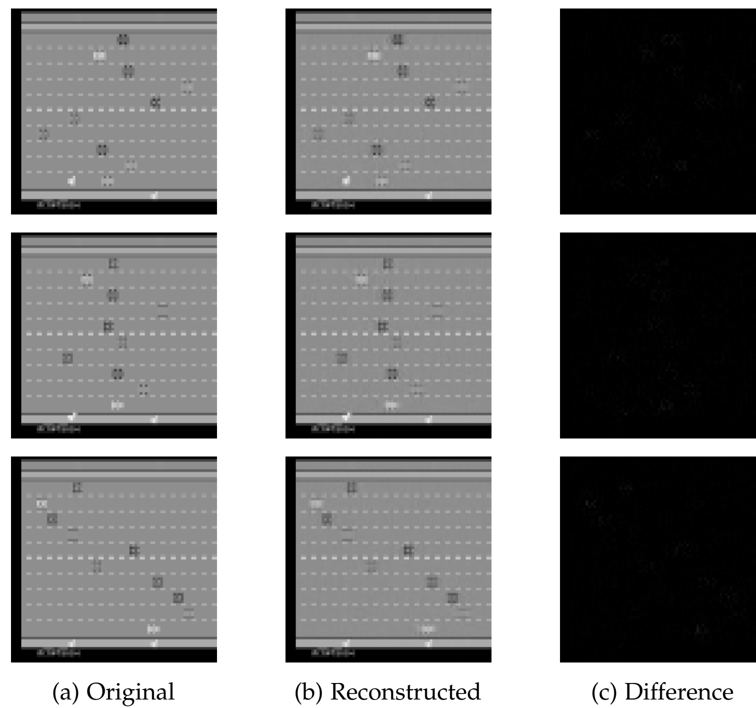
(a) Original (b) Reconstructed (c) Difference

Figure 8.5: Reconstructions with a variational autoencoder on Catcher. The difference images indicate that all the details are properly captured in the reconstruction.



(a) Original (b) Reconstructed (c) Difference

Figure 8.6: Reconstructions with a variational autoencoder on Pong. The difference images indicate that the overall structure of the frames is captured. Some small details are missing in the reconstructions.



(a) Original (b) Reconstructed (c) Difference

Figure 8.7: Reconstructions with a variational autoencoder on Freeway. The difference images indicate that the overall structure of the frames is captured. Some small details are missing in the reconstructions.

Looking closely at the decoded images, similar issues can be seen as with the vanilla autoencoder. However, the reconstruction quality is improved drastically.

8.2.3 *Vector-Quantised Variational Autoencoder*

Vector-Quantised Variational Autoencoders (VQ-VAEs) learn a discrete latent embedding map of the data. In the experiments, the 80x80 input are encoded as a 20x20 embedding map. Figure 8.8, 8.9 and 8.10 illustrate the quality of the VQ-VAEs. The first three columns again represent the original, reconstruction and the difference images. Additionally a fourth column is added to visualise the 20x20 embedding of the respective input frame. Each pixel in the embedding map indicates one of four categories and can be encoded using 2 bits. Therefore, the embedding map uses 800 bits to encode the original image.

