Project 06: World Models

December 5, 2023

Lorenzo Levantesi

1. Introduction

One way to categorize RL algorithms is with *model-based* and *model-free* RL methods.

In **model-based** RL methods, the *agent* is also composed of a **model** which is capable of simulating the dynamics of the environment in which it operates. Using this model, an *agent* can perform an action having available a representation of the *transition model* and *reward function*, which may not perfectly reflect the real environment. The **model** can be given or learned by using the interactions of the *agent* inside the environment. Having this model available lets the *agent* simulate human behaviour when an action is performed, this is because all of us have a model of the world in our brains that we use to perform actions to try to reach a desirable state. With such a model, given a sequence of actions, an agent can obtain a prediction of the future states and rewards, and use these predictions to choose the next action that better fits the reach of its goal, all without directly interacting with the real environment. During the learning phase, the *agent* can also operate exclusively inside the learned model, completely replacing the real environment.

A **model-free** RL algorithm does not use such a model. The *agent* can improve its policy only by using the data sampled from the interactions with the real environment, and it does not try to learn the dynamics of the environment. So, an agent performs an action based only on the current state of the environment and by using the experience acquired until that moment. An example of a model-free algorithm is *Q-learning*, where the *agent* estimates the value function $Q(s, a)$ by interacting directly and solely with the environment.

2. Related work

One important work in the model-based RL scenario is **World Models** (Ha & Schmidhuber, 2018). In this paper, the agent uses previous experience to create a *world model* of the environment, then this *model* is used to train the agent's *controller*, which chooses the next action to perform. The idea behind this technique is to simulate the human mental model of the world, which is constructed only by the limited number of senses available for the human, in this case the senses replicated by the *agent* are **Vision (V)** and **Memory (M)**.

The *world model* is learned in an unsupervised manner by a **vision model (V)** that, from an observation of the environment, produces a low-dimensional representation. This representation is then given in input to a **Memory (M)** model, which will maintain the most suitable historical information to predict future states. As the last step, a small **Controller (C)** uses the representations from **V** and **M** to perform the current best action, which results in a return of a reward and most likely in a new state.

One of the latest successful works on *world models* is *DreamerV3* (Hafner et al., 2023). This algorithm is scalable, meaning that if the size of the model is increased, the performances increase accordingly, and it is general, which means that the same model with fixed hyperparameters performs well in a wide range of domains. *DreamerV3* is composed of 3 neural networks: the **world model**, which predicts future outcomes of potential actions, the **critic**, which judges the value of each situation, and the **actor**, which learns to maximize its rewards. The key to the generalization of the algorithm is the use of the *symlog* function on the outputs of the models, which is invertible and compresses the magnitudes of both large positive and negative values.

DreamerV3 outperforms state of the art methods for specific domains. Most importantly, it solves the challenge of collecting diamonds in the popular video game Minecraft, without the need for human data. This challenge has been an important and difficult problem for artificial intelligence, given the sparsity of the rewards, exploration difficulty, and long-term beneficial actions.

3. Method

The *model-based* RL model used in this work is the same as Ha and Schmidhuber (Ha & Schmidhuber, 2018).

Email: Lorenzo Levantesi <levantesi.1785694@studenti.uniroma1.it>.

Deep Learning and Applied AI 2022, Sapienza University of Rome, 2nd semester a.y. 2021/2022.

The **Vision (V)** component is a *Convolutional Variational Autoencoder (ConvVAE)*, where the *encoder* takes in input an image 64x64, passes it through 4 convolutional layers and encodes it into the vectors μ and σ. The latent vector z is sampled from the Gaussian prior $N(\mu, \sigma I)$. Then, the *decoder* passes z through 4 deconvolution layers and a final sigmoid layer to reconstruct the image. The loss is the sum between the L^2 distance of the input image and reconstruction, and the KL divergence weighted by the *variational beta* parameter.

The **Memory (M)** component is an *LSTM* recurrent neural network with a Mixter Density Network as the output layer. This network models the probability distribution of z in the next time step as a mixture of Gaussian distribution. The model's input is z concatenated with the executed action, and the outputs are the parameters of the mixture of Gaussian. The loss function is the log-likelihood of the model's output and ground truth.

