Learning Environment Simulators from Sparse Signals

by

Yonadav Goldwasser Shavit

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2017

Author . . . .
Department of Electrical Engineering and Computer Science
August 16, 2017

Certified by. . . . .
Leslie P. Kaelbling
Professor
Thesis Supervisor

Accepted by . . . .
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Learning Environment Simulators from Sparse Signals

by

Yonadav Goldwasser Shavit

Submitted to the Department of Electrical Engineering and Computer Science
on August 16, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

To allow planning in novel environments that have not been mapped out by hand, we need ways of learning environment models. While conventional work has focused on video prediction as a means for environment learning, this work instead seeks to learn from much sparser signals, like the agent's reward. In Chapter 1, we establish a taxonomy of environments and the attributes that make them easier or harder to model through learning. In Chapter 2, we review prior work in the field of environment learning. In Chapter 3, we propose a model-learning architecture based purely on reward prediction, and analyze its performance on illustrative problems. Finally, in Chapter 4, we propose and evaluate a model-learning architecture that uses both reward and sparse "features" extracted from the environment.

Thesis Supervisor: Leslie P. Kaelbling
Title: Professor

Acknowledgments

A little-known corollary to John Donne's famous quote states that "no thesis is written by an island". This thesis was written by many people[1], all of whom deserve thanks. Thank you to Patrick Barragan for introducing me to LIS, and to all of the LIS group for welcoming me this year - you're wonderful people. Thank you to Adam Kalai for helping bring a dead-end idea back to life, and to Adam Yala for helping me understand RNNs. Thank you to Jiajun Wu for talking through my ideas with me and giving me an expert's perspective into the state of the field. Of course, I owe a huge thanks to Leslie Kaelbling for all of her advice and insight. I had too many questions to count; some were technical, but many more were about research and career more broadly, and she never failed to provide me with perspective and understanding. To Tomas, I offer a blank stare, hoping he will appreciate its meaning.

Of course, I also could not have written this thesis without all the wonderful people in my life who both distract me and (ocassionally) push me to work. Thank you to Mikhail, Vinay, Nicole, and Onion for making our home feel so damn homey. Thank you to my girlfriend MJ for pushing me and for listening and for being cute. Thank you to Suhas for being my friend. Thank you to Imma and Abba and Bor, for anchoring me always.

Finally, last and most, I'd like to thank Caelan Garrett, for a lot of things. He was the first UROP mentor who really mentored and taught me and made me feel confident enough to motivate me on a project. All throughout my thesis, he was eager to chat every single time I barged into his office rambling about some new observation. It's thanks to our discussions that I sorted through the ideas in my head enough to really get my project going. In the end, he has nothing to show for all of it but my appreciation, my friendship, this acknowledgement, and my thanks.

[1] To the academic office, I mean this figuratively.

Contents

List of Figures

Chapter 1

Problems in Deep Environment Learning

1.1 Introduction

Model-free RL systems are limited by their reliance on policy-learning to choose actions, foregoing the potentially powerful toolset of planning algorithms to methodically evaluate future possibilities. To be able to plan in unknown environments, we must be able to acquire a simulator of the environment, to be used while planning to evaluate the future consequences of current actions. One attractive framework is to learn the environment simulator using familiar statistical techniques, e.g. deep learning. To facilitate the development of methods for learning to model RL environments, we propose a series of benchmark tasks, each of which highlights a different aspect of the problem.

We restrict our discussion to environments that have discrete timesteps and discrete action-spaces. We define an environment as a tuple $\{\mathbb{S}, \mathbb{A}, \mathbb{O}, P, V, R\}$ where $\mathbb{S}$ is the space of environment states, $\mathbb{A}$ is the space of valid actions, and $\mathbb{O}$ is the space of observations. T is the transition function governing the evolution of the environment over time, such that $s_{t+1} \sim P(s_{t+1} \mid s_t, a_t)$. Similarly, R is the reward function $r_t \sim R(r_t \mid s_t, a_t)$ (note that the results of R and T are correlated). V is the observation function providing an observation $o_t \sim V(o_t \mid s_t)$. In general, given a current state $s_t \in \mathbb{S}$, a single "step" of the environment involves choosing an action $a \in \mathbb{A}$, transitioning the environment forward via T and receiving a reward via R, and then providing the agent with an observation using O. Note that the environment is always Markovian w.r.t. the state — s_t captures within it all information necessary to predict the environment's future state s_{t+1}.

The canonical environment-learning task, in its most basic form, is as follows: given experienced observation/action trajectories $\{o_1, a_1, o_2, a_2, \dots\}_n$, we wish to learn a low-dimensional state representation s for the environment. We know, per our definition of environments, that some such true state representation s exists, but we know nothing about s, and so instead we learn a proxy state space. Specifically, we wish to find a state encoder $x_t = E(o_t[, o_{t-1}, \dots])$, and a transition function on that state representation $x_{t+1} \sim T(x_{t+1} \mid x_t, a_t)$. Other objectives include learning to extract features from the state, like a "reward function" from x_t to r_t, or a state-dependent value function, or an inverse mapping back to the current observation o_t. Ultimately, the goal is to use E and T to enable long-term planning in latent-space to improve an agent's performance.

1.2 Environment Characteristics

We can identify five characteristics of RL environments that provide different challenges for environment learning. These are:

- **Transition stochasticity:** environments with **deterministic transitions** always yield the same underlying state s_{t+1} given the previous s_t and a_t, whereas **stochastic transitions** will draw this next state from a distribution of possible states.

- **Observation stochasticity:** environments with **deterministic observations** always yield the same observation o_t given the state s_t, though there need not exist an inverse mapping from o_t to s_t. **Stochastic observations** yield different o_t's for the same s_t.

- **Completeness of observations:** environments are **completely observable** if given a state s and its resulting observation o, there is a unique inverse mapping from o back to s (in other words, o encodes all the information in s). In **partially observable** environments, no such inverse mapping exists, though one can be approximated by looking at multiple observations over time.

- **State complexity:** we describe **few-state** environments as those where the size of the set of states $\mathbb{S}$ is finite and relatively small, and the environment evolves by transitioning between these few states. On the other hand, in **many-state** environments there can be a pseudo-infinite number of states, so that an agent might reasonably encounter some states only a few times, if ever.

- **Reducibilty of states while preserving reward:** environments are **reward-reducible** if maximizing the long term reward does not require knowledge of the full state[1], i.e. there exists some reduced state representation $x_t = J(s_t)$, $dim(x) << dim(s)$ which has a Markovian transition model T_x and a reward function R_x that perfectly encapsulates the original environment's reward dynamics (a sequence of actions results in the same distribution of rewards). If predicting the reward requires the full state, and no such reduced state x is sufficient to predict future reward dynamics, the environment is said to have **reward-irreducible**.

1.3 Sample Environments

Here we describe a set of novel environments, each of which displays a different combination of the attributes discussed above.

[1]In this case, the "full state" refers to the "true" underlying state of the data generating process, rather than any learned state representation

1.3.1 MNIST-based Games

The primary approach to date for learning latent simulators is to train video prediction algorithms [25, 7], which train models to simulate games forward to do pixel prediction of future game images (in our notation, that means $f_t = o_t$). While recent work has begun considering stochastic video prediction [21], no current work has explored action-dependent video prediction for stochastic environments.

