CHAPTER 12

World Models

Chapter Goals

In this chapter you will:

- Walk through the basics of reinforcement learning (RL).
- Understand how generative modeling can be used within a *world model* approach to RL.
- See how to train a variational autoencoder (VAE) to capture environment observations in a low-dimensional latent space.
- Walk through the training process of a mixture density network–recurrent neural network (MDN-RNN) that predicts the latent variable.
- Use the covariance matrix adaptation evolution strategy (CMA-ES) to train a controller that can take intelligent actions in the environment.
- Understand how the trained MDN-RNN can itself be used as an environment, allowing the agent to train the controller within its own hallucinated dreams, rather than the real environment.

This chapter introduces one of the most interesting applications of generative models in recent years, namely their use within so-called world models.

Introduction

In March 2018, David Ha and Jürgen Schmidhuber published their "World Models" paper.[1] The paper showed how it is possible to train a model that can learn how to perform a particular task through experimentation within its own generated dream environment, rather than inside the real environment. It is an excellent example of

how generative modeling can be used to solve practical problems, when applied alongside other machine learning techniques such as reinforcement learning.

A key component of the architecture is a generative model that can construct a probability distribution for the next possible state, given the current state and action. Having built up an understanding of the underlying physics of the environment through random movements, the model is then able to train itself from scratch on a new task, entirely within its own internal representation of the environment. This approach led to world-best scores for both of the tasks on which it was tested.

In this chapter we will explore the model from the paper in detail, with particular focus on a task that requires the agent to learn how to drive a car around a virtual racetrack as fast as possible. While we will be using a 2D computer simulation as our environment, the same technique could also be applied to real-world scenarios where testing strategies in the live environment is expensive or infeasible.

In this chapter we will reference the excellent TensorFlow implementation of the "World Models" paper available publicly on GitHub, which I encourage you to clone and run yourself!

Before we start exploring the model, we need to take a closer look at the concept of reinforcement learning.

Reinforcement Learning

Reinforcement learning can be defined as follows:

> Reinforcement learning (RL) is a field of machine learning that aims to train an agent to perform optimally within a given environment, with respect to a particular goal.

While both discriminative modeling and generative modeling aim to minimize a loss function over a dataset of observations, reinforcement learning aims to maximize the long-term reward of an agent in a given environment. It is often described as one of the three major branches of machine learning, alongside *supervised learning* (predicting using labeled data) and *unsupervised learning* (learning structure from unlabeled data).

Let's first introduce some key terminology related to reinforcement learning:

Environment
: The world in which the agent operates. It defines the set of rules that govern the game state update process and reward allocation, given the agent's previous action and current game state. For example, if we were teaching a reinforcement learning algorithm to play chess, the environment would consist of the rules that

govern how a given action (e.g., the pawn move `e2e4`) affects the next game state (the new positions of the pieces on the board) and would also specify how to assess if a given position is checkmate and allocate the winning player a reward of 1 after the winning move.

Agent
: The entity that takes actions in the environment.

Game state
: The data that represents a particular situation that the agent may encounter (also just called a *state*). For example, a particular chessboard configuration with accompanying game information such as which player will make the next move.

Action
: A feasible move that an agent can make.

Reward
: The value given back to the agent by the environment after an action has been taken. The agent aims to maximize the long-term sum of its rewards. For example, in a game of chess, checkmating the opponent's king has a reward of 1 and every other move has a reward of 0. Other games have rewards constantly awarded throughout the episode (e.g., points in a game of *Space Invaders*).

Episode
: One run of an agent in the environment; this is also called a *rollout*.

Timestep
: For a discrete event environment, all states, actions, and rewards are subscripted to show their value at timestep t.

The relationship between these concepts is shown in Figure 12-1.

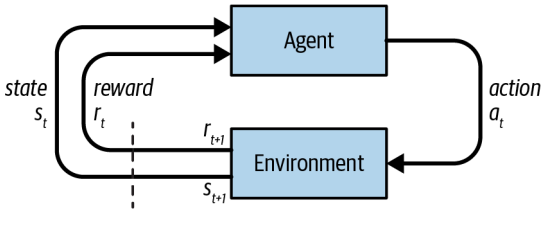

Figure 12-1. Reinforcement learning diagram

The environment is first initialized with a current game state, s_0. At timestep t, the agent receives the current game state s_t and uses this to decide on its next best action a_t, which it then performs. Given this action, the environment then calculates the next state s_{t+1} and reward r_{t+1} and passes these back to the agent, for the cycle to

begin again. The cycle continues until the end criterion of the episode is met (e.g., a given number of timesteps elapse or the agent wins/loses).

How can we design an agent to maximize the sum of rewards in a given environment? We could build an agent that contains a set of rules for how to respond to any given game state. However, this quickly becomes infeasible as the environment becomes more complex and doesn't ever allow us to build an agent that has superhuman ability in a particular task, as we are hardcoding the rules. Reinforcement learning involves creating an agent that can learn optimal strategies by itself in complex environments through repeated play.

Let's now take a look at the `CarRacing` environment that simulates a car driving around a track.

The CarRacing Environment

`CarRacing` is an environment that is available through the Gymnasium package. Gymnasium is a Python library for developing reinforcement learning algorithms that contains several classic reinforcement learning environments, such as `CartPole` and `Pong`, as well as environments that present more complex challenges, such as training an agent to walk on uneven terrain or win an Atari game.

Gymnasium

Gymnasium is a maintained fork of OpenAI's Gym library—since 2021, further development of Gym has shifted to Gymnasium. In this book, we therefore refer to Gymnasium environments as Gym environments.

All of the environments provide a *step* method through which you can submit a given action; the environment will return the next state and the reward. By repeatedly calling the step method with the actions chosen by the agent, you can play out an episode in the environment. There is also a *reset* method for returning the environment to its initial state and a *render* method that allows you to watch your agent perform in a given environment. This is useful for debugging and finding areas where your agent could improve.

Let's see how the game state, action, reward, and episode are defined for the `CarRacing` environment:

Game state
: A 64 × 64–pixel RGB image depicting an overhead view of the track and car.

Action
: A set of three values: the steering direction (–1 to 1), acceleration (0 to 1), and braking (0 to 1). The agent must set all three values at each timestep.

Reward
: A negative penalty of –0.1 for each timestep taken and a positive reward of 1,000/*N* if a new track tile is visited, where *N* is the total number of tiles that make up the track.

Episode
: The episode ends when the car completes the track or drives off the edge of the environment, or when 3,000 timesteps have elapsed.

These concepts are shown on a graphical representation of a game state in Figure 12-2.