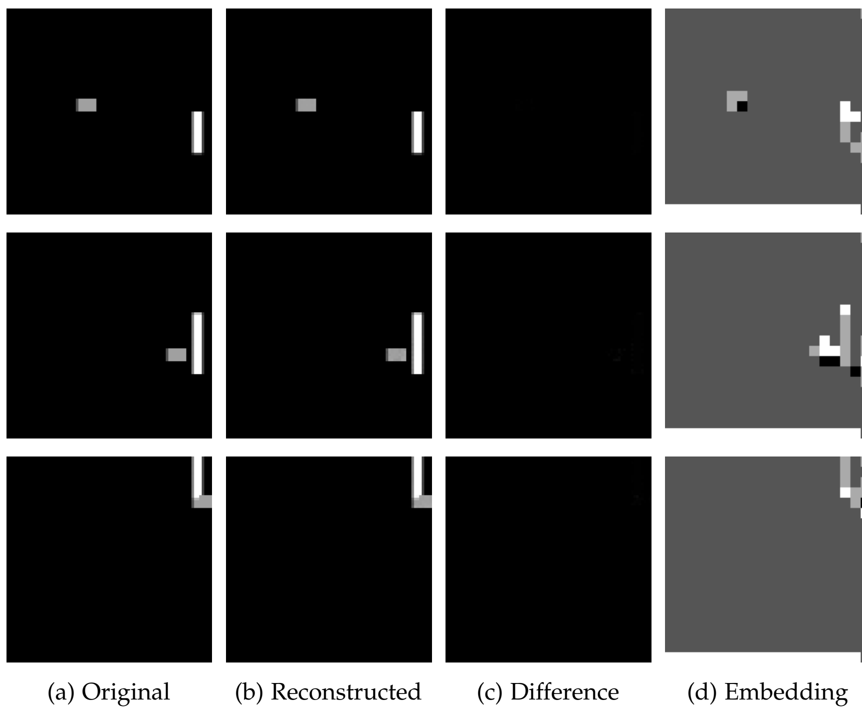
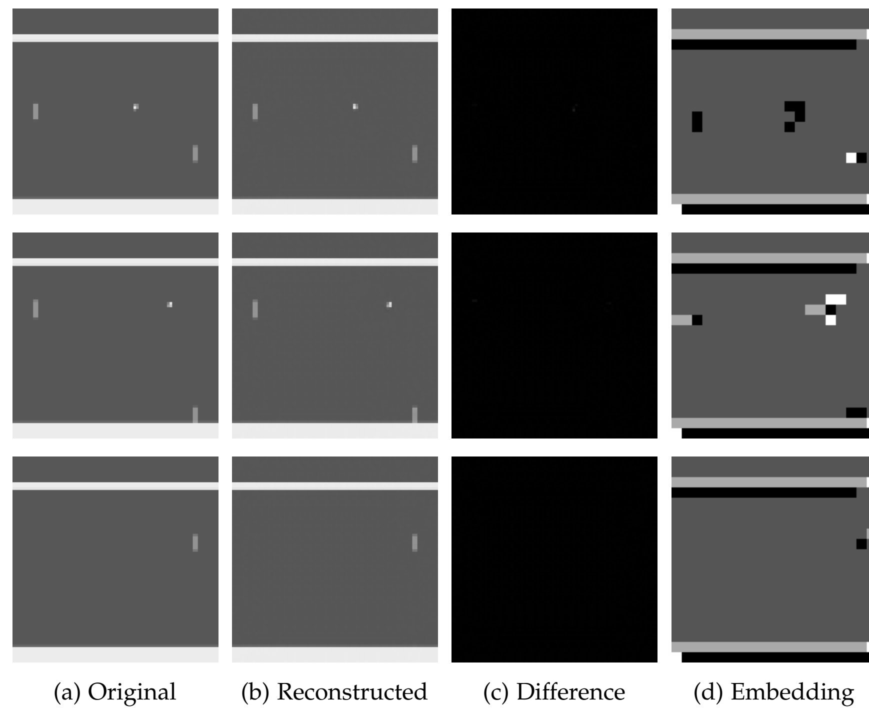


Figure 8.8: Reconstructions with VQ-VAE on Catcher. The difference images indicate that the encoding properly captures the environment for different frames. The embedding of the original image is listed in the last column.



(a) Original (b) Reconstructed (c) Difference (d) Embedding

Figure 8.9: Reconstructions with VQ-VAE on Pong. The difference images indicate that the encoding properly captures the environment for different frames. The embedding of the original image is listed in the last column.

As seen in Figure 8.9, VQ-VAE is able to properly capture the information of the Pong environment. The reconstructions are almost identical to the original frame meaning that the original input can be represented as a 20x20 embedding map in which each pixel is one of 4 classes. The embedding map shows some spatial similarities to the original images. It is evident that the multiple pixels in the embedding map are required to capture the variability of complex objects. An example is the large number of pixels used to capture the ball in the first two embedding images. Since an individual pixel can only store 4 values, it requires the neighbouring pixels to capture the variability of the ball in more detail.

