Adversarial Models of Reality (AMoRe)

CS 229: Progress Report in Theory & Reinforcement Learning

Eric Zelikman
ezelikma
Stanford University
ezelikman@stanford.edu

Nathan Schager
nschager
Stanford University
nschager@stanford.edu

Abstract

The *World Models* paper [1] proposed a novel technique to train a game-playing agent by building a compressed representation of the world, learning how that representation evolves with a long short-term memory (LSTM) network, and then using the representation and the LSTMs memory as input to the agent's policy. We extended the algorithm in several ways to improve performance on more complex games, by updating the representation of the world with new trials, applying the existing model to anticipate future events in real-time, encourage exploration, and (for CS236) adversarially improving the quality of generated future frames. Our starting model, as evaluated on *Sonic*, using the code for Retro World Models [2] in Pytorch [3], performed poorly, earning scores around 1,000 on most levels (with 3,000 considered complete) since we avoided using human examples. Our improved model performed better but not incredibly well, coming to an average performance of 3,600 across a number of levels.

1 Introduction

What is the nature of excellence? Is it the envisioning of a task in one's mind over and over again, or perhaps is it incremental improvement over repeated practice? Think of how an athlete trains daily while also envisioning her future success. In this paper, we propose our adversarial learning framework AMoRe to help ascertain whether these ideas can be applied to a reinforcement learning framework.

2 Related Work

The *World Models* paper [1] was groundbreaking in its approach but has much room for improvement. For one, randomly trying many rollouts with a policy to generate dreams requires a lot of play-time in order to be generated accurately and will overfit to tested policies. For another, predicting the future single frames at a time has been shown to lead to compounding errors, as shown in the *Deep multi-scale video prediction beyond mean square error* paper [4]. Qualitatively, this sequential-adverserial tradeoff is part of the difference between something like Google's DeepDream [5] and something like Nvidia's progressively-grown Celebra images [6]. We used OpenAI's Gym [7] setup to test our model in game environments, evaluating our model's performance based on both the ultimate performance and the rate at which it reaches this performance. [8].

3 Dataset and Features

We initially conducted tests using OpenAI's racecar setup, because it has a simple world with a clear distinction between reward and penalty areas. Afterwards, we tested our algorithm in Sonic the Hedgehog levels using OpenAI's Retro Contest dataset. Because the levels in Sonic are long and complex, our method of informing the VAE and LSTM using previous policies helped it pick out salient features over a constantly changing game environment. Sonic levels are also be a good test for the GAN dreams given that its world-space is much more complicated than the racecar setting, and is thus also more likely to benefit from prior

experience. Because different Sonic levels can be used for the training and test sets, this dataset is also be useful to see whether or not our algorithm lends itself well to our method of iterative training across experiments.

4 Methods

Using a generative adversarial network (GAN) consisting of two MLD-LSTMs, a variation of an RNN that generates distributions, we seek to train a discriminator to discern between actual observations and dreams, in order for the generator to generate dreams that are closer to reality and thus allow for more generalizable game-play. The structure of this model is as follows:

1. Perform initial rollout using the JERK algorithm (Just Enough Retained Knowledge)[8]
2. Use β variational auto-encoder (β-VAE) to compress the world representation
3. Train a GAN (consisting of two LSTMs) to generate a plausible future world given a policy, to be compared to the actual effects of the policy
4. Enact the policy with a controller, with a module we call the anticipator intercepting and changing actions that have a poor expected outcome
5. Before a given rollout, use the generator to model some large number of possible policies, and sample one based on performance (e.g. corresponding to Zipf's law on their ranks).
6. Repeat the process, using the generated policy to inform rollouts of the next iteration.

