Master in Artificial Intelligence

Master Thesis

Generative Query Networks for World Models in 2D and 3D Environments

Author:
Marek Merten

Supervisors:
Cecilio Angulo
Mario Martin

Universitat Politècnica de Catalunya
Universitat de Barcelona
Universitat Rovira i Virgili

21 January 2020

Abstract

In this thesis, the application of World Models [8] in 2D and 3D environments is explored. Particularly in the CarRacing-v0 environment [3, 16] and the DeepMind Lab [1] environment.

It is shown that the Variational Autoencoder [12] used in the World Models architecture has some drawbacks and a Generative Query Network (GQN) [4] is a viable alternative for the vision component and allows an agent trained with those models to achieve higher scores in both environments. The ability of a GQN to compute a structural representation, which is invariant to changes in rotation and small changes in position, is demonstrated.

Additionally, it is shown that a fast parallel training of agents can significantly reduce the training time necessary to train a full World Model.

Contents

List of Figures

List of Tables

List of Videos

Acronyms

RL	Reinforcement Learning
VAE	Variational Autoencoder
GQN	Generative Query Network
MDN	Mixture Density Network
RNN	Recurrent Neural Network
CNN	Convolutional Neural Network
LSTM	Long Short-Term Memory
ReLU	Rectified Linear Unit
CMA-ES	Covariance Matrix Adaptation Evolution Strategy
UMAP	Uniform Manifold Approximation and Projection
DRAW	Deep Recurrent Attentive Writer
CPU	Central Processing Unit
GPU	Graphics Processing Unit

Introduction

1.1 Motivation

Most Reinforcement Learning (RL) approaches rely on taking observations of an environment and applying actions to it step by step. This leads to very high training times of days or weeks for big problems. The environment quickly becomes the bottleneck since most RL training can be done rather fast on modern hardware.

An improvement to this issue is the architecture called World Models described in [8]. Ha and Schmidhuber propose to use a Variational Autoencoder [12] to extract feature vectors from the environment. A Mixture Density Network [2] combined with a Recurrent Neural Network (MDN-RNN) is then trained on the feature vectors to model the probability of future states based on which actions are taken. A simple controller can be trained by employing default RL approaches to take actions based upon what the RNN predicts. After this step, the agent can then take an action on the actual environment. (See Figure 1.1)

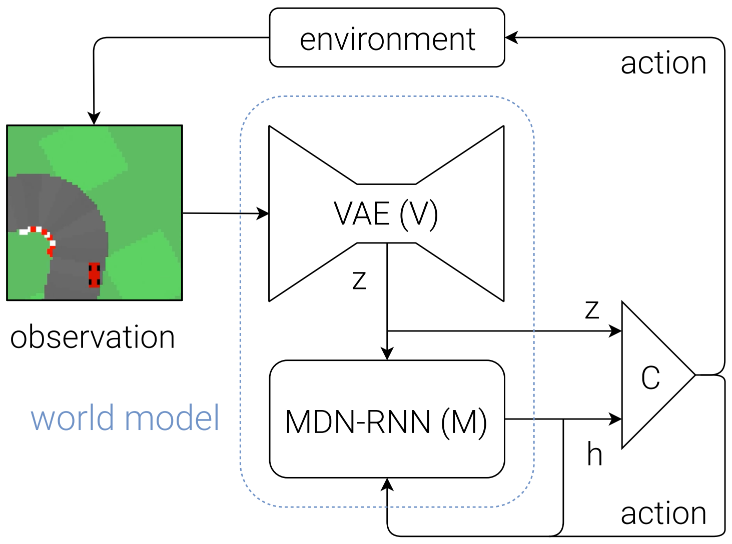


Figure 1.1: World Model architecture [8]

This idea of this architecture is based on how humans perceive their environment and models a similar ability of the human mind. That is the ability to predict the future based on imagined scenarios. This process is useful to reduce the time needed to train the architecture by splitting it up into different models.

One of the most crucial parts of the World Model architecture is the Variational Autoencoder. The quality of its output vector significantly influences of how well the training of the rest of the model works. Ideally, the VAE should have a high capability to generalize observed data and encode the structure of the perceived environment.

In the paper ”Neural scene representation and rendering”, Eslami et al. [4] describe an interesting alternative to VAEs called Generative Query Network (GQN) which offers a higher capability to abstract the structure of a 3D environment and reduced learning times when used together with RL. The idea is to train the GQN by having an input of several observations from different angles of a 3D scene and then ask a generation network to reconstruct the view from different angles based on the feature vector the network has created. This forces the encoder part to capture the structure and properties of the objects (e.g. color and shape) within the scene.

1.2 Objectives

The main objective of this thesis is to explore the viability of using an Generative Query Network (GQN) instead of an Variational Autoencoder (VAE) in the World Models architecture described by Ha and Schmidhuber [8] and to compare the resulting encoded representations that represent the model's vision.

Additionally, it is examined how well the VAE and GQN perform in a 3D navigation environment (e.g. what differences there are between 2D and 3D environments).

Another objective of this thesis is to explore ways of efficient parallel model training to reduce the time required to train agents on different environments. This objective originated from the fact that the original implementation of the World Models architecture [8] can take weeks to fully train the described models on a single GPU desktop computer. This is not acceptable if hyperparameter tuning is used and several different models are trained.

Environments

The models developed for this thesis were trained and evaluated on two different environments: the CarRacing-v0 environment [3, 16] and the DeepMind Lab [1] environment.

The following definitions will be used to describe the environments:

- $\mathcal{S}$ is the state space
- $\mathcal{A}$ is the action space
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the reward function, which maps a state and action to a scalar reward
- $s \in \mathcal{S}$ is the (observable) state of the environment.
  At each time step the environment returns an observation vector o which is a partial observation of the internal state of the environment. For simplicity in the following chapters a notation where $o = s$ will be used.
- $a \in \mathcal{A}$ is an action that is performed on the environment

The CarRacing and DeepMind Lab environments both offer a Python [19] interface that allows to reset the environment, perform an action (advancing the environment one step further), and receive an observation and reward for the currently executed step.

2.1 CarRacing v0 Environment

The CarRacing-v0 environment [16] is part of the collection of machine learning environments called OpenAI gym [3]. In each trial, a closed circuit track consisting of 1000 tiles is randomly generated and the agent is rewarded for visiting as many tiles of the track as possible. Additionally, the agent is punished with a negative reward of -0.1 at each time step if it has not finished a lap round the track yet. This results in a theoretical maximum achievable reward of around 950 and it gets increasingly more difficult to get closer to this maximum.

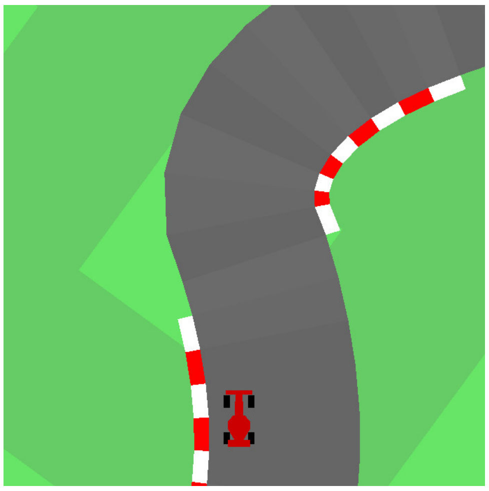


Figure 2.1: CarRacing environment (rendered in higher resolution)

The environment contains a physics simulation of each tire and the body of the car. Furthermore, the traction on different surfaces (road, grass) is modeled. If the car accelerates too much, it will not be able to make hard turns and starts to spin around its own axis. If the car leaves the road and drives on the more slippery grass, this effect is amplified. Therefore, the challenge of an agent navigating this environment is to find the right balance of steering, accelerating on straight sections of the track, and braking before entering turns.

Each observation (s) of the environment is an image with a width and height of 64 pixels and 3 color channels (red, green, blue). Therefore, the observable state space is a subset of $\mathbb{R}^{64\times64\times3}$ with values ranging from 0.0 to 1.0.

The CarRacing environment has a three dimensional continuous action space ($\mathbb{R}^3$) with the following possible actions:

- steer left/right (a_{steer})
- accelerate (a_{acc})
- decelerate (brake) (a_{dec})

This can be described with the following vector a':

$$a' = \begin{pmatrix} a_{steer} \\ a_{acc} \\ a_{dec} \end{pmatrix} \tag{2.1}$$

In order to reduce the complexity of the model and simplify the controller used to interact with the environment, the implementation in this thesis maps the 3 dimensional action space to a two dimensional action space ($\mathcal{A} \subset \mathbb{R}^2$) by combining the accelerate and decelerate actions:

$$a = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} a_{steer} \\ |a_{acc}| - |a_{dec}| \end{pmatrix} \tag{2.2}$$

To perform an action on the CarRacing environment the original action vector a' is needed. However, the new action vector a will be used for the models discussed later. Thus, it is necessary to apply the following transformation to a to acquire the original action vector a':

$$a' = \begin{pmatrix} a_{steer} \\ a_{acc} \\ a_{dec} \end{pmatrix} = \begin{pmatrix} a_1 \\ \max(a_2, 0) \\ \min(a_2, 0) \end{pmatrix} \tag{2.3}$$

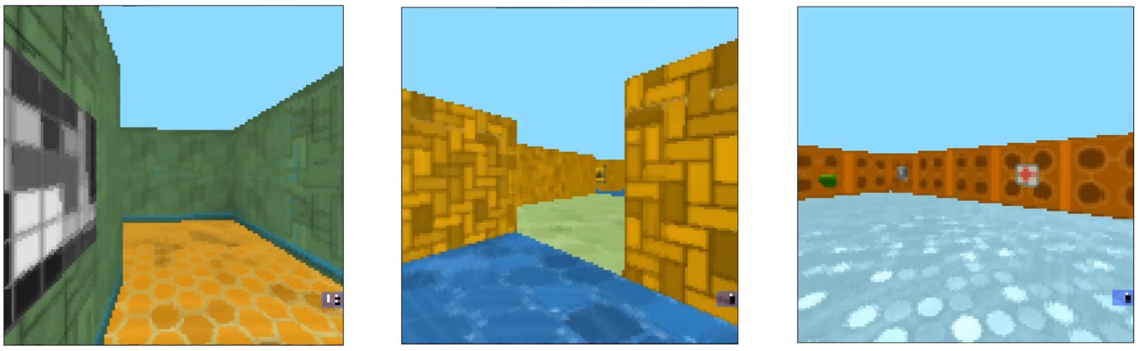
Rendering performance

The OpenAI gym uses a relatively slow rendering system based on OpenGL [17] in Python. It requires a GPU to render each environment and scales poorly when running multiple environments simultaneously and therefore can quickly become a bottleneck when training any RL model on it. To improve the rendering performance and the ability to run it in parallel, a 3D mesh renderer developed by Genova et al. [5] has been used instead. This allows for fast parallel rendering on CPU cores and assures that GPU resources are fully available to train agent models.

Specifically, IBM Power9 nodes, which execute 160 parallel threads and contain 4 NVIDIA V100 GPUs provided by the Barcelona Supercomputing Center, were used to train the models for this thesis.

2.2 DeepMind Lab Maze Environment

DeepMind Lab [1] is a 3D learning environment which contains a variety of different customizable environments for navigation and puzzle solving tasks. Three different mazes have been chosen from these environments to train the models developed for this thesis (Figure 2.2). To offer a wider range of variability the agent is randomly placed in one of the three mazes at a random start position. The agent is rewarded for visiting new areas within the maze and punished for staying in areas it has already explored. This way, the agent is encouraged to develop a sense of curiosity.



