COMP4434 Group Project - Group 5

WANG Yuqi
PolyU, COMP
[redacted]
@connect.polyu.hk

YANG Xikun
PolyU, COMP
[redacted]
@connect.polyu.hk

CUI Mingyue
PolyU, COMP
[redacted]
@connect.polyu.hk

Abstract

Sample efficiency remains a critical challenge in reinforcement learning (RL). While state-of-the-art world models like DreamerV3 achieve remarkable efficiency, their complexity motivates the exploration of simpler alternatives. This project investigates the trade-offs between model complexity and performance in efficient RL. We implement and evaluate four distinct approaches: (1) the DreamerV3 world model, serving as a performance upperbound; (2) classical RL baselines employing Q-Learning with function approximation; (3) the widely-used model-free deep RL algorithm, Proximal Policy Optimization (PPO); and (4) a simplified Dreamer-like method utilizing contrastive learning instead of reconstruction. Our comparative analysis reveals that although the proposed contrastive approach does not fully match the performance capabilities of DreamerV3, it significantly outperforms both the classical baselines and deep RL baseline in sample-scarce environments. These findings suggest promising directions for developing less complex yet sample-efficient model-based RL agents.

1 Introduction

Reinforcement learning has enabled intelligent agents to make decisions in novel situations by learning a representation of the environment through interactions. Traditional model-free reinforcement learning methods learns this representation through large amounts of trial-and-error. While this approach has enabled some of the most impressive achievements in realms such as the game of Go, Dota, and Starcraft, as well as the recent success of reasoning large language models, its brute-force nature makes it extremely sample inefficient [1], [2], [3]. A typical RL agent often requires millions of samples to learn simple tasks, limiting its application to high-dimensional and complex environments.

In recent years, world models have emerged as a promising approach to overcome these issues. It reduces the amount of samples required by explicitly learning the environment's dynamics and representation. The learned model can then be used to generate large quantities of imagined data, or be used as a simulator for planning. In the paper we chose, DreamerV3, world model has been shown to outperform specialized expert algorithms across diverse domains with fixed parameters [4].

While DreamerV3 excels in both performance and efficiency, it is computationally intensive due to expensive reconstruction and probabilistic modeling needed. We present **WM-PPO**, a reconstruction-free contrastive learning world model combined with a PPO agent. While

WM-PPO does not match the peak performance of DreamerV3, it outperforms the PPO baseline by large margins, while being much more computationally efficient.

Our project work can be summarized as followings:
1. Reproduced the results of DreamerV3 and achieving similar metrics.
2. Implementated a traditional machine learning method using Q-Learning for value iteration and linear regression (or SVM) for value estimation
3. Reproduced a deep RL algorithm, specifically PPO
4. Created a reconstruction-free world model using contrastive learning where a PPO agent is trained in its imagined rollouts, similar to DreamerV3.

In the following sections, we will first introduce the background knowledge needed to understand both the traditional ML baseline and deep RL baseline in Section 2 , followed by their implementation details in Section 3 . Our proposed method is discussed in depth in Section 4 . The results of all aforementioned models are listed in Section 5 .

2 Background

2.1 Q-Learning

Q-Learning is a model-free reinforcement learning algorithm for estimating the value function of an optimal policy[5]. A value function $V^\pi(s)$ tells us the expected discounted return if we were to follow policy π from state s onwards.

$$\begin{aligned}
V^\pi(s_0) &= \mathbb{E}_\pi[\tau \mid s] \\
&= R(s_0) + \mathbb{E}[\gamma R(s_1) + \gamma^2 R(s_2) + ... \mid s_0 = s, \pi] \\
&= R(s_0) + \mathbb{E}[\gamma V^\pi(s_1) \mid s_0 = s, \pi] \\
&= R(s_0) + \gamma \sum_{a_0 \in A} \sum_{s' \in S} \pi(a|s) p(s'|s_0, a_0) V^\pi(s') \\
&= R(s_0) + \gamma \sum_{s' \in S} p_{a_0 \sim \pi(s_0)}(s'|s_0, a_0) V^\pi(s')
\end{aligned}$$

However, knowning the value function alone does not allow us to determine the optimal action at each step unless a model of the environment is available. Q-Learning solves this by estimating an action-value function $Q^{\pi(s,a)}$ that gives the expected return for taking action a from state s and following policy π thereafter.

$$\begin{aligned}
V^\pi(s) &= R(s) + \gamma \sum_{a \in A} \sum_{s' \in S} \pi(a|s) p(s'|s, a) V^\pi(s') \\
&= R(s) + \gamma \sum_{a \in A} \pi(a|s) \sum_{s' \in S} p(s'|s, a) V^\pi(s') \\
&= R(s) + \gamma \sum_{a \in A} \pi(a|s) Q^\pi(s, a) \\
\therefore Q^\pi(s, a) &= \sum_{s' \in S} p(s'|s, a) V^\pi(s') = \mathbb{E}[\tau \mid s, a]
\end{aligned}$$

2.2 Dreamer

Dreamer is a series of work that attempts to learn an accurate enough world model that allows a reinforcement learning agent to learn purely from its synthetic data. The world model essentially acts as a virtual simulator that extrapolates the existing training data, thereby enabling sample efficiency.

