PlaNet, ST and Gumbel-softmax, DreamerV2+3, MERLIN

Milan Straka

May 14, 2025

EUROPEAN UNION
European Structural and Investment Fund
Operational Programme Research,
Development and Education

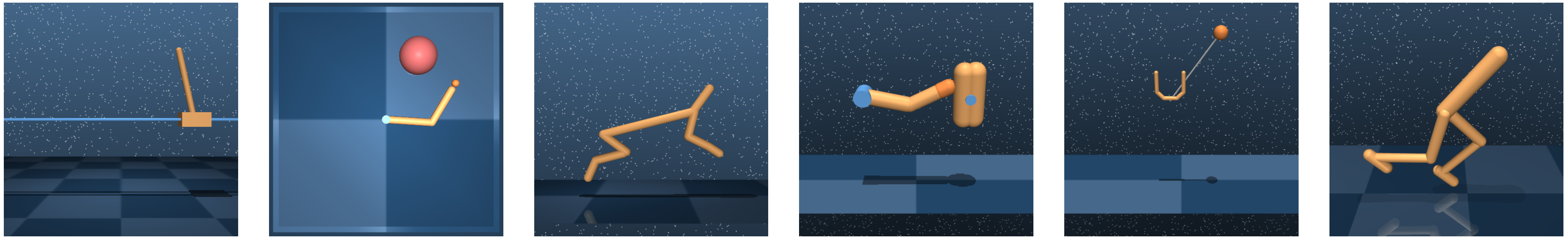
Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

PlaNet

PlaNet

In Nov 2018, an interesting paper from D. Hafner et al. proposed a **Deep Planning Network (PlaNet)**, which is a model-based agent that learns the MDP dynamics from pixels, and then chooses actions using a CEM planner utilizing the learned compact latent space.

The PlaNet is evaluated on selected tasks from the DeepMind control suite



(a) Cartpole (b) Reacher (c) Cheetah (d) Finger (e) Cup (f) Walker

Figure 1 of "Learning Latent Dynamics for Planning from Pixels", https://arxiv.org/abs/1811.04551

PlaNet

In PlaNet, partially observable MDPs following the stochastic dynamics are considered:

$$\begin{aligned}
\text{transition function:} \quad & s_t \sim p(s_t | s_{t-1}, a_{t-1}), \\
\text{observation function:} \quad & o_t \sim p(o_t | s_t), \\
\text{reward function:} \quad & r_t \sim p(r_t | s_t), \\
\text{policy:} \quad & a_t \sim p(a_t | o_{\le t}, a_{<t}).
\end{aligned}$$

The main goal is to train the first three – the transition function, the observation function, and the reward function.

PlaNet – Data Collection

Algorithm 1: Deep Planning Network (PlaNet)

Input :

R	Action repeat	$p(s_t \mid s_{t-1}, a_{t-1})$	Transition model
S	Seed episodes	$p(o_t \mid s_t)$	Observation model
C	Collect interval	$p(r_t \mid s_t)$	Reward model
B	Batch size	$q(s_t \mid o_{\leq t}, a_{<t})$	Encoder
L	Chunk length	$p(\epsilon)$	Exploration noise
α	Learning rate		

```
Initialize dataset $\mathcal{D}$ with $S$ random seed episodes.
Initialize model parameters $\theta$ randomly.
while not converged do
    // Model fitting
    for update step $s = 1..C$ do
        Draw sequence chunks $\{(o_t, a_t, r_t)_{t=k}^{L+k}\}_{i=1}^{B} \sim \mathcal{D}$
          uniformly at random from the dataset.
        Compute loss $\mathcal{L}(\theta)$ from Equation 3.
        Update model parameters $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta)$.
    // Data collection
    $o_1 \leftarrow$ env.reset()
    for time step $t = 1..\lceil \frac{T}{R} \rceil$ do
        Infer belief over current state $q(s_t \mid o_{\leq t}, a_{<t})$ from
          the history.
        $a_t \leftarrow$ planner($q(s_t \mid o_{\leq t}, a_{<t}), p$), see
          Algorithm 2 in the appendix for details.
        Add exploration noise $\epsilon \sim p(\epsilon)$ to the action.
        for action repeat $k = 1..R$ do
            $r_t^k, o_{t+1}^k \leftarrow$ env.step($a_t$)
        $r_t, o_{t+1} \leftarrow \sum_{k=1}^{R} r_t^k, o_{t+1}^R$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(o_t, a_t, r_t)_{t=1}^{T}\}$
```

Algorithm 1 of "Learning Latent Dynamics for Planning from Pixels", https://arxiv.org/abs/1811.04551

Because an untrained agent will most likely not cover all needed environment states, we need to iteratively collect new experience and train the model. The authors propose $S=5$, $C=100$, $B=50$, $L=50$, R between 2 and 8.

For planning, CEM algorithm (capable of solving all tasks with a true model) is used; $H=12$, $I=10$, $J=1000$, $K=100$.

Algorithm 2: Latent planning with CEM

Input :

H	Planning horizon distance	$q(s_t \mid o_{\leq t}, a_{<t})$	Current state belief
I	Optimization iterations	$p(s_t \mid s_{t-1}, a_{t-1})$	Transition model
J	Candidates per iteration	$p(r_t \mid s_t)$	Reward model
K	Number of top candidates to fit		

```
Initialize factorized belief over action sequences $q(a_{t:t+H}) \leftarrow \operatorname{Normal}(0, \mathbb{I})$.
for optimization iteration $i = 1..I$ do
    // Evaluate $J$ action sequences from the current belief.
    for candidate action sequence $j = 1..J$ do
        $a_{t:t+H}^{(j)} \sim q(a_{t:t+H})$
        $s_{t:t+H+1}^{(j)} \sim q(s_t \mid o_{1:t}, a_{1:t-1}) \prod_{\tau=t+1}^{t+H+1} p(s_\tau \mid s_{\tau-1}, a_{\tau-1}^{(j)})$
        $R^{(j)} = \sum_{\tau=t+1}^{t+H+1} \operatorname{E}[p(r_\tau \mid s_\tau^{(j)})]$
    // Re-fit belief to the $K$ best action sequences.
    $\mathcal{K} \leftarrow \operatorname{argsort}(\{R^{(j)}\}_{j=1}^{J})_{1:K}$
    $\mu_{t:t+H} = \frac{1}{K} \sum_{k \in \mathcal{K}} a_{t:t+H}^{(k)}, \quad \sigma_{t:t+H} = \frac{1}{K-1} \sum_{k \in \mathcal{K}} |a_{t:t+H}^{(k)} - \mu_{t:t+H}|.$
    $q(a_{t:t+H}) \leftarrow \operatorname{Normal}(\mu_{t:t+H}, \sigma_{t:t+H}^2 \mathbb{I})$
return first action mean $\mu_t$.
```

Algorithm 2 of "Learning Latent Dynamics for Planning from Pixels", https://arxiv.org/abs/1811.04551

PlaNet – Latent Dynamics

First let us consider a typical latent-space model, consisting of

$$\begin{aligned} \text{transition function:} &\quad s_t \sim p(s_t | s_{t-1}, a_{t-1}), \\ \text{observation function:} &\quad o_t \sim p(o_t | s_t), \\ \text{reward function:} &\quad r_t \sim p(r_t | s_t). \end{aligned}$$

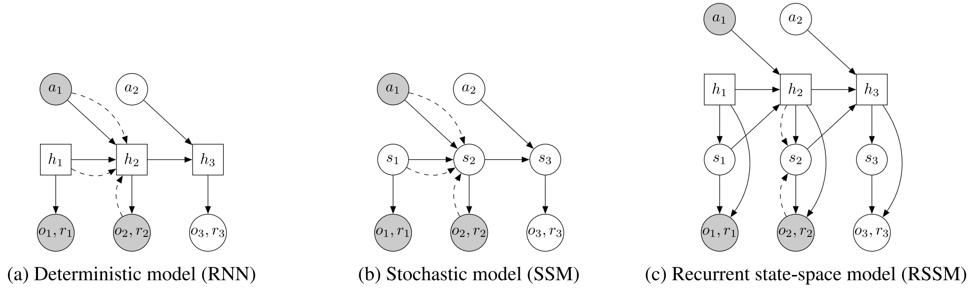

a_1
a_2
h_1
h_2
h_3
s_1
s_2
s_3
o_1, r_1
o_2, r_2
o_3, r_3
(a) Deterministic model (RNN)
(b) Stochastic model (SSM)
(c) Recurrent state-space model (RSSM)

Figure 2 of "Learning Latent Dynamics for Planning from Pixels", https://arxiv.org/abs/1811.04551

The transition model is Gaussian with mean and variance predicted by a network, the observation model is Gaussian with identity covariance and mean predicted by a deconvolutional network, and the reward model is a scalar Gaussian with unit variance and mean predicted by a neural network.

To train such a model, we turn to variational inference, and use an encoder $q(s_{1:T} | o_{1:T}, a_{1:T-1}) = \prod_{t=1}^T q(s_t | s_{t-1}, a_{t-1}, o_t)$, which is a Gaussian with mean and variance predicted by a convolutional neural network.

PlaNet – Training Objective

Using the encoder, we obtain the following variational lower bound on the log-likelihood of the observations (for rewards the bound is analogous):

$$\begin{aligned}
&\log p(o_{1:T} | a_{1:T}) \\
&\quad = \log \int \prod_t p(s_t | s_{t-1}, a_{t-1}) p(o_t | s_t)\, \mathrm{d}s_{1:T} \\
&\quad \geq \sum_{t=1}^{T} \Bigg(\underbrace{\mathbb{E}_{q(s_t | o_{\leq t}, a_{<t})} \log p(o_t | s_t)}_{\text{reconstruction}} - \underbrace{\mathbb{E}_{q(s_{t-1} | o_{\leq t-1}, a_{<t-1})} D_{\mathrm{KL}}\Big(q(s_t | o_{\leq t}, a_{<t}) \| p(s_t | s_{t-1}, a_{t-1})\Big)}_{\text{complexity}} \Bigg).
\end{aligned}$$

We evaluate the expectations using a single sample, and use the reparametrization trick to allow backpropagation through the sampling.

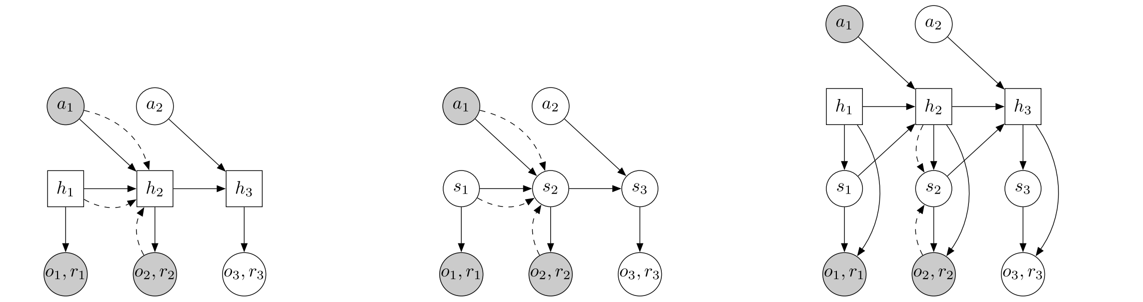
PlaNet – Training Objective Derivation

To derive the training objective, we employ importance sampling and the Jensen's inequality:

$$\begin{aligned}
&\log p(o_{1:T}|a_{1:T}) \\
&\quad= \log \mathbb{E}_{p(s_{1:T}|a_{1:T})} \prod_{t=1}^T p(o_t|s_t) \\
&\quad= \log \mathbb{E}_{q(s_{1:T}|o_{1:T},a_{1:T})} \prod_{t=1}^T p(o_t|s_t) p(s_t|s_{t-1},a_{t-1}) / q(s_t|o_{\le t},a_{<t}) \\
&\quad\ge \mathbb{E}_{q(s_{1:T}|o_{1:T},a_{1:T})} \sum_{t=1}^T \log p(o_t|s_t) + \log p(s_t|s_{t-1},a_{t-1}) - \log q(s_t|o_{\le t},a_{<t}) \\
&= \sum_{t=1}^T \Big(\underbrace{\mathbb{E}_{q(s_t|o_{\le t},a_{<t})} \log p(o_t|s_t)}_{\text{reconstruction}} - \underbrace{\mathbb{E}_{q(s_{t-1}|o_{\le t-1},a_{<t-1})} D_\textrm{KL}\big(q(s_t|o_{\le t},a_{<t}) \| p(s_t|s_{t-1},a_{t-1})\big)}_{\text{complexity}} \Big).
\end{aligned}$$

PlaNet – Recurrent State-Space Model

The purely stochastic transitions struggle to store information for multiple timesteps. Therefore, the authors propose to include a deterministic path to the model (providing access to all previous states), obtaining the **recurrent state-space model (RSSM)**:



(a) Deterministic model (RNN) (b) Stochastic model (SSM) (c) Recurrent state-space model (RSSM)

Figure 2 of "Learning Latent Dynamics for Planning from Pixels", https://arxiv.org/abs/1811.04551

$$\begin{aligned} \text{deterministic state model:} \quad & h_t = f(h_{t-1}, s_{t-1}, a_{t-1}), \\ \text{stochastic state function:} \quad & s_t \sim p(s_t | h_t), \\ \text{observation function:} \quad & o_t \sim p(o_t | h_t, s_t), \\ \text{reward function:} \quad & r_t \sim p(r_t | h_t, s_t), \\ \text{encoder:} \quad & q_t \sim q(s_t | h_t, o_t). \end{aligned}$$

PlaNet – Results

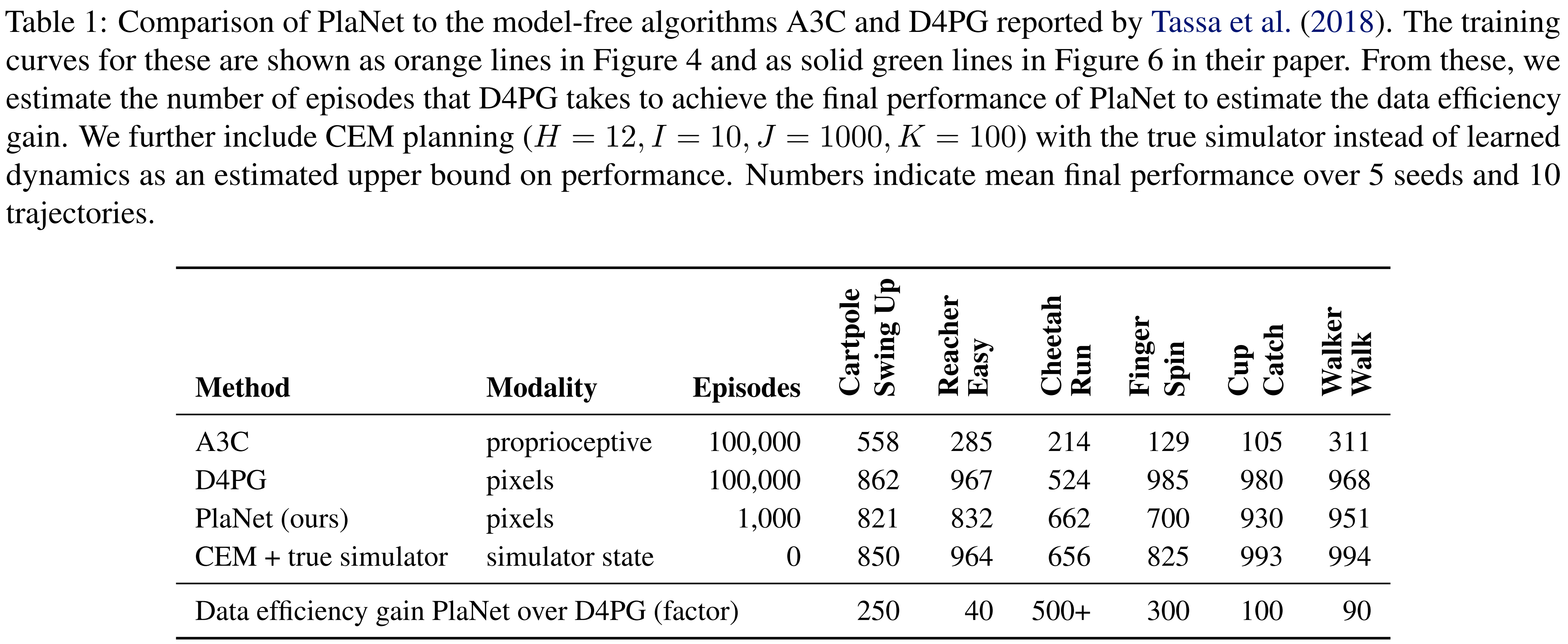


Table 1: Comparison of PlaNet to the model-free algorithms A3C and D4PG reported by Tassa et al. (2018). The training curves for these are shown as orange lines in Figure 4 and as solid green lines in Figure 6 in their paper. From these, we estimate the number of episodes that D4PG takes to achieve the final performance of PlaNet to estimate the data efficiency gain. We further include CEM planning ($H = 12, I = 10, J = 1000, K = 100$) with the true simulator instead of learned dynamics as an estimated upper bound on performance. Numbers indicate mean final performance over 5 seeds and 10 trajectories.