(a) Maze 1 (b) Maze 2 (c) Maze 3

Figure 2.2: Different DeepMind Lab mazes

Similar to the CarRacing environment, each observation is a frame that is rendered with a width and height of 64 pixels and 3 color channels.

The original DeepMind Lab environment has 7 different possible actions an agent can perform: look left or right, look up or down, strafe left or right, move forward or backward, fire, jump and crouch.

For simplicity, the action space used in this thesis has been reduced to two possible continuous actions encoded with real values ($\mathcal{A} \subset \mathbb{R}^2$):

- turn left or right
- move forward

The ability of an agent to move backward has been removed to avoid finding policies where the agent never moves forward and only explores the environment by going backward, which would offer the advantage that the agent still sees most of the surroundings even if it moves into a corner or hits a wall. By only going forward, the agent is more likely to deal with the difficult situation of directly standing in front of a wall and having to decide to either go left or right. Before an action is applied to the actual environment, it is translated to the original 7 dimensional discretized action space.

In order to speed up the training process every action performed on the environment is repeated 8 times. This way, an observation has to be obtained from the environment only on every 8th step and the training time is reduced considerably. Additionally this stabilizes the way the agent interacts with the environment by making its movements less jittery.

The custom reward function defined to evaluate the agent on the mazes is the following:

$$\mathcal{R}(s,a) = \begin{cases} 1 & \text{if } \|a\| \neq 0 \text{ and tile not visited} \\ -0.2 & \text{if } \|a\| \neq 0 \text{ and tile already visited} \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

Architecture

In this chapter, the architecture developed based on the World Models paper by Ha and Schmidhuber [8] is discussed. Furthermore, different ways of combining its parts and an approach on how to efficiently train the model in parallel are presented. The World Model architecture consists of three main parts:

- The Vision (V) model, which processes input images and returns a compact representation of what it sees.
- The Memory (M) model, which takes actions and representations to predict future observations.
- The Controller (C) model, which based on the encoded representation from the V model and internal state of the M model performs actions on the environment.

3.1 Vision (V) Model

Although training an RL agent directly on observed images is certainly possible, as demonstrated by Deep Q-Learning [14], where the image processing (vision) and action calculation is done in a single neural network, splitting up the agents logic into a separate vision and controller model has certain advantages. A vision (V) model can be trained in an unsupervised manner (regarding the task an agent has to solve) and does not necessarily require the agent to perform meaningful actions. In the simplest case it can be trained on a random rollout, where an agent follows a random policy to interact with the environment. Another way to train the V model is to randomly place the agent at different coordinates within the environment, obtain an observation, place it in a different location and repeat the process. This approach has been chosen for this thesis to train the V model.

In general, the V model consists of two parts:

- The encoder that processes observations (RGB images) and encodes them into a latent vector z.
- The decoder that reconstructs an RGB image from the latent vector z. The quality of this reconstruction can then be used as a loss function to train the complete V model.

The original World Models paper proposes to use a Variational Autoencoder as vision model to encode each observed frame into a lower dimensional representation (latent vector). This is achieved trough a Convolutional Neural Network (CNN), which processes the input images by applying convolutional kernels to them. The last output layer of the CNN is used to compute the mean μ and standard deviation σ of an observation. From this, a latent vector z is sampled.

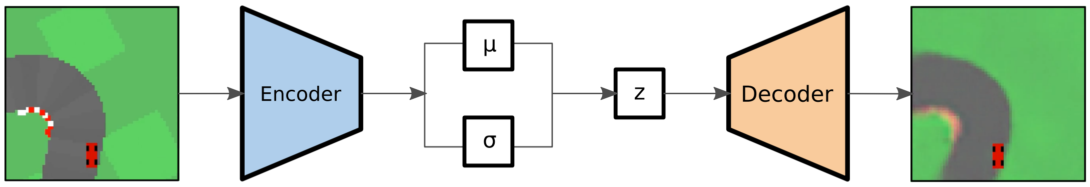


Figure 3.1: VAE model

Figure 3.1 shows the VAE model. It encodes a single input image with a size of 64x64 pixels and three color channels into a 32 dimensional latent vector.

In contrast to the VAE model, the Generative Query Network (GQN) (Figure 3.2) is trained with several input images at a time. Between 1 and 5 input images were encoded into a 64 dimensional representation for the CarRacing-v0 and DeepMind Lab environment.

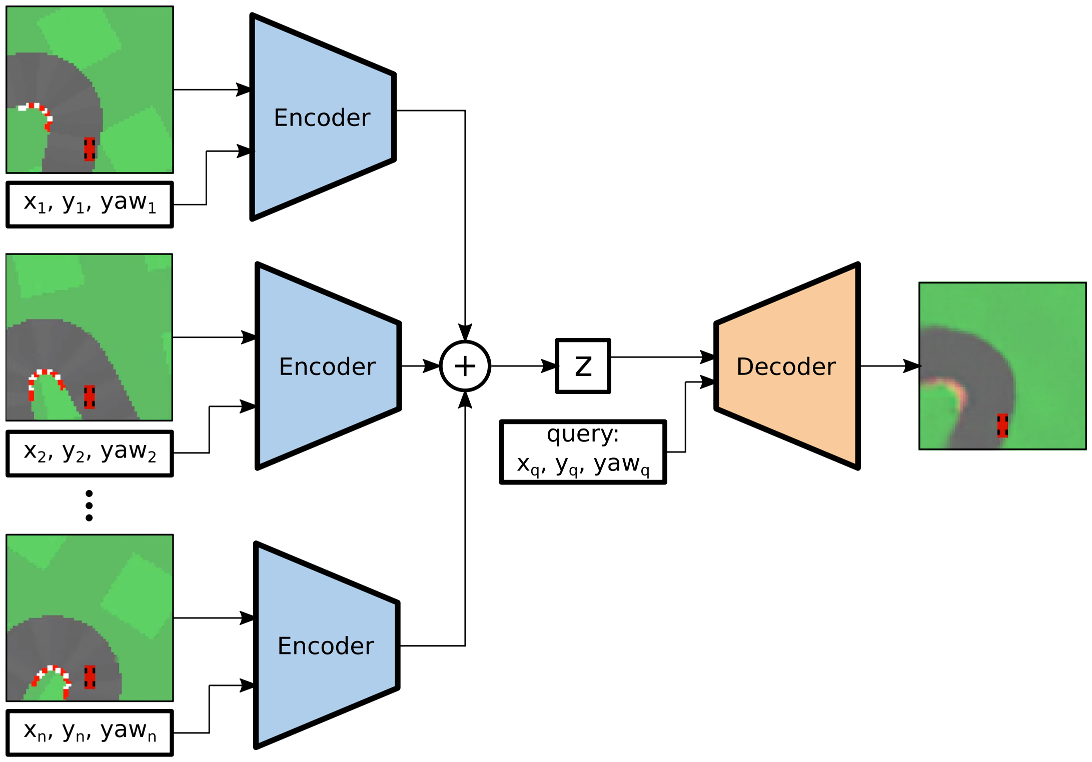


Figure 3.2: GQN model

The GQN uses the convolutional Deep Recurrent Attentive Writer (DRAW) architecture described by Gregor et al. [6] as a decoder. This decoder is based on recurrent convolutional layers and is originally used as a form of conceptual compression algorithm [6]. This means that it allows a compressed vector to contain global information about the scene and less important details are filled in by the model. In the most extreme case an image could be compressed down to a single bit. This bit could, for instance, represent either a cat or a dog. The DRAW architecture then takes this bit and generates a likely image of the defined animal. If there are more bits available in the compressed image, then these will be used to encode additional features like the color and shape, and allow for a more precise reconstruction of the original image.

Later, only the encoder part will be used as vision model of an agent and the decoder is only required during the training of the V model and to evaluate the quality of encoded representations visually. However, the architecture of the decoder matters substantially, because it forces the encoder to generate a specific representation that is compatible with the decoder. In other words, the encoder is trained to match the decoder, due to the fact that the gradient of the training loss flows backwards from the final decoded image.

The original GQN implementation uses a 7 dimensional view description, which consists of x,y,z coordinates and sine and cosine of yaw and pitch angles. For this thesis, this has been reduced to a 4 dimensional description of its position and angle, which will be called view ($v \in \mathbb{R}^4$). It consists of:

- Relative Δx and Δy
- Sine and cosine of relative yaw angle ($\Delta\psi$)

The coordinates are transformed using a rotation matrix R_ψ:

$$R_\psi = \begin{pmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{pmatrix}$$

The mean is subtracted from the absolute coordinates and angle and the resulting difference is rotated by angle ψ:

$$\begin{aligned} \Delta\, x &= R_\psi \cdot (x - \bar{x}) \\ \Delta\, y &= R_\psi \cdot (y - \bar{y}) \\ \Delta\psi &= \qquad \psi - \bar{\psi} \end{aligned} \tag{3.1}$$

$$v = (\Delta x, \Delta y, \sin(\Delta\psi), \cos(\Delta\psi)) \tag{3.2}$$

In case of using a single image, the coordinates and angles are equal to their mean:

$$\begin{aligned} x &= \bar{x} \\ y &= \bar{y} \\ \psi &= \bar{\psi} \end{aligned}$$

Therefore, the relative position and angle become zero:

$$\Delta x = \Delta y = \Delta \psi = 0$$

Hence the view v_0 of a single input image is constant:

$$v_0 = (0, 0, 0, 1) \tag{3.3}$$

Encoder models

Both encoders for the VAE and GQN are composed of convolutional layers to process the input image.

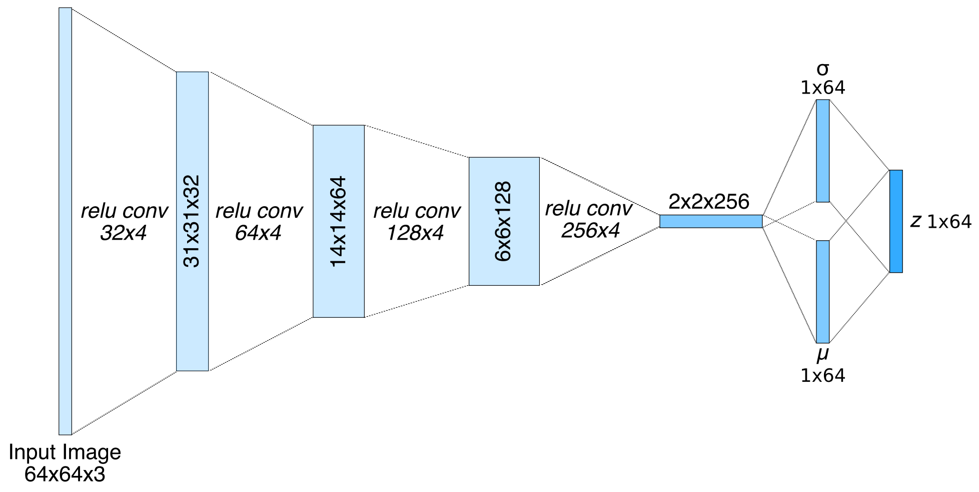


Figure 3.3: VAE encoder

The encoder part of the VAE model (Figure 3.3) consists of 4 convolutional layers with Rectified Linear Unit (ReLU) activation and two densely connected-layers to compute the mean μ and standard deviation σ.