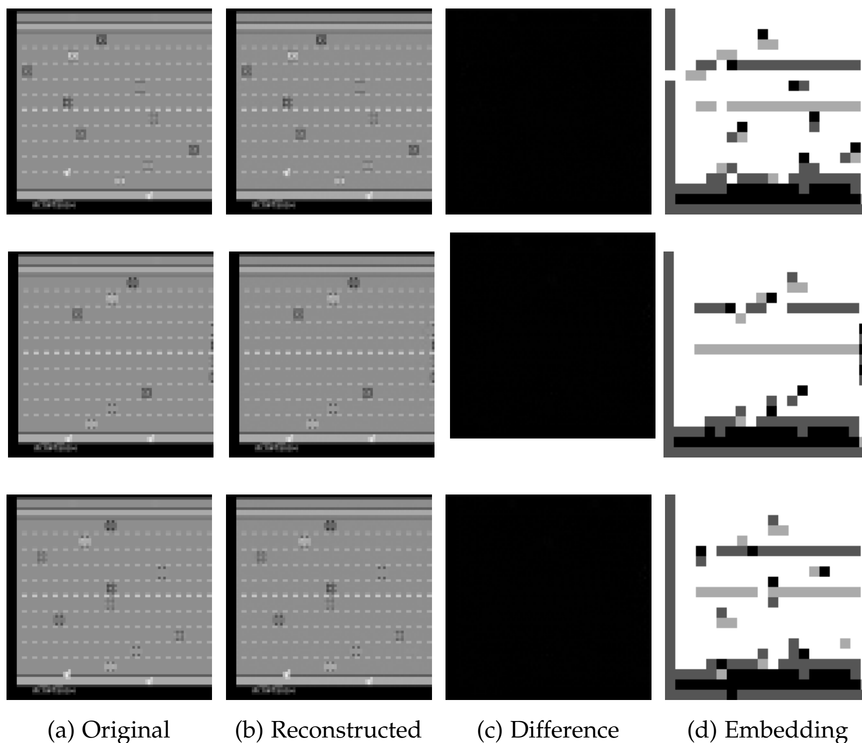


Figure 8.10: Reconstructions with VQ-VAE on Freeway. The difference images indicate that the encoding properly captures the environment for different frames. The images in the last column show the embedding of the original image.

Pong is relatively simple to encode due to the limited amount of objects in the input images. Freeway, which is visually much more complex, could be more challenging to encode. The results for Freeway are depicted in Figure 8.10. It appears that 800 bits is enough to encode the Freeway images properly. Spatially, there are again similarities between the original images and their respective encoding.

The vanilla and variational autoencoder had issues with capturing and representing the objects of the environments properly. Evidently, VQ-VAE is not susceptible to these issues and allows for better reconstructions. The next section illustrates the results when combining the different autoencoders with AIRL.

8.3 AIRL

This section describes the overall results from the AIRL experiments on all environments. The demonstrated reconstructed reward curves below represent the rewards obtained from the ground-truth reward function when evaluated on the policy that is directly obtained from the reconstructed reward function through AIRL. In order to establish the effectiveness of the low-dimensional embedding of the autoencoders, a baseline is required for each of the environments. These baselines do not make use of any autoencoders and act directly on

the 80x80 input images. Afterwards, AIRL is trained with the autoencoders and the results are compared with the baselines. The goal is to see whether the autoencoders reduce the overfitting problem of the discriminator in the AIRL architecture. Reducing the overfitting problem should yield better reward curves. In the following subsections, the reward curves are displayed and discussed per environment. Chapter 9 investigates in more detail the intricacies of the learnt behaviour.

8.3.1 *AIRL performance on Catcher*

The easiest environment to learn is the Catcher environment. Although there are only two objects present in the environment, the raw observation is still 80x80 pixels. Figure 8.11 displays the reward curves obtained in the Catcher environment.

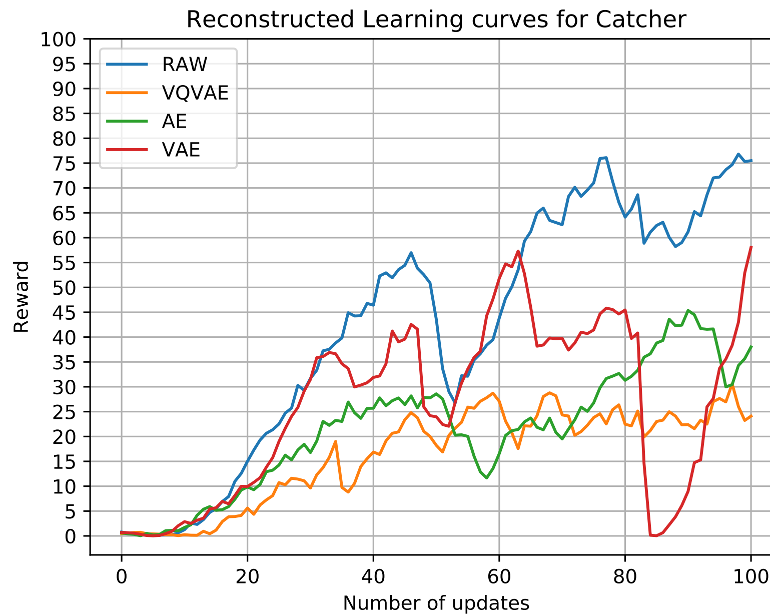


Figure 8.11: The reconstructed reward curves of the Catcher environment.

