Learning Latent Dynamics for Planning from Pixels

E6691.2022Spring.WMRL.SF3043.TJG2148

Sam Fieldman
M.S Computer Science
Columbia University
sf3043@Columbia.edu

Trevor Gordon
M.S Electrical Engineering
Columbia University
tjg2148@columbia.edu

***Abstract*—Planning with known environment dynamics is a highly effective way to solve complex control problems. However, for unobserved environments, we must utilize observations of agent interaction to learn models of the world. Deep Planning Network (PlaNet) is an approach that learns the approximate environment dynamics from images, and chooses actions through fast online planning in latent space. PlaNet uses a latent dynamics model, which contains both deterministic and stochastic transition components, to solve continuous control tasks which exceed the difficulty of tasks previously solved with similar methods. PlaNet is also incredibly data-efficient, and outperforms several STOT model-free Reinforcement Learning methods with on average 200× fewer environment interactions and similar computation time. In this report we reproduce the key results of the paper Hafner et al. [2018] and breakdown key improvements made in the original paper to show their additive performance.**

I. Introduction

Following the success of Reinforcement Learning on highly complex environments such as Go, StarCraft, and robotic control (Silver et al. [2016], Arulkumaran et al. [2019], OpenAI et al. [2019]) reinforcement learning has become a prominent approach for learning optimal decision making strategies in highly complex domains. However, deep reinforcement learning is notoriously data hungry, and must actively interact with the environment to collect this data. This makes it difficult to utilize reinforcement learning in setting where data collection is limited by cost of safety restriction, or where an accurate simulator is unavailable.

Model-Based reinforcement learning has become one increasingly effective ways of tackling this issue of data-efficiency. This paradigm utilizes the collected data to train an approximate model of the underlying environment, which we can then utilize for both planning and off-line training. Model-based planning can be more data efficient as it is able to leverage a richer training signal (entire images versus scalar rewards) and, in the case of Model-Free RL methods, does not require propagating rewards through Bellman backups. Further, by utilizing planning, we can often increase performance simply by increasing the computational budget of our action search Silver et al. [2017]. Finally, the learned dynamics are agnostic of the underlying task and can therefore be used to perform transfer learning to other unobserved tasks in the environment.

Despite the promise of Model-Based approaches, constructing an accurate approximation of the environment is a non-trivial problem. Even in the case of working at the level of a robots proprioceptive signal (joint activation and velocities), the state space can grow unwieldy as the complexity of the robot increases. To tackle this problem, the papers Hafner et al. [2018] and Ha and Schmidhuber [2018] utilize Variational AutoEncoders to reduce the dimensionality of the observed state space along with both feed-forward and recurrent networks to model the transition dynamics in this learned latent space. This modeling approach is very similar to that proposed in the works of Doerr et al. [2018], Buesing et al. [2018], and Karl et al. [2016], which likewise utilize a learned latent state withing which to learn the model dynamics. These models can all be thought of as sequential VAEs, where we jointly learn to embed and decode latent states, along side learning the transition dynamics between states. An important finding in all of these papers, is that the joint inclusion of both a stochastic and deterministic component in the model are crucial for successful planning.

While the approaches of Hafner et al. [2018] and Ha and Schmidhuber [2018] are similar in approach, we focus on the work done in the PlaNet paper of Hafner et al. [2018]. There are two important distinctions to be made between theses papers. Firstly, Ha and Schmidhuber [2018] utilizes a genetic algorithm to learn

a linear controller which takes latent state as input, and predicts the optimal action. Contrary to this, Hafner et al. [2018] utilizes a sampling based approach known as the cross entropy method (CEM Rubinstein [1997]). CEM is a population based optimization algorithm which iteratively learns a distribution over action sequences to maximize a given objective function. Secondly, and likely more importantly, Ha and Schmidhuber [2018] decomposes the model into VAE and RNN component which are learned separately. This makes the model conceptually simpler and easier to implement, but limits the ability of the model to effectively learn from actively collected data. PlaNet on the other hand uses a single model where the encoder, decoder, transition function, and reward models are all learned jointly. While this model is able to continue to learn in an online manner, it is conceptually more complex.