Method	**Modality**	**Episodes**	**Cartpole Swing Up**	**Reacher Easy**	**Cheetah Run**	**Finger Spin**	**Cup Catch**	**Walker Walk**
A3C	proprioceptive	100,000	558	285	214	129	105	311
D4PG	pixels	100,000	862	967	524	985	980	968
PlaNet (ours)	pixels	1,000	821	832	662	700	930	951
CEM + true simulator	simulator state	0	850	964	656	825	993	994
Data efficiency gain PlaNet over D4PG (factor)			250	40	500+	300	100	90

Table 1 of "Learning Latent Dynamics for Planning from Pixels", https://arxiv.org/abs/1811.04551

PlaNet – Ablations

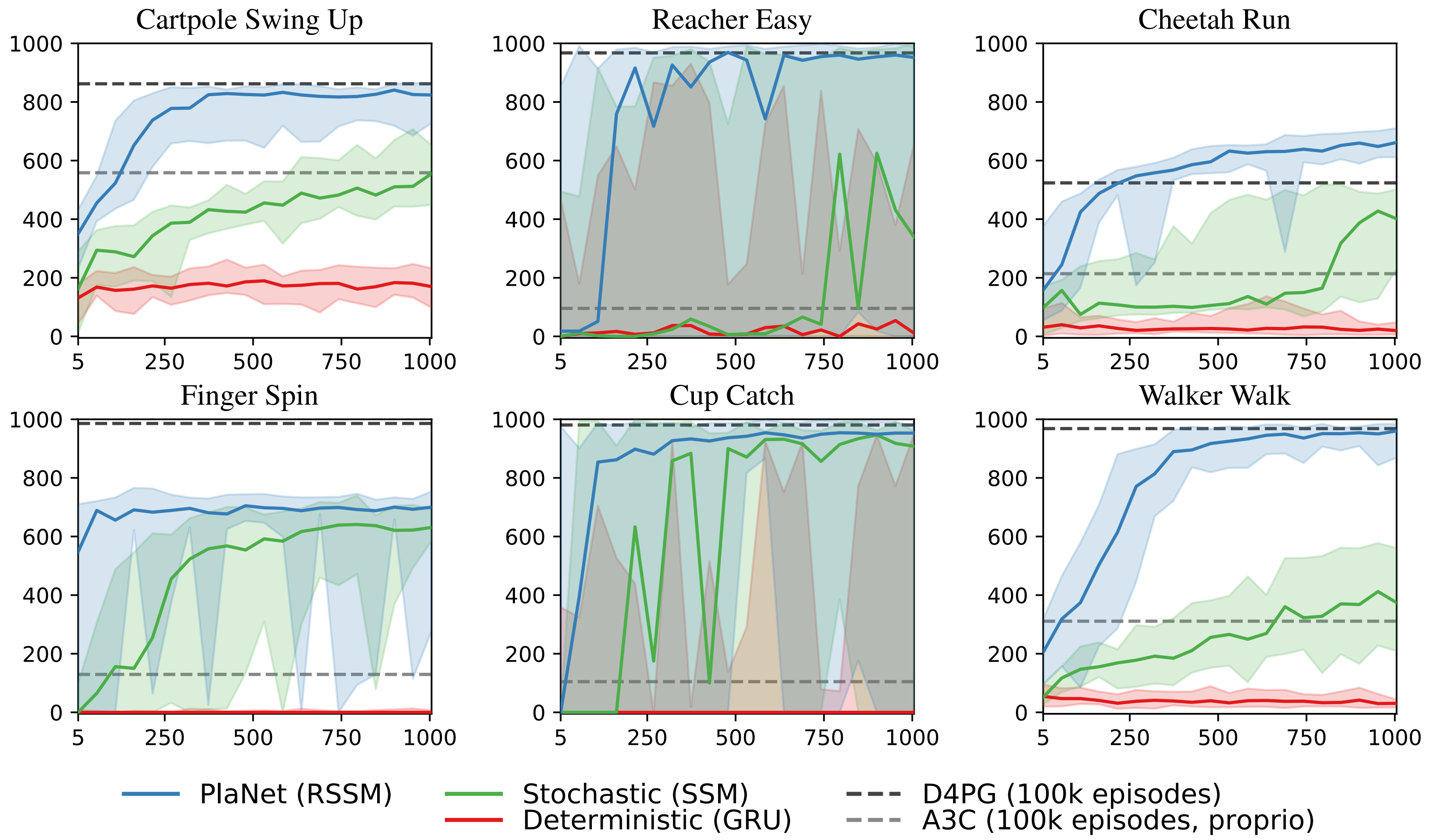


Figure 4 of "Learning Latent Dynamics for Planning from Pixels", https://arxiv.org/abs/1811.04551

PlaNet – Ablations

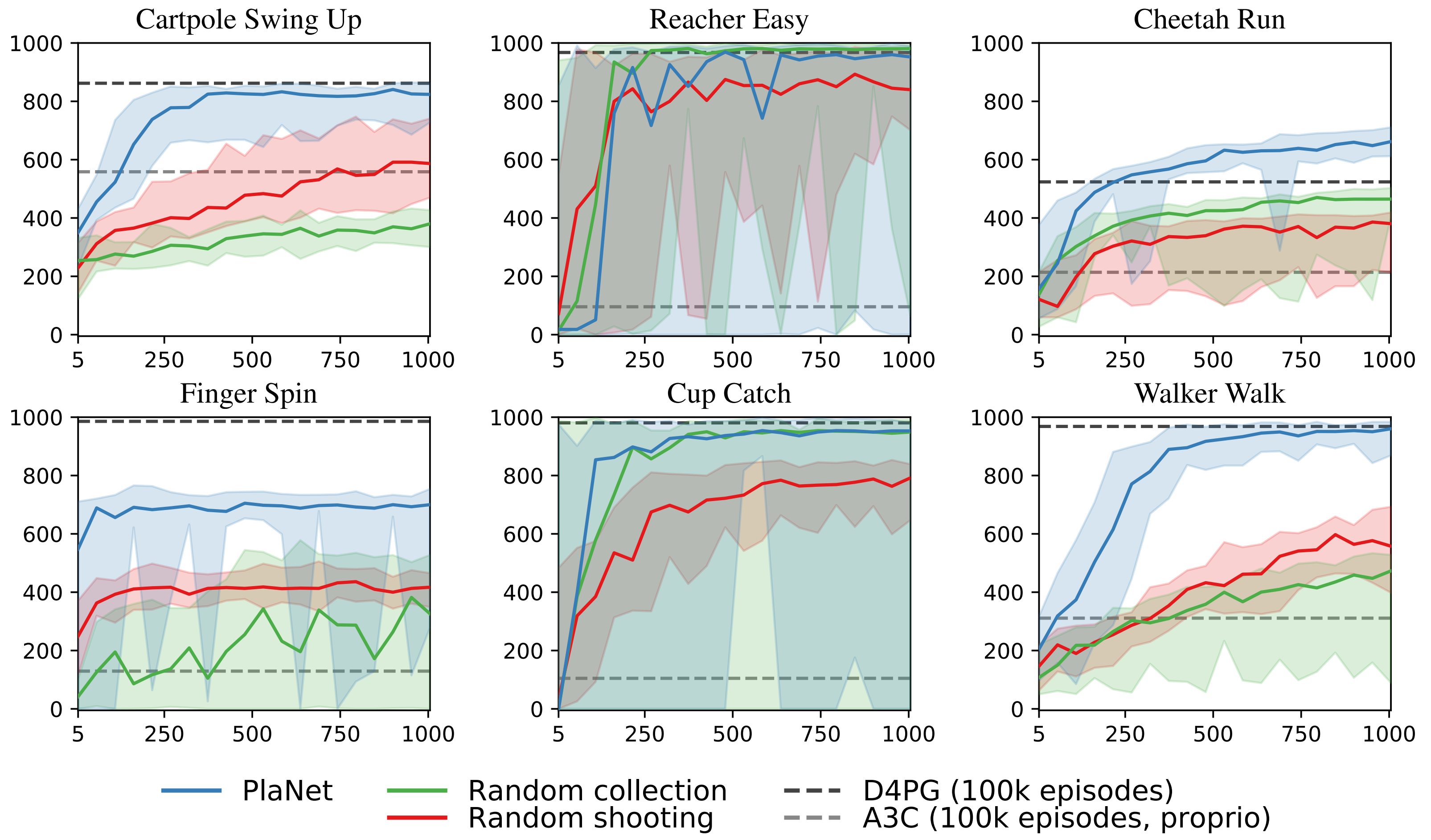


Figure 5 of "Learning Latent Dynamics for Planning from Pixels", https://arxiv.org/abs/1811.04551

Random collection: random actions; random shooting: best action out of 1000 random seqs.

Straight-Through (ST) Estimator

Discrete Latent Variables

Consider that we would like to have discrete neurons on the hidden layer of a neural network.

Note that on the output layer, we relaxed discrete prediction (i.e., an $\arg\max$) with a continuous relaxation – $\operatorname{softmax}$. This way, we can compute the derivatives and also predict the most probable class. (It is possible to derive $\operatorname{softmax}$ as an entropy-regularized $\arg\max$.)

However, on a hidden layer, we also need to *sample* from the predicted categorical distribution, and then backpropagate the gradients.

Stochastic Gradient Estimators

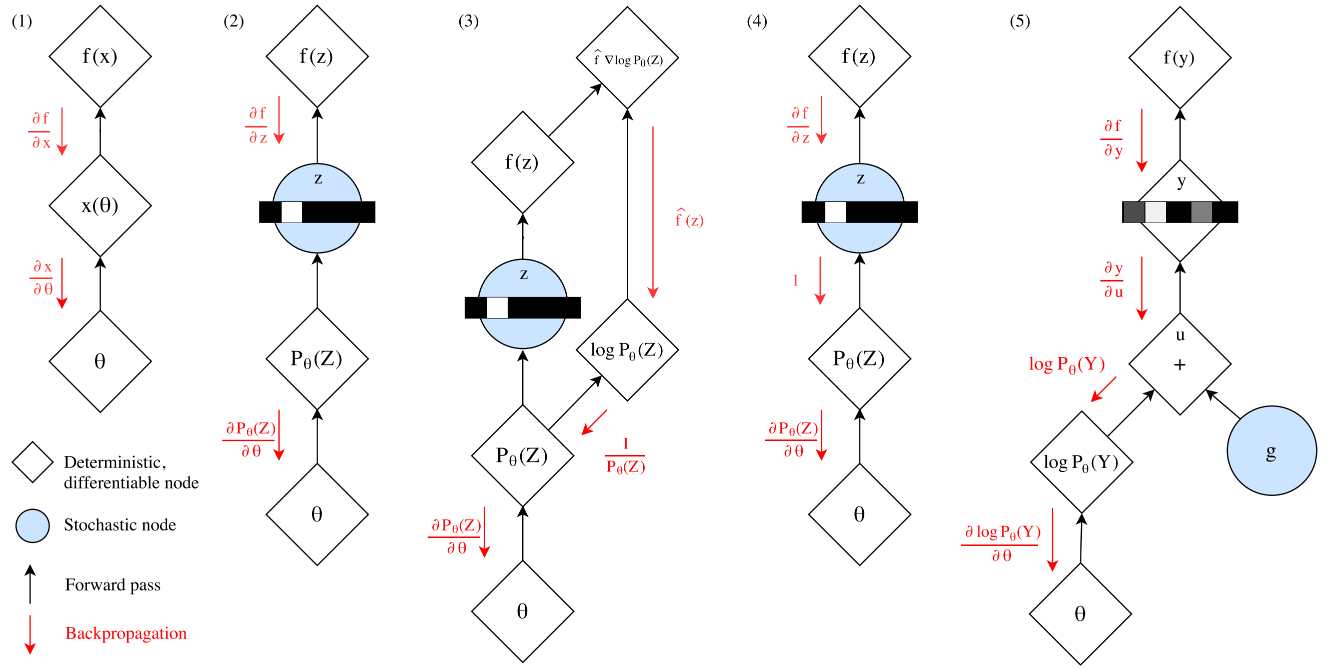


Figure 2: Gradient estimation in stochastic computation graphs. (1) $\nabla_\theta f(x)$ can be computed via backpropagation if $x(\theta)$ is deterministic and differentiable. (2) The presence of stochastic node z precludes backpropagation as the sampler function does not have a well-defined gradient. (3) The score function estimator and its variants (NVIL, DARN, MuProp, VIMCO) obtain an unbiased estimate of $\nabla_\theta f(x)$ by backpropagating along a surrogate loss $\hat{f} \log p_\theta(z)$, where $\hat{f} = f(x) - b$ and b is a baseline for variance reduction. (4) The Straight-Through estimator, developed primarily for Bernoulli variables, approximates $\nabla_\theta z \approx 1$. (5) Gumbel-Softmax is a path derivative estimator for a continuous distribution y that approximates z. Reparameterization allows gradients to flow from $f(y)$ to θ. y can be annealed to one-hot categorical variables over the course of training.

Figure 2 of "Categorical Reparameterization with Gumbel-Softmax", https://arxiv.org/abs/1611.01144

Stochastic Gradient Estimators

Consider a model with a discrete categorical latent variable $\boldsymbol z$ sampled from $p(\boldsymbol z; \boldsymbol\theta)$, with a loss $L(\boldsymbol z; \boldsymbol\omega)$. Several gradient estimators have been proposed:

- A REINFORCE-like gradient estimation.

  Using the identity $\nabla_{\boldsymbol\theta} p(\boldsymbol z; \boldsymbol\theta) = p(\boldsymbol z; \boldsymbol\theta)\nabla_{\boldsymbol\theta} \log p(\boldsymbol z; \boldsymbol\theta)$, we obtain that

  $$\nabla_{\boldsymbol\theta} \mathbb{E}_{\boldsymbol z}\big[L(\boldsymbol z; \boldsymbol\omega)\big] = \mathbb{E}_{\boldsymbol z}\big[L(\boldsymbol z; \boldsymbol\omega)\nabla_{\boldsymbol\theta} \log p(\boldsymbol z; \boldsymbol\theta)\big].$$

  Analogously as before, we can also include the baseline for variance reduction, resulting in

  $$\nabla_{\boldsymbol\theta} \mathbb{E}_{\boldsymbol z}\big[L(\boldsymbol z; \boldsymbol\omega)\big] = \mathbb{E}_{\boldsymbol z}\big[(L(\boldsymbol z; \boldsymbol\omega) - b)\nabla_{\boldsymbol\theta} \log p(\boldsymbol z; \boldsymbol\theta)\big].$$

- A **straight-through (ST)** estimator.

  The straight-through estimator has been proposed by Y. Bengio in 2013. It is a biased estimator, which assumes that $\nabla_{\boldsymbol\theta} \boldsymbol z \approx \nabla_{\boldsymbol\theta} p(\boldsymbol z; \boldsymbol\theta)$, which implies $\nabla_{p(\boldsymbol z;\boldsymbol\theta)} \boldsymbol z \approx 1$. Even if the bias can be considerable, it seems to work quite well in practice.

Gumbel-Softmax

Gumbel-Softmax

The **Gumbel-softmax** distribution was proposed independently in two papers in Nov 2016 (under the name of **Concrete** distribution in the other paper).

It is a continuous distribution over the simplex (over categorical distributions) that can approximate *sampling* from a categorical distribution.

Let z be a categorical variable with class probabilities $\boldsymbol{p} = (p_1, p_2, \ldots, p_K)$.

Recall that the Gumbel-Max trick (based on a 1954 theorem from E. J. Gumbel) states that we can draw samples $z \sim \boldsymbol{p}$ using

$$z = \operatorname{one-hot}\Big(\arg\max_i \big(g_i + \log p_i\big)\Big),$$

where g_i are independent samples drawn from the $\operatorname{Gumbel}(0, 1)$ distribution.

To sample g from the distribution $\operatorname{Gumbel}(0, 1)$, we can sample $u \sim U(0, 1)$ and then compute $g = -\log(-\log u)$.

Gumbel-Softmax

To obtain a continuous distribution, we relax the $\arg\max$ into a $\operatorname{softmax}$ with temperature T as

$$z_i = \frac{e^{(g_i + \log p_i)/T}}{\sum_j e^{(g_j + \log p_j)/T}}.$$

As the temperature T goes to zero, the generated samples become one-hot, and therefore the Gumbel-softmax distribution converges to the categorical distribution $p(z)$.