The reward curves indicate that all the AIRL agents are able to learn the objective of catching the ball with the pedal. It should be noted that the highest reward values were obtained when raw pixel data was considered. This is surprising since it indicates that the discriminator might not be overfitting to an extreme extent on the raw input images. An explanation could be that the input space is relatively simple, hence not requiring an autoencoder to alleviate the problem of overfitting. The following sections demonstrate the results obtained from the more challenging Pong and Freeway environments.

8.3.2 *AIRL on Pong*

The next environment is Pong. Although Pong is similar to Catcher, the state space is considerably more diverse. Pong offers an opponent and two pedals instead of one. Additionally the representation of the objects is richer and provides more variation. The results of AIRL for Pong are displayed in Figure 8.12.

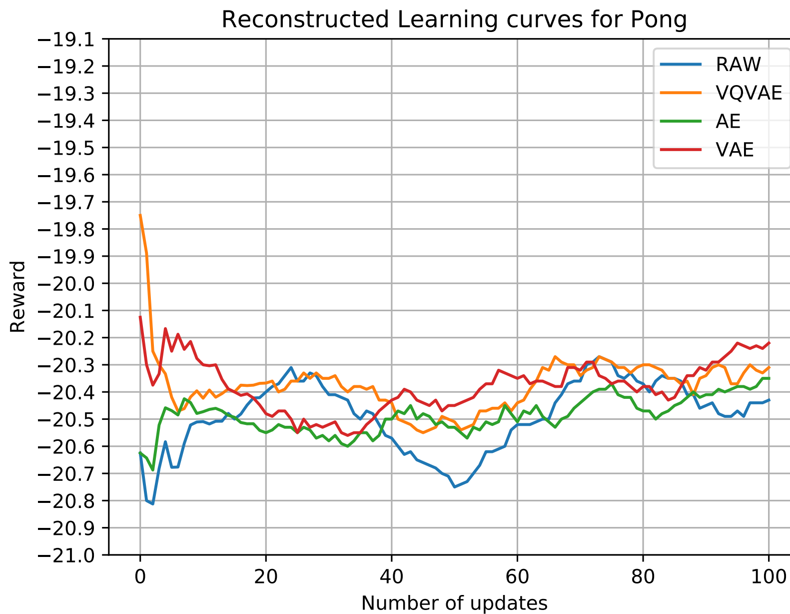


Figure 8.12: The reconstructed reward curves of the Pong environment.

Unfortunately, the reward curves indicate that the environment is hard to learn. The rewards stay low, even with the autoencoders, and there is no clear evidence that specific behaviour is learnt directly from the figure.

8.3.3 *AIRL on Freeway*

The last environment is Freeway. Compared to Pong, Freeway is visually a lot more demanding. However, the controls are easier since the only task is to cross the road. Every time the player crosses the road, a point is scored. It should be noted that scoring points is easier in Freeway compared to Pong. This makes Freeway an interesting environment to evaluate AIRL on. Figure 8.13 demonstrates the reconstructed reward curves of Freeway.

