Learning to Simulate Dynamic Environments with GameGAN Supplementary Material

We provide detailed descriptions of the model architecture (Section A), training scheme (Section B), and additional figures (Section C).

A. Model Architecture

We provide architecture details of each module described in Section 3. We adopt the following notation for describing modules:

Conv2D(a,b,c,d): 2D-Convolution layer with output channel size **a**, kernel size **b**, stride **c**, and padding size **d**.

Conv3D(a,b,c,d,e,f,g): 3D-Convolution layer with output channel size **a**, temporal kernel size **b**, spatial kernel size **c**, temporal stride **d**, spatial stride **e**, temporal padding size **f**, and spatial padding size **g**.

T.Conv2D(a,b,c,d,e): Transposed 2D-Convolution layer with output channel size **a**, kernel size **b**, stride **c**, padding size **d**, and output padding size **e**.

Linear(a): Linear layer with output size **a**.

Reshape(a): Reshapes the input tensor to output size **a**.

LeakyReLU(a): Leaky ReLU function with slope **a**.

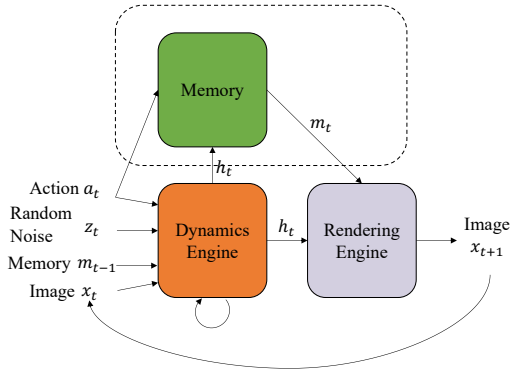


Figure 1: Overview of GameGAN: The *dynamics engine* takes a_t, z_t, and x_t as input to update the hidden state at time t. Optionally, it can write to and read from the external *memory module* (in the dashed box). Finally, the *rendering engine* is used to decode the output image x_{t+1}. The whole system is trained end-to-end. All modules are neural networks.

A.1. Dynamics Engine

The input action $a \sim \mathcal{A}$ is a one-hot encoded vector. For Pacman environment, we define $\mathcal{A} = \{left, right, up, down, stay\}$, and for the counterpart actions $\hat{a}$ (see Section 3.4.2), we define $\hat{left} = right$, $\hat{up} = down$, and vise versa. The images x have size $84 \times 84 \times 3$. For VizDoom, $\mathcal{A} = \{left, right, stay\}$, and $\hat{left} = right$, $\hat{right} = left$. The images x have size $64 \times 64 \times 3$.

At each time step t, a 32-dimensional stochastic variable z_t is sampled from the standard normal distribution $\mathcal{N}(0; I)$. Given the history of images $x_{1:t}$ along with a_t and z_t, GameGAN predicts the next image x_{t+1}.

For completeness, we restate the computation sequence of the dynamics engine (Section 3.1) here.

$$v_t = h_{t-1} \odot \mathcal{H}(a_t, z_t, m_{t-1}) \tag{1}$$

$$s_t = \mathcal{C}(x_t) \tag{2}$$

$$i_t = \sigma(W^{iv} v_t + W^{is} s_t), f_t = \sigma(W^{fv} v_t + W^{fs} s_t), \\ o_t = \sigma(W^{ov} v_t + W^{os} s_t) \tag{3}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tanh(W^{cv} v_t + W^{cs} s_t) \tag{4}$$

$$h_t = o_t \odot \tanh(c_t) \tag{5}$$

where h_t, a_t, z_t, c_t, x_t are the hidden state, action, stochastic variable, cell state, image at time step t. $\odot$ denotes the hadamard product.

The hidden state of the dynamics engine is a 512-dimensional vector. Therefore, $h_t, o_t, c_t, f_t, i_t, o_t \in \mathbb{R}^{512}$.

$\mathcal{H}$ first computes embeddings for each input. Then it concatenates and passes them through two-layered MLP: [Linear(512),LeakyReLU(0.2),Linear(512)]. h_{t-1} can also go through a linear layer before the hadamard product in eq.13, if the size of hidden units differ from 512.