The latent vector z can then be computed by sampling randomly from a distribution with the mean μ and standard deviation σ:

$$z = \mu + \sigma \odot \mathcal{N}(0, I) \tag{3.4}$$

In case of the VAE, the encoded representation z_t is equal to the sampled latent vector z at time step t:

$$z_t = z(t) \tag{3.5}$$

For the GQN, each input image and its view v_i is encoded by a pool encoder, which consists of 6 convolutional layers and an average pooling operation at the end. After the third convolutional layer, the view v_i is broadcasted to match the shape of the convolutional layer and concatenated into its output. This way, for each input image and view, a representation r_i is generated, as visualized in Figure 3.4.

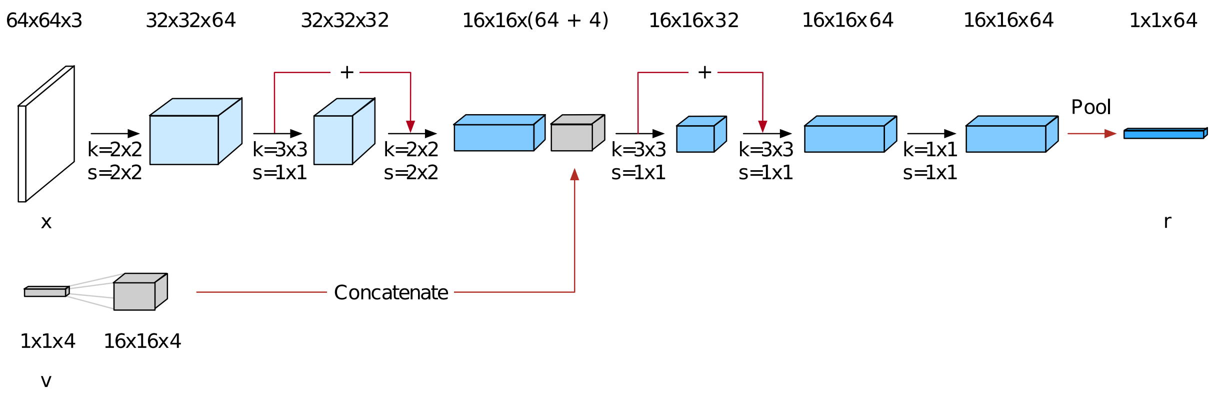


Figure 3.4: GQN pool encoder [4]

In case of the GQN the encoded representation z_t is equal to the sum of input representations at time step t:

$$z_t = \sum_i r_i(t) \tag{3.6}$$

Model size

The amount of parameters needed to train a model properly is an indicator of how powerful it is. Generally, bigger models tend to yield better results, but less parameters are preferred in order to minimize computational cost and memory requirements. Therefore, choosing the right model size with the least amount of parameters necessary to achieve good results, is a crucial part of the architecture's design.

Model	**Parameter count**
VAE encoder	755 744
GQN encoder	72 352

Table 3.1: Parameter count of encoder used for CarRacing task

For the VAE model, the identical model as defined in the original World Models architecture [8] was used. The implementation of the GQN for this thesis (based on the work of O. Groth [15]) offers to define the number of units of the convolutional layers as an adjustable hyperparameter. Hence, the encoder part of the V model could be trained for the GQN with a parameter count 10 times fewer than for the VAE model, as illustrated in Table 3.1.

Model	**Parameter count**
VAE decoder	3 592 803
GQN decoder	1 020 483

Table 3.2: Parameter count of decoder used for CarRacing task

The parameter count of the decoder is considerably higher than in the encoder part of the V model (see Table 3.2). Nevertheless, the decoder of the GQN employing the DRAW architecture requires 3 times fewer parameters.

3.2 Memory (M) Model

In order to mimic the human ability to foresee how an environment will change based on the own movements and actions taken, Ha and Schmidhuber propose to incorporate a memory model into the World Models architecture.

This is achieved by combining a Mixture Density Network [2] with a Recurrent Neural Network (MDN-RNN) and training it on encoded representations to model the probability of future states based on what actions are taken. This can be written as $P(z_{t+1} \mid a_t, z_t, h_t)$ and is estimated through a mixture of Gaussians [7] in the original model that is used to together with the VAE vision model. The recurrent part of the model consists of Long Short-Term Memory (LSTM) cells [11], which are connected through the hidden state h at each time step as illustrated in Figure 3.5.

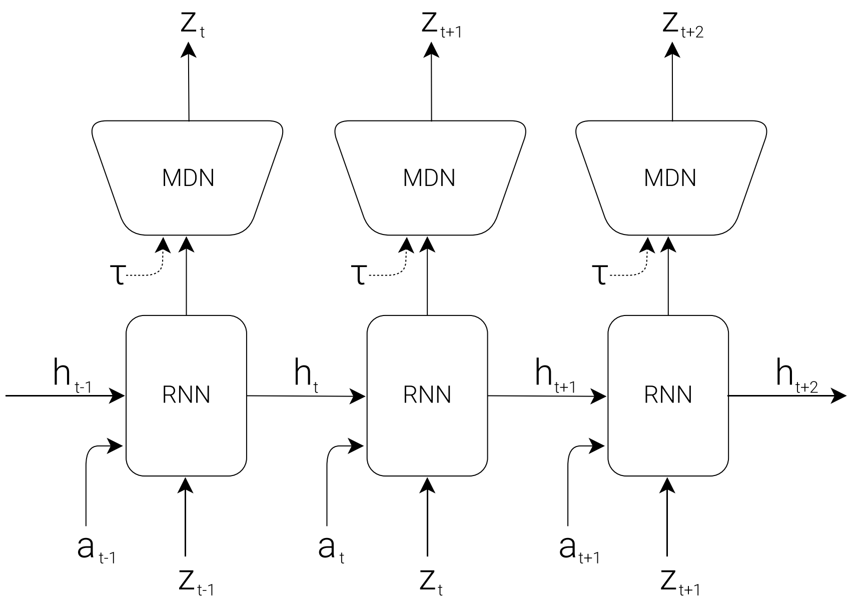


Figure 3.5: MDN-RNN (source: [8])

During the sampling of z_{t+1}, the model uncertainty can be adjusted by a temperature parameter τ, as described by Ha and Eck [7]. Lower values of τ cause a more deterministic prediction of z_{t+1} and higher values cause the the output vectors to become more random and unpredictable.

The original M model uses a single densely-connected layer for the Mixture Density Network. The implementation for this thesis uses 3 fully connected layers in order to improve the quality of predicted states instead. Furthermore, it is not limited to predicting z_{t+1}. Instead, a tunable hyperparameter δ is used to allow the MDN-RNN to predict future vectors $z_{t+\delta}$ that are further than one step ahead, if necessary. However, the model has to be trained separately for different values of δ.

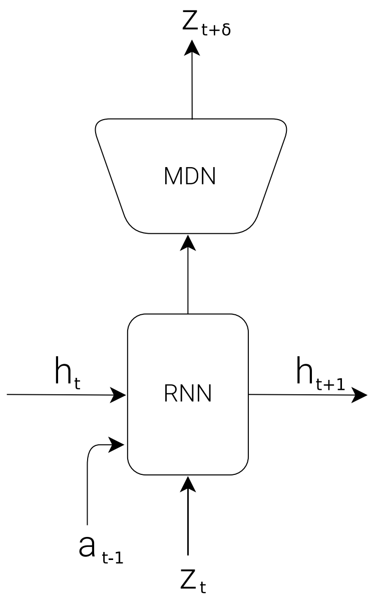


Figure 3.6: MDN-RNN modified with action a_{t-1} and $z_{t+\delta}$

One issue with the original implementation from Ha and Schmidhuber [8] is the fact that, in order to predict the future representation $z_{t+\delta}$, it is necessary to know the internal RNN state h_t, the current state of the environment (encoded as z_t), and the current action a_t. However, it is not possible to obtain $z_{t+\delta}$ with a single prediction step if the calculation of the action a_t depends on it, but it is desirable to allow an agent to use this predicted future representation as a means of anticipating changes in its environment.

For this reason, the implementation for this thesis uses the previous action a_{t-1} instead of a_t together with z_t to predict $z_{t+\delta}$ and to update the internal RNN state h_{t+1} (see Figure 3.6). This has the advantage that a_t can be a function of $z_{t+\delta}$:

$$a_t = f_a(h_{t+1}, z_t, z_{t+\delta}) \tag{3.7}$$

With $z_{t+\delta}$ being a function dependent on h_t, z_t and a_{t-1}:

$$z_{t+\delta} = f_z(h_t, z_t, a_{t-1}) \tag{3.8}$$

Now Equation 3.7 and Equation 3.8 together with $\delta = 1$ result in:

$$\begin{aligned} z_{t+1} &= f_z(h_t, z_t, f_a(h_t, z_{t-1}, z_{t+\delta-1})) && (3.9) \\ &= f'_z(h_t, z_{t-1}, z_t) && (3.10) \end{aligned}$$

Which means a predicted future state z_{t+1} can be obtained from previous states and the internal state of the RNN.

And similar h_t can be written as:

$$h_{t+1} = f_h(h_t, z_t, a_{t-1}) \tag{3.11}$$

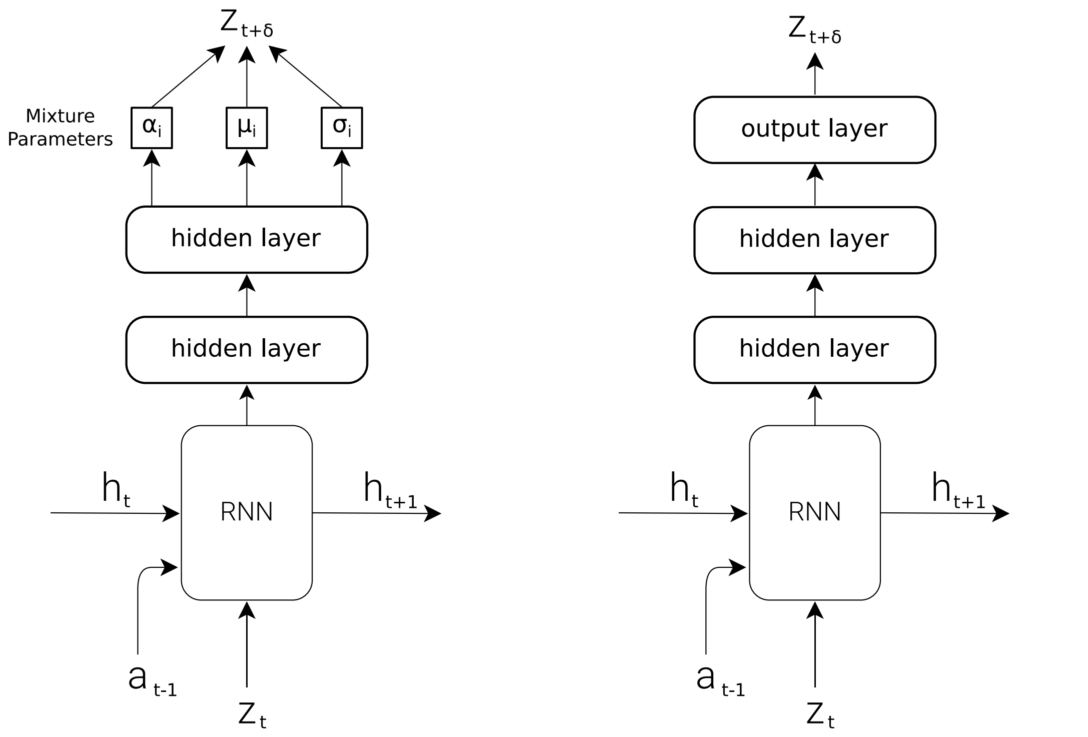
Now the equations 3.7, 3.10 and 3.11 together (with $\delta = 1$) result in:

$$\begin{aligned} a_t &= f_a(f_h(h_t, z_t, a_{t-1}), z_t, f'_z(h_t, z_{t-1}, z_t)) && (3.12) \\ &= f'_a(a_{t-1}, h_t, z_t, z_{t-1}) && (3.13) \end{aligned}$$

Equation 3.13 demonstrates that even though an action a_t uses the future predicted representation $z_{t+\delta}$, it can be computed from just the previous states and actions. The original model described by Ha and Schmidhuber was not able to achieve this, unless two predictions were made (predicting z_{t+1} and z_{t+2} in two separate steps), due to a cyclic dependency between a_t and z_{t+1} (both would depend on each other).