The MNIST game formulation is intended to highlight this deficiency: all of these environments use randomly drawn MNIST images to represent their underlying digit observation, and thus exhibit the "stochastic observations" that are handled poorly using current video-prediction methods.

Basic MNIST Game

In this simple game, the player's observation is an MNIST digit image, and their actions are "+0", "+1", "x2", and "x3". The observation at the next timestep will be a new MNIST image of the digit after the action's operation was applied, modulo 10. A reward of 1 is received when the player reaches "0", and every other transition yields a reward of $-.1$. For example transitions, see Fig. 1-1. This environment is interesting because it has stochastic observations, but is otherwise straightforward, having deterministic transitions, few states, being reward-irreducible, and being completely observable, .

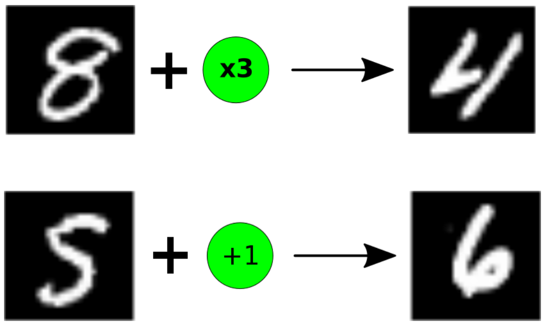


Figure 1-1: Example transitions from the Basic MNIST Game. On top, $8 \cdot 3 \mod 10 = 4$. On bottom, $5 + 1 \mod 10 = 6$

.

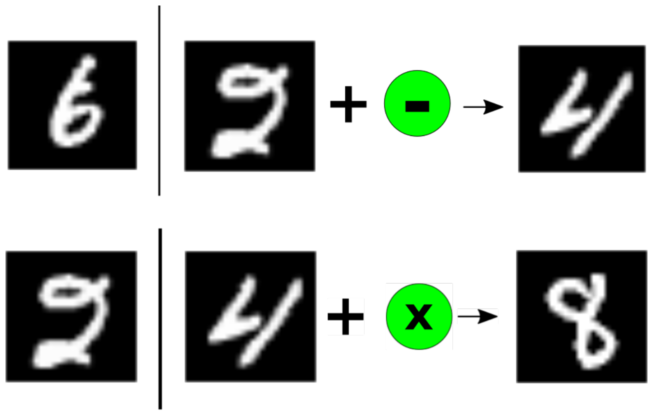


Figure 1-2: Example transitions from the Fibonacci MNIST game. On top, $6 - 2 = 4 \mod 10$. On bottom, $2 \cdot 4 = 9 \mod 10$.

Fibonacci MNIST Game

In this MNIST game variant, the observations are once again MNIST digits, but the actions are "+", "-", and "x", and the transition dynamics are slightly different. Given the previous state's underlying digit d_{t-1}, current digit d_t, and action a_t, the next digit will be $d_{t+1} = d_{t-1} a_t d_t \mod 10$. The objective, again, is to get to 0. For examples, see Fig 1-2. This game has both stochastic observations and incompleteness of observations (since the full state contains information not part of the current observation), but still has stochastic transitions, few states, and is reward-irreducible.

Colored MNIST Game

This MNIST game variant is identical to the Classical MNIST Game, except that each digit's background also has one of the six primary colors (red, orange, yellow, green, blue, violet). The number digit displayed is updated as before, but on "+1" actions the background color will take a step along the list of colors (and if violet, loop back around to red). Any action other than "+1" does not change the background color, and the objective of the game is still to get to 0, regardless of color. For examples, see Fig. 1-3. This environment has stochastic observations and is reward-reducible (because the color is not relevant to achieving the goal of reaching 0), but still has few states, deterministic transitions, and is completely observable.

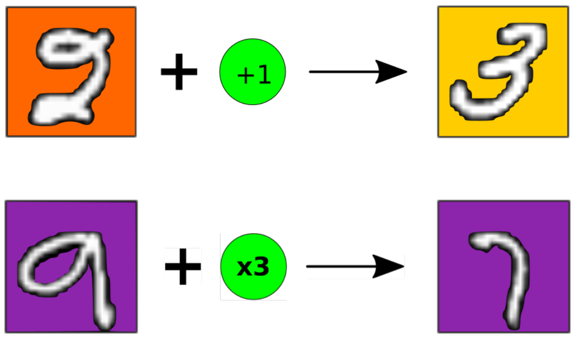


Figure 1-3: Example transitions from the Colored MNIST Game. On top, $2 + 1 = 3 \mod 10$ and the action is "+1" so the color moves one step along the rainbow from orange to yellow. On bottom, $9 \cdot 3 = 7 \mod 10$, and the action isn't "+1" so the color remains the same.

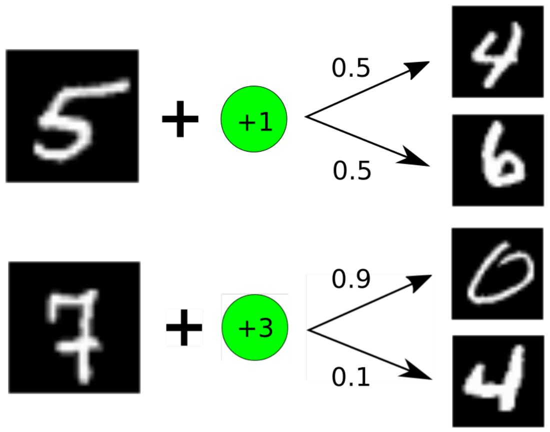


Figure 1-4: Examples of transitions from the Stochastic MNIST Game. On top, given the choice of action "+1", there's a 50% chance of the transition $5 - 1 = 4 \mod 10$, and a 50% chance of $5 + 1 = 6 \mod 10$. On bottom, there's a 90% chance of the transition $7 + 3 = 0 \mod 10$, and just a 10% chance of the result being $7 - 3 = 4 \mod 10$.

Stochastic MNIST Game

In the final MNIST game variant, the observations are regular MNIST images, and the actions are "0", "1", "2", and "3". Each action has a certain probability of either adding that number or subtracting it - for example, "+1" is equally likely to increment the current digit, or decrement it. The goal is still to reach 0. For example transitions, see Fig. 1-4. This example has both stochastic observations and stochastic transitions, while still being completely observable, reward-irreducible, and having few states.

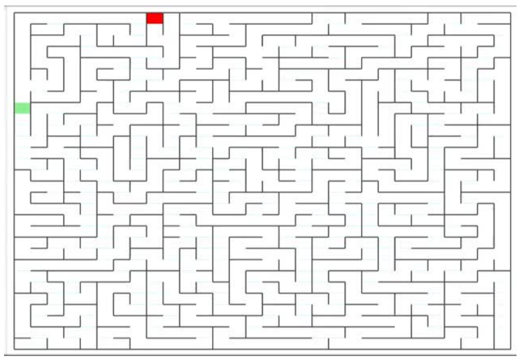


Figure 1-5: Example of a gridworld observation. The agent (red) only receives a reward when it reaches the objective (green).

