Model-Based Reinforcement Learning Technique

Model-Based Reinforcement Learning

Model-based Reinforcement Learning (Model-based RL) uses environment, action, and reward to get the most reward from the action. It focuses on the model than the policy, so the quality of the model is crucial for success. It uses machine learning models like neural networks, random forest, and others.

Unlike other reinforcement learning, Model-based RL does not require specific prior knowledge. However, having prior knowledge can speed up the agent's learning. Model-based RL requires fewer data to learn a policy and less interaction with the environment to train models. It does not need to learn directly from interactions with the real environment. The sample is used more efficiently in model-based learning because we do not need to sample again to optimize the policy once the model and the cost function are known.

The agent captures the transition function and the reward function during the interaction with the environment. If the observation environment is high dimensional during the training, the agent will not have a clear reward function, so the agent could not know whether the task is succeeded or not. We have to define the reward function by providing an image of the goal. Based on the transition and reward function, the agent can predict the next state and the reward.

Dealing with high dimensional observations

In the World Model paper, the agent has only access to high dimensional observations during the training. For that situation, we can choose to work from three different approaches: learning a model in latent space or learning a model directly from the high dimensional observations or learning inverse models. In the paper, they learn their model in latent space and apply it to the Model-Based RL, so we will be only explaining the latent approach.

Using Latent Models with the Model-Based RL for the high dimensional observations

In Latent Models, unlike the other Markov Decision Process (MDP) approach, all the MDP elements: state, action, transition, and rewards do not work on the single model. They have their

own models. Latent models are also not connected directly to the input and output. Instead, they are connected to other models and signals. It captured the interaction in different models: observation, transition, representation, and rewards, and they are either trained in self-supervised like variational autoencoder or unsupervised or recurrent networks. Basically, we learn the model in latent space and then apply it to the standard model-based reinforcement learning.

Let's look at how the latent model algorithm works. First of all, we have to run the base policy like a random policy to collect the sample data, and then we use that sampled data to learn latent embedding observations and learn dynamics model in that latent space. After that, we use that model to optimize over a sequence of actions. Then, we execute those planned actions and append visited tuples: action, observation, and the next observation.

World Model's Agent

World Model's agent has three main components: Vision (V), Memory (M), and Controller (C). They are trained separately. Vision components help the agent to encode the high-dimensional input image frame into a small latent vector z . Memory components help the agent to make the prediction about the future based on the previous information. A controller is a decision-making component which helps the agent to make the decision and take action based on the information from the vision and memory component.

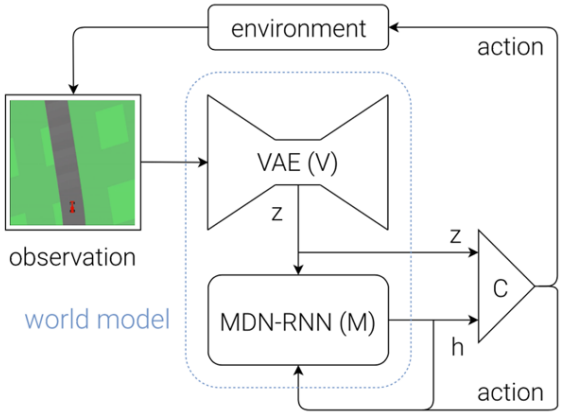

Figure 1 Flow diagram of how V, M, and C interacts with the environment in World Model

Car Racing Experiment Procedure

Step 1: Generate Random Rollouts

They let their agent explore the environment multiple times to collect 10,000 rollouts. In the meantime, they save their agent's random actions at each time step.

Step 2: Train VAE

After they have collected 10,000 rollouts, they train their agent's vision (V) model. The agent has only access to high dimensional observations during the training, so they use Convolutional Variational Autoencoder (ConvVAE) to learn their model in latent space. It helps to encode each frame into a low dimensional latent vector z. It also helps them to compress the space.