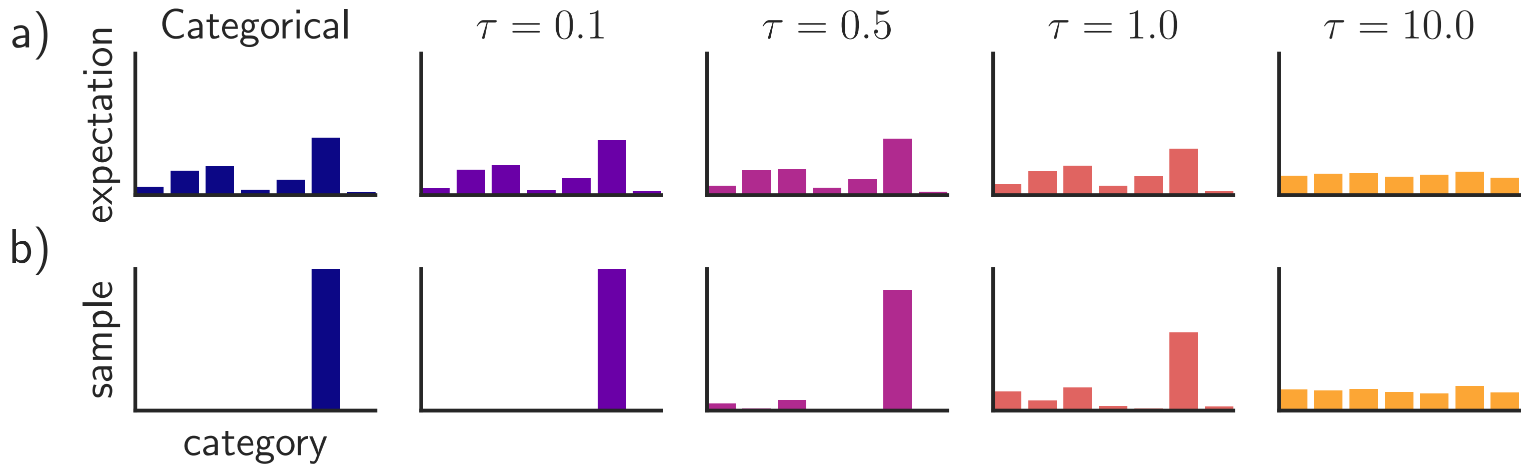


Figure 1 of "Categorical Reparameterization with Gumbel-Softmax", https://arxiv.org/abs/1611.01144

Gumbel-Softmax Estimator

The Gumbel-softmax distribution can be used to reparametrize the sampling of the discrete variable using a fully differentiable estimator.

However, the resulting sample is not discrete, it only converges to a discrete sample as the temperature T goes to zero.

If it is a problem, we can combine the Gumbel-softmax with a straight-through estimator, obtaining ST Gumbel-softmax, where we:

- discretize $\boldsymbol{y}$ as $\boldsymbol{z} = \arg\max \boldsymbol{y}$,
- assume $\nabla_{\boldsymbol\theta} \boldsymbol{z} \approx \nabla_{\boldsymbol\theta} \boldsymbol{y}$, or in other words, $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}} \approx 1$.

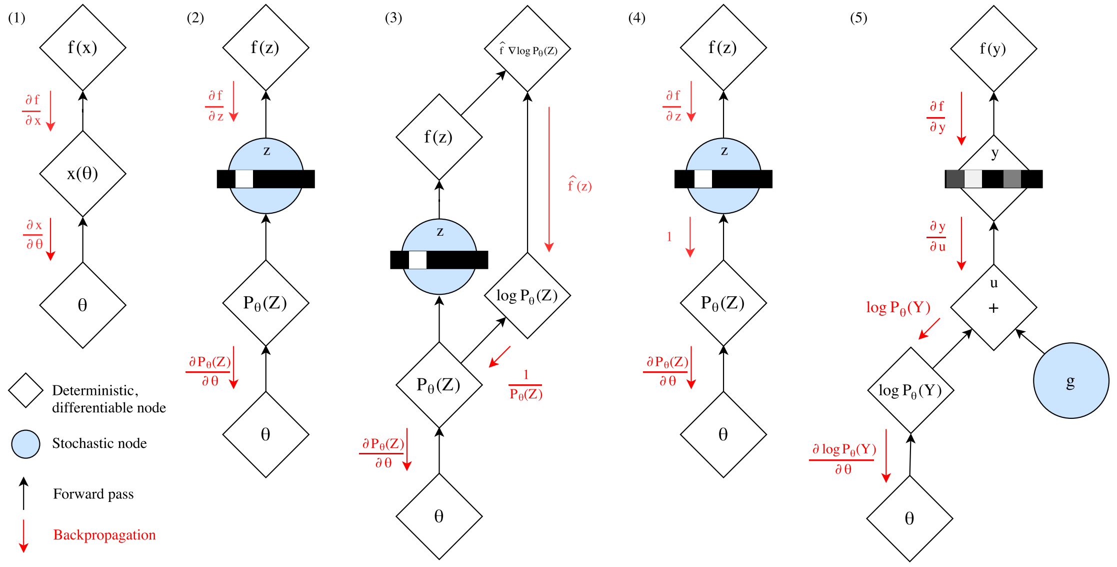


Figure 2: Gradient estimation in stochastic computation graphs. (1) $\nabla_\theta f(x)$ can be computed via backpropagation if $x(\theta)$ is deterministic and differentiable. (2) The presence of stochastic node z precludes backpropagation as the sampler function does not have a well-defined gradient. (3) The score function estimator and its variants (NVIL, DARN, MuProp, VIMCO) obtain an unbiased estimate of $\nabla_\theta f(x)$ by backpropagating along a surrogate loss $\hat{f} \log p_\theta(z)$, where $\hat{f} = f(x) - b$ and b is a baseline for variance reduction. (4) The Straight-Through estimator, developed primarily for Bernoulli variables, approximates $\nabla_\theta z \approx 1$. (5) Gumbel-Softmax is a path derivative estimator for a continuous distribution y that approximates z. Reparameterization allows gradients to flow from $f(y)$ to θ. y can be annealed to one-hot categorical variables over the course of training.

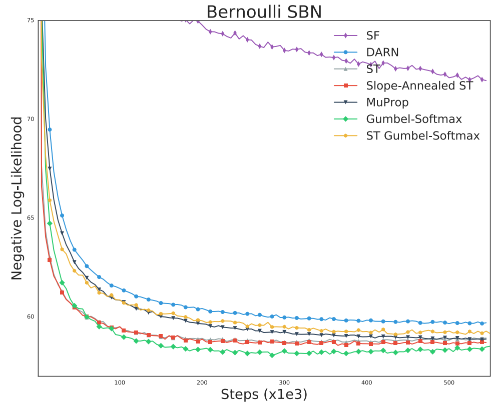
Figure 2 of "Categorical Reparameterization with Gumbel-Softmax", https://arxiv.org/abs/1611.01144

Gumbel-Softmax Estimator Results

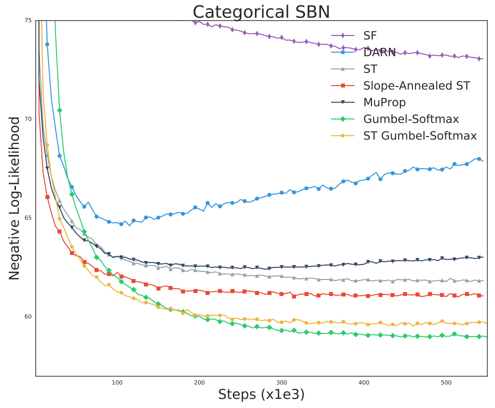
Table 1: The Gumbel-Softmax estimator outperforms other estimators on Bernoulli and Categorical latent variables. For the structured output prediction (SBN) task, numbers correspond to negative log-likelihoods (nats) of input images (lower is better). For the VAE task, numbers correspond to negative variational lower bounds (nats) on the log-likelihood (lower is better).

	SF	DARN	MuProp	ST	Annealed ST	Gumbel-S.	ST Gumbel-S.
SBN (Bern.)	72.0	59.7	58.9	58.9	58.7	**58.5**	59.3
SBN (Cat.)	73.1	67.9	63.0	61.8	61.1	**59.0**	59.7
VAE (Bern.)	112.2	110.9	109.7	116.0	111.5	**105.0**	111.5
VAE (Cat.)	110.6	128.8	107.0	110.9	107.8	**101.5**	107.8

Table 1 of "Categorical Reparameterization with Gumbel-Softmax", https://arxiv.org/abs/1611.01144



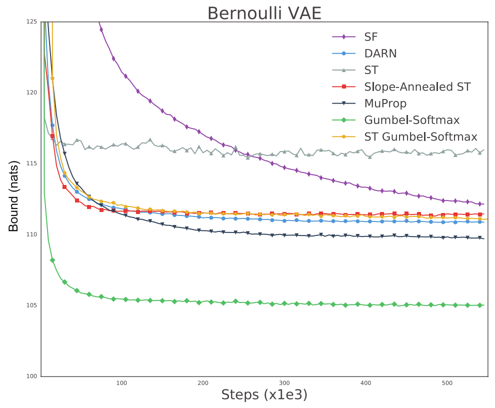
(a)



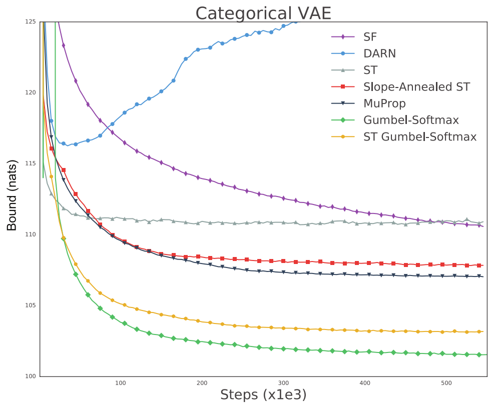
(b)

Figure 3: Test loss (negative log-likelihood) on the structured output prediction task with binarized MNIST using a stochastic binary network with (a) Bernoulli latent variables (392-200-200-392) and (b) categorical latent variables (392-(20 × 10)-(20 × 10)-392).

Figure 3 of "Categorical Reparameterization with Gumbel-Softmax", https://arxiv.org/abs/1611.01144



(a)



(b)

Figure 4: Test loss (negative variational lower bound) on binarized MNIST VAE with (a) Bernoulli latent variables ($784 - 200 - 784$) and (b) categorical latent variables ($784 - (20 \times 10) - 200$).

Figure 4 of "Categorical Reparameterization with Gumbel-Softmax", https://arxiv.org/abs/1611.01144

Applications of Discrete Latent Variables

The discrete latent variables can be used among others to:

- allow the SAC algorithm to be used on **discrete** actions, using either Gumbel-softmax relaxation (if the critic takes the actions as binary indicators, it is possible to pass not just one-hot encoding, but the result of Gumbel-softmax directly), or a straight-through estimator;
- model images using discrete latent variables
  - VQ-VAE, VQ-VAE-2 use “codebook loss” with a straight-through estimator

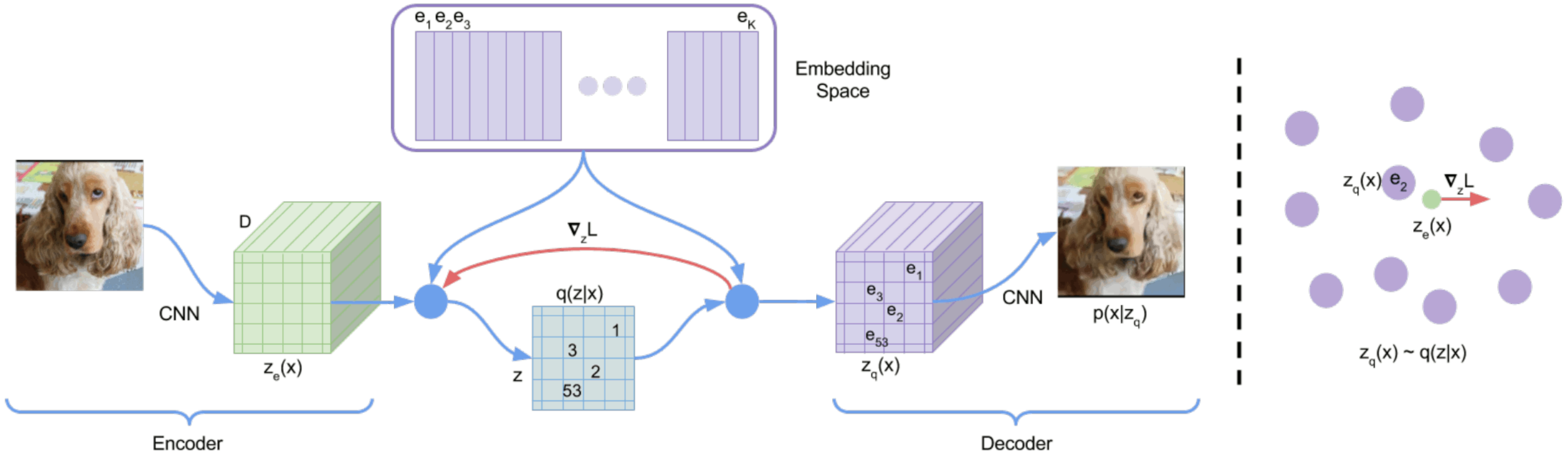


Figure 1 of "Neural Discrete Representation Learning", https://arxiv.org/abs/1711.00937

Applications of Discrete Latent Variables

- VQ-GAN combines the VQ-VAE and Transformers, where the latter is used to generate a sequence of the *discrete* latents.

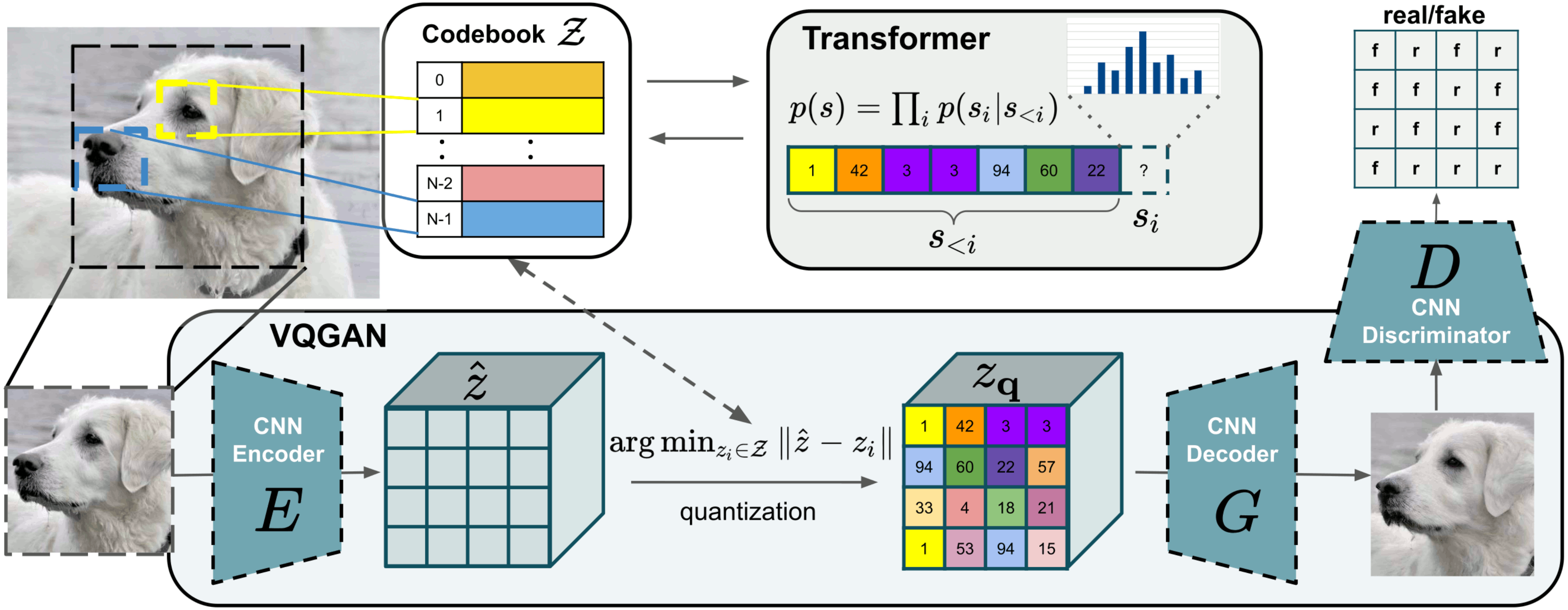


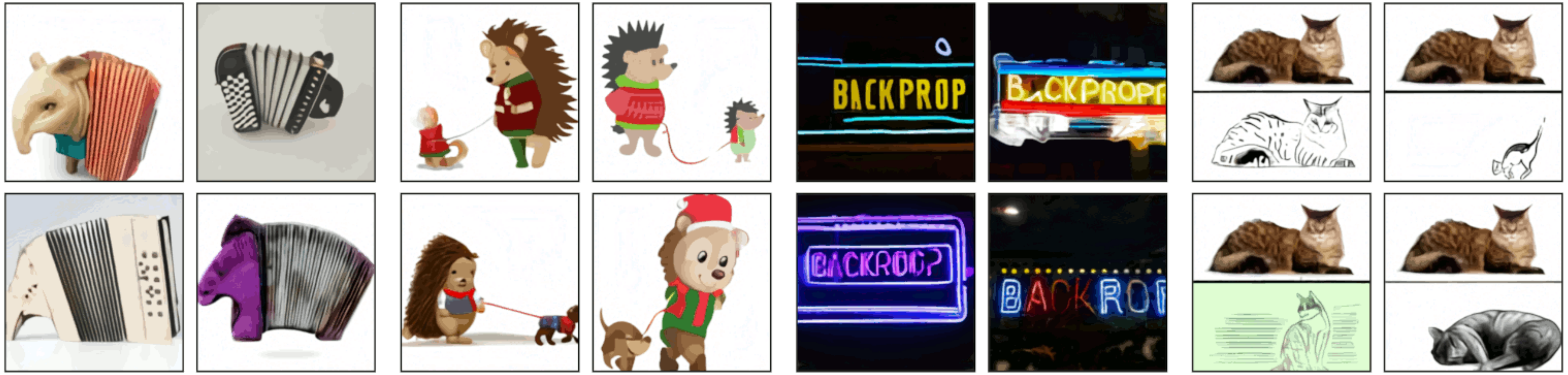
Figure 2 of "Taming Transformers for High-Resolution Image Synthesis", https://arxiv.org/abs/2012.09841

Applications of Discrete Latent Variables – VQ-GAN

Applications of Discrete Latent Variables – DALL-E

- In DALL-E, Transformer is used to model a sequence of words followed by a sequence of the discrete image latent variables.