DreamerV3 is the latest iteration of the Dreamer series of work, which uses a variational autoencoder (VAE) to encode the observation into a categorical latent space, which is then regularized by a KL divergence loss with the dynamics prior[4]. The VAE is trained to reconstruct the observation, while the KL loss encourages the latent to encode the only information that is relevant to predictions.

Algorithm 1: Dreamer

Initialize dataset $\mathcal{D}$ with S random episodes; network w/ parameters θ, φ, ψ randomly.

while not converged **do**
  for update step $c = 1..C$ **do**
    Draw B data sequences $\{(a_t, o_t, r_t)\}_{t=k}^{k+L} \sim \mathcal{D}$.
    Compute model states $s_t \sim p_\theta(s_t \mid s_{t-1}, a_{t-1}, o_t)$.
    Update θ using representation learning.
    Imagine trajectories $\{(s_\tau, a_\tau)\}_{\tau=t}^{t+H}$ from each s_t
    Predict rewards $\mathbb{E}(q_\theta(r_\tau \mid s_\tau))$ and values $v_\psi(s_\tau)$.
    Compute value estimates $V_\lambda(s_\tau)$ via Equation 6.
    Update $\varphi \leftarrow \varphi + \alpha \nabla_\varphi \sum_{\tau=t}^{t+H} V_\lambda(s_\tau)$.
    Update $\psi \leftarrow \psi - \alpha \nabla_\psi \sum_{\tau=t}^{t+H} \frac{1}{2} \left\| v_\psi(s_\tau) - V_\lambda(s_\tau) \right\|^2$.
  end for
  for time step $t = 1..T$ **do**
    Compute $s_t \sim p_\theta(s_t \mid s_{t-1}, a_{t-1}, o_t)$ from history.
    Compute $a_t \sim q_\varphi(a_t \mid s_t)$ with the action model.
    Add exploration noise to action.
    $r_t, o_{t+1} \leftarrow \text{env.step}(a_t)$.
    Add experience to dataset $\mathcal{D} \leftarrow \mathcal{D} \cup \left\{(o_t, a_t, r_t)_{t=1}^{T}\right\}$
  end for
end while

Algorithm 1: Dreamer training procedure. Source: [6]

2.3 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a policy gradient method that, instead of using a KL trust region to constrain the policy updates as in TRPO, uses a clipped surrogate objective. Just like TRPO, its standard implementation follows an actor-critic framework. In which, critic is a value network $V_\varphi(s)$ that aims to accurately estimate the expected return under state s; the actor is a policy network $\pi_\theta(a|s)$ that aims to output actions that maximizes the expected reward according to $V_\varphi(s)$.

Traditionally, the actor network $\pi_\theta(a|s)$ is trained to directly maximize the expected advantage $\mathbb{E}[A^{\pi_{\theta_k}}(s,a)]$, where $A^{\pi_{\theta_k}}(s,a) = Q^{\pi_{\theta_k}}(s,a) - V^{\pi_{\theta_k}}(s)$. But this can leads to unstable training, as the locally optimal policy improvement direction (gradient) might actually lead to worse performance.

To overcome this, we need to ensure that the new policy π_θ monotonically increases the expected advantage $A^{\pi_{\theta_k}}(s,a)$ over the old policy π_{θ_k}. If it can be guaranteed that a new policy's generated trajectory will yield higher advantage estimation according to the old critic network, then it can be proven that the new policy will converge. However, evaluating the new policy using the value function of the old policy causes a chicken-and-egg problem, because the new policy is the unknown variable.

$$\arg\max_\theta \mathbb{E}\left[\min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a)\right), \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\varepsilon, 1+\varepsilon\right)A^{\pi_{\theta_k}}(s,a)\right]$$

PPO solves this by using a surrogate objective that evaluates episodes from the old policy, essentially assuming that the new policy's state visitation will be similar to the old policy's. To meet such assumption, the difference between π_θ and π_{θ_k} is clipped between a hyperparameter ε.

3 Baselines

3.1 Machine Learning Baseline

Algorithm 2: Q-Learning with Traditional Machine Learning Models

Initialize Q_θ with *SVM* or *Linear Regression*
Initialize replay buffer $\mathcal{D} \leftarrow \emptyset$
for episode $= 1, M$ **do**
  for $t = 1, T$ **do**
    With probability ε select random action a_t
    otherwise select $a_t = \arg\max_a Q_\theta(s_t, a)$
    Execute action a_t in emulator and observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample batch of B transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
    Set $y_j = r_j$ **if** s_{j+1} is terminal **otherwise** $y_j = r_j + \gamma \max_{a'} Q_\theta(s_{j+1}, a')$
    Update Q_θ using $\mathcal{L}_{\text{MSE}} = \left(y_j - Q_\theta(s_j, a_j)\right)^2$
  end for
end for

Algorithm 2: Q-Learning with Traditional ML Models. Source: [7]

3.1.1 Action Discretization

The *DeepMind Control Suite* contains purely continuous control tasks. However, the default implementation of Q-Learning can only handle discrete action spaces. Therefore, to properly apply Q-Learning to the tasks, discretization is required. Specifically, each action dimension is divided into N_{bins} intervals according to the configured *action_bins* number and generate all possible action combinations. Formally, let the action space be a continuous space $A = \{a_1, a_2, ..., a_n\} \subset \mathbb{R}^m$. Each dimension of the space a_i is then discretized into $a_i = \left\{b_1, ..., b_{N_{\text{bins}}}\right\}$, where $b_j = j \cdot \frac{1}{N_{\text{bins}}-1}$.

