Goal-Aware Prediction:
Learning to Model What Matters
Supplementary Material

Suraj Nair†, Silvio Savarese†, Chelsea Finn†
†Stanford University
Email: surajn@stanford.edu
sites.google.com/stanford.edu/gap

A. Related Work

Recent years have seen impressive results from reinforcement learning [58] applied to challenging problems such as video games [41, 46], Go [56], and robotics [38, 47, 34]. However, the dependence on large quantities of labeled data can limit the applicability of these methods in the real world. One approach is to leverage self-supervision, where an agent only uses data that it can collect autonomously.

Self-Supervised Reinforcement Learning: Self-supervised reinforcement learning explores how RL can leverage data which the agent can collect autonomously to learn meaningful behaviors, without dependence on task specific reward labels, with promising results on tasks such as robotic grasping and object re-positioning [50, 16, 68]. One approach to self-supervised RL has been combining goal-conditioned policy learning [33, 54, 9] with goal re-labeling [4] or sampling goals [44, 43]. While there are numerous ways to leverage self-supervised data, ranging from learning distance metrics [67, 29], generative models over the state space [35, 18, 17, 39, 45], and representations [60], one of the most heavily utilized techniques is learning the dynamics of the environment [64, 20, 1].

Model-Based Reinforcement Learning: Learning a model of the dynamics of the environment and using it to complete tasks has been a well studied approach to solving reinforcement learning problems, either through planning with the model [11, 64, 40, 6, 8, 2, 28, 42] or optimizing a policy in the model [51, 25, 70, 37, 32, 63, 27, 24, 17]. Numerous works have explored how these methods might leverage deep neural networks to extend to high dimensional problem settings, such as images. One technique has been to learn large video prediction models [20, 5, 15, 16, 49, 35, 61, 65], however model under-fitting remains an issue for these approaches [10]. Similarly, many works have explored learning low dimensional latent representations of high dimensional states [64, 14, 69, 28, 35, 31, 62, 37, 23] and learning the dynamics in the latent space. Unlike these works, we aim to make the problem easier by encouraging the network to predict only task-relevant quantities, while also changing the objective, and hence the distribution of prediction errors, in a task-driven way. This allows the prediction problem to be more directly connected to the downstream use-case of task-driven planning.

Addressing Model Errors: Other works have also studied the problem of model error and exploitation. Approaches such as ensembles [8, 59] have been leveraged to measure uncertainty in model predictions. Similarly, Janner et al. [32] explore only leveraging the learned model over finite horizons where it has accurate predictions and Levine et al. [38] use local models. Exploration techniques can also be used to collect more data where the model is uncertain [48].

Most similar to our proposed approach are techniques which explicitly change the models objective to optimize for performance on downstream tasks. [55, 30] explore only predicting future reward to learn a latent space in which they learn dynamics, Freeman et al. [22] learn a model with the objective of having a policy achieve high reward from training in it, and Amos et al. [3]. Srinivas et al. [57] embed a model/planner inside a neural network. Similarly, Farahmand et al. [19], D’Oro et al. [13] explore how model training can be re-weighted using value functions or policy gradients to emphasize task specific performance. Unlike these works, which depend heavily on task-specific supervision, our proposed approach can be learned on purely self-supervised data, and generalize to unseen tasks.

B. Method Implementation Details

In this section we go over implementation details for our method as well as our comparisons.

1) Implementing Goal-Aware Prediction

We implement GAP with a latent dynamics model. Given a dataset of trajectories \([\tau_1, ..., \tau_N]\), we sample sequences of states \([(s_1, a_1), ..., (s_T)]\) where we re-label goal for the trajectory as \(s_g = s_T\).

The GAP model consists of three components, (1) an encoder \(f_{enc}(z_t | s_t, s_g; \theta_{enc})\) that encodes the state \(s_t\) and goal \(s_g\) into a latent space \(z_t\), (2) a decoder \(f_{dec}(s_g - s_t | z_t; \theta_{dec})\) that decodes samples from the latent distribution into \(s_g - s_t\), and (3) a forward dynamics model in the latent space \(f_{dyn}(z_{t+1} | z_t, a_t; \theta_{dyn})\) which learns to predict the future latent distribution over \(z_{t+1}\) from \(z_t\) and action \(a_t\). In our experiments we work in the setting where states are images, so \(f_{enc}(z_t | s_t, s_g)\) and \(f_{dec}(s_g - s_t | z_t)\) are convolutional neural networks, and \(f_{dyn}(z_{t+1} | z_t, a_t)\) is a fully-connected network. The full set of parameters \(\theta = \{\theta_{enc}, \theta_{dec}, \theta_{dyn}\}\)
are jointly optimized. Exact architecture and training details for all models can be found in the supplement. Following prior works [21 29 9], we train for multi-step prediction. More specifically, given $s_t, a_{t:t+H}, s_g$, the model is trained to reconstruct $(s_g - s_t), \ldots, (s_g - s_{t+H})$.

