CS577 Project Report

Recurrent World Models Facilitate Policy Evolution

Sankalp Sanand (A20448061) ssanand@hawk.iit.edu

Arinjay Jain (A20447307) ajain80@hawk.iit.edu

Abstract

We explore building generative neural network models of popular reinforcement learning environments. Our model can be trained quickly in an unsupervised manner to learn a compressed spatial and temporal representation of the environment. By using features extracted from the world model as inputs to an agent, we can train a very compact and simple policy that can solve the required task. We can even train our agent entirely inside of its own dream environment generated by its model, and transfer this policy back into the actual environment.

Problem Statement

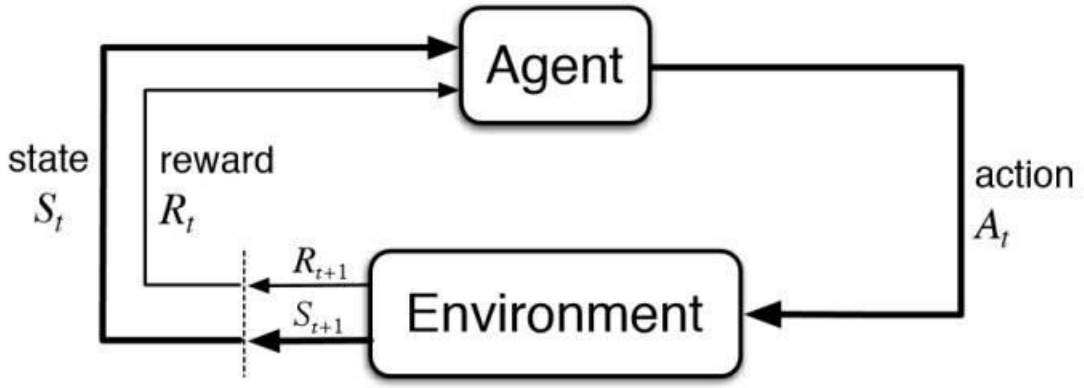
Can agents learn inside of their own dreams? Building generative neural network models of Popular reinforcement learning environments. Our model can be trained quickly in an unsupervised manner to learn a compressed spatial and temporal representation of the environment. By using features extracted from the world model as inputs to an agent, we can train a very compact and simple policy that can solve the required task.

1.1. What is Reinforcement Learning?

Reinforcement learning is the training of machine learning models to make a sequence of decisions. The agent learns to achieve a goal in an uncertain, potentially complex environment. In reinforcement learning, an artificial agent faces a game-like situation. The computer employs trial and error to come up with a solution to the problem. To get the machine to do what the programmer wants, the agent gets either rewards or penalties for the actions it performs. Its goal is to maximize the total reward.

Although the designer sets the reward policy–that is, the rules of the game–he gives the model no hints or suggestions for how to solve the game. It's up to the model to figure out how to perform the task to maximize the reward, starting from totally random trials and finishing with sophisticated tactics and superhuman skills. By leveraging the power of search and many trials, reinforcement learning is currently the most effective way to hint at a machine's creativity. In contrast to human beings,

artificial intelligence can gather experience from thousands of parallel gameplays if a reinforcement learning algorithm is run on a sufficiently powerful computer infrastructure.



1.2. Data

Since our project uses Reinforcement Learning, data is generated on the fly as the agent in question continues to play the game and learn a policy. We use one of OpenAI Gym's environments namely DoomTakeCover for performing our experiments. These environments generate frames of their respective games which is then used as input to our world model. Thus, in a way it can be seen that we don't need any external source of data.

1.3. DoomTakeCover Environment

This environment is 3 dimensional in nature and has rather simple controls of moving left, moving right, or standing still. The objective in this game is to dodge the fireballs hurling towards the agent from the other side by enemies and remain alive for as much time as possible. The number of frames through which the agent manages to stay alive is calculated as the score. Due to this mechanics of the game, a more necessary parameter is also given to the agent as input, which is a boolean value indicating whether the agent is dead in the current frame or not.