$\mathcal{C}$ is implemented as a 5 (for 64×64 images) or 6 (for 84×84 images) layered convolutional networks, followed by a linear layer:

Pacman	VizDoom
Conv2D(64, 3, 2, 0)	Conv2D(64, 4, 1, 1)
LeakyReLU(0.2)	LeakyReLU(0.2)
Conv2D(64, 3, 1, 0)	Conv2D(64, 3, 2, 0)
LeakyReLU(0.2)	LeakyReLU(0.2)
Conv2D(64, 3, 2, 0)	Conv2D(64,3, 2, 0)
LeakyReLU(0.2)	LeakyReLU(0.2)
Conv2D(64, 3, 1, 0)	Conv2D(64, 3, 2, 0)
LeakyReLU(0.2)	LeakyReLU(0.2)
Conv2D(64, 3, 2, 0)	Reshape(7*7*64)
LeakyReLU(0.2)	Linear(512)
Reshape(8*8*64)	
Linear(512)	

A.2. Memory Module

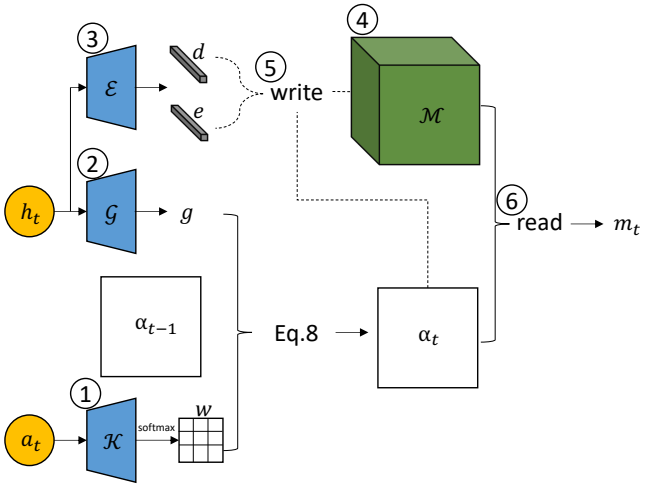


Figure 2: *Memory Module*. Descriptions for each numbered circle are provided in the text.

For completeness, we restate the computation sequence of the memory module (Section 3.2) here.

$$w = \text{softmax}(\mathcal{K}(a_t)) \in \mathbb{R}^{3\times 3} \tag{6}$$

$$g = \mathcal{G}(h_t) \in \mathbb{R} \tag{7}$$

$$\alpha_t = g \cdot \text{Conv2D}(\alpha_{t-1}, w) + (1-g)\cdot \alpha_{t-1} \tag{8}$$

$$\mathcal{M} = \text{write}(\alpha_t, \mathcal{E}(h_t), \mathcal{M}) \tag{9}$$

$$m_t = \text{read}(\alpha_t, \mathcal{M}) \tag{10}$$

See Figure 2 for each numbered circle.

(1) $\mathcal{K}$ is a two-layered MLP that outputs a 9 dimensional vector which is softmaxed and reshaped to $3\times$ 3 kernel w: [Linear(512), LeakyReLU(0.2), Linear(9), Softmax(), Reshape(3,3)].

(2) $\mathcal{G}$ is a two-layered MLP that outputs a scalar value followed by the sigmoid activation function such that $g \in [0, 1]$: [Linear(512),LeakyReLU(0.2),Linear(1),Sigmoid()].

(3) $\mathcal{E}$ is a one-layered MLP that produces an erase vector $e \in \mathbb{R}^{512}$ and an add vector $d \in \mathbb{R}^{512}$: [Linear(1024), split(512)], where Split(512) splits the 1024 dimensional vector into two 512 dimensional vectors. e additionally goes through the sigmoid activation function.

(4) Each spatial location in the memory block $\mathcal{M}$ is initialized with 512 dimensional random noise $\sim N(0, I)$. For computational efficiency, we use the block width and height $N = 39$ for training and $N = 99$ for downstream tasks. Therefore, we use $39 \times 39 \times 512$ blocks for training, and $99 \times 99 \times 512$ blocks for experiments. Note that the shift-based memory module architecture allows any arbitrarily sized blocks to be used at test time.

