

Deep Generative Models (AI314)

Master of Science in AI



Dr. Nibras Abdullah

*Assistant professor
Faculty of Informatics
Midocean University*



Real-world Applications

Introduction

Reinforcement Learning

World Model Overview

Collecting Random Rollout Data

Training the VAE

Collecting Data to Train the MDN-RNN

Training the Controller

World Models and Reinforcement Learning



In this presentation, we will explore the concept of World Models and delve into the fundamentals of Reinforcement Learning. We'll examine how these techniques can be applied to train agents to perform optimally in virtual environments, with a focus on a car racing simulation.

Introduction to World Models

World Models represent an innovative approach to training agents in complex environments. This technique allows agents to learn and make decisions based on a simulated representation of their environment. In our exploration, we'll focus on a task where an agent must learn to drive a car around a virtual racetrack as quickly as possible. This example demonstrates how World Models can be applied to optimize performance in challenging scenarios.

1

Environment Simulation

Create a virtual representation of the racetrack environment.

2

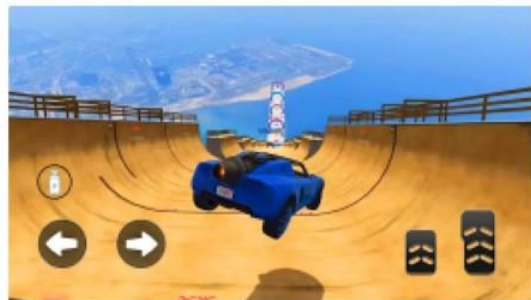
Agent Training

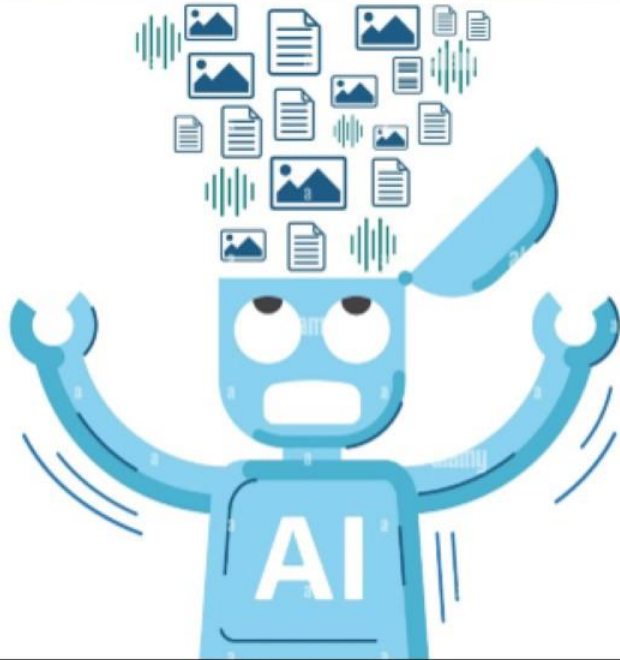
Use the World Model to train the agent in the simulated environment.

3

Performance Optimization

Refine the agent's strategies to achieve optimal racing performance.





Reinforcement Learning: An Overview

Reinforcement Learning (RL) is a branch of machine learning that focuses on training agents to perform optimally within a given environment, with respect to a particular goal. Unlike discriminative and generative modeling, which aim to minimize loss functions over datasets, RL seeks to maximize the long-term reward of an agent in its environment. It stands alongside supervised and unsupervised learning as one of the three major branches of machine learning.

Goal-Oriented

RL aims to maximize long-term rewards for the agent.

Environment Interaction

Agents learn through repeated interactions with their environment.

Adaptive Learning

RL allows agents to adapt and improve their strategies over time.

The Environment in Reinforcement Learning

The environment is a crucial element in Reinforcement Learning, serving as the world in which the agent operates. It defines the set of rules governing game state updates and reward allocation based on the agent's actions and current game state. In our racing example, the environment would include the track layout, physics simulations, and scoring mechanisms. The environment provides the context and challenges that the agent must learn to navigate and overcome.