However, this can lead to a large number of combinations, for high dimensional continous action space. For those tasks, N_{sample} actions are uniformly drawn from the entire action space. In our code, the method *_create_discrete_actions* implements this process, checking the type of action space and generating the appropriate discrete actions following the aforementioned rules.

3.1.2 Epsilon-Greedy Strategy

The ε-greedy strategy is a popular method used to balance *exploration* and *exploitation* in reinforcement learning. *Exploration* represents that the agent tries new actions to discover their potential rewards, which is crucial for learning about the environment. *Exploitation* represents that the agent selects the action that it believes has the highest expected reward based on its current knowledge.

ε is a parameter that determines the probability of choosing a random action (exploration) versus the best-known action (exploitation). Formally, a ε-greedy policy $\pi_\varepsilon(a|s)$ chooses the optimal action with probability $\frac{\varepsilon}{|A|} + 1 - \varepsilon$ and any other actions with a uniform probability $\frac{\varepsilon}{|A|}$.

$$\pi_\varepsilon(a|s) = \begin{cases} \dfrac{\varepsilon}{|A|} + 1 - \varepsilon & \text{if } a = \arg\max\limits_a Q(s,a) \\ \dfrac{\varepsilon}{|A|} & \text{otherwise} \end{cases}$$

3.1.3 Update the Q-Function

First, the agent interacts with the environment, collects state transition samples $(s_t, a_t, r_t, s_{t+1}, \text{done})$, and stores them in an experience replay buffer. When it's time to update the Q-function, a batch of samples is randomly drawn from the buffer. This batch sampling mechanism (Experience Replay) breaks the temporal correlation between samples, making training more stable while improving sample efficiency. For each experience sample, we first convert the observation state into a flattened *feature vector*. This feature vector is passed into the Q-function approximator which predicts $\min\big(|A| \times N_{\text{bins}}, N_{\text{sample}}\big)$ Q-values, each corresponding to an discretized action's Q-value.

Next, a temporal difference (TD) target value is calculated for each transition sample:

$$y = \begin{cases} r & \text{if terminal} \\ r + \gamma \max_{a'} Q(s', a') & \text{otherwise} \end{cases}$$

For non-terminal states, we find the maximum Q-value across all possible actions in the next state, multiply by the discount factor gamma, and add the immediate reward.

Finally, the Q-function, which is a SVM or Linear Regression model implemented via sklearn under the hood is fitted by minimizing the MSE between the predicted Q-values and the TD target:

$$\mathcal{L}_{\text{MSE}} = \big(y_j - Q_\theta(s_j, a_j)\big)^2$$

3.2 Deep Learning Baseline

Algorithm 3: PPO, Actor-Critic Style

for iteration$= 1, 2, \ldots$ **do**
  for actor$= 1, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Compute policy difference, $r_{t(\theta)} \leftarrow \pi_\theta(a_t|s_t)/\pi_{\theta_{\text{old}}}(a_t|s_t)\hat{A}_t$
  Minimize $\mathcal{L}_{\text{CLIP}}(\theta) = \min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t\right)$
end for

Algorithm 3: PPO training procedure. Source: [8]

For our deep learning baseline, we chose the Proximal Policy Optimization (PPO) algorithm, which is a popular policy gradient method that uses a clipped surrogate objective to update the policy. For simplicity, we directly adopt the PPO implementation from https://github.com/vwxyzjn/cleanrl [9] and slightly modified it to fit our needs.

The CleanRL implementation is a single-file PPO implementation that is designed to be simple and easy to understand[10]. In this implementation, serveral slight improvements were made compared to the original author's approach, such as gradient clipping and learning rate annealing. The performance of the PPO baseline is shown in Section 5.

4 Method

4.1 Hypothesis

In DreamerV3, an expensive variational autoencoder (VAE) is used to encode the observation into a latent space, which is then regularized by a KL divergence loss with the dynamics prior. While these operations allows DreamerV3 to learn robust and accurate representations for the actor-critic agent to learn purely from latent space imaginations, we hypothesize that we can achieve lower but still competitive performance without the expensive reconstruction and probabilistic modeling.