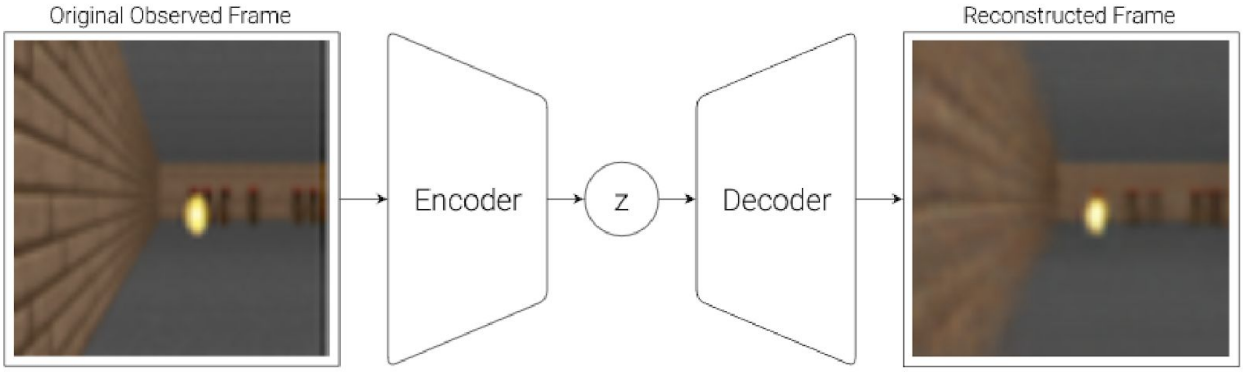
DoomTakeCover Environment

Proposed method

We present a simple model inspired by our own cognitive system. In this model, our agent has a visual sensory component that compresses what it sees into a small representative code. It also has a memory component that makes predictions about future codes based on historical information. Finally, our agent has a decision-making component that decides what actions to take based only on the representations created by its vision and memory components.