I

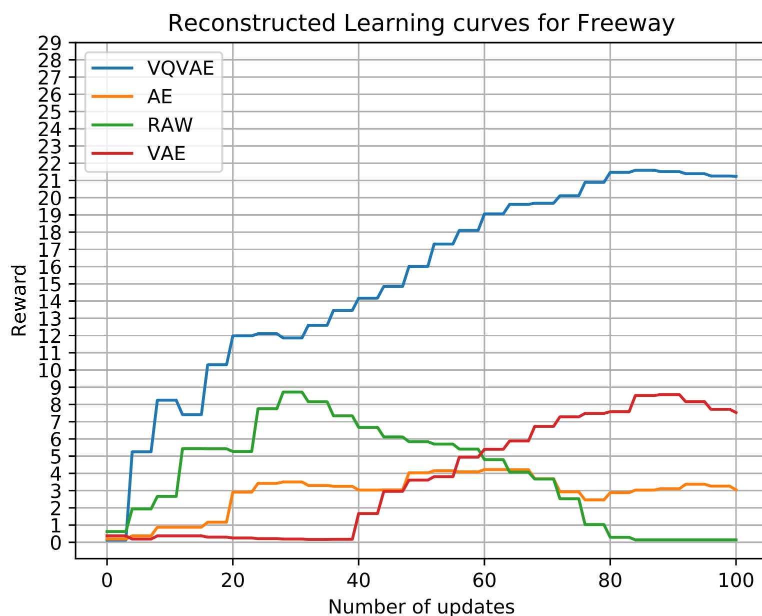


Figure 8.13: The reconstructed reward curves of the Freeway environment.

The effectiveness of the Vector Quantised Variational Autoencoder is clearly visible in the reconstructed reward curves. After 100 epochs the rewards stabilised at around 21. The other agents failed to get close to this result. This indicates that the use of autoencoders, in specific discrete latent autoencoders, can help to improve the performance of AIRL algorithms. The next and last chapter wraps up this paper by zooming in on the specific behaviours that follow from the policy derived from the reconstructed reward functions.

9 DISCUSSION AND CONCLUDING REMARKS

In Chapter 8 the empirical results of the experiments are presented. Firstly, this chapter is meant as an extension on the results to provide more insight into the specific behaviours that follow from the Adversarial Inverse Reinforcement Learning (AIRL) algorithm. Ultimately, the behaviours are the most important aspect of the agents and should be evaluated. Secondly, the research questions as stated in Section 1.4 are answered. Finally, some concluding remarks are written to wrap up this thesis.

9.1 AIRL BEHAVIOURS

In order to assess the quality of the reconstructed reward functions, it is important to inspect the behaviour of the agent in each environment carefully.

9.1.1 *Pygame: Catcher*

The reconstructed reward curves in Section 8.3 indicate that Catcher could be solved with AIRL both with and without the autoencoders.

RAW INPUT The highest rewards were obtained without the use of autoencoders. The agent performs some unnecessary movements before catching a ball by first moving the bat all the way to the right (or top) of the environment. Even with this behaviour, the agent is able to robustly catch the ball in the majority of the cases.

VANILLA AUTOENCODER The agent displays behaviour which is rather chaotic behaviour. A lot of unnecessary movements are performed in between the process of catching balls. However, even with the unnecessary movements the agent is still able to successfully catch the balls in the majority of the cases.

VARIATIONAL AUTOENCODER The agent performs the task and only moves towards the ball when this is needed. In contrast to the Vanilla Autoencoder, the agent displays fewer unnecessary movements and it is successful at catching the ball in the majority of the cases.

VECTOR QUANTISED VARIATIONAL AUTOENCODER Some unnecessary movements are visible, although less evident compared to

the results with the vanilla autoencoder. Again, the agent is able to successfully catch the ball.

9.1.2 *Atari: Pong*

The results in Section 8.3 indicate that Pong is hard to solve with AIRL. Even in combination with autoencoders, no usable reward function is reconstructed. In all of the experiments for Pong the behaviour of the agent is similar by displaying random movement of the bat. It should be noted that, compared to Catcher, it is much harder to score points. The opponent is able to return the ball in the majority of the cases even if the ball is hit by the agent. Rewards are therefore scarce and this influences the learning process of the AIRL algorithm.

9.1.3 *Atari: Freeway*

In contrast to Pong, the results for Freeway look promising as displayed in Section 8.3.

RAW INPUT When considering the raw input images, the results do not exceed a reward score of 9. Looking closely at the behaviour of the agent reveals that it crosses the road but is unable to do this in an efficient manner. In some situations the agent suddenly stops moving forward while this is not necessary. The agent only moves forward and never moves back.

VANILLA AUTOENCODER Compared to the raw input version, this agent performs poorly. It takes the agent longer to cross the road. Also the agent moves back in situations when this is not necessary. This results in lower rewards compared to the raw input version.

VARIATIONAL AUTOENCODER The agent performs almost identically to the raw input version. Again, the agent only moves forward and does not move backwards. The amount of time it takes to cross the road is on average also similar compared to the raw input version.

VECTOR QUANTISED VARIATIONAL AUTOENCODER The best rewards were obtained using a Vector Quantised Variational Autoencoder. A reward close to 22 was obtained. When looking at the specific behaviour of the agent, it becomes clear that the agent tries to move forward as fast as possible. Unfortunately the agent did not learn to avoid the cars. The agent learned to optimise its rewards by crossing the road as quickly as possible.