(5) write operation is implemented similar to the Neural Turing Machine [4]. For each location $\mathcal{M}^i$, write computes:

$$\mathcal{M}^i = \mathcal{M}^i(1 - \alpha_t^i \cdot e) + \alpha_t^i \cdot d \tag{11}$$

where i denotes the spatial x, y coordinates of the block $\mathcal{M}$. e is a sigmoided vector which erases information from $\mathcal{M}^i$ when $e = \mathbf{1}$, and d writes new information to $\mathcal{M}^i$. Note that if the scalar α_t^i is 0 (*i.e.* the location is not being attended), the memory content $\mathcal{M}^i$ does not change.

(6) read operation is defined as:

$$m_t = \sum_{i=0}^{N\times N} \alpha_t^i \cdot \mathcal{M}^i \tag{12}$$

where $\mathcal{M}^i$ denotes the memory content at location i.

A.3. Rendering Engine

For the simple rendering engine of GameGAN -M, we first pass the hidden state h_t to a linear layer to make it a $7 \times 7 \times 512$ tensor, and pass it through 5 transposed convolution layers to produce the 3-channel output image x_{t+1}:

Pacman	VizDoom
Linear(512*7*7)	Linear(512*7*7)
LeakyReLU(0.2)	LeakyReLU(0.2)
Reshape(7, 7, 512)	Reshape(7, 7, 512)
T.Conv2D(512, 3, 1, 0, 0)	T.Conv2D(512, 4, 1, 0, 0)
LeakyReLU(0.2)	LeakyReLU(0.2)
T.Conv2D(256, 3, 2, 0, 1)	T.Conv2D(256, 4, 1, 0, 0)
LeakyReLU(0.2)	LeakyReLU(0.2)
T.Conv2D(128, 4, 2, 0, 0)	T.Conv2D(128, 5, 2, 0, 0)
LeakyReLU(0.2)	LeakyReLU(0.2)
T.Conv2D(64, 4, 2, 0, 0)	T.Conv2D(64, 5, 2, 0, 0)
LeakyReLU(0.2)	LeakyReLU(0.2)
T.Conv2D(3, 3, 1, 0, 0)	T.Conv2D(3, 4, 1, 0, 0)

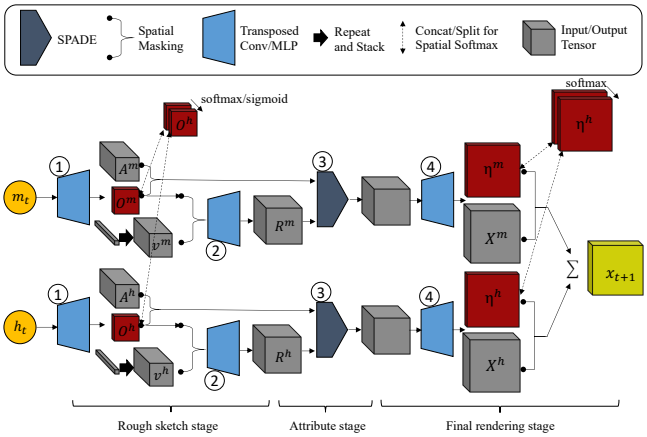


Figure 3: *Rendering engine* for disentangling static and dynamic components. Descriptions for each numbered circle are provided in the text.

For the specialized disentangling rendering engine that takes a list of vectors $\mathbf{c} = \{c^1, ..., c^K\} = \{m_h, h_t\}$ as input, the followings are implemented. See Figure 3 for each numbered circle.

