Spatio-Temporal Transformers as Interactive Game Engines

Andrew Boessen Ian Bourgin Theodore Grace

Abstract

For our project, we present an approach to modeling interactive game environments using Spatio-Temporal Transformers. Unlike traditional video generation models that operate non-causally, our approach explicitly models the causal relationship between user actions and state transitions, enabling real-time interaction. We train our model on gameplay data collected from a reinforcement learning agent playing the Atari game Skiing, demonstrating that neural networks can learn both the visual aspects and the underlying dynamics of game environments. Our work bridges the gap between world models, traditionally used for reinforcement learning, and interactive environment simulation, showing that transformer-based architectures can serve as efficient and effective neural game engines.

1 Introduction

One of the most exciting and influential developments in recent years is the Transformer architecture proposed in Vaswani et al. (2023). The Transformer was originally introduced for the task of machine translation, but it has been adopted across many fields for use in Natural Language Processing (NLP) (Brown et al., 2020; Devlin et al., 2019), Computer Vision (CV) (Dosovitskiy et al., 2021), and Speech Recognition (Radford et al., 2022). In general, Transformers have demonstrated exceptional capability as a powerful sequence modeling architecture (Bhattamishra et al., 2020; Wang and E, 2024). In our work, we explore using a Transformer based architecture to model an *Interactive Game Environment*.

We define an game environment, $\mathcal{E}$, as a tuple $\mathcal{E}(\mathcal{A}, \mathcal{S}, \mathcal{O}, V, T)$ where $\mathcal{A}$ is a set of actions, $\mathcal{S}$ is a set of states, $\mathcal{O}$ is a set of observations, $V : \mathcal{S} \to \mathcal{O}$ is a projection function, and T is a transition function. In the context of a video game, $\mathcal{A}$ is the set of possible user inputs, $\mathcal{S}$ is the programs dynamic memory content, and $\mathcal{O}$ is the rendered screen pixel values. The transition function $T(s'|s, a)$ where $s \in \mathcal{S}$ and $a \in \mathcal{A}$ is represented by the game engine and V is represented by the game's rendering logic. In our work, we model T and V while representing $\mathcal{S}$ as a discrete latent space, creating a interactive environment powered entirely by neural networks.

This approach allows for the generation of dynamic, responsive environments that can adapt to users' actions in real-time. Transformers can be adapted for use with image pixel values (Nguyen et al., 2024), but perform best when operating on sequences of discrete tokens (Devlin et al., 2019; Brown et al., 2020). For textual data, it is easy to build a token vocabulary (Fleshman and Durme, 2023; Kudo and Richardson, 2018), but there is

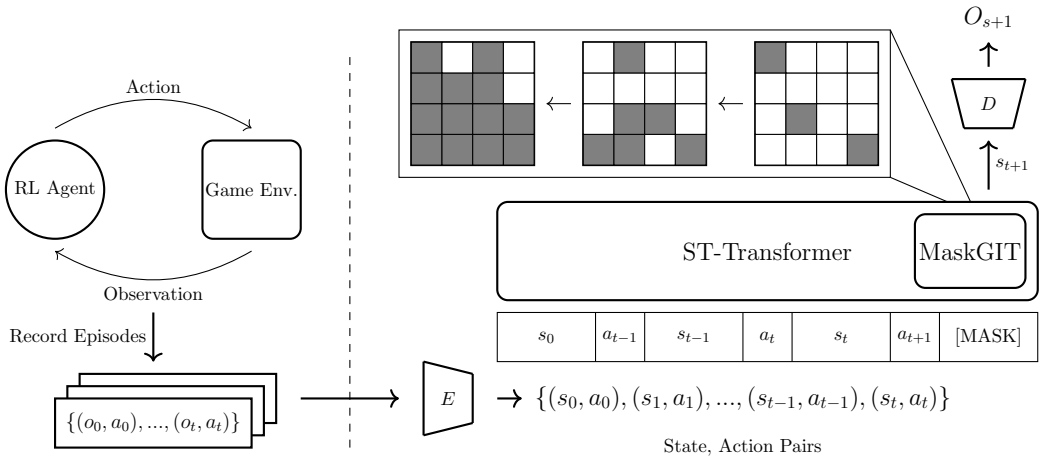


Figure 1: Interactive Game Engine. An RL agent is used to create a dataset consisting of observation, action pairs. The observations are encoded into states with the VAE encoder E. The sequential model (ST-Transformer) takes the encoded state, action pairs and predicts the next state s_{t+1}. The state to be predicted, initially represented as a mask token, [MASK], is iteratively generated in a non-auto-regressive manner with MaskGIT and bi-directional attention. Predicted states are projected to pixel space with the VAE decoder D.

no straightforward way to do this for images. Ideally, we would like to build a small discrete vocabulary instead of using raw pixel values as tokens. For this task, we use a Vector Quantized Variational Auto-encoder (VQ-VAE) (van den Oord et al., 2018) to encode images into a discrete latent space. This latent space will form the vocabulary for the sequence prediction model. This model utilizes a Spatio-Temporal Transformer (ST-Transformer) (Xu et al., 2021) as a *neural game engine* to predict the next state in a sequence of past states and actions. The models predictions are conditioned on users' input by using special action tokens and interleaving these with the image tokens in the model's context window. The decoder from the VQ-VAE can then be used to project the predicted tokens into the pixel space. The predicted tokens can then be fed back into the model, extending the context window for the next state to be predicted, and the next action can be received from the user. This can be repeated to auto-regressively model the game environment.

Game environments have been simulated by neural networks in the past (Ha and Schmidhuber, 2018), but are limited in visual quality, stability over long time horizons, and complexity in game environment. We experiment with using ST-Transformers to improve upon these results and simulate game environments at high quality with neural networks. Achieving this would have application beyond just an interactive game. For example, neural network based game engines could be a new paradigm for game design replacing existing rendering techniques with ones where games are automatically generated, similar to how image and videos have been generated by neural networks in recent years (Rombach et al., 2022) (Ramesh et al., 2021). Additionally, if these models are scaled in terms of parameters and compute and trained on a large dataset of game-play from numerous genres and types

of games, they could become foundation models that can be fine-tuned to generate novel game environment from a few images or video clips. Notably, Bruce et al. (2024) have trained a 11B parameter Transformer based model on a large dataset of over 200,000 hours of game-play video on the internet which is able to create novel 2D game environment from a prompt containing just a single image.

A crucial component in simulating a interactive game environment is the ability to create images with high visual quality. Generative models have been widely used in the field of text-to-image generation to create high-resolution images. There are several architectures and techniques used, with the most popular being Generative Adversarial Networks (GANs) (Goodfellow et al., 2014), Variational Auto-encoders (VAEs) (Kingma and Welling, 2022), and Diffusion Models (Ho et al., 2020). While state-of-the-art image generation models (Yu et al., 2022; Rombach et al., 2022) have consistently used diffusion, Transformers have achieved comparable performance in generating high quality images (Ramesh et al., 2021; Chang et al., 2022). In Transformer based models, images are generated in two stages; the first stage is to quantize an image to a sequence of discrete tokens. In the second stage, an auto-regressive model (e.g., transformer) is learned to generate image tokens sequentially based on the previously generated result. Unlike diffusion models which gradually de-noise images through an iterative process of removing Gaussian noise, this approach instead learns to directly model the distribution of image tokens in a discrete space. In our work, we explore using the MaskGIT algorithm (Chang et al., 2022), a non-autoregressive method, to generate tokens. Following Chang et al. (2022), we use bidirectional attention to predict tokens in a constant number of steps. Specifically, at each iteration, the model predicts all tokens simultaneously in parallel but only keeps the most confident ones. The remaining tokens are masked out and will be re-predicted in the next iteration. The mask ratio is decreased until all tokens are generated with a few iterations of refinement. This method has been shown to outperform state-of-the-art Transformer models while generating images 30 times faster by using an order of magnitude less steps (Chang et al., 2022).

2 Method

2.1 Problem Formulation

In our work, we explore modeling an *Interactive Game Environment* for the Atari video game *Skiing* (Whitehead, 1980). *Skiing* is a simple 2D game where the objective is to reach the bottom of the ski course in the least amount of time. To reach the bottom of the course, players must dodge obstacles and pass through a series of gates, indicated by flagpoles. We formulate the problem of simulating an *Interactive Game Environment*, $\mathcal{E}$ as described in 1, with RGB image observations $o_t \in \mathbb{R}^{H \times W \times 3}$, discrete latent states $s_t \in \{1, ..., C\}^K$, and discrete actions $a_t \in \{1, ..., A\}$ where C is the vocab size, K is the number to tokens per image, and A is the number of actions. This process of a game environment can be thought of as a variation of a Partially Observable Markov Decision Process (POMDP) (Fig. 2) without the reward function or discount rate. The dynamics of the game environment are modeled by two learned models: $T(s_{t+1}|s_{\leq t}, a_{\leq t})$, a conditional transition function between states and actions, and $V : \mathcal{S} \to \mathcal{O}$, a projection function between latent states and image observations. A notable difference between the original game environment and the sim-

ulated one is in the state space $\mathcal{S}$. Because, the training data consists of just the pixel value observations, we do not have direct access to the exact values of game variables like score, player health, etc. and information about objects off screen is unknown. Because of this, our model's state space is a compressed representation of the pixel values which is a weaker representation of the original game state. To mitigate this lack of information, we condition the current state the model is trying to predict on several previous states and actions, instead of just the immediate one. By doing this, we essentiality give the model a *memory* of what has happened in the past which enables consistency across time steps in the generation process.