The remainder of our paper will provide a brief overview of both citeplanet and Ha and Schmidhuber [2018], present the software and modeling approaches that we took in reproducing the original papers results, and finally go over the results of this reconstruction. We conclude with a discussion of the results obtained in the replication process.

II. Summary of the Original Paper

A. Methodology of World Models

The original paper Hafner et al. [2018] bears many resemblance to the work done in Ha and Schmidhuber [2018]. In Ha and Schmidhuber [2018], the authors improve the state of the art for model based reinforcement learning by utilizing a VAE to encode high-dimensional pixel images into a lower dimensional latent space. With this learned latent space, we can then learn the transition dynamics of the environment. The primary benefit in this case is the dimensionality reduction of the sequence modelling problem. Now rather than learning a complex non-linear map between pixel images, we can learn a much lower dimensional map between latent state. The second important contribution is the use of the RNN-Gaussian Mixture Model to include stochasticity into the sequence modelling problem. It is very rare that the environment an agent is acting within to be purely deterministic, so modelling the environment with only an RNN, which is deterministic would be an inn appropriate choice. By combining these two ideas, Ha and Schmidhuber [2018] was able to learn a controller purely in 'imagination' or purly by generating training data with the learned model.

To train the model, Ha and Schmidhuber [2018] start by first training the VAE separately. Then, using sequences of images, we can create sequences of latent states. This allows us to learn the the state space model by simply training on these sequences directly. A linear controller is then trained purely within the simulated environment of the dynamics model. This model is linear in the *latent* space, not the observation space, and is therefore at most the size of the lower dimensional space. This is another point of interest, as the models used in reinforcement learning are frequently very large convolutional or feedforward networks.

B. Methodology of Planning From Pixels

Hafner et al. [2018] takes an approach very similar to that of Ha and Schmidhuber [2018] in that they use an VAE to learn the latent space representation as well as a state space model which predicts future latent states and rewards. However, there are several important differences, which we will discuss further below.

Learning Latent State Dynamics The modelling choices in in Hafner et al. [2018] are fairly similar to that of Ha and Schmidhuber [2018] . Firstly, purely deterministic (RNN Model) and purely stochastic (SSM Model) models are considered. The Stochastic model is nearly identical to the Gaussian Mixture Model used in Ha and Schmidhuber [2018], but does not utilize a mixture distribution. Likewise, the deterministic RNN model would simply be the independent RNN of Ha and Schmidhuber [2018]. Finally, a part-stochastic, part-deterministic latent variable (RSSM Model) is proposed. Again, this mirrors the RNN-GMM of the Ha and Schmidhuber [2018] paper. What is clear here is that both papers utilize similar model design features, but what is unclear is why there is any difference between the two works. This is primarily revealed when we consider the method in which the models are trained.

As mentioned above, Ha and Schmidhuber [2018] trains the VAE, RNN-GMM, and then the model all independently and in succession. This simplifies the training methodology for each step of the model, but also means that *gradients* are not shared between them. This is important, because features that the VAE learn, may be independent of the features that are important for predicting the following state, or for performing planning. Consider the case where a majority of the image is noise, and a small portion of the image contains information about the distribution of next states and rewards. When the VAE is trained on its own, the model can safely ignore this information while still minimizing

the reconstruction error. However, by backpropgating errors from the state transitions and reward predictions, we will be more likely to encode this information into the latent space. The primary idea, is that the latent space learned by independently by the VAE may not be usefull for the downstream prediction tasks. This is the primary point of divergence between Ha and Schmidhuber [2018] and Hafner et al. [2018] - Hafner et al. [2018] learns the entire model at once, rather than component by component.

Training these models end-to-end is a non trivial task, as we can no longer naively use the encoded images as direct training targets with which to perform log-liklihood maximization. Since the entire model is learned at once, the prediction error will actually be backpropogated back through both the encoder, as well as the transition model, so both the targets and predictions will change as the model continues to learn. We describe the loss formulation in more detail, but the main idea is to consider the transition model as predicting a *prior* distribution over the true latent state, while the encoder is providing the *posterior* distribution over states. We can then use the language and concepts of VAEs to formulate our training objective.