Gridworld Maze

One further toy problem is the classical gridworld maze, in which the agent is placed in an initially-random position, and must traverse through a maze of walls (constant across game-instances) to reach a randomly-chosen objective. For an example, see Fig. 1-5. Unlike in the previous examples, both the observations and transitions are deterministic, the environment is fully observable, and the full state (the position of both agent and objective) is necessary to predict the rewards. However, for a grid of any reasonable size, there are many more random states than are likely to ever be encountered.

1.3.2 Classical RL Tasks

We can also use our environment taxonomy to analyze several state-of-the-art RL environment benchmarks, and quantify why planning makes them difficult. Example observations from these tasks are shown in Fig. 1-6.

Pacman

The classic Atari game Pacman, in which the player navigates an agent through a maze while avoiding enemies, is relatively simple with respect to our criteria. The game certainly has a true underlying state (the RAM). The observations are deterministic functions of the state, and though the ghosts may seem to move stochastically,

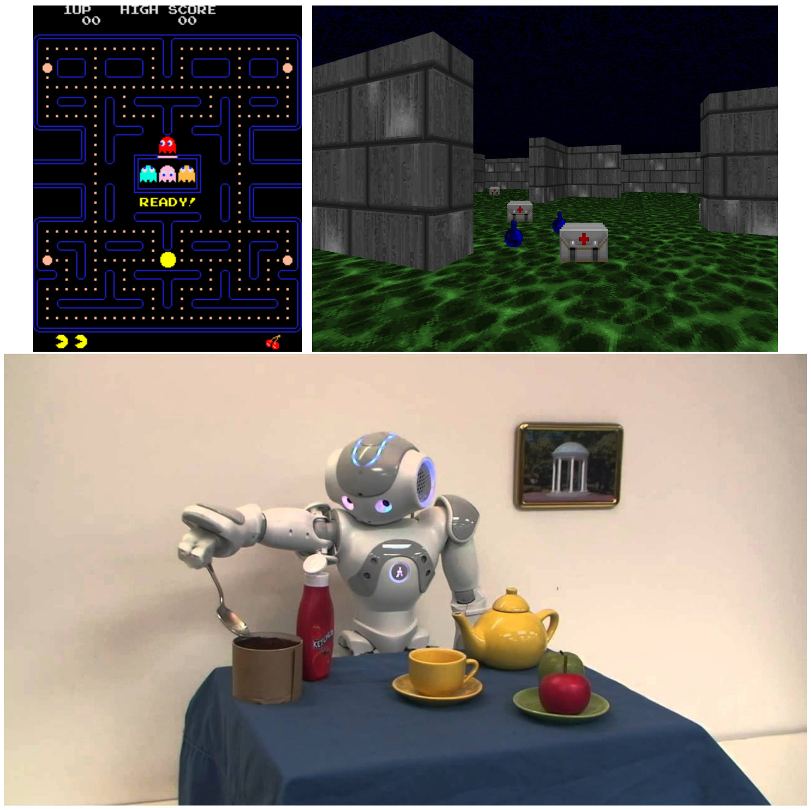


Figure 1-6: Classical reinforcement learning tasks. Top left: Atari PacMan. Top Right: VizDoom. Bottom: robotic task and motion planning.

their motion is actually a deterministic function of the previous state, making the transitions deterministic as well. The environment is completely observable (the image alone is sufficient to recover state) and is reward-irreducible (if the state s is defined as the latent representation necessary for future image prediction). However, unlike with the toy examples, this is a many-state environment. The system may never encounter most states during training, but must still generalize and at test-time must be able to simulate unseen scenarios.

VizDoom

Like Pacman, the observations in VizDoom (a 3D shooter game in which the player must navigate hallways and collect/avoid items) are fully deterministic, and under most variants the the environment has deterministic transitions. However, the game is only partially observable: the player's gaze and position restrict the sliver of the game world they see at any given time. This means that a simulator will have difficulty making predictions about a future which it does not have sufficient information to infer (like the layout of hallways in a level it has yet to explore). As with Pacman, the state complexity is potentially very high. In general, the environment is often reward-reducible: for example, the full state may include the the positions of all the apples and all the oranges, while the player might need only to collect the originals.

Robotic Task-and-Motion Planning

Learning a simulator for robotic motion problems requires tackling every single one of the more-difficult environment characteristics. Transitions are stochastic because the real world can evolve in multiple ways: a nearby human might take one of several different actions in the next timestep, and even the action of raising a robot arm has stochastic results due to actuator error. Observations are stochastic because the robot will inevitably not model certain portions of the world (e.g. leaves rustling outside the window), and the unmodelled components of the observation will seem to evolve randomly. The robot will only ever be able to detect part of the world's state (due to occlusions and sensor-limitations), making the environment partially observable. For most tasks, there are many more states than will ever be encountered (many-state). Lastly, the environment's full state (the state of the robot, as well as of every other object/agent related to the task) is in general much more complicated than is necessary for the completion of a single task (pouring the hot water from the kettle right next to the robot into an adjacent mug), implying reward-reducibility. Clearly, these challenges will have to be surmounted if robots are to learn to simulate their environments to plan.

Chapter 2

Related Works

2.1 Model Learning

In the past few years, substantial work has focused on the problem of learning a model for environments. Embed-to-control [32] was among the first, and learned a one-step prediction model for several control tasks. The input observation is an image (encoded via an autoencoder), which is then simulated a few steps forward via a linear transition model. We define a cost function for acting in this latent space (rather than learning one), and then uses model-predictive control to choose a trajectory. This system is capable of solving simple planning problems like swing-up on an inverted pendulum or balancing a cartpole, but its most interesting aspect is this: it allows for transition-stochasticity. By modeling the transition-model as a mixture of Gaussians, and achieving latent-space self-consistency by minimizing the KL-divergence between "true" latent states and "predicted" latent states, this paper tackled a class of environment models that many subsequent papers have not yet addressed.

Another class of model-learning algorithms centered around learning a model representation useful for planning. These generally include a recurrent transition model that takes as input a latent state and action and outputs a future latent state, and a decoder of some sort. In Figure 2-1, we provide diagrams for many of the architectures we'll discuss in this section.