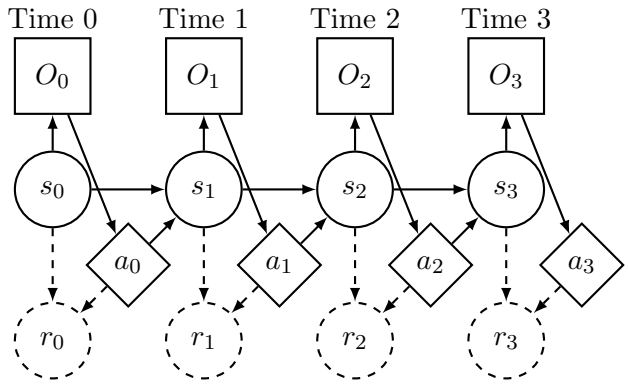


Figure 2: POMDP where $s_{t+1} = T(s_{t+1}|s_t, a_t)$ and $O_t = V(s_t)$. This diagram illustrates the components and relationships in a POMDP: states (s_t), actions (a_t), observations (O_t), and rewards (r_t) over time steps $t = 0$ to $t = 3$. Arrows indicate transitions and dependencies between components.

In order to learn these models, we first collect a large dataset of game-play observation and action pairs by using a RL agent to interact with the environment and recoding its trajectories. This approach was inspired by Valevski et al. (2024) and will allow us to create a large and diverse dataset. We train both the auto-encoder and sequence model on the same dataset. The auto-encoder contains an encoder E and decoder D, such that a reconstructed image $\hat{X} = D(E(X))$. We define a perceptual loss function $\mathcal{L}$, so that we can optimize the model to minimize $\mathcal{L}(\hat{X}, X)$. Additionally, we use a loss function $\mathcal{L}_{VQ}$ to learn the vector quantization process. This loss combines a L_2 reconstruction loss with a codebook and commitment loss (Eq. 10). The model T is implemented with a Transformer architecture and utilizes a simpler training procedure. Given a sequence of state and action pairs $X = \{(s_o, a_o), ..., (s_t, a_t)\}$, the sequential model $T(s_{t+1}|s_{\leq t}, a_{\leq t})$ can be used to get a distribution of the next state $p(s_{t+1}) = T(s_{t+1}|X_{\leq t})$ and we can optimize T to minimize the cross-entropy loss between $p(s_{t+1})$ and s_{t+1}.

2.2 Image Auto-Encoder

The goal of an auto-encoder, is to learn a compact representation of an image. In the context of our work, the auto-encoder serves as a way to express an image in a compressed format; specifically, we want to construct contextually rich representations of images with

discrete tokens. For this task, we use a VQ-VAE (van den Oord et al., 2018), which is a variation of a VAE. Variational Auto-Encoders (VAEs) (Kingma and Welling, 2022) learn to express images as a distribution by predicting the mean and standard deviation of a distribution in a latent space. Fundamentally, a VAE consists of an encoder E, a decoder D, and a latent space $\mathcal{Z}$. The encoder learns a posterior distribution $q(z|x)$, and the decoder learns a distribution $p(x|z)$ over the input data. VQ-VAEs expand upon this, by learning a discrete categorical posterior, instead of a continuous one. This is accomplished by defining a codebook and using vector quantization (VQ) (Gray, 1984) to match continuous embeddings to the closest embedding in the codebook. We can then re-purpose the codebook to tokenize an image by replacing the embeddings for an image with their corresponding ids in the codebook.

In our work, we experiment with using two different architectures for the auto-encoder: Transformer based and CNN based. The de facto standard for computer vision is the CNN, but Transformers have also been used to learn discrete representations of images (Yu et al., 2024) and are able to represent images in as few as 32 tokens. We implement a similar architecture to Yu et al. (2024) and use a Vision Transformer (ViT) (Nguyen et al., 2024) as the encoder E and decoder D. Each input can freely interact with every other input in a transformer, since it has no architectural constraints favoring local connections. While this flexibility allows it to discover intricate patterns, it also means the transformer must learn every relationship from scratch - unlike CNNs, which are specifically structured to take advantage of the fact that nearby pixels in images tend to be strongly related. In our experiments, we found that due to the low variance in our dataset, the Transformer based auto-encoder would fail to learn a meaningful posterior and instead converge to an average of the dataset for all inputs (Appendix B). We found that CNNs fit better to our task due to their inductive bias on locality (i.e. assuming that local pixel are closely related) meaning that the network is not required to learn every relationship between pixels. Becasue of these advantages, we use the CNN based VQ-VAE for the rest of our work.

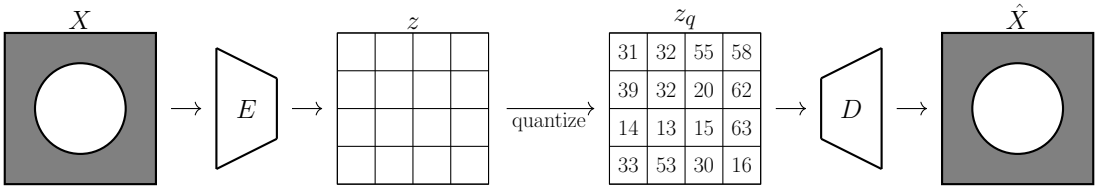


Figure 3: Vector Quantized Variational Auto-Encoder (VQ-VAE). Images X are encoded into latent embeddings z with encoder the E. Latent embeddings are quantized using a codebook $\mathcal{C}$ to get z_q. Quantized embeddings are projected to pixel values with the decoder D to get the reconstructed image $\hat{X}$.

Following Esser et al. (2021), we use residual convolutional layers followed by self-attention to construct the CNN based encoder and decoder networks. The encoder network E processes an input image $x \in \mathbb{R}^{H \times W \times C}$ through multiple down-sampling stages. The initial features are obtained through a convolutional layer:

$$h_0 = \text{Conv}_{3\times3}(x) \tag{1}$$

At each stage i, the features go through n_r residual blocks. For the j-th residual block:

$$h_{i,j} = h_{i,j-1} + \text{ResBlock}(h_{i,j-1}) \tag{2}$$

where each ResBlock follows the structure:

$$\text{ResBlock}(h) = \text{Conv}_2(\text{Drop}(\text{NonLin}(\text{Norm}(\text{Conv}_1(\text{NonLin}(\text{Norm}(h))))))) \tag{3}$$

At specified resolutions, self-attention is applied through query, key, and value projections:

$$Q = W_q h, \quad K = W_k h, \quad V = W_v h \tag{4}$$

$$\text{Attention}(h) = h + W_o(\text{softmax}(\frac{QK^T}{\sqrt{d_k}})V) \tag{5}$$

Between stages, spatial downsampling is performed:

$$h_{i+1,0} = \text{Conv}_{3\times3}(h_{i,n_r}, \text{stride} = 2) \tag{6}$$

The final encoder output z is obtained through normalization and projection:

$$z = \text{Conv}_{3\times3}(\text{NonLin}(\text{Norm}(h_{L,n_r}))) \tag{7}$$

The decoder D mirrors this architecture with transposed convolutions for upsampling:

$$h_{i-1} = \text{ConvTranspose}_{3\times3}(h_i, \text{stride} = 2) \tag{8}$$

Depending on the number of down-sampling stages, s, the encoder scales the original image by a factor of $\frac{1}{2^s}$ resulting in a compressed representation on the original image $z \in \mathbb{R}^{\frac{H}{2^s}\times\frac{W}{2^s}\times d}$, where d is the model dimension. We then quantize this representation using Vector Quantization (VQ) (Gray, 1984). Here, VQ can be seen as an application of the K-means algorithm where we have a codebook, $\mathcal{C} = \{e_1, e_2, ..., e_C\}$ containing C codes or cluster centers $e_i \in \mathbb{R}^d$. In order to quantize z we match each of its $\frac{H}{2^s} \times \frac{W}{2^s}$ embeddings to the closest code in the codebook. This is accomplished with a nearest neighbor search (Eq. 9) for each embedding z_k in z.

$$q(z_k) = \text{argmin}_{e_i\in\mathcal{C}}||z_k - e_i||_2 \tag{9}$$

The quantized representation of z, z^q, where $z^q_k = q(z_k)$ forms the input to the decoder and the indices of the embeddings in the codebook become the token values used later to train the game engine model. The codes in the codebook initially start as random values, but we want to optimize them to best fit the distribution of z. The classical method to find the best codebook consists of the Expectation Maximization algorithm which alternates between assigning all data-points to the closest code and then assigning the mean of these embeddings as the new code. Applying this approach to our task however is impractical

and would likely not converge. For example, the underlying distribution of z changes as the parameters of the encoder are modified during training, so this method could not be used to learn the codebook at the same time as the encoder. We could potentially train the encoder and decoder without the quantization step and learn a codebook after, but if the auto encoder and codebook are optimized independently it is unlikely that performance will be optimal. Becasue of this, it is desirable that the codebook and auto-encoder's joint interaction is taken into account during training.