Modules Description

1. Vision Model

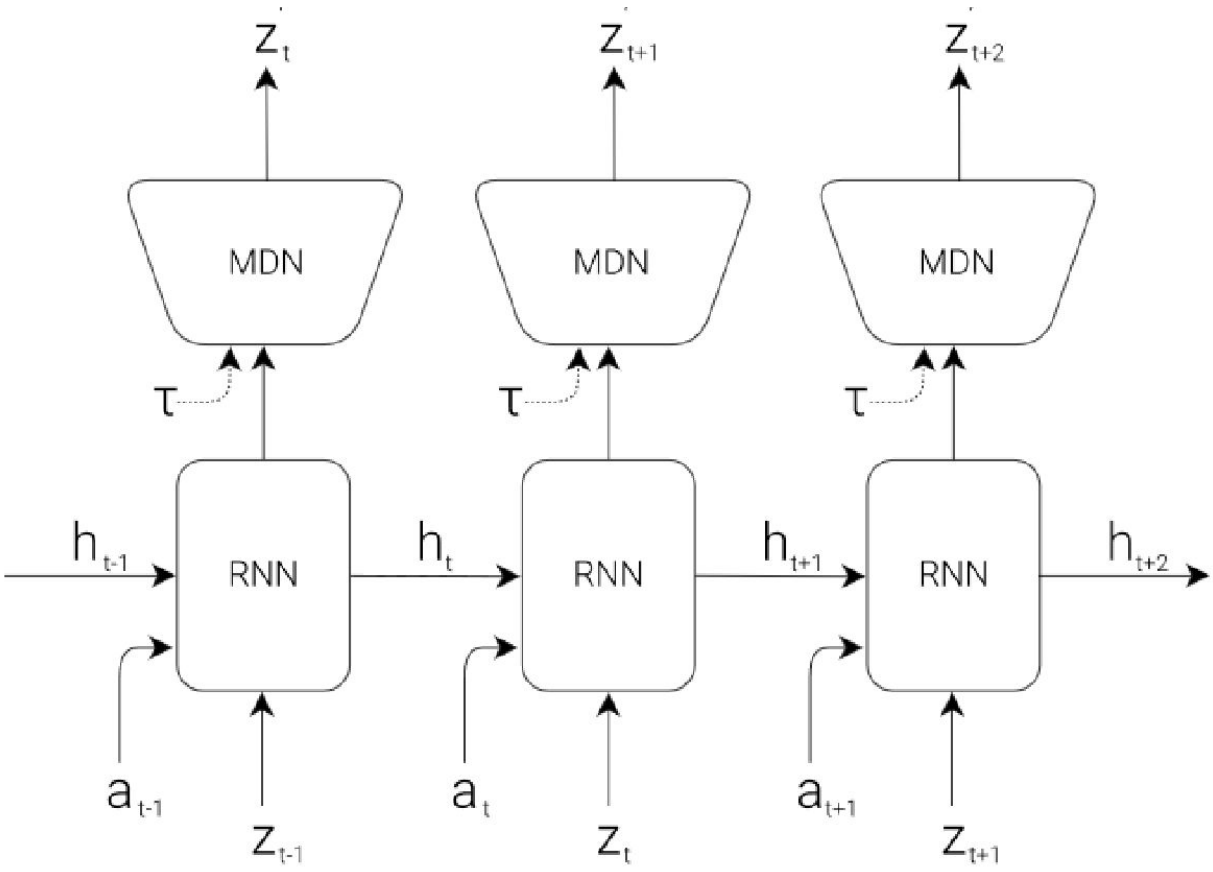


Encoding and Decoding of latent vector z

The Vision model receives the original raw environment images or frames as input and tries to create a similar but compressed representation of that frame. It learns what parts of the frame are necessary and what is being repeated in order to create a latent vector which will be fed to the components in further steps.

This learning takes place through a Variational Autoencoder which first encodes the frame obtained to a low dimensional latent vector z. Then it tries to recreate the original frame from that vector z and compares the original with the recreated version while adjusting the weights in its neural network. Convolutional Neural Networks (CNN) are an obvious choice in this case since the operations are being performed on an image. This z vector is then provided as input to the Memory model.

2. Memory Model



Memory model with RNN in combination with MDN

The Memory model learns about the changes in the environment as a result of the action performed and keeps those in the agent's memory so that it can learn from those alterations and improve its decision making process. That information about modifications and its subsequent results is stored in the hidden variable h which is concatenated with z in the later stages and sent to the Controller model. This component is used as a predictive model of future z vectors that will be produced by the Vision model, and thus it gradually learns to predict the content of the next frame. Since most of the complex environments are stochastic in nature, this model produces a probability density distribution instead of a deterministic value of z.

As there are several shortcomings of basic RNNs, for example, they are not able to model long term dependencies and perform poorly if the context had occurred far back in the sequence, an LSTM is used to efficiently address this issue. By doing this, long term dependencies can now be taken into account by the network and the problem of vanishing gradient will also be solved.

Since the agent now has a way to learn from its previous states and predict what might happen in the next states it has become capable enough to anticipate the danger or reward beforehand and take necessary decisions accordingly.

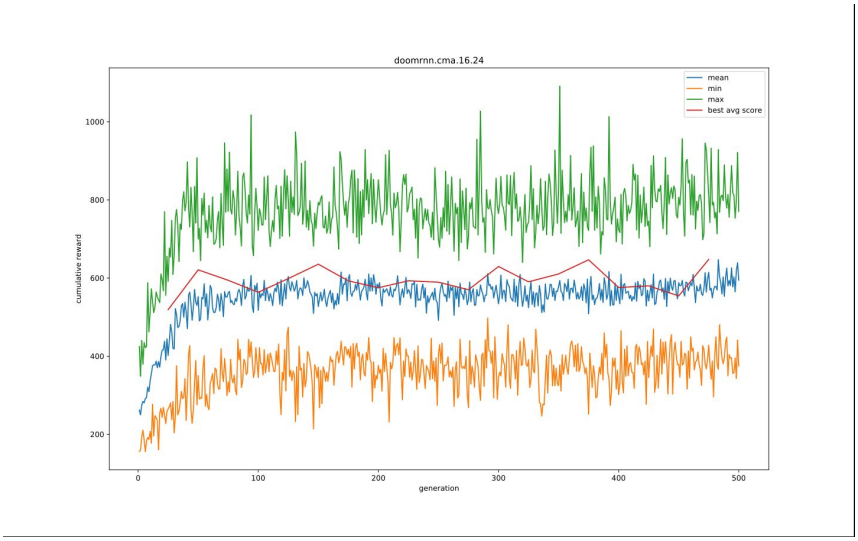
3. Controller Model

The controller model takes the responsibility of determining what course of action would be most rewarding to the agent. Since the previously discussed models perform most of the heavy lifting in this architecture and are quite complex, it was better to keep this model as simple as possible. Which is why a single layer linear model was used which maps the outputs of previous two models z and h to an action a.