9.2 RESEARCH QUESTIONS

In Section 1.4, the research questions are stated. This section aims to provide an answer to all research questions.

CAN VANILLA AND VARIATIONAL AUTOENCODERS HELP IMPROVE THE RECONSTRUCTION OF REWARD FUNCTIONS ON HIGH-DIMENSIONAL DATA? Although vanilla and variational autoencoders could alleviate the problem of overfitting in the discriminator, they fail to provide an accurate enough embedding. As seen in Section 8.2.1 and 8.2.2, the decoded images do not accurately represent and capture the details of the environments. However, there is enough information present in the encoding to allow the agent to make informed decisions.

In Catcher, the best results were obtained without the use of autoencoders. This could be explained by the fact that the environment is relatively sparse in state. However, in a complex environment such as Freeway, the use of autoencoders could make the training process more robust and could increase the reward. This can also be seen in Figure 8.13 which demonstrates that the rewards increase steadily without a large variance.

HOW DO THESE RESULTS COMPARE TO INSTANCES WHERE THE RAW INPUT OF THE ORIGINAL DATA IS CONSIDERED? For more complex environments, the use of autoencoders helped to improve the quality of the reward functions. In Catcher, both experiments with and without autoencoders were able to learn a stable policy. For more complex environments, the use of autoencoders is recommended as this decreases the likelihood of the discriminator overfitting on the data.

VANILLA AND VARIATIONAL AUTOENCODERS LEARN A LOW-DIMENSIONAL CONTINUOUS REPRESENTATION. WOULD AUTOENCODERS THAT LEARN A BINARY OR DISCRETE REPRESENTATION INSTEAD ALLOW FOR MORE ROBUST REWARD FUNCTION RECONSTRUCTIONS? In order to obtain a binary encoding, a vector quantised variational autoencoder (VQVAE) was used. In complex environments such as Freeway, the usefulness of a binary encoding became evident. The best results for Freeway were obtained with VQVAE indicating that a discrete representation could allow for more robust reward function reconstructions.

9.3 CONCLUDING REMARKS

This section concludes the thesis by summarising the findings of the study, describes the limitations and highlights the importance of future research.

FINDINGS OF THE STUDY A set of AIRL experiments were performed on three different environments. In order to reduce the overfitting problem of the discriminator in AIRL, different encodings are proposed from several autoencoders: vanilla autoencoders, variational autoencoders and vector quantised variational autoencoders.

In simple environments, the use of autoencoders could thwart the process to learn reconstructed reward functions. This became apparent when training an agent to play Catcher. However, in more visually complex environments such as Freeway in which the rewards are relatively easy to obtain, the use of discrete autoencoders could help to learn more robust reward functions. It is difficult to successfully train an agent using AIRL in environments such as Pong where it is relatively hard to score points.

LIMITATIONS Inverse Reinforcement Learning and some Imitation Learning algorithms are computationally very expensive. Even with the introduction of algorithm such as Generative Adversarial Imitation Learning and AIRL which do not require to evaluate the entire policy after each reward learning step, it is still a computationally expensive process.

In this research, the parameters such as batch size and the number of demonstrations are based on previous research. All environments are different in their representation and testing for different batch sizes and number of demonstrations could yield better reward reconstructions.

Additionally in all AIRL experiments, the agent is trained for 100 epochs. Although this exceeds previous research by a factor of 2.5, more epochs could yield an improvement in the quality of the reconstructed reward curves.

IMPACT AND FUTURE RESEARCH Understanding observed behaviour is a hard but important problem. Current Inverse Reinforcement Learning (IRL) algorithms offer ways to learn the intuition behind behaviour, but decline in performance as the data dimensionality increases. Although in this research some measures are taken to alleviate the problem of the current state-of-the-art algorithm Adversarial Inverse Reinforcement Learning (AIRL), future work should be done to allow for even more robust ways of recovering reward functions.

The impact IRL could have in the field of Machine Learning is massive. In cases where reward functions are hard or even impossible to manually specify, IRL could provide the abstractions required to learn reasonable policies. As discussed in 6.2, reward functions are more transferable than policies. Finally, it could also be preferable to capture the expert behaviour in the system in contrast to forming a policy directly from a reward function.