To accomplish this, we use gradient descent to optimize the codebook. There is a slight problem with this however because the backpropagation algorithm can not be directly applied. The nearest neighbor search introduces a discontinuity meaning the gradients become zero which prevents gradient propagation to downstream operations in the computation graph. A practical solution is known as pass-through, where gradients are passed unchanged through the quantizer. This approximation is simple to implement and provides often adequate performance. To learn the codebook through gradient descent, we define an objective function (Eq. 10) that fits codes to the encoder's output but also attempts to create tight clusters with a commitment loss that penalizes the distance between the encoder's output and its corresponding code.

$$\mathcal{L}_{VQ} = \|x - \hat{x}\|^2 + \sum_{k=1}^{K} \|sg(z_k) - e_i\|_2^2 + \beta \|z_k - sg(e_i)\|_2^2 \tag{10}$$

Where $K = \frac{HW}{2^{2s}}$, $\mathcal{L}_{rec} = \|x - \hat{x}\|^2$ is a reconstruction loss, $sg()$ denotes the stop-gradient operator, and β is a hyperparameter controlling the commitment to the codebook. The second term in $\mathcal{L}_{VQ}$ updates the codebook vectors to match the encoder output, while the third term encourages the encoder to commit to codebook vectors. This formulation combined with the perceptual VAE loss described in 2.1 allows for end-to-end training of the entire system.

2.3 Spatio-Temporal Transformer

The Spatio-Temporal Transformer (ST-Transformer) (Xu et al., 2021) is a critical component of the architecture of the simulation. It is responsible for modeling the dynamics of our game environment by predicting future latent states based on previous states and actions. It extends the standard Transformer architecture (Vaswani et al., 2023) to handle spatio-temporal data by attending to spatial and temporal dependencies separately. The self-attention mechanism enables the model to relate different parts of the sequence, effectively learning how spatial configurations can change over time given a series of user actions. Inputs from user actions are what drives the determination of the next state of the game. Given an input sequence of T state and action token pairs $X = \{(s_o, a_o), ..., (s_T, a_T)\}$ where $s_t \in \{1, ..., C\}^K$ and $a_t \in \{1, ..., A\}$, we can then learn embeddings for the state and action tokens to get the Transformer's input $X \in \mathbb{R}^{T \times (K+1) \times d_m}$ where d_m is the Transformer model dimension.

The ST-Transformer model is formed by combining spatial $\mathcal{S}$ and temporal $\mathcal{T}$ blocks in sequence for L layers (Fig. **??**). The spatial block attends to the spatial dependencies in

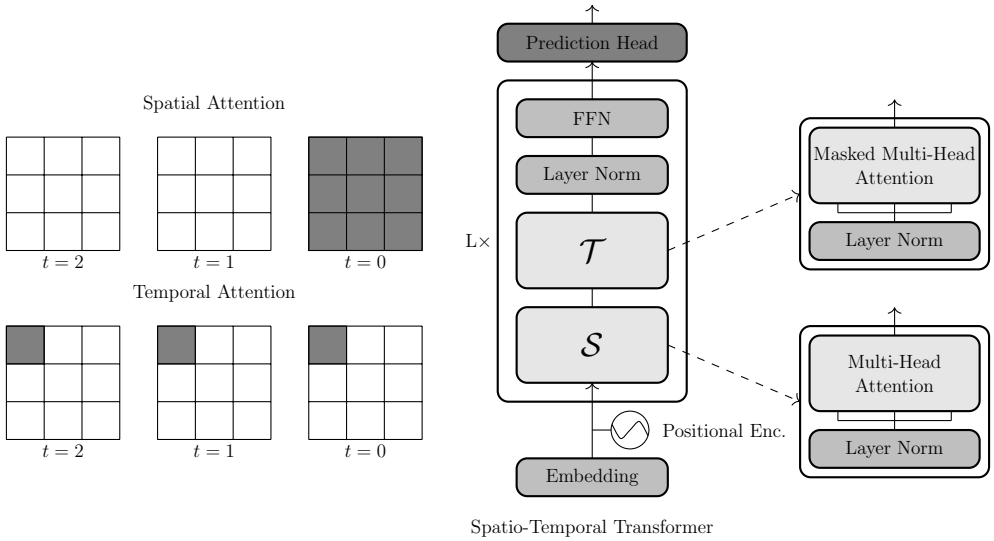


Figure 4: The Spatio-Temporal Transformer uses separate spatial and temporal attention blocks. The spatial block $\mathcal{S}$ attends to all tokens in a single image, and the temporal block $\mathcal{T}$ attends to the same token across all time steps. The diagram represents the spatial and temporal inputs, X_0^S and X_0^T. The attention blocks are applied sequentially and followed by a feed-forward network (FFN) for L layers. A prediction head is used at the end to get a distribution across the token vocabulary.

each of the T state and action pairs. We define the spatial input $X_t^S \in \mathbb{R}^{(K+1)\times d_m}$ and can use $\mathcal{S}$ to extract the spatial features Y^S.

$$Y^S = \mathcal{S}(X_t^S) \qquad t = 1...T \tag{11}$$

The temporal block attends to the temporal dependencies in the sequence, where the temporal input $X_k^T \in \mathbb{R}^{T\times d_m}$, is formed by combining the token in state index k across all T states. We can then use $\mathcal{T}$ to extract the temporal features Y^T.

$$Y^T = \mathcal{T}(X_k^T) \qquad k = 1...K+1 \tag{12}$$

The spatial $\mathcal{S}$ and temporal $\mathcal{T}$ blocks both use multi-headed self-attention (MSA), see (Appendix C) and layernorm (LN) along with a residual connection at the end. A causal mask is applied to the attention heads in the temporal blocks.

$$\mathcal{S} = \mathcal{T} = MSA(LN(X)) + X \tag{13}$$

Putting this all together, a single layer consists of applying $\mathcal{S}$ and $\mathcal{T}$ in sequence and a Feed Forward Network (FFN) (Appendix E) at the end. We then apply this for L layers.

$$X_{l+1} = FFN(LN(\mathcal{T}(\mathcal{S}(X_l)))) + X_l \qquad l = 1...L \tag{14}$$

To attain the logits, we use a prediction head P at the end of the network to project the embeddings into the dimension of the vocab (App. F). Note that this layer does not apply softmax because we perform this later in the cross-entropy loss function.

$$Y = P(X_L) \tag{15}$$

We also use learned positional embeddings to inject spatial and temporal position information into the model's input. This can be done by learning separate spatial and temporal positional embeddings $\mathcal{P}^S \in \mathbb{R}^{(K+1)\times d_m}$ and $\mathcal{P}^T \in \mathbb{R}^{T\times d_m}$. We can then tile these embeddings along the temporal and spatial dimension respectively to get $\mathcal{P}'^S \in \mathbb{R}^{T\times(K+1)\times d_m}$ and $\mathcal{P}'^T \in \mathbb{R}^{T\times(K+1)\times d_m}$. We then add both of these to the input X to get the positionally embedded input X'.

$$X' = X + P'^S + \mathcal{P}'^T \tag{16}$$

2.4 State Generation

While we model the sequence of state, actions pairs in an auto-regressive manner, the tokens for a single state are not predicted sequentially. A naive approach would just predict all tokens in a single step in parallel. This approach would result in fast generation times, but predicting multiple tokens at the same time is difficult and results in degradation of quality. The next obvious approach is to predict each token in the state sequentially. While this would work, and has been used for image generation in the past (Esser et al., 2021), it requires many steps (e.g. one step per token) and other techniques can be used to generate higher quality images (Chang et al., 2022). In our work, we use the Masked Generative Image Transformer (MaskGIT) (Chang et al., 2022) algorithm, which utilizes bidirectional attention and a mask schedule to generate tokens in a constant number of steps.

Specifically, to generate a new image we start with a blank canvas with all tokens masked out. To achieve this, we use a special [MASK] token. We denote the current masked tokens at iteration t as Y_M^t, so Y_M^0 correspond to the initial iteration with all tokens in the state being the [MASK] token. The mask M is represented as a binary array where 1 is masked and 0 is unmasked. For an iteration t, our model predicts the probabilities for all masked tokens in parallel, denoted as p^t. The number of tokens to mask is determined by a schedule. Given a fixed number of iterations T, a mask schedule, γ, and the number of tokens N, we compute the number of tokens to mask, n, by

$$n = \lceil \gamma(t/T)N \rceil \tag{17}$$

The choice of schedule function γ is a simple architectural decision that can greatly affect the generative performance of the model. The function γ must be continuous and bounded between 0 and 1. It should also be monotonically decreasing with respect to t/T. Chang et al. (2022) explore three categories of schedule functions: linear, concave, and convex. In their experiments they find that a cosine schedule works best, so that is what we chose to use in our work as well.

The location of these masked tokens in the mask, is determined by the previous iterations probabilities and "confidence" scores. For example, we can obtain Y_M^{t+1} by masking n tokens in Y_M^t. Specifically, at each masked location i we sample a token y_i^t from the distribution $p_i^t \in \mathbb{R}^C$, where C is the number of tokens in the model's vocabulary. The sampled token's

probability is used as the confidence score c_i for index i. The confidence score for a non-masked token is simply 1.0. Putting this together, the mask M^{t+1} is calculated by

$$m_i^{t+1} = \begin{cases} 1, & c_i < \text{sorted}_j(c_j)[n] \\ 0, & \text{otherwise} \end{cases} \tag{18}$$

To summarize, the generative algorithm predicts a state with N tokens in a constant T steps. At each iteration, all tokens are predicted simultaneously, but we only keep the most confident ones. The remaining tokens are masked out and re-predicted. The mask ratio decreases with each iteration until all tokens are predicted (i.e. no tokens are masked and all confidence scores are 1.0).

