Reproducing World Models

Evan Racah
Montreal Institute for Learning Algorithms, Université de Montréal

1 Introduction/Motivation

World Models (Ha & Schmidhuber (2018)) is interesting approach to model-based RL, which achieves impressive results on CarRacing and Doom. While traditional model-based RL learns a model online, while exploring the environment and then alternates learning from the environment and the model, World Models learns the model exclusively offline using a random behavior policy. Also, World Models (except for in their section on learning from dreams) only uses the model as a way to learn good representations for the controller, not as way to train the controller without interacting with the real environment. This off-policy model learning is interesting because World Models basically decouples the representation learning from the actual evaluation and control of the RL agent. The modules used for learning the representation are trained independently of the controller using just rollouts created by a random policy. While using prediction of the next frame conditioned on the action in an environment to learn a good representation is not new, they instead try to predict the representation of the next frame not its pixels, which sidesteps issues of pixelwise loss, though can have issues of excessive dependence on the representation as discussed later. Lastly, they explore an interesting avenue of gradient-free policy optimization: evolutionary methods. This approach sidesteps the need to assign credit from the reward to different actions and parts of the model by just doing black box optimization on the cumulated reward. This too can have its downsides, such as extreme computation expense.

2 World Models

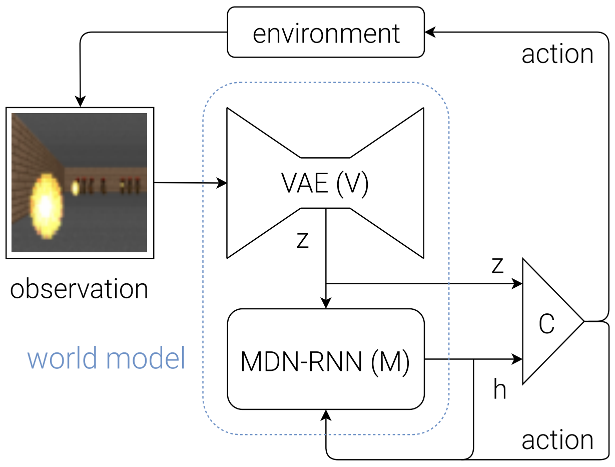

Figure 1: Schematic of World Models from Ha & Schmidhuber (2018)

World Models proceeds in four major stages. First, pixelled frames from 10,000 episodes from the environment are generated from a random policy. These frames along with the actions that led to them are collected. In the second stage a VAE, V, is trained to encode these frames into a low-dimensional vector, z. In the third stage, the an LSTM with a mixture density network, M, is trained to predict the next z given the current z and the current action. The parameters of V and M are updated through SGD using backpropagation. Lastly, the controller is trained online in the environment by encoding the pixels with the pretrained V into z and then passing this z through M to get its hidden state, h. The controller uses z and h to choose the next action. The parameters of the controller are updated by an evolutionary search algorithm called CMA-ES. The fitness of each possible "solution" suggested by the evolutionary algorithm is measurred by averaging the the cumulative reward of the controller from multiple episodes.

3 Methods

I wrote all the code myself using PyTorch, the evolutionary algorithm library, pycma and OpenAI gym. The code is available here:https://github.com/rllabmcgill/final-project-world-modelz

3.1 VAE

For the VAE, I used the exact architecture as the original authors specified in the paper. The VAE was a convolutional autoencoder that featured 4 convolutional layers with relu activation, followed by a fully connected layer and a sampling layer, which was followed by 4 deconvolutional layers with relu. Like the paper, I resized the 96x96 frames from CarRacing to 64x64 before inputting them to the VAE. In the original paper they trained the VAE on 10,000 episodes of the environment with a random policy. Because a CarRacing episode usually ends after 1,000 frames, that amounts 10 million frames. Due to time, I only trained the VAE on 1,000 episodes with one pass through the 1000 frames in each episode (1 million total frames). I trained the VAE with a minibatch size of 128, a learning rate of 0.001 and using the Adam optimizer.