  The Gumbel-softmax relaxation is used to train the discrete latent states, with temperature annealed with a cosine decay from 1 to 1/16 over the first 150k (out of 3M) updates.



(a) a tapir made of accordion. a tapir with the texture of an accordion.

(b) an illustration of a baby hedgehog in a christmas sweater walking a dog

(c) a neon sign that reads "backprop". a neon sign that reads "backprop". backprop neon sign

(d) the exact same cat on the top as a sketch on the bottom

Figure 2 of "Zero-Shot Text-to-Image Generation", https://arxiv.org/abs/2102.12092

DreamerV2

DreamerV2

The PlaNet model was followed by Dreamer (Dec 2019) and DreamerV2 (Oct 2020), which train an agent using reinforcement learning using the model alone. After 200M environment steps, it surpasses Rainbow on a collection of 55 Atari games (the authors do not mention why they do not use all 57 games) when training on a single GPU for 10 days per game.

During training, a policy is learned from 486B compact states "dreamed" by the model, which is 10,000 times more than the 50M observations from the real environment (with action repeat 4).

Interestingly, the latent states are represented as a vector of several **categorical** variables – 32 variables with 32 classes each are utilized in the paper.

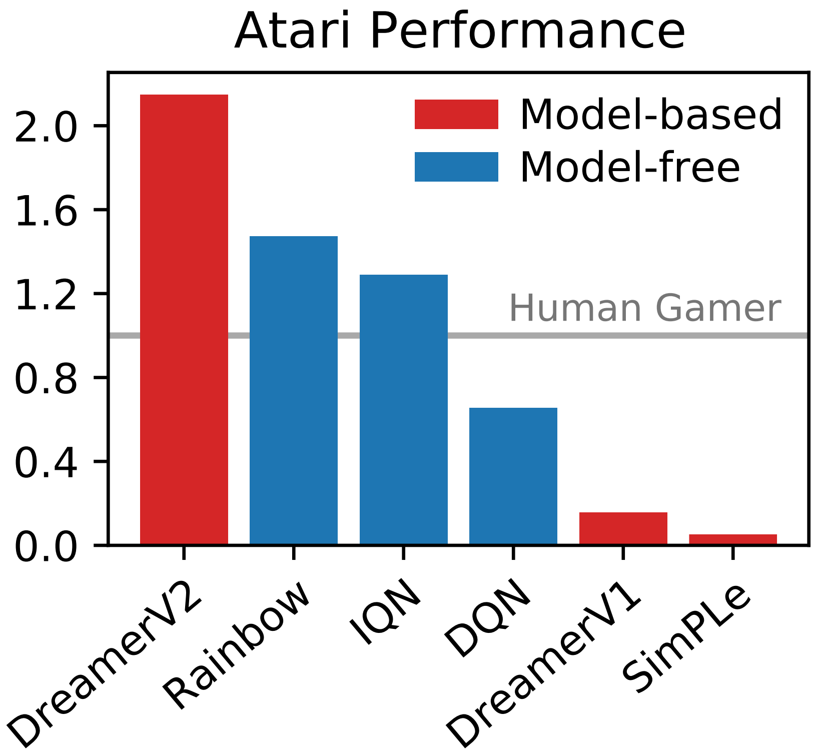


Figure 1 of "Mastering Atari with Discrete World Models", https://arxiv.org/abs/2010.02193

DreamerV2 – Model Learning

The model in DreamerV2 is learned using the RSSM, collecting agent experiences of observations, actions, rewards, and discount factors (0.995 within episode and 0 at an episode end). Training is performed on batches of 50 sequences of length at most 50 each.

$$
\begin{aligned}
\text{recurrent model:} &\quad h_t = f_\varphi(h_{t-1}, s_{t-1}, a_{t-1}),\\
\text{representation model:} &\quad s_t \sim q_\varphi(s_t | h_t, x_t),\\
\text{transition predictor:} &\quad \bar s_t \sim p_\varphi(\bar s_t | h_t),\\
\text{image predictor:} &\quad \bar x_t \sim p_\varphi(\bar x_t | h_t, s_t),\\
\text{reward predictor:} &\quad \bar r_t \sim p_\varphi(\bar r_t | h_t, s_t),\\
\text{discount predictor:} &\quad \bar\gamma_t \sim p_\varphi(\bar\gamma_t | h_t, s_t).
\end{aligned}
$$

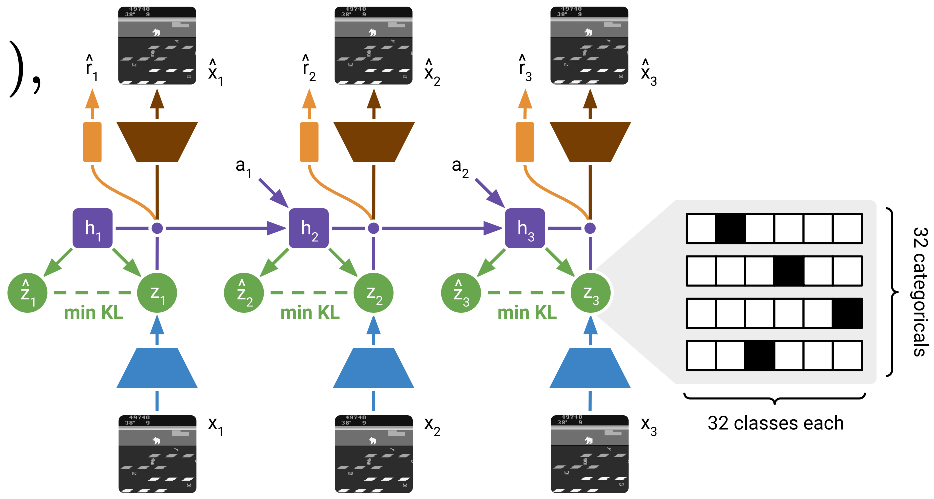


Figure 2 of "Mastering Atari with Discrete World Models", https://arxiv.org/abs/2010.02193

Algorithm 1: Straight-Through Gradients with Automatic Differentiation

```
sample = one_hot(draw(logits))            # sample has no gradient
probs  = softmax(logits)                  # want gradient of this
sample = sample + probs - stop_grad(probs) # has gradient of probs
```

Algorithm 1 of "Mastering Atari with Discrete World Models", https://arxiv.org/abs/2010.02193

DreamerV2 – Model Learning

The following loss function is used:

$$\mathcal{L}(\varphi) = \mathbb{E}_{q_\varphi(s_{1:T}|a_{1:T},x_{1:T})}\Big[\sum\nolimits_{t=1}^T \underbrace{-\log p_\varphi(x_t|h_t,s_t)}_{\text{image log loss}} \underbrace{-\log p_\varphi(r_t|h_t,s_t)}_{\text{reward log loss}} \underbrace{-\log p_\varphi(\gamma_t|h_t,s_t)}_{\text{discount log loss}}$$

$$\underbrace{+\beta D_{\mathrm{KL}}\big[q_\varphi(s_t|h_t,x_t)\|p_\varphi(s_t|h_t)\big]}_{\text{KL loss}}\Big].$$

In the KL term, we train both the prior and the encoder. However, regularizing the encoder towards the prior makes training harder (especially at the beginning), so the authors propose **KL balancing**, minimizing the KL term faster for the prior ($\alpha = 0.8$) than for the posterior.

Algorithm 2: KL Balancing with Automatic Differentiation

```
kl_loss =       alpha  * compute_kl(stop_grad(approx_posterior), prior)
        + (1 - alpha) * compute_kl(approx_posterior, stop_grad(prior))
```

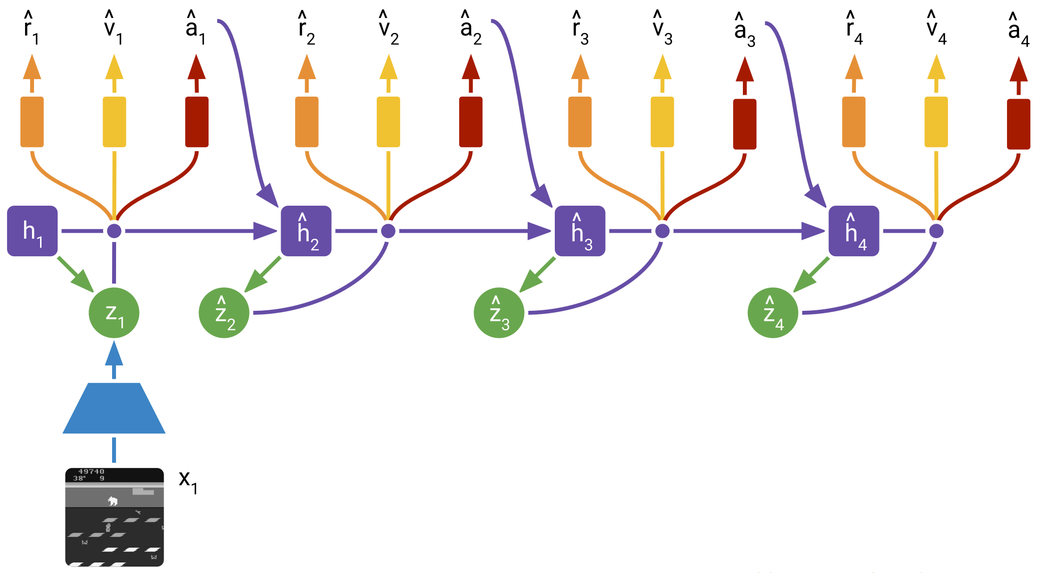
Algorithm 2 of "Mastering Atari with Discrete World Models", https://arxiv.org/abs/2010.02193

DreamerV2 – Policy Learning

The policy is trained solely from the model, starting from the encountered posterior states and then considering $H = 15$ actions simulated in the compact latent state.

We train an actor predicting $\pi_\psi(a_t|s_t)$ and a critic predicting

$$v_\xi(s_t) = \mathbb{E}_{p_\varphi,\pi_\psi}\Big[\sum\nolimits_{r\geq t}\big(\prod\nolimits_{r'=t+1}^{r}\gamma_{r'}\big)r_t\Big].$$



$\hat r_1$ $\hat v_1$ $\hat a_1$
$\hat r_2$ $\hat v_2$ $\hat a_2$
$\hat r_3$ $\hat v_3$ $\hat a_3$
$\hat r_4$ $\hat v_4$ $\hat a_4$
h_1 $\hat h_2$ $\hat h_3$ $\hat h_4$
z_1 $\hat z_2$ $\hat z_3$ $\hat z_4$
x_1

Figure 3 of "Mastering Atari with Discrete World Models", https://arxiv.org/abs/2010.02193

The critic is trained by estimating the truncated λ-return as

$$V_t^\lambda = r_t + \gamma_t \begin{cases}(1-\lambda)v_\xi(\hat z_{t+1}) + \lambda V_{t+1}^\lambda & \text{if } t < H,\\ v_\xi(\hat z_H) & \text{if } t = H.\end{cases}$$

and then minimizing the MSE.

DreamerV2 – Policy Learning

The actor is trained using two approaches:

- the REINFORCE-like loss (with a baseline), which is unbiased, but has a high variance (even with the baseline);
- the reparametrization of discrete actions using a straight-through gradient estimation, which is biased, but has lower variance.

$$\mathcal{L}(\psi) = \mathbb{E}_{p_\varphi, \pi_\psi}\Big[\sum\nolimits_{t=1}^{H-1} \Big(\underbrace{-\rho \log \pi_\psi(a_t|s_t)\,\mathrm{stop\_gradient}(V_t^\lambda - v_\xi(s_t))}_{\text{reinforce}} \underbrace{-(1-\rho)V_t^\lambda}_{\text{dynamics backprop}} \underbrace{-\eta H(a_t|s_t)}_{\text{entropy regularizer}}\Big)\Big]$$

For Atari domains, authors use $\rho = 1$ and $\eta = 10^{-3}$ (they say it works "substantially better"), while for continuous actions, $\rho = 0$ works "substantially better" (presumably because of the bias in case of discrete actions) and $\eta = 10^{-4}$ is used.

DreamerV2 – Results

The authors evaluate on 55 Atari games. They argue that the commonly used metrics have various flaws:

- **gamer-normalized median** ignores scores on half of the games,
- **gamer-normalized mean** is dominated by several games where the agent achieves super-human performance by several orders.

They therefore propose two additional ones:

- **record-normalized mean** normalizes with respect to any registered human world record for each game; however, in some games the agents still achieve super-human-record performance;
- **clipped record-normalized mean** additionally clips each score to 1; this measure is used as the primary metric in the paper.

DreamerV2 – Results

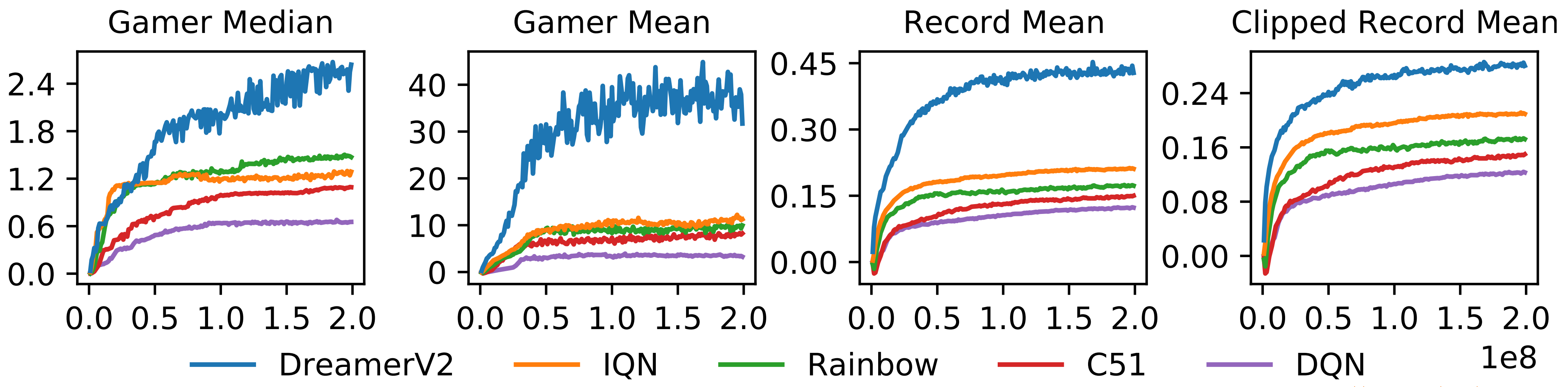


Figure 4 of "Mastering Atari with Discrete World Models", https://arxiv.org/abs/2010.02193

Agent	Gamer Median	Gamer Mean	Record Mean	Clipped Record Mean
DreamerV2	2.15	**42.26**	**0.44**	**0.28**
DreamerV2 (schedules)	**2.64**	31.71	0.43	**0.28**
IMPALA	1.92	16.72	0.34	0.23
IQN	1.29	11.27	0.21	0.21
Rainbow	1.47	9.95	0.17	0.17
C51	1.09	8.25	0.15	0.15
DQN	0.65	3.28	0.12	0.12

Table 1 of "Mastering Atari with Discrete World Models", https://arxiv.org/abs/2010.02193

Scheduling anneals actor gradient mixing ρ (from 0.1 to 0), entropy loss scale, KL, lr.

DreamerV2 – Ablations

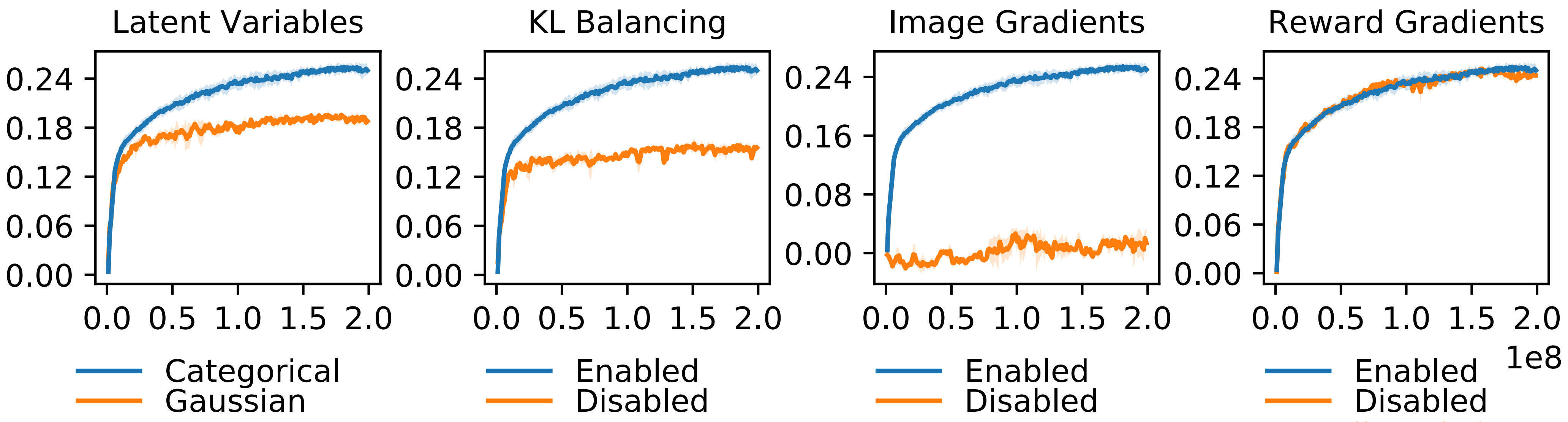


Figure 5 of "Mastering Atari with Discrete World Models", https://arxiv.org/abs/2010.02193

Agent	Gamer Median	Gamer Mean	Record Mean	Clipped Record Mean
DreamerV2	1.64	13.39	0.36	0.25
No Layer Norm	1.66	11.29	0.38	0.25
No Reward Gradients	1.68	14.29	0.37	0.24
No Discrete Latents	0.85	3.96	0.24	0.19
No KL Balancing	0.87	4.25	0.19	0.16
No Policy Reinforce	0.72	5.10	0.16	0.15
No Image Gradients	0.05	0.37	0.01	0.01

Table 2 of "Mastering Atari with Discrete World Models", https://arxiv.org/abs/2010.02193

DreamerV2 – Discrete Latent Variables

Categorical latent variables outperform Gaussian latent variables on 42 games, tie on 5 games and decrease performance on 8 games (where a tie is defined as being within 5%).