4.2 Design

We introduce **WM-PPO**, short for **W**orld **M**odel-**PPO**, a simplified Dreamer-like method that learns its world model through contrastive learning instead of reconstruction. The models starts by sampling a batch of B trajectories $\tau_{i=1\ldots B} = \{s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_T, a_T, r_T\}_i$ from the replay buffer $\mathcal{D}$. Then, using an encoder $q_\varphi : o_t \to z_t$ that maps each raw observation to latent representations. This gives us a batch of B latent trajectories $\boldsymbol{z}_{i=1\ldots B} = \{z_0, z_1, \ldots, z_T\}_i$, which we will from now on refer to as the *anchor latents*. Next, the dynamics model $p_\varphi : z_t, a_t \to z_{t+1}$ predicts an imagined latent trajectory $\hat{\boldsymbol{z}} = \{\hat{z_1}, \hat{z_2}, \ldots, \hat{z_T}\}$ from the first latent z_0 and actions $a_{0\ldots T-1}$.

Here, we make the critical assumption that the latent imagination is strictly worse than the latent directly encoded from the observation. This allows us to treat imagined data as negative samples. Given this, the model can then be optimized using a constrastive learning strategy that resembles InfoNCE, except we use batches of imagined trajectories $\hat{\boldsymbol{z}}$ as

negatives instead of the anchor latents $\boldsymbol{z}$. Specifically, we train the model to minimize the difference between $(z_i, \hat{z}_i)$ using cross entropy; then, the softmax function will implicitly push away $\left(z_i, \hat{z}_j\right)$ for $j \neq i$. Formally, the contrastive loss can be written as:

$$\mathcal{L}_{\text{contrast}}(\boldsymbol{z}, \hat{\boldsymbol{z}}) = -\mathbb{E}_i \left[\log \left(\frac{\exp(s(\boldsymbol{z}_i, \hat{\boldsymbol{z}}_i)/\tau)}{\sum_j \exp\left(s\left(\boldsymbol{z}_i, \hat{\boldsymbol{z}}_j\right)/\tau\right)} \right) \right] \quad \text{where, } s(\boldsymbol{u}, \boldsymbol{v}) = \frac{\boldsymbol{u}^T \boldsymbol{v}}{\|\boldsymbol{u}\| \|\boldsymbol{v}\|}$$

Intuitively, given pairs of anchor and imagined latent trajectories $\boldsymbol{z}_i$ and $\hat{\boldsymbol{z}}_i$, a $T \times T$ cosine similarity matrix is computed between each pair of latent vectors. It is straightforward to see

Algorithm 4: WM-PPO Implementation

Initialize replay buffer $\mathcal{D}$, world model params θ, agent params φ.
Initialize optimizers Adam_θ, Adam_φ.
Prefill $\mathcal{D}$ with N_{prefill} episodes using random actions $a \sim \text{Uniform}()$.
while not converged **do**
  for update step $c = 1..C$ **do**
    for $i = 1..N_{\text{wm}}$ **do**
      Sample B trajectories $\{(o_t, a_t, r_t)\}_{t=k}^{k+L} \sim \mathcal{D}$.
      Encode anchor: $\boldsymbol{z} \leftarrow q_\theta(o_{t=1\ldots T})$
      Imagine latent: $\hat{\boldsymbol{z}} \leftarrow p_\theta(z_0, a_{0\ldots T-1})$
      Predict reward: $\hat{\boldsymbol{r}} \leftarrow r_\theta(\hat{\boldsymbol{z}})$.
      Compute: contrastive loss $\mathcal{L}_{\text{contrast}}(\boldsymbol{z}, \hat{\boldsymbol{z}})$
      Compute: reward loss $\mathcal{L}_{\text{reward}}(\hat{\boldsymbol{r}}, \boldsymbol{r})$.
      Compute world model loss $\mathcal{L}_{\text{WM}} = \mathcal{L}_{\text{contrast}} + c_r \mathcal{L}_{\text{reward}}$.
      Update $\theta \leftarrow \text{Adam}_\theta(\nabla_\theta \mathcal{L}_{\text{WM}})$.
    end for
    $\boldsymbol{Z}_{\text{flat}} : \mathbb{R}^{B \times L} \leftarrow \text{Flatten}(\boldsymbol{z})$
    Imagine $B \times L$ rollouts of H steps from $\boldsymbol{Z}_{\text{flat}}$
    Train PPO actor-critic π_φ and v_φ on $B \times L$ imagined rollouts
  end for
  $o_t = \text{env.reset}()$
  for $i = 1\ldots N_{\text{interact}}$ **do**
    while episode is not done **do**
      Encode $z_t = q_\theta(o_t)$
      Sample action $a_t \sim \pi_\varphi(\cdot \mid z_t)$
      Execute action $o_{t+1}, r_t, d_t = \text{env.step}(a_t)$
      Store (o_t, a_t, r_t, d_t) in $\mathcal{D}$
    end while
  end for
end while

that the objective should be to maximize the values on the diagonal. Therefore, the objective can be formulated as a classification problem with a batch of T samples whose labels are 0, 1, 2, ..., T.

$$\begin{pmatrix} \cos(z_1, \hat{z}_1) & \cos(z_1, \hat{z}_2) & \dots & \cos(z_1, \hat{z}_T) \\ \cos(z_2, \hat{z}_1) & \cos(z_2, \hat{z}_2) & \dots & \cos(z_2, \hat{z}_T) \\ \vdots & \vdots & \ddots & \vdots \\ \cos(z_T, \hat{z}_1) & \cos(z_T, \hat{z}_2) & \dots & \cos(z_T, \hat{z}_T) \end{pmatrix}$$

With this objective, the model is encoder q_φ and dynamics model p_φ are trained end-to-end, except for a reward predictor that predict the immediate reward r_t from the anchor z_t at each timestep which is trained independently without its gradient flowing back to the p_φ and q_φ.