**Data Collection and Model Training:** In our self-supervised setting, data collection simply corresponds to rolling out a random exploration policy in the environment. Specifically, we sample uniformly from the agent’s action space, and collect 2000 episodes, each of length 50, for a total of 100,000 frames of data.

During training, sub-trajectories of length 30 time steps are sampled from the data set, with the last timestep labeled as the goal $s_g = s_{30}$. Depending on the current value of $H$, loss is computed over $H$ step predictions starting from states $s_{t:t+H}$. We use a curriculum when training all models, where $H$ starts at 0, and is incremented by 1 every 50,000 training iterations. All models are trained to convergence, for about 300,000 iterations on the same dataset.

**Planning with GAP:** For all trained models, when given a new goal at test time $s_g$, we plan using model predictive control (MPC) in the latent space of the model. Specifically, both the current state $s_t$ and $s_g$ are encoded into their respective latent spaces $z_t$ and $z_g$ (Algorithm 1, Line 3).

**Algorithm 1 Latent MPC ($f_{enc}, f_{dyn}, s_t, s_g$)**

1: Let $D = 1000$, $D^t = 10$, $H = 15$
2: Receive current state $s_t$ and goal state $s_g$
3: Encode $z_t = f_{enc}(s_t, s_g)$, $z_g = f_{enc}(s_g, s_g)$
4: Initialize $N(\mu, \sigma^2) = N(0, 1)$
5: Let cost function $C(z_t, z_g) = ||z_t - z_g||^2$
6: while iterations $\leq 3$
7: $a^1_t, \ldots, a^D_t = \text{argmin}_a(C(z_t, a_t; z_g))$
8: $s_{t+1}^H = f_{dyn}(z_t, a^1_{t:H}, \ldots, a^D_{t:H})$
9: $z_t = \text{encoder}(z_{t+1}^H, z_g)$
10: $a_{t+1} = \text{argmin}_a(C(z_{t+1}^H, z_g))$
11: Refit $\mu, \sigma^2$ to $a_{t+1}$
12: end while
13: Return $a_{t+1}$

Then using the model $f_{dyn}(z_{t+1}|z_t, a_t)$, the agent plans a sequence of $H$ actions to minimize cost $\sum_{h=0}^H ||z_{g-h} - z_{t+h}||^2$ (Algorithm 1, Lines 4-11). Following prior works [20 28], we use the cross-entropy method [52] as the planning optimizer. Finally, the best sequence of actions is returned and executed in the environment (Algorithm 1, Line 13).

While executing the plan, our model re-plans every $H$ timesteps. That is, it starts at state $s_t$, uses Latent MPC (Algorithm 1) to first plan a sequence of $H$ actions, executes them in the environment resulting in a state $s_{t+H}$, then replans an additional $H$ actions, and executes them resulting in a final state $s_T$. Success is computed based the difference between $s_T$ and $s_g$.

2) Architecture Details

**Block/Door Domain:** All comparisons leverage a nearly identical architecture, and are trained on an Nvidia 2080 RTX. In the block pushing domain input observations are [64,64, 6] in the case of our model (GAP), as well as the ablations, and [64,64, 3] in the case of Standard.

All use an encoder $f_{enc}$ with convolutional layers (channels, kernel size, stride): [(32, 4, 2), (32, 4, 1), (64, 4, 2), (64, 3, 1), (128, 4, 2), (128, 3, 1), (256, 4, 2), (256, 3, 1)] followed by fully connected layers of size $[512, 2 \times L]$ where $L$ is the size of the latent space (mean and variance). All layers except the final are followed by ReLU activation.

The decoder $f_{dec}$ takes a sample from the latent space of size $L$, then is fed through fully connected layers [128, 128, 128], followed by de-convolutional layers (channels, kernel size, stride): [(128, 5, 2), (64, 5, 2), (32, 6, 2), (3, 6, 2)]. All layers except the final are followed by ReLU activation, except the last layer which is a Sigmoid in the case of Standard, and GAP (-Residual).

For all models the dynamics model $f_{dyn}$ are a fully connected network with layers [128, 128, 128, L], followed by ReLU activation except the final layer.

The inverse model baseline utilizes the same $f_{enc}$ and $f_{dyn}$ as above, but $f_{dec}$ is instead a fully connected network of size [128, 128, action size] where action size is 4 (corresponding to delta x,y, z motion and gripper control). All layers except the final are followed by ReLU activation.