The authors provide several hypotheses why could the categorical latent variables be better:

- Categorical prior can perfectly match aggregated posterior, because mixture of categoricals is categorical, which is not true for Gaussians.
- Sparsity achieved by the 32 categorical variables with 32 classes each could be beneficial for generalization.
- Contrary to intuition, optimizing categorical variables might be easier than optimizing Gaussians, because the straight-through estimator ignores a term which would otherwise scale the gradient, which could reduce exploding/vanishing gradient problem.
- Categorical variables could be a better match for modeling discrete aspect of the Atari games (defeating an enemy, collecting reward, entering a room, …).

DreamerV2 – Comparison, Hyperparametres

Algorithm	Reward Modeling	Image Modeling	Latent Transitions	Single GPU	Trainable Parameters	Atari Frames	Accelerator Days
DreamerV2	✓	✓	✓	✓	22M	200M	10
SimPLe	✓	✓	✗	✓	74M	4M	40
MuZero	✓	✗	✓	✗	40M	20B	80
MuZero Reanalyze	✓	✗	✓	✗	40M	200M	80

Table 2 of "Mastering Atari with Discrete World Models", https://arxiv.org/abs/2010.02193

World Model			Behavior			Common		
Dataset size (FIFO)	—	$2 \cdot 10^6$	Imagination horizon	H	15	Environment steps per update	—	4
Batch size	B	50	Discount	γ	0.995	MPL number of layers	—	4
Sequence length	L	50	λ-target parameter	λ	0.95	MPL number of units	—	400
Discrete latent dimensions	—	32	Actor gradient mixing	ρ	1	Gradient clipping	—	100
Discrete latent classes	—	32	Actor entropy loss scale	η	$1 \cdot 10^{-3}$	Adam epsilon	ϵ	10^{-5}
RSSM number of units	—	600	Actor learning rate	—	$4 \cdot 10^{-5}$	Weight decay (decoupled)	—	10^{-6}
KL loss scale	β	0.1	Critic learning rate	—	$1 \cdot 10^{-4}$			
KL balancing	α	0.8	Slow critic update interval	—	100			
World model learning rate	—	$2 \cdot 10^{-4}$						
Reward transformation	—	tanh						

Table D.1 of "Mastering Atari with Discrete World Models", https://arxiv.org/abs/2010.02193

DreamerV3

DreamerV3

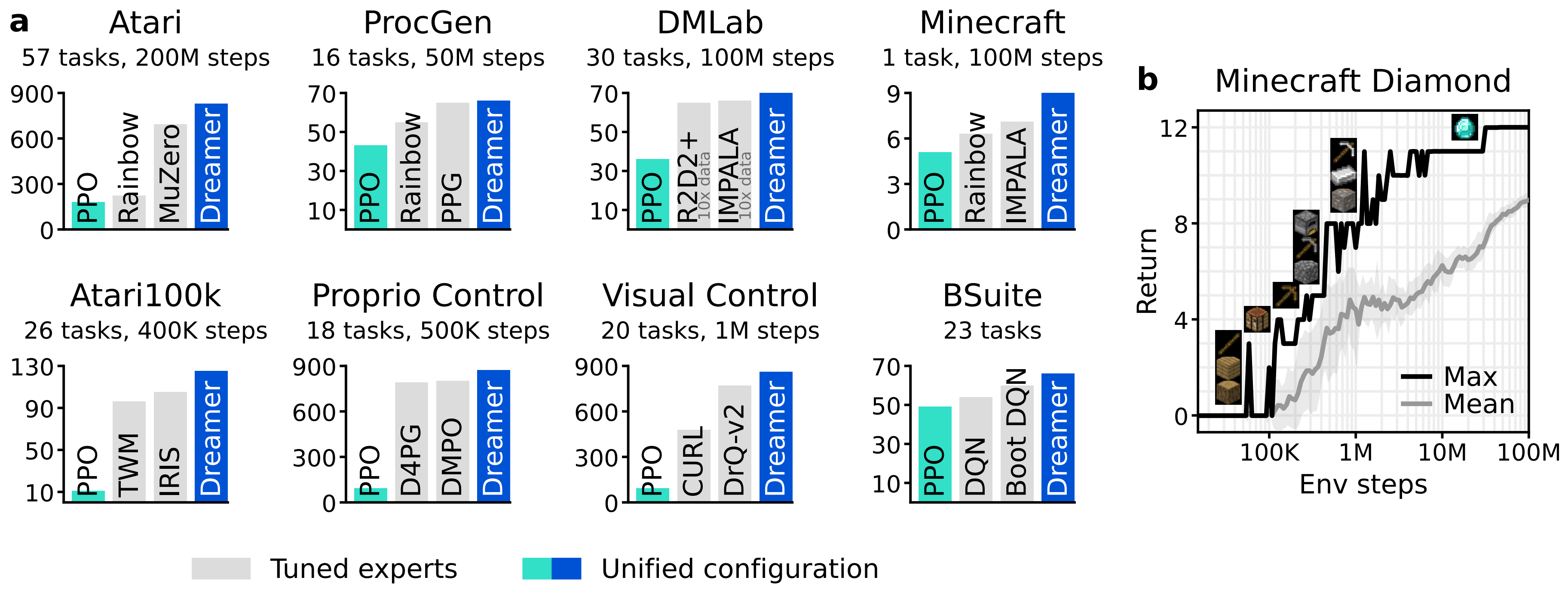
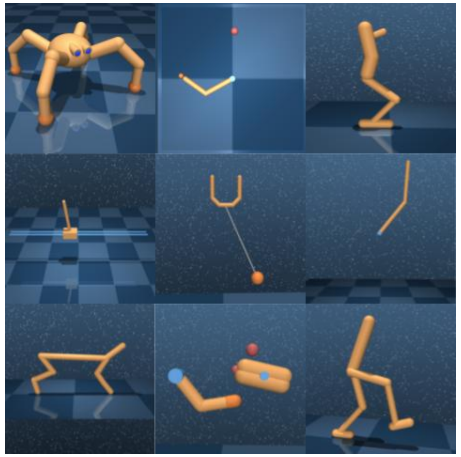
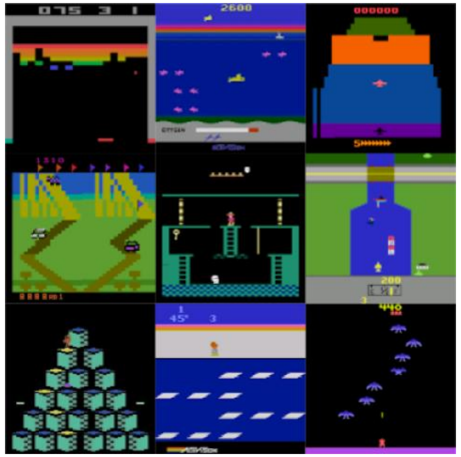


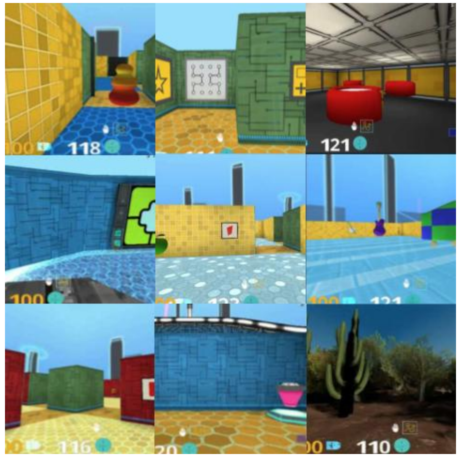
Figure 1: Benchmark summary. **a**, Using fixed hyperparameters across all domains, Dreamer outperforms tuned expert algorithms across a wide range of benchmarks and data budgets. Dreamer also substantially outperforms a high-quality implementation of the widely applicable PPO algorithm. **b**, Applied out of the box, Dreamer learns to obtain diamonds in the popular video game Minecraft from scratch given sparse rewards, a long-standing challenge in artificial intelligence for which previous approaches required human data or domain-specific heuristics.

Figure 1 of "Mastering Diverse Domains through World Models", https://arxiv.org/abs/2301.04104v2

DreamerV3


(a) Control Suite


(b) Atari

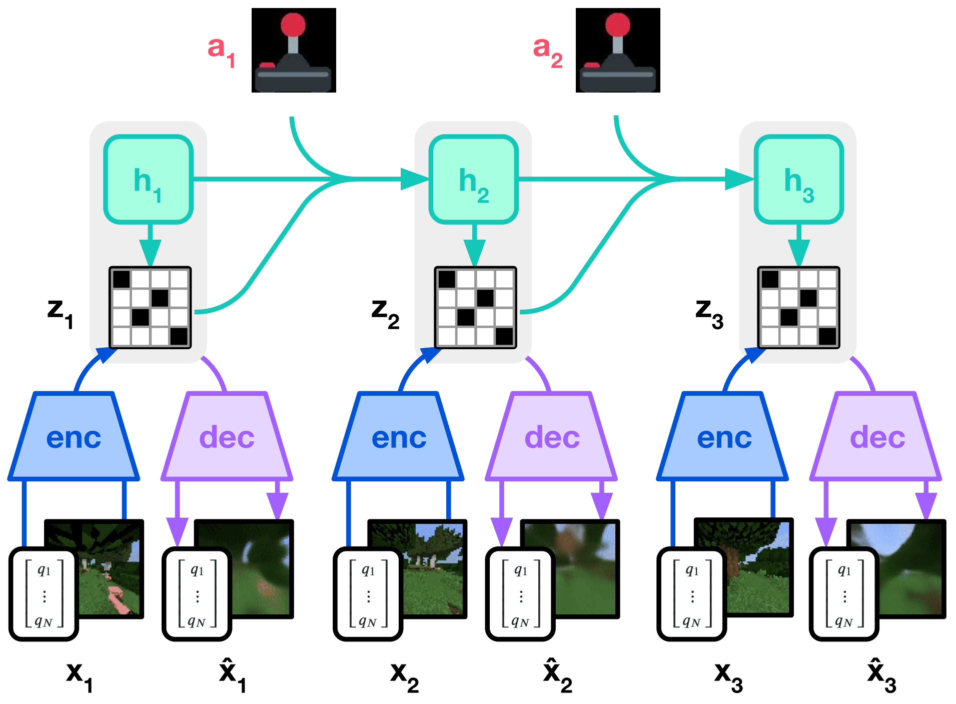

(c) DMLab

(d) Minecraft

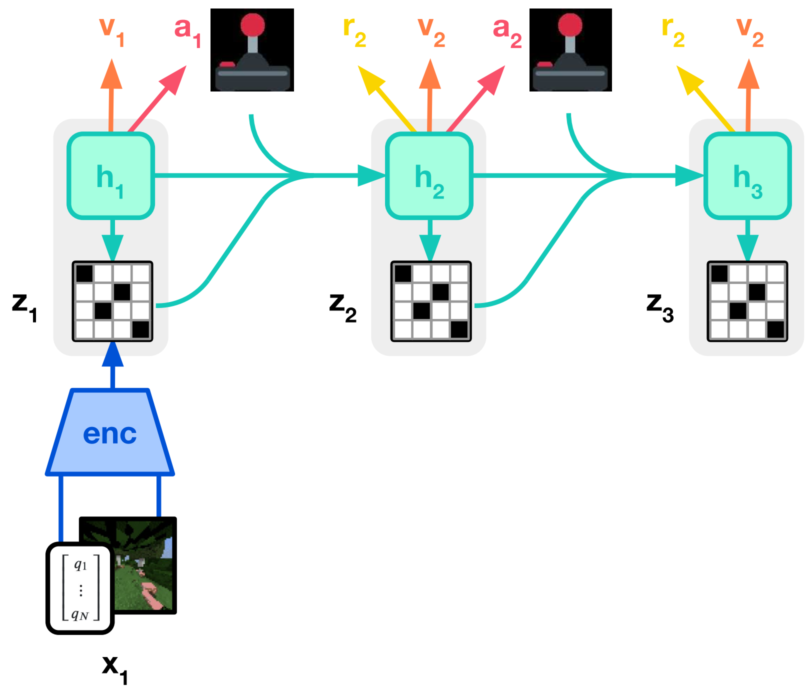
Figure 2: Four visual domains considered in this work. DreamerV3 succeeds across these diverse domains, ranging from robot locomotion and manipulation tasks over Atari games with 2D graphics to complex 3D domains such as DMLab and Minecraft that require spatial and temporal reasoning.

Figure 2 of "Mastering Diverse Domains through World Models", https://arxiv.org/abs/2301.04104v1

DreamerV3



(a) World Model Learning



(b) Actor Critic Learning

Figure 3: Training process of DreamerV3. The world model encodes sensory inputs into a discrete representation z_t that is predicted by a sequence model with recurrent state h_t given actions a_t. The inputs are reconstructed as learning signal to shape the representations. The actor and critic learn from trajectories of abstract representations predicted by the world model.

Figure 3 of "Mastering Diverse Domains through World Models", https://arxiv.org/abs/2301.04104v1

DreamerV3

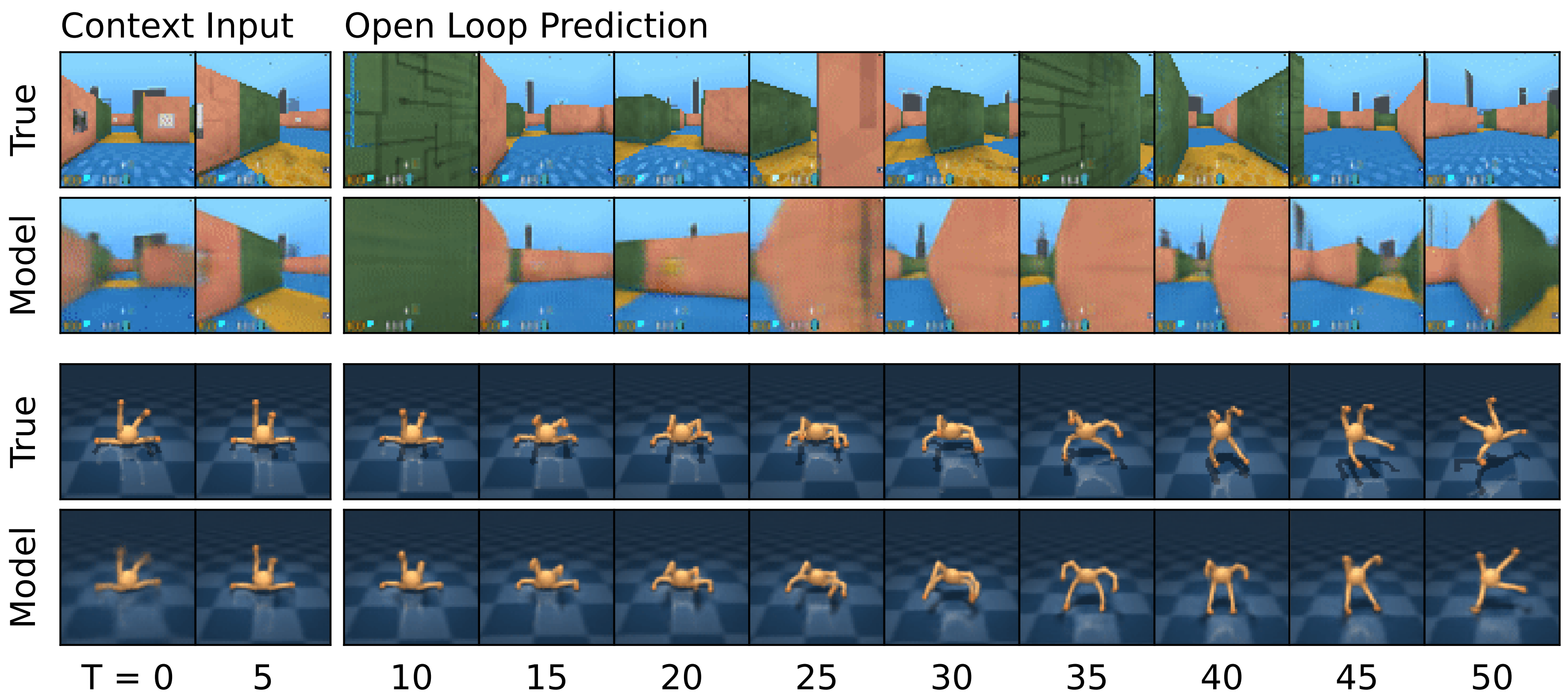


Figure 5: Multi-step video predictions in DMLab (top) and Control Suite (bottom). From 5 frames of context input, the model predicts 45 steps into the future given the action sequence and without access to intermediate images. The world model learns an understanding of the underlying 3D structure of the two environments. Refer to Appendix H for additional video predictions.