① First, $A^k \in \mathbb{R}^{H_1 \times H_1 \times D_1}$ and $O^k \in \mathbb{R}^{H_1 \times H_1 \times 1}$ are obtained by passing c^k to a linear layer to make it a $3 \times 3 \times 128$ tensor, and the tensor is passed through two transposed convolution layers with filter sizes 3 to produce $\mathbb{R}^{7 \times 7 \times 32+1}$ tensor (hence, $H_1 = 7, D_1 = 32$):

Pacman & VizDoom
Linear(3*3*128)
Reshape(3,3,128)
LeakyReLU(0.2)
T.Conv2D(512, 3, 1, 0, 0)
LeakyReLU(0.2)
T.Conv2D(32+1, 3, 1, 0, 0)

We split the resulting tensor channel-wise to get A^k and O^k. v^k is obtained from running c^k through one-layered MLP [Linear(32),LeakyReLU(0.2)] and stacking it across the spatial dimension to match the size of A^k.

② The rough sketch tensor R^k is obtained by passing v^k masked by O^k (which is either spatially softmaxed or sigmoided) through two transposed convolution layers:

Pacman	VizDoom
T.Conv2D(256, 3, 1, 0, 0)	T.Conv2D(256, 3, 1, 0, 0)
LeakyReLU(0.2)	LeakyReLU(0.2)
T.Conv2D(128, 3, 2, 0, 1)	T.Conv2D(128, 3, 2, 1, 0)

③ We follow the architecture of SPADE [9] with instance normalization as the normalization scheme. The attribute map A^k masked by O^k is used as the semantic map that produces the parameters of the SPADE layer, γ and β.

④ The normalized tensor goes through two transposed convolution layers:

Pacman	VizDoom
T.Conv2D(64, 4, 2, 0, 0)	T.Conv2D(64, 3, 2, 1, 0)
LeakyReLU(0.2)	LeakyReLU(0.2)
T.Conv2D(32, 4, 2, 0, 0)	T.Conv2D(32, 3, 2, 1, 0)
LeakyReLU(0.2)	LeakyReLU(0.2)

From the output tensor of the above, η^k is obtained with a single convolution layer [Conv2D(1, 3, 1)] and then is concatenated with other components for spatial softmax. We also experimented with concatenating η and passing them through couple of 1×1 convolution layers before the softmax, and did not observe much difference. Similarly, X^k is obtained with a single convolution layer [Conv2D(3, 3, 1)].

A.4. Discriminators

There are several discriminators used in GameGAN . For computational efficiency, we first get an encoding of each frame with a shared encoder:

Pacman	VizDoom
Conv2D(16, 5, 2, 0)	Conv2D(64, 4, 2, 0)
BatchNorm2D(16)	BatchNorm2D(64)
LeakyReLU(0.2)	LeakyReLU(0.2)
Conv2D(32, 5, 2, 0)	Conv2D(128, 3, 2, 0)
BatchNorm2D(32)	BatchNorm2D(128)
LeakyReLU(0.2)	LeakyReLU(0.2)
Conv2D(64, 3, 2, 0)	Conv2D(256, 3, 2, 0)
BatchNorm2D(64)	BatchNorm2D(256)
LeakyReLU(0.2)	LeakyReLU(0.2)
Conv2D(64, 3, 2, 0)	Conv2D(256, 3, 2, 0)
BatchNorm2D(64)	BatchNorm2D(256)
LeakyReLU(0.2)	LeakyReLU(0.2)
Reshape(3, 3, 64)	Reshape(3, 3, 256)

Single Frame Discriminator The job of the single frame discriminator is to judge if the given single frame is realistic or not. We use two simple networks for the patch-based (D_{patch}) and the full frame (D_{full}) discriminators:

D_{patch}	D_{full}
Conv2D(dim, 2, 1, 1)	Conv2D(dim, 2, 1, 0)
BatchNorm2D(dim)	BatchNorm2D(dim)
LeakyReLU(0.2)	LeakyReLU(0.2)
Conv2D(1, 1, 2, 1)	Conv2D(1, 1, 2, 1)

where dim is 64 for Pacman and 256 for VizDoom. D_{patch} gives 3×3 logits, and D_{full} gives a single logit.