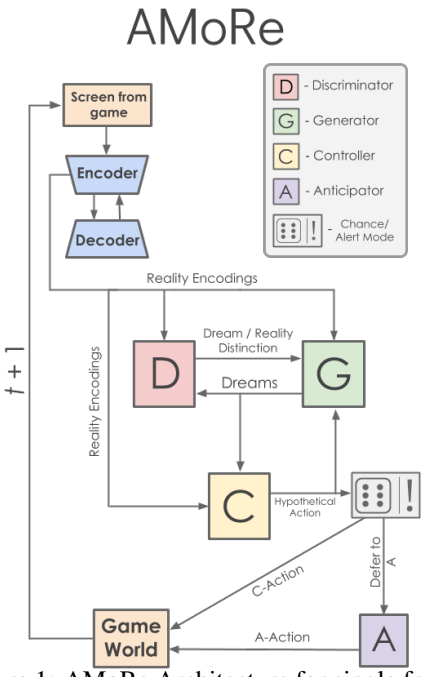

Figure 1: AMoRe Architecture for single frame

Using a GAN to generate and discriminate dream states helps prevent compounding errors in the image generation, as shown in [4]. Practically, repeated probabilistic predictions of the future lend themselves to predicting an unrealistically "average" outcome. This should allow the algorithm to make better predictions by having hallucinations that better reflect the world. Although this part is primarily for CS236, in order to implement this effectively the code was based on the code for the Deep Generative Models paper. Making these two approaches compatible required the observations from the rollouts to be saved as their encoded forms combined with their actions, basically treating them as a highly compressed video frame. We tested to see whether accounting for actions of the model in the future actually helps. In our implementation, we train the discriminator and the generator and then combine the overall loss.

We made the entire process iterative across experiments, sampling using policies from past iterations to perform rollouts for the current iteration. This allowed the VAE to optimize its encoder and decoder to better recognize features that are more useful to scoring well. The tricky part is ensuring that the iterations do not create a policy that is stuck in a local minima, and this will largely depend on how policies are sampled during the rollout phase. Because the CMA-ES generates samples from a Gaussian distribution, one can vary the sigma value to get a larger variety of outcomes depending on your performance needs.

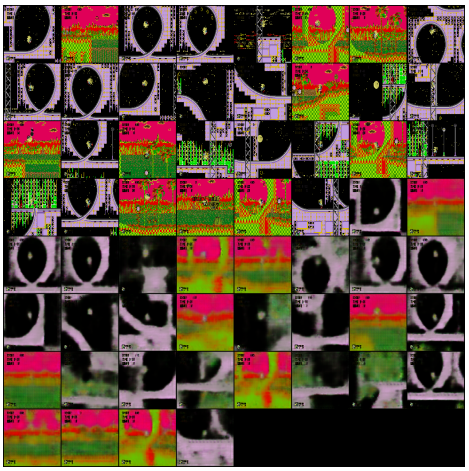

Figure 2: Frames generated by the VAE on bottom corresponding to input frames shown on the top.

4.1 β-VAE

The VAE is used to encode and decode frames. That is, it takes a frame as input and maps it to a distribution in a lower-dimensional space, in our case using a 4-layer convolutional neural network and a 200 dimensional encoding space. Encodings from the frames went to the LSTM, while the decoder was used primarily to train the encoder (and for visualization purposes). It used 5 deconvolutionary layers, as in the original code.

A VAE is normally trained to jointly minimize two things: one, the KL-divergence between the encoded real data and a prior distribution [1]; two, a reconstruction loss corresponding to negative log likelihood of the decoder producing the data given the prior distribution. The β-VAE is a special variant of the VAE that has a parameter β, weighting the KL divergence term to tune a balance between "latent channel capacity and independence constraints with reconstruction accuracy" [9].

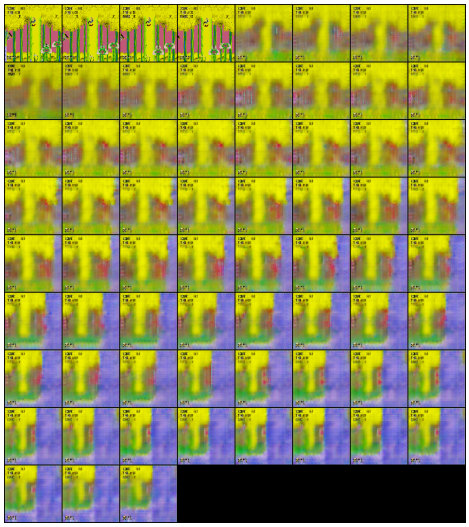

Figure 3: Estimated frames over time from generator. Notice how it can account for the passage of both time and space.