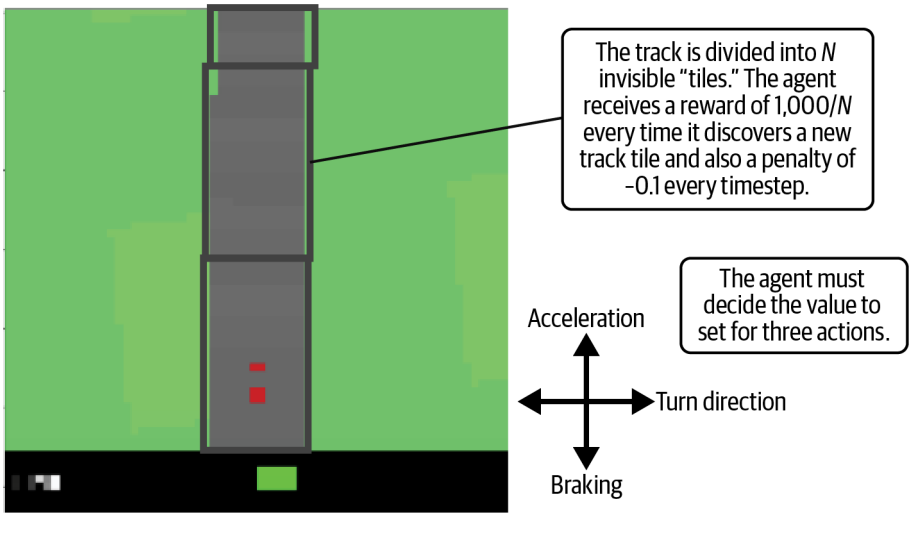

Figure 12-2. A graphical representation of one game state in the `CarRacing` environment

Perspective

We should imagine the agent floating above the track and controlling the car from a bird's-eye view, rather than viewing the track from the driver's perspective.

World Model Overview

We'll now cover a high-level overview of the entire world model architecture and training process, before diving into each component in more detail.

Architecture

The solution consists of three distinct parts, as shown in Figure 12-3, that are trained separately:

V
: A variational autoencoder (VAE)

M
: A recurrent neural network with a mixture density network (MDN-RNN)

C
: A controller

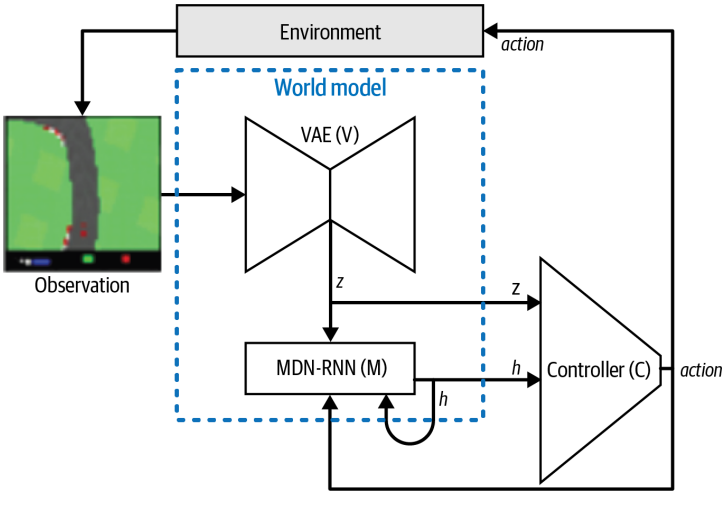

Figure 12-3. World model architecture diagram

The VAE

When you make decisions while driving, you don't actively analyze every single pixel in your view—instead, you condense the visual information into a smaller number of latent entities, such as the straightness of the road, upcoming bends, and your position relative to the road, to inform your next action.

We saw in Chapter 3 how a VAE can take a high-dimensional input image and condense it into a latent random variable that approximately follows a standard Gaussian

distribution, through minimization of the reconstruction error and KL divergence. This ensures that the latent space is continuous and that we are able to easily sample from it to generate meaningful new observations.

In the car racing example, the VAE condenses the 64 × 64 × 3 (RGB) input image into a 32-dimensional normally distributed random variable, parameterized by two variables, `mu` and `logvar`. Here, `logvar` is the logarithm of the variance of the distribution. We can sample from this distribution to produce a latent vector `z` that represents the current state. This is passed on to the next part of the network, the MDN-RNN.

The MDN-RNN

As you drive, each subsequent observation isn't a complete surprise to you. If the current observation suggests a left turn in the road ahead and you turn the wheel to the left, you expect the next observation to show that you are still in line with the road.

If you didn't have this ability, your car would probably snake all over the road as you wouldn't be able to see that a slight deviation from the center is going to be worse in the next timestep unless you do something about it now.

This forward thinking is the job of the MDN-RNN, a network that tries to predict the distribution of the next latent state based on the previous latent state and the previous action.

Specifically, the MDN-RNN is an LSTM layer with 256 hidden units followed by a mixture density network (MDN) output layer that allows for the fact that the next latent state could actually be drawn from any one of several normal distributions.

The same technique was applied by one of the authors of the "World Models" paper, David Ha, to a handwriting generation task, as shown in Figure 12-4, to describe the fact that the next pen point could land in any one of the distinct red areas.

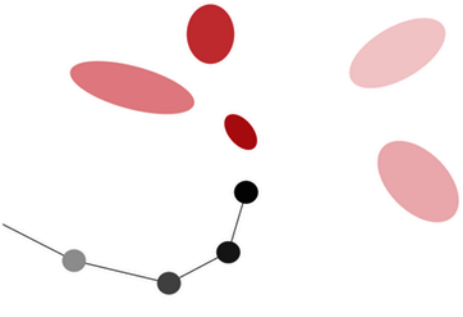

Figure 12-4. MDN for handwriting generation

In the car racing example, we allow for each element of the next observed latent state to be drawn from any one of five normal distributions.

The controller

Until this point, we haven't mentioned anything about choosing an action. That responsibility lies with the controller. The controller is a densely connected neural network, where the input is a concatenation of `z` (the current latent state sampled from the distribution encoded by the VAE) and the hidden state of the RNN. The three output neurons correspond to the three actions (turn, accelerate, brake) and are scaled to fall in the appropriate ranges.

The controller is trained using reinforcement learning as there is no training dataset that will tell us that a certain action is *good* and another is *bad*. Instead, the agent discovers this for itself through repeated experimentation.

As we shall see later in the chapter, the crux of the "World Models" paper is that it demonstrates how this reinforcement learning can take place within the agent's own generative model of the environment, rather than the Gym environment. In other words, it takes place in the agent's *hallucinated* version of how the environment behaves, rather than the real thing.

To understand the different roles of the three components and how they work together, we can imagine a dialogue between them:

> *VAE* (looking at latest $64 \times 64 \times 3$ observation): This looks like a straight road, with a slight left bend approaching, with the car facing in the direction of the road (`z`).
>
> *RNN*: Based on that description (`z`) and the fact that the controller chose to accelerate hard at the last timestep (`action`), I will update my hidden state (`h`) so that the next observation is predicted to still be a straight road, but with slightly more left turn in view.
>
> *Controller*: Based on the description from the VAE (`z`) and the current hidden state from the RNN (`h`), my neural network outputs `[0.34, 0.8, 0]` as the next action.

The action from the controller is then passed to the environment, which returns an updated observation, and the cycle begins again.

Training

The training process consists of five steps, run in sequence, which are outlined here:

1. Collect random rollout data. Here, the agent does not care about the given task, but instead simply explores the environment using random actions. Multiple episodes are simulated and the observed states, actions, and rewards at each timestep are stored. The idea is to build up a dataset of how the physics of the environment works, which the VAE can then learn from to capture the states efficiently as latent vectors. The MDN-RNN can then subsequently learn how the latent vectors evolve over time.