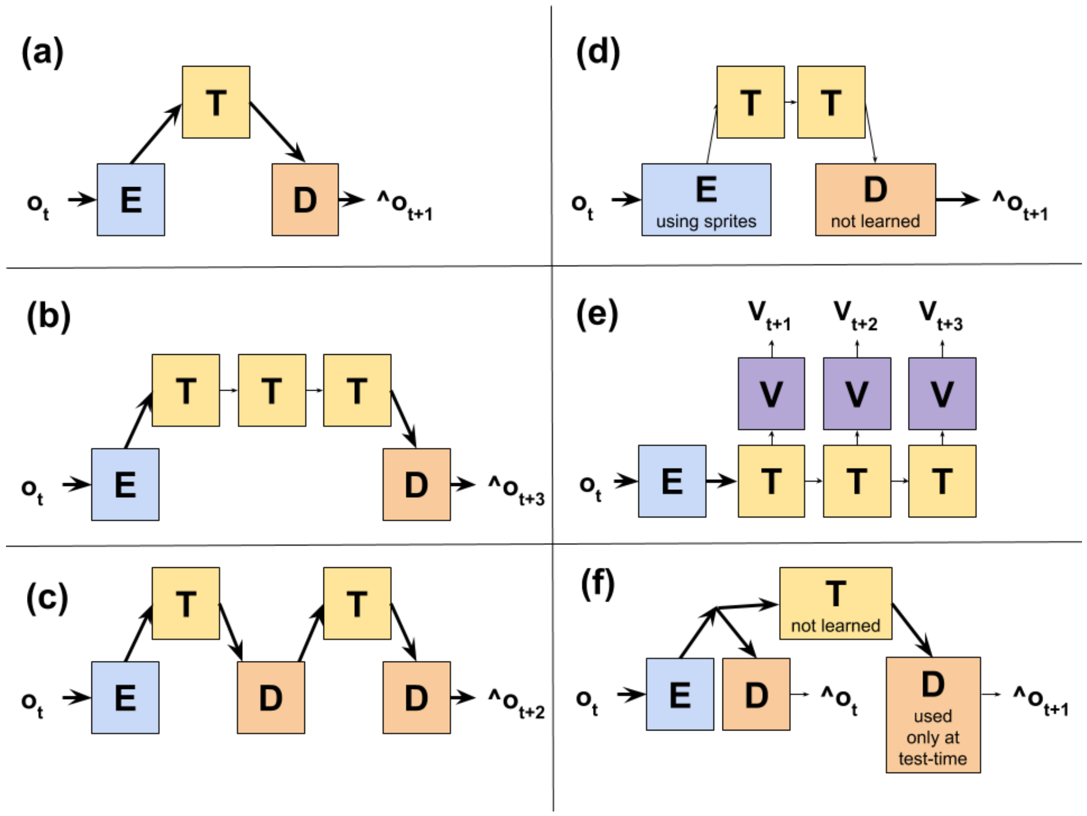


Figure 2-1: Diagrams for the model-learning architectures in previous approaches. **(a)** Oh et al. [25], **(b)** Chiappa et al. [7], **(c)** from same paper by Chiappa et al., **(d)** Guzdial et al. [15], **(e)** Oh et al. [26], **(f)** Asai et al. [4].

Oh et al. [25] was the first paper to propose training each of these components as a fully-differentiable neural network. Their approach learned a 1-step image prediction model (where the decoder maps from a latent state to the pixels of the future image), and repeated the one-step block for longer horizons to gain an environment model for deterministic-transition, deterministic-observation, fully-observable environments. They show good performance on an Atari environment and use the model to inform an exploration stategy for a policy-learning algorithm.

Chiappa et al. [7] expand on this approach by training an environment model with multiple transition steps. Their encoder takes in several frames to generate the initial latent state, making their algorithm viable in partially-observable environments (which must still have deterministic transitions and observations). They propose two different architectures. In the first, the model encodes the observation, simulates it forward n timesteps, and then decodes it back into an image. In the second more complex approach, the model encodes an observation, simulates it one timestep forward, decodes it, and then re-encodes it and simulates it forward again, n times. They found that the latter approach worked better for pixel-level image prediction over longer sequences, which they attribute to the repeated "grounding" of the latent state in some sort of true image.

Much simpler model-prediction architectures were applied in the real world in "Learning to Poke by Poking" [1]. First, the authors learn an encoder by training a model to take two subsequent observations as input, encode both of them, and then using that encoding to predict the action that occurred between them. Using this encoding, they trained a simple one-timestep pixel prediction model. This was all done on a real robot whose goal was to poke objects, and a simple one-step policy using simulations from the learned model to maximize the displacement of the object proved effective in their experiments.

Fundamentally, even highly-trained models will suffer from some level of inaccuracy, which will compound with longer prediction horizons. There is a danger, as seen in Gu et al. [14], that planning agents will exploit the inaccuracies in these models to generate plans that maximize an inaccurate reward signal. Weber et al. [33] propose a reinforcement learning algorithm that extracts useful information from flawed models for use in a model-free policy learner. By training a flawed single-step model, doing multiple rollouts of that model, and using the results of those rollouts as an informative input to the policy network, the authors are able to attain good results on complex planning-based domains like Sokoban.

Guzdial et al. [15] aim to learn a video-predictive model by essentially hand-engineering an encoding. They inform the program about the different sprites in the game Super Mario Bros., and then train their model to combine these sprites to reconstruct the game observations on a pixel level. They then learn a transition model for these high-level sprite representations, and are able to successfully plan in the environment. This suggests a way of incorporating prior domain knowledge into model-learning.

Another recent work by Oh et al. [26] proposed an entirely different framework for learning a model that, much like this current work, does away with image-prediction and latent-state invertibility entirely. They train a model with an encoder, an optional-conditional transition module, a reward-prediction module, and a value-prediction module. The reward prediction module is used only to impose structure on the learned latent space (much like our "features" in Ch. 4). The model is trained using a modified form of Q-learning (where the predicted values are the output of encoding an observation, transitioning it forward, and then calling the value-extractor). After training, the algorithm uses model-based planning to choose a trajectory that maximizes its estimated final value, and are tested on a number of maze and Atari planning tasks. The algorithm leverages the strength of a model-based structure for forward-simulation, while still learning using model-free RL approaches, and in doing so can leverage the benefits of both.

Another paper by Jonschkowski et al. [17] proposes learning a model for the environment that similarly does not require image reconstruction, but instead learns a latent space consisting purely of position-velocity encodings by forcing the latent space to obey a variety of real-world robotics priors. These include variation (random frames should have different representations), slowness (positions should not change quickly between frames), and inertia (velocities change slowly). The learned latent state can then be used to fit a value function, and the approach was thus able to solve several classical planning tasks directly from images. A more thorough study of possible robotic priors for any physical latent space can be found in another work by Jonschkowski [16].

Asai and Fukunaga [4] propose a method for model-learning that allows for classical PDDL planning in latent space. They train an encoder via an autoencoder with a constraint to make the latent-space be almost-discrete, and use as a transition model a lookup table of all possible transitions. By providing the algorithm with an initial observation and a goal observation, the algorithm can use an off-the-shelf classical planner to compute a plan using only the latent representations. However, the experiments in this paper are only on observation-deterministic environments, and further work is needed to demonstrate the viability of this approach in a broader class of environments.

2.2 Video Prediction

A related field to model-learning is video-prediction, the problem of predicting future video frames given previous frames [30]. The two problem-types share many challenges, with the only substantial difference being that video-prediction need not consider actions or rewards.

One of the more interesting recent ideas in video prediction is adversarial video generation [21]. As previously discussed, pixel-level MSE loss for video prediction does not take into account the fact that the real world is stochastic, and that there are often multiple ways for a video sequence to progress (rather than the single, observed way). This paper uses the generative adversarial framework introduced by Goodfellow et al. [13] to train networks that stochastically generate a video-sequence, and are then critiqued by a discriminator which attempts to distinguish whether the predicted future is likely to be true or not. This modification, in model-learning, would allow learning in stochastic-observation environments (including e.g. the MNIST game).