Once the world model is setup, we follow the same Dreamer training procedure as specified in Algorithm 1 . The world model is trained on batches of real trajectories from the buffer $\mathcal{D}$, a PPO agent is trained on imagined rollouts from the world model, then the training is paused while the PPO agent interacts with the environment using the latest policy to collect a specified number of new episodes.

4.3 Improvements

In this subsection, we will discuss the attempts that we made to improve the model performance.

Learning Rate Balancing throughout our experiments, it has been observed that the p_φ is much harder to train than q_φ, which intuitively makes sense since the environment dynamics is complex and much harder to model. Regularizing the representations of the dynamics imagination and the encoder's output will equal strength destabilizes training. We take inspiration from DreamerV2 and introduce a hyperparameter $\alpha < 0.5$ that regularizes the encoder towards the dynamics with strength α and the reverse with strength $1 - \alpha$. This way, the encoder is not being forced to match a poorly modeled dynamics during early stage of the training.

Activation Functions Numerous different activation functions are tested. Eventually, we found that the SiLU with layer normalization and bias disabled performs the best. Common activations such as ReLU frequently experience dying neuron problems, especially in the reward predictor, as the reward values are often zero during early stage of the training.

Reward Normalization the reward values are normalized to have ranges $[0, 1]$. We notice that this improves the stablity of the agent learning, especially in environments with sparse yet high rewards. From implementation perspective, subtracting the reward by a bias does not change the gradient; therefore, the reward value is directly divided by a constant.

Action Embedding Initially, the raw actions were fed into the dynamics model. This oversight was later noticed and an additional action encoder is introduced to embed the raw actions, which gets concatenated with the latent z_t before passed into the dynamics model.