During training, a sequence of observations and actions and rewards are fed to the transition model. The observations are fed through a fully connected layer An overview of the model is shown in Figure 1

Cross Entropy Method: Actions are selected using the cross entropy method (CEM). CEM works by starting with random candidate actions over a simulation horizon. With access to the environment dynamics, each candidate action sequence can be rolled out over the horizon to determine the resulting states and rewards. The top K candidate actions are used to calculate the mean and variance to sample actions from for the next iteration. This process repeats until the final mean actions are returned. This is an iterative solution to quickly finding the best actions to use. The dynamics used to roll out actions can come from the true simulator or a model that estimates future states and rewards.

Loss Function: The models are trained to maximize the data log-likelihood, and the loss function ends up being composed very similarly to that of a standard VAE. We will focus on the loss with respect to observations here, as the reward loss is limited to a simple squared error loss term. We start with the data log-likelihood:

$$\begin{aligned}\mathcal{L}(o_{1:T}|a_{1|T}) &= \ln p(o_{1:T}|a_{1|T}) \\ &= \ln \int \prod_t p(o_t|s_t)p(s_t|s_{t-1}a_{t-1})\partial_{s:_{1:T}}\end{aligned}$$

The second equality follows from the assumption that the distribution of observations O_t are conditional on some unobserved latent state s_t, and that the probability of observing s_t depends only on the prior state and action, s_{t-1} and a_{t-1}. This is very close to the first step of the VAE approach, but we have introduced a temporal dependency, as well as the action variables. we can the create a variational lower bound using Jensen's inequality, again similar to the VAE approach (see the papers appendix for details)

$$\begin{aligned}&\ln \int \prod_t p(o_t|s_t)p(s_t|s_{t-1}a_{t-1})\partial_{s:_{1:T}} \geq \\ &\sum_{t=1}^{T}(\underbrace{\mathbf{E}_{q(s_t|o_{\leq t},a_{\leq t})}[\ln p(o_t|s_t)]}_{\text{reconstruction loss}} - \\ &\qquad \underbrace{\mathbf{E}_{q(s_{t-1}|o_{\leq t-1},a_{\leq t-1})}[q(s_t|o_{\leq t},a_{\leq t} \parallel p(s_t|s_{t-1},a_{t-1}))}_{\text{complexity}}\end{aligned}$$

In the above, $q(s_t|o_{\leq t}, a_{\leq t}) = \prod_1^t q(s_t|s_{t-1}, o_t, a_t)$ represents our encoder, and allows us to approximate the state posterior from the prior states, observations, and actions. Further, the $p(o_t|s_t)$ represents our decoder, and the loss term translates to the square error between our reconstructed image and the truly observed image at time step t. Finally, the $p(s_t|a_{t-1}, s_{t-1})$ represents the transition model, and so we have a KL-divergence term to measure the ability of our model to truly estimate the learned posterior. Similar to a VAE, we could also add a structural prior on the distribution $q(s_t|o_{\leq t}, a_{\leq t})$, by adding a further KL loss term of form $\mathbf{E}_{q(s_{t-1}|o_{\leq t-1},a_{\leq t-1})}[q(s_t|o_{\leq t}, a_{\leq t} \parallel \mathcal{N}(0, 1))$, which pushes the learned distribution to be close to the unit normal distribution.

A detailed diagram of how the models are structured and where the loss terms occur is provided in diagram 7.

1) Latent Overshooting: A limitation of the above learning objective is that the KL-loss term only ever teaches us to accurately predict one-step transitions - the gradient flows through $p(s_t|s_{t-1}, a_{t-1})$ directly into $q(s_{t-1})$ but does not get backpropogated back through a chain of multiple transitions, $p(s_t|s_{t-1}, a_{t-1})$. The prior variational bound can be generalized to handle latent

overshooting, which trains all multi-step predictions in latent space. The inclusion of this loss term also turns out to critical in the performance of the SSM model, however did not show any positive impact on the full RSSM model.