Furthermore, it should be noted that the MDN-RNN is forced to internally learn a policy of how an action a_t is implicitly computed from a_{t-1} and the states z_t and h_t. Consequently, it can not be trained independently from the used Controller (C) model.

In order to use the RNN together with a GQN model, a modification was made to the M model so that $P(z_{t+1} \mid a_{t-1}, z_t, h_t)$ is not modeled by a mixture of Gaussians, but instead $z_{t+\delta}$ is predicted directly without sampling from Gaussian distributions. This is visualized in Figure 3.7, where the graph on the left side shows the original MDN-RNN with additional hidden layers, and the right side the modified RNN architecture, which allows the use of a GQN as vision model.



(a) MDN-RNN for use with a VAE (b) Modified RNN for use with a GQN

Figure 3.7: RNN architecture comparison

As depicted in Figure 3.7a, $z_{t+\delta}$ is sampled from several mixture parameters for the MDN-RNN used together with the VAE. These parameters are the mixing coefficients α_i, the means μ_i and standard deviations σ_i.

Based on Bishop [2], the probability of $z_{t+\delta}$ can then be computed from α_i, μ_i and σ_i, with ϕ_i as a function that computes a Gaussian distribution, in the following way:

$$\begin{aligned} P(z_{t+\delta} \mid a_{t-1}, z_t, h_t) &= P(z_{t+\delta} \mid \alpha_i, \mu_i, \sigma_i^2) \\ &= \sum_i \alpha_i \; \phi_i(z_{t+\delta} \mid \mu_i, \sigma_i^2) \end{aligned} \tag{3.14}$$

Each prediction in the original MDN-RNN (Figure 3.7a) is computed with a mixture of 5 Gaussian distributions (i denotes the index of the distribution). This results in a total of 15 parameters which need to be calculated by the MDN for one prediction of $z_{t+\delta}$. On the contrary, $z_{t+\delta}$ is computed as a single parameter in Figure 3.7b for the modified model.

As a consequence of predicting $z_{t+\delta}$ directly in the case of the GQN, it is not possible to adjust the randomness of the sampled vector with the temperature parameter τ. However, to some extend a certain degree of randomness helps to improve training results. Therefore, this modification might have disadvantages in stability when an agent is trained using the Word Models architecture.

3.3 Controller (C) Model

In the simplest case, C is a single layer neural network that maps the input vector $x_t \in \mathbb{R}^{n_z}$ at time step t to an action $a_t \in \mathcal{A}$ (with $\mathcal{A}$ being the action space of the environment):

$$a_t = W_c \; x_t + b_c \tag{3.15}$$

$W_c \in \mathbb{R}^{n_z \times \dim \mathcal{A}}$ and $b_c \in \mathbb{R}^{\dim \mathcal{A}}$ are the weight matrix and bias vector and n_z is the number of elements in the input vector. In case of having a hidden layer in the C model, the equation is extended with the weight matrix W_h and bias b_h of the hidden layer:

$$a_t = W_c \; (W_h \; x_t + b_h) + b_c \tag{3.16}$$

The input vector x_t can either be the encoded representation z_t from the V model or a concatenated vector of either $[z_t h_t]$ or $[z_t z_{t+\delta}]$ when the Memory (M) model is used. h_t is the hidden state of the M model at at time step t. With a concatenated vector $[z_t z_{t+\delta}]$ an actual prediction of a future vector δ time steps ahead is used instead of h_t.

The C model is trained using Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [9]. In order to train the weights and biases of the C model, they need to be transformed into a vector of parameters $\theta \in \mathbb{R}^{n_\theta}$:

$$\theta = (W_{c_{i,j}}, b_{c_i}) \in \mathbb{R}^{n_\theta} \tag{3.17}$$

In case the model contains a hidden layer:

$$\theta = (W_{c_{i,j}}, b_{c_i}, W_{h_{i,j}}, b_{h_i}) \in \mathbb{R}^{n_\theta} \tag{3.18}$$

Where i and j iterate over all elements in each matrix and vector. n_θ is the number of parameters to train. In case of the single layer C model, the parameter count n_θ can be computed from the size of W_c and b_c in the following way:

$$n_\theta = (n_z + 1) \cdot \dim \mathcal{A} \tag{3.19}$$

The goal of the evolution strategy is then to minimize the cost function $J(\theta) : \mathbb{R}^{n_\theta} \rightarrow \mathbb{R}$ which is the negative sum of rewards obtained during t_{max} time steps:

$$J(\theta) = -\sum_{t=1}^{t_{max}} \mathcal{R}(s_t, a_t(\theta)) \tag{3.20}$$

$\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the reward function and $s_t \in \mathcal{S}$ is the state of the environment at time step t. $\mathcal{S}$ is the state space.

The environments used in this thesis are randomly initialized. (In case of the CarRacing environment a random track is generated and for the DeepMind Lab a random maze with a random initial position is chosen.) Thus, the obtained cost J for a set of parameters θ is nondeterministic. In order to compute a more stable cost for a set of parameters θ, it is necessary to perform several trials and average the obtained cost:

$$\bar{J}(\theta) = \frac{1}{k} \sum_{i=1}^{k} J_i(\theta) \tag{3.21}$$

k denotes the number of trials, on which the parameters are evaluated trough J.

The Covariance Matrix Adaptation Evolution Strategy is in initialized in the following way:

- **Sample random initial parameters** θ from a Cauchy distribution
- **Set population size** $\lambda \in \mathbb{N}$

Afterwards, the CMA-ES performs an update of the parameters θ with each generation, as follows [10]:

1. **Sample distribution** $P(x|\theta) \rightarrow x_1, \ldots, x_\lambda \in \mathbb{R}^{n_\theta}$
2. **Evaluate** $x_1, ..., x_\lambda$ k **times on** J
3. **Update parameters** $\theta \leftarrow F_\theta(\theta, x_1, \ldots, x_\lambda, \bar{J}(x_1), \ldots, \bar{J}(x_\lambda))$

With F_θ being the update function which CMA-ES uses. In order to evaluate the parameters on the cost function J, they are split up and reshaped to form the original weights and biases of the neural network C model.

3.4 Combined Models

Training the models on an environment is possible either with or without the M model. In case of training without the M model, an agent consists of the V model and C model as seen in Figure 3.8.

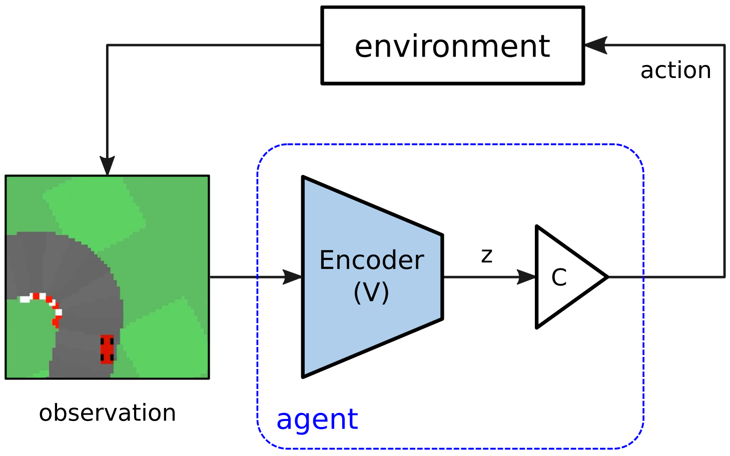


Figure 3.8: World Model using V and C only

There are only three components in this simpler scenario. The environment takes an action form the agent, and returns an observation and the reward for performing this action. The observation is encoded into the latent vector z_t and directly used to train the controller model. Because the agent does not contain any memory model, there is no internal state which is saved between time steps. This means the agent needs to extract all necessary information to, i.e., drive a car from the current static observation. This is a single image and therefore does not contain any information like the speed of the car.

The full World Model architecture also contains the M model (Figure 3.9). The MDN-RNN used as M model maintains a hidden state h between time steps. With each observation and performed action this internal state is updated.

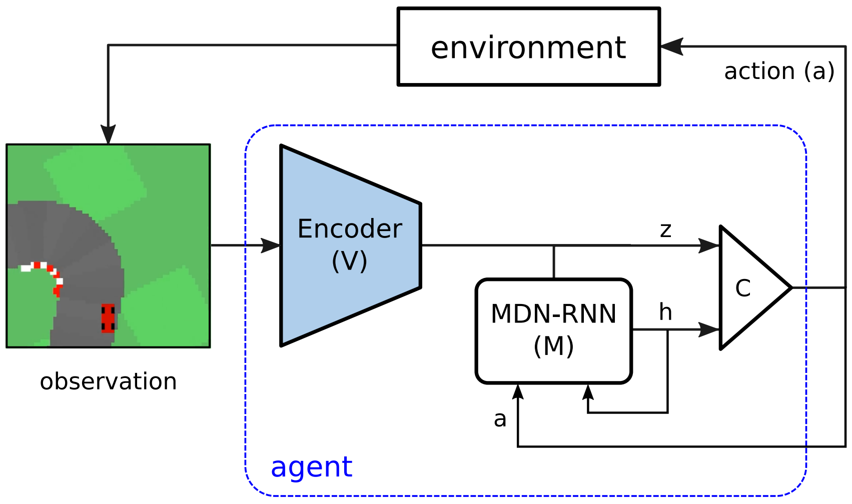


Figure 3.9: Full World Model using V, M and C

Contrary to the first agent architecture, this design contains a memory model. This allows the model to form a sense of short term memory. Based on the actions previously taken, the agent can make an estimation of the cars velocity and trajectory, which is encoded in the memory models hidden state h_t. A controller model can then use this information to take proactive actions. For instance, it allows an agent to know the exact amount of braking required to not miss a turn in the CarRacing environment.

Additionally, Ha and Schmidhuber [8] describe an interesting third approach to train the controller model exclusively based on the output of the memory model. In this case, the M model has to be pretrained and also predict the rewards obtained from each taken action. This allows the C model to be trained entirely without the actual environment and vision model, and only interact with a generated dream-like simulation instead. This has a dramatic impact on the required time to train the model, because the usually slow interaction with the real environment is not necessary and thus the M model can quickly predict the next state of environment. However, this approach is only feasible for simpler stochastic environments. More sophisticated environments require a better memory model, than the MDN-RNN trained for this thesis. For this reason, this approach has not been investigated any further.

Iterative Training Procedure

The World Models paper [8] proposes an iterative training procedure for the M and C model consisting of four steps (originally defined by Schmidhuber [18]):

1. Initialize M, C with random model parameters.
2. Rollout to actual environment N times and save all actions a_t and observations to storage.
3. Train M to model $P(z_{t+1} \mid a_t, z_t, h_t)$ as a mixture of Gaussians.
4. Go back to (2) if termination condition is not met.

This training procedure is usually implemented using a rather large number of rollouts N with steps (2) and (3) executed sequentially. This means the M and C models are restarted and loaded with new parameters in each iteration.