2. Train the VAE. Using the randomly collected data, we train a VAE on the observation images.
3. Collect data to train the MDN-RNN. Once we have a trained VAE, we use it to encode each of the collected observations into `mu` and `logvar` vectors, which are saved alongside the current action and reward.
4. Train the MDN-RNN. We take batches of episodes and load the corresponding `mu`, `logvar`, `action`, and `reward` variables at each timestep that were generated in step 3. We then sample a `z` vector from the `mu` and `logvar` vectors. Given the current `z` vector, `action`, and `reward`, the MDN-RNN is then trained to predict the subsequent `z` vector and `reward`.
5. Train the controller. With a trained VAE and RNN, we can now train the controller to output an action given the current `z` and hidden state, `h`, of the RNN. The controller uses an evolutionary algorithm, CMA-ES, as its optimizer. The algorithm rewards matrix weightings that generate actions that lead to overall high scores on the task, so that future generations are also likely to inherit this desired behavior.

Let's now take look at each of these steps in more detail.

Collecting Random Rollout Data

The first step is to collect rollout data from the environment, using an agent taking random actions. This may seem strange, given we ultimately want our agent to learn how to take intelligent actions, but this step will provide the data that the agent will use to learn how the world operates and how its actions (albeit random at first) influence subsequent observations.

We can capture multiple episodes in parallel by spinning up multiple Python processes, each running a separate instance of the environment. Each process will run on a separate core, so if your machine has lots of cores you can collect data much faster than if you only have a few cores.

The hyperparameters used by this step are as follows:

`parallel_processes`
: The number of parallel processes to run (e.g., `8` if your machine has ≥8 cores)

`max_trials`
: How many episodes each process should run in total (e.g., `125`, so 8 processes would create 1,000 episodes overall)

`max_frames`
: The maximum number of timesteps per episode (e.g., `300`)

Figure 12-5 shows an excerpt from frames 40 to 59 of one episode, as the car approaches a corner, alongside the randomly chosen action and reward. Note how the reward changes to 3.22 as the car rolls over new track tiles but is otherwise –0.1.

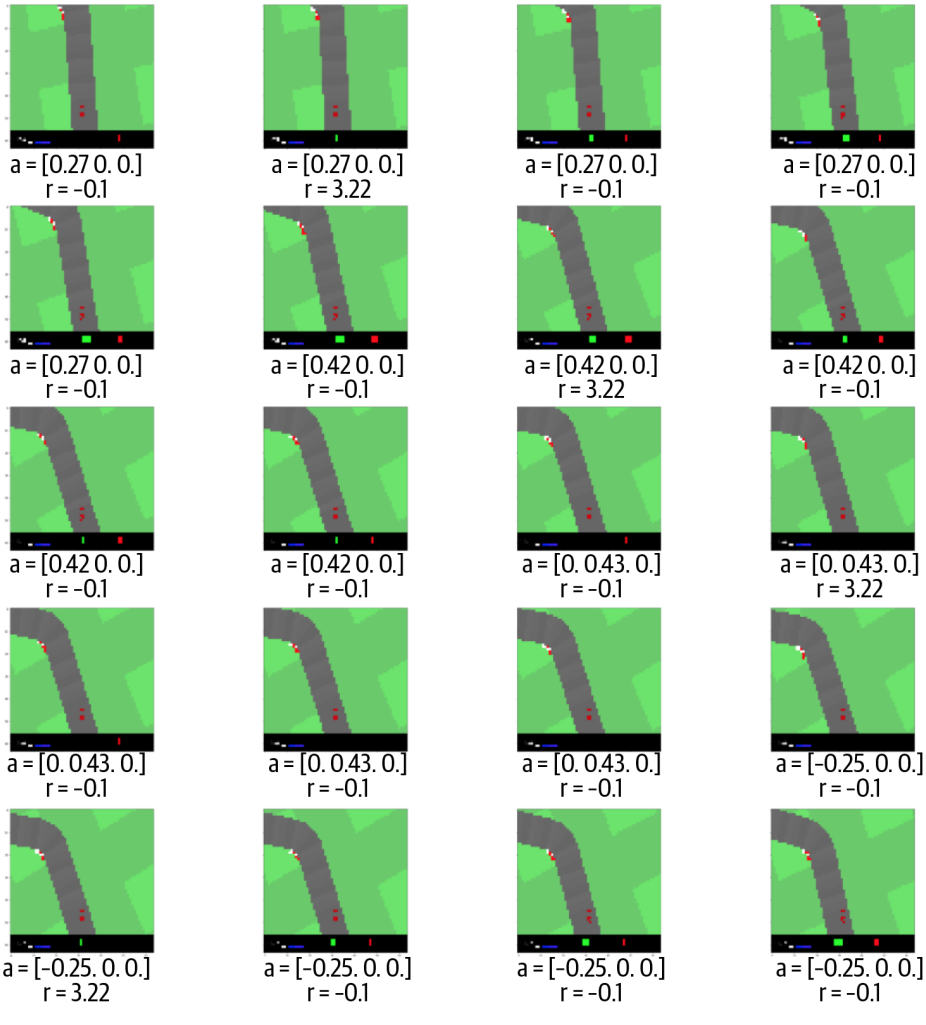

Figure 12-5. Frames 40 to 59 of one episode

Training the VAE

We now build a generative model (a VAE) on this collected data. Remember, the aim of the VAE is to allow us to collapse one 64 × 64 × 3 image into a normally distributed random variable `z`, whose distribution is parameterized by two vectors, `mu` and

`logvar`. Each of these vectors is of length 32. The hyperparameters of this step are as follows:

`vae_batch_size`
: The batch size to use when training the VAE (how many observations per batch) (e.g., `100`)

`z_size`
: The length of latent `z` vector (and therefore `mu` and `logvar` variables) (e.g., `32`)

`vae_num_epoch`
: The number of training epochs (e.g., `10`)

The VAE Architecture

As we have seen previously, Keras allows us to not only define the VAE model that will be trained end-to-end, but also additional submodels that define the encoder and decoder of the trained network separately. These will be useful when we want to encode a specific image or decode a given `z` vector, for example. We'll define the VAE model and three submodels, as follows:

`vae`
: This is the end-to-end VAE that is trained. It accepts a 64 × 64 × 3 image as input and outputs a reconstructed 64 × 64 × 3 image.

`encode_mu_logvar`
: This accepts a 64 × 64 × 3 image as input and outputs the `mu` and `logvar` vectors corresponding to this input. Running the same input image through this model multiple times will produce the same `mu` and `logvar` vectors each time.

`encode`
: This accepts a 64 × 64 × 3 image as input and outputs a sampled `z` vector. Running the same input image through this model multiple times will produce a different `z` vector each time, using the calculated `mu` and `logvar` values to define the sampling distribution.

`decode`
: This accepts a `z` vector as input and returns the reconstructed 64 × 64 × 3 image.

A diagram of the model and submodels is shown in Figure 12-6.