3.2 MDNRNN

For the MDN-RNN, the architecture was not as clearly specified as the VAE, but I tried to reproduce as closely as possible what they said in the paper. I ended up implementing a one layer LSTM with a hidden layer size of 256 and an input size of 35 (32 unit z concatenated with the 3-dimensional action fro CarRacing). The output of the LSTM was a mixture density network, which I parametrized as mentioned in the paper to be a mixture of 5 gaussians. I mostly followed the setup of Graves (2013), which involved inferring a vector of means and variances for each gaussian as well as a mixing coefficient. The mixing coefficients were parametrized as a categorical distribution using a softmax activation function and the variances were constrained to be positive using an exp operation. Each gaussian was modelled to be diagonal, so there were no covariance terms. The loss was the negative log-likelihood of the next z under the output mixture of guassians. The network was trained using teacher forcing for 200 epochs, using all 1 million z-action pairs from the VAE. I used a learning rate of 0.001, Adam optimization and temperature of 1.

3.3 Controller

I used the same architecture as the original paper: a single linear layer, which takes as input, a 288 dimensional vector (the 32-dimensional autoencoded current state plus the 256-dimensional hidden state from the MDN-RNN model). The output was 3-dimensional because an action in CarRacing has three terms, steer, gas and brake. Steer is between -1 and 1, so I used a tanh activation for that, whereas gas and brake are between 0 and 1, so I used a softplus activation for them. The original paper trained the controller using CMA-ES **?** with a population size of 64 for each generation for 2000 generations, where the fitness of a suggested solution involves the average cumulative reward over 16 rollouts with the controller. It is extremely expensive to train something with an evolutionary algorithm. While the authors used a 64 core machine and parallelized the training, I could not get the multicore module in the evolutionary method library to work, so I had to train this part fully serially, which was very computationally expensive. So due to time constraints I could only train the controller for 20 generations with a population size of 8 and using a fitness of average cumulative reward from 8 rollouts. For the z only controller, I trained it for 80 generations with a population size of 64 and (erroneously) using only 1 rollout to evaluate the fitness of a solution.

4 Results

4.1 VAE

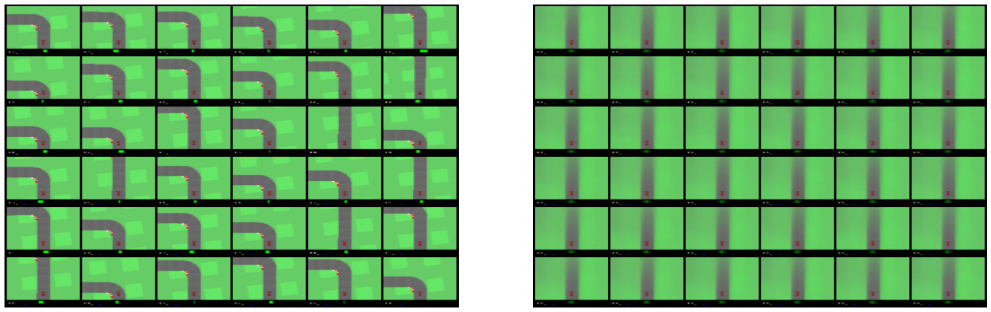
Figure 2: Original (left) and reconstructed (right) images in VAE

The training of the VAE seemed to go fine as it converged to a low loss. By visualizing the images in figure 2, however, we can see that the VAE achieves some form of mode collapse. It learns that it can still have a relatively low loss by just unconditionally reconstructing the same straight road. This behavior persisted even when I shuffled the minibatches of frames (so it wouldn't see 100's of straight road frames before nay curved road). This is very problematic because it seems since the decoder is not really using any information about the current frame, the encoder is possibly not encoding any useful information either. This means the z representation could be just noise with little to no correlation to the actual input. This is a big issue as every part of the World Models depends heavily on this z representation being a good one. Also, this exposes one big downside to training each component separately: the VAE does not know what is important to encode because it does not know what the task of the controller of even the LSTM is, so by being decoupled and just trying to minimize reconstruction error, it can learn an erroneous representation that can mess up the rest of the model.

