Applying World Models to Atari Breakout: A Reinforcement Learning Approach

Alexis Moumtzis, Jack Wang, Harry Yang
University of Southern California
Los Angeles, CA
{moumtzis, yangyiha, pinruwan}@usc.edu

Abstract

This paper presents an implementation and adaptation of the World Models architecture for the Atari Breakout environment. Building upon (Ha and Schmidhuber, 2018) framework, we develop a vision-memory-controller pipeline that learns a compact spatiotemporal representation of the game environment. Our approach employs a Variational Autoencoder (VAE) for spatial compression, a Mixture Density Network combined with a Recurrent Neural Network (MDN-RNN) for temporal dynamics modeling, and a lightweight controller optimized via Evolution Strategies. This work contributes to understanding how model-based reinforcement learning can be applied to classic arcade games and examine the learned visual and dynamical representations in the Atari setting.

1 Introduction

The World Models architecture, introduced by Ha and Schmidhuber (2018), presents an elegant framework for model-based RL. The key insight is to separate the agent into three components: a Vision model (V) that compresses high-dimensional observations into compact latent representations, a Memory model (M) that captures temporal dependencies, and a Controller (C) that maps representations to actions. This separation allows each component to be trained efficiently and enables the agent to learn and plan within its learned model of the world.

Why it matters Reinforcement learning (RL) has achieved remarkable success in mastering complex games, from board games like Go to video games such as Atari titles. However, many successful approaches rely on model-free methods that require extensive interaction with the environment.

Problem&Goal An alternative paradigm, model-based RL, learns an internal model of the environment's dynamics, potentially enabling more sample-efficient learning and better generalization. In this work, we adapt the World Models framework to Atari Breakout as shown in Figure 1, a classic arcade game that presents unique challenges. The environment of Atari Breakout (Breakout-v5) is imported from the OpenAI Gym (Brockman et al., 2016). Unlike the continuous-control tasks originally explored in the World Models paper, Breakout involves discrete actions and requires precise timing to successfully hit the ball and break bricks.

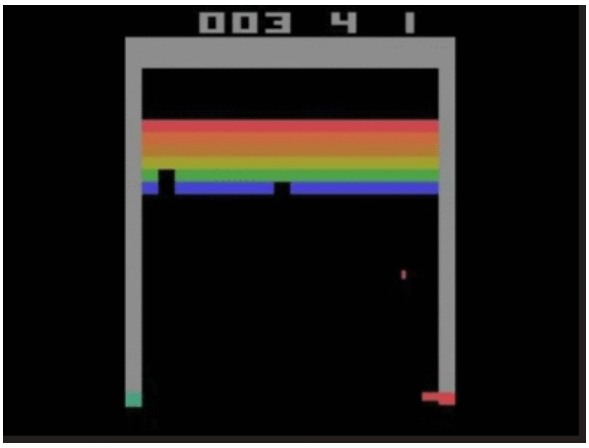

Figure 1: A screenshot of the Atari Breakout gameplay

Our goal is to complete the implementation of the World Models pipeline adapted for Atari Breakout that:

- Design a preprocessing and data-collection pipeline that converts ALE/Breakout-v5 frames to 84×84 grayscale images and records episodes for training the world model.
- Successful training of a convolutional vision model (V) that learns meaningful latent representations.
- Design and implementation of an memory model (M) using MDN-RNN that models temporal dynamics and life-loss events.

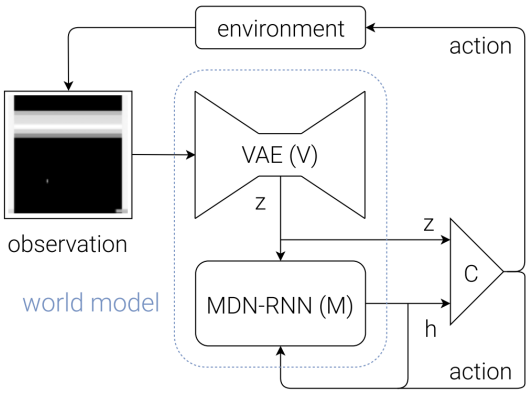

Figure 2: The flow diagram of our Agent model, similar to Ha and Schmidhuber (2018).