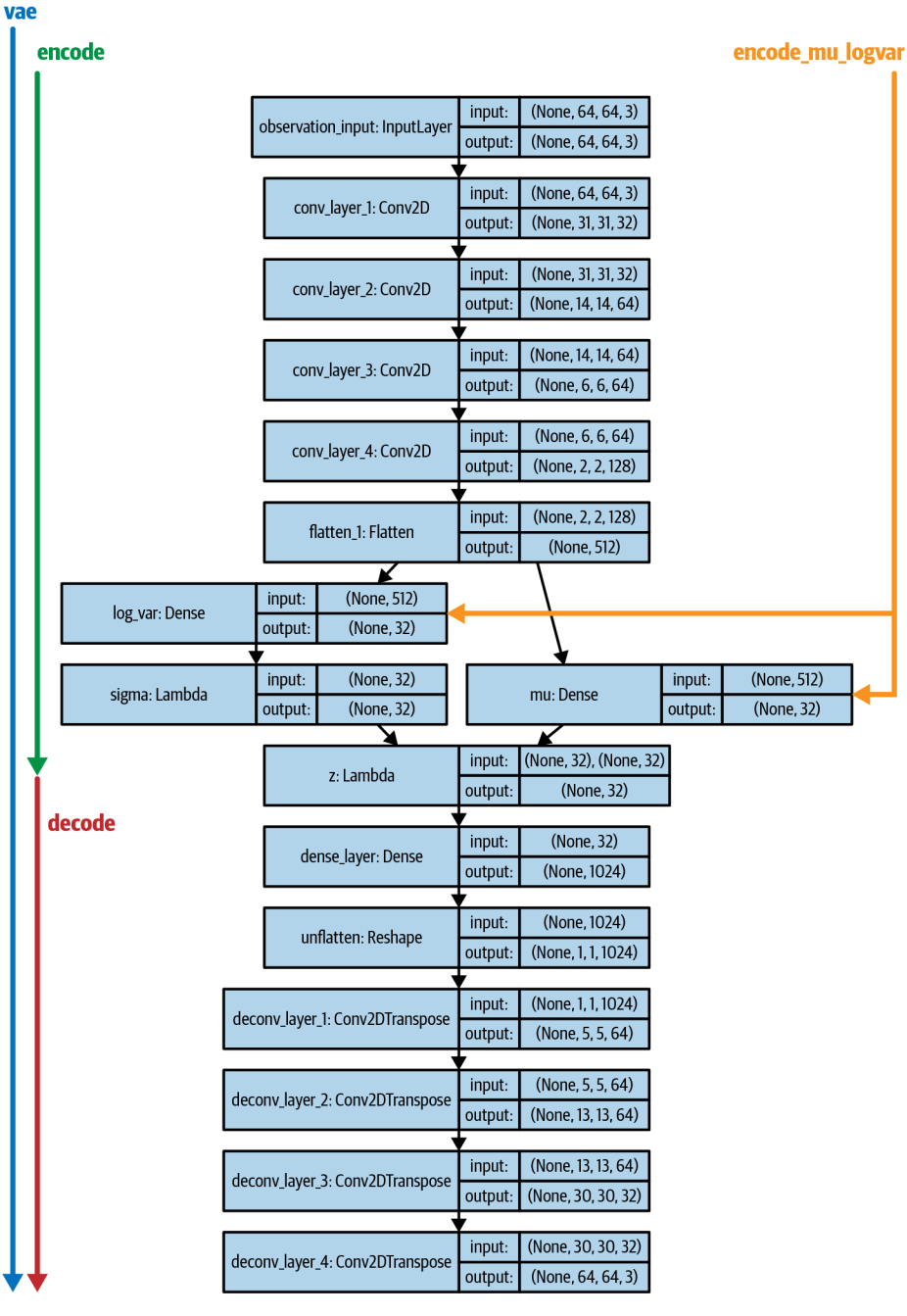

Figure 12-6. The VAE architecture from the "World Models" paper

Exploring the VAE

We'll now take a look at the output from the VAE and each submodel and then see how the VAE can be used to generate completely new track observations.

The VAE model

If we feed the VAE with an observation, it is able to accurately reconstruct the original image, as shown in Figure 12-7. This is useful to visually check that the VAE is working correctly.

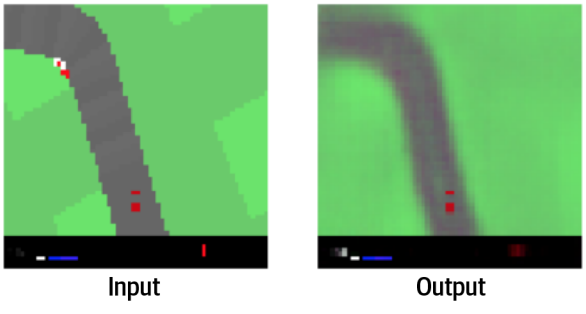

Figure 12-7. The input and output from the VAE model

The encoder models

If we feed the `encode_mu_logvar` model with an observation, the output is the generated `mu` and `logvar` vectors describing a multivariate normal distribution. The `encode` model goes one step further by sampling a particular `z` vector from this distribution. The diagram showing the output from the two encoder models is shown in Figure 12-8.

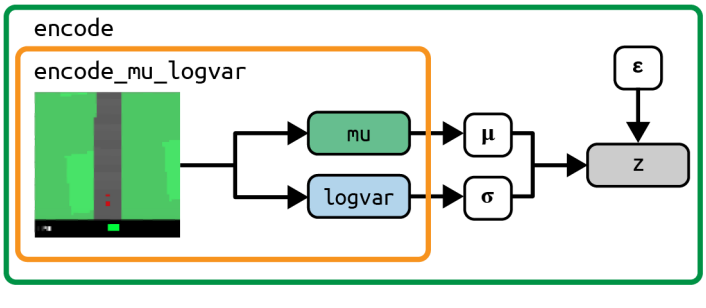

Figure 12-8. The output from the encoder models

The latent variable `z` is sampled from the Gaussian defined by `mu` and `logvar` by sampling from a standard Gaussian and then scaling and shifting the sampled vector (Example 12-1).

Example 12-1. Sampling `z` from the multivariate normal distribution defined by `mu` and `logvar`

```
eps = tf.random_normal(shape=tf.shape(mu))
sigma = tf.exp(logvar * 0.5)
z = mu + eps * sigma
```

The decoder model

The `decode` model accepts a `z` vector as input and reconstructs the original image. In Figure 12-9 we linearly interpolate two of the dimensions of `z` to show how each dimension appears to encode a particular aspect of the track—in this example `z[4]` controls the immediate left/right direction of the track nearest the car and `z[7]` controls the sharpness of the approaching left turn.

This shows that the latent space that the VAE has learned is continuous and can be used to generate new track segments that have never before been observed by the agent.