Rules

Defines how actions affect the game state.

State Updates

Calculates new states based on agent actions.

Reward System

Allocates rewards based on agent performance.

Key Components of Reinforcement Learning

Reinforcement Learning involves several key components that work together to create an effective learning system. These elements include the environment, agent, game state, actions, and rewards. Understanding these components is crucial for grasping how RL operates and how it can be applied to various scenarios, including our virtual racing example.



Environment

The world in which the agent operates, defining rules and rewards.



Agent

The entity that takes actions within the environment.



Game State

Data representing a particular situation the agent may encounter.



Reward

Value given to the agent after an action is taken.

Agents and Actions in Reinforcement Learning



In Reinforcement Learning, the agent is the entity that takes actions within the environment. The agent's goal is to learn the optimal strategy for maximizing long-term rewards. Actions are the feasible moves that an agent can make at any given time. In our racing scenario, the agent would be the AI controlling the car, and actions might include steering, accelerating, or braking. The agent must learn to choose the best actions based on the current game state to achieve the fastest lap times.

1

Observe State

Agent receives current game state information.

2

Choose Action

Agent selects the best action based on its learned strategy.

3

Perform Action

Agent executes the chosen action in the environment.

4

Receive Feedback

Agent gets new state and reward information.

Rewards and Episodes in Reinforcement Learning

Rewards are values given to the agent by the environment after an action is taken. The agent's primary objective is to maximize the long-term sum of these rewards. In our racing example, rewards might be based on factors such as lap time improvements or maintaining optimal racing lines. An episode, also known as a rollout, represents one complete run of an agent in the environment. For our racing

scenario, an episode might consist of a single lap or an entire race.

Action	Reward
Optimal racing line	+5 points
Improved lap time	+10 points
Collision	-20 points
Completing a lap	+50 points



The Reinforcement Learning Cycle

The Reinforcement Learning cycle involves a continuous interaction between the agent and the environment. This process begins with the environment initializing the current game state. The agent then receives this state and decides on its next best action. After performing the action, the environment calculates the next state and reward, which are then passed back to the agent. This cycle continues until the episode's end criterion is met, such as a specific number of timesteps elapsing or the agent winning or losing.

**1****Agent**

Receives state, chooses action

2**Environment**

Updates state, calculates reward

3**Agent**

Receives new state and reward

Designing Effective Reinforcement Learning Agents

Creating an agent that can maximize rewards in a given environment is a complex task. While it's possible to design an agent with a set of predefined rules, this approach becomes impractical as environments grow more complex. It also limits the agent's ability to achieve superhuman performance. Instead, Reinforcement Learning involves creating agents that can learn optimal strategies independently through repeated interactions with the environment. This approach allows for the development of highly adaptable and efficient agents capable of excelling in complex tasks, such as our virtual racing scenario.

1 Adaptive Learning

Agents learn and improve strategies through repeated interactions.

2 Complex Environments

RL allows agents to navigate increasingly complex scenarios.

3 Superhuman Performance

Agents can potentially achieve beyond-human levels of performance.

4 Continuous Improvement

Agents refine their strategies over time for optimal results.

World Model Architecture Overview

This presentation provides a high-level overview of the world model architecture and training process for an AI agent learning to drive in a simulated environment. The solution consists of three key components that work together to enable the agent to perceive its environment, predict future states, and make decisions: a variational autoencoder (VAE), a recurrent neural network with a mixture density network (MDN-RNN), and a controller. We'll explore how these components interact and the step-by-step training process used to develop the agent's capabilities.



World Model Components

Variational Autoencoder (VAE)

Condenses high-dimensional input images into a latent random variable, representing the current state.

MDN-RNN

Predicts the distribution of the next latent state based on the previous state and action.

Controller

Chooses actions based on the current latent state and RNN hidden state.