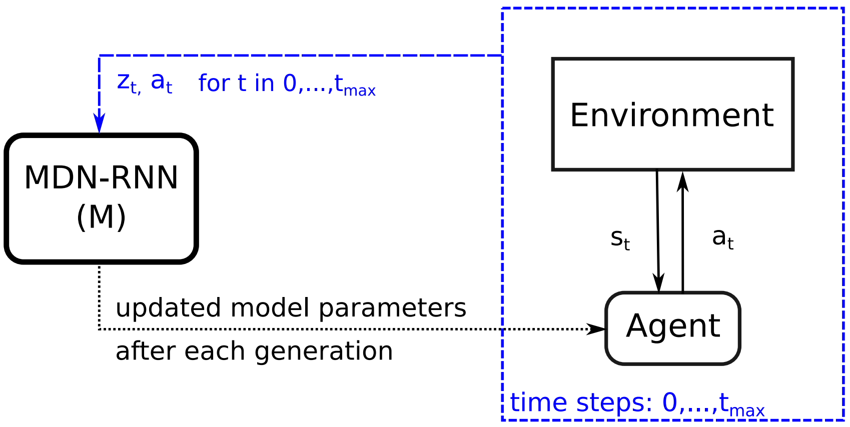

z_t, a_t for t in 0,...,t_{max}
MDN-RNN (M)
Environment
s_t
a_t
Agent
updated model parameters after each generation
time steps: 0,...,t_{max}

Figure 3.10: MDN-RNN concurrent training

In contrast, a mechanism has been developed for this thesis that allows to train the M and C models (steps (2) and (3)) in parallel (depicted in Figure 3.10). Each generation of the CMA-ES algorithm is considered a rollout and copied to the training process for the M model using shared memory. On the other hand, the training process of the MDN-RNN sends updated model parameters back to the agents running the CMA-ES algorithm. This way, the two models are updated at every generation and can quickly converge without having to be restarted.

Parallel Training

Ha and Schmidhuber [8] propose a default way of training the C model, which requires λ agents, each operating on a separate environment for k trials times t_{max} iterations using two nested loops that go through each time step and trial. At each time step, the agent has to process the current observation s_t with the V model and take an action based on the output of the C model. This results in model inference with a batch size of 1 at each time step (see Figure 3.11).

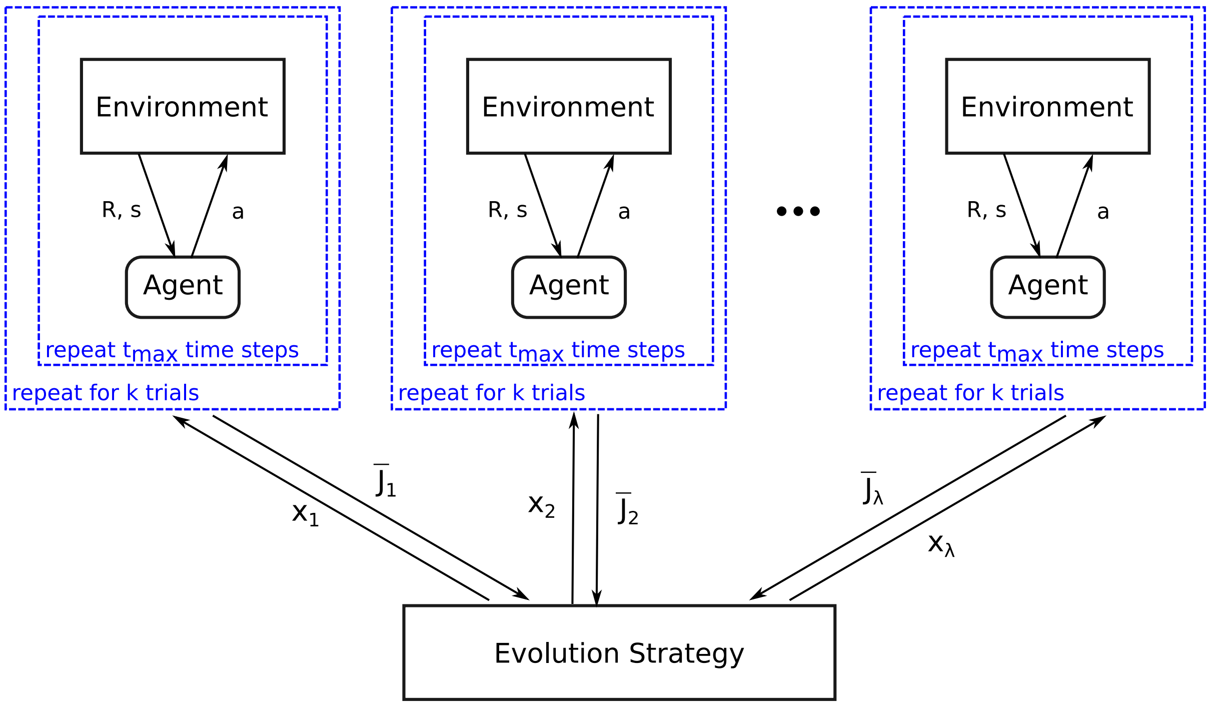


Figure 3.11: Evolution Strategy training with iterative agents

However, neural models are very effective in leveraging the parallel computing power of CPUs and GPUs by using bigger batch sizes. Therefore, a different approach has been developed for this thesis that does not use the outer loop of performing k trials over a set of parameters, but instead performs a single trial with a batch size of k. This is illustrated in Figure 3.12.

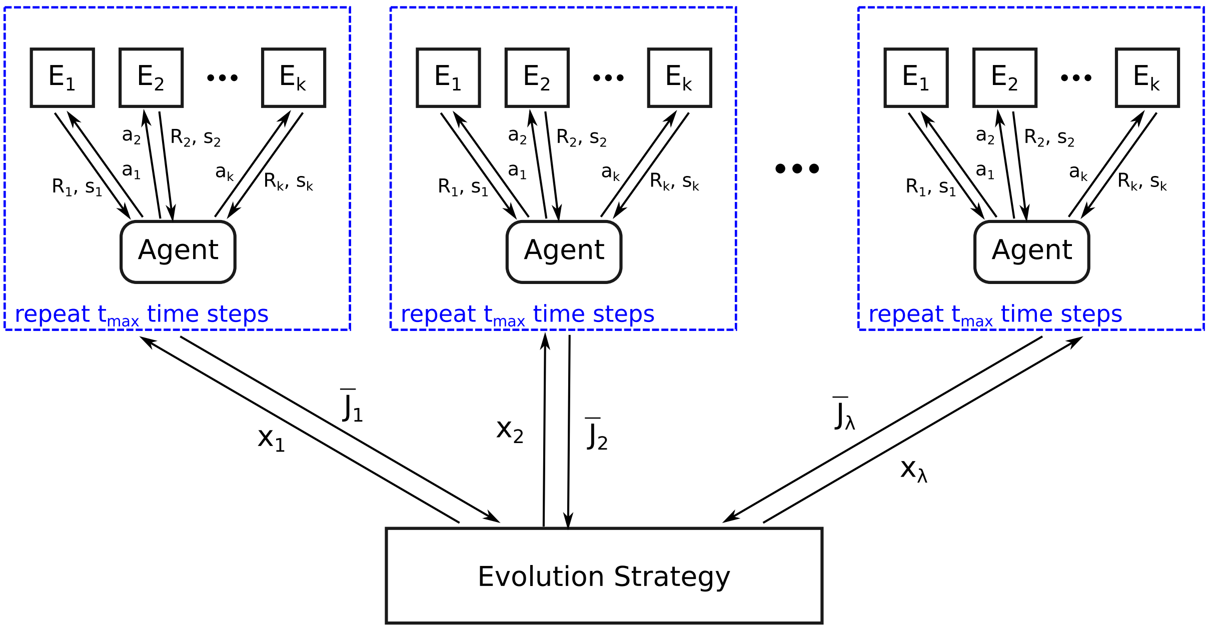


Figure 3.12: Evolution Strategy training with parallel (batch) agents

Each agent has k associated environments $(E_1, \ldots, E_k)$, which run in parallel. For that reason, the agent computes an action matrix $(a \in \mathbb{R}^{k \times \dim \mathcal{A}})$ instead of an action vector $(a \in \mathcal{A})$ at each time step. Table 3.3 shows the execution times of one iterative agent implementation and parallel implementation for 16 trials running on a computer with 16 virtual CPU cores. The model inference at each time step takes place on a GPU device.

Agent	**Time in** s
iterative agent	28.2
parallel agent	3.9

Table 3.3: Execution times per generation for the DeepMind Lab environment with $k = 16$ and $t_{max} = 2000$

In this case, the parallel agent implementation is about 7 times faster, although this factor can only be obtained when there are enough CPU cores available to run all environments in parallel. If there are not enough computing resources available, the speed advantage of the parallel implementation over the iterative implementation decreases. However, in contrast to gradient decent based training methods, which use back propagation, evolution strategy based training scales very well to multiple nodes of a cluster of computers [9]. Each node can execute a set of agents and return the calculated cost for a subset of the population that is trained by CMA-ES.

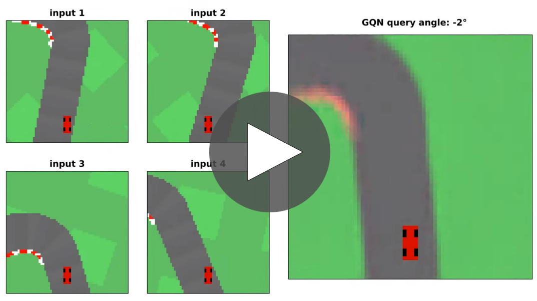
As a consequence, all results obtained for this thesis were run using the parallel (batch) agent approach to reduce training times.

Results

In this chapter, the results of applying the models developed in chapter 3 to the CarRacing and DeepMind Lab maze environments are presented. First, the result of training the GQN model on both environments is analyzed in section 4.1. Subsequently, section 4.2 compares the computed representations of the VAE and GQN models (V model). It is followed by a review of the scores obtained in the CarRacing environment in section 4.3 and 4.4. Finally, the results obtained from the DeepMind Lab maze are discussed in section 4.5.

4.1 Generative Query Network

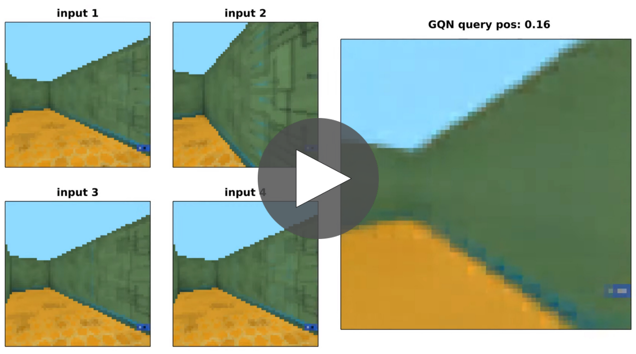
One of the advantages of the GQN is its ability to produce a representation which is mostly rotation and translation invariant. In order to test this property, the GQN was asked to render an output image with different query angles $\Delta\psi_q$ or a translational offset in x direction (Δx_q) to predict the resulting image. $\Delta\psi_q$ and Δx_q are continuously increased and decreased to show how precise the predictions are. The correctness of this prediction can then be easily evaluated by the human eye, because humans are able to do the same prediction of what happens when an object (in this case the street) is moved. For example, the shape of a two dimensional object should not change when it is rotated or shifted.