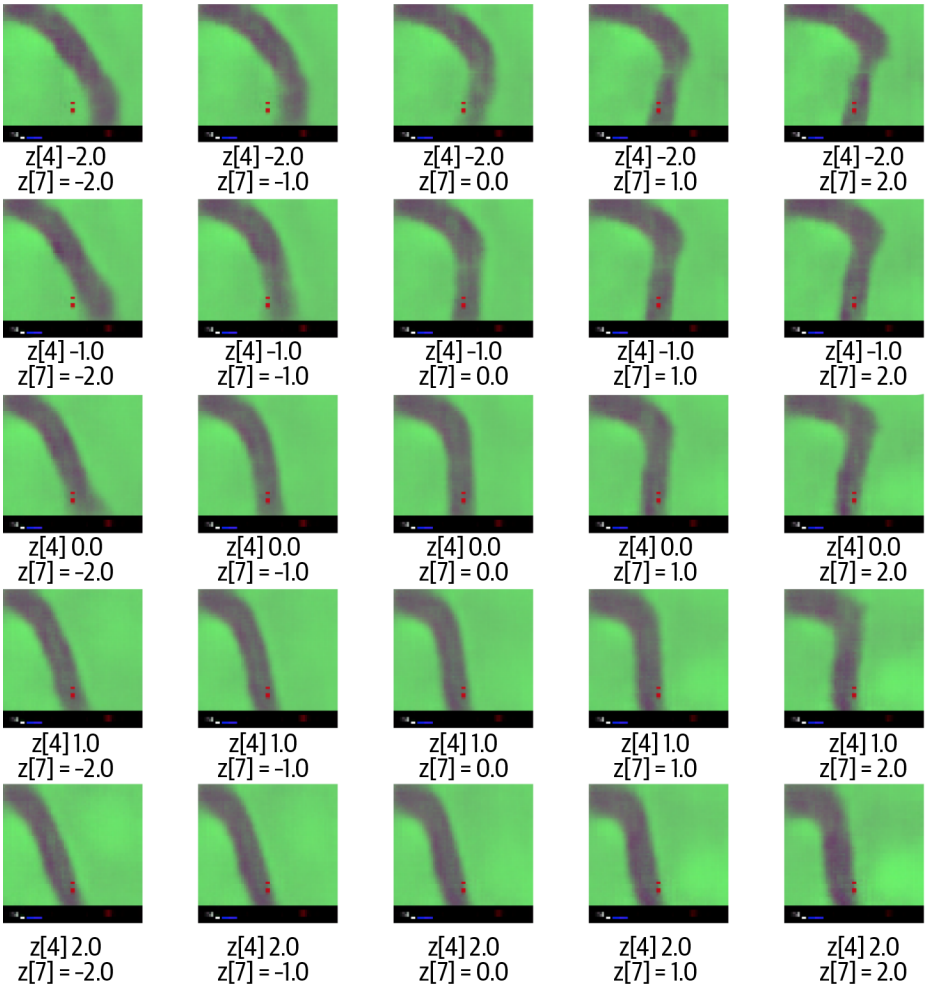

Figure 12-9. A linear interpolation of two dimensions of z

Collecting Data to Train the MDN-RNN

Now that we have a trained VAE, we can use this to generate training data for our MDN-RNN.

In this step, we pass all of the random rollout observations through the `encode_mu_logvar` model and store the `mu` and `logvar` vectors corresponding to each observation. This encoded data, along with the already collected `action`, `reward`, and `done` variables, will be used to train the MDN-RNN. This process is shown in Figure 12-10.

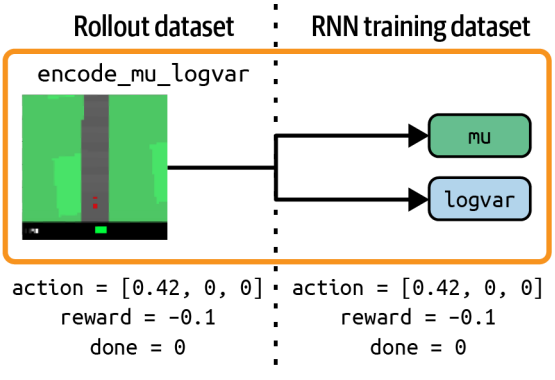

Figure 12-10. Creating the MDN-RNN training dataset

Training the MDN-RNN

We can now train the MDN-RNN to predict the distribution of the next `z` vector and reward one timestep ahead into the future, given the current `z` vector, current action, and previous reward. We can then use the internal hidden state of the RNN (which can be thought of as the model's current understanding of the environment dynamics) as part of the input into the controller, which will ultimately decide on the best next action to take.

The hyperparameters of this step of the process are as follows:

`rnn_batch_size`
: The batch size to use when training the MDN-RNN (how many sequences per batch) (e.g., `100`)

`rnn_num_steps`
: The total number of training iterations (e.g., `4000`)

The MDN-RNN Architecture

The architecture of the MDN-RNN is shown in Figure 12-11.

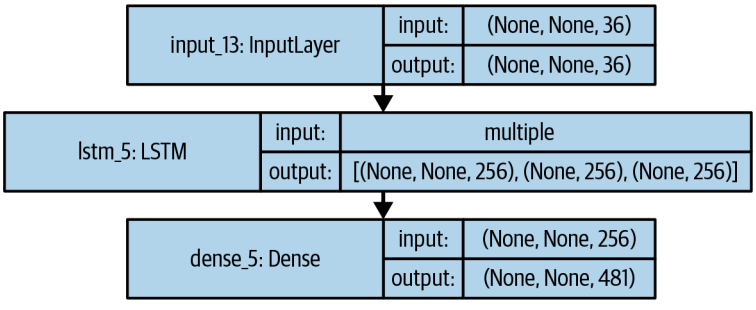

Figure 12-11. The MDN-RNN architecture

The MDN-RNN consists of an LSTM layer (the RNN), followed by a densely connected layer (the MDN) that transforms the hidden state of the LSTM into the parameters of a mixture distribution. Let's walk through the network step by step.

The input to the LSTM layer is a vector of length 36—a concatenation of the encoded `z` vector (length 32) from the VAE, the current action (length 3), and the previous reward (length 1).

The output from the LSTM layer is a vector of length 256—one value for each LSTM cell in the layer. This is passed to the MDN, which is just a densely connected layer that transforms the vector of length 256 into a vector of length 481.

Why 481? Figure 12-12 explains the composition of the output from the MDN-RNN. The aim of a mixture density network is to model the fact that our next `z` could be drawn from one of several possible distributions with a certain probability. In the car racing example, we choose five normal distributions. How many parameters do we need to define these distributions? For each of the 5 mixtures, we need a `mu` and a `logvar` (to define the distribution) and a log-probability of this mixture being chosen (`logpi`), for each of the 32 dimensions of `z`. This makes $5 \times 3 \times 32 = 480$ parameters. The one extra parameter is for the reward prediction.