Variational Autoencoder (VAE)

1 Input Processing

Takes $64 \times 64 \times 3$ (RGB) input image and condenses it into a 32-dimensional normally distributed random variable.

2 Latent Representation

Parameterized by μ and $\log\sigma^2$ (logarithm of variance).

3 Sampling

Samples from the distribution to produce a latent vector z representing the current state.

MDN-RNN: Predicting Future States

1

LSTM Layer

256 hidden units process the sequence of latent states and actions.

2

Mixture Density Network

Output layer allows for multiple possible distributions of the next latent state.

3

Multiple Distributions

Each element of the next observed latent state can be drawn from any of five normal distributions.



The Controller: Action

Input

Concatenation of z (current latent state) and RNN hidden state.

Network Structure

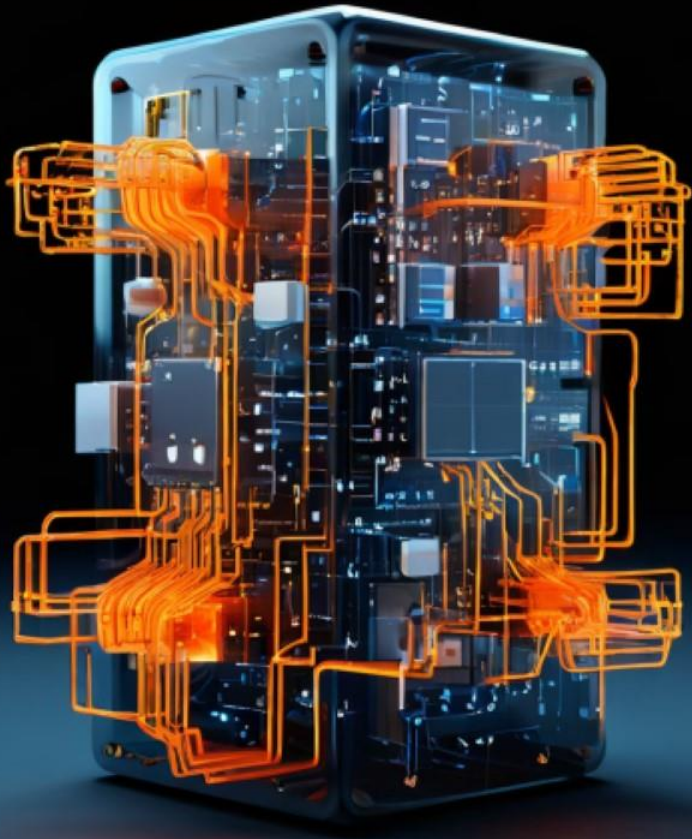
Densely connected neural network.

Output

Three neurons corresponding to turn, accelerate, and brake actions.

Training

Uses reinforcement learning to discover optimal actions.





Component Interaction

1

VAE Observation

Processes $64 \times 64 \times 3$ image into latent vector z .

2

RNN Prediction

Updates hidden state based on z and previous action.

3

Controller Decision

Outputs action based on z and RNN hidden state.

4

Environment Update

Returns new observation based on action.

Training Process Overview

1

1. Collect Random Rollout Data

Agent explores environment using random actions, storing observations, actions, and rewards.

2

2. Train VAE

Learn to encode observations into latent vectors efficiently.

3

3. Generate Training Data

Use trained VAE to create dataset of latent vectors, actions, and rewards.

4

4. Train MDN-RNN

Learn to predict subsequent latent vectors and rewards.

5

5. Train Controller

Use evolutionary algorithm to optimize action selection.