However, a single adversary might not be sufficient to learn stochastic-transition models, in which each timestep branches stochastically into future timesteps in probability distributions of compounding complexity. Esteban et al. [12] attempt to address this problem by learning a stochastic time-series using adversarial losses for predictions in each timestep. In order to stabilize the learning process, they used a form of curriculum learning in which the number of predicted steps was slowly increased. While the work shows initial progress, more results are needed to explore the viability of a compounding adversarial approach in estimating environments with complicated stochastic transitions.

Villegas et al. [31] address another aspect of video-prediction that is related to our work: high-level feature spaces. By utilizing the known high-level structure of the process, they are able to train a model to predict future video frames much more accurately. Specifically, the authors extract body pose position (using a pretrained neural net) from frames, predict those pose positions into the future, and then decode those images back into real images. This work also uses an adversarial loss to construct video without requiring that the model adhere to the original video.

Denton et al. [10] learn a video-predictor which explicitly disentangles different components of the latent representation. By imposing a custom adversarial loss, the model is forced to learn video "content" in one component of the latent space, and intra-video "pose" information in the rest.

One related area of interest is physics approximation networks [5, 6], a class of models that aims to explicitly learn physical transition dynamics for objects by modeling each object's collision with each other object separately. Perez et al. [29] extend this idea to video-prediction, using the prediction network to anticipate the motion of many 2D objects in a simulated scene.

Similarly, Wu et al. [34] use a pre-built classical game physics engine as a transition model, and learns a mapping from real object states to components in the physics simulator. To train, the model forces an encoder to provide the physics engine with a problem specification that will yield a video similar to the one that followed in the real world. While this requires highly accurate simulators, it also yields a thoroughly interpretable feature space (which can be immediately used by the same physics simulator in many different ways).

2.3 Alternative Approaches to Learning Models for Reinforcement Learning

There are many interesting applications for learning models to be used in contexts other than classical action-planning. Gu et al. [14] use an iteratively-fitted time-varying linear model to approximate the dynamics. They then generated simulated trajectories and provided them as input to a model-free RL algorithm, allowing the algorithm to artificially generate more data for itself and thus utilize less data to achieve comparable results.

Similarly, Munk et al. [23] leveraged model-learning to extract useful features for model free planning. The authors trained a simple one-step observation predictor and one-step reward predictor model, both neural networks, and then used the combined first layers of each model as the first layer of a model-free RL algorithm to learn a feature-representation to reduce the parameter-complexity, and thus require fewer examples to yield a good policy.

Dosovitsky et al. [11] attempt direct long-horizon future feature prediction without a recurrent component. They train an RL agent on-policy to complete a certain task, and then train a CNN to estimate, given each possible action, the value of certain measurements (rewards) far into the future given the current policy. The policy chooses the action with the highest expected associated reward. Training the long-horizon reward predictor in this way does eventually stabilize, and is even able to perform well on tasks in non-trivial 3D game environments.

Finally, Karkus et al. [18] learn a transition model specifically for the POMDP-solving QMDP algorithm. The "model" consists of the spatial transition kernel weights, which are then applied iteratively using the QMDP algorithm. The objective is to minimize the imitation cross-entropy loss between an expert's chosen trajectories and the trajectories generated using the learned-model QMDP planner. Interestingly, the model learns a transition structure that is inaccurate, but tuned to be utilized well by the planning algorithm (which is not optimal).

Chapter 3

Model-Learning as Reward Prediction

3.1 Introduction

In this chapter, we will discuss a novel approach to predictive learning that relies purely on reward prediction. Predictive learning is an alternative to model-free reinforcement learning, though both have the same goal: to train agents that can choose actions in novel environments to maximize an objective. Model-free RL seeks to learn a successful policy $a \sim \pi(s)$ without truly understanding the environment. In contrast, predictive learning seeks to learn a transition model $s, r \sim T(s, a)$ to explain the environment's behavior, and then uses this model to foresee the results of its actions and choose the best action accordingly.

These two approaches can be understood by analogy to human learning. When learning to swim, a child initially does not know how to push her body forward, and may fall down. By trial, error, and instruction, she learns a strategy to move forward in the water. Yet afterwards, if we asked her to tell us what would happen if she only used one foot, she might not know. This is because she has learned a **policy** for swimming, not an **environment model** for her body in the pool. On the other hand, if we were to give the same child a paintbrush and ask her what would happen if she moved her hand in each of several different patterns, she could accurately guess the shapes that would result on the paper without having laid down any paint. This is because she has learned an **environment model** for painting.

This notion of learning environment models has been explored primarily in the context of video-prediction, where the observations are images [25, 7, 21, 31]. These approaches generally have the same few model components. There is an encoder $E(o_t) \to s$ mapping a high-dimensional observation (like an image) to a simple latent state, a transition-model $T(s_t) \to s_{t+i}$ simulating the latent state forward, and a decoder $D(s_{t+i}) \to o_{t+i}$ mapping back from a latent state to an observation (or an image). The training objective is either the observation reconstruction loss (pixel-value MSE) or an adversarial loss (inability of an adversary to distinguish between real and generated future observations).

All of these approaches have the same flaw: they aim to learn an embedding (and associated transition model) complex enough to perfectly reconstruct every part of the state space, by regenerating a complete image/observation. Training a decoder alone to generate high-dimensional images from a sparse vector would be challenging on its own. However, there are many instances in which such a detailed environment model is even unnecessary, or simply unlearnable. Imagine predicting pedestrian behavior as a self-driving car: while it is vital to be able to guess the future location of a pedestrians, it should not be necessary to encode the shape of a man's stroller or the patterning on his backpack. Yet such features are given roughly equal weight when the learner's goal is pixel-level video-prediction. Our real intended objective, which pixel-prediction serves as a proxy for, is to predict the evolution of the higher-level features of the environment.

This work aims to approach the environment-learning problem from the opposite direction: by predicting the future reward, rather than future environment observations. The model still learns an encoder and a transition model, but replaces the image decoder with a "goal decoder", which extracts the goal value from the current state. When trained, this model should learn some reduced[1] representation of the state that is necessary to evaluate the reward at the present and in the future. Ultimately, when planning, the most important thing as that the model accurately

[1]Any "reward-reducible" environment (per Chapter 1) will be modeled using the simplest state the learner finds satisfactory.

predict the goal, and this allows us to optimize for such behavior directly. Further, the latent representation need not be invertible back to a complete image observation, and so can be simpler and sparser (a major difference from image-prediction-based approaches). We thus avoid the need to learn to model complex and irrelevant aspects of the environment. Finally, the model learns to extract not only features necessary for calculating immediate reward (e.g. am I currently colliding with a pedestrian?), but also features that predict reward in future timesteps (is the pedestrian currently on a path such that, if I keep the pedal to the metal, I will collide with him?).