Video 4.1: GQN query rendering for the CarRacing environment

Video 4.1 demonstrates this by using 4 static input images and their corresponding relative views (v_i), which were taken from the CarRacing environment. The model is able to produce correct predictions about how the track looks like when the position and angle changes.

Similar to the 2D CarRacing environment, the GQN is also able to infer the correct shape of elements in a 3D scene if the viewing position changes. Video 4.2 demonstrates this for the DeepMind Lab maze environment. The model renders the floor, walls and the sky correctly, but due to the low resolution of 64×64 pixels, the rendering is not detailed and blurred. The transitions between frames are slightly jittery, but, overall, the GQN is able to capture the most important structures of the scene.

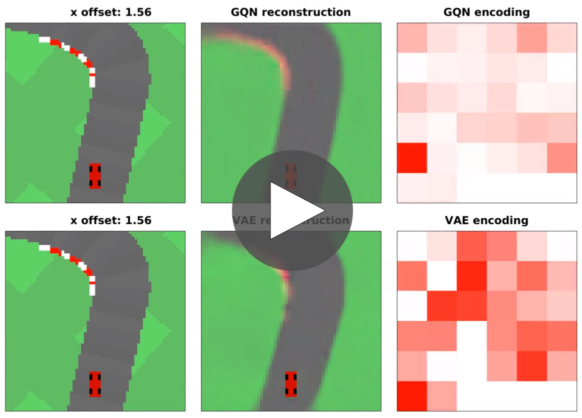


Video 4.2: GQN query rendering for the DeepMind Lab Maze

In conclusion, the adaptation of the GQN to the two new environments using relative coordinates and angles can be considered successful regarding its ability to encode and decode a variable number of input images and render a correct output depending on the query view.

4.2 Encoding

Normally, it is not possible to directly see the encoded representation that is generated by the VAE and GQN. Only through the used decoder can a visually perceivable image be reconstructed. However, it is possible to translate the value of each dimension of an encoded vector to a color and draw color intensities. This has been done in Video 4.3, where the left column represents the original observation, which is rotated or translated, the second column shows the decoded frame in case of the VAE and GQN, and the third column shows the actual 32 dimensional representation drawn in a 6×6 matrix. For the VAE, a range of $[-1, 1]$ is mapped to a red color value between 0 and 255 and for the GQN, the mapped range is $[0, 1]$.

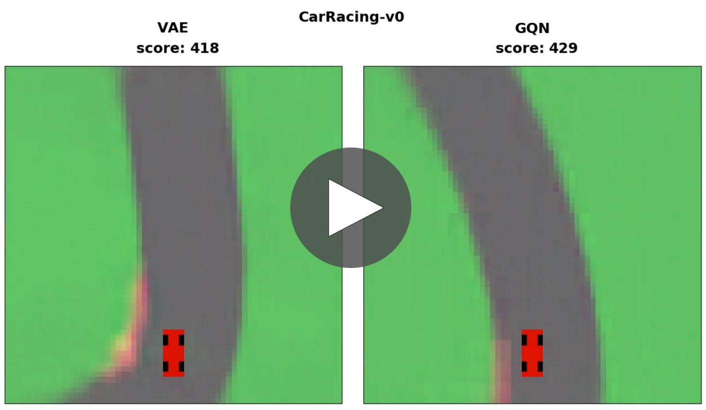


Video 4.3: CarRacing VAE and GQN encoding

In order to observe the encoding during translation or rotation of the input image, an offset has been applied to the car's position and yaw angle. As seen in Video 4.3 the encoding of the GQN is much more stable during rotational and positional transformations. The VAE encoding flickers considerably due to the fact that each encoded representation is randomly drawn from a distribution (represented by μ and σ).

In a direct side-by-side comparison of the best runs of models trained with a VAE and GQN vision model (Video 4.4), it is noticeable that the steering of the agent using the VAE model is more shaky compared to the GQN model.

This can be explained with the more unstable encoding as well (as seen in Video 4.3).



Video 4.4: CarRacing VAE and GQN comparison

In order to compare the actions of the agent trained with the VAE and GQN vision model, it is also possible to plot the intensity of each action taken during an episode, namely how much acceleration or deceleration and steering has been applied at each time step. Analogous to a real driver driving a car, it is preferable to perform more precise actions and do small corrections when necessary rather than to constantly hit the brake and the gas pedal and do very abrupt steering movements.

As seen in Figure 4.1, the actions of the agent based on the GQN model concentrate more around the center with little steering and mild acceleration. Moreover, there is an increased number of actions which combine steering left and braking. This makes sense considering that the car goes around the track counterclockwise and there are usually no right turns. Consequently, the agent has to brake and steer left before each turn. In contrast, the agent consisting of the VAE model and controller drives more aggressively with more acceleration and steering left and right. Due to the oversteering to the right, the agent then has to do also stronger corrective actions and as a result the shaky steering from Video 4.4 is observed.

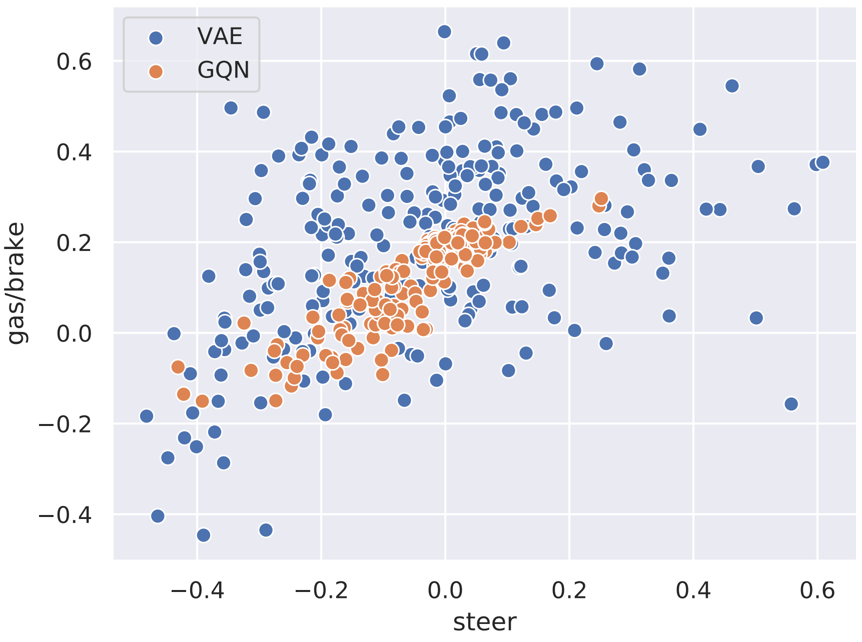
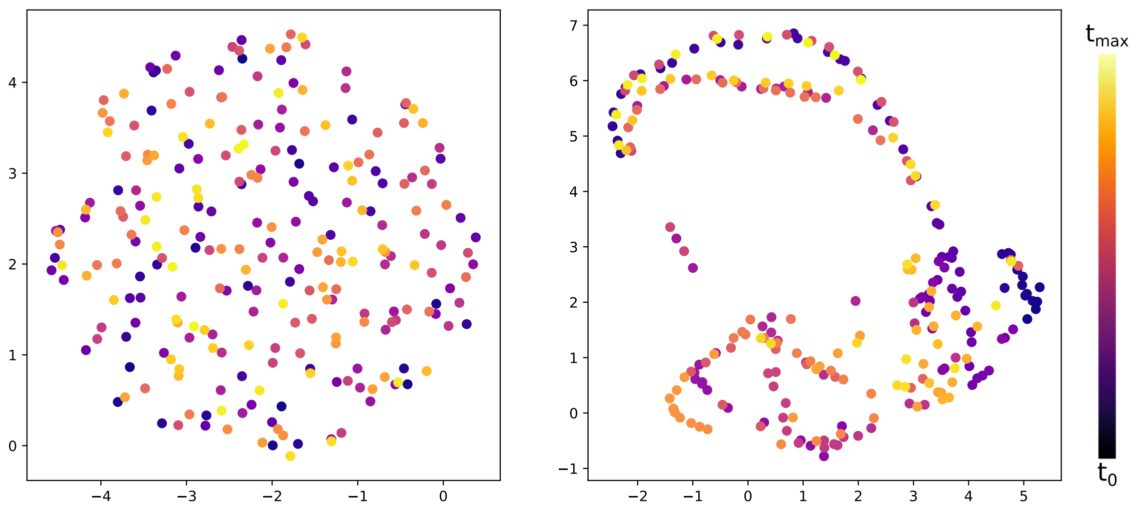


Figure 4.1: Comparison of the action space of VAE and GQN models

Another possibility of visualising the different found representations of the VAE and GQN models is represented in Figure 4.2. It depicts a projection of an episode (car going around the track) using Uniform Manifold Approximation and Projection (UMAP) [13]. UMAP is an algorithm that reduces high dimensional data into a 2D representation which allows to print the 32 dimensional vector z_t in a visually perceivable scatter plot. Each time step from 0 to t_{max} is mapped to a single point in the diagram. The color gradient in Figure 4.2 represents the current time step during the episode. Dark blue colors are at the beginning, followed by red colors and finally the episode (track) ends with the yellow colors.

The plot of an episode using the VAE model (Figure 4.2a) shows a cloud without any clear pattern. The time steps represented by the colors are distributed evenly across the graph, although some points are closer to each other and leave small white spaces in between. In contrast, the diagram on the right, which depicts an episode encoded with a GQN (Figure 4.2b) possesses much more structure and seems to form rings corresponding to different parts of the track. Dark blue colors indicating the beginning of an episode are predominant on the right and top part. Light yellow colors corresponding to the end of the episode are also more prevalent in the top area. Essentially, the most important aspects of the track like the curvature are mapped into a distorted representation.



(a) VAE (b) GQN

Figure 4.2: CarRacing visualisation of z_t with UMAP during an episode

Overall, it was demonstrated that the GQN model offers advantages by encoding the observed structure into the latent vector z_t that is to some extend translation and rotation invariant and therefore allows agents trained with this representation to take more precise actions. One disadvantage of using the GQN though, is the considerably higher training time to achieve visually comparable results to the VAE model. The implementation adapted for this thesis [15] needed to be trained about 10 times longer.

4.3 CarRacing Scores

In this section, the scores achieved in the CarRacing environments for agents consisting of a Vision (V) and Controller (C) model are discussed. For the V model the results using a GQN and VAE are compared and tested in two cases:

- C model with a single densely-connected layer
- C model with two layers (one hidden layer and an output layer)

Depending on the number of possible actions and the number of dimensions of z_t (n_z), the number of parameters n_θ the CMA-ES algorithm has to train changes. n_θ can be computed according to Equation 3.19. The results depicted here were generated with a size of 32 dimensions for z_t for the VAE model and 64 dimensions for the GQN model. The actual size of the latent vector is an adjustable hyperparameter and the chosen values were found to work well for the models, but do not dramatically impact the scores. In case of the VAE, the original described architecture by Ha and Schmidhuber used a 32 dimensional latent vector and this was adopted for the experiments run for this thesis. The GQN model seems to perform slightly better with a size of 64 dimensions rather than with 32 dimensions, but due to the stochasticity of the obtained results, no scores investigating this further are presented here. The scores in Table 4.1 were computed by evaluating the best found policy for each model through 100 trials and computing the mean and standard deviation (indicated after ±) of the rewards achieved.

Model	**Score**	**Parameter count** n_θ
GQN single input image	799 ± 153	160
VAE	**806 ± 174**	66
original VAE [8]	632 ± 251	99

Table 4.1: Results for single layer C Model

The VAE model attained a slightly better result than the GQN for the single layer V model (Table 4.1). Both GQN and VAE scores are much higher than the scores achieved by the original World Models paper and have a lower variance.