MDN-RNN Training

Details

1 Data Preparation

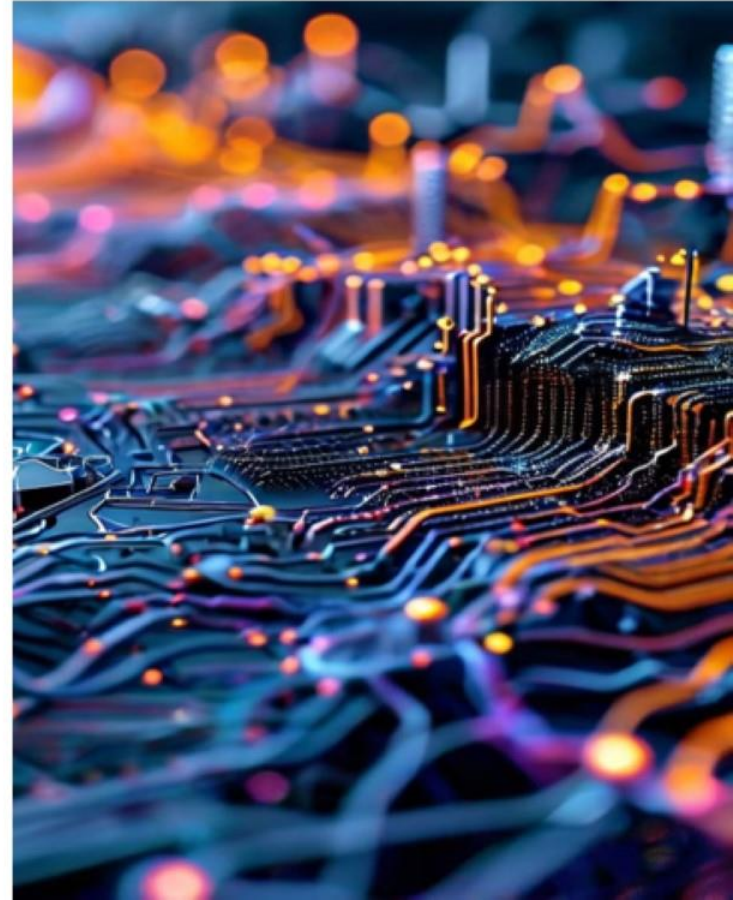
Load batches of episodes with μ , $\log\sigma$, action, and reward variables.

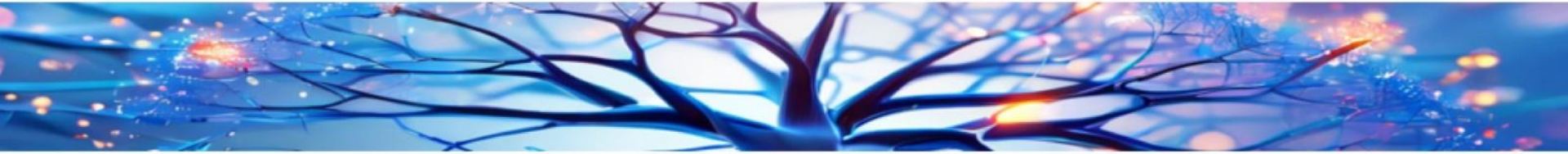
2 Latent Vector Sampling

Sample z vectors from μ and $\log\sigma$ distributions.

3 Prediction Task

Given current z , action, and reward, predict next z and reward.





Controller Training with CMA-ES

Evolutionary Algorithm

Uses CMA-ES (Covariance Matrix Adaptation Evolution Strategy) as optimizer.

Reward Mechanism

Favors matrix weightings that generate high-scoring actions.

Generational Learning

Future generations inherit and improve upon successful behaviors.

Environment Interaction

Training occurs within the agent's generative model of the environment.

Collecting Random Rollout Data for Reinforcement Learning

In reinforcement learning, the first step is to collect rollout data from the environment using an agent taking random actions. This process provides the foundation for the agent to learn about the world and how its actions influence subsequent observations.

The Importance of Random Actions

Collecting data using random actions may seem counterintuitive, given that we ultimately want our agent to take intelligent actions. However, this approach is crucial for providing the initial data that the agent will use to learn how the world operates and how its actions influence subsequent observations.

Foundation for Learning

Random actions provide a diverse set of experiences for the agent to learn from.