Figure 5 of "Mastering Diverse Domains through World Models", https://arxiv.org/abs/2301.04104v1

DreamerV3

To be able to predict rewards and returns in different scales, the authors propose to use symmetrical and shifted logarithm:

$$\operatorname{symlog}(x) \overset{\text{def}}{=} \operatorname{sign}(x)\log\big(|x|+1\big),$$
$$\operatorname{symexp}(x) \overset{\text{def}}{=} \operatorname{sign}(x)\big(\exp(|x|)-1\big).$$

The loss could then be

$$\mathcal{L}(\boldsymbol\theta) \overset{\text{def}}{=} \tfrac{1}{2}\big(f(\boldsymbol x;\boldsymbol\theta) - \operatorname{symlog}(y)\big)^2,$$

and we can reconstruct the original quantity as

$$\hat y \overset{\text{def}}{=} \operatorname{symexp}\big(f(\boldsymbol x;\boldsymbol\theta)\big).$$

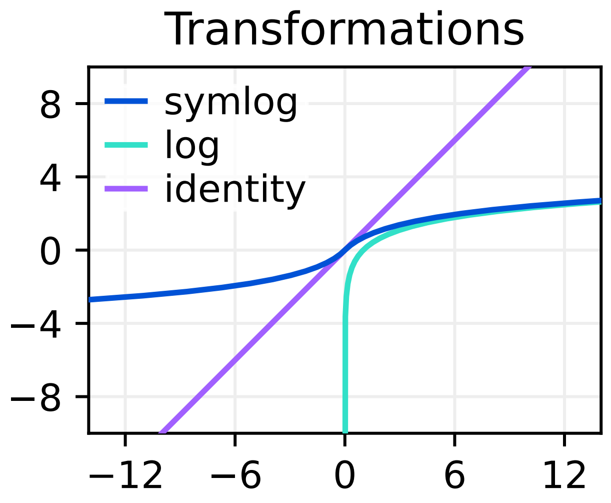


Figure 4: The symlog function compared to logarithm and identity.

Figure 4 of "Mastering Diverse Domains through World Models", https://arxiv.org/abs/2301.04104v1

We can use this approach to predict also a whole distribution, trained using twohot loss:

$$B \overset{\text{def}}{=} \operatorname{symexp}\big([-20,\ldots,20]\big),\qquad \hat y \overset{\text{def}}{=} \operatorname{softmax}\big(f(\boldsymbol x;\boldsymbol\theta)\big)^T B,\qquad \mathcal{L}(\boldsymbol\theta) \overset{\text{def}}{=} -\operatorname{twohot}^T \log \hat y.$$

DreamerV3

The world model is the same as in DreamerV2, here using the original notation:

$$\begin{array}{lll} \text{RSSM} & \text{sequence model:} & h_t = f_\varphi(h_{t-1}, s_{t-1}, a_{t-1}), \\ \text{RSSM} & \text{encoder:} & s_t \sim q_\varphi(s_t | h_t, x_t), \\ \text{RSSM} & \text{dynamics predictor:} & \bar{s}_t \sim p_\varphi(\bar{s}_t | h_t), \\ & \text{decoder:} & \bar{x}_t \sim p_\varphi(\bar{x}_t | h_t, s_t), \\ & \text{reward predictor:} & \bar{r}_t \sim p_\varphi(\bar{r}_t | h_t, s_t), \\ & \text{continue predictor:} & \bar{c}_t \sim p_\varphi(\bar{c}_t | h_t, s_t). \end{array}$$

The observations are transformed using the symlog function, both on the encoder input and decoder targets.

DreamerV3

The overall world model loss is:

$$\mathcal{L}(\boldsymbol\varphi) \overset{\text{def}}{=} \mathbb{E}_{q_\varphi}\Big[\sum_{t=1}^T \big(\underbrace{\beta_\text{pred}}_{1.0}\mathcal{L}_\text{pred}(\boldsymbol\varphi) + \underbrace{\beta_\text{dyn}}_{1.0}\mathcal{L}_\text{dyn}(\boldsymbol\varphi) + \underbrace{\beta_\text{rep}}_{0.1}\mathcal{L}_\text{rep}(\boldsymbol\varphi)\big)\Big],$$

where the individual components are

$$\begin{aligned}
\mathcal{L}_\text{pred}(\boldsymbol\varphi) &\overset{\text{def}}{=} -\log p_{\boldsymbol\varphi}(x_t|h_t, s_t) - \log p_{\boldsymbol\varphi}(r_t|h_t, s_t) - \log p_{\boldsymbol\varphi}(c_t|h_t, s_t),\\
\mathcal{L}_\text{dyn}(\boldsymbol\varphi) &\overset{\text{def}}{=} \max\Big(1, D_\text{KL}\big(\operatorname{sg}(q_{\boldsymbol\theta}(s_t|h_t, x_t))\,\big\|\quad p_{\boldsymbol\varphi}(s_t|h_t)\ \big)\Big),\\
\mathcal{L}_\text{rep}(\boldsymbol\varphi) &\overset{\text{def}}{=} \max\Big(1, D_\text{KL}\big(\quad q_{\boldsymbol\theta}(s_t|h_t, x_t)\ \big\|\operatorname{sg}(p_{\boldsymbol\varphi}(s_t|h_t))\big)\Big).
\end{aligned}$$

To stabilize training, the authors use *free bits* (clipping the dynamics and representation losses below 1 nat ≈ 1.44 bits) and parametrize the categorical distributions of the encoder, dynamics predictor, and actor distribution as a mixture of 1% uniform and 99% neural network output.

DreamerV3

The critic is trained to predict from the current state $z_t = \{h_t, s_t/\bar{s}_t\}$ the return $v_t(z_t; \boldsymbol\varphi)$ as a categorical distribution over exponentially spaced bins B.

As target, the boostrapped λ-return is used during training:

$$R_t^\lambda \stackrel{\text{def}}{=} r_t + \gamma \bar{c}_t \big((1-\lambda)v_t + \lambda R_{t+1}^\lambda\big),$$

where for the imaginary horizon $T = 16$, we use just the bootstrapped return $R_T^\lambda \stackrel{\text{def}}{=} v_T$.

The critic is trained using a mixture of imaginary trajectories $\{h_t, \bar{s}_t\}$ with loss scale $\beta_{val} = 1$ and trajectories sample from the replay buffer $\{h_t, s_t\}$ with loss scale $\beta_{repval} = 0.3$.

The training is stabilized by a regularization loss of the critic to its exponentially moving average of its own parameters, allowing to use the current network for computing the returns.

DreamerV3

The actor is trained using entropy-regularized REINFORCE loss, for both discrete and continuous actions:

$$\mathcal{L}(\boldsymbol{\theta}) \stackrel{\text{def}}{=} -\sum_{t=1}^{T} \operatorname{sg}\Big(\big(R_t^\lambda - v_t(z_t;\boldsymbol{\varphi})/\max(1,S)\big)\Big) \log \pi(a_t|z_t;\boldsymbol{\theta}) + \eta H\big[\pi(a_t|s)t;\boldsymbol{\theta})\big],$$

where the returns are normalized using

$$S \stackrel{\text{def}}{=} \operatorname{EMA}\big(\operatorname{Quantile}(R_t^\lambda, 0.95) - \operatorname{Quantile}(R_t^\lambda, 0.05), 0.99\big).$$

During training, uniform sampling is used, even if authors mention that prioritized replay improved performance.

Each batch is a combination of online data (from the current interactions) and data sampled from the replay buffer and then followed by the actor and the world model.

The *replay ratio* is the number of time steps trained for every single step collected from the environment (without action repeat); e.g., for a replay ratio of 32, action repeat (frame skip) 4, and batches of 64 sequences of 16 steps, a gradient update is performed every $4 \cdot 64 \cdot 16/32 = 128$ environment steps.

DreamerV3

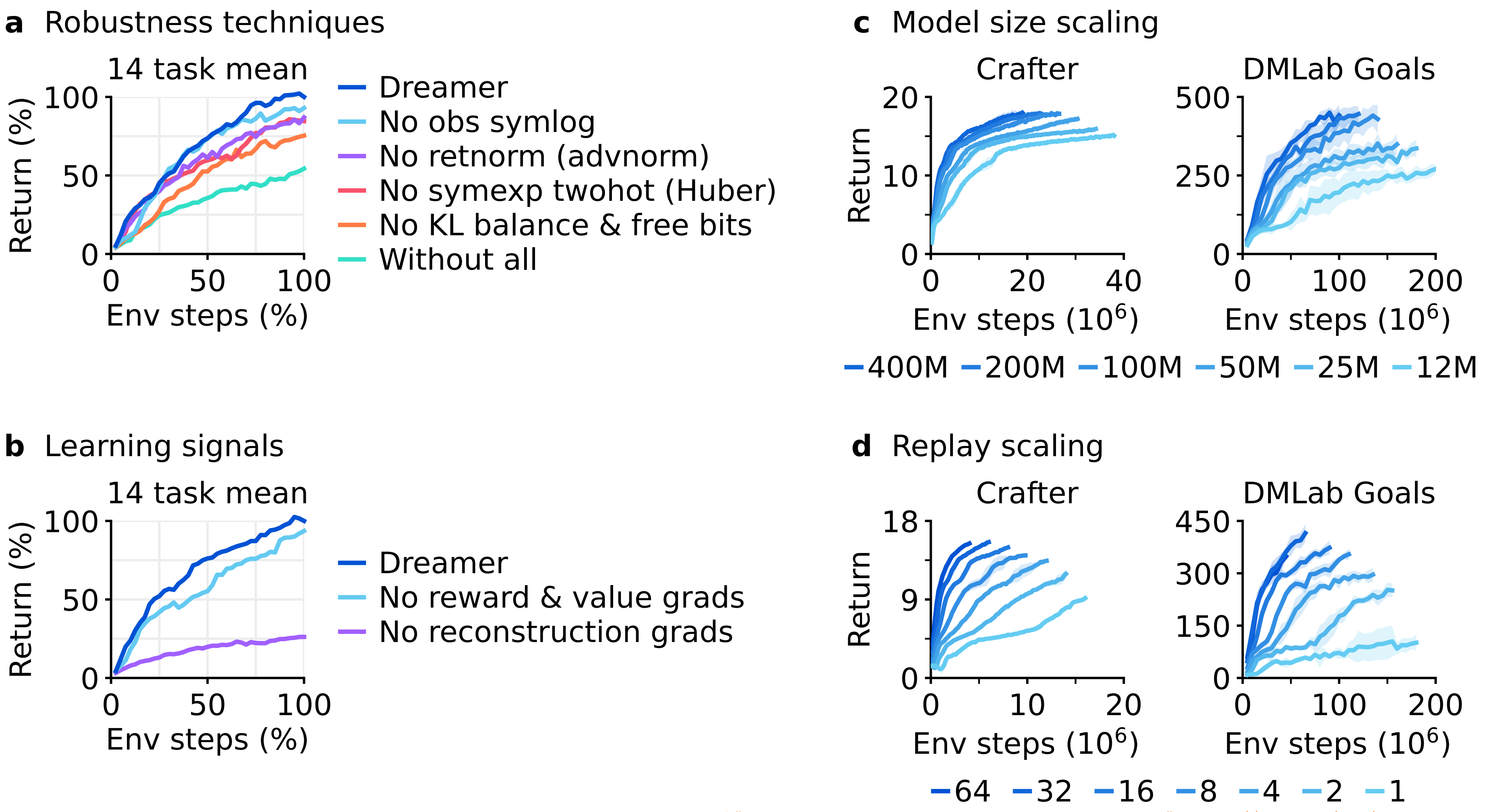


Figure 6 of "Mastering Diverse Domains through World Models", https://arxiv.org/abs/2301.04104v2

DreamerV3

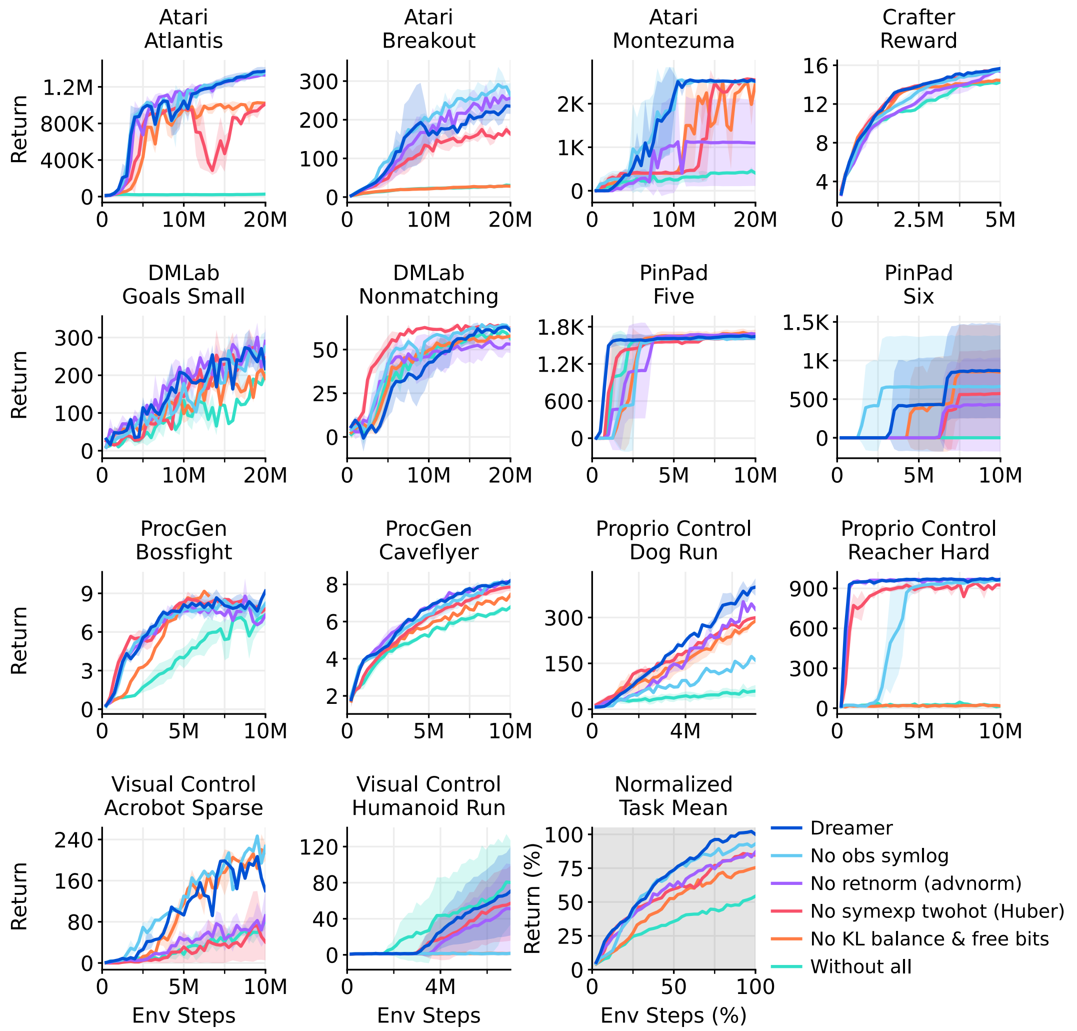


Figure 17 of "Mastering Diverse Domains through World Models", https://arxiv.org/abs/2301.04104v2