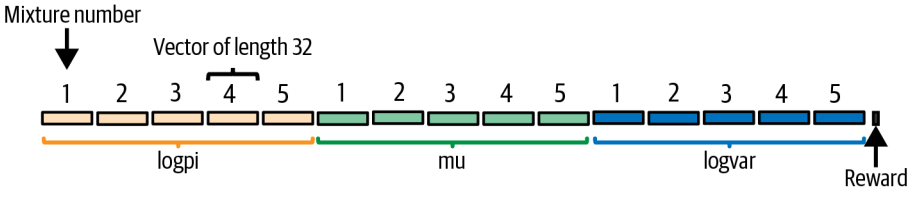

Figure 12-12. The output from the mixture density network

Sampling from the MDN-RNN

We can sample from the MDN output to generate a prediction for the next `z` and reward at the following timestep, through the following process:

1. Split the 481-dimensional output vector into the 3 variables (`logpi`, `mu`, `logvar`) and the reward value.
2. Exponentiate and scale `logpi` so that it can be interpreted as 32 probability distributions over the 5 mixture indices.
3. For each of the 32 dimensions of `z`, sample from the distributions created from `logpi` (i.e., choose which of the 5 distributions should be used for each dimension of `z`).
4. Fetch the corresponding values of `mu` and `logvar` for this distribution.
5. Sample a value for each dimension of `z` from the normal distribution parameterized by the chosen parameters of `mu` and `logvar` for this dimension.

The loss function for the MDN-RNN is the sum of the `z` vector reconstruction loss and the reward loss. The `z` vector reconstruction loss is the negative log-likelihood of the distribution predicted by the MDN-RNN, given the true value of `z`, and the reward loss is the mean squared error between the predicted reward and the true reward.

Training the Controller

The final step is to train the controller (the network that outputs the chosen action) using an evolutionary algorithm called the covariance matrix adaptation evolution strategy (CMA-ES).

The hyperparameters of this step of the process are as follows:

`controller_num_worker`
: The number of workers that will test solutions in parallel

`controller_num_worker_trial`
: The number of solutions that each worker will be given to test at each generation

`controller_num_episode`
: The number of episodes that each solution will be tested against to calculate the average reward

`controller_eval_steps`
: The number of generations between evaluations of the current best parameter set

The Controller Architecture

The architecture of the controller is very simple. It is a densely connected neural network with no hidden layers. It connects the input vector directly to the action vector.

The input vector is a concatenation of the current `z` vector (length 32) and the current hidden state of the LSTM (length 256), giving a vector of length 288. Since we are connecting each input unit directly to the 3 output action units, the total number of weights to tune is $288 \times 3 = 864$, plus 3 bias weights, giving 867 in total.

How should we train this network? Notice that this is not a supervised learning problem—we are not trying to *predict* the correct action. There is no training set of correct actions, as we do not know what the optimal action is for a given state of the environment. This is what distinguishes this as a reinforcement learning problem. We need the agent to discover the optimal values for the weights itself by experimenting within the environment and updating its weights based on received feedback.

Evolutionary strategies are a popular choice for solving reinforcement learning problems, due to their simplicity, efficiency, and scalability. We shall use one particular strategy, known as CMA-ES.

CMA-ES

Evolutionary strategies generally adhere to the following process:

1. Create a *population* of agents and randomly initialize the parameters to be optimized for each agent.
2. Loop over the following:
    a. Evaluate each agent in the environment, returning the average reward over multiple episodes.
    b. Breed the agents with the best scores to create new members of the population.
    c. Add randomness to the parameters of the new members.
    d. Update the population pool by adding the newly created agents and removing poorly performing agents.

This is similar to the process through which animals evolve in nature—hence the name *evolutionary* strategies. "Breeding" in this context simply means combining the existing best-scoring agents such that the next generation are more likely to produce high-quality results, similar to their parents. As with all reinforcement learning solutions, there is a balance to be found between greedily searching for locally optimal solutions and exploring unknown areas of the parameter space for potentially better

solutions. This is why it is important to add randomness to the population, to ensure we are not too narrow in our search field.

CMA-ES is just one form of evolutionary strategy. In short, it works by maintaining a normal distribution from which it can sample the parameters of new agents. At each generation, it updates the mean of the distribution to maximize the likelihood of sampling the high-scoring agents from the previous timestep. At the same time, it updates the covariance matrix of the distribution to maximize the likelihood of sampling the high-scoring agents, given the previous mean. It can be thought of as a form of naturally arising gradient descent, but with the added benefit that it is derivative-free, meaning that we do not need to calculate or estimate costly gradients.

One generation of the algorithm demonstrated on a toy example is shown in Figure 12-13. Here we are trying to find the minimum point of a highly nonlinear function in two dimensions—the value of the function in the red/black areas of the image is greater than the value of the function in the white/yellow parts of the image.

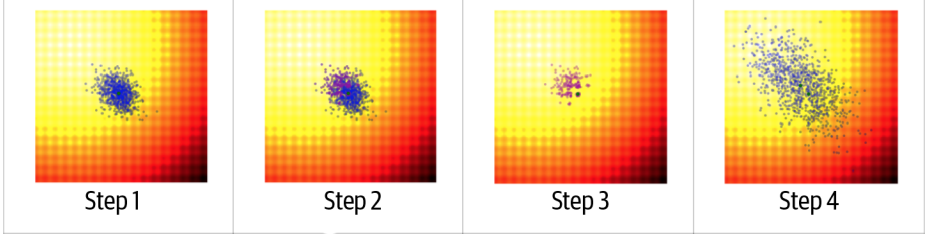

Figure 12-13. One update step from the CMA-ES algorithm (source: Ha, 2017)[2]

The steps are as follows:

1. We start with a randomly generated 2D normal distribution and sample a population of candidates, shown in blue in Figure 12-13.
2. We then calculate the value of the function for each candidate and isolate the best 25%, shown in purple in Figure 12-13—we'll call this set of points `P`.
3. We set the mean of the new normal distribution to be the mean of the points in `P`. This can be thought of as the breeding stage, wherein we only use the best candidates to generate a new mean for the distribution. We also set the covariance matrix of the new normal distribution to be the covariance matrix of the points in `P`, but use the existing mean in the covariance calculation rather than the current mean of the points in `P`. The larger the difference between the existing mean and the mean of the points in `P`, the wider the variance of the next normal distribution. This has the effect of naturally creating *momentum* in the search for the optimal parameters.

4. We can then sample a new population of candidates from our new normal distribution with an updated mean and covariance matrix.

Figure 12-14 shows several generations of the process. See how the covariance widens as the mean moves in large steps toward the minimum, but narrows as the mean settles into the true minimum.

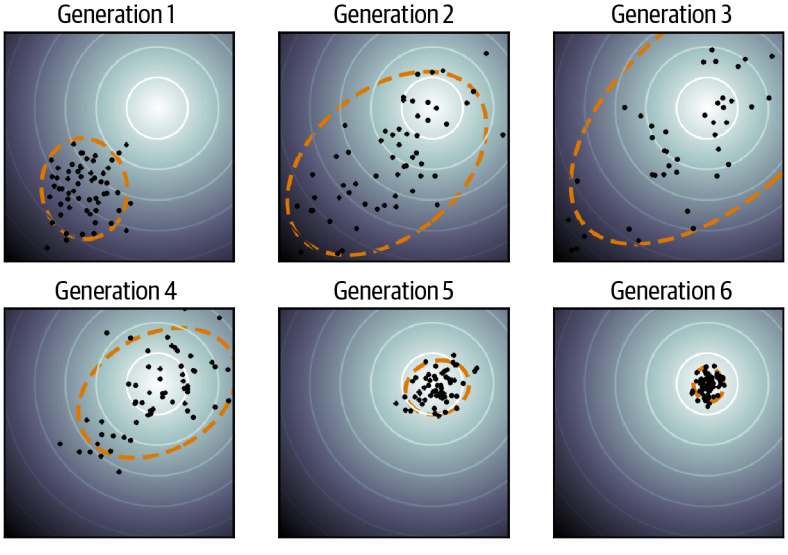

Figure 12-14. CMA-ES (source: Wikipedia)

For the car racing task, we do not have a well-defined function to maximize, but instead an environment where the 867 parameters to be optimized determine how well the agent scores. Initially, some sets of parameters will, by random chance, generate scores that are higher than others and the algorithm will gradually move the normal distribution in the direction of those parameters that score highest in the environment.