Unbiased Exploration

Random actions allow the agent to explore the environment without any preconceived notions.

Data Diversity

Random actions help collect a wide range of data points, covering various scenarios in the environment.



Parallel Data Collection

To efficiently collect rollout data, we can capture multiple episodes in parallel by spinning up multiple Python processes. Each process runs a separate instance of the environment on a separate core, allowing for faster data collection on machines with multiple cores.

1**Initialize Processes**

Set up multiple Python processes, each running a separate environment instance.

2**Assign Cores**

Allocate each process to a separate core on the machine.

3**Collect Data**

Run episodes simultaneously across all processes to gather rollout data efficiently.

4**Aggregate Results**

Combine the data collected from all parallel processes for further analysis and learning.

Hyperparameters for Data Collection

The data collection process is governed by several key hyperparameters that determine the scale and scope of the rollout data.

Hyperparameter	Description	Example Value
parallel_processes	Number of parallel processes to run	8 (if machine has ≥ 8 cores)
max_trials	Episodes each process should run in total	125 (1,000 episodes overall with 8 processes)
max_frames	Maximum number of timesteps per episode	300

Understanding Episode Structure

An episode consists of a series of frames, each representing a timestep in the environment.

Frame Sequence

The episode excerpt shows frames 40 to 59, providing a snapshot of the car's progress.

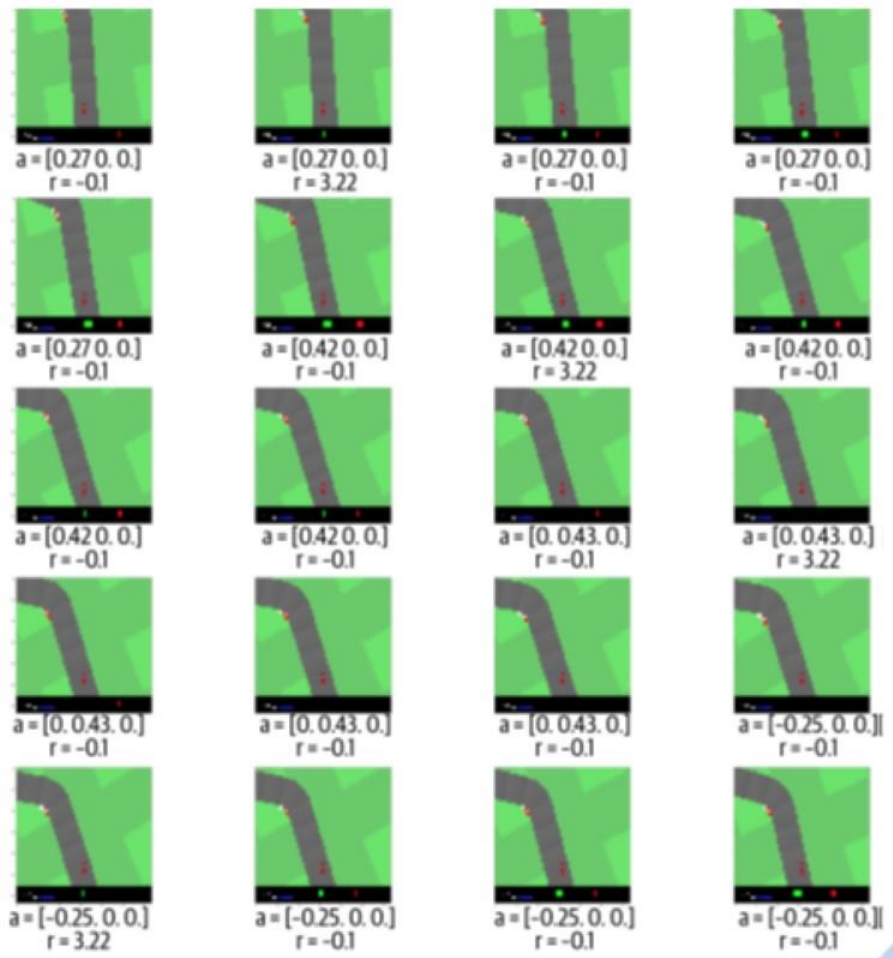
Action and Reward

Each frame is associated with a randomly chosen action and a corresponding reward.