- Implementation of a lightweight linear controller (C) with Evolution Strategies (CMA-ES) training framework for policy optimization in both real and learned environments.
- Analysis of challenges specific to discrete-action arcade games.

2 Research Methods

Our implementation follows the three-component architecture of World Models as in Figure 2, with specific adaptations for the Breakout environment. Apart from small changes in hyperparameters, our Agent model is very similar to the one Ha and Schmidhuber (2018) have proposed. At each time step (t), the raw observation is passed through the vision model (V) to obtain a latent vector (z_t). The controller (C) then takes as input the concatenation of (z_t) and the memory model (M)'s hidden state (h_t), and outputs an action vector (a_t) that is applied to the environment. The memory model (M) subsequently uses the current (z_t) and (a_t) to update its hidden state, producing (h_{t+1}), which is used at the next time step ($t+1$).

2.1 Environment Preprocessing

Atari Breakout provides observations as $210 \times 160 \times 3$ RGB frames. We apply standard preprocessing to reduce computational requirements while preserving essential information:

1. **Cropping**: Remove the scoreboard and bottom border, extracting a 160×160 region
2. **Resizing**: Downsample to 84×84 pixels using bilinear interpolation
3. **Grayscale conversion**: Convert using luminance weights $(0.299R + 0.587G + 0.114B)$

The resulting $84 \times 84 \times 1$ frames are normalized to $[0, 1]$ before being fed to the VAE.

2.2 Vision Model (V): VAE

The Vision model is responsible for learning a compact latent representation of each frame. We employ a convolutional Variational Autoencoder (VAE) (Kingma and Welling, 2013). The details of the tensor shapes of the model is showed in Figure 3.

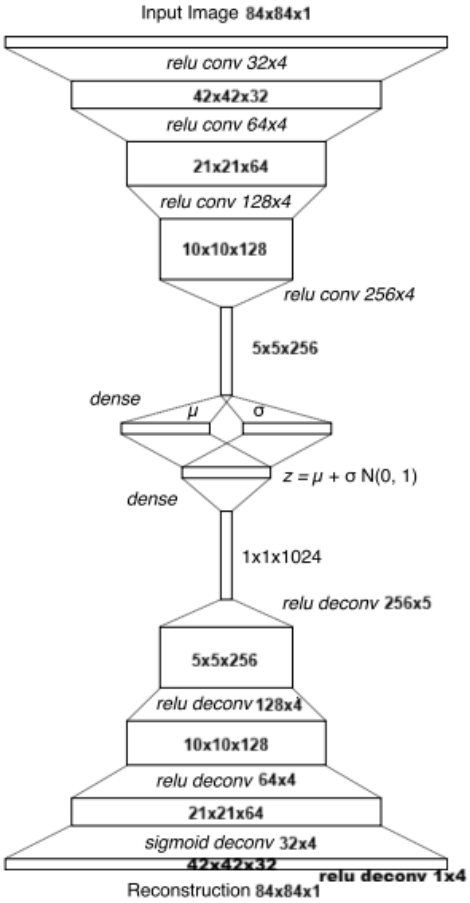

Figure 3: Description of tensor shapes at each layer of ConvVAE.

Encoder The encoder consists of four convolutional layers, each with kernel size 4 and stride 2, followed by a ReLU activation function. These layers progressively downsample the input from 84×84 to a 5×5 feature map with 256 channels. The output is then flattened and passed through two fully connected (FC) layers that compress the 6,400-dimensional feature vector to 1,024 dimensions, and finally to 128 dimensions to produce

the mean (μ) and log-variance ($\log \sigma$) of the latent distribution.

