VideoWorld: Exploring Knowledge Learning from Unlabeled Videos

Supplementary Material

A. Implementation Details

Training details. We present the detailed training configurations of latent dynamics model and auto-regressive transformer in Tab. A.1. The batch size is set to 256 for Go and 32 for CALVIN, requiring approximately 4 and 2 days of training on 8 A100 GPUs, respectively.

Latent dynamics model. The decoder can be divided into three parts based on three functions: *i*) using an encoder to causally extract image features; *ii*) using learnable embeddings to extract change information from the extracted features; *iii*) using a decoder to causally reconstruct video frames based on image feature of the initial frame and the learned embeddings. We present the PyTorch-style pseudocodes for the overall LDM and each part in Alg. 1.

Robotics action execution. Our model generates a set of latent codes $\{\hat{z}_t^h\}_{h=1}^H$ and next frame prediction $\hat{x}_{t+1}$ based on language instruction and input sequence $x_{1:t}$. We feed $\{\hat{z}_t^h\}_{h=1}^H$, $\hat{x}_{t+1}$ and x_t into the Inverse Dynamics Model to obtain corresponding action. The inferred action is executed in an open-loop manner: after predicting each action, we input it into the environment engine to obtain a new observation x_{t+1}, which is then appended to the input sequence for the next prediction cycle. In Fig. B.1, we visualize the model's next-frame predictions and the actual control results of the robotic arm. We find that the model's next-frame predictions align with the task's execution intent and effectively control the robotic arm to complete the tasks. We visualize the latent codes at different time steps during testing and analyze them in Sec. 5.5.

Inverse dynamics model. The training objective of IDM is to predict the control signals for the robotic arm, represented as a 7-dimensional vector that includes the arm's displacement along the XYZ axes, Euler angles, and the gripper's open/close state. Specifically, IDM uses a ResNet-18 to process the predicted frame, applying a global average pooling layer to the penultimate feature map to obtain a feature vector of size (1, 512). Simultaneously, a shared MLP transforms the feature vectors of each latent code into a size of (H, 512), where H, as defined in Sec. 3.2, represents the number of learnable embeddings or prediction steps in LDM. Then, the features from the latent codes and the generated frames are concatenated into a vector of size ($H+1$, 512), which is then passed through another MLP with dimensions (512, 7) followed by a global average pooling layer to produce the final 7 control signals. IDM is trained using AdamW with a learning rate of 1e-4 for a total of 1 million steps, with mean squared error as the objective.

Config	LDM	AR Transformer
optimizer	AdamW	AdamW
base learning rate	5.4e-5	3.0e-5
weight decay	0.01	0
optimizer monmentum	β_1, β_2=0.5,0.9	β_1, β_2=0.9,0.98
batch size	16	256(Go),32(CALVIN)
learning rate schedule	WarmipDecayLR	WarmipDecayLR
warmup iterations	3e+4	3e+4
max iterations	1e+5	1e+6
augmentations	None	None
Training Loss	L2 loss	Cross Entropy loss
Training Target	Reconstruction	Next token pred.

Table A.1. **Training configurations** for the latent dynamics model (LDM) and auto-regressive (AR) transformer.

Prediction Target	Go		CALVIN		
	Act-Value	Act-Acc.	Push	Open/Close	Turn on/off
video	47.5	44.3	12.7	20.8	15.6
code	73.0	78.6	47.2	70.0	65.1
code/video	73.9	80.9	50.3	71.1	69.7

Table A.2. **Latent code prediction only** with 50M parameters.

RLBench evaluation. In Sec. 5.4, we test VideoWorld's ability to perform tasks in two different environments: CALVIN and RLBench. For CALVIN, we maintain the original task settings. For RLBench, we generate 20k trajectory data using a script. RLBench uses the same robotic arm and action space as CALVIN, but the environment appearance and task settings differ significantly. We use the combined RLBench and CALVIN datasets to train both the LDM and Transformer, while IDM is trained separately in each environment. Fig. D.5 shows VideoWorld performing robotic tasks in RLBench environment.