Reward Dynamics

The reward changes to 3.22 as the car rolls over new track tiles but is otherwise -0.1 .

Figure 12-5 shows an excerpt from frames 40 to 59 of one episode, as the car approaches a corner.



Visualizing Episode Data

Figure 12-5 provides a visual representation of the episode data, showcasing the car's approach to a corner and the associated actions and rewards.



Car Approaching Corner

The figure shows the car's position as it nears a corner on the track.



Random Actions

Each frame is associated with a randomly chosen action, simulating the agent's exploration.



Reward Visualization

The changing rewards are displayed, highlighting the 3.22 reward for new track tiles.

The Role of Random Actions in Learning

While the actions taken during data collection are random, they play a crucial role in the learning process. These random actions allow the agent to explore the environment and gather diverse data points.

1

Random Exploration

The agent takes random actions to explore the environment without bias.

2

Data Collection

Each action and its consequences are recorded as part of the rollout data.

3

Pattern Recognition

The collected data is used to identify patterns and relationships in the environment.

4

Policy Improvement

These patterns inform the development of an improved action policy for the agent.

Balancing Exploration and Exploitation

While this initial data collection phase focuses on random actions for exploration, it's important to note that reinforcement learning ultimately aims to balance exploration with exploitation of learned knowledge.



Exploration

Random actions allow the agent to discover new states and potential rewards in the environment.



Exploitation

As the agent learns, it will start to exploit known good actions to maximize rewards.



Balance

Effective reinforcement learning algorithms strike a balance between exploration and exploitation.



Next Steps in the Learning Process

After collecting the random rollout data, the next steps in the reinforcement learning process involve analyzing the data, identifying patterns, and developing an improved action policy for the agent.

1

Data Analysis

Examine the collected rollout data to identify patterns and relationships between actions, states, and rewards.

2

Model Development

Use the analyzed data to develop a model of the environment and the agent's impact on it.

3

Policy Improvement

Based on the model, develop an improved policy for the agent to take more intelligent actions in the environment.

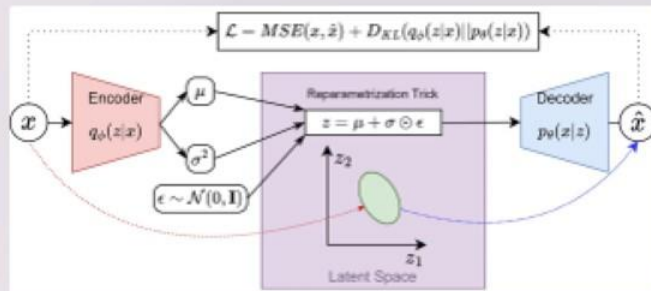
4

Iterative Learning

Continue to refine the model and policy through further iterations of data collection and analysis.

Training a Variational Autoencoder (VAE)

We will now explore the process of building and training a generative model called a Variational Autoencoder (VAE) on collected image data. The VAE allows us to compress 64x64x3 images into a normally distributed random variable z , parameterized by μ and $\log\sigma$ vectors. This presentation will cover the key aspects of VAE architecture, training, and implementation using Keras.



VAE Hyperparameters

1**vae_batch_size**

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

2**z_size**

The length of latent z vector (and therefore μ and logvar variables) (e.g., 32)

3**vae_num_epoch**

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

VAE Architecture

Overview

Keras allows us to define not only 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 submodels will be useful for encoding specific images or decoding given z vectors.