The higher scores for the VAE might be explainable by:

- The fact that 2 instead of 3 actions were used to train the model (accelerate and brake have been combined).
- A new action has only been performed on the environment every 4th step. The rest of the time, the last action was repeated, which made the steering more stable.
- The model's high sensitivity to initial model parameters. For this thesis, the agents were trained on an initial pool of completely random parameters. The best performing agent was chosen and its parameters formed the initial parameters of the CMA-ES algorithm. Depending on the initial parameters, it is possible that the CMA-ES algorithm converges into different local minima instead of the global minimum for cost $\bar{J}$.

The scores of the agents with a hidden layer consisting of 16 units show a significantly higher average for the agent trained with the GQN as V model (see Table 4.2). The agent trained with a VAE was not able to obtain results similar to the original paper and performs 150 points worse on average. However, this might be improved by adjusting different hyper-parameters (e.g. not repeating an action 4 times). Similarly to Table 4.1, the scores were computed by evaluating the final policy on 100 trials and computing the mean and standard deviation of the obtained cumulative rewards.

Additionally, it should be noted that training the C model with a hidden layer considerably increases the number of parameters needed and therefore the computational cost required to find an optimal policy.

Model	**Score**	**Parameter count** n_θ
GQN single input image	**859 ± 118**	1090
VAE	632 ± 201	562
original VAE [8]	788 ± 141	1443

Table 4.2: Results for agents using the hidden layer C model

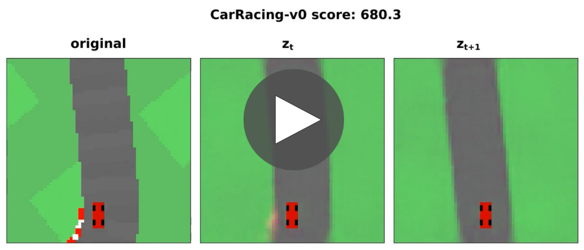
To conclude, the best score (859) for the agent trained using the V and C model was attained by using a hidden layer and the GQN model. The second best result (806) was obtained with a VAE single layer model. Contrary to the results from the original paper [8] the VAE model with a single layer performed better than the VAE model with an additional hidden layer.

4.4 CarRacing Results for the full Architecture (V, M and C)

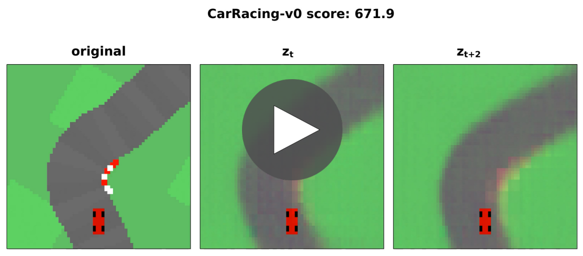
This section discusses results and scores obtained by using the full World Models architecture, which includes a vision (V) model, memory (M) model and controller (C) as defined in Figure 3.9.

The M model can be used to obtain a predicted latent vector $z_{t+\delta}$, which is an estimation of the future state of the environment. Video 4.5 visualizes this for the CarRacing environment with the right column being the prediction at time $t + \delta$ and the left and middle column representing the original observation and decoded latent vector at time t.

Due to the way the GQN representation is calculated, it is not possible to estimate a future state as a mixture of Gaussians. Therefore, the results in Video 4.5b were computed without a mixture of Gaussian distributions, and instead predict the future state directly without any random sampling.



(a) VAE with $\delta = 1$



(b) GQN with $\delta = 2$

Video 4.5: CarRacing with predicted future vector $z_{t+\delta}$

In most instances, the prediction for the VAE (Video 4.5a) is correct and moves slightly ahead of the frame at time step t. Although the prediction sometimes showed red stripes which indicate the beginning of a turn even if there isn't one.

The results for the GQN model (Video 4.5b), are in comparison more stable and also allow to look further into the future without the creation of unrealistic states. For this reason the prediction in Video 4.5b is visualized for z_{t+2}, as opposed to the estimation of z_{t+1} in case of the VAE model. The quality of predictions for the VAE model considerably worsens for forecasts further than one time step ahead.

The full World Model with a VAE or GQN as vision model was trained for 500 generations while repeating each step either four or two times, in order to reduce the computation time required to fully train the model. As depicted in Table 4.3, the mean scores with 4 repeated steps (computed over 100 trials) are higher for the GQN model in comparison to the VAE model, and have a lower standard-deviation. For the models, which repeated each step twice, the VAE achieved an average score of 909 and the GQN model obtained a slightly higher average score of 911, which is 5 points higher than the result from the original paper [8]. However, the original model has a lower standard-deviation, meaning it has less outliers of runs that are above or below the average reward.

Model	**Repeated steps**	**Score**
VAE	4	853 ± 106
GQN	4	859 ± 99
VAE	2	909 ± 60
GQN	2	$\mathbf{911} \pm 63$
original VAE [8]	1	906 ± 21

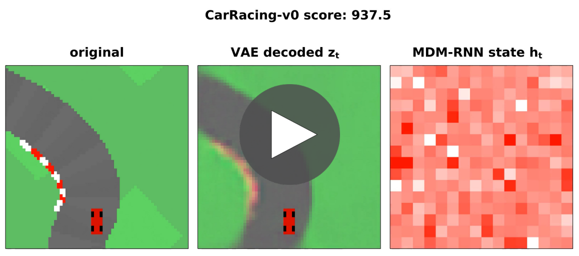
Table 4.3: Results for MDN-RNN + C Model

The results in Table 4.3 show better scores for models, which repeat an action twice instead of 4 times. However, training this model with less repeated steps comes at the cost of increased training times per generation, as seen in Table 4.4. The measured times have been computed using the full World Model architecture (V, M and C) with 12 environments running in parallel per agent and a population size of 16. As a result, actions for 192 environments over 250, 500 or 1000 steps (depending on the number of repeated steps) had to be computed.

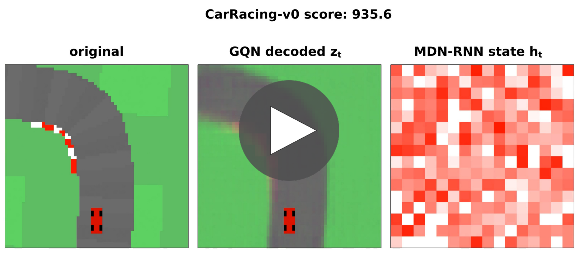
Model	**Repeated steps**	**Time in** s
VAE	4	22
VAE	2	37
GQN	2	45
VAE	1	68

Table 4.4: Execution times per generation for the CarRacing environment with $k = 12$ and $\gamma = 16$

The measurements in Table 4.4 were taken on an IBM Power9 node which executes 160 parallel threads and contains 4 NVIDIA V100 GPUs. With a population size of $\gamma = 16$, this means that 4 agents share a GPU to run the model inference.



(a) VAE



(b) GQN

Video 4.6: CarRacing trained with the hidden state h_t of the memory model

As seen in Video 4.6, the models trained with the memory model are both able to complete the track in 2/3 of the available 1000 time steps, which shows that using the predictive abilities of the M model allowed the models to drive faster without loosing control.

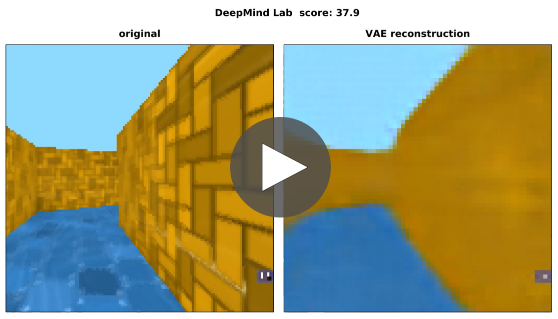
Video 4.6 also visualizes the hidden state h_t of the RNN that was concatenated with the current latent vector z_t and used as input to the C model. The actual 256 dimensional vector h_t is drawn in a 16×16 matrix. Its values in a range of $[-0.5, 0.5]$ are mapped to a red color value between 0 and 255.

As evident in Video 4.6a, for an agent trained with the VAE model, h_t changes some its values quite quickly during the course (which makes the corresponding square flicker), while other values remain mostly unchanged or change gradually. In contrast, if the agent was trained using the GQN model (Video 4.6b), the values of the RNNs hidden state barley change over the course of the track. Only during turns the red colors slightly change intensity. Interestingly, the in section 4.2 discussed visualisation of the encoded vector z_t (Video 4.3) seems to some extend to repeat itself, in terms of how quickly the values changes, in the hidden state h_t of the RNN.

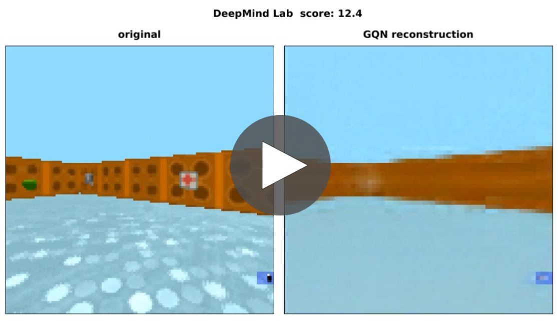
In summary, the modifications made to the M and C model described in section 3.2 and 3.3 allowed the trained models with two repeated actions to surpass the score of the original implementation from Ha and Schmidhuber [8] with the GQN model achieving the best scores. Furthermore, the overall improvement of using a memory model together with the vision model and controller in both cases for the VAE model and GQN model was demonstrated.

4.5 DeepMind Lab

In this section, the results of agents trained using the V and C model on the DeepMind Lab maze environment are discussed. In Video 4.7a, an episode consisting of 3000 time steps is shown for an agent using the VAE as vision model. In Video 4.7b, the same is demonstrated for an agent trained with the GQN model. In each video the left column shows the original received observation from the environment and the right column is the decoded image, reconstructed from the current latent vector z_t. For both models, the latent vector has 64 dimensions.



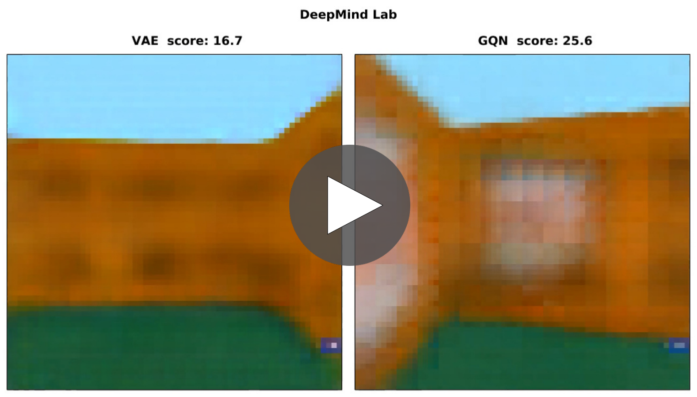
(a) DeepMind Lab VAE



(b) DeepMind Lab GQN

Video 4.7: DeepMind Lab VAE vs. GQN

The reconstruction of the latent vector z_t works sufficiently well for both VAE and GQN to perceive the layout of the environment. The color and shape of the walls and the floor are correctly reconstructed. Only less important details like fine textures are blurred and often not visible anymore. Moreover, both models were not able to reconstruct objects like the green apples in the environment. The ability to reconstruct details should increase if, instead of a 64 dimensional vector z_t, more dimensions are used to describe the scene.