Of course, such an approach is not without its drawbacks. First, such a model is by its nature less adaptable to changing objectives/reward functions, since it only learns to model the portion of the environment that is relevant to the reward function it's trained on. Second, the model must learn from an extremely weak signal. Rewards are inherently sparse signals that may only occur infrequently, and may contain little information content at that. ("Is the goal reached?" is a binary signal.) Relatedly, the longer the time between an action and its effect on the reward, the more difficult it is to determine how that action affected the state, and the weaker the learning signal. Imagine trying to learn the rules of chess by repeatedly looking at a board configuration, then leaving to another room and calling out your pieces' moves. Your only feedback is being told that "the game isn't over", or you "won" or "lost". It would be difficult to build any meaningful hypothesis for the rules of chess, let alone a robust one.

To ameliorate the low-signal problem, we will introduce a few proposed auxiliary losses that will provide more structure and signal to the latent space. First and foremost, we force the latent state representations to be self-consistent: the current latent state should be the same if extracted from the current observation, or from an older observation simulated forward in time. In Chapter 4 we will predict pre-crafted features in addition to the predicted reward, providing more signal to construct a latent space while still forcing the algorithm to infer other important information details. In Appendix A, we will explore the idea of requiring the latent state to contain sufficient information for a classifier to distinguish between true and false environment transitions. As we introduce each of these objectives, we will discuss how they each affect the learned latent environment representations, and why they do or don't make sense in different settings.

To clarify, the goal of this work is **not** to present reward prediction as a practical approach to training functional environment simulators[2]. Rather, our goal is to explore the conceptual merit of a fundamentally alternative approach to learning an action-conditioned environment state representation. The usual approach, most prevalent in video prediction, is to require the model to learn a state representation that can predict a ton of at-times irrelevant information about the environment (the value of every pixel) and to hope that, in the process, it picks up the features that are important. By contrast, our approach asks the model to predict information insufficient to even model the environment dynamics (e.g. the reward) and relies on the learner to infer the rest of the state from that partial signal. Doing so, learning from this weaker signal, is certainly a harder learning problem, but at the same guarantees that the features we do learn are the ones we want, and it is worth understanding its drawbacks and successes.

3.2 Problem Statement

We will now detail our precise problem statement. We are given an environment with discrete timesteps and observations o_t, actions a_t, and a reward signal g_t. The environment must have deterministic transitions (see Ch. 1.2) and must be fully-observable, but can have stochastic observations (many different observations which represent the same state). We are given access to historical environment trajectories $\{(o_t, a_t, g_t, o_{t+1}, a_{t+1}, g_{t+1}, \dots, o_{t+T}, g_{t+T})_{1,\dots,n}\}$. To simplify our problem setting, we make the reward a binary goal. [3]

Our primary objective is to learn a model that, given a new observation o_t and

[2]If you build a car that drives purely by reward prediction, please use a holdout set.

[3]The reward or "goal value" is 0 whenever we have not achieved the goal, and 1 when we have (at which point the episode ends). We define goal satisfaction as a function of the state, rather than the combined state and action. Relatedly, when specifying one of multiple goals, we provide a "target observation" o_g that indicates to the agent which goal it is aiming for.

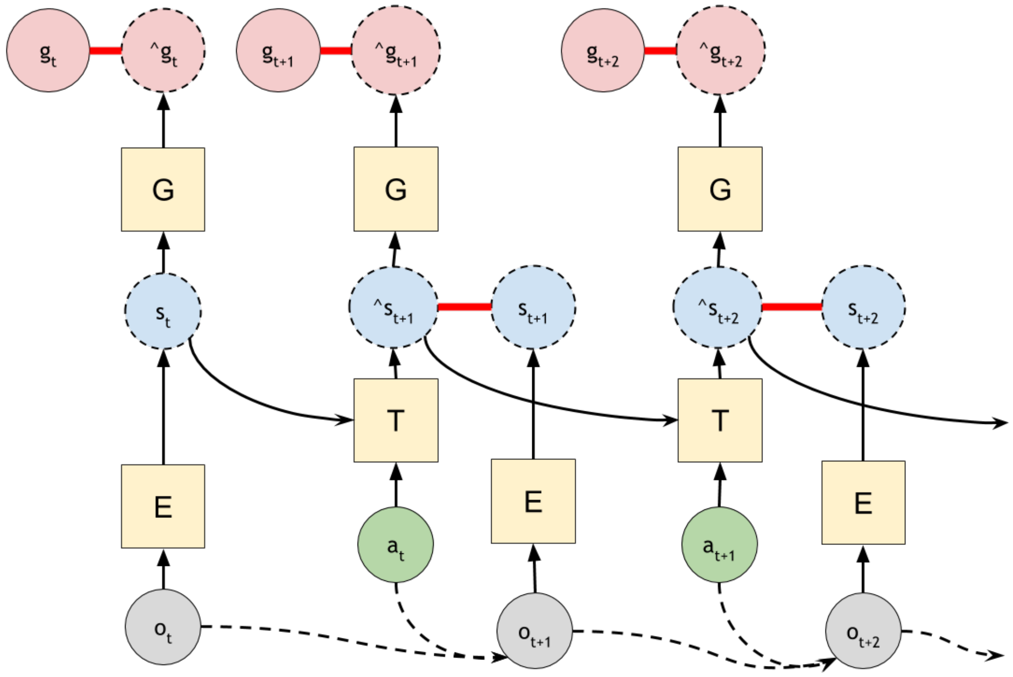


Figure 3-1: The layout of two steps of the reward prediction model. Circles with solid borders are provided from the environment, while circles with dashed borders are learned during training. Red lines denote pairs of elements whose distances to each other are minimized.

a sequence of actions $a_t, a_{t+1}, \ldots, a_{t+T}$ predicts the value of goal at time $(t+T)$ $\hat{g}_{t+T} = f(o_t, a_t, a_{t+1}, \ldots)$ that minimizes the cross-entropy $-\sum_i g_{t+T}^{(i)} \log \widehat{g_{t+T}}^{(i)}$.

3.3 Model

We propose learning an environment simulator with three components: an encoder E, a transition function T, and a goal-extractor ("goaler") G (see Fig. 3-1). The encoder maps from the high-dimensional observation input o_t to a learned low-dimensional state representation x_t.[4] The transition function $T(x, a_1, \ldots, a_n)$ takes as input a latent state and simulates it forward in time given a sequence of actions, resulting in the estimated subsequent latent state. Finally, the goal extractor $G(x)$ takes as input a latent state, and outputs the likelihood (between 0 and 1) that this latent state is

[4]As a reminder, the environment is fully observable and thus a sufficient state can be extracted from just the observation. This assumption can be relaxed, as we discuss further in 3.6.