To train the model, we use a similar proxy task to BERT (Devlin et al., 2019), a language model that uses masked bi-directional attention. BERT uses a fixed mask ratio in training of 15%, but this will not transfer to our task, because we need to be able to predict across a range of mask ratios. To accomplish this, we randomly sample a ratio $r \in [0, 1)$ during training to simulate various generation iterations. At each iteration in the training process we create a mask, m, by sampling N i.i.d. Bernoulli random variables with probabilities r. During training, we use a cross-entropy loss on the models logits, x, and target token distributions, y, which is represented as a one-hot encoding of the target token values. Specifically, we used a masked cross-entropy loss, $\mathcal{L}_{\text{MaskCE}}$, where we average the losses of only the masked positions.

$$\mathcal{L}_{\text{MaskCE}} = \frac{1}{\sum_{n=1}^{N} m_n} \sum_{i=1}^{N} -m_i \sum_{j=1}^{C} \log \frac{e^{x_i^j}}{\sum_{c=1}^{C} e^{x_i^c}} y_i^j \tag{19}$$

2.5 Data Collection via Agent Play

Since our goal is to build a simulation that human players can interact with in real time, our training data should reflect human game-play. Unfortunately, there is no large dataset of human game-play of Atari Skiing, so we are forced to find an alternate source of data. Similarly to Valevski et al. (2024), we train an RL agent to collect sequences of game-play data by interacting with the game environment. Unlike a typical RL setup, our goal is not for the agent to maximize a reward. Rather, our aim is for the agent to generate a diverse dataset containing a variety of scenarios to maximize the similarity to a human-generated dataset. Typically, reinforcement learning is modeled as a Markov decision process which is a tuple consisting of a state space $\mathcal{S}$, an action space $\mathcal{A}$, a transition function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$, and a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to U \subseteq \mathbb{R}$. The RL agent navigates the state space by choosing actions. Hence, the goal of the agent is to learn a policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ that maximizes the expected cumulative reward. This policy is what dictates the agent's actions given the current state and set of actions.
In our case, $\mathcal{S} = [0, 255]^{210 \times 160 \times 3}$ is the set of possible RGB pixel values that correspond to a state of the game and $\mathcal{A}$ consists of three discrete actions the agent can take $\{0, 1, 2\}$ corresponding to {"No Action", "Move Right", "Move Left"}. Since the goal of the game is to complete the course in as little time as possible, our reward function $\mathcal{R}$ is the in-game

time. Hence, maximizing the reward function is equivalent to minimizing the time spent on the course. To learn this policy, we make use of the Double Q-Learning algorithm proposed by Hasselt (2010). Ordinary Q-Learning is a model-free algorithm that aims to learn an optimal Q-function denoted $\mathcal{Q} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ which maps each state-action pair to the expected cumulative reward of taking said action in said state (with the added assumption that the agent follows the optimal policy thereafter). In order to determine these $\mathcal{Q}$-values, we begin with a random initialization. We then proceed to iteratively use the following procedure while the agent interacts with $\mathcal{S}$ to update the $\mathcal{Q}$-values:

$$\mathcal{Q}_{t+1}(s_t, a_t) = \mathcal{Q}_t(s_t, a_t) + \alpha_t(s_t, a_t)\left(\mathcal{R}(s, a) + \gamma \max_a\{\mathcal{Q}_t(s_{t+1}, a)\} - Q_t(s_t, a_t)\right) \tag{20}$$

where $\gamma \in (0, 1)$ is the *discount rate*, a hyperparameter which affects how much value is accorded to future rewards and $\alpha_t(s_t, a_t) \in (0, 1]$ is a standard *learning rate*. We have chosen to implement this algorithm via a convolutional neural network. In other words, our goal will be to approximate these $\mathcal{Q} - values$ rather than (traditionally) explicitly calculating them and storing them into a table. Although $\mathcal{Q}$-learning has been shown to converge to the optimum action-value pairs with probability one Watkins (1989), it can suffer from poor performance due to compounding positive bias in the estimates Hasselt (2010). This positive bias is the result of using $\max_a Q(s_{t+1}, a)$ as a proxy for $\max_a \mathbb{E}[Q(s_{t+1}, a)]$. To prove that this causes a positive bias, notice that since $\max_{a \in S} : S \to s \in S$ is a convex function, we can use Jensen's inequality to obtain the following:

$$\begin{aligned} &\max_a \mathbb{E}[Q(s_{t+1}, a)] \leq \mathbb{E}[\max_a Q(s_{t+1}, a)] \\ \Leftrightarrow &\max_a \mathcal{Q}_t(s_{t+1}, a) \leq \mathbb{E}[\max_a Q(s_{t+1}, a)] \quad \text{by our proxy use} \end{aligned}$$

Hence, we find that:

$$Bias(\max_a Q(s_{t+1}, a)) = \mathbb{E}[\max_a Q(s_{t+1}, a)] - \max_a \mathcal{Q}_t(s_{t+1}, a) \geq 0$$

Secondly, implementing this procedure via a DQN (Deep $\mathcal{Q}$-network) presents another issue. Indeed, we will be using the same model for prediction as well as evaluation (i.e. minimization of our given loss function) which can threaten convergence. The solution to both of these problems is the previously mentioned double $\mathcal{Q}$-learning algorithm.
Rather than having one function for both action selection and evaluation, double $\mathcal{Q}$-learning uses a model $\mathcal{Q}_\theta$ for selection and a target model $\mathcal{Q}'_{\theta^*}$ for evaluation. Following the work by Hasselt et al. (2015) we implement this model using two convolutional neural networks. Training these models is a two step process. First, we let the agent interact with environment following $\mathcal{Q}_\theta$, storing the tuples (s_t, a_t, s_{t+1}, r_t) into $\mathcal{D}$ a memory buffer. This memory buffer will be what we train our model parameters on. Once enough data has been collected (i.e. we have played long enough that we are heuristically certain that our agent has interacted in many different ways) we sample batches from $\mathcal{D}$ and compute the target value:

$$\mathcal{Q}_{\theta^*}(s_t, a_t) \approx r_t + \gamma \mathcal{Q}_\theta(s_{t+1}, \arg\max_{a'} \mathcal{Q}_{\theta^*}(s_{t+1}, a')) \tag{21}$$

We then minimize $MSE(\mathcal{Q}_{\theta^*}(s_t, a_t), \mathcal{Q}_\theta(s_t, a_t))$ via gradient descent. At this point we have trained only our target model, so the way our agent interacts with the environment has not changed. The crucial step in the double $\mathcal{Q}$-learning is the syncing step where we periodically sync the model parameters θ to θ^*. We then repeat this process. As shown by Hasselt et al. (2015) this procedure solves the aforementioned issues with the simple DQN. Note that the observations are correlated by construction of the MDP which is known make learning unstable Mnih et al. (2015). The memory buffer $\mathcal{D}$ is an implementation of *experience replay* which solves this problem by uniformly sampling across past observations, removing correlations in the observation sequence Mnih et al. (2015). Finally, that a fundamental principle is to ensure the agents behavior has a balance of *exploration* and *exploitation*. In other words, if at the beginning our agent behaves randomly, after some training has occurred this will no longer be the case as the agent has a policy that will guide to certain actions. This can prevent our agent from finding new behaviors which may yield better result. Our solution to this is to implement the ϵ-greedy algorithm. Hence, at each time step t, the agent will take a random action with a probability ϵ and a learned action (the one which maximizes the expected reward) with probability $1 - \epsilon$. Over the course of training, we decay ϵ such that the agent takes an increasing amount of learned actions.

3 Related Work

3.1 World Models

In the field of Reinforcement Learning (RL), *world models*, first introduced in Ha and Schmidhuber (2018), are used to learn abstract representations of the spatial and temporal dynamics in a environment. By training an agent in this abstract representation of the world, it is able to learn a policy entirely within a "dream" environment. A key advantage of this approach, is that it allows for sample efficient learning (Micheli et al., 2023) by training on predictions about the future state of the environment, rather than real-world observations. This technique has demonstrated success in training agents to maximize a reward function, but the predicted sequences have low visual quality and quickly degrade over extended periods of time (Ha and Schmidhuber, 2018). Alonso et al. (2024) addresses this issue by prioritizing visual details in their world model architecture. Inspired by diffusion model's success in text-to-image generation, Alonso et al. (2024) utilize diffusion for world modeling. This diffusion world model is stable over long time horizons and is able to generate high quality images and visual details. The model's effectiveness is demonstrated by achieving a mean score of 1.46 times the human average across a range of Atari games on the Atari 100k benchmark, making it valuable for RL agents, but more significantly for our research, since it can be adapted to function as a real-time neural game engine. Notably, Alonso et al. (2024) train the diffusion world model on static game-play footage from *Counter-Strike: Global Offensive*, demonstrating that it can successfully model 3D game environments.