B. Extended Analysis and Ablative Studies

B.1. Understanding Learned Knowledge with LDM

In Sec. 5.5 of the main text, we analyze the role of LDM in the Go scenario, showing that latent codes contain information about board changes and can model long-range changes progressively. Similar findings are observed in the robotic scenario. We visualize the predicted latent codes during inference across different tasks in Fig. B.1. Here, $H = 9$, meaning the transformer generates 9 latent codes per time step, corresponding to 9 prediction steps. As shown, the latent codes for different prediction steps are grouped by task type, indicating that they capture task-relevant dynamics. Codes for steps 1–4 show greater overlap, likely because they focus on fine-grained displacements shared across tasks. In contrast, steps 5–9 show more distinct separation by task type, highlighting the model's abil-

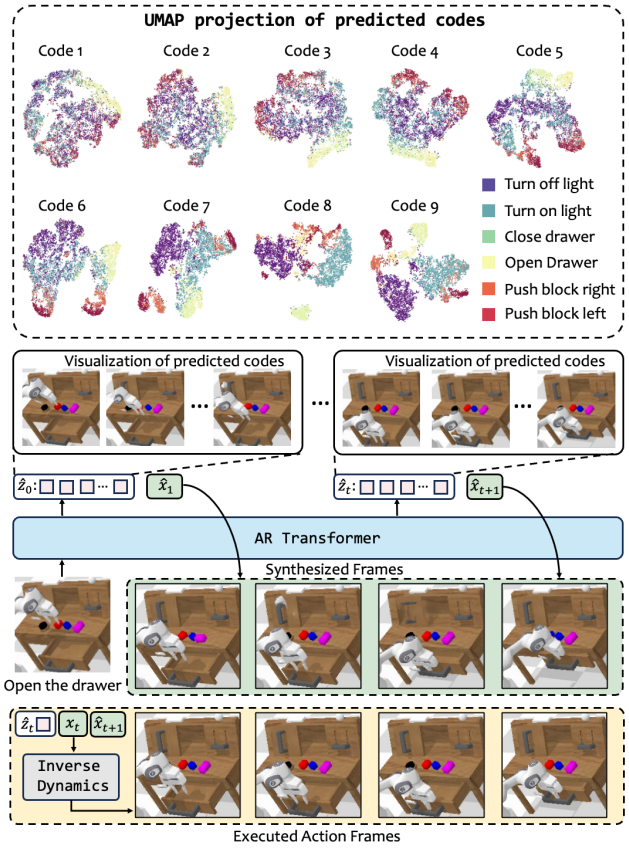


Figure B.1. **Illustration of robotic manipulation** and UMAP projection of the predicted latent code during inference. Latent codes are visualized through the LDM decoder. The UMAP projection illustrates the 9 predicted latent codes (i.e. $H = 9$) across different tasks, with each point color-coded by task type. Visualizations with a yellow background show the model's actual robotic arm control during inference, while those with a green background represent the model's next-frame predictions during training.

ity to progressively capture long-range changes specific to each task. LDM increases TFlops by only 3.8% as its codes are far fewer than frame tokens.

B.2. Ablative Studies

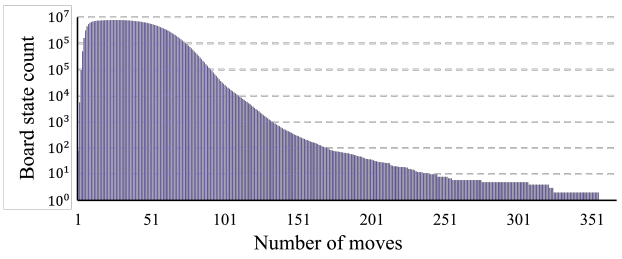
Latent code prediction only. To demonstrate the necessity of jointly predicting the latent code $\{\hat{z}_t^h\}_{h=1}^H$ and next frame $\hat{x}_{t+1}$ given $x_{1:t}$, we remove the supervision for the next frame $\hat{x}_{t+1}$ during training. In this setup, frames are only used as input, and only the latent codes are subject to the CE loss. In this case, we retrain an IDM that only receives latent code inputs. As shown in Tab. A.2, for both Go and robotic scenarios, using codes alone significantly improves performance, and incorporating next-frame prediction further enhances it. We hypothesize this is because next-frame prediction enhances the model's understanding of the environment and helps generate more accurate codes.

Algorithm 1: Pseudo codes of LDM.