VAE Model Components

vae

This is the end-to-end VAE that is trained. It accepts a $64 \times 64 \times 3$ image as input and outputs a reconstructed $64 \times 64 \times 3$ image.

encode_mu_logvar

This accepts a $64 \times 64 \times 3$ 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 $64 \times 64 \times 3$ 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 Submodel

Input

The decode submodel accepts a z vector as input.

Process

It processes the z vector through the decoder network.

Output

It returns the reconstructed $64 \times 64 \times 3$ image.



VAE Training Process

1**Data Preparation**

Prepare the 64x64x3 image dataset for training.

2**Model Configuration**

Set up the VAE architecture with the specified hyperparameters.

3**Training**

Train the VAE for the specified number of epochs using the defined batch size.

4**Evaluation**

Assess the performance of the trained VAE using reconstruction quality metrics.

Encoding Process

1

Input Image

Start with a 64x64x3 input image.

2

Encoder Network

Pass the image through the encoder network.

3

Mu and Logvar

Generate mu and logvar vectors (length 32 each).

4

Sampling

Sample the z vector from the distribution defined by μ and $\log\varphi$.

Decoding Process

1

Input Z Vector

Start with a z vector (length 32).

2

Decoder Network

Pass the z vector through the decoder network.

3

Reconstruction

Generate a reconstructed 64x64x3 image.

VAE Applications and Benefits



Dimensionality Reduction

VAEs can compress high-dimensional data into a lower-dimensional latent space.



Image Generation

VAEs can generate new images by sampling from the learned latent space.



Interpolation

VAEs allow for smooth interpolation between different images in the latent space.



Anomaly Detection

VAEs can be used to detect anomalies by identifying poorly reconstructed inputs.



Training an MDN- RNN for World Models



This presentation outlines the process of collecting data and training a Mixture Density Network Recurrent Neural Network (MDN-RNN) as part of building a world model. We'll explore how to use a trained Variational Autoencoder (VAE) to generate training data, and then how to train the MDN-RNN to predict future states and rewards.

Collecting Data for MDN-RNN Training

1

Step 1: Use Trained VAE

Start with a trained Variational Autoencoder (VAE) model.

2

Step 2: Process Rollout Observations

Pass all random rollout observations through the `encode_mu_logvar` model.

3

Step 3: Store Encoded Data

Store the `mu` and `logvar` vectors corresponding to each observation.

4

Step 4: Combine Data

Use the encoded data along with collected action, reward, and done variables.



Understanding the Data Collection Process

Random Rollout Observations

Collect observations from random rollouts in the environment.

VAE Encoding

Use the trained VAE to encode observations into μ and $\log\sigma$ vectors.

Additional Variables

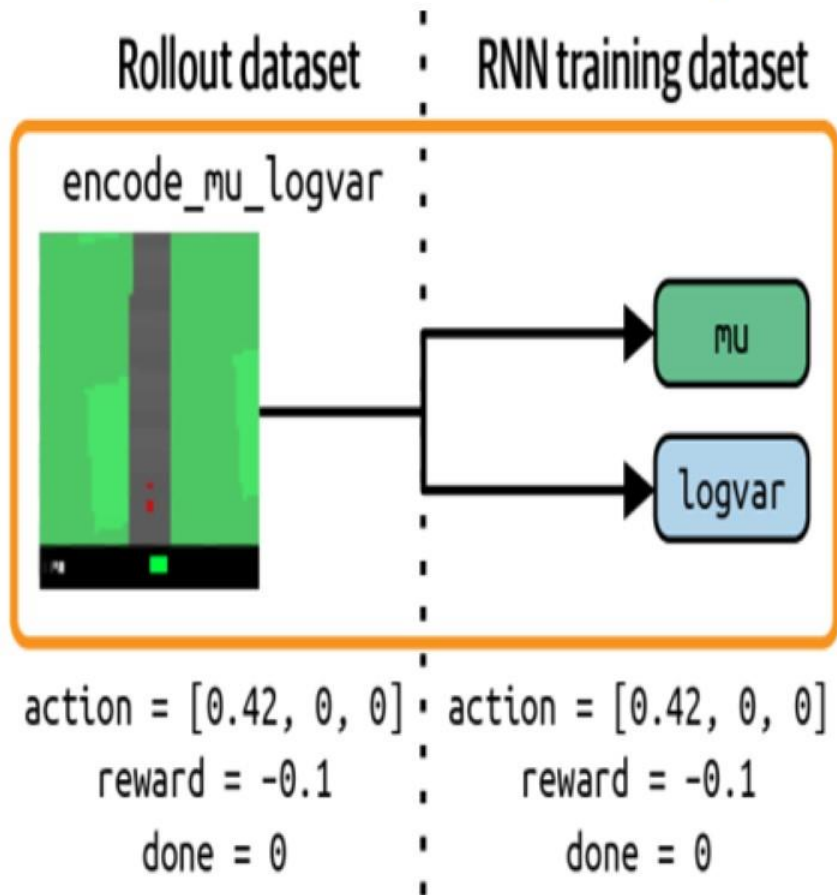
Include action, reward, and done variables collected during rollouts.