5 Results and Discussions

5.1 Traditional machine learning approaches (Linear Regression and SVM)

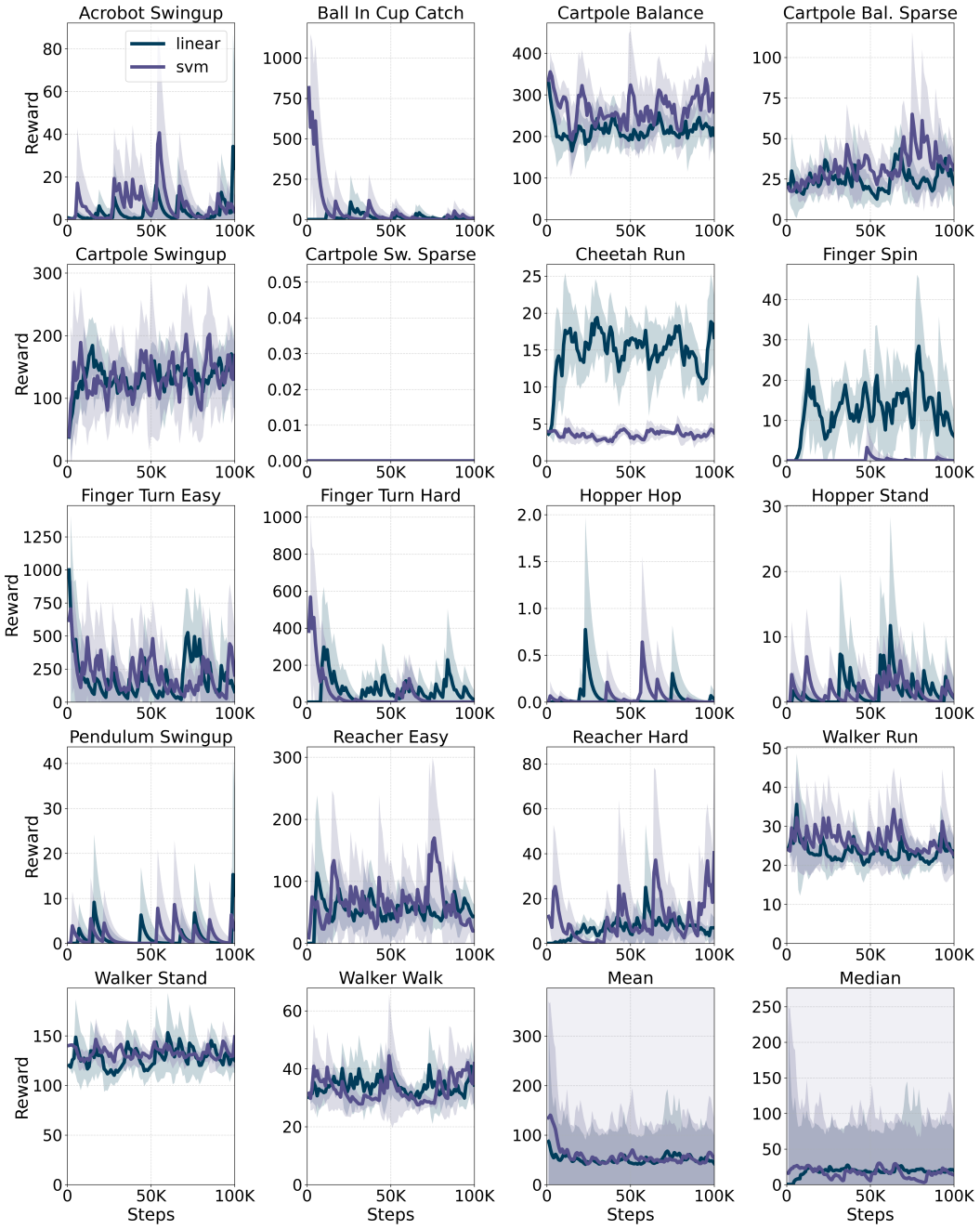

Figure 1: Results of running 18 classic DeepMind Control proprioceptive tasks using Linear Regression and SVM. Neither SVM nor linear regression based Q-Learning exhibits noticeable increase across the tasks. These algorithms clearly failed to converge and are no better than random actions. This aligns with our expectation that these simple machine learning models does not have the expressiveness necessary for fitting the complex Q-function, let alone continuos controls.

5.2 DreamerV3 and our replication of DreamerV3

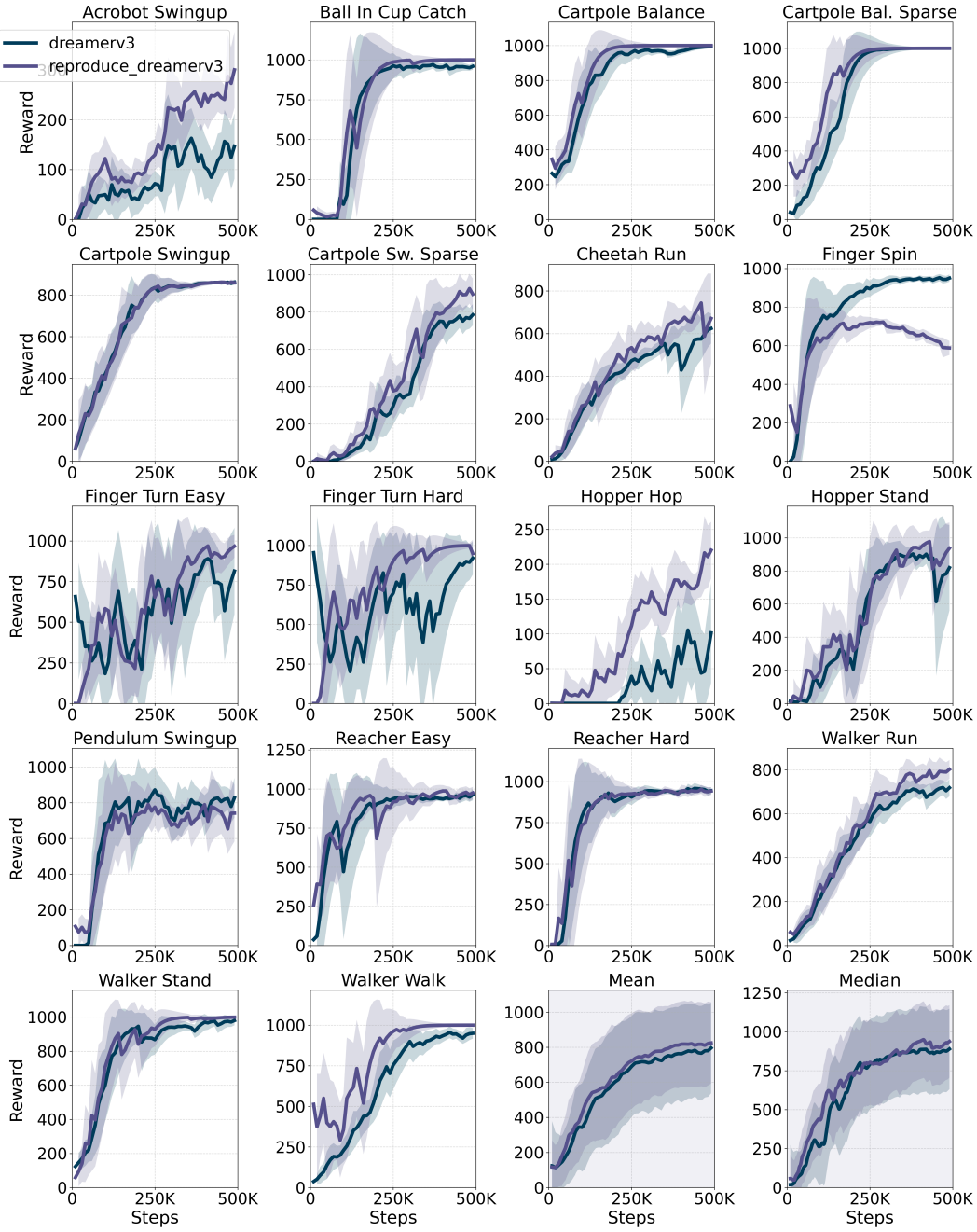

Figure 2: Official and reproduced results of DreamerV3. The official results are sourced from https://github.com/danijar/dreamerv3/tree/main/scores . The reproduced results use a third party PyTorch-based implementation of DreamerV3 instead of the official JAX results due to compatibility concerns. Therefore, the scores in certain tasks differ from the official results; this is expected as noted by the author of the torch implementation (See: https://github.com/NM512/dreamerv3-torch README.md for more details)