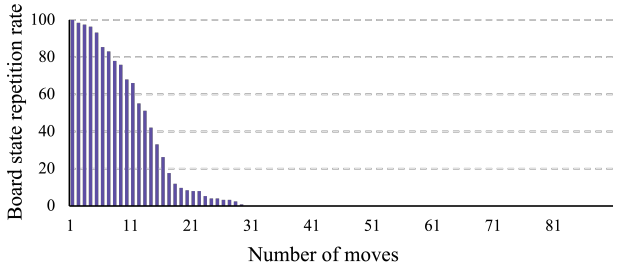
```
# Inputs: video:The first frame and its subsequent
  H frames [1+H, h, w, 3];
# Variables: ldm_q: Learnable embeddings for H
  time spans [H,C]; image_pe:position embedding of
  image features;
# Functions: CrossAttention();MLP(); up_scale();
  down_scale(); FSQ(); Causal3DCNN()
def encoder(video):
    # video:video sequences.[1+H,h,w,3]
    # The encoder consists of a set of encoder
      layers, composed of Causal3DCNN and
      down_scale layers.
    f = Causal3DCNN(video);# f:[1+H,h,w,C]
    for layer in encoder_layers:
        # process and downsample features using
          Causal3DCNN and down_scale.
        f = layer(f)
    # Capture dynamic changes in video.
    z = ldm_qformer(f)
    return z, f[:0];# f:[1+H,h',w',C]. z:[H,C]
def ldm_qformer(f):
    # f:features of each frame.[1+H, h', w', C]
    q_list = []
    for h in range(H):
        query = ldm_q[h];# query:[1,C]
        f_h = f[:(1+h)];# f_h:[1,h',w',C]
        key = f_h + image_pe;# key:[1,h',w',C]
        q_h = CrossAttention(q=query, k=key,
          v=f_h);# q_h:[1,C]
        q_h = MLP(q_h);# q_h:[1,C]
        q_list.append(q_h)
    q_list = stack(q_list);# q_list:[H,C]
    return q_list
def decoder(z, first_f):
    # z:ldm query embeddings that captures change
      information in video frames.[H, C];
      first_f:image features of first frame.[1, h',
      w', C]
    # The decoder consists of a set of decoder
      layers, composed of Causal3DCNN and up_scale
      layers.
    z = repeat_interleave(z, h', dim=-2)
    z = repeat_interleave(z, w', dim=-1)
    rec_video = cat(first_f, z);# [1+H, h', w', C]
    for layer in decoder_layers:
        # process and upsample features using
          Causal3DCNN and up_scale.
        rec_video = layer(rec_video)
    rec_video = Causal3DCNN(rec_video)
    return rec_video;# rec_video:[1+H, h, w, C]
# Main Function
def latent_dynamics_model(video):
    # video:video sequences
    # extract change information from video frames
    z, first_f = encoder(video)
    z = FSQ(z);# quantize z using FSQ
    # LDM training stage
    if is_train:
        # obtain reconstructed video frames.
        rec_video = decoder(z, first_f)
        # we train ldm using mse loss.
         loss = MSE(rec_video, video)
        return loss
    return z
```

C. Details on Video-GoBench

In this section, we systematically analyze Video-GoBench. The benchmark includes many unique board states, offering



(a) Distribution of board state count in training set



(b) Repetition rate during online battle.

Figure C.2. **Dataset statistics**. Best viewed digitally.

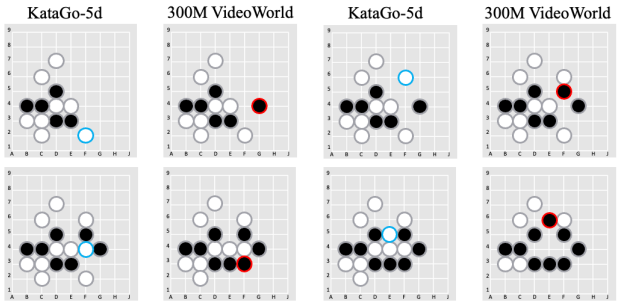
extensive references for the model to learn from. Moreover, due to the combinatorial explosion effect, the proportion of repeated board states—those encountered during inference that also appear in the training set—decreases sharply as the game progresses when our model competes against reinforcement learning agents. This ensures that good performance cannot rely on memorizing training scenarios, highlighting the model's generalization ability.