Action-conditioned Discriminator We give three pairs to the action-conditioned discriminator D_{action}: (x_t, x_{t+1}, a_t), $(x_t, x_{t+1}, \bar{a}_t)$ and $(\hat{x}_t, \hat{x}_{t+1}, a_t)$. x_t denotes the real image, $\hat{x}_t$ the generated image, and $\bar{a}_t \in \mathcal{A}$ a negative action which is sampled such that $\bar{a}_t \neq a_t$. The job of the discriminator is to judge if two frames are consistent with respect to the given action. First, we get an embedding

vector for the one-hot encoded action through an embeddinig layer [Linear(dim)], where $dim = 32$ for Pacman and $dim = 128$ for VizDoom. Then, two frames x_t, x_{t+1} are concatenated channel-wise and merged with a convolution layer [Conv2D(dim, 3, 1, 0), BatchNorm2D(dim), LeakyReLU(0.2), Reshape(dim)]. Finally, the action embedding and merged frame features are concatenated together and fed into [Linear(dim), BatchNorm1D(dim), LeakyReLU(0.2), Linear(1)], resulting in a single logit.

Temporal Discriminator The temporal discriminator $D_{temporal}$ takes several frames as input and decides if they are a real or generated sequence. We implement a hierarchical temporal discriminator that outputs logits at several levels. The first level concatenates all frames in temporal dimension and does:

Conv3D(64, 2, 2, 1, 1, 0, 0)
BatchNorm3D(64)
LeakyReLU(0.2)
Conv3D(128, 2, 2, 1, 1, 0, 0)
BatchNorm3D(128)
LeakyReLU(0.2)

The output tensor from the above are fed into two branches. The first one is a single layer [Conv3D(1, 2, 1, 2, 1, 0, 0)] that produces a single logit. Hence, this logit effectively looks at 6 frames and judges if they are real or not. The second branch uses the tensor as an input to the next level:

Conv3D(256, 3, 1, 2, 1, 0, 0)
BatchNorm3D(256)
LeakyReLU(0.2)

Now, similarly, the output tensor is fed into two branches. The first one is a single layer [Conv3D(1, 2, 1, 1, 1, 0, 0)] producing a single logit that effectively looks at 18 frames. The second branch uses the tensor as an input to the next level:

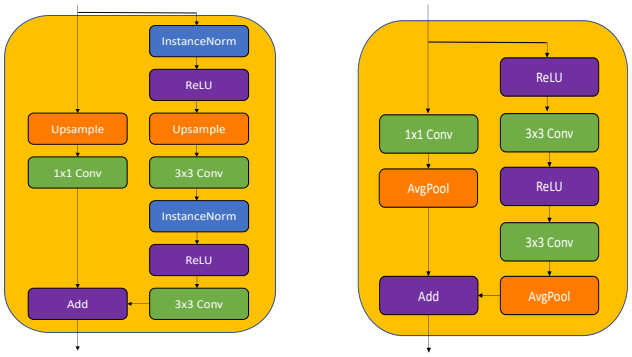
Conv3D(512, 3, 1, 2, 1, 0, 0)
BatchNorm3D(512)
LeakyReLU(0.2)
Conv3D(1, 3, 1, 1, 1, 0, 0)

which gives a single logit that has an temporal receptive field size of 32 frames.

The Pacman environment uses two levels (up to 18 frames) and VizDoom uses all three levels.

A.5. Adding More Capacity

Both the rendering engine and discriminator described in A.3 and A.4 consist of only a few linear and convolutional layers which can limit GameGAN 's expressiveness. Motivated by recent advancements in image generation GANs [1, 9], we experiment with having higher capacity residual blocks. Here, we highlight key differences compared



(a) (b)

Figure 4: Residual blocks for the higher capacity model, modified and redrawn from Figure 15 of [1]. (a) A block for rendering engine, (b) A block for discriminator.

to the previous sections. The code will be released for reproducibility.

Rendering Engine: The convolutional layers in rendering engine are replaced by residual blocks described in Figure 4a. The residual blocks follow the design of [1] with the difference being that batch normalization layers [5] are replaced with instance normalization layers [11]. For the specialized rendering engine, having only two object types as in A.3 could also be a limiting factor. We can easily add more components by adding to the list of vectors (for example, let $\mathbf{c} = \{m_h, h_t^1, h_t^2, ...h_t^n\}$ where $h_t = concat(h_t^2, ...h_t^n)$) as the architecture is general to any number of components. However, this would result in $length(\mathbf{c})$ number of decoders. We relax the constraint for dynamic elements by letting $v^k = MLP(h_t) \in \mathbb{R}^{H_1 \times H_1 \times 32}$ rather than stacking v^k across the spatial dimension.