3.2 Video Generation

Video generation has emerged as an important research area with seeming relevance to interactive environments, though fundamental differences in causality create significant

challenges in applying these techniques to neural game engines. While early approaches extended GAN architectures (Vondrick et al., 2016; Saito et al., 2017), recent work has focused on Transformer-based models and diffusion techniques. Video Diffusion Models (Ho et al., 2022) and Transformer-based approaches like VideoGPT (Yan et al., 2021), Make-A-Video (Singer et al., 2022), and Phenaki (Villegas et al., 2022) have demonstrated impressive results in generating coherent video sequences. However, these models typically operate in a non-causal manner, having access to both past and future frames during training and inference through bidirectional attention mechanisms. This non-causal nature, while effective for pre-planned video generation, fundamentally conflicts with the requirements of an interactive environment where future states must be conditioned solely on past observations and current user inputs. Even controlled generation approaches like CogVideo (Hong et al., 2022) and the work of Wang et al. (2023), while allowing for some degree of content control, operate on fixed sequences rather than adapting to real-time user interactions. The fundamental differences in causality and conditioning requirements highlight why traditional video generation techniques, despite their impressive visual quality, cannot be directly applied to creating interactive environments, motivating our approach that explicitly models the causal relationship between user actions and state transitions.

3.3 Interactive World Simulation

Recent advances in neural networks have enabled the development of models capable of simulating interactive game environments, representing a convergence of world models, video generation, and real-time rendering. Unlike non-causal video generation models, these approaches explicitly account for user inputs and maintain causal relationships between actions and state transitions. GameGAN (Kim et al., 2020) pioneered this direction by using generative adversarial networks to learn and simulate classic games like Pac-Man, demonstrating that neural networks could capture both the visual aspects and underlying dynamics of simple game environments. A significant advancement in this field came with Genie (Bruce et al., 2024), which demonstrated the ability to learn and generate novel 2D environments from video demonstrations or a single image. Genie's key innovation lies in its use of a Transformer based world model that operates in a learned discrete latent space, allowing it to capture complex 2D dynamics while maintaining computational efficiency. DIAMOND (Alonso et al., 2024) further improved upon this approach by incorporating diffusion models into the world modeling process, achieving higher visual fidelity while preserving real-time interactivity. Unlike traditional diffusion models used in video generation, DIAMOND (Alonso et al., 2024) employs a specialized architecture that conditions the diffusion process on user inputs, enabling responsive environment simulation while maintaining temporal consistency. Most recently, Valevski et al. (2024), introduced GameNGen, a diffusion based architecture capable of a real-time interaction with a complex environment over long trajectories at high quality. Notably, GameNGen was able to simulate the classic video game *DOOM* at over 20 frames per second on a single TPU.

4 Experimental Setup

In this section, we outline our experimental procedures and offer insight into the data set, models, training procedures, and hyper-parameters. All model and training procedures are defined through various configuration files.

4.1 Dataset

Using a data set of approximately 100 episodes, full episodes of game play from the Atari video game *Skiing* using a trained Double Q-Learning agent. This data was split in a ratio of 80-20 between training and validation sets respectively. The total number of observation in the training set was around $33,000$ while the total in the validation set was near $8,000$. No additional shuffling was applied before the 80-20 split in order to preserve temporal coherence. Each episode contained around 400 - 500 frames, each stored as $256 \times 256 \times 3$ RGB image. We recorded the corresponding action from a discrete set of three possible actions: "No Action", "Move Right", "Move Left" alongside each frame.

4.2 Auto-Encoder

The image tokenizer used is a Vector Quantized Variational Auto-Encoder (VQ-VAE). The encoder contains 4 down-sampling layers which compresses each $256 \times 256 \times 3$ RGB image into a 16×16 grid of tokens yielding 256 tokens per image. The token is selected from a vocabulary of size 512. The decoder mirrors this with 4 up-sampling layers to reconstruct the RGB image. The configuration details for the tokenizer are as follows:

For the architecture of the tokenizer, a convolutional backbone with residual blocks and selective attention layers at resolutions of 16 and 32 were used for both the encoder and decoder. In regards to channel dimensions, we used a base channel count of 64 for the 2D convolutional layers. Additionally, 512 latent embedding dimensions were used. This results in a model size of 4M parameters. A details list of parameters per layer is given in Table 1.

For resolution, the input images are center-cropped or resized to 256×256 for stable encoding and decoding. The tokenizer was pre-trained for 15 epochs or approximately 130,000 steps until convergence. Once trained, frames were converted into token sequences offline and stored for further training of the ST-Transformer. The final tokenizer checkpoint for the ST-Transformer training is specified as `tokenizer_checkpoint: "checkpoint_epoch_15.pt"` and used for purposes of training the ST-Transformer.

4.3 ST-Transformer

The ST-Transformer serves as our neural game engine by modeling a sequence of state-action pairs over time. It predicts future states conditioned on a context of past states and actions. The key configurations are as follows: For dimensions, we used the following: `embedding dimensions (dim):` 512, `attention heads (num_heads):` 8, `transformer layers (num_layers):` 4. This results in a model size of 17M parameters. A detailed list of parameters per layer is available in Table 2.

A context length of 20 state and action pairs was used as well during training. Furthermore, since each frame is represented by 256 tokens and one action token, and we consider

Layer (depth-idx)	Param #
Tokenizer	–
Encoder: 1-1	–
Conv2d: 2-1	1,792
DownsampleList: 2-2	–
DownModule: 3-1	185,152
DownModule: 3-2	185,152
DownModule: 3-3	185,152
DownModule: 3-4	218,688
ResModule: 3-5	181,760
ResnetBlock: 3-6	74,112
AttnBlock: 3-7	16,768
ResnetBlock: 3-8	74,112
GroupNorm: 2-4	128
Conv2d: 2-5	295,424
Conv2d: 1-2	262,656
Codebook 1-3	262,144
Conv2d: 1-4	262,656
Decoder: 1-5	–
Conv2d: 2-6	294,976
ResnetBlock: 3-9	74,112
AttnBlock: 3-10	16,768
ResnetBlock: 3-11	74,112
UpsampleList: 2-8	–
ResModule: 3-12	222,336
UpModule: 3-13	259,264
UpModule: 3-14	259,264
UpModule: 3-15	309,568
UpModule: 3-16	309,568
GroupNorm: 2-9	128
Conv2d: 2-10	1,731
Total params	4,027,523

Table 1: VQ-VAE Model Parameter Counts

multiple frames, this context length controls how many past tokens and actions are attended to.

As previously mentioned, a vocab size of 512 was used alongside 3 discrete action types. Each action is then inserted into the sequence, allowing the model to condition its prediction on user inputs. We also used a mask token index, represented by `mask_token: 512`, which occurs just after the last vocabulary token.

For dropout rates, we had three dropout hyper-parameters: `Attention Dropout (attn_drop): 0.0`, `Projection Dropout (proj_drop): 0.2`, and a `Feed-Forward Network Dropout (ffn_drop): 0.2`

Layer (depth-idx)	Param #
SpatioTemporalTransformer	–
TokenEmbedding: 1-1	264,192
TemporalEmbedding: 1-2	131,072
SpatialEmbedding: 1-3	10,240
SpatioTemporalLayersList: 1-4	–
SpatioTemporalLayer: 2-1	–
Sequential: 3-1	4,200,960
SpatioTemporalLayer: 2-2	–
Sequential: 3-2	4,200,960
SpatioTemporalLayer: 2-3	–
Sequential: 3-3	4,200,960
SpatioTemporalLayer: 2-4	–
Sequential: 3-4	4,200,960
PredictionHead: 1-5	–
Linear: 2-5	262,656
Total params	17,472,000

Table 2: ST-Transformer Model Parameter Counts

The ST-Transformer was trained with the masked cross-entropy loss function Eq. 19 to handle masked image tokens, which enables MaskGIT. During training, a variable mask ratio would simulate the iterative refinement procedure performed inference time. We use a curriculum learning schedule to help aid model convergence. Specifically, we increase the max possible mask ratio, $r(i)$, over $50,000$ steps following a sin schedule. A minimum training mask ratio `min_train_mask_ratio: 0.05` were employed to gradually increase masking difficulty as training progressed.

$$r(i) = \min(1, \max(0.05, \sin(\frac{i}{50,000}\frac{\pi}{2}))) \tag{22}$$

4.4 Training Details

Both models were trained on a single Nvidia RTX 4070 with 12GB of memory. For training, we used mixed precision training (enabled through `mixed_precision: true`) to reduce training time and memory usage. For batching and data loading, we used a batch size of 8 and 12 data loader workers (`num_workers`). For optimization, we used the Adam optimizer with a learning rate of 1.0×10^{-4} and a weight decay of 1.0×10^{-5}.