We first need to define a multi-step prediction, which is the crux of our final lower bound:

$$p(s_t|s_{t-d} = \int \prod_{\tau=t-d}^{t} p(s_\tau|s_{\tau-1})\partial_{t-d+1:t-1}$$
$$= \mathbf{E}_{p(s_{t-1}|s_{t-d})}[p(s_t|s_{t-1})]$$

In the case of $d = 1$, then this is the same as the single step prediction case. With this value, we can derive a lower bound that is almost identical to that of the previous section,

$$\ln \int \prod_t p(o_t|s_t)p(s_t|s_{t-d}a_{t-d:d})\partial_{s_{:1:T}} \geq$$
$$\sum_{t=1}^{T}(\underbrace{\mathbf{E}_{q(s_t|o_{\leq t},a_{\leq t})}[\ln p(o_t|s_t)]}_{\text{reconstruction loss}} -$$
$$\underbrace{\mathop{\mathbf{E}}_{\substack{p(s_{t-1}|s_{t-d})*\\ q(s_{t-d}|o_{\leq t-d},a_{\leq t-d})}} [q(s_t|o_{\leq t}, a_{\leq t} \parallel p(s_t|s_{t-1}, a_{t-1}))}_{\text{complexity}}$$

is likewise almost identical. The actual loss term computed will be the mean for all values 1 to the chosen horizon factor D, and can also contain a β_d weighting factor so that we can potentially prioritize the loss at a chosen time horizon (potentially the time horizon of our MPC). This weighting is analogous to what is done in the $\beta-$VAE Model Higgins et al. [2017]. An important implementation detail for the overshooting loss is that you must stop the gradients of the target posterior distribution. This is so that the priors are trained towards the informed posteriors, but the posteriors are not trained towards the multi-step predictions.

The complete final loss function would then be as follows:

$$\mathcal{L} = \mathcal{L}_{MSE}(o_t, \hat{o}_t) + \mathcal{L}_{KL}(s_t^{prior}, s_t^{post}) + \mathcal{L}_{MSE}(r_t, \hat{r}_t)$$
$$+ \frac{1}{D}\sum_{d=1}^{D} \mathcal{L}_{OS-MSE}(o_t, \mathbf{E}[o_t|s_{t-d}^{post}])$$
$$+ \frac{1}{D}\sum_{d=1}^{D} \beta_d \mathcal{L}_{OS-KL}(\mathbf{E}[s_t^{prior}|s_{t-d}^{post}], \ulcorner s^{post} \urcorner)$$

where $o_t, \hat{o}_t$ are the true and predicted pixel observations, s_{prior}, s_{post} are the predicted prior and posterior state distributions, $r_t, \hat{r}_t$ are the true and predicted rewards, $\ulcorner \cdot \urcorner$ denotes the stop-gradient operator, and $\mathbf{E}[\cdot|s_{t-d}^{post}]$ is the model based expectation of the state of observation given an initial posterior observation s_{t-d}^{post} and the sequence of actions $a_{t-d:t}$. The computation graph along with loss terms is visualized included in figure 7.

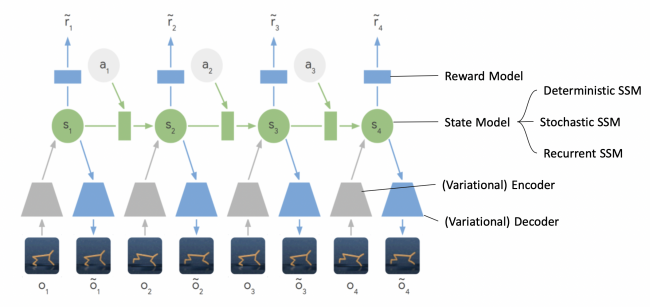


Figure 1: An overview of the transition model from Ha and Schmidhuber [2018]

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 θ 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)
    $r_t, o_{t+1} \leftarrow \sum_{k=1}^{R} r_t^k, o_{t+1}^R$
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{(o_t, a_t, r_t)_{t=1}^{T}\}$

Figure 2: An overview of the training loop from Ha and Schmidhuber [2018]

C. Key Results of the Original Paper

A key result of the original paper is that they were able to achieve high scores on a variety of simulated environments while trained on a fraction of the data compared to previous approaches. Furthermore, they

achieved similar performance compared to having access to the true simulator. A summary of the RSSM performance is shown in Figure **??**. Another key result is that the original authors explored training agents that could perform well on a variety of tasks.