DreamerV3

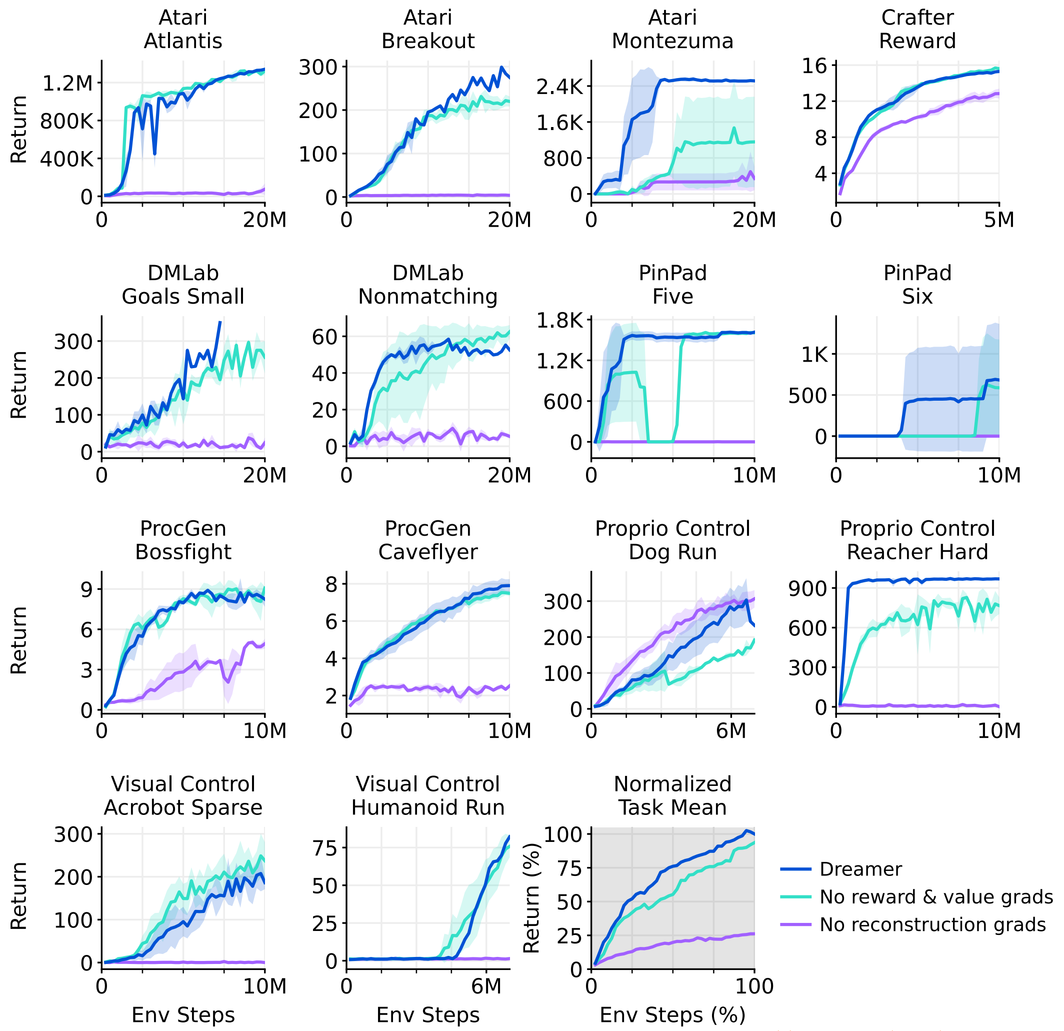


Figure 18 of "Mastering Diverse Domains through World Models", https://arxiv.org/abs/2301.04104v2

DreamerV3

Benchmark	Tasks	Env Steps	Action Repeat	Env Instances	Replay Ratio	GPU Days	Model Size
Minecraft	1	100M	1	64	32	8.9	200M
DMLab	30	100M	4	16	32	2.9	200M
ProcGen	16	50M	1	16	64	16.1	200M
Atari	57	200M	4	16	32	7.7	200M
Atari100K	26	400K	4	1	128	0.1	200M
BSuite	23	—	1	1	1024	0.5	200M
Proprio Control	18	500K	2	16	512	0.3	12M
Visual Control	20	1M	2	16	512	0.1	12M

Table 2: Benchmark overview. All agents were trained on a single Nvidia A100 GPU each.

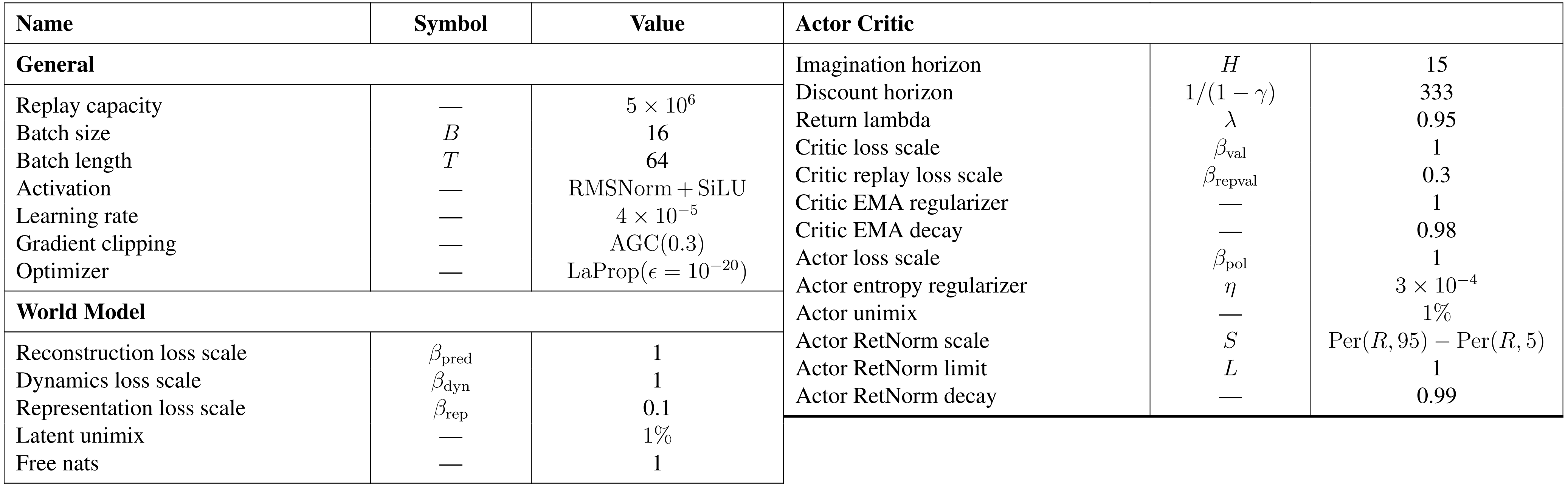
Table 2 of "Mastering Diverse Domains through World Models", https://arxiv.org/abs/2301.04104v2

Parameters	12M	25M	50M	100M	200M	400M
Hidden size (d)	256	384	512	768	1024	1536
Recurrent units ($8d$)	1024	3072	4096	6144	8192	12288
Base CNN channels ($d/16$)	16	24	32	48	64	96
Codes per latent ($d/16$)	16	24	32	48	64	96

Table 3: Dreamer model sizes. The number of MLP hidden units defines the model dimension, from which recurrent units, convolutional channels, and number of codes per latent are derived. The number of layers and latents is constant across model sizes.

Table 3 of "Mastering Diverse Domains through World Models", https://arxiv.org/abs/2301.04104v2

DreamerV3



Name	Symbol	Value
General		
Replay capacity	—	5×10^6
Batch size	B	16
Batch length	T	64
Activation	—	$\mathrm{RMSNorm} + \mathrm{SiLU}$
Learning rate	—	4×10^{-5}
Gradient clipping	—	$\mathrm{AGC}(0.3)$
Optimizer	—	$\mathrm{LaProp}(\epsilon = 10^{-20})$
World Model		
Reconstruction loss scale	β_{pred}	1
Dynamics loss scale	β_{dyn}	1
Representation loss scale	β_{rep}	0.1
Latent unimix	—	1%
Free nats	—	1
Actor Critic		
Imagination horizon	H	15
Discount horizon	$1/(1-\gamma)$	333
Return lambda	λ	0.95
Critic loss scale	β_{val}	1
Critic replay loss scale	β_{repval}	0.3
Critic EMA regularizer	—	1
Critic EMA decay	—	0.98
Actor loss scale	β_{pol}	1
Actor entropy regularizer	η	3×10^{-4}
Actor unimix	—	1%
Actor RetNorm scale	S	$\mathrm{Per}(R, 95) - \mathrm{Per}(R, 5)$
Actor RetNorm limit	L	1
Actor RetNorm decay	—	0.99

Table 4: Dreamer hyperparameters. The same values are used across all benchmarks, including proprioceptive and visual inputs, continuous and discrete actions, and 2D and 3D domains. We do not use any hyperparameter annealing, prioritized replay, weight decay, or dropout.

Table 4 of "Mastering Diverse Domains through World Models", https://arxiv.org/abs/2301.04104v1

MERLIN

MERLIN

In a partially-observable environment, keeping all information in the RNN state is substantially limiting. Therefore, *memory-augmented* networks can be used to store suitable information in external memory (in the lines of NTM, DNC, or MANN models).

We now describe an approach used by Merlin architecture (*Unsupervised Predictive Memory in a Goal-Directed Agent* DeepMind Mar 2018 paper).

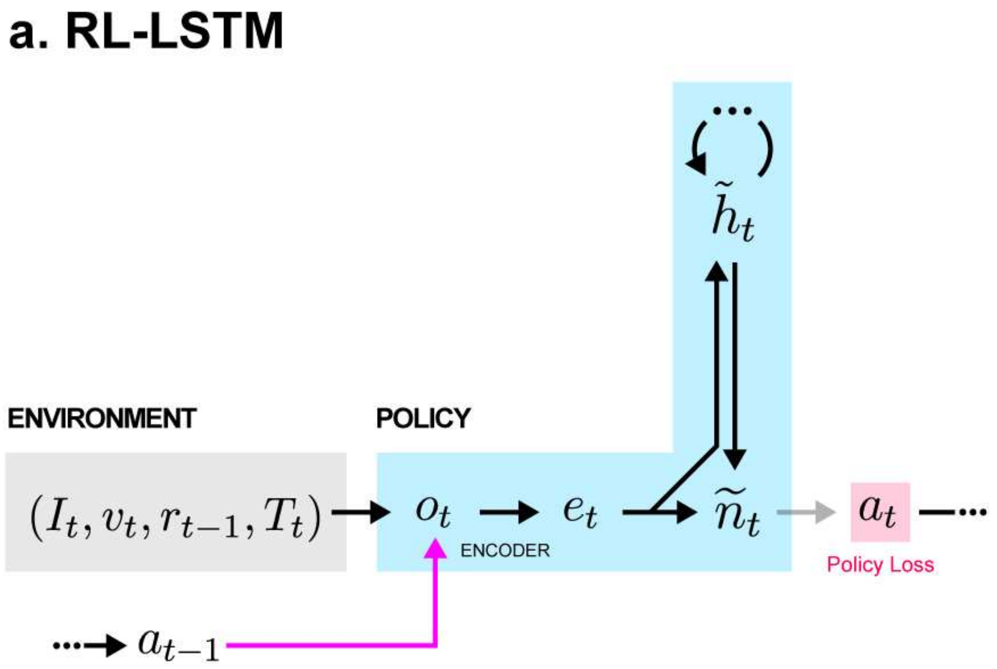


Figure 1a of "Unsupervised Predictive Memory in a Goal-Directed Agent", https://arxiv.org/abs/1803.10760

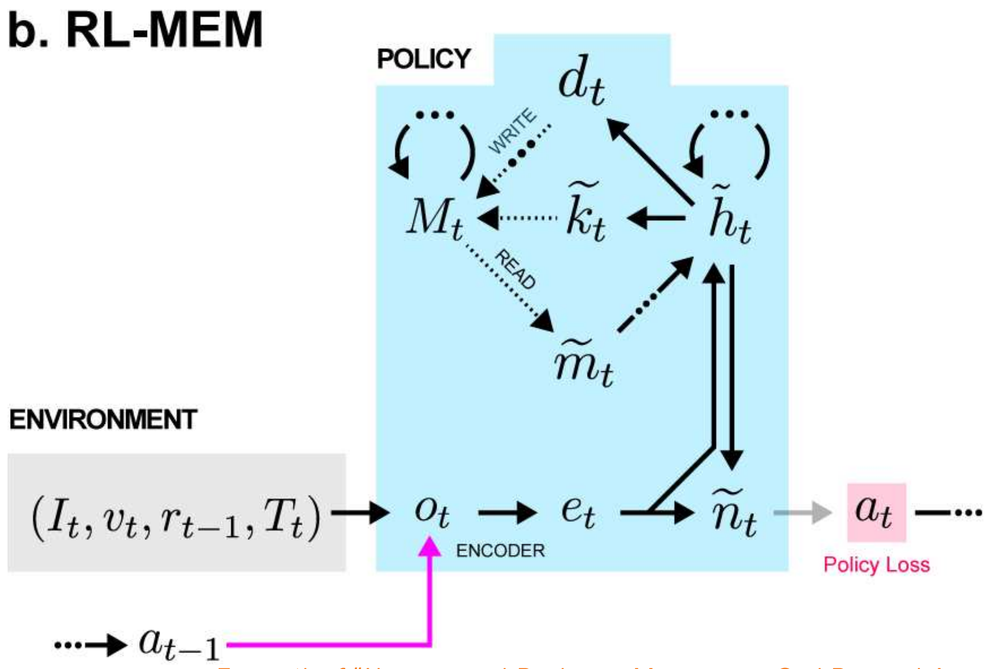


Figure 1b of "Unsupervised Predictive Memory in a Goal-Directed Agent", https://arxiv.org/abs/1803.10760

MERLIN – Memory Module

Let $\boldsymbol{M}$ be a memory matrix of size $N_{mem} \times 2|\boldsymbol{e}|$.

Assume we have already encoded observations as $\boldsymbol{e}_t$ and previous action a_{t-1}. We concatenate them with K previously read vectors and process them by a deep LSTM (two layers are used in the paper) to compute $\boldsymbol{h}_t$.

Then, we apply a linear layer to $\boldsymbol{h}_t$, computing K key vectors $\boldsymbol{k}_1, \ldots, \boldsymbol{k}_K$ of length $2|\boldsymbol{e}|$ and K positive scalars $\beta_1, \ldots, \beta_K$.

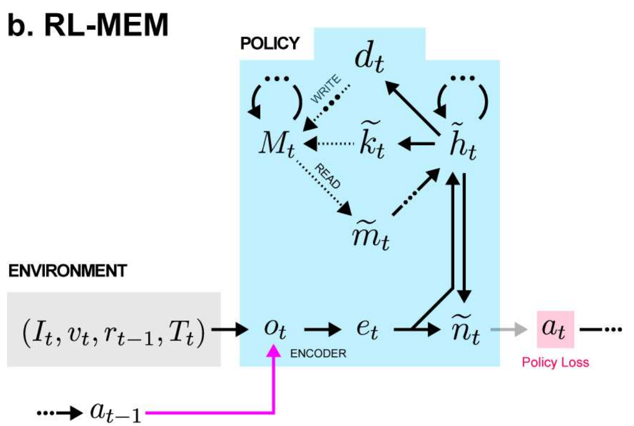


Figure 1b of "Unsupervised Predictive Memory in a Goal-Directed Agent", https://arxiv.org/abs/1803.10760

Reading: For each i, we compute cosine similarity of $\boldsymbol{k}_i$ and all memory rows $\boldsymbol{M}_j$, multiply the similarities by β_i and pass them through a $\operatorname{softmax}$ to obtain weights $\boldsymbol{\omega}_i$. The read vector is then computed as $\boldsymbol{M}\boldsymbol{\omega}_i$.

Writing: We find one-hot write index $\boldsymbol{v}_{wr}$ to be the least used memory row (we keep usage indicators and add read weights to them). We then compute $\boldsymbol{v}_{ret} \leftarrow \gamma \boldsymbol{v}_{ret} + (1-\gamma)\boldsymbol{v}_{wr}$, and retroactively update the memory matrix using $\boldsymbol{M} \leftarrow \boldsymbol{M} + \boldsymbol{v}_{wr}[\boldsymbol{e}_t, 0] + \boldsymbol{v}_{ret}[0, \boldsymbol{e}_t]$.

MERLIN — Prior and Posterior

However, updating the encoder and memory content purely using RL is inefficient. Therefore, MERLIN includes a *memory-based predictor (MBP)* in addition to policy. The goal of MBP is to compress observations into low-dimensional state representations $\boldsymbol{z}$ and storing them in memory.

We want the state variables not only to faithfully represent the data, but also emphasise rewarding elements of the environment above irrelevant ones. To accomplish this, the authors follow the hippocampal representation theory of Gluck and Myers, who proposed that hippocampal representations pass through a compressive bottleneck and then reconstruct input stimuli together with task reward.

In MERLIN, a (Gaussian diagonal) *prior* distribution over $\boldsymbol{z}_t$ predicts next state variable conditioned on history of state variables and actions $p(\boldsymbol{z}_t^\text{prior}|\boldsymbol{z}_{t-1}, a_{t-1}, \ldots, \boldsymbol{z}_1, a_1)$, and *posterior* corrects the prior using the new observation $\boldsymbol{o}_t$, forming a better estimate $q(\boldsymbol{z}_t|\boldsymbol{o}_t, \boldsymbol{z}_t^\text{prior}, \boldsymbol{z}_{t-1}, a_{t-1}, \ldots, \boldsymbol{z}_1, a_1) + \boldsymbol{z}_t^\text{prior}$.

MERLIN — Prior and Posterior

To achieve the mentioned goals, we add two terms to the loss.

- We try reconstructing input stimuli, action, reward and return using a sample from the state variable posterior, and add the difference of the reconstruction and ground truth to the loss.
- We also add KL divergence of the prior and the posterior to the loss, to ensure consistency between the prior and the posterior.