4.2 MDNRNN

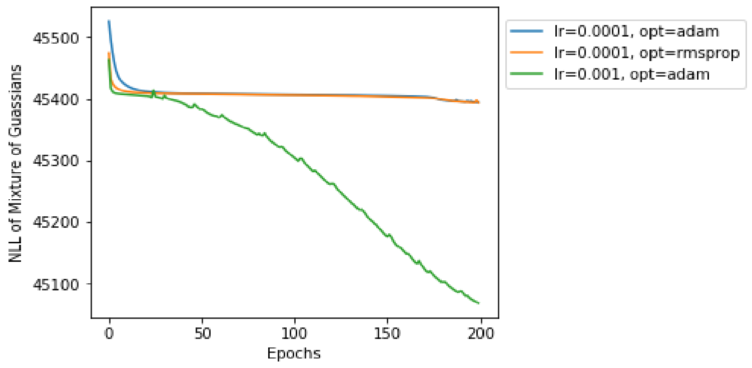

Figure 3: MDN RNN Learning Curve

Training the MDN RNN went seamlessly without a hitch with the best hyperparameters shown above in the learning curve in figure 3. However, the fact that z may have been very noisy may have rendered the encoding of h from the mdn rnn component to be moot.

4.3 Controller

For training the controller, we see in the plots above that the performance from my implementation was nowhere near as good as theirs. This is probably for two reasons; one, for time reasons, I could not use the same hyperparameters as they did in the evolutionary search, so I ran it for a shorter

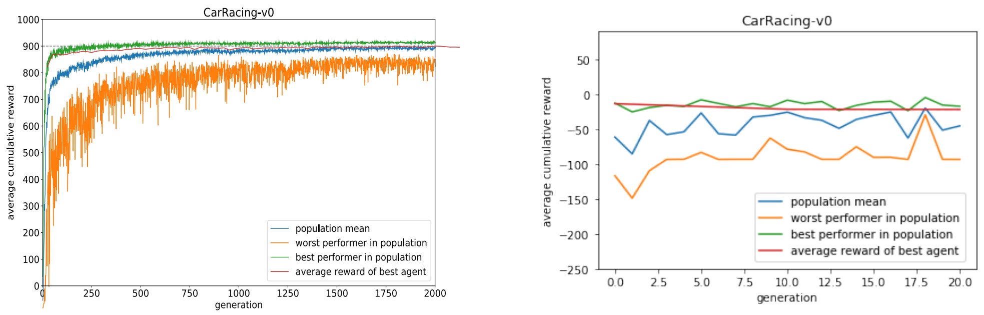

Figure 4: Plots for training of controller. Left is the original authors' training curve. Right is my training curve for controller.

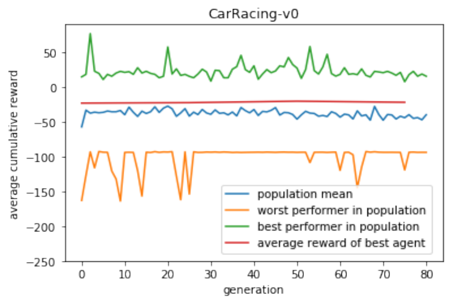

Figure 5: Plot of training for my implementation of the z-only controller.

time and with a smaller "population" size and the fitness estimate was higher variance because it used fewer rollouts to estimate the average cumulative reward. Also, the noisy encoding of z also probably had a large effect here as the controller uses z and h (which is a function of z) as inputs, so if they don't represent the state very well, the controller cannot do a good job and neither can the evolutionary solver.

4.4 Overall Car Racing Results

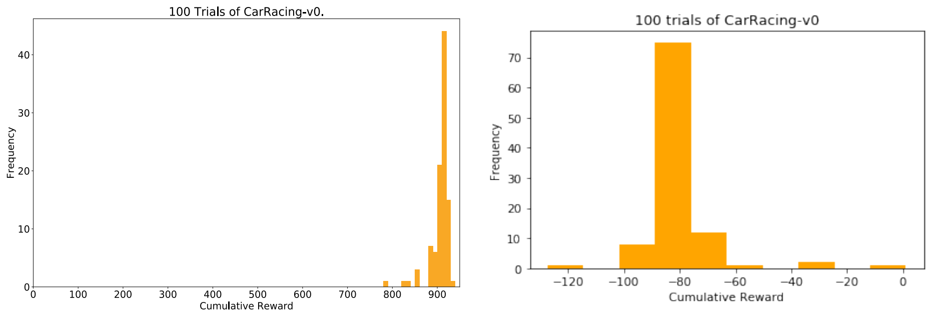

Figure 6: Distribution of cumulative rewards for best overall controller over 100 rollouts. Left is theirs, right is mine

As we can see from the table and plots above, I was nowhere near matching the author's performance. My models might even have done worse than a random policy. Also, we see that for the original author's empirically the full World model is a large improvement over the V only model. This is likely due to the extra infromation h gives the controller. For my implementation, the two models are roughly the same in performance. This likely means that h is providing no extra information and that might be because z provides very little information about the state to begin with.