a state that has achieved the goal. These functions are summarized in the notation below:

$$x_t^i = E(o_t^i) \tag{3.1}$$

$$\widehat{x_{t+j|t}}^i = T(x_t^i, a_t^i, a_{t+1}^i, \ldots, a_{t+j}^i) \quad , \quad \widehat{x_{t|t}}^i = x_t^i \tag{3.2}$$

$$\widehat{g_{t+j|t}}^i = G(\widehat{x_{t+j|t}}^i) \tag{3.3}$$

During training, we break each episode's experience into short subsequence of transitions (of length up to h_{max}). Our primary training loss is, as previously mentioned, the reward prediction loss for each timestep in each subsequence, defined as:

$$\mathcal{L}_G = \frac{1}{Z_1} \sum_{i=1}^{n} \sum_{h=1}^{h_{max}} \sum_{t=0}^{T^{(i)}-h} \sum_{j=1}^{h} -g_{t+j}^{(i)} \log \widehat{g_{t+j|t}}^{(i)} \tag{3.4}$$

where n is the total number of training episodes, h_{max} is the maximum sequence length the model is trained to predict at each time, $T^{(i)}$ is the total number of steps in the ith episode, and Z_1 is the total number of terms in the loss. Note that we break each training trajectory up into groups of different-length sequences (from 1 to h_{max}), and train on all of them. If we did not do this, and only provided sequences of h_{max} steps, the network could learn that goals only occur exactly h_{max} steps after the encoded image (because episodes end after the goal occurs, and so can only occur at the end of a subsequence). This way, goals are interspersed at different timesteps relative to the initial state.

However, minimizing this loss alone would fail to yield a meaningful model, because there is no guaranteed similarity between latent states from different runs. To understand this issue, consider the following game. The first observation is always $o_1 = 1$, and there is only one action which increments o_t by 1, until $o_{100} = 100$ at which point the goal is satisfied ($g_{101} = 1$). However, during training, we only predict training trajectory chunks of maximum length $h_{max} = 5$ consecutive actions. Then any time the trainer ever sees examples $o = 10$, or $o = 20$, or $o = 94$, it knows that all 5 subsequent states will not achieve the goal — after all, only training trajectories starting at $o = \{96, 97, 98, 99, 100\}$ would ever yield nonzero goals. Thus there is no signal differentiating $o = 10$ from $o = 20$, and the encoder cannot learn to separate them — so it cannot learn the correct game dynamics beyond the length of its training trajectories.

To remedy this problem, we introduce a "latent consistency loss", which requires that latent state representations not only be predictive of the future goal value, but also be consistent between themselves. That is, the same state arrived at through multiple different simulated paths should yield similar latent representations. More formally, we minimize the mean squared error between an observation o_t's true latent encoding (computed via the encoder as $E(o_t)$) and the encoding estimated by simulating a previous encoding forward in time (see Eq. 3.2). That is,

$$\mathcal{L}_C = \frac{1}{Z_2} \sum_{i=1}^{n} \sum_{h=1}^{h_{max}} \sum_{t=0}^{T^{(i)}-h} \sum_{j=1}^{h} \left| x_{t+j}^{(i)} - \widehat{x_{t+j}}^{(i)} \right|_2^2 \tag{3.5}$$

where Z is the total number of latent-pair terms in the loss. We combine these two losses into one overall loss:

$$\mathcal{L} = \lambda_G \mathcal{L}_G + \lambda_C \mathcal{L}_C \tag{3.6}$$

where λ_C and λ_G are hyperparameters defining the relative importance of the two losses (and are 1, unless otherwise stated). To return to our earlier example game, take the representation of x_1 vs. x_{95}. Previously, both yielded the same goal values over 5 timesteps and thus were inseparable — though x_{96} would yield a goal state after exactly 5 steps, and thus was distinguishable as its own state. Now, x_{95} leads to x_{96} after one timestep, whereas x_1 does not — meaning that there is signal separating the two states. In this manner, by bootstrapping the representation of each state's relative relationship to other states all the way back to those that are actually near a goal, this entire game's dynamics can be learned.

This "learning-by-bootstrapping" approach is the expected (and in practice observed) learning sequence for much more complicated game dynamics. So long as the environment's underlying state has deterministic transitions (which guarantees us that starting at the same state executing the same actions twice will yield us two equivalent states), the state consistency loss can in theory infer an arbitrary transition graph. Of course, the farther away a state is from any goal states, the longer the network must train (and learn each intermediate state) before it can identify the faraway state, and the weaker its training signal. In our experiments, we will show that this approach works for short-horizon environments, but breaks down over longer horizons and more complicated transition models.

To understand the behavior of such an optimization, one must understand that the two goals — "predicting rewards" and "ensuring similar states are near each other in latent space" — are fundamentally at odds. This is because the trivial solution for the latter latent consistency loss is that *all states have the same latent representation.* This immediately reduces the latent loss to zero, and is a powerful local minimum: any differentiation at all between states for the purposes of reward-prediction needs to be more-than-worth its cost in resulting distance between that and every other state (and in our experience, systems do not escape this mode once it has occurred). This phenomenon, latent collapse, was observed empirically whenever λ_C was raised too high (and generally makes training systems with sparse reward signal difficult).

Because all state representations are constructed relative to the goal states, it is particularly vital that the model accurately identify which states are goal states. In certain environments, especially those with longer horizons, such states may occur infrequently. If a game last on average 20 steps, but our training chunks are of size 2, it would be very rare for a chunk to contain a nonzero goal — making the only effective loss term the latent consistency term, and making latent collapse all but certain. To specifically avoid this scenario and boost the goal signal strength in environments with rare goal-achievements, we optionally introduce a 0-step goal-boosting loss:

$$\mathcal{L}_{GB} = \frac{1}{n} \sum_{i=1}^{n} -g_{T^{(i)}}^{(i)} \log \widehat{g_{T^{(i)}|T^{(i)}}}^{(i)} \tag{3.7}$$

where n is the number of training episodes and $T^{(i)}$ is the last timestep of each training episode. (If there are episodes that end without reaching the goal, we discard them.)

In effect, we are training the model specifically to recognize goal states as a sort of auxiliary training signal[5]. Our final loss function is thus:

$$\mathcal{L} = \lambda_G \mathcal{L}_G + \lambda_C \mathcal{L}_C + \lambda_{GB} \mathcal{L}_{GB} \tag{3.8}$$