Parallelizing CMA-ES

One of the great benefits of CMA-ES is that it can be easily parallelized. The most time-consuming part of the algorithm is calculating the score for a given set of parameters, since it needs to simulate an agent with these parameters in the environment. However, this process can be parallelized, since there are no dependencies between individual simulations. There is a orchestrator process that sends out parameter sets to be tested to many node processes in parallel. The nodes return the results to the orchestrator, which accumulates the results and then passes the overall result of the generation to the CMA-ES object. This object updates the mean and covariance

matrix of the normal distribution as per Figure 12-13 and provides the orchestrator with a new population to test. The loop then starts again. Figure 12-15 explains this in a diagram.

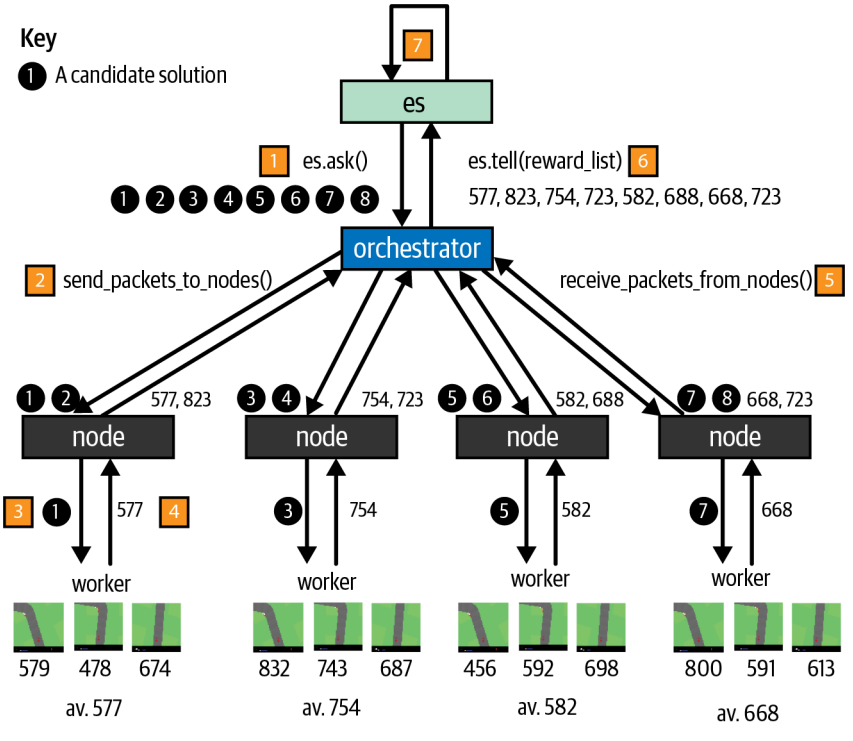

Figure 12-15. Parallelizing CMA-ES—here there is a population size of eight and four nodes (so t = 2, the number of trials that each node is responsible for)

❶ The orchestrator asks the CMA-ES object (`es`) for a set of parameters to trial.

❷ The orchestrator divides the parameters into the number of nodes available. Here, each of the four node processes gets two parameter sets to trial.

❸ The nodes run a worker process that loops over each set of parameters and runs several episodes for each. Here we run three episodes for each set of parameters.

❹ The rewards from each episode are averaged to give a single score for each set of parameters.

❺ Each node returns its list of scores to the orchestrator.

❻ The orchestrator groups all the scores together and sends this list to the `es` object.

❼ The `es` object uses this list of rewards to calculate the new normal distribution as per Figure 12-13.

After around 200 generations, the training process achieves an average reward score of around 840 for the car racing task, as shown in Figure 12-16.

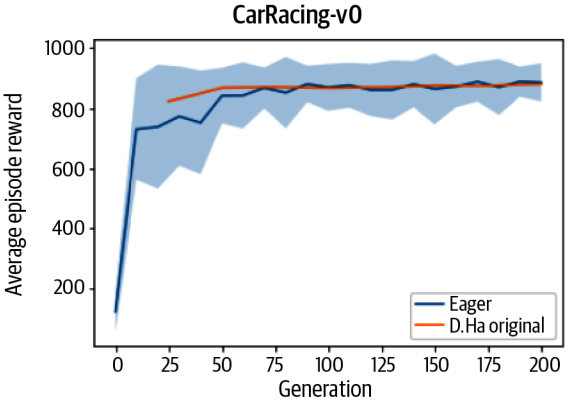

Figure 12-16. Average episode reward of the controller training process, by generation (source: Zac Wellmer, "World Models")

In-Dream Training

So far, the controller training has been conducted using the Gym `CarRacing` environment to implement the step method that moves the simulation from one state to the next. This function calculates the next state and reward, given the current state of the environment and chosen action.

Notice how the step method performs a very similar function to the MDN-RNN in our model. Sampling from the MDN-RNN outputs a prediction for the next `z` and reward, given the current `z` and chosen action.

In fact, the MDN-RNN can be thought of as an environment in its own right, but operating in `z`-space rather than in the original image space. Incredibly, this means that we can actually substitute the real environment with a copy of the MDN-RNN and train the controller entirely within an MDN-RNN-inspired *dream* of how the environment should behave.

In other words, the MDN-RNN has learned enough about the general physics of the real environment from the original random movement dataset that it can be used as a proxy for the real environment when training the controller. This is quite

remarkable—it means that the agent can train itself to learn a new task by *thinking* about how it can maximize reward in its dream environment, without ever having to test out strategies in the real world. It can then perform well at the task the first time, having never attempted the task in reality.

A comparison of the architectures for training in the real environment and the dream environment follows: the real-world architecture is shown in Figure 12-17 and the in-dream training setup is illustrated in Figure 12-18.