Furthermore, we applied a cosine annealing learning rate schedule with a warmup phase. The following hyperparameters were used: `min_lr: 1.0e-5`, `warmup_epochs: 5`, and `warmup_start_lr: 1.0e-6`. For training duration we used 100 epochs with gradient clipping 1.0. Validation was then performed every 5 epochs and we saved only the best-performing model for the validation set. Training logs and metrics were recorded every 100 steps and tracked in a logging system specified by `project_name: "game_transformer` and `run_name: "training_run_v1"`. Log files are available in Google Drive: `https://drive.google.com/drive/folders/13-3egPYDD30uGBetvIPFUzVcoHoFbwsk?usp=drive_link`

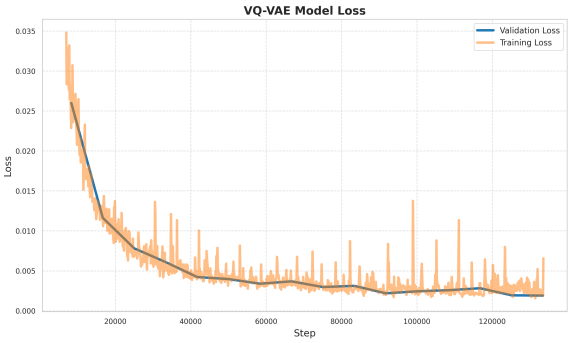


Figure 5: VQ-VAE Learning Curve

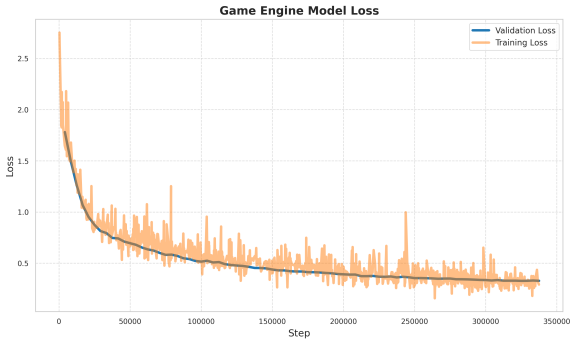


Figure 6: ST-Transformer Learning Curve

Figure 7: Learning Curves

4.5 Learning Curve

Learning curves for the VQ-VAE and ST-Transformer models are available in Fig. 7.

5 Results

To evaluate the performance of the models we use several metrics. The image auto-encoder is used to compress images into a latent space and reconstruct the original image. Because of this we use a reconstruction loss which measures the mean squared error between the original image and reconstruction. To get a finer detail on the quality of images we use LPIPS (Zhang et al., 2018) which is a perceptual loss. We also use the Fréchet Inception Distance (FID) which assess the quality of images based on the difference between the distribution of generated images and original target images. To evaluate the performance of the ST-Transformer, we compute similar metrics, but on the decoded images of the predicted tokens. We use the same LPIPS loss as well as Peak Signal to Noise Ratio (PSNR). The PSNR measures the amount of noise in a generated image and is measured in decibels. For example, the typical values for a lossy image compression algorithm is 30 to 50 db.

5.1 Auto-Encoder

We perform the evaluation on a validation dataset of around 8,000 images. The results for reconstrcution loss, LPIPS, and FID are available in Tab. 3.

Metric	**Value** $\downarrow$
Reconstruction Loss	1.429e-3
Perceptual Loss (LPIPS)	3.828e-4
Fréchet Inception Distance (FID)	1.398e-1

Table 3: Performance Metrics

Additionally, we have included a sample of reconstructed images from this validation dataset in Fig. 8

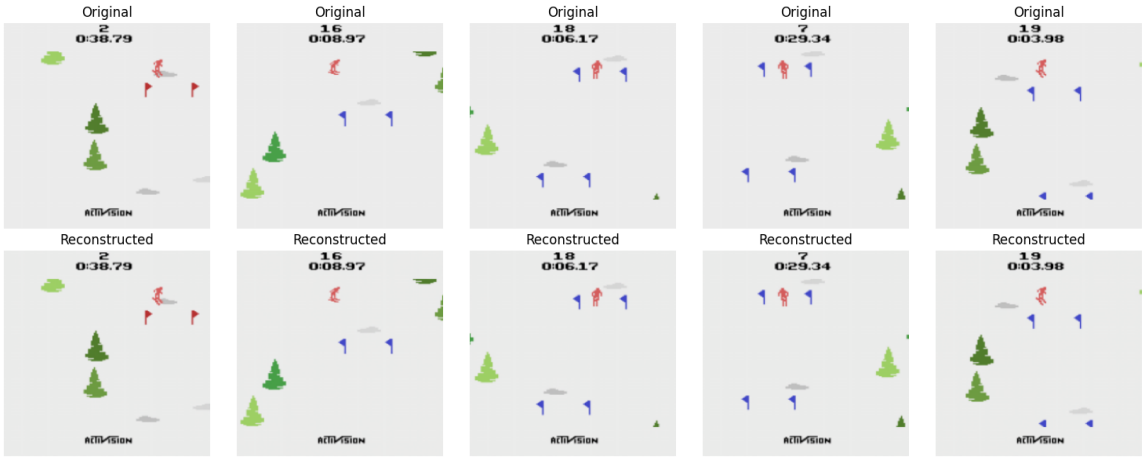


Figure 8: VQ-VAE Reconstructions. *Top*: Original images, *Bottom*: Reconstructed images

5.2 Embedding Analysis

We also perform an analysis of the latent embedding dimension. We use a UMAP (McInnes et al., 2020) dimensionality reduction to visualize the 512 dimensional space in 2 dimensions. Results for a sample of $1,000$ images from the validation dataset are seen in Fig. 9.

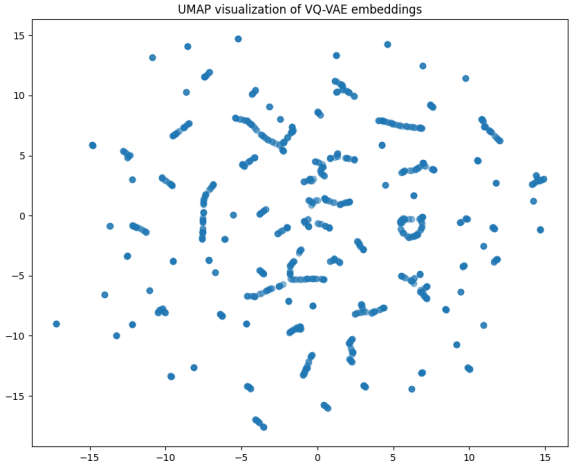


Figure 9: UMAP dimensionality reduction on embeddings. Embeddings sampled from 1000 images

5.3 Codebook Usage

We also explore the usage of the codebook across images. The average token values for the same sample of $1,000$ image is seen in Fig. 10. Because the spatial position of the

pixels in the image are preserved in token grid, this demonstrates how certain token values correspond to certain parts of the image.

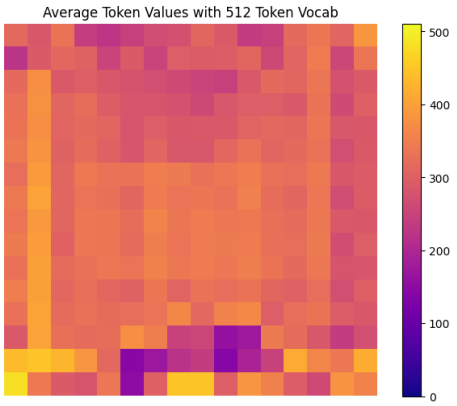


Figure 10: Average token values for 16x16 compressed image representation. Token values calculated from sample of 1000 images

5.4 Neural Game Engine

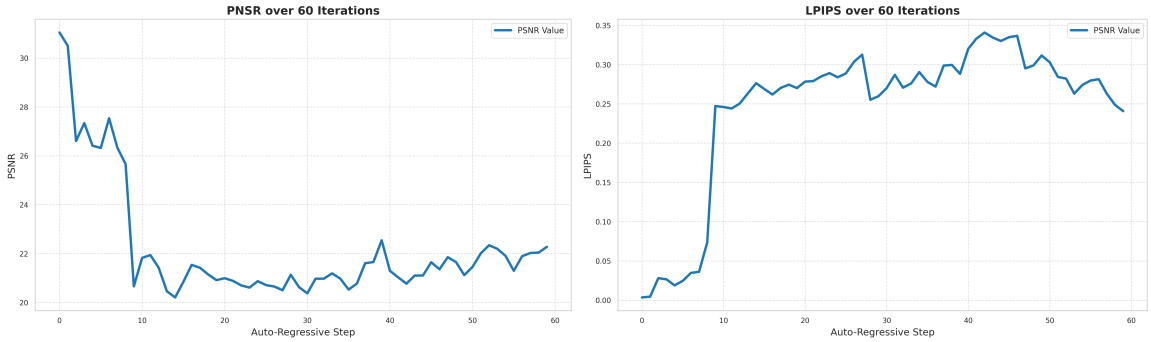
To evaluate the performance of the performance of the transformer, we use the same validation dataset used for the auto-encoder. Each example in the dataset contains a sequence of 20 image and action pairs. We tokenize the images and interleave the actions for the input to the model. We mask out the last images tokens with the [MASK] token and predict the image with 8 iterations and a temperature of 0.1. The LPIPS and PNSR values are calculated on the original image that we masked and the decoded pixel values of the predicted tokens. Results are available in Tab. 4.

Metric	**Value**
Perceptual Loss (LPIPS) ↓	3.478e-3
Peak Signal to Noise Ratio (PSNR) ↑	31.042

Table 4: Game Engine Performance Metrics

5.5 Hyper-Parameter Tuning

We experiment we several different combinations of hyper-parameters for the process of generating images using the MaskGiT algorithm. The generation process can be controlled by choosing the number of iterations to generate an image in and the temperature which controls the predicted token's softmax distribution. More information is available in App. A.