Discriminator: Increasing the capacity of rendering engine alone would not produce better quality images if the capacity of discriminators is not enough such that they can be easily fooled. We replace the shared encoder of discriminators with a stack of residual blocks shown in Figure 4b. Each frame fed through the new encoder results in a $N \times N \times D$ tensor where $N = 4$ for VizDoom and $N = 5$ for Pacman. The single frame discriminator is implemented as [Sum(N,N), Linear(1)] where Sum(N,N) sums over the $N \times N$ spatial dimension of the tensor. The action-conditioned and temporal discriminators use similar architectures as in A.4.

Figure 5 shows rollouts on Pacman trained with the higher capacity GameGAN model. It clearly shows that the higher capacity model can produce more realistic sharp images. Consequently, it would be more suitable for future works that involve simulating high fidelity real-world environments.

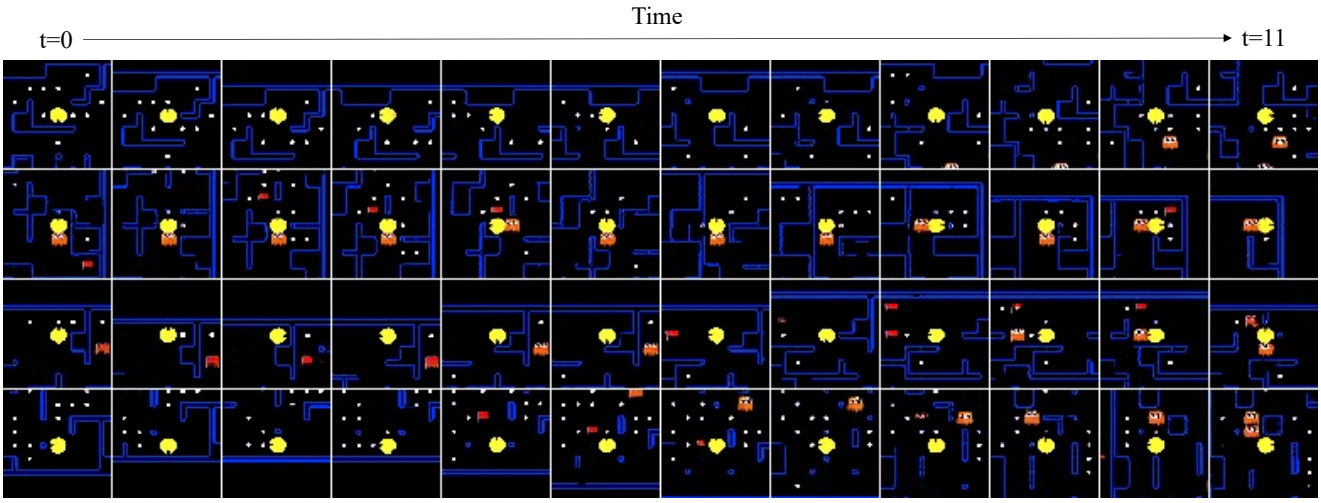


Figure 5: Pacman rollouts with GameGAN described in Section A.5. The image quality is visibly better than the simpler GameGAN model shown in Figure 7 of the main text.

B. Training Scheme

We employ the standard GAN formulation [3] that plays a min-max game between the generator G (in our case, GameGAN) and the discriminators D. Let $\mathcal{L}_{\text{GAN}}$ be the sum of all GAN losses (we use equal weighting for single frame, action-conditioned, and temporal losses). Since conditional GAN architectures [8] are known for learning simplified distributions ignoring the latent code [12, 10], we add information regularization [2] $\mathcal{L}_{\text{Info}}$ that maximizes the mutual information $I(z_t, \phi(x_t, x_{t+1}))$ between the latent code z_t and the pair (x_t, x_{t+1}). To help the action-conditioned discriminator, we add a term that minimizes the cross entropy loss $\mathcal{L}_{\text{Action}}$ between a_t and $a_t^{pred} = \psi(x_{t+1}, x_t)$. Both ϕ and ψ are MLP that share layers with the action-conditioned discriminator except for the last layer. Lastly, we found adding a small L2 reconstruction losses in the image ($\mathcal{L}_{\text{recon}} = \frac{1}{T}\sum_{t=0}^{T} ||x_t - \hat{x}_t||_2^2$) and feature spaces ($\mathcal{L}_{\text{feat}} = \frac{1}{T}\sum_{t=0}^{T} ||feat_t - \hat{feat}_t||_2^2$) helps stabilize the training. x and $\hat{x}$ are the real and generated images, and $feat$ and $\hat{feat}$ are the real and generated features from the shared encoder of discriminators, respectively.