Decoder The decoder mirrors the encoder architecture in reverse. It starts with two fully connected layers that expand the 128-dimensional latent vector to 6,400 dimensions, which is then reshaped into a $256 \times 5 \times 5$ tensor. Four transposed convolutional layers (with kernel size 4 and stride 2) progressively upsample this tensor back to the original 84×84 frame size. A sigmoid activation is applied to the final layer to ensure pixel values are in the range $[0, 1]$. The latent dimension $z \in \mathbb{R}^{128}$ provides sufficient capacity to capture game-relevant features.

Loss Function The VAE minimizes:

$$\mathcal{L} = \mathcal{L}_{\text{recon}} + \mathcal{L}_{\text{KL}} \tag{1}$$

where $\mathcal{L}_{\text{recon}}$ is a weighted reconstruction loss that measures how well the VAE can reconstruct the input image from its latent representation. It ensures that the encoder-decoder pair can accurately recover the original observations after compression.

Given an input frame $x_t \in [0, 1]^{84\times84\times1}$, the encoder produces the parameters of a diagonal Gaussian posterior, $q_\phi(z \mid x_t) = \mathcal{N}(\mu_t, \text{diag}(\sigma_t^2))$, where the network outputs μ_t and $\log \sigma_t^2$ (denoted `mu` and `logvar` in code). The decoder then reconstructs the frame as $\hat{x}_t$ from a sampled latent vector z_t using the reparameterization trick. The reconstruction loss is computed as a per-pixel squared error, $\|x_t - \hat{x}_t\|_2^2$, with an additional weighting mask to emphasize non-background pixels: we define a binary mask $m_{i,j} = \mathbb{I}[x_{t,i,j} > 0.02]$ and set pixel weights $w_{i,j} = 1 + 10m_{i,j}$, yielding

$$\mathcal{L}_{\text{recon}} = \frac{1}{N} \sum_{n=1}^{N} \sum_{i,j,c} w_{i,j}^{(n)} \left(x_{t,i,j,c}^{(n)} - \hat{x}_{t,i,j,c}^{(n)}\right)^2 \tag{2}$$

where N is the batch size. This weighting is important for Breakout-style frames, where large background regions are near zero; up-weighting pixels above the threshold encourages the model to preserve salient objects (e.g., ball, paddle, bricks) rather than minimizing error by predicting mostly background.

The KL term $\mathcal{L}_{\text{KL}}$ regularizes the latent distribution by penalizing deviation from the unit Gaussian prior $p(z) = \mathcal{N}(0, I)$:

$$\mathcal{L}_{\text{KL}} = D_{\text{KL}}(q_\phi(z \mid x_t) \,\|\, \mathcal{N}(0, I)) \tag{3}$$

Then we can furtherly rewrite KL loss using the following equation:

$$\mathcal{L}_{\text{KL}} = \frac{1}{2} \sum_{d=1}^{z} \left(\mu_{t,d}^2 + \sigma_{t,d}^2 - 1 - \log \sigma_{t,d}^2\right) \tag{4}$$

To avoid posterior collapse and to maintain a minimum information capacity in the latent code, we apply a KL tolerance floor by clamping $\mathcal{L}_{\text{KL}}$ to be at least `kl_tolerance` $\cdot$ `z_size` before averaging over the batch. The final objective minimized during training is the sum of these two components, $\mathcal{L} = \mathcal{L}_{\text{rec}} + \mathcal{L}_{\text{KL}}$, balancing faithful reconstructions with a smooth, sampleable latent space.

A demonstration of our trained VAE on an example frame is showed in Figure 4

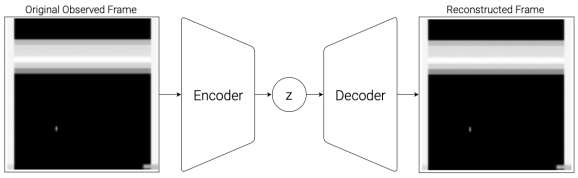

Figure 4: An example of the input and output of our VAE

2.3 Memory Model (M): MDN-RNN

The Memory model captures temporal dynamics in the latent space using a Mixture Density Network combined with an LSTM (Bishop, 1994; Graves, 2013), modeling $p(z_{t+1}|z_t, a_t, h_t)$.