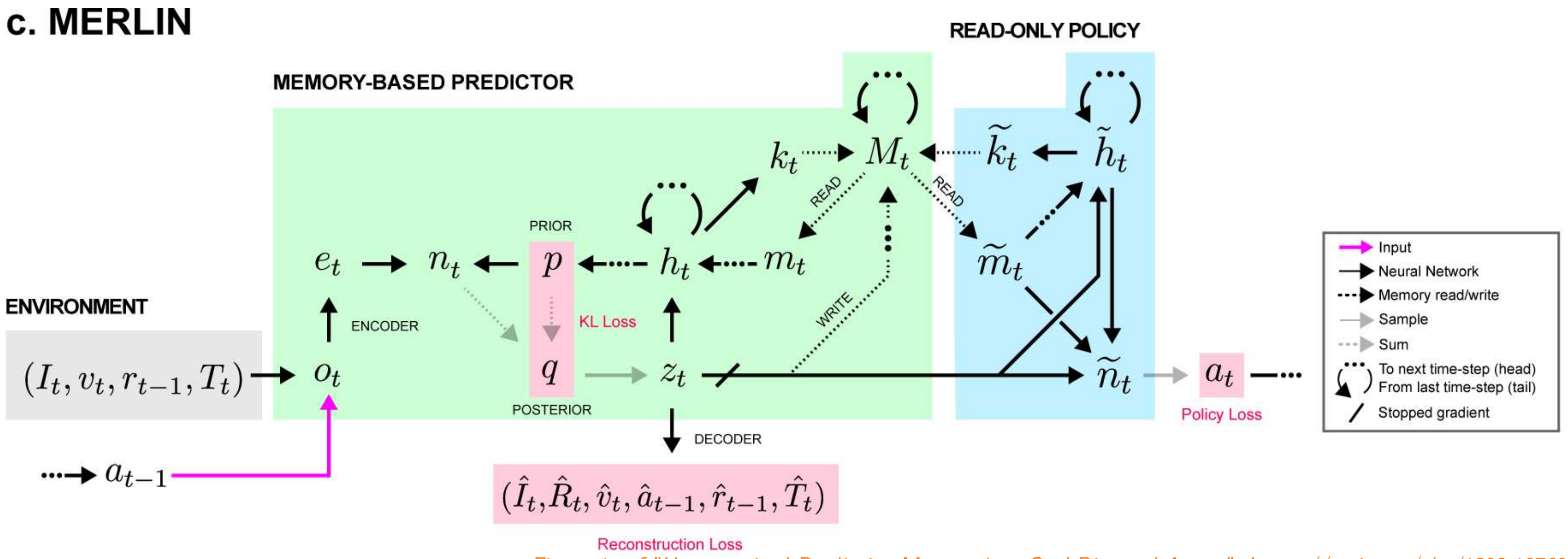


Figure 1c of "Unsupervised Predictive Memory in a Goal-Directed Agent", https://arxiv.org/abs/1803.10760

MERLIN — Algorithm

Algorithm 1 MERLIN Worker Pseudocode

```
// Assume global shared parameter vectors $\theta$ for the policy network and $\chi$ for the memory-based predictor; global shared counter $T := 0$
// Assume thread-specific parameter vectors $\theta', \chi'$
// Assume discount factor $\gamma \in (0, 1]$ and bootstrapping parameter $\lambda \in [0, 1]$
Initialize thread step counter $t := 1$
repeat
    Synchronize thread-specific parameters $\theta' := \theta; \chi' := \chi$
    Zero model's memory & recurrent state if new episode begins
    $t_\text{start} := t$
    repeat
        Prior $\mathcal{N}(\mu_t^\text{p}, \log\Sigma_t^\text{p}) = p(h_{t-1}, m_{t-1})$
        $e_t = \text{enc}(o_t)$
        Posterior $\mathcal{N}(\mu_t^\text{q}, \log\Sigma_t^\text{q}) = q(e_t, h_{t-1}, m_{t-1}, \mu_t^\text{p}, \log\Sigma_t^\text{p})$
        Sample $z_t \sim \mathcal{N}(\mu_t^\text{q}, \log\Sigma_t^\text{q})$
        Policy network update $\tilde{h}_t = \text{rec}(\tilde{h}_{t-1}, \tilde{m}_t, \text{StopGradient}(z_t))$
        Policy distribution $\pi_t = \pi(\tilde{h}_t, \text{StopGradient}(z_t))$
        Sample $a_t \sim \pi_t$
        $h_t = \text{rec}(h_{t-1}, m_t, z_t)$
        Update memory with $z_t$ by Methods Eq. 2
        $R_t, o_t^r = \text{dec}(z_t, \pi_t, a_t)$
        Apply $a_t$ to environment and receive reward $r_t$ and observation $o_{t+1}$
        $t := t + 1; T := T + 1$
    until environment termination or $t - t_\text{start} == \tau_\text{window}$
    If not terminated, run additional step to compute $V_\nu^\pi(z_{t+1}, \log\pi_{t+1})$
    and set $R_{t+1} := V^\pi(z_{t+1}, \log\pi_{t+1})$ // (but don't increment counters)
    Reset performance accumulators $\mathcal{A} := 0; \mathcal{L} := 0; \mathcal{H} := 0$
    for $k$ from $t$ down to $t_\text{start}$ do
        $\gamma_t := \begin{cases} 0, & \text{if } k \text{ is environment termination} \\ \gamma, & \text{otherwise} \end{cases}$
        $R_k := r_k + \gamma_t R_{k+1}$
        $\delta_k := r_k + \gamma_t V^\pi(z_{k+1}, \log\pi_{k+1}) - V^\pi(z_k, \log\pi_k)$
        $A_k := \delta_k + (\gamma\lambda)A_{k+1}$
        $\mathcal{L} := \mathcal{L} + \mathcal{L}_k$ (Eq. 7)
        $\mathcal{A} := \mathcal{A} + A_k \log\pi_k[a_k]$
        $\mathcal{H} := \mathcal{H} - \alpha_\text{entropy} \sum_i \pi_k[i] \log\pi_k[i]$  (Entropy loss)
    end for
    $d\chi' := \nabla_{\chi'}\mathcal{L}$
    $d\theta' := \nabla_{\theta'}(\mathcal{A} + \mathcal{H})$
    Asynchronously update via gradient ascent $\theta$ using $d\theta'$ and $\chi$ using $d\chi'$
until $T > T_\text{max}$
```

Algorithm 1 of "Unsupervised Predictive Memory in a Goal-Directed Agent", https://arxiv.org/abs/1803.10760

MERLIN

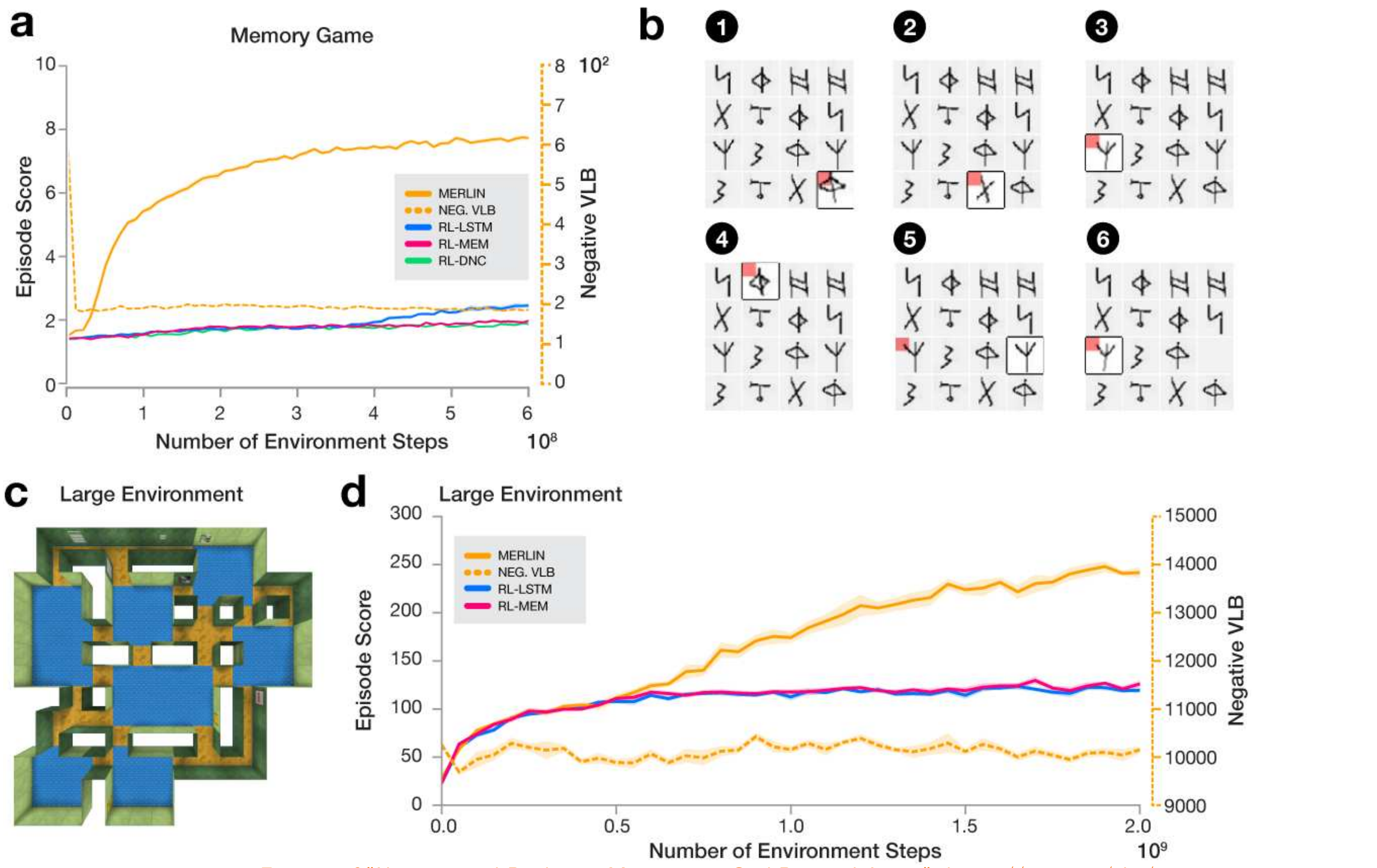


Figure 2 of "Unsupervised Predictive Memory in a Goal-Directed Agent", https://arxiv.org/abs/1803.10760

MERLIN

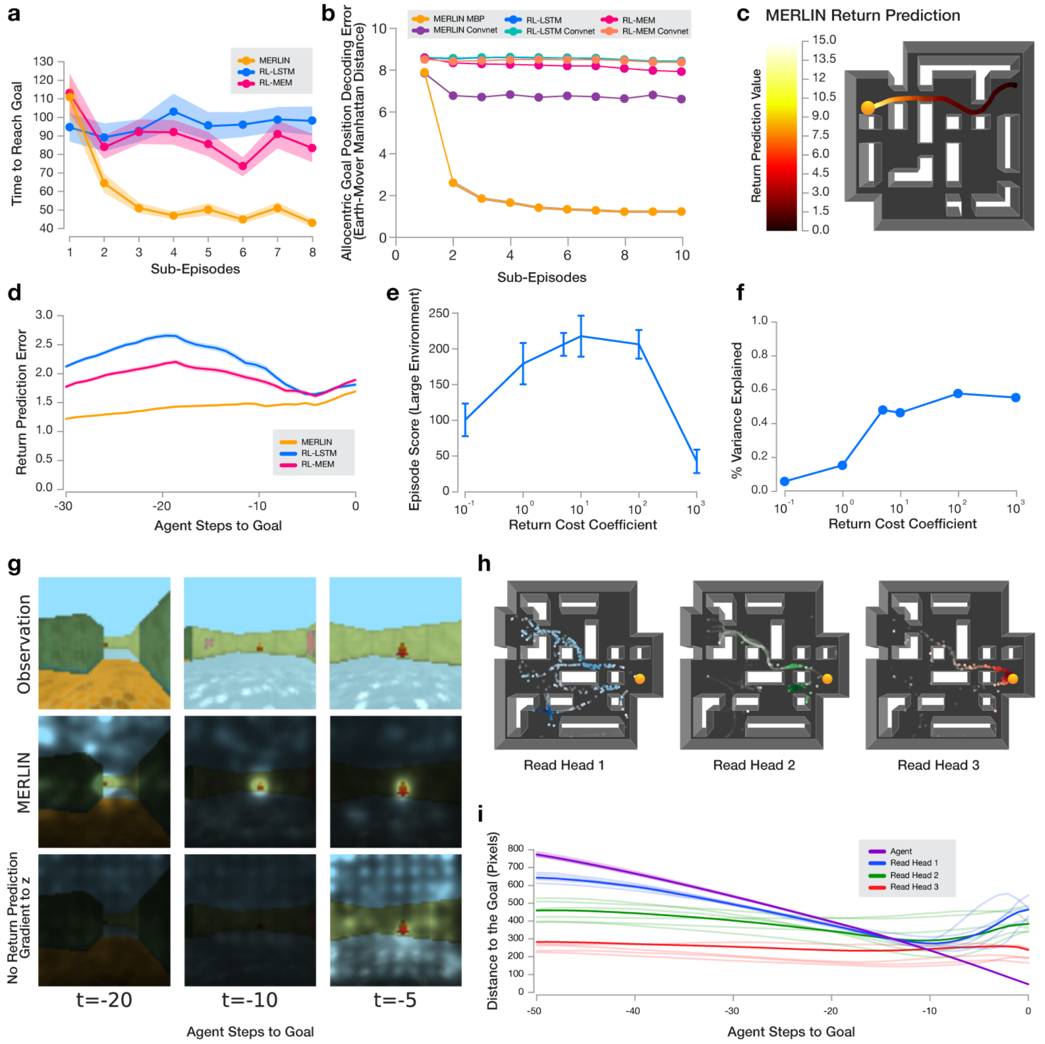
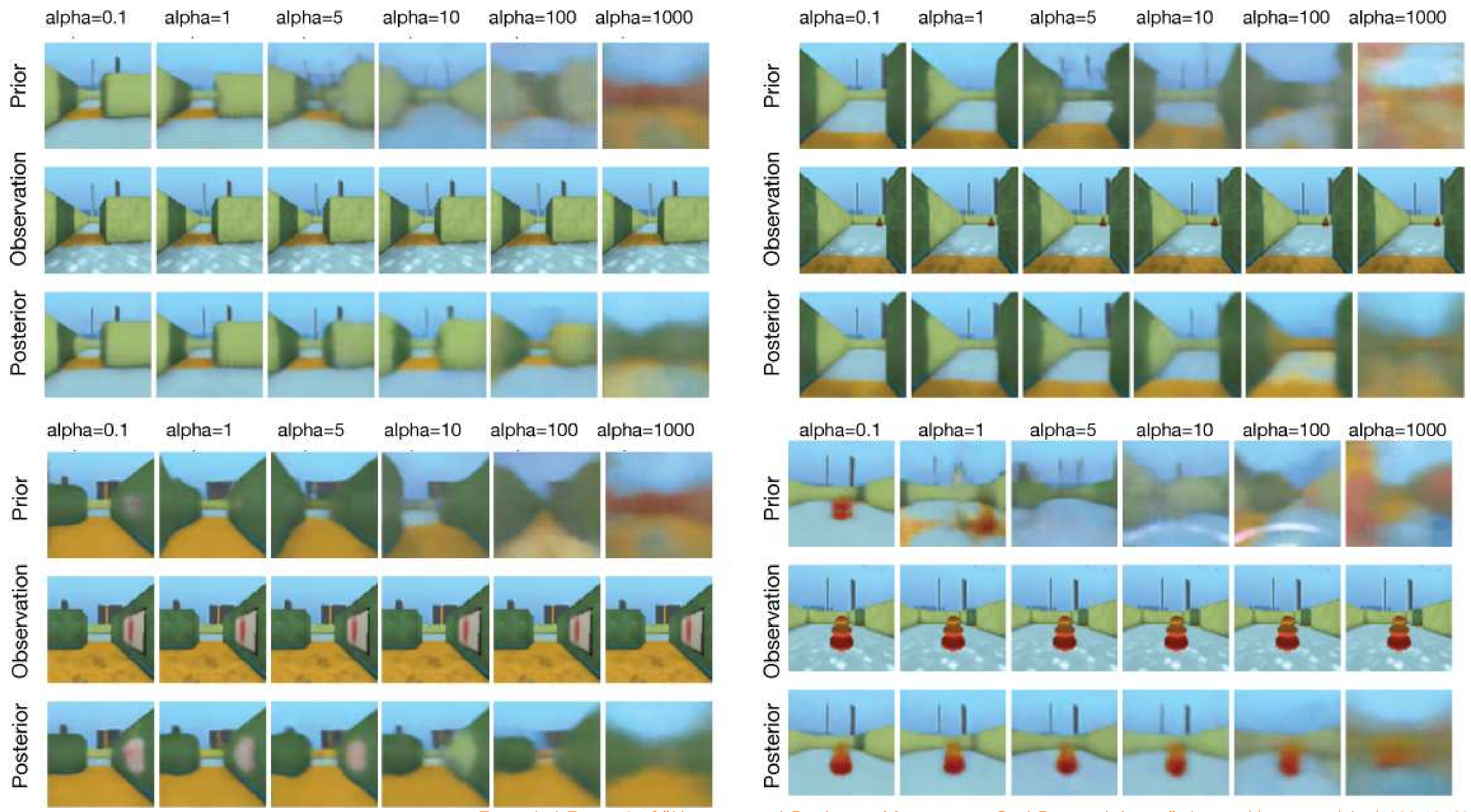


Figure 3 of "Unsupervised Predictive Memory in a Goal-Directed Agent", https://arxiv.org/abs/1803.10760

MERLIN



Extended Figure 3 of "Unsupervised Predictive Memory in a Goal-Directed Agent", https://arxiv.org/abs/1803.10760