III. Methodology of this project

A. Objectives and Technical Challenges

In this project we aim to implement the results from Hafner et al. [2018]. The main challenges involved in reproducing these results comes from the intertwined nature of this solution. All aspects of the solution (Cross entropy method, state space model, encoder, decoder, loss functions, data processing) are trained and evaluated at the same time. This proves challenging as if there are problems with any single component in this system, we will not be able to reproduce the results. Another technical implementation challenge is dealing with the duel modes required by the transition models. During training, the models accept observations for training. But there is also a generative process where the models need to predict forward in latent space without observations.

B. Key Differences between the original paper and our work

Due to the availability of open source simulators we are able to reproduce the problem nearly identically to the original offers. Some key differences are that we lack time and compute so we generally are testing performance after 100-200 episodes where the original authors discussed performance after 1000 episodes. Furthermore, we did not replicate their experiments for multi task learning.

IV. Implementation

While the original paper has made their repo available online, it is written using TensorFlow. We aim to reproduce key elements of the performance using PyTorch and generally creating our repo from scratch.

A. Simulator / Data

We used OpenAI Gym mujoco and Deepmind Control Suite (Tunyasuvunakool et al. [2020]) for the agent environments. All data is collected in an online manner and is aggregrated in a variable size *memory buffer*. We can then draw random samples of length 50 sequences from the buffer with which to then train our models. We follow the original authors implementation to downsample the environment images to (3,64,64) tensors.

B. Software Design

In this section we give an overview of the software design.

One benefit of this approach is that we don't need to work with a large dataset, we sample from the environment and train all models at the same time. While this means we don't need to hold on to a large dataset, it also makes passing data around to all the right modules a complicated task. As such, it was very important that we approached this problem with a modular design.

First, we give a brief overview of the training loop flow to motivate some architecture choices:

1) a CPU based environment simulator is initialized for the desired game (cartpole-balance / walker-walk)
2) We initialize a transition model, that takes in pixel observations along with actions and predicts future states and rewards
3) We sample a small number of seed episodes while the agent takes random actions and we store these in an experience buffer of (Observation, reward, done). The observations are pixels from the game.
4) We randomly sample experience and train our model from some iterations.
5) Another module is used to choose a best action, given a current observation and a dynamics model that predict what rewards and final states we would get for specific actions. MPC can use the true simulator for the environment, or our learned simulator (the transition model)
6) We test the performance of our transition model + MPC by rolling out test episodes in the environment. Our MPC + transition model chooses the best action and we observe the real rewards we get from the environment

Main Training Loop: src/main.py: This is the entry point for training. Here we load the config information, instantiate all the other modules, run the training loop, test our models, and report metrics. An overview of the planning algorithm from Hafner et al. [2018] is shown in Figure 2

Environment Module: src/env.py: In our repo we interact with two different types of reinforcement learning agent simulators. We wrap these with a general environment class to ensure that we have the same interface and specific calls are available. Furthermore, we add useful pre-processing functions like reducing the size of images from the environments.

Config: src/Config/: All config information is stored here including: the structure of the pytorch models, hyperparameters for training/checkpointing / saving roll-outs.

Transition Model: src/Models: As described above, we consider 3 different variations on the transition model. All of these models derive from the same base class which implements the following shared methods: encode, observation_to_state_belief as well as the loss functions. Building on top of this, each of the inherited classes must implement a forward method.

Dynamics: src/dynamics.py: This file contains the Model Predictive control class for predicting future rewards and states given real or simulated dynamics, which are also stored in this file. The dynamics classes are mostly interface classes to provide a common functionality for MPC to call. The TrueDynamics function simply calls the real simulator. The Learned dynamics function calls the transition model.

Utilities: src/utils.py: Other general utily functions are kept here.

V. Experiment Overview

Our goals are as follows:

1) Reproduce the agents performance with respect to rewards achieved
2) Examine the performance of the VAE
3) Compare the affect of using the RSSM or the SSM
4) Understand the sensitivity to hyper parameter selection in reproducing results

VI. Results

In this section we discuss key results from our experiments.