BIBLIOGRAPHY

[1] Pieter Abbeel and Andrew Y Ng. "Apprenticeship learning via inverse reinforcement learning." In: *Proceedings of the twenty-first international conference on Machine learning*. 2004, p. 1.

[2] Adebiyi A Ariyo, Adewumi O Adewumi, and Charles K Ayo. "Stock price prediction using the ARIMA model." In: *2014 UKSim-AMSS 16th International Conference on Computer Modelling and Simulation*. IEEE. 2014, pp. 106–112.

[3] Martin Arjovsky, Soumith Chintala, and Léon Bottou. "Wasserstein gan." In: *arXiv preprint arXiv:1701.07875* (2017).

[4] Samuel A Barnett. "Convergence problems with generative adversarial networks (gans)." In: *arXiv preprint arXiv:1806.11382* (2018).

[5] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. "The arcade learning environment: An evaluation platform for general agents." In: *Journal of Artificial Intelligence Research* 47 (2013), pp. 253–279.

[6] Richard Bellman. "A Markovian decision process." In: *Journal of mathematics and mechanics* (1957), pp. 679–684.

[7] Xin-Qiang Cai, Yao-Xiang Ding, Yuan Jiang, and Zhi-Hua Zhou. "Expert-Level Atari Imitation Learning from Demonstrations Only." In: *arXiv preprint arXiv:1909.03773* (2019).

[8] Augustin Cauchy. "Méthode générale pour la résolution des systemes d'équations simultanées." In: *Comp. Rend. Sci. Paris* 25.1847 (1847), pp. 536–538.

[9] Haskell B Curry. "The method of steepest descent for non-linear minimization problems." In: *Quarterly of Applied Mathematics* 2.3 (1944), pp. 258–261.

[10] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. *OpenAI Baselines*. https://github.com/openai/baselines. 2017.

[11] Prafulla Dhariwal, Heewoo Jun, Christine Payne, Jong Wook Kim, Alec Radford, and Ilya Sutskever. "Jukebox: A generative model for music." In: *arXiv preprint arXiv:2005.00341* (2020).

[12] Vincent Dumoulin and Francesco Visin. "A guide to convolution arithmetic for deep learning." In: *arXiv preprint arXiv:1603.07285* (2016).

[13] Chelsea Finn, Sergey Levine, and Pieter Abbeel. "Guided cost learning: Deep inverse optimal control via policy optimization." In: *International conference on machine learning*. 2016, pp. 49–58.

[14] Chelsea Finn, Paul Christiano, Pieter Abbeel, and Sergey Levine. "A connection between generative adversarial networks, inverse reinforcement learning, and energy-based models." In: *arXiv preprint arXiv:1611.03852* (2016).

[15] Justin Fu, Katie Luo, and Sergey Levine. "Learning robust rewards with adversarial inverse reinforcement learning." In: *arXiv preprint arXiv:1710.11248* (2017).

[16] Cristina Gârbacea, Aäron van den Oord, Yazhe Li, Felicia SC Lim, Alejandro Luebs, Oriol Vinyals, and Thomas C Walters. "Low bit-rate speech coding with VQ-VAE and a WaveNet decoder." In: *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2019, pp. 735–739.

[17] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial nets." In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[18] Robert Gray. "Vector quantization." In: *IEEE Assp Magazine* 1.2 (1984), pp. 4–29.

[19] Ronald A Howard. *Dynamic programming and markov processes.* MIT Press, Cambridge, 1960.

[20] Sham M Kakade. "A natural policy gradient." In: *Advances in neural information processing systems*. 2002, pp. 1531–1538.

[21] Jagat Narain Kapur and Hiremaglur K Kesavan. "Entropy optimization principles and their applications." In: *Entropy and energy dissipation in water resources*. Springer, 1992, pp. 3–20.

[22] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization." In: *arXiv preprint arXiv:1412.6980* (2014).

[23] Diederik P Kingma and Max Welling. "Auto-encoding variational bayes." In: *arXiv preprint arXiv:1312.6114* (2013).

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks." In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[25] Maxim Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd, 2018.

[26] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. "Object recognition with gradient-based learning." In: *Shape, contour and grouping in computer vision*. Springer, 1999, pp. 319–345.

[27] Honglak Lee, Alexis Battle, Rajat Raina, and Andrew Y Ng. "Efficient sparse coding algorithms." In: *Advances in neural information processing systems*. 2007, pp. 801–808.