4.2 Adversarial Future Generation (Focus of CS236)

For our adversarial predictor, we used two long-short term memory (LSTM) networks which output a mixture of Gaussian distributions. LSTMs maintain a gated memory representation across different items in a sequence, with its updated memory accessible to the next state. In addition, MDN-RNNs are a type of LSTM, which output a distribution as a mixture of Gaussians rather than a single prediction, with a weight, mean, and variance for each Gaussian. While the choice to treat the discriminator as generating a probability density may seem strange, this was the first approach to convergence that yielded a model which didn't experience exploding or vanishing gradients in the generator. The loss used for the generator was partially inspired by the loss used in a VAE, accounting for both the likelihood of the predicted next frame[2] and an adversarial loss, corresponding to how effectively it tricked the discriminator.

[1] Often a mean-0, variance-1 normal distribution

[2] The loss used in [1]

4.3 Controller

The controller is a simple single-layer neural network with a sigmoid activation: it takes in[3] the encoding of the current frame, as well as the hidden state and previous output of the LSTM (which took in the previous action and frame-encoding) and outputs an action with the same dimension as the action space. The action is $a_t = W_c[z_t h_t] + b_c$ for controller c at timestamp t. We experimented with varying the complexity of this policy, but found that it made it more difficult to update from new levels. It is likely that a joint training model, that is, one which trains on all the levels at the same time, would be more compatible with this approach.

4.4 CMA-ES

CMA-ES, for covariance matrix adaptation evolution strategy, [10] is an evolutionary strategy for optimizing matrices with a few thousand parameters. In our formulation, this algorithm is responsible for optimizing the weights for the controller model. It adjusts a multivariate Gaussian distribution (with covariance between parameters) as a function of performance, making regions of the parameter space with higher performance more likely.

4.4.1 Exploration Reward

To encourage CMA-ES to choose explorative policies, while not included in the final reward, we added an exploration reward for CMA-ES: since variational autoencoders put similar frames closer together, we simply kept the most recent several seconds of frame encodings, adding a small reward corresponding to the minimum Euclidean distance of the encoding of the current frame from the existing frames. One interesting side-effect of this was the tendency for Sonic to take the long route around levels, exploring dead ends and sometimes finding very non-obvious rewards due to it (like hidden ledges that required jumping while going down a waterfall in LabyrinthZone.Act3 of the first sonic). It was also, amusingly, a fan of spectacle, choosing to watch an explosion instead of proceeding. This is reminiscent of [11], where a purely curiosity-driven AI ended up getting addicted to screens displaying novel content.

4.5 Anticipator

The anticipator uses a parameter we call alertness to determine whether or not it will intercept an action from the controller. With a low (Roughly 1/5 probability in every frame), it cyclically generates actions from the controller in response to LSTM-generated future frames to predict the reward of its intended action. If this turns out to be negative, the anticipator puts itself into alert mode, argmaxing its future reward (assuming the controller takes over) over its available moves. The motivation for this sparse application of the anticipator is twofold: first, argmaxing future rewards over actions is greedy and likely to lead to local minima; second, it's fairly computationally expensive and slows the runs if enough parallel games do it at the same time.

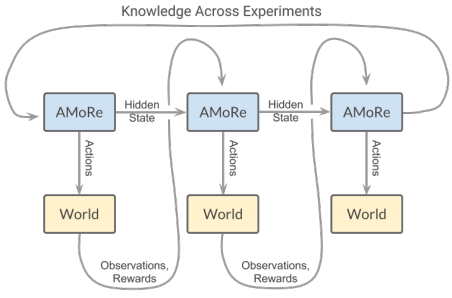

Figure 4: AMoRe Architecture across experiments

Initially, an interesting issue came up: we were at the time letting the anticipator choose moves every time it ran, since it went through all of the possible actions, but often, not moving had the same reward as the controller making its move, since the controller would eventually accomplish whatever the best action accomplished. The consequence was that the anticipator often did nothing when it didn't think it could contribute. We solved this first by discounting future rewards with an impatience term (an exponential drop-off in the value of future rewards). One accidental effect was that the anticipator accidentally created short-sighted loops if in alert mode, like repeatedly hitting a 10-point bumper. We updated alertness to only activate if the controller performed badly.