Lastly, the RIG [44] baseline uses a VAE with identical $f_{enc}$ and $f_{dec}$ to the standard approach above, except learns a policy in the latent space. The policy architecture used is the default SAC [26] from RLkit, namely 2 layer MLPs of size 256.

**SVG+GAP:** In all SVG [12] based experiments on real robot data, the architecture used is identical to the SVG architecture as described in official repo [12] with the VGG encoder/decoder. All BAIR dataset experiments take as input sequences of 2 frames and predict 10 frames, while all RoboNet experiments take as input 2 frames and predict 20 frames. The latent dimension is 64, and the encoder output dimension is 128. All models are trained with batch size 32.

3) Training Details

**Block/Door Domain:** We collect a dataset of 2,000 trajectories, each 50 timesteps with a random policy. All models are trained on this dataset to convergence for roughly 300,000 iterations. All models are trained with learning rate of 1e-4, and batch size 32.

The RIG baseline is trained using the default SAC parameters in RLkit, for an additional 3 million steps.

**BAIR Robot Dataset:** We train on the BAIR Robot Dataset [15] as done in the original SVG paper, except with action conditioning.

**RoboNet:** We train on the subset of the RoboNet dataset which considers only the Sawyer arm and the front facing camera view, and use a random 80/20 train test split.

4) Task/Evaluation Details

**Tasks.** All tasks are defined in a Mujoco simulation built on top of the Meta-World environment [66]. In Task 1, the agent must push a single block to a target position, as specified by a

https://github.com/edenton/svg
goal image. The task involves either pushing the pink, green, or blue block. Success if defined as being within 0.08 of the goal position. In Task 2 the agent must push 2 blocks, specifically the green and blue block to their respective goal positions, again indicated by a goal image. Success is determined as being within \( \pi/6 \) radians of the goal angle.

**Evaluation.** During all control experiments, evaluation is done using model predictive control with the latent space dynamics model. Specifically, we do latent MPC as described in Algorithm 1, specifically by planning 15 actions, executing them in the environment, then planning 15 more actions and executing them. Each stage of planning uses the cross entropy method, specifically sampling 1000 action sequences, sorting them by the mean latent distance cost to the goal, refitting to the top 10, and repeating 3 times, before selecting the total lowest cost action.

5) **Experimental Comparisons**

**Comparisons:** We compare to several model variants in our experiments. GAP is our approach of learning dynamics in a latent space conditioned on the current state and goal, and reconstructing the residual between the current state and goal latent space conditioned on the current state and goal. GAP (-Goal Cond) is an ablation of GAP that does not use goal conditioning. Instead of conditioning on the goal and predicting the residual to the goal, it is conditioned on the initial state, and predicts the residual to the initial state. This is similar to prior work on goal conditioned video prediction [55]. Standard refers to a standard latent dynamics model, representative of approaches such as PlaNet [28], but without reward prediction since we are in the self-supervised setting.

When studying task performance, we also compare to two alternative self-supervised reinforcement learning approaches. First, we compare to an Inverse Model, which is a latent dynamics model where the latent space is learned via an action prediction loss (instead of image reconstruction), as done in Pathak et al. [48]. Second, we compare to a model-free approach: reinforcement learning with imagined goals (RIG) [44], where we train a VAE on the same pre-collected dataset as the other models, then train a policy in the latent space of the VAE to reach goals sampled from the VAE. Further implementation details can be found in the supplement.

C. **Additional Results**

1) **Theorem 3.1 Proof**

**Proof:** For the specified policy, violating \( \epsilon \)-optimality will only occur if the cost of the best action sequence \( c^*_1:T \) is overestimated or if the cost of a sub-optimal action sequence \( (i | c^*_i > c^*_i + \epsilon) \) is underestimated. Thus, let us define the "worst case" cost predictions as the ones for which \( c^*_i \) is most overestimated and \( c^*_i \) \( \forall i \) \( c^*_i > c^*_i + \epsilon \) are most underestimated (while still satisfying Equations 2 and 3). Concretely we write the worst case cost estimates as

\[
\tilde{c}_i := \min \hat{c}_i \quad \forall i \mid c^*_i > c^*_i + \epsilon
\]

\[
\tilde{c}_i := \max \hat{c}_i
\]

s.t. Eq. 2 and 3 hold. We will now show that \( \tilde{c}_i < \hat{c}_i \quad \forall i \mid c^*_i > c^*_i + \epsilon \). First, since \( \hat{c}_i \) satisfies Eq. 3, we have that

\[
\tilde{c}_i > c^*_i - (c^*_i - c^*_i) + \epsilon
\]