5.3 Comparison of DreamerV3, PPO, and our proposed WM-PPO

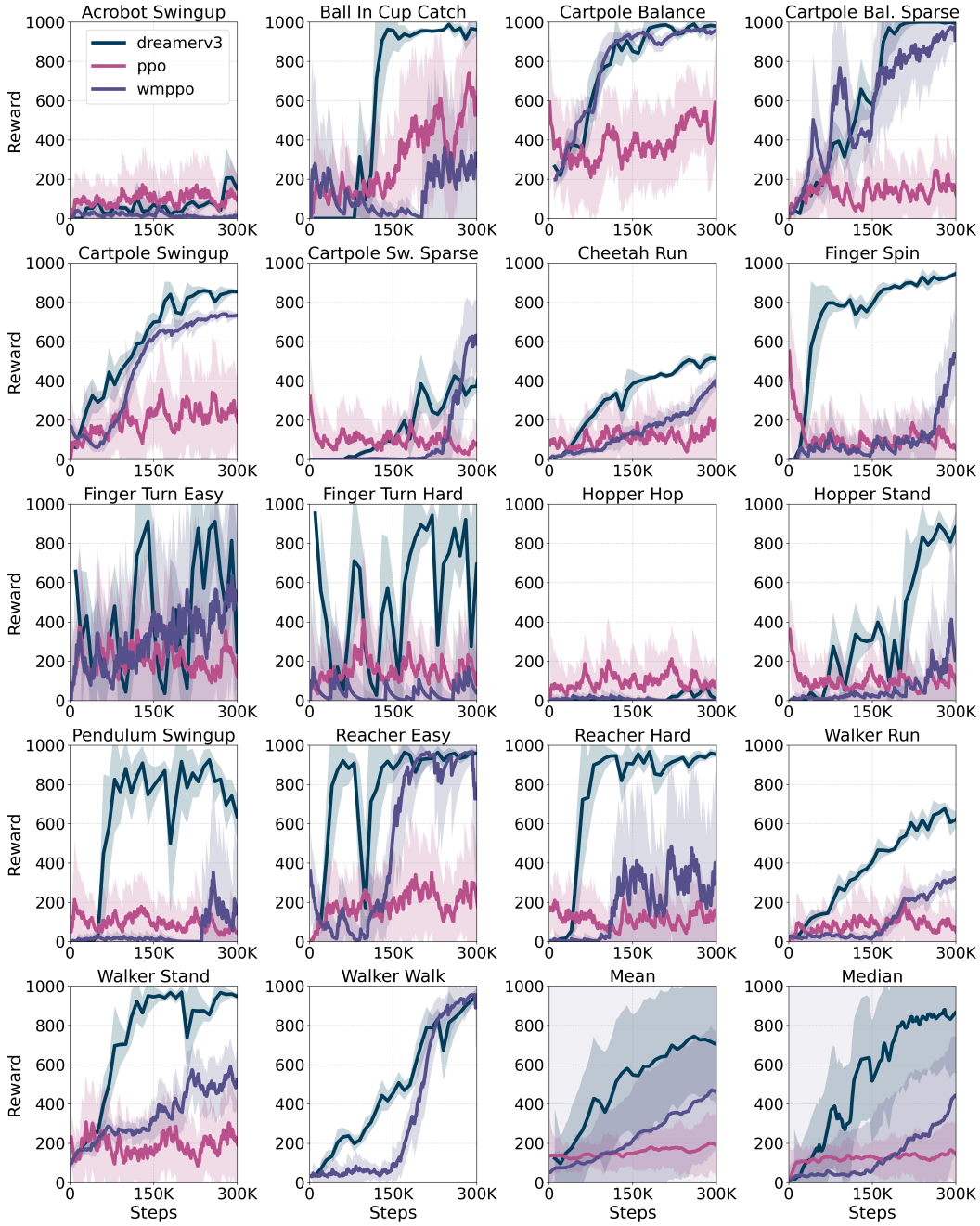

Figure 3: Results directly comparing DreamerV3, PPO and our proposed WM-PPO approach. As shown by the results. WM-PPO excels in many tasks where the baseline PPO stagnates. While our WM-PPO's peak performance does not match that of DreamerV3's, we hypothesize that cause to be the slower learning speed of the world model due to lack of reconstruction gradients. This evident by a spiky imagined reward values during early stage of training, that gradually converges after 100K environment steps, suggesting that the PPO agent might be exploiting an immature world model during early stages of training.