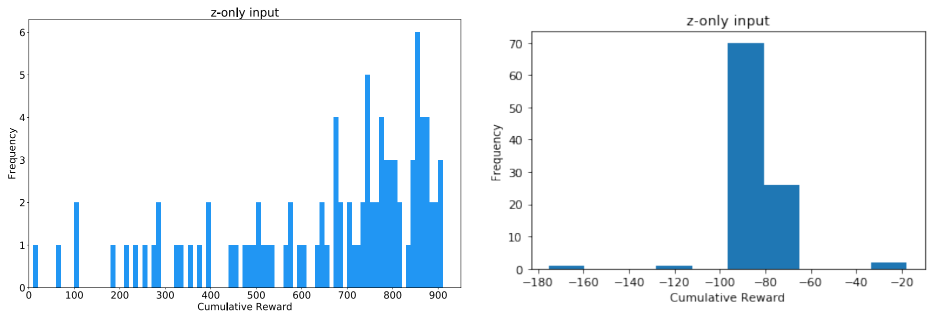

Figure 7: Distribution of cumulative rewards for controller with z-only input over 100 rollouts. Left is theirs, right is mine.

Method	Avg. Score
V model	632 ± 251
V model (mine)	-82.47 ± 14.21
Full World Model	906 ± 21
Full World Model (mine)	-79.84 ± 13.15

Table 1: `CarRacing-v0` scores for their World Model vs. mine

5 Discussion/Conclusion

World Models achieved some impressive results and popularized some interesting ideas for learning representations for RL. However, there were challenges that arose from these approaches that made it hard to match their exact procedure, which resulted in a failure to match their performance. Also, there were some disadvantages to their approaches that may have made the model more susceptible to failure. Namely, the decoupling of the training of the VAE and the MDN RNN from the controller. This makes for a nice setup because the VAE and RNN have their own training signals and do not have to rely on sparse credit signals from the controller and they can be trained offline, which can result in faster online training. However, the VAE does not know the useful things to encode for the task because it is just trained to reconstruct (and minimize a KL term). Thus, if the VAE representation is poor, it ruins the whole model because the RNN depends on a good z representation, so it can learn a good representation for trying to predict the next z, but if the z's are just noise then the RNN learns nothing from the environment and its hidden states' are noise too. Lastly, this messes up the controller because the controller's only window to the state of the environment are the VAE and the RNN, so if these are noise the controller is just making random decisions. This enormous sensitivity to a vanilla VAE is a big issue for World Models. Another painful issue for me was the evolutionary training. I had a lot of trouble getting PyTorch, the rendering in gym to get the pixels, and the multicore module in PyCMA (the evolutionary library) to play nicely together on a gpu with cuda. Even on a cpu, anytime I tried to invoke gym with the PyCMA multicore module, it always failed. Eventually, I had to give up and run serially. However, even serially, I had to run the code in a screen session in an interactive session on a gpu cluster, while taking care to import gym and do some rendering before importing certain libraries like tensorboardX or torchvision in order to prevent crashing of the code.

6 Future Work

In order to further match the results, I would have to do further debugging and tuning of the VAE to make sure it learns salient representations. Perhaps, I should have trained the VAE longer or with frames generated from a pretrained policy. Or maybe using normalizing flows or a disentangling autoencoder would have helped the VAE. Also, I would have to train the evolutionary algorithm for much longer, potentially taking advantage of a multicore setup to speed up the training in order to have any hope of matching the author's results. Potential future work would be trying to learn the controller with policy gradient methods instead of evolutionary ones, like vanilla REINFORCE or actor-critic. Additionally, one could have the gradient be fine-tuned back through the RNN and the VAE too, so they could get some signal from the rewards. Also, it would be interesting to see if this model works on Atari, where there are even smaller "objects" of interest than the car that might give the VAE even more trouble. Giving the VAE the action as input along with the frame would also be an interesting thing to do to coax the VAE into learning a salient representation.

References

Graves, Alex. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

Ha, David and Schmidhuber, Jürgen. World models. *arXiv preprint arXiv:1803.10122*, 2018.