Similarly, since \( \hat{c}_i \) satisfies Eq. 2, we have that

\[
\hat{c}_i < c^*_i + \epsilon
\]

Substituting, we see that

\[
\tilde{c}_i > c^*_i - (c^*_i - c^*_i) + \epsilon = c^*_i + \epsilon > \hat{c}_i \quad \forall i \mid c^*_i > c^*_i + \epsilon \quad (1)
\]

Hence even in the worst case, Equations 2 and 3 ensure that \( \tilde{c}_i > \hat{c}_i \quad \forall i \mid c^*_i > c^*_i + \epsilon \), and thus no action sequence \( i \) for which \( c^*_i > c^*_i + \epsilon \) will be selected and the policy will remain \( \epsilon \)-optimal. Note that action sequences besides \( i = 1 \) for which \( c^*_i \leq c^*_i + \epsilon \) costs are unbounded, as it is ok for them to be significantly underestimated since selecting them still allows the policy to be \( \epsilon \)-optimal.

2) **Verifying Theorem 3.1 Experimentally**

We now verify the above analysis through a controlled study of how prediction error affects task performance. To do so, we will use the true model of an environment and true cost of an action sequence for planning, but will artificially add noise to the cost/model predictions to generate model error.

Consider a 2 dimensional navigation task, where the agent is initialized at \( s_0 = [0.5, 0.5] \) and is randomly assigned a goal \( s_g \in [0,1]^2 \). Assume we have access to the underlying model of the environment, and cost defined as \( C(s_t, s_g) = ||s_t - s_g||_2 \).

We can run the policy described in Theorem 3.1, specifically sampling \( N = 100 \) action sequences, and selecting the one with lowest predicted cost, where we consider 2 cases: (1) predicted cost is using the true model, but with noise \( \alpha \) added to the true cost \( \hat{c}_i = c^*_i + \alpha \) of some subset of action sequences, and (2) predicted cost is true cost, but with noise \( \alpha \) added to the model predictions \( s_{t+1} = \tilde{s}_{t+1} + \alpha \) where \( \tilde{s}_{t+1} \sim p(s_{t+1}|s_t, a_t) \) of some subset of action sequences. The
The first case relates directly to Theorem 3.1, while the second case relates to what we can control when training a self-supervised dynamics model. When selectively adding noise, we will use uniform noise $\alpha \sim U(-\varepsilon, \varepsilon)$. We specifically study the difference in task performance when adding noise $\alpha$ to model predictions for the first 10% of trajectories with lowest true cost, the second 10% lowest true cost trajectories, etc., up to the 10% of trajectories with highest true cost. Here “true cost” refers to the cost of the action sequence under the true model and cost function without noise. For each noise augmented model we measure the task performance, specifically the success rate (reaching within 0.1 of the goal), over 500 random trials.

We see in Figure 2 that for multiple values of noise $\varepsilon$, when adding noise to the better (lower true cost) trajectories we see a significant drop in task performance, while when adding noise to the worse (higher true cost) trajectories task performance remains relatively unchanged (except for the case with very large $\varepsilon$). In particular, we notice that when adding noise to cost predictions, performance scales almost linearly as we add noise to worse trajectories. Note there is one exception to this trend: if we add noise only to the top 10% of trajectories, performance is not optimal, but reasonable because the best few trajectories will occasionally be assigned a lower cost under the noise model.

In the case of model error, we see a much steeper increase in performance, where adding model error to the best 10 trajectories significantly hurts performance, while adding to the others does not. This is because, in this environment, noise added to model predictions generally makes the cost of those predictions worse; so if no noise is added to the best trajectories, the best action sequence is still likely to be selected. The exact relationship between model prediction error and cost prediction error depends on the domain and task. But, we can see that in both cases in Figure 2, the conclusion from Theorem 3.1 holds true: accuracy on good action sequences matters much more than accuracy on bad action sequences.

3) Additional Real Robot Results

We compare the prediction error of SVG to SVG+GAP on goal reaching trajectories (Figure 3) from two real robot datasets (the BAIR Dataset [15] and the RoboNet Dataset [10]). We see that action-conditioned SVG combined with GAP has lower prediction error on the goal reaching trajectories than standard action-conditioned SVG. We observe that the GAP ablation which also conditions on the goals, but predicts residuals is equally effective in this setting.

**REFERENCES**


[52] Oleh Rybkin, Karl Pertsch, Frederik Ebert, Dinesh Jayaraman, Chelsea Finn, and Sergey Levine. Goal-
conditioned video prediction, 2020. URL https://openreview.net/forum?id=B1g79grKPr