[28] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. "Continuous control with deep reinforcement learning." In: *arXiv preprint arXiv:1509.02971* (2015).

[29] Bing Liu, Wynne Hsu, and Yiming Ma. "Integrating classification and association rule mining." In: *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*. 1998, pp. 80–86.

[30] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity." In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[31] Donald Michie, Michael Bain, and J Hayes-Miches. "Cognitive models from subcognitive skills." In: *IEE control engineering series* 44 (1990), pp. 71–99.

[32] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. "Playing atari with deep reinforcement learning." In: *arXiv preprint arXiv:1312.5602* (2013).

[33] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. "Human-level control through deep reinforcement learning." In: *Nature* 518.7540 (2015), p. 529.

[34] M. A. Moussa and M. S. Kamel. "A connectionist model for learning robotic grasps using reinforcement learning." In: *Proceedings of International Conference on Neural Networks (ICNN'96)*. Vol. 3. 1996, 1771–1776 vol.3.

[35] Andrew Y Ng, Stuart J Russell, et al. "Algorithms for inverse reinforcement learning." In: *Icml*. Vol. 1. 2000, p. 2.

[36] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, and CV Jawahar. "Cats and dogs." In: *2012 IEEE conference on computer vision and pattern recognition*. IEEE. 2012, pp. 3498–3505.

[37] Bilal Piot, Matthieu Geist, and Olivier Pietquin. "Bridging the gap between imitation learning and inverse reinforcement learning." In: *IEEE transactions on neural networks and learning systems* 28.8 (2016), pp. 1814–1826.

[38] Dean A Pomerleau. "Alvinn: An autonomous land vehicle in a neural network." In: *Advances in neural information processing systems*. 1989, pp. 305–313.

[39] Ali Razavi, Aaron van den Oord, and Oriol Vinyals. "Generating diverse high-fidelity images with vq-vae-2." In: *Advances in Neural Information Processing Systems*. 2019, pp. 14866–14876.

[40] Herbert Robbins and Sutton Monro. "A stochastic approximation method." In: *The annals of mathematical statistics* (1951), pp. 400–407.

[41] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[42] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[43] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. Vol. 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[44] Stuart Russell. "Learning agents for uncertain environments." In: *Proceedings of the eleventh annual conference on Computational learning theory*. ACM. 1998, pp. 101–103.

[45] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. "Trust region policy optimization." In: *International conference on machine learning*. 2015, pp. 1889–1897.

[46] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. "Proximal policy optimization algorithms." In: *arXiv preprint arXiv:1707.06347* (2017).

[47] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. "Deterministic Policy Gradient Algorithms." In: *International Conference on Machine Learning*. 2014, pp. 387–395.

[48] Richard S Sutton. "Temporal credit assignment in reinforcement learning." In: *PhD thesis, University of Massachusetts* (1985).

[49] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[50] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. "Policy gradient methods for reinforcement learning with function approximation." In: *Advances in neural information processing systems*. 2000, pp. 1057–1063.

[51] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude." In: *COURSERA: Neural networks for machine learning* 4.2 (2012), pp. 26–31.

[52] Emanuel Todorov, Tom Erez, and Yuval Tassa. "Mujoco: A physics engine for model-based control." In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2012, pp. 5026–5033.

[53] Aaron Tucker, Adam Gleave, and Stuart Russell. "Inverse reinforcement learning for video games." In: *arXiv preprint arXiv: 1810.10593* (2018).

[54] Aaron Van Den Oord, Oriol Vinyals, et al. "Neural discrete representation learning." In: *Advances in Neural Information Processing Systems*. 2017, pp. 6306–6315.

[55] Christopher John Cornish Hellaby Watkins. "Learning from delayed rewards." In: *PhD thesis, King's College, Cambridge* (1989).

[56] Marco Wiering and Martijn Van Otterlo. *Reinforcement Learning*. Vol. 12. Springer, 2012.

[57] Ronald J Williams and David Zipser. "A learning algorithm for continually running fully recurrent neural networks." In: *Neural computation* 1.2 (1989), pp. 270–280.

[58] Yanxia Zhang and Yongheng Zhao. "Automated clustering algorithms for classification of astronomical objects." In: *Astronomy & Astrophysics* 422.3 (2004), pp. 1113–1121.

[59] Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, and Anind K Dey. "Maximum Entropy Inverse Reinforcement Learning." In: *AAAI*. Vol. 8. Chicago, IL, USA. 2008, pp. 1433–1438.