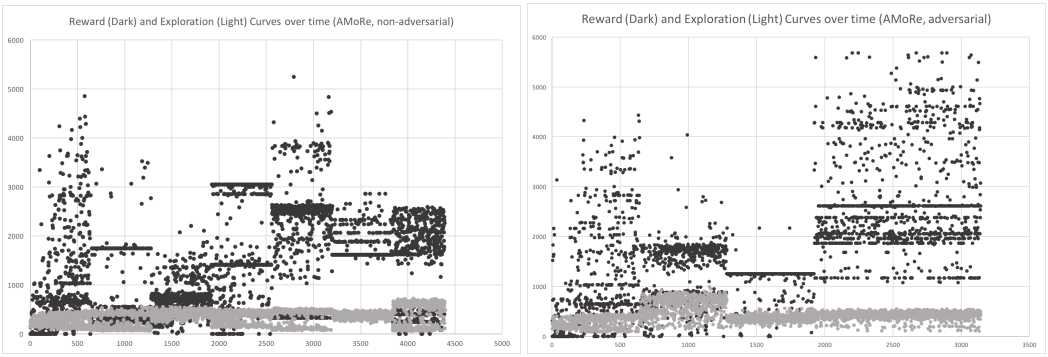

Figure 5: Non-adversarial and adversarial rewards in a sample epoch

4.6 Iterative architecture

For our initial experimental round, we use the JERK method to perform rollouts. The iterative architecture then uses policies generated from the previous experimental round as rollouts for the next experimental round, which should improve the training of both the VAE and the GAN as they will be training on frames that score higher (balanced with some exploration).

5 Experiments/Results/Discussion

We started with the code provided by OpenAI for their World Models experiment but have since modified it to implement our own algorithm. We rewrote the procedure to be iterative and to reuse previous policies during rollouts, as well as adding GAN capacity and some features to action selection in the code. We carried out our experiments in OpenAI-Gym, experimenting on levels sampled from Sonic The Hedgehog 1-3. Our results were successful in that our model outperformed the original paper and OpenAI baseline scores. For this particular challenge, your agent must achieve a score of 3000 to consider the level "solved", and our agent received an average score of 3600 across many levels. Upon inspection, the agent adapted novel behaviors, such as one instance where the agent jumped on a button over and over as the optimal action. Using the GAN model certainly improves the agent's ability to generate estimated future world states, while the iterative model provided superior rollouts for each experiment while also allowing the entire algorithm's architecture to be more in line with how we think science and cognition function.

6 Conclusion/Future Work

Our goal for this paper was to improve upon the original World Models architecture, and in that sense we think we succeeded. We learned that our algorithm could solve the Sonic levels, which opens up some avenues of additional research questions. For one, there are certainly more cognitive principles that could further inspire our algorithm's architecture. Additionally, varying GAN architectures and models could probably enhance the performance further, as could using metalearning to tune parameters and hyperparameters before the actual training begins. Perhaps the most compelling future improvement is a stochastic update to the the LSTM and VAE with every new experiment, alongside the per-epoch training we introduced.

[3] As a single concatenated input vector

References

[1] D. Ha and J. Schmidhuber, "World models," 2018. [Online]. Available: https://worldmodels.github.io

[2] D. Dijan, "Retro contest sonic," https://github.com/dylandjian/retro-contest-sonic, 2018.

[3] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[4] M. Mathieu, C. Couprie, and Y. LeCun, "Deep multi-scale video prediction beyond mean square error," *ArXiv e-prints*, Nov. 2015.

[5] A. Mordvintsev, C. Olah, and M. Tyka, "Inceptionism: Going deeper into neural networks," *Google Research Blog. Retrieved June*, vol. 20, no. 14, p. 5, 2015.

[6] T. Karras, T. Aila, S. Laine, and J. Lehtinen, "Progressive Growing of GANs for Improved Quality, Stability, and Variation," *ArXiv e-prints*, Oct. 2017.

[7] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *CoRR*, vol. abs/1606.01540, 2016. [Online]. Available: http://arxiv.org/abs/1606.01540

[8] A. Nichol, V. Pfau, C. Hesse, O. Klimov, and J. Schulman, "Gotta Learn Fast: A New Benchmark for Generalization in RL," *ArXiv e-prints*, Apr. 2018.

[9] L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinick, S. Mohamed, and A. Lerchner, "-vae: Learning basic visual concepts with a constrained variational framework," 2016.

[10] N. Hansen, "The CMA Evolution Strategy: A Tutorial," *arXiv e-prints*, p. arXiv:1604.00772, Apr. 2016.

[11] Y. Burda, H. Edwards, D. Pathak, A. Storkey, T. Darrell, and A. A. Efros, "Large-scale study of curiosity-driven learning," in *arXiv:1808.04355*, 2018.