GameGAN optimizes:

$$\mathcal{L} = \mathcal{L}_{\text{GAN}} + \lambda_{\text{A}}\mathcal{L}_{\text{Action}} + \lambda_{\text{I}}\mathcal{L}_{\text{Info}} + \lambda_{\text{r}}\mathcal{L}_{\text{recon}} + \lambda_{\text{f}}\mathcal{L}_{\text{feat}} \tag{13}$$

When the memory module and the specialized rendering engine are used, $\lambda_{\text{c}}\mathcal{L}_{\text{cycle}}$ (Section 3.4.2) is added to $\mathcal{L}$ with $\lambda_{\text{c}} = 0.05$. We do not update the rendering engine with gradients from $\mathcal{L}_{\text{cycle}}$, as the purpose of employing $\mathcal{L}_{\text{cycle}}$ is to help train the dynamics engine and the memory module for long-term consistency. We set $\lambda_{\text{A}} = \lambda_{\text{I}} = 1$ and $\lambda_{\text{r}} = \lambda_{\text{f}} = 0.05$. The discriminators are updated after each optimization step of GameGAN . When traning the discriminators, we add a term γL_{GP} that penalizes the discriminator gradient on the true data distribution [7] with $\gamma = 10$.

We use Adam optimizer [6] with a learning rate of 0.0001 for both GameGAN and discriminators, and use a batch size of 12. GameGAN on Pacman and VizDoom environments are trained with a sequence of 18 and 32 frames, respectively. We employ a warm-up phase where 9 (for Pacman) / 16 (for VizDoom) real frames are fed into the dynamics engine in the first epoch, and linearly reduce the number of real frames to 1 by the 20-th epoch (the initial frame x_0 is always given).

C. Additional Figures

Additional rollouts from the come-back-home experiment (Figure 6) and playing with the swapped background/foreground (Figure 7) are included on the next page.