(a) PSNR over 60 steps

(b) LPIPS over 60 steps

Figure 11: Auto-Regressive Drift of PSNR and LPIPS

5.6 Auto-Regressive Drift

Due to the interactive game environment being generated in an auto-regressive manner, errors can compound and quality tends to degrade over time. The model is conditioned on its previously generated predictions. However, during this iterative sampling process, the predicted trajectory tends to deviate from the ground-truth trajectory after just a few steps. This divergence primarily stems from the incremental accumulation of subtle differences in frame-to-frame movement velocities. Consequently, the performance metrics reflect this divergence: the per-frame Peak Signal-to-Noise Ratio (PSNR) gradually declines, while the Learned Perceptual Image Patch Similarity (LPIPS) increases over time, as demonstrated in Figure. 11.

5.7 Object Interaction

The neural game engine performs well and is able to generate coherent sequences of images of the trajectories are similar to those in the training dataset. However, the model can struggle and fail altogether in some edge cases. One particular area where the model fails is in object interaction. Specifically, when the player collides with an object like a flag pole or a tree, the player sprite becomes occluded and can disappear entirely from the screen. This obviously is not the intended behavior nor how the original game behaves. An example of this can be seen in Fig. 12. Another observed undesired behavior is that if the player drift too far from the center of the screen, the generation will become unstable and the player sprite can move erratically. This instabilities in generation are likely due to a lack of diversity in the training data. There could be insufficient data of object interaction and trajectories outside the center of the screen leading the model to not be able to adapt to these situations.

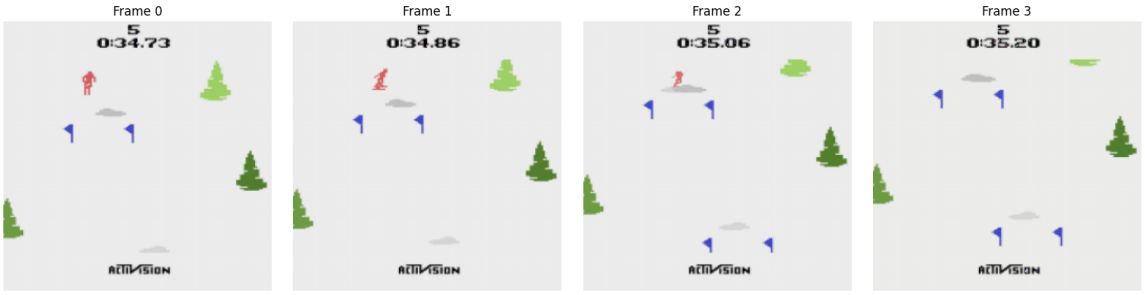


Figure 12: Player Collision Resulting in Sprite Disappearing

6 Contribution

6.1 Andrew Boessen

Wrote code for transformer based auto-encoder. Trained CNN based auto-encoder. Wrote code and trained Transformer game engine model. Wrote report sections for intro, related work, problem formulation, image encoder, results, appendix, and created figures for report. Created colab notebook

6.2 Ian Bourgin

Wrote code for initial RL agent and training process for data collection. Wrote data collection via RL agent section in report. Made the power point for the video presentation. Also performed literature review.

6.3 Theodore Grace

Wrote code for original CNN based auto-encoder. Wrote ST-Transformer section in report.Wrote section 4 in its entirety. Contributed to designing video presentation.

Appendix A. MaskGiT Hyper-Parameter Tuning

We experiment we several different combinations of hyper-parameters for the process of generating images using the MaskGiT algorithm. The generation process can be controlled by choosing the number of iterations of generate and image in and the temperature which controls the predicted softmax distribution. Example generated images are available in Fig. 13. Video example are also available in the colab demo.

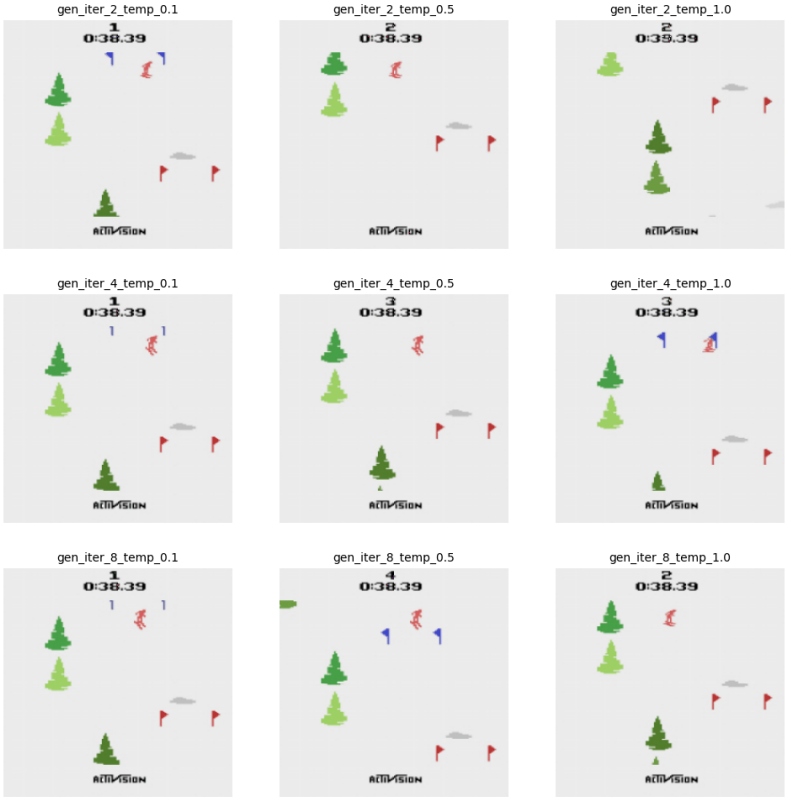


Figure 13: Effect of temperature and generation iterations. Temperatures: [0.1, 0.5, 1.0] Iterations: [2, 4, 8]

Appendix B. Vision Transformer Auto-Encoder

Results of training Vision Transformer (ViT) based auto-encoder. The Transformer based auto-encoder did not perform well for our task. In the game *Skiing* a majority of the pixels are white and there is little variation between images in the training set. The model was not able to obtain valid reconstructions and would return an average of the data regardless of the input as demonstrated in Fig. 14. Increasing training steps did not seem to solve this problem as loss plateaued after 10k steps(Fig. 15).

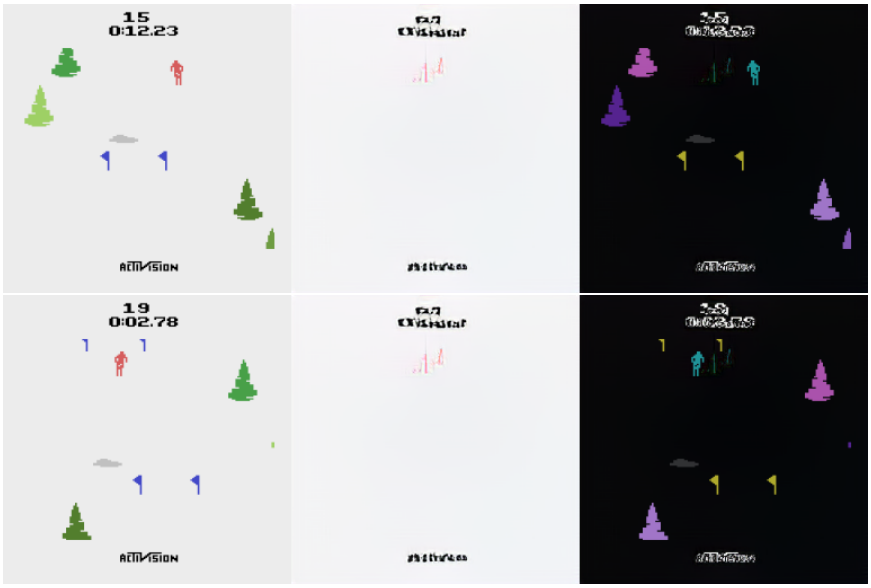


Figure 14: *Left*: Input image; *Middle*: Reconstructed image; *Right*: Difference between original and reconstructed image.

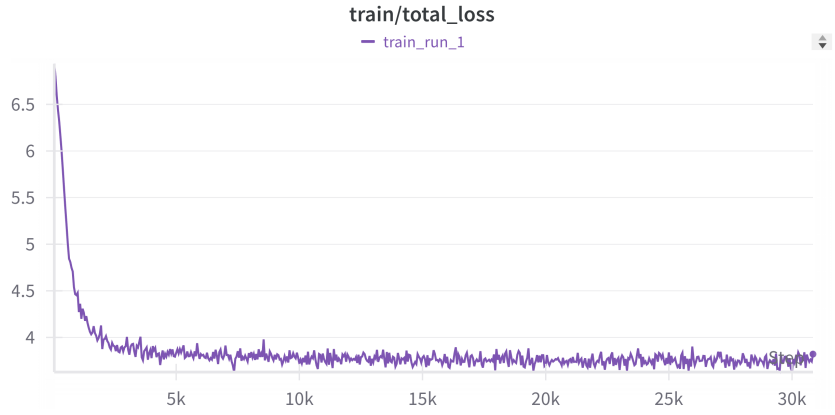


Figure 15: Vision Transformer (ViT) Training Curve

Appendix C. Multi-head Self-Attention

$$\text{MSA}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{23}$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{24}$$

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{25}$$

Where:

- Q, K, and V are the query, key, and value matrices
- W_i^Q, W_i^K, and W_i^V are learned parameter matrices for the i-th head
- W^O is the output projection matrix
- h is the number of attention heads
- d_k is the dimension of the keys

Appendix D. Multi-Layer Perceptron (MLP)

The forward pass of a Multi-Layer Perceptron with L layers can be expressed as:

$$h_l = f_l(W_l h_{l-1} + b_l), \quad \text{for } l = 1, \ldots, L \tag{26}$$

$$y = h_L \tag{27}$$

Where:

- h_l is the output of the l-th layer
- W_l is the weight matrix for the l-th layer
- b_l is the bias vector for the l-th layer
- f_l is the activation function for the l-th layer
- $h_0 = x$ is the input to the network
- y is the final output of the network

Appendix E. Feed-Forward Network (FFN)

A Feed-Forward Network, often used in transformer architectures, can be described as:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{28}$$

Where:

- x is the input to the FFN
- W_1 and W_2 are weight matrices
- b_1 and b_2 are bias vectors
- $\max(0, \cdot)$ is the ReLU activation function

The dimensionality of the FFN is typically as follows:

- Input x: $[d_{\text{model}}]$
- After first linear layer: $[d_{\text{ff}}]$
- Output: $[d_{\text{model}}]$

Where d_{model} is the model dimension and d_{ff} is the feed-forward dimension, usually defined as $d_{\text{ff}} = 4d_{\text{model}}$.