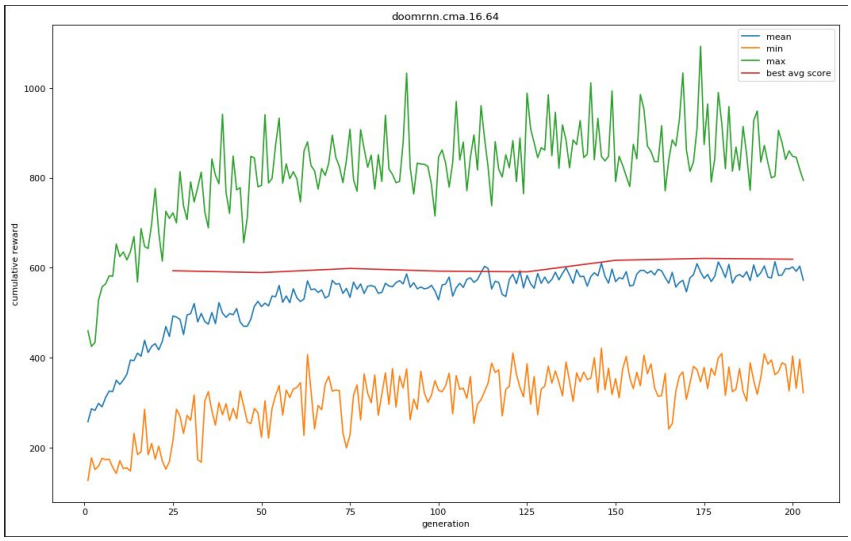
In order to train the controller model and optimize its parameters to give the best performance in its environment, a type of evolutionary strategy algorithm was used known as Covariance Matrix Adaptation (CMA). The uniqueness of this algorithm lies in its dynamic standard deviation which makes the search space adaptively increase or decrease depending upon the situation. A fitness shaping function is also applied to the fitness score received by each set of parameters so that the algorithm doesn't converge to local minima.

Results

After training the agent to play on a 24 core for nearly 500 generations the following changes in cumulative reward were obtained. The score shows the number of time steps the agent managed to stay alive before getting shot by the fireball. For contrast, the best reported score before this came out was around 820.

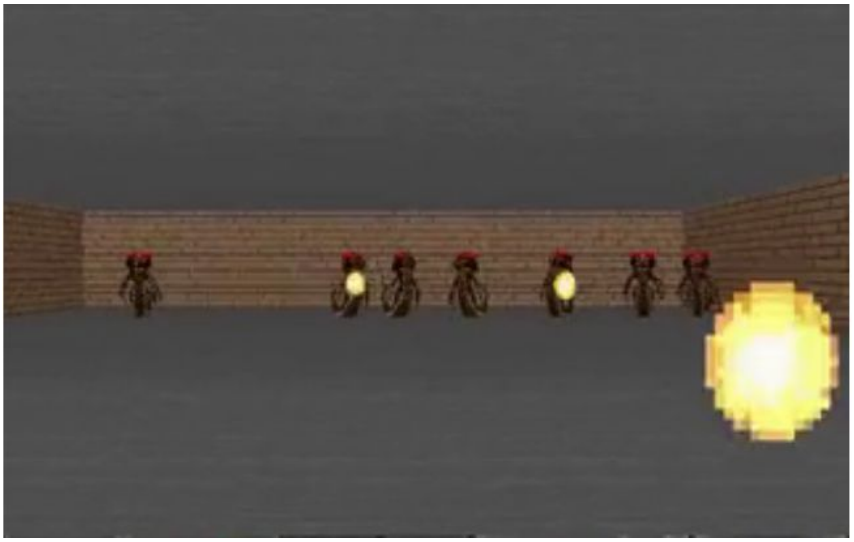


Below are the results obtained after running for 200 generations on a 64 core processor.
In all the cases when VAE and MDN-RNN was required to be trained, a NVIDIA P100 GPU was used.



Number of cores used	Average score of the agent with 5 sets of 100 iterations each
24 - 500 generations	838.58
64 - 200 generations	821.39

Visualizations of the mid training data can be seen as,
The real environment - on which the agent has to perform,



The hallucinated environment with a temperature gradient which increases the difficulty for the agent but also makes it perform better in the original environment as it has been trained in this environment.

The results mentioned above in the graphs and tables are obtained after training the agent in the dream environment and then letting it play in the real environment. Results obtained by the authors were greatly better since they trained the model for multiple days with better resources.

References

1. World Models blogpost by the authors, https://worldmodels.github.io/

2. OpenAI Gym, https://gym.openai.com/

3. Evolutionary strategy CMA, was used and understood from here.
https://blog.otoro.net/2017/10/29/visual-evolution-strategies/

4. Variational Autoencoders, https://jmetzen.github.io/2015-11-27/vae.html

5. Mixture Density Networks,
https://blog.otoro.net/2015/11/24/mixture-density-networks-with-tensorflow/

6. DoomTakeCover-v0,
https://gym.openai.com/envs/DoomTakeCover-v0/

7. Reference for our implementation,
https://github.com/worldmodels/worldmodels.github.io