While it is the role of the V model to compress what the agent sees at each time frame, we also want to compress what happens over time. The Memory model serves as a predictive model of the future latent vectors that V is expected to produce. An LSTM with 512 hidden units receives the concatenation of the current latent vector $z_t \in \mathbb{R}^{128}$ and the one-hot encoded action $a_t \in \mathbb{R}^4$ at each timestep, maintaining a hidden state h_t that encodes the agent's memory of past observations and actions.

Since the Breakout environment contains stochastic elements—such as variable ball trajectories and brick configurations—we train our RNN to output a probability density function $p(z_{t+1})$ rather than a deterministic prediction. Following Bishop (1994) and Graves (2013), we model

this distribution as a mixture of $K = 5$ Gaussian distributions, where the RNN outputs mixture weights $\pi_{t,d,k}$ via softmax, means $\mu_{t,d,k}$, and log standard deviations $\log \sigma_{t,d,k}$ for each latent dimension $d \in \{1, \ldots, 128\}$. More specifically, the MDN-RNN models $p(z_{t+1}|a_t, z_t, h_t)$, where a_t is the action taken at time t and h_t is the hidden state of the RNN at time t can be written as:

$$\prod_{d=1}^{128} \sum_{k=1}^{K} \pi_{t,d,k} \mathcal{N}\left(z_{t+1,d};\, \mu_{t,d,k},\, \sigma^2_{t,d,k}\right) \quad (5)$$

The mixture density formulation allows the model to capture multimodal distributions in the latent space, which is essential for environments where multiple plausible future states exist given the current state and action.

Loss Function Training minimizes the negative log-likelihood of the target latents z_{t+1} under this mixture distribution (implemented with a numerically stable $\log \sum \exp$ aggregation across mixture components), averaged across latent dimensions and time. As a Breakout-specific adaptation, M also includes a single binary head that predicts the probability of losing a life at the next step, $p(\text{life\_loss}_{t+1} = 1 \mid z_t, a_t, h_t)$, trained with binary cross-entropy on the logits. The final objective combines both terms:

$$\mathcal{L}_M = \mathcal{L}_{\text{MDN}} + \lambda \mathcal{L}_{\text{life}}, \qquad \lambda = 5.0, \quad (6)$$

encouraging accurate stochastic predictions in latent space while simultaneously learning life-loss dynamics. During imagination ("dreaming"), the MDN output is sampled by first drawing a mixture component and then sampling from the corresponding Gaussian (optionally with a temperature parameter), producing z_{t+1} and an updated recurrent state to roll forward in time.

$\lambda = 5.0$ balances the two objectives. The hidden state h_t thus becomes a compact learned representation of both the spatial state (through z_t) and the temporal context of the game, including an implicit understanding of remaining lives and risk assessment.

2.4 Controller (C): Linear Policy

The controller C is implemented as a lightweight linear policy network that maps the combined latent representation to action probabilities. At each timestep t, the controller receives a feature vector constructed by concatenating the VAE latent code z_t (dimension 128) and the MDN-RNN hidden state h_t (dimension 512), yielding an input of size 640. This feature vector is then transformed via a simple linear layer: logits $= W \cdot [z_t, h_t] + b$, where $W \in \mathbb{R}^{640 \times 4}$ and $b \in \mathbb{R}^4$ are the trainable parameters, and 4 represents Breakout's action space size. The logits are converted to action probabilities using the softmax function, and actions are sampled stochastically during training or selected greedily during evaluation.