Video 4.8: DeepMind Lab Maze VAE and GQN side by side

In the side-by-side comparison of the VAE and GQN model in Video 4.8, it is observable that the policy learned by the agent trained with the GQN allows it to stay away from the walls in most cases, whereas the policy of the agent trained with the VAE is mostly to hit walls and then turn left or right and walk towards the next wall.

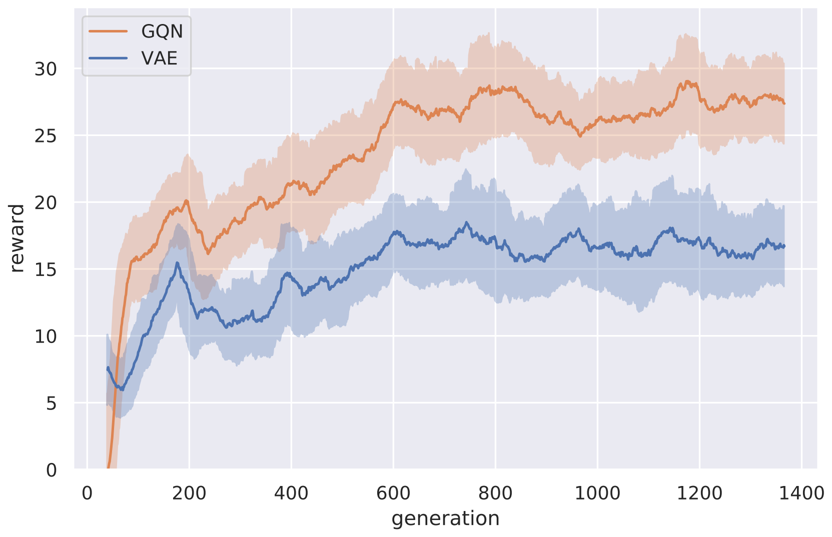


Figure 4.3: DeepMind Lab maximum reward obtained by VAE and GQN models

In Figure 4.3, the average reward obtained by the best performing agent over a training period of 1400 generations is depicted for the VAE and GQN used as vision model. The values are averaged over 40 generations for better readability. Apart from the very beginning, the agent trained using the GQN model consistently achieved higher maximum scores, which were about 10 points higher after 1400 generations than the scores achieved with the VAE model.

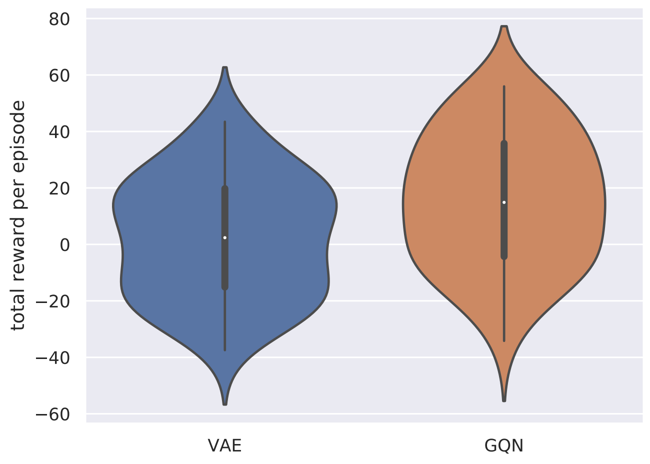
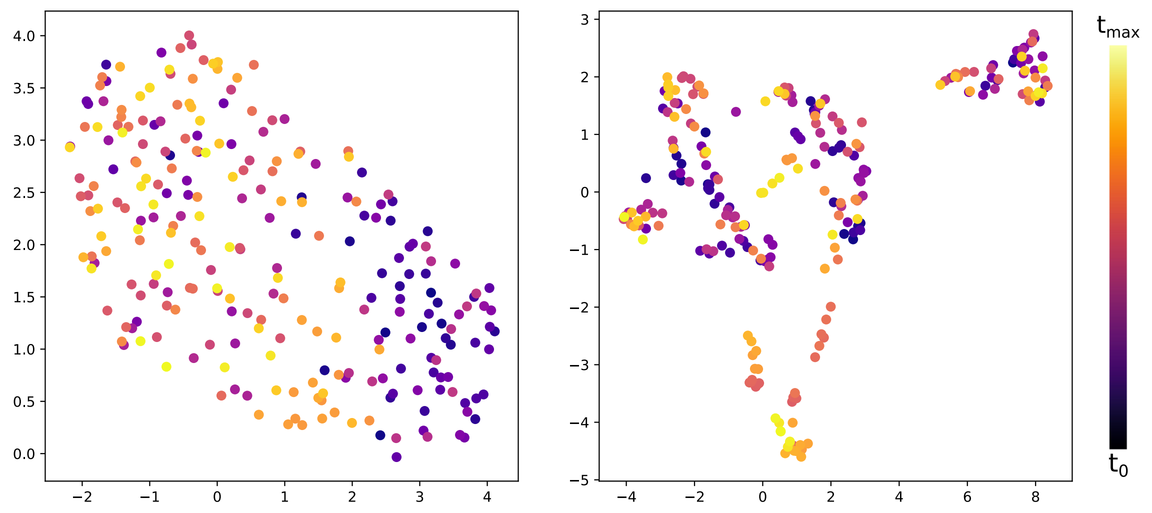


Figure 4.4: DeepMind Lab reward distribution for VAE and GQN models

The reward distribution of the trained agent models illustrated in Figure 4.4 shows that the scores of the GQN model have a higher range than the VAE model. However, the lowest scores obtained in both models are both close to -60. In general, the plot shows the high variability in scores received due to the random nature of the environment.

Similarly to Figure 4.2, the color gradient in Figure 4.5 represents the current time step during an episode. Dark blue colors are at the beginning of the episode, followed by red colors, and finally the episode ends with the yellow colors. The 64 dimensional vector z_t is reduced to a 2D point using the UMAP [13] algorithm.



(a) VAE (b) GQN

Figure 4.5: DeepMind Lab visualisation of z_t with UMAP during an episode

The plot of the agent trained using the VAE model results in a point cloud. Darker points (closer to the start) are more concentrated on the right side of the graph. In contrast, the plot for the GQN model shows much more structure with a denser center part and separate areas on the right and bottom, whereas later time steps seem to be predominantly in the bottom part. Similar to the results obtained from the CarRacing environment, it is evident that the GQN representation is more structured than the latent vector used by the VAE.

To summarize, the GQN model improved the ability of an agent navigating a 3D simulation compared to agents using the VAE as vision model and allowed it to explore a bigger area of the environment. As a result, it attained higher scores due to a more structured encoding of each observation.

Conclusions

In this master thesis, it was shown that the VAE used in the World Models architecture has some drawbacks and that a Generative Query Network (GQN) is a viable alternative for the models vision component, allowing an agent trained with those models to achieve higher scores in the CarRacing and DeepMind Lab environments. The advantages of including a memory model in the architecture were explored and it was illustrated that a combination of a modified MDN-RNN and GQN allows to obtain better results for the CarRacing environment than an agent consisting of a MDN-RNN and VAE model. The ability of a GQN to compute a structural representation that is invariant to changes in rotation and small changes in position was demonstrated. Furthermore, it was illustrated that the developed models are applicable to both, 2D and 3D environments without the need to treat them differently.

Additionally, it was demonstrated that a fast parallel training of agents can significantly reduce the time necessary to train the full World Model.

5.1 Contributions

- A novel approach of combining the World Models architecture with the GQN as Vision (V) model has been developed.
- The GQN was trained on new environments not discussed by Eslami et al. [4] which are the CarRacing environment and a custom DeepMind Lab maze environment.
- The existing World Models architecture was improved for the efficient use of parallel computing units by running several environments per agent during training.

- The Memory (M) model used in the World Models architecture was improved to produce more accurate predictions of future latent vectors by using the previous input action a_{t-1} at each time step and using several layers to model the Mixture Density Network that forms a part of the MDN-RNN
- The memory model has been modified and adopted to allow the use together with the GQN model in a way that does not require future vectors z_{t+1} to be modeled as a mixture of Gaussians.

5.2 Future Work

In this thesis, the VAE and GQN models were trained separately from the controller and memory models. This is practical for environments which allow sufficient sampling at random positions and angles. In a more complex changing environment like the real world, an agent would not be able to do this. Therefore, it could be investigated how to dynamically update the vision model while the agent explores new areas of an environment. To do this effectively for the GQN model, additional information like the position and view angle of the agent is required. These could either be directly supplied by the environment or estimated by another neural network. The advantage of estimating the agent's position is that this approach could be used in the real world where, apart from rather rough positioning systems like GPS (with accuracies ranging in the range of several meters), there is no precise positional data available. Given the current observation and previous actions of the agent, its current position could be estimated, similar to how a human is able to infer his or her position in e.g. a room.

Finally, investigating how to train an agent in an completely imagined environment (dream) is an interesting topic for future work. The authors of the World Models paper [8] tested this approach with a game environment called VizDoom. This game is fairly simple to emulate and they show that an agent solely trained in a dream achieves good results in the actual environment. However, they admit that this is only viable in a stochastic environment. The MDN-RNN they used as memory model is not able to produce a good deterministic sequence of states to emulate a more complex environment. For this reason, it is more difficult to create a realistic dream of the CarRacing task, because it requires consecutive observations to properly simulate the physics of the car and continue the track correctly.

References

[1] Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. Deepmind lab, 2016.

[2] Christopher M. Bishop. Mixture density networks. 1994.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[4] S. M. Ali Eslami, Danilo Jimenez Rezende, Frederic Besse, Fabio Viola, Ari S. Morcos, Marta Garnelo, Avraham Ruderman, Andrei A. Rusu, Ivo Danihelka, Karol Gregor, David P. Reichert, Lars Buesing, Theophane Weber, Oriol Vinyals, Dan Rosenbaum, Neil Rabinowitz, Helen King, Chloe Hillier, Matt Botvinick, Daan Wierstra, Koray Kavukcuoglu, and Demis Hassabis. Neural scene representation and rendering. *Science*, 360(6394):1204–1210, 2018. doi: 10.1126/science.aar6170.

[5] Kyle Genova, Forrester Cole, Aaron Maschinot, Aaron Sarna, Daniel Vlasic, and William T. Freeman. Unsupervised training for 3d morphable model regression. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[6] Karol Gregor, Frederic Besse, Danilo Jimenez Rezende, Ivo Danihelka, and Daan Wierstra. Towards conceptual compression, 2016.

[7] David Ha and Douglas Eck. A neural representation of sketch drawings, 2017.

[8] David Ha and Jürgen Schmidhuber. World models. *CoRR*, 2018.

[9] N. Hansen, S.D. Muller, and P. Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES). *Evolutionary Computation*, 11(1):1–18, 2003.

[10] Nikolaus Hansen. The cma evolution strategy: A tutorial, 2016.

[11] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[12] Diederik P Kingma and Max Welling. Auto-Encoding Variational Bayes. *arXiv e-prints*, art. arXiv:1312.6114, Dec 2013.

[13] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction, 2018.

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

[15] O. Groth. tf-gqn. https://github.com/ogroth/tf-gqn, 2019.

[16] OpenAI. CarRacing-v0 environment. https://gym.openai.com/envs/CarRacing-v0/.

[17] CORPORATE OpenGL Architecture ReviewBoard. *OpenGL Reference Manual: The Official Reference Document for OpenGL, Release 1*. Addison-Wesley Longman Publishing Co., Inc., USA, 1992.

[18] Juergen Schmidhuber. On learning to think: Algorithmic information theory for novel combinations of reinforcement learning controllers and recurrent neural world models, 2015.

[19] Guido Van Rossum and Fred L Drake Jr. *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.