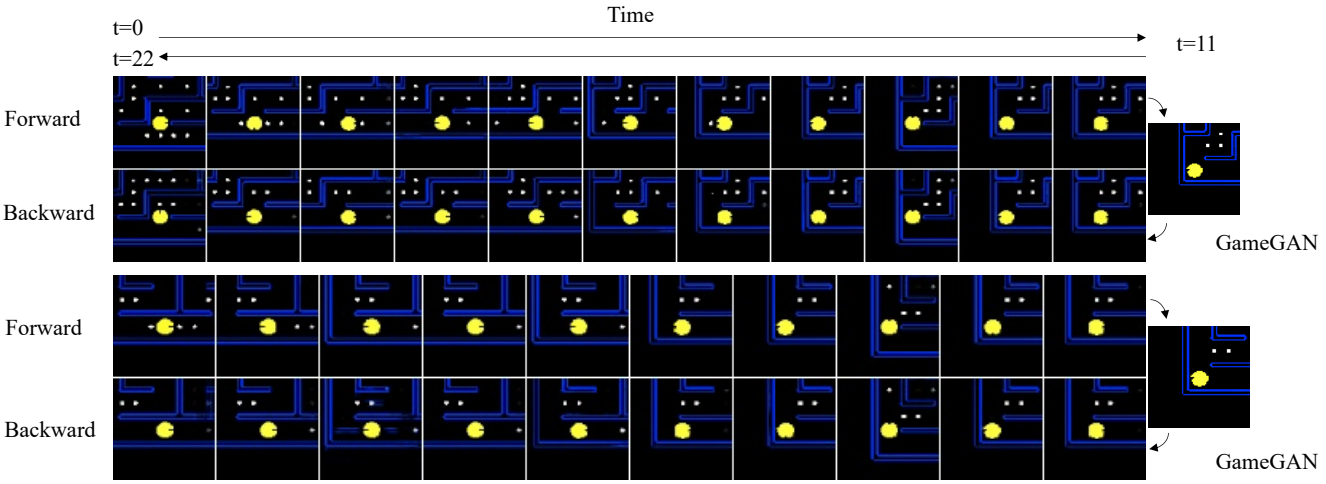


Figure 6: Come-back-home task rollouts. The top row shows the forward path going from the initial position to the goal position. The bottom row shows the backward path coming back from the goal position to the initial position.

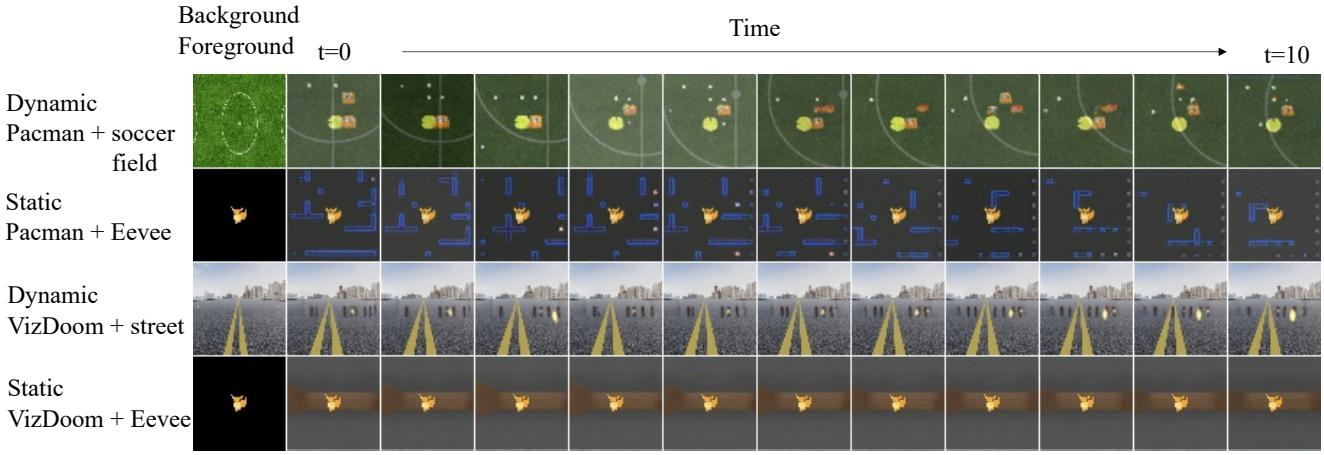


Figure 7: GameGAN on Pacman and VizDoom with swapping background/foreground with random images.

References

[1] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale gan training for high fidelity natural image synthesis. *arXiv preprint arXiv:1809.11096*, 2018. 4

[2] Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In *Advances in neural information processing systems*, pages 2172–2180, 2016. 5

[3] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014. 5

[4] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014. 2

[5] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. 4

[6] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 5

[7] Lars Mescheder, Andreas Geiger, and Sebastian Nowozin. Which training methods for gans do actually converge? *arXiv preprint arXiv:1801.04406*, 2018. 5

[8] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014. 5

[9] Taesung Park, Ming-Yu Liu, Ting-Chun Wang, and Jun-Yan Zhu. Semantic image synthesis with spatially-adaptive normalization. In *Proceedings of the IEEE Conference on Com-*

puter Vision and Pattern Recognition, pages 2337–2346, 2019. 3, 4

[10] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in neural information processing systems*, pages 2234–2242, 2016. 5

[11] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016. 4

[12] Dingdong Yang, Seunghoon Hong, Yunseok Jang, Tianchen Zhao, and Honglak Lee. Diversity-sensitive conditional generative adversarial networks. *arXiv preprint arXiv:1901.09024*, 2019. 5