We train the controller using Covariance Matrix Adaptation Evolution Strategy (CMA-ES) (Hansen, 2016)(Hansen and Ostermeier, 2001), a derivative-free optimization algorithm well-suited for reinforcement learning problems with compact policy networks. The training process is distributed across 8 workers using MPI, where the master process maintains the ES population and coordinates parameter evaluation. At each generation, CMA-ES generates a population of 64 candidate parameter vectors by perturbing the current mean with Gaussian noise. Each worker evaluates multiple candidates in parallel by running episodes in the Atari environment, using the frozen VAE encoder and MDN-RNN to extract features. The fitness of each candidate is computed as the mean reward over 16 episodes, and these fitness values are aggregated and returned to the master process. CMA-ES then updates its internal covariance matrix and mean vector to bias the search toward high-performing regions of the parameter space. We employ antithetic sampling to reduce variance and periodically evaluate the current best controller on longer episodes to track true performance and prevent overfitting to the training distribution.

3 Training and Evaluation

3.1 VAE Training Results

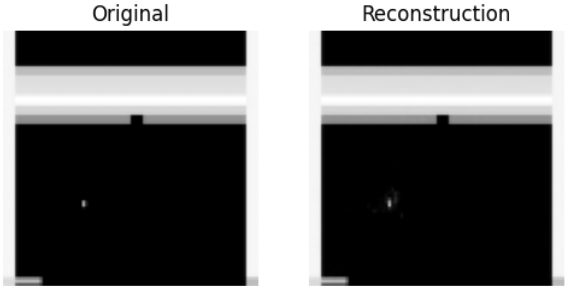

Figure 5: Appearing artifacts near the fast moving ball

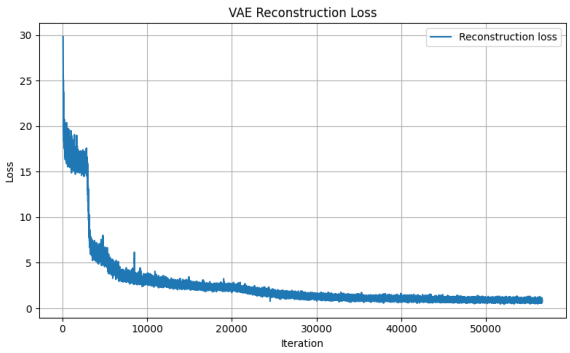

Figure 6: Our VAE model's reconstruction loss during the full training process

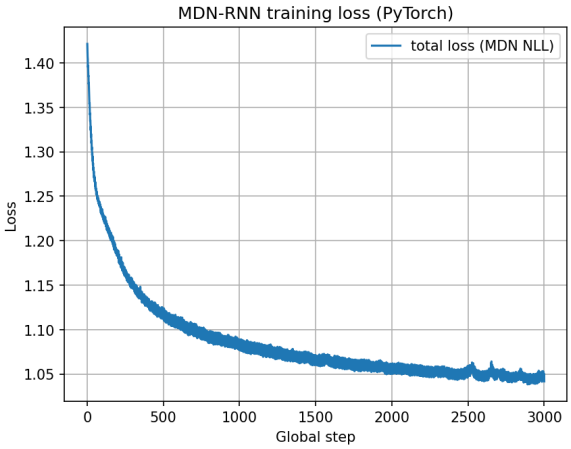

Figure 7: MDN-RNN training loss (negative log-likelihood) over 600 epochs

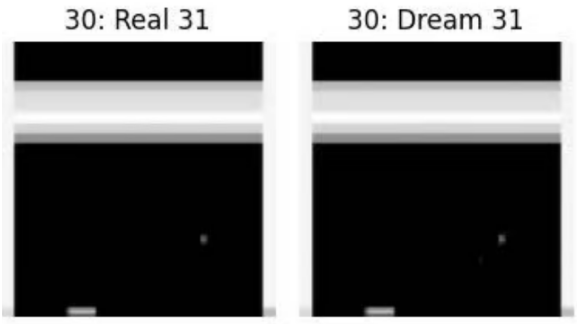

Figure 8: The memory model M predicts the 31st step given the 30th step

First we collect 188623 grayscale game frames from Atari Breakout by running a random policy and saving the preprocessed 84×84×1 observations to disk as .npz files. Each file contains a sequence of observations of uint8 images in [0,255] and the corresponding discrete actions. The frames are divided into batches, each one with 100 frames. The model V is train through 30 epochs, with an learning rate of 10^{-4}, over around 56k training iterations. After training for 30 minutes using the NVIDIA GeForce RTX 4060 Laptop GPU, our VAE model with 15 million parameters and 128-dimensional latent space successfully captures meaningful structure with smooth interolation between nearby game states. As shown in Figure 4, the model acurrately capture the ball and paddle positions. The brick configurations are well-preserved with minor detail loss. In areas where the ball is doing a rapid motion, some artifacts would appear as shown in Figure 5.