A. Training Performance

It can be seen in Figure 3 below that we are able to train the RSSM and SSM models. Throughout training, we see a steady decrease in observation loss as our VAE learns to successfully encode and decode images. Furthermore, the reward loss steadily decreases as our model continues to improve it's ability to predict rewards. It can be seen that the RSSM model is able to achieve better reward and observation loss compared to the SSM model. Lastly, the KL loss can be seen to increase as the other two losses decrease. This makes sense as the KL loss represents the difference between our posterior latent state the considers the observation and the prior latent state that doesn't. The posterior prediction is trained using the true images as it's reference. However, the prior state is trained by tracking close to the posterior state. So, as the posterior state get's pulled towards the true latent state, the difference between the posterior and the prior (and the KL loss term) will increase slightly until the states converge again.

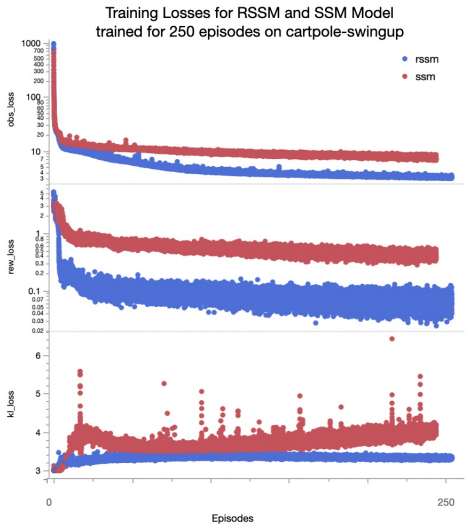


Figure 3: The observation loss, KL loss and reward loss for the SSM model (red) and RSSM model (blue) after training on the game cartpole-swingup for 250 episodes. The observation loss (top) and the reward loss (middle) are plotted using a log axis to more clearly show the decrease in value.

B. VAE Performance

Reconstructed images are provided below for the RSSM and SSM models after 50 episodes of training. The agents have clearly learned to reproduce the image background to a high degree of detail. Plus, the agents have also reproduced the more important aspect of the position of the pole. It is difficult to compare the RSSM and SSM reconstructions frame by frame as the position and speed of the poles are different. We can see from the loss functions that the RSSM model is able to achieve lower loss in all categories.

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
CEM + true simulator state	simulator state	0	850	964	656	825	993	994
PlaNet (theirs)	pixels	1000	821	832	662	700	930	951
PlaNet (theirs)	pixels	200	∼600	∼0-400	∼400	∼700	∼800	∼400
PlaNet (ours)	**pixels**	**200**						

Figure 4: Original (left) and reconstructed image for the RSSM model after being encoded / decoded in latent space

Figure 5: Original (left) and reconstructed image for the SSM model after being encoded / decoded in latent space

C. Reward Performance

We were able to successfully reproduce the results of the original paper when considering a smaller number of training episodes. A video our our agent learning to play cartpole swing up is available here. In this video we show to observed frames on the left as well as the reconstructed images using the agent's latent state. In this video it can be seen that in early episode the agent starts to learn that it can achieve higher rewards by moving around sporadically as the pole randomly swings up, giving a higher reward. As the agent trains on more and more episodes it learns to eventually swing up then balance the pole.

D. Comparison of Results with Original Paper

We show a comparison below in figure

The results for training with the complete RSSM model are shown below. We were able to replicate the

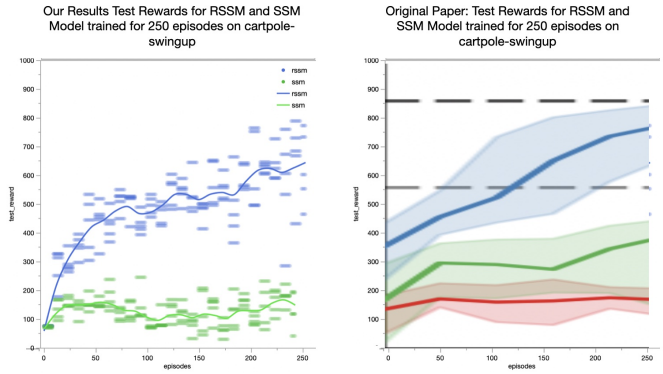


Figure 6: Left: A comparison of SSM and RSSM rewards in cartpole-swingup after 250 episodes. Right: The original papers rewards for cartpole swingup and 250 episodes. The colors for RSSM and SSM line up between the two plots. We did not run the RNN model in green

results of the original paper for performance at the 200 episode mark.