Appendix F. Prediction Head without Softmax

A variant of the standard transformer prediction head can be formulated without applying softmax. In this case, it is a simple linear layer

$$P(Y|X) = h_L W_p^T + b_p \tag{29}$$

Where:

- h_L represents the final hidden state from the transformer encoder
- W_p is the prediction head weight matrix
- $W_p \in \mathbb{R}^{C \times d_m}$ where C is the vocab size and d_m is the model dimension
- b_p is the bias

References

Eloi Alonso, Adam Jelley, Vincent Micheli, Anssi Kanervisto, Amos Storkey, Tim Pearce, and François Fleuret. Diffusion for world modeling: Visual details matter in atari, 2024. URL https://arxiv.org/abs/2405.12399.

Satwik Bhattamishra, Arkil Patel, and Navin Goyal. On the computational power of transformers and its implications in sequence modeling, 2020. URL https://arxiv.org/abs/2006.09286.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

Jake Bruce, Michael Dennis, Ashley Edwards, Jack Parker-Holder, Yuge Shi, Edward Hughes, Matthew Lai, Aditi Mavalankar, Richie Steigerwald, Chris Apps, Yusuf Aytar, Sarah Bechtle, Feryal Behbahani, Stephanie Chan, Nicolas Heess, Lucy Gonzalez, Simon Osindero, Sherjil Ozair, Scott Reed, Jingwei Zhang, Konrad Zolna, Jeff Clune, Nando de Freitas, Satinder Singh, and Tim Rocktäschel. Genie: Generative interactive environments, 2024. URL https://arxiv.org/abs/2402.15391.

Huiwen Chang, Han Zhang, Lu Jiang, Ce Liu, and William T. Freeman. Maskgit: Masked generative image transformer, 2022. URL https://arxiv.org/abs/2202.04200.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. URL https://arxiv.org/abs/1810.04805.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021. URL https://arxiv.org/abs/2010.11929.

Patrick Esser, Robin Rombach, and Björn Ommer. Taming transformers for high-resolution image synthesis, 2021. URL https://arxiv.org/abs/2012.09841.

William Fleshman and Benjamin Van Durme. Toucan: Token-aware character level language modeling, 2023. URL https://arxiv.org/abs/2311.08620.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014. URL https://arxiv.org/abs/1406.2661.

R. Gray. Vector quantization. *IEEE ASSP Magazine*, 1(2):4–29, 1984. doi: 10.1109/MASSP.1984.1162229.

David Ha and Jürgen Schmidhuber. World models. 2018. doi: 10.5281/ZENODO.1207631. URL https://zenodo.org/record/1207631.

Hado Hasselt. Double q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010. URL https://proceedings.neurips.cc/paper_files/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf.

Hado Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML 2015)*, pages 2094–2102. JMLR.org, 2015.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models, 2020. URL https://arxiv.org/abs/2006.11239.

Jonathan Ho, Tim Salimans, Alexey Gritsenko, William Chan, Mohammad Norouzi, and David J. Fleet. Video diffusion models, 2022. URL https://arxiv.org/abs/2204.03458.

Wenyi Hong, Ming Ding, Wendi Zheng, Xinghan Liu, and Jie Tang. Cogvideo: Large-scale pretraining for text-to-video generation via transformers, 2022. URL https://arxiv.org/abs/2205.15868.

Seung Wook Kim, Yuhao Zhou, Jonah Philion, Antonio Torralba, and Sanja Fidler. Learning to simulate dynamic environments with gamegan, 2020. URL https://arxiv.org/abs/2005.12126.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2022. URL https://arxiv.org/abs/1312.6114.

Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing, 2018. URL https://arxiv.org/abs/1808.06226.

Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction, 2020. URL https://arxiv.org/abs/1802.03426.

Vincent Micheli, Eloi Alonso, and François Fleuret. Transformers are sample-efficient world models, 2023. URL https://arxiv.org/abs/2209.00588.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015. doi: 10.1038/nature14236.

Duy-Kien Nguyen, Mahmoud Assran, Unnat Jain, Martin R. Oswald, Cees G. M. Snoek, and Xinlei Chen. An image is worth more than 16x16 patches: Exploring transformers on individual pixels, 2024. URL https://arxiv.org/abs/2406.09415.

Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision, 2022. URL https://arxiv.org/abs/2212.04356.

Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation, 2021. URL https://arxiv.org/abs/2102.12092.

Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models, 2022. URL https://arxiv.org/abs/2112.10752.

Masaki Saito, Eiichi Matsumoto, and Shunta Saito. Temporal generative adversarial nets with singular value clipping, 2017. URL https://arxiv.org/abs/1611.06624.

Uriel Singer, Adam Polyak, Thomas Hayes, Xi Yin, Jie An, Songyang Zhang, Qiyuan Hu, Harry Yang, Oron Ashual, Oran Gafni, Devi Parikh, Sonal Gupta, and Yaniv Taigman. Make-a-video: Text-to-video generation without text-video data, 2022. URL https://arxiv.org/abs/2209.14792.

Dani Valevski, Yaniv Leviathan, Moab Arar, and Shlomi Fruchter. Diffusion models are real-time game engines, 2024. URL https://arxiv.org/abs/2408.14837.

Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning, 2018. URL https://arxiv.org/abs/1711.00937.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

Ruben Villegas, Mohammad Babaeizadeh, Pieter-Jan Kindermans, Hernan Moraldo, Han Zhang, Mohammad Taghi Saffar, Santiago Castro, Julius Kunze, and Dumitru Erhan. Phenaki: Variable length video generation from open domain textual description, 2022. URL https://arxiv.org/abs/2210.02399.

Carl Vondrick, Hamed Pirsiavash, and Antonio Torralba. Generating videos with scene dynamics, 2016. URL https://arxiv.org/abs/1609.02612.

Mingze Wang and Weinan E. Understanding the expressive power and mechanisms of transformer for sequence modeling, 2024. URL https://arxiv.org/abs/2402.00522.

Yaohui Wang, Xinyuan Chen, Xin Ma, Shangchen Zhou, Ziqi Huang, Yi Wang, Ceyuan Yang, Yinan He, Jiashuo Yu, Peiqing Yang, Yuwei Guo, Tianxing Wu, Chenyang Si, Yuming Jiang, Cunjian Chen, Chen Change Loy, Bo Dai, Dahua Lin, Yu Qiao, and Ziwei Liu. Lavie: High-quality video generation with cascaded latent diffusion models, 2023. URL https://arxiv.org/abs/2309.15103.

C.J.C.H. Watkins. Learning from delayed rewards. In *Machine Learning, Proceedings of the 1989 International Workshop on*, pages 286–291, San Mateo, CA, 1989. Morgan Kaufmann.

Bob Whitehead. Skiing. Video game, 1980. Released for the Atari 2600 console.

Mingxing Xu, Wenrui Dai, Chunmiao Liu, Xing Gao, Weiyao Lin, Guo-Jun Qi, and Hongkai Xiong. Spatial-temporal transformer networks for traffic flow forecasting, 2021. URL https://arxiv.org/abs/2001.02908.

Wilson Yan, Yunzhi Zhang, Pieter Abbeel, and Aravind Srinivas. Videogpt: Video generation using vq-vae and transformers, 2021. URL https://arxiv.org/abs/2104.10157.

Jiahui Yu, Yuanzhong Xu, Jing Yu Koh, Thang Luong, Gunjan Baid, Zirui Wang, Vijay Vasudevan, Alexander Ku, Yinfei Yang, Burcu Karagol Ayan, Ben Hutchinson, Wei Han, Zarana Parekh, Xin Li, Han Zhang, Jason Baldridge, and Yonghui Wu. Scaling autoregressive models for content-rich text-to-image generation, 2022. URL https://arxiv.org/abs/2206.10789.

Qihang Yu, Mark Weber, Xueqing Deng, Xiaohui Shen, Daniel Cremers, and Liang-Chieh Chen. An image is worth 32 tokens for reconstruction and generation, 2024. URL https://arxiv.org/abs/2406.07550.

Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric, 2018. URL https://arxiv.org/abs/1801.03924.