After the full training process of the V model, the reconstruction loss has lowered to less then 1 from 30 as shown in Figure 6, symbolise the success of our VAE model training process.

3.2 MDN-RNN Training Results

After encoding all collected frames through the trained VAE, we obtain latent sequences of $(\mu_t, \log \sigma_t^2, a_t)$ tuples. The MDN-RNN is trained on these pre-encoded sequences by sampling $z_t \sim \mathcal{N}(\mu_t, \sigma_t^2)$ and constructing batches of 64 contiguous chunks with sequence length 200 from individual episodes. The model is trained for 600 epochs with an initial learning rate of 10^{-3} exponentially decayed to 10^{-5}, using gradient clipping with maximum norm 1.0 for stability. Our MDN-RNN with a 512-unit LSTM successfully learns to predict future latent states using a mixture of 5 Gaussians.

As shown in Figure 7, the training loss converges smoothly from approximately 1.40 to 1.05, demonstrating successful learning of temporal dynamics as shown in Figure 8. We evaluate the model using two approaches: teacher-forcing mode, where true latents are provided at each step, shows accurate one-step predictions that preserve object positions and brick configurations when decoded; open-loop dreaming, where the model recursively feeds its own predictions, maintains plausible physics for short horizons but exhibits natural divergence over extended rollouts(Ha and Schmidhuber, 2018).

4 Conclusions

Unfortunately, our world model failed to adapt the Atari Breakout environment and failed to move the paddle towards the ball to keep playing. The model places the paddle to the very left, as the ball will always flies towards that area at the very beginning of a life, ensuring a guaranteed block break at the every start of a life. However, it does not learn that keep hitting the ball again and again will gain more

points so the model believes that always putting the paddle on the very left, not moving it, and gaining one point per life is the optimum solution to play this game.

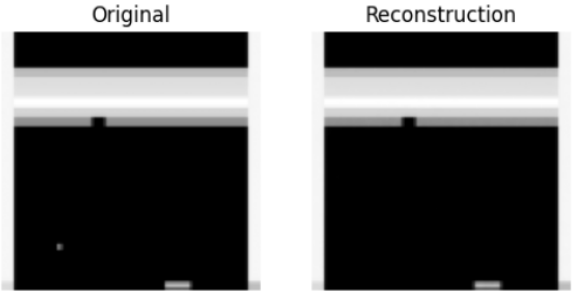

Figure 9: VAE failed to reconstruct the figure of the ball in the output frame

4.1 Limitations

Since the pixel of the ball is very small, by using the VAE to encode and decode the original frame sometime causes the ball to not constructed in the reconstructed output as shown in Figure 9.

4.2 Future Work

Since the game Atari Breakout is too hard to actually learned by the world model, we decided to move back to the car racing game (Ha and Schmidhuber, 2018) and implement our own world model from scratch.

Acknowledgments

We thank the TA Sampad Bhusan Mohanty and Professor Victor Adamchik for guidance throughout this project. We also thank the original World Models authors, (Ha and Schmidhuber, 2018), whose work inspired this implementation. It has been a wonderful journey learning CSCI 467.

References

Christopher M Bishop. 1994. Mixture density networks.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. Openai gym. *arXiv preprint arXiv:1606.01540*.

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

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

Nikolaus Hansen. 2016. The cma evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772*.

Nikolaus Hansen and Andreas Ostermeier. 2001. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195.

Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.