MDN-RNN Training Dataset

Combine encoded data and additional variables to form the training dataset.

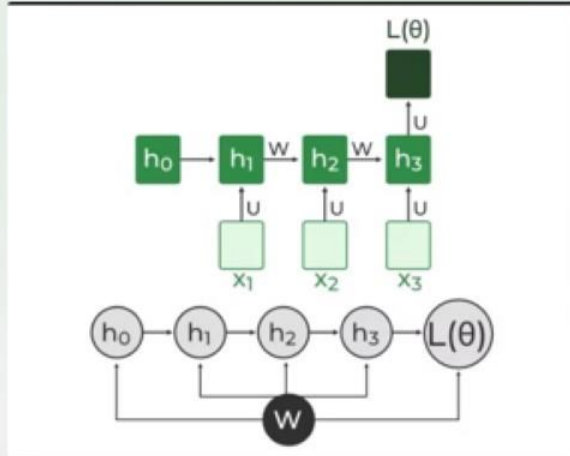
Visualizing the Data Collection Process

Figure 12-10 in the original text illustrates the process of creating the MDN-RNN training dataset. This visual representation helps to understand how the encoded data from the VAE is combined with other variables to form the complete training dataset for the MDN-RNN.



Training the MDN-RNN:

Overview



1 Purpose

Train the MDN-RNN to predict the distribution of the next z vector and reward one timestep ahead into the future.

2 Inputs

Current z vector, current action, and previous reward.

3 Outputs

Distribution of the next z vector and predicted reward.

MDN-RNN: Internal Hidden State

Definition

The internal hidden state of the RNN can be thought of as the model's current understanding of the environment dynamics.

Importance

This hidden state serves as a crucial input for the controller, which will ultimately decide on the best next action to take.

Application

By incorporating the hidden state, the controller can make more informed decisions based on the model's understanding of the environment.

MDN-RNN Training Process

1

Data Preparation

Prepare the encoded data from VAE along with action, reward, and done variables.

2

Model Configuration

Set up the MDN-RNN architecture to predict z vector distributions and rewards.

3

Training

Train the MDN-RNN using the prepared dataset to learn environment dynamics.

4

Evaluation

Assess the model's performance in predicting future states and rewards.

Integrating MDN-RNN with the Controller



MDN-RNN Predictions

The MDN-RNN provides predictions of future states and rewards.



Hidden State

The internal hidden state represents the model's understanding of environment dynamics.



Controller

The controller uses MDN-RNN outputs to decide on the best next action.

Conclusion: Building a Complete World Model

By training the MDN-RNN using data generated from the VAE, we've completed a crucial step in building a world model. This model can now predict future states and rewards, providing valuable information to the controller. The combination of the VAE for encoding observations, the MDN-RNN for predicting future states, and the controller for decision-making forms a powerful system for understanding and interacting with complex environments.

*Thank
You*