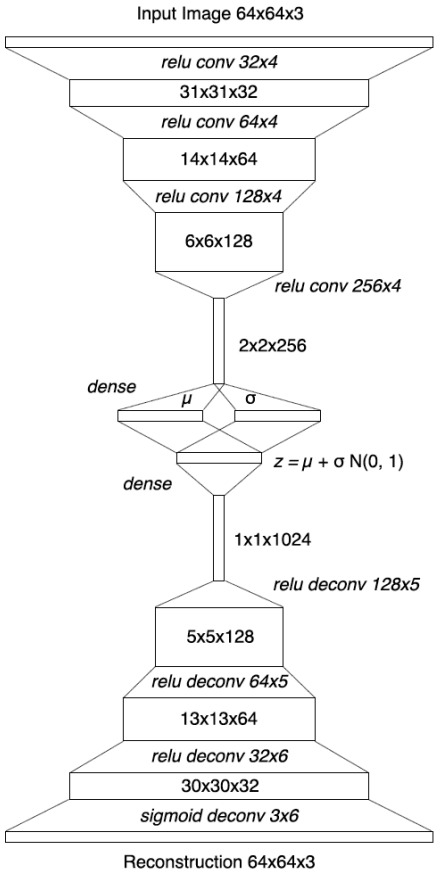

Figure 2 Convolutional Variational Autoencoder (ConvVAE)

First of all, they resize each of the high dimensional input images into 64x64 pixels and stored each of the pixels as three floating points to represent each of the RGB channels. The ConvVAE takes resized image: 64x64x3 (width, height and RGB depth), and encodes it into low dimensional vectors latent vector z of size 32 by passing through four ReLU convolutional layers. Now, the agent does not need to deal with higher representation, so the agent can learn more efficiently. After that, they decode and reconstruct the image by passing through the latent vector into three ReLU deconvolution layers and a sigmoid deconvolution layer. The output has to be between 0 and 1, so they used sigmoid in their output layer.

Step 3: Train MDN-RNN

To train their agent's M Model, they use a Mixture Density Network (MDN) combined with an RNN approach. It predicts the future vector z that will be produced by the V model. They use long short-term memory (LSTM) recurrent neural network for the RNN because it helps to resolve the RNN's vanishing gradient problem. It remembers the past data in memory and trains the model by using backpropagation.

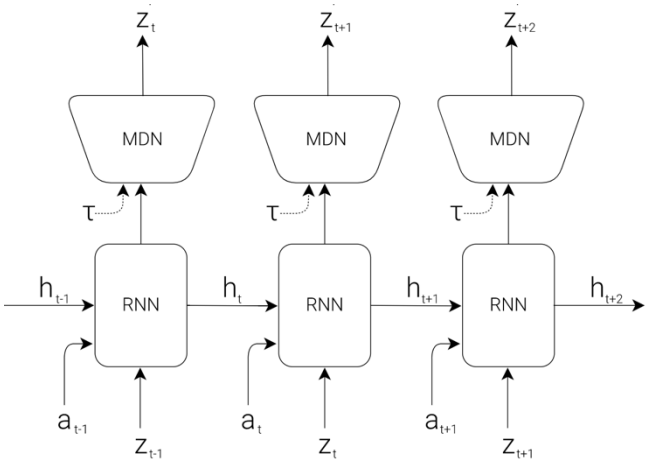

Figure 3 Recurrent Neural Network with Mixture Density Network

To generate RNN, they use the latent vector z from the VAE model, the agent's recorded random actions a, and the hidden state h of the RNN. After that, they fed it to the MDN with the temperature parameter τ. They approximate the probability density function (pdf) as a mixture of Gaussian distribution and train the RNN to get the probability distribution of the next latent vector z_{t+1}.

Step 4: Train the Controller

They trained the controller model separately from the vision and memory model. Their agent's vision and memory do not know the actual signal from reward. Only the controller (C) model knows the reward information from the environment. Their agent controller controls three continuous actions: steering left or right, acceleration, and brake.

For their agent's controller components, they use a simple single-layer linear model:

$$a_t = W_c[z_t \mathrm{h}_t] + b_c$$

In that linear model, the weight matrix W_c and bias vector b_c maps the concatenated latent vector at time t: z_t and the hidden state of the RNN at time t: h_t to the action taken at time t: a_t . They use Covariance Matrix Adaptation – Evolution Strategy (CMA-ES) to find the optimal parameters of the weight matrix W_c and bias vector b_c. CMA-ES creates multiple random initialized copies and then tests each of them inside the environment. It saves the state of the process from each generation and produces the best weight after each generation.