E. Discussion of Insights Gained

As was described in the original paper, we were able to see the benefit of using a hybrid Recurrent Stochastic State Space Model. When used in parallel with an action finding algorithm like the cross entropy method, our model is able to learn and then make decisions based on uncertainty in the environment. Because of this, agents were able to make better decisions under the hybrid RSSM model

Next, we found that training all aspects of the model at the same time can prove troublesome when there are issues. Specifically testing our MPC implementation at the same time as the our transition model made diagnosing training problems hard. We run into a "Chicken and the Egg" scenario when we don't know what is causing the issue. Is our model bad because our MPC is choosing bad actions and we aren't exploring the state space well?

Or is our MPC not choosing good actions because our transition model isn't working? We anticipated problems in this area and actually tested the MPC using the true dynamics simulator first. However, we weren't able to determine whether the MPC was working as the true environment simulator is CPU based and is actually many times slower that our learned GPU based transition model. In the future if we had more compute or the opportunity to use a better simulator we could mitigate this issue.

Lastly, as is the case with many reinforcement learning problems we found the selection of hyperparameters to be critical in reproducing results. While we were able to eventually reproduce the results of the paper, we found that even slightly modifying aspects like the learning rate significantly affect the results.

VII. Conclusion and Future Work

In this report we were able to reproduce key aspects of the original paper. While we were able to run experiments on a variety of environments, we were not able to include these partial results in this report. For future work, we will finish comparing our results with the original paper for the remaining tasks. We also would like to implement this in a real world setting. As these models can learn from pixels, there are many possible learning environments where we can make use of just a camera. This would give insight into how well this reinforcement learning algorithm may transfer to the real world.

VIII. Contributions

The approximate contributions by each member are shown below.

References

Kai Arulkumaran, Antoine Cully, and Julian Togelius. Alphastar: An evolutionary computation perspective, 2019. URL http://arxiv.org/abs/1902.01724.

Lars Buesing, Theophane Weber, Sebastien Racaniere, S. M. Ali Eslami, Danilo Rezende, David P. Reichert, Fabio Viola, Frederic Besse, Karol Gregor, Demis Hassabis, and Daan Wierstra. Learning and querying fast generative models for reinforcement learning, 2018. URL https://arxiv.org/abs/1802.03006.

Andreas Doerr, Christian Daniel, Martin Schiegg, Duy Nguyen-Tuong, Stefan Schaal, Marc Toussaint, and Sebastian Trimpe. Probabilistic recurrent state-space models, 2018. URL https://arxiv.org/abs/1801.10395.

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

Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels, 2018. URL https://arxiv.org/abs/1811.04551.

Irina Higgins, Loïc Matthey, Arka Pal, Christopher P. Burgess, Xavier Glorot, Matthew M. Botvinick, Shakir Mohamed, and Alexander Lerchner. beta-vae: Learning basic visual concepts with a constrained variational framework. In *ICLR*, 2017.

Maximilian Karl, Maximilian Soelch, Justin Bayer, and Patrick van der Smagt. Deep variational bayes filters: Unsupervised learning of state space models from raw data, 2016. URL https://arxiv.org/abs/1605.06432.

OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving rubik's cube with a robot hand, 2019. URL https://arxiv.org/abs/1910.07113.

Reuven Y. Rubinstein. Optimization of computer simulation models with rare events. *European Journal of Operational Research*, 99(1): 89–112, 1997. ISSN 0377-2217. doi: https://doi.org/10.1016/S0377-2217(96)00385-2. URL https://www.sciencedirect.com/science/article/pii/S0377221796003852.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016. ISSN 0028-0836. doi: 10.1038/nature16961.