Board state count. The Video-GoBench training set contains approximately 400M unique board states. We analyze their distribution based on move numbers, i.e., the number of stones each state contains. As shown in Fig. C.2a, the data features a diverse range of board states, primarily concentrated within the first 100 moves, with significantly fewer states beyond that.

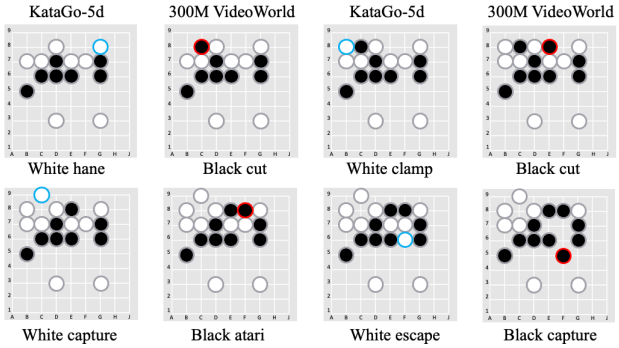
Board state repetition rate. We collect 400 game records between our model and KataGo-9d, calculating the overlap rate of board states with the training set across different move numbers. As shown in Fig. C.2b, the repetition rate drops sharply as the games progress. By move 30, it reaches zero, with all games continuing beyond this point. This eliminates the possibility of relying on memory alone to achieve high performance.

D. Additional Visualizations

We provide more visualizations of VideoWorld performing Go and robotic tasks in Fig. D.3 and Fig. D.4, respectively. The Go visualizations are from matches between our 300M VideoWorld and KataGo-5d. With comparable Elo scores, these matches fairly demonstrate of how the model applies its learned knowledge. In Fig. D.3a, our model captures op-



(a) Capture opponent's stones



(b) Sacrificing short-term gains to capture more opponent stones.

Figure D.3. **Visualizations of learned Go Strategies**. our model captures opponent stones using the squeeze tactic and self-sacrifice tactic. New black stones are red, new white stones are blue.

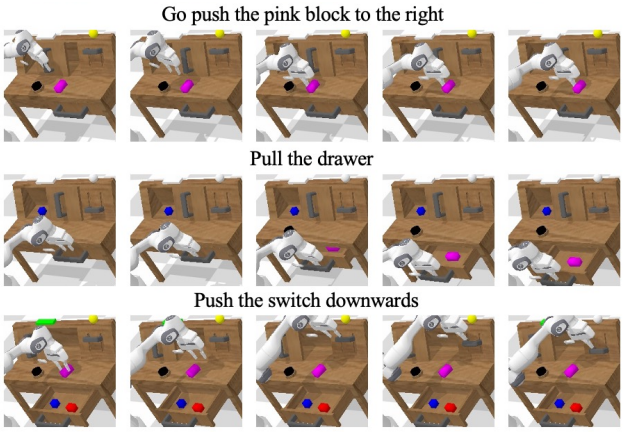


Figure D.4. **Visualizations of performing CALVIN tasks.**

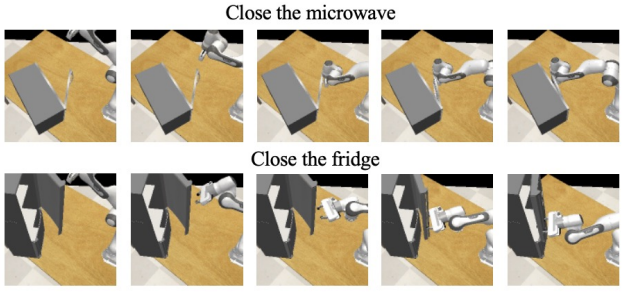


Figure D.5. **Visualizations of performing RLBench tasks.**

ponent stones using the squeeze tactic. In Fig. D.3b, VideoWorld deliberately sacrifices a single stone, prompting the opponent to capture it, thereby creating an opportunity to capture more of the opponent's stones. This highlights the model's ability to prioritize long-term planning over short-term gains. For the robotic scenario, we show the model's actual control results for the robotic arm, demonstrating its understanding of manipulation tasks and ability to execute them effectively.