The **Controller (C)** component is a single fully connected layer with sigmoid as an activation function, which takes as input the concatenation of z with the *hidden state* of the *LSTM* and the output is the action to perform. The optimization phase is performed with the *Covariance-Matrix Adaptation Evolution Strategy (CMA-ES)*, an optimization algorithm which performs well with up to thousands of parameters in input.

4. Results

The PyTorch implementation of the agent is available on GitHub (`https://github.com/loreleva/DL-AAI-project-A.Y.-2021-2022`). The game on which the training and testing of the model is performed is the OpenAI PROCGEN game *leaper* (Cobbe et al., 2020), which is inspired by the classic arcade game Frogger from 1981.

The **V** model is the most important component, as it must be able to reconstruct small pixel objects of the game such as cars, logs and the frog itself. To find the right dimension for the latent code z, the following dimensions have been tested with only the reconstruction loss as a loss: 32, 64, 128, 256, 512. The dimension value of 512 performed better, but starting the training with the loss parameter *variational beta* at 1 prevented obtaining a meaningful reconstruction. Then, the technique used to train the *ConvVAE* has been to start the training with only the reconstruction loss as a loss and then once a meaningful reconstruction was obtained, the value of the *variational beta* has been incrementally increased until 1, and the training has been stopped when the reconstruction started to degrade.

To try to obtain a better reconstruction, a bigger *ConvVAE* has been trained by adding one additional convolutional layer to the *encoder* and one deconvolution layer to the *decoder*. This new network obtained the same results as the original one, so, following Occam's razor principle, the original architecture of the *ConvVAE* has been chosen. The model has been trained for a total of 478 epochs on a dataset of 10.000 rollouts of the game.

The **M** model has been trained with a hidden size of 1024 and 16 as the number of Gaussians to output. The model couldn't do so much with the prediction of the next latent vector, given that the latent space representation of the **V** model was not smooth enough. Even this model has been trained on a dataset of 10.000 rollouts of the game, for a total of 45 epochs.

For the optimization of the **C** model, the Python implementation *cma* of the CMAES algorithm has been used, with a population size of 30. The fitness score of each element of the population is defined as the mean reward of 30 rollouts of the game. The rollouts ran concurrently on 4 cpu cores for 39 generations, where the best controller has obtained a fitness score of 4.666 (the reward for a win is 10).

The final model complete of all the 3 components has been tested on 100 rollouts, obtaining 14 wins and 86 game over.

5. Discussion and conclusions

A **model-based** method is an approach to consider when creating an RL *agent*, as it generally results in faster learning of a "good" policy than *model-free* methods. One of the main **advantage** of the *model-based* RL is the reduced *sample complexity* (i.e., number of samples/interactions obtained from the real environment) in the training phase. Being able to plan a strategy according to what will be the results of the actions, help the *agent* choose the proper actions to perform, resulting in low trial-and-error costs. Another benefit of *model-based* RL methods is that the learned model could be used to accomplish different tasks, so it is possible to train multiple agents with different objectives using the same model. By being able to model the dynamics of the environment we can also train the agent on scenarios that in the real world may be rare or impossible to see, resulting in a more robust policy. A **downside** of *model-based* RL is that the performances of the *agent* are dependent on the quality of the learned model, which may be low when the task is complex and high dimensional. This problem can result in training agents which perform well inside the model, but when put into action inside the real environment the performances are worse. This happened in World Models (Ha & Schmidhuber, 2018), where the agent has learned to fool the world model to maximize its expected cumulative reward.

References

Cobbe, K., Hesse, C., Hilton, J., and Schulman, J. Leveraging procedural generation to benchmark reinforcement learning. In *International conference on machine learning*, pp. 2048–2056. PMLR, 2020.

Ha, D. and Schmidhuber, J. World models. *arXiv preprint arXiv:1803.10122*, 2018.

Hafner, D., Pasukonis, J., Ba, J., and Lillicrap, T. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.