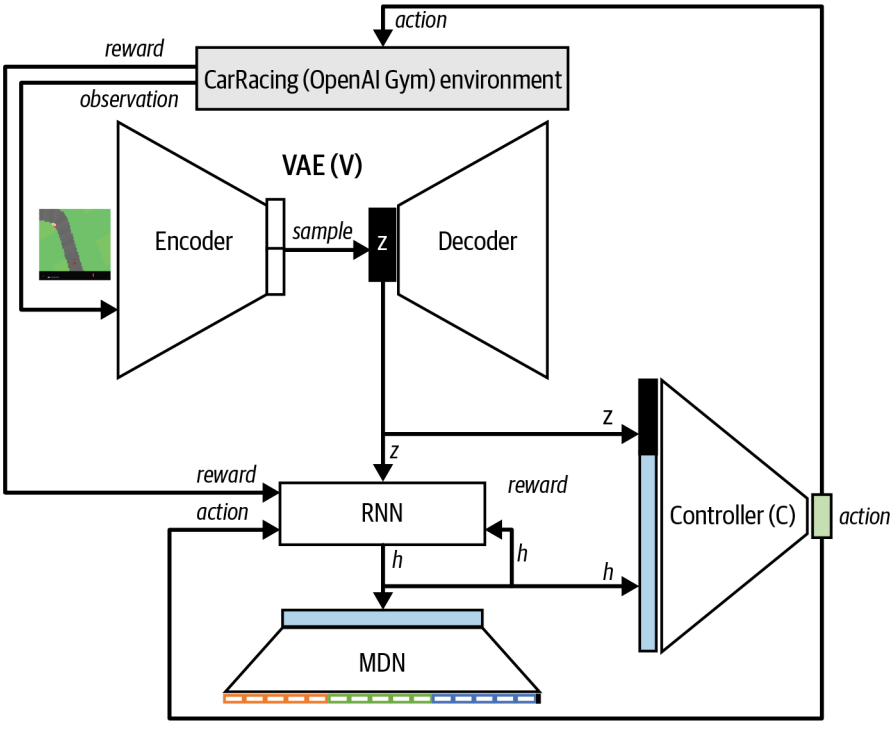

Figure 12-17. Training the controller in the Gym environment

Notice how in the dream architecture, the training of the controller is performed entirely in z-space without the need to ever decode the z vectors back into recognizable track images. We can of course do so, in order to visually inspect the performance of the agent, but it is not required for training.

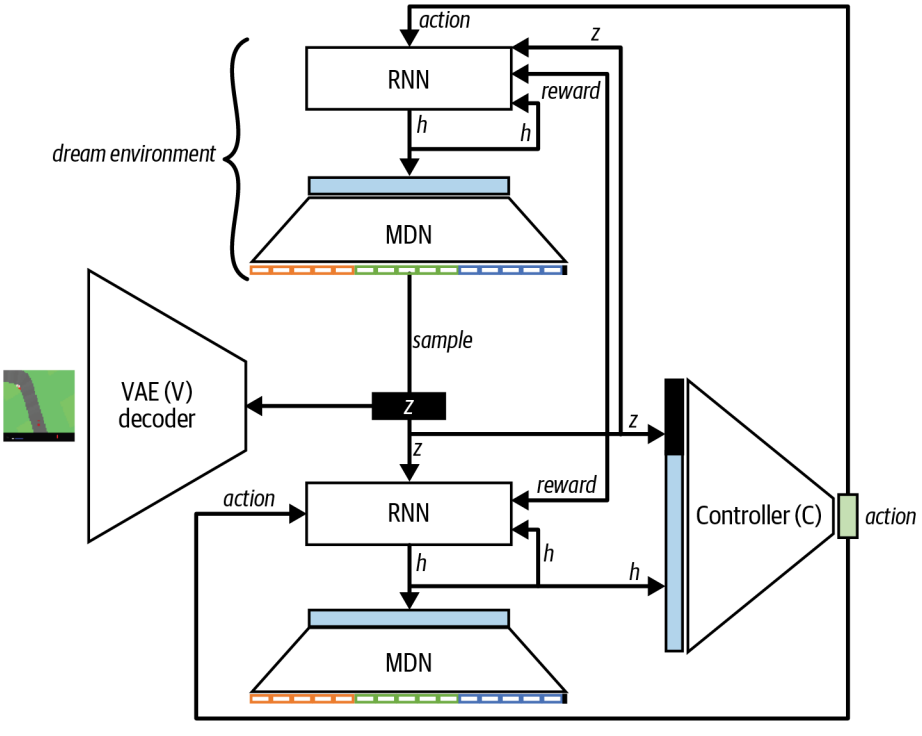

Figure 12-18. Training the controller in the MDN-RNN dream environment

One of the challenges of training agents entirely within the MDN-RNN dream environment is overfitting. This occurs when the agent finds a strategy that is rewarding in the dream environment but does not generalize well to the real environment, due to the MDN-RNN not fully capturing how the true environment behaves under certain conditions.

The authors of the original paper highlight this challenge and show how including a `temperature` parameter to control model uncertainty can help alleviate the problem. Increasing this parameter magnifies the variance when sampling `z` through the MDN-RNN, leading to more volatile rollouts when training in the dream environment. The controller receives higher rewards for safer strategies that encounter well-understood states and therefore tend to generalize better to the real environment. Increased temperature, however, needs to be balanced against not making the environment so volatile that the controller cannot learn any strategy, as there is not enough consistency in how the dream environment evolves over time.

In the original paper, the authors show this technique successfully applied to a different environment: `DoomTakeCover`, based around the computer game *Doom*.

Figure 12-19 shows how changing the `temperature` parameter affects both the virtual (dream) score and the actual score in the real environment.

TEMPERATURE τ	VIRTUAL SCORE	ACTUAL SCORE
0.10	2086 ± 140	193 ± 58
0.50	2060 ± 277	196 ± 50
1.00	1145 ± 690	868 ± 511
1.15	918 ± 546	1092 ± 556
1.30	732 ± 269	753 ± 139
RANDOM POLICY	N/A	210 ± 108
GYM LEADER	N/A	820 ± 58

Figure 12-19. Using temperature to control dream environment volatility (source: Ha and Schmidhuber, 2018)

The optimal temperature setting of 1.15 achieves a score of 1,092 in the real environment, surpassing the current Gym leader at the time of publication. This is an amazing achievement—remember, the controller has *never* attempted the task in the real environment. It has only ever taken random steps in the real environment (to train the VAE and MDN-RNN *dream* model) and then used the dream environment to train the controller.

A key benefit of using generative world models as an approach to reinforcement learning is that each generation of training in the dream environment is much faster than training in the real environment. This is because the `z` and reward prediction by the MDN-RNN is faster than the `z` and reward calculation by the Gym environment.

Summary

In this chapter we have seen how a generative model (a VAE) can be utilized within a reinforcement learning setting to enable an agent to learn an effective strategy by testing policies within its own generated dreams, rather than within the real environment.

The VAE is trained to learn a latent representation of the environment, which is then used as input to a recurrent neural network that forecasts future trajectories within the latent space. Amazingly, the agent can then use this generative model as a pseudo-environment to iteratively test policies, using an evolutionary methodology, that generalize well to the real environment.

For further information on the model, there is an excellent interactive explanation available online, written by the authors of the original paper.

References

1. David Ha and Jürgen Schmidhuber, "World Models," March 27, 2018, *https://arxiv.org/abs/1803.10122*.

2. David Ha, "A Visual Guide to Evolution Strategies," October 29, 2017, *https://blog.otoro.net/2017/10/29/visual-evolution-strategies*.