Task	Linear	SVM	PPO	DreamerV3	WM-PPO
Environment steps	100K	100K	500K	500K	500K
Acrobot Swingup	5.8	5.1	99.6	**132.8**	7.5
Ball In Cup Catch	7.9	10.6	620.0	**956.9**	261.0
Cartpole Balance	216.0	274.6	464.4	**978.1**	952.3
Cartpole Balance Sparse	26.1	40.9	341.0	**1000.0**	887.7
Cartpole Swingup	136.8	146.4	278.8	**857.8**	721.6
Cartpole Swingup Sparse	0.0	0.0	80.6	**760.1**	290.1
Cheetah Run	15.2	3.7	138.2	**557.4**	281.9
Finger Spin	13.4	0.1	105.3	**945.6**	238.1
Finger Turn Easy	230.6	161.3	296.7	**758.1**	447.5
Finger Turn Hard	59.8	0.1	195.9	**738.6**	45.2
Hopper Hop	0.0	0.0	91.5	**72.4**	0.5
Hopper Stand	1.6	1.5	100.0	**837.1**	151.4
Pendulum Swingup	2.4	1.5	106.0	**800.2**	111.6
Reacher Easy	52.1	65.4	355.6	**945.9**	900.9
Reacher Hard	7.6	15.1	165.2	**944.1**	323.5
Walker Run	23.2	24.6	104.0	**706.4**	266.6
Walker Stand	130.4	133.1	172.3	**962.9**	510.4
Walker Walk	33.9	35.1	106.1	**936.6**	867.9
Task Mean	53.49	51.06	212.29	**771.72**	403.7
Task Median	19.2	12.85	151.7	**847.45**	286.0

Table 1: The final EMA scores of the methods.

6 Hyperparameters

Q-Learning

Hyperparam.	*Notation*	*Value*
Discount factor	γ	0.99
Learning rate	α	0.1
Exploration rate	ε	0.3
Episodes	M	100
Episode length	T	1000
Buffer capacity	—	10000
Batch size	B	64
Action bins	N_{bins}	5
Action samples	N_{samples}	1000

PPO

Learning rate	α	3e-4
Discount factor	γ	0.99
GAE parameter	λ	0.95
Clipping coefficient	ε	0.2
Entropy coefficient	c_1	0.0
Critic coeff.	c_2	0.5
Update epochs	K	10
Rollout length	T	2048
Max gradient norm	—	0.5

WM-PPO

Hyperparam.	*Notation*	*Value*
WM Batch size	B	50
WM learning rate	α_{WM}	0.0002
LR Balancing factor	α	0.2
Temperature	τ	0.3
Reward coeff.	c_r	1.0
Latent dimension	z_{dim}	128
FC dimension	fc_{dim}	256
FC Layer count	$\text{fc}_{\text{layers}}$	8
PPO Learning rate	α_{PPO}	0.00001
Discount factor	γ	0.99
GAE parameter	λ	0.95
Clipping coeff.	ε	0.2
Critic coeff.	c_v	0.5
Entropy coeff.	c_e	0.001
PPO Update epochs	K	4
Imag. horizon	H	15
PPO batch size	—	256
WM Window size	L	50
WM updates ratio	N_{wm}	10
LR annealing	—	True

7 Dataset Description

We hereby confirm that we have met the requirements in the project description. For our project, there is not fixed dataset, the training data is collected on-the-fly through agent interaction. For the *DeepMind Control Suite* which we used, each environment has an observation space of roughly 5-10, with an action space of up to 20 ($\boldsymbol{d \approx 30}$). Since we trained for 100K steps ($\boldsymbol{m = 100,000}$), this sums up to $(20 + 10) \times 100,000 = 3,000,000$, successfully meeting the requirement of $m * d \geq 1,000,000$.

8 References

[1] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016, doi: 10.1038/nature16961.

[2] OpenAI *et al.*, "Dota 2 with Large Scale Deep Reinforcement Learning." [Online]. Available: https://arxiv.org/abs/1912.06680

[3] O. Vinyals *et al.*, "StarCraft II: A New Challenge for Reinforcement Learning." [Online]. Available: https://arxiv.org/abs/1708.04782

[4] D. Hafner, A. Pashevich, J. Ba, N. Heess, T. Lillicrap, and M. Norouzi, "Mastering diverse domains through world models," *arXiv preprint arXiv:2301.04104*, 2023.

[5] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3–4, pp. 279–292, 1992.

[6] D. Hafner, T. Lillicrap, M. Norouzi, and J. Ba, "Dream to control: Learning behaviors by latent imagination," *arXiv preprint arXiv:1912.01603*, 2019.

[7] V. Mnih and others, "Playing Atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[8] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[9] CleanRL, "PPO implementation." [Online]. Available: https://github.com/vwxyzjn/cleanrl

[10] S. Huang *et al.*, "CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms," *Journal of Machine Learning Research*, vol. 23, no. 274, pp. 1–14, 2022.