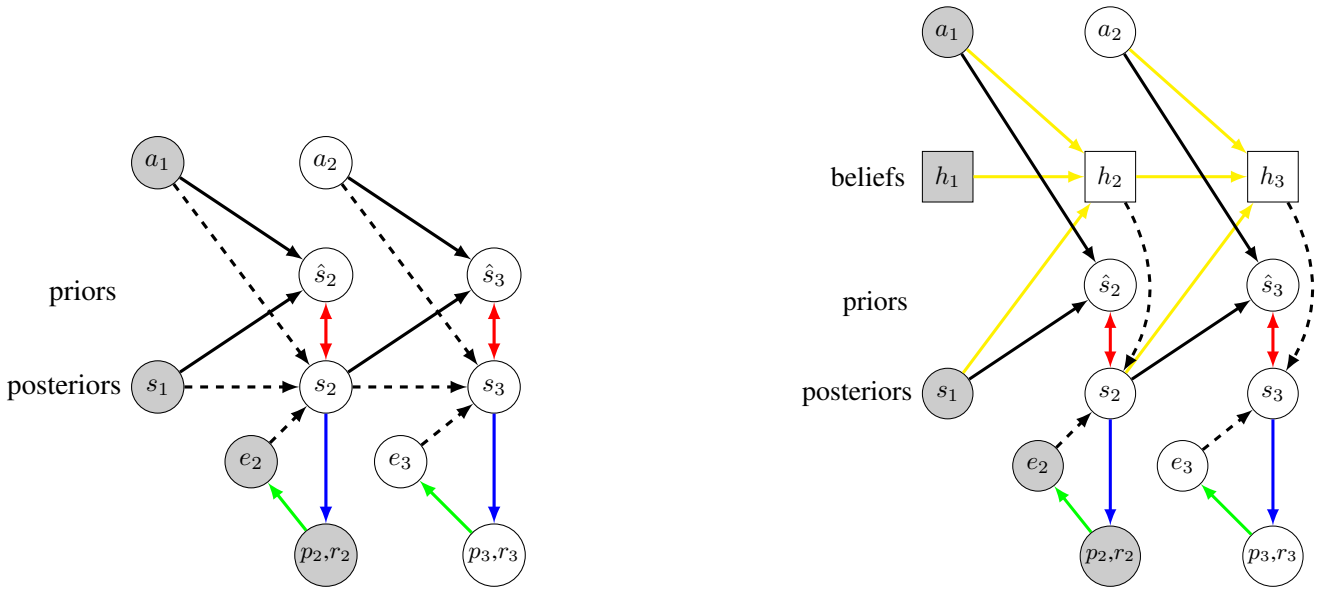
David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nat.*, 550(7676):354–359, 2017. URL http://dblp.uni-trier.de/db/journals/nature/nature550.html#SilverSSAHGHBLB17.

Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, Nicolas Heess, and Yuval Tassa. dm$_{c}ontrol$: $Softwareandtasksforcontinuouscontrol$.*Software Impacts*, 6 : $100022, 2020$. $ISSN2665 - 9638$. doi : . URL https://www.sciencedirect.com/science/article/pii/S2665963820300099.

Contribution

Contribution Table

	main	utils	models	dynamics	env	config
Sam	50%	50%	90%	10%	50%	50%
Trevor	50%	50%	10%	90%	50%	50%



(a) Stochastic model (SSM)

(b) Recurrent state-space model (RSSM)

Figure 7: Latent dynamics model designs. We are training the model to observe the first two time steps and predict the third. Circles represent stochastic variables and squares deterministic variables. Dashed Lines denote inference and represent the posterior computation. Solid lines denote the generative process and compute the latent state priors. Green lines represent the encoder, and allow us to pass from pixel level images to the embedded layer of the Auto-Encoder. The red lines denote a KL Divergence loss term computed between the priors and the posterior. The blue lines represent the observations model, and is the decoder of the Auto-Encoder. The blue lines therefore represent the reconstruction loss. Finally, the yellow lines denote the forward pass through our RNN. Note that when we are training the model, the initial state and belief (hidden state) are given as vectors of zeros.