3.4 Methodology

3.4.1 Architecture

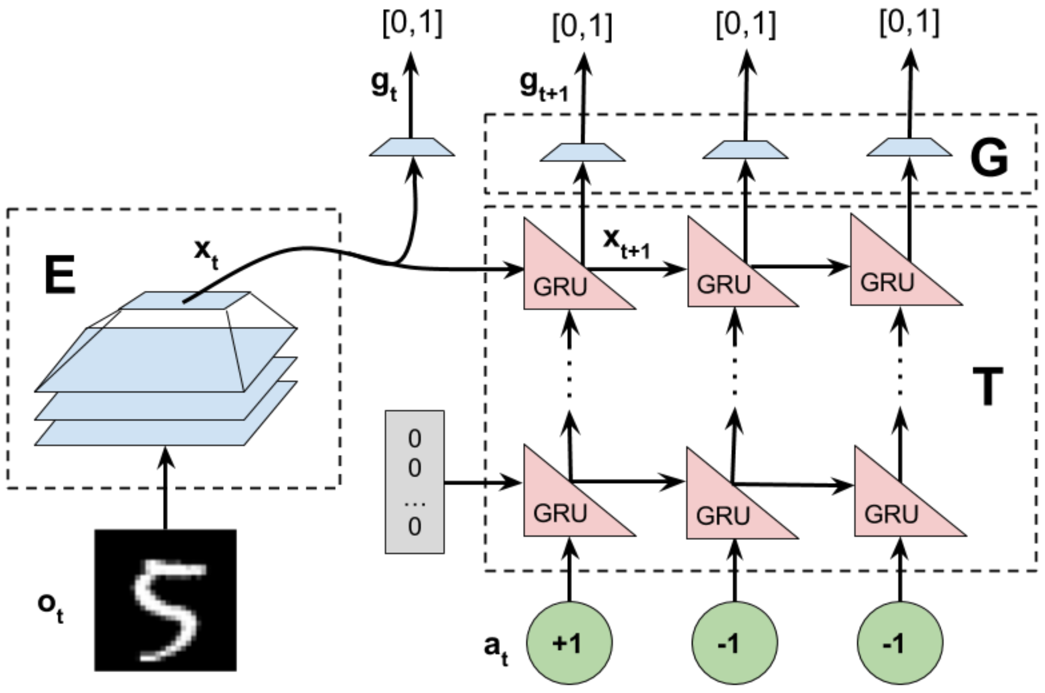


Figure 3-2: A diagram of the full prediction pipeline, for 3 transition timesteps and with stacked recurrent units. From left to right: an observation o_t is encoded using E into x_t, which is concatenated with 0s and passed as the initial latent state of the RNN. The RNN takes actions as inputs and combines these with the initial latent state to compute the subsequent latent states. Up top, the goaler G is used to extract estimates for $g_t, g_{t+1}, \ldots$ from latent states.

Our encoder, transition-model, and goaler were all implemented as deep neural networks. The encoder E is charged with extracting a low-dimensional state from

[5]Of course, optimizing the goal-boosting loss alone would make the goal-predicter always guess the goal were achieved, so it must be trained in conjunction with the regular goal-prediction loss, which will have many more goal-not-achieved examples.

a high-dimensional, spatially structured image. To that end, it has 3 convolutional layers, with each layer having 32 filters of size (5,5) and stride (2, 2), followed by an exponential-linear nonlinearity [9]. The encoder's last layer is a fully-connected layer reducing the output dimension to a latent state of size 128.

The transition model T is a recurrent neural network whose hidden state is the latent environment state, and whose inputs are actions. Our network used Gated Recurrent Units [8] with tanh nonlinearities, which have the same "output" and "hidden state"[6] (unlike LSTMs). The initial hidden state of the RNN is the initial latent state x_t. One possible drawback is that we only have a single hidden layer of nodes for each recurrent unit. We experimented with increasing the network size by using a stacked RNN [27] which adds more representational capacity by feeding lower-layer outputs of the RNN as higher-layer inputs. In this case, our latent state was the output of the highest layer, and the initial hidden state was x_t concatenated with an appropriate number of 0s (as the initial states of the lower layers), allowing the network to do more complex calculations on strings of actions. However, unless otherwise noted, the experiments used a single RNN layer.

The goal-extractor G is a simple network with one fully-connected layer with 256 nodes followed by an exponential linear unit, and then a fully-connected layer leading to a single unit, whose output is run through a sigmoid (and represents the goal value).

3.4.2 Environments

Our chosen environments are all based on the same underlying principle: we want to test how well our model learns to construct a latent representation of the environment, from the reward alone. To that end, we want to create environments with simple latent representations that are difficult to extract from the observation (and therefore require a lot of signal to perceive). We also want a relatively simple transition model

[6]This is desirable because we would like the network to output the latent state at each timestep, and also be able to then take that latent state and continue simulating the state forward (like a hidden state).

and goal function, so that learning these components is relatively straightforward. We chose to work with discrete actions and state spaces to be able to clearly capture the ability of the model to segment the state space.

To that end, we experimented with two variants of the MNIST game (originally introduced in Chapter 1). As a reminder, the game's observations are MNIST digits ($28 \cdot 28$ black-and-white images with pixel values normalized to be between 0 and 1), randomly generated based on the current "true" underlying state, which is a digit from 0 to 9. We will explore the behavior of the model in learning two variants of the MNIST game, "linear" and "complex", which differ in their transition structure. In both variants, the goal is achieved when the the internal digit becomes 0.

The "linear" variant of this game has only two possible actions: "+1" and "-1" (where e.g. "+1" means "add one to the current digit, modulo 10"). Thus the environment is a slightly more complicated version of the earlier thought-experiment about the game with 100 states and 1 action. This environment is interesting because its transition structure is extremely simple (the numbers can be thought of as a ring, where reaching either end achieves the goal). Yet each state leads to a reward pattern, that is, executing any sequence of actions that reaches a goal will distinguish the original state from every other original state (because each other state would be a different distance from the end 0s).

The "complex" variant, on the other hand, has an intentionally complicated reward structure, with actions "+0", "+1", "x2", and "x3". The visit frequency is no longer uniform in this game: 1 only occurs as an initial state or from a 7 that is multiplied "x3". Similarly, 0 (the goal) can only be reached by getting to 5 and multiplying by 2, or getting to 9 and adding 1. Similarly, the distribution of states is "stickier" and more locally-consistent: even numbers tend to follow even numbers because "x2" always moves from odd to even (unlike the other actions which are symmetric w.r.t. evenness). These different state distributions should lead to a more complicated task as far as learning latent spaces is concerned.

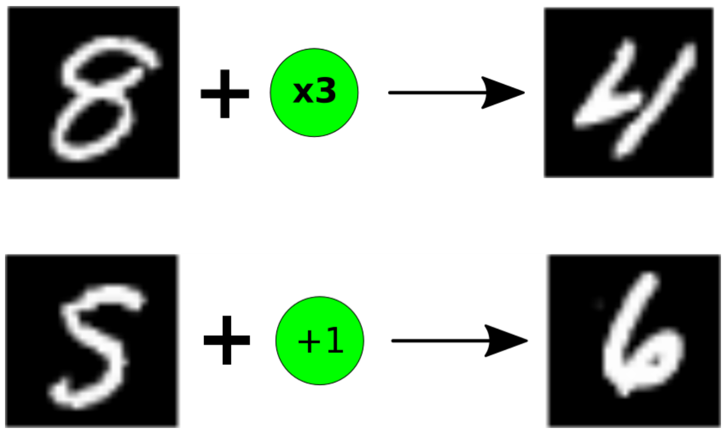


Figure 3-3: A pair of example transitions from the

3.4.3 Training

We train each algorithm for 5000000 training batch steps, using the Adam optimizer [19] with step-size $\mu = 10^{-4}$. The trainer is provided roughly 5000 sample games, though only has access to 500 at a time (the games are kept in a buffer, with 10 new games replacing old ones every 10000 batch steps). To generate sample trajectories from each environment, we randomly sample actions until the goal is achieved.

3.4.4 Metrics

We will use four metrics for our models. The first is a latent-space visualization based on Google's TensorBoard application. The model is given a set of 1024 trajectories (o_t, a_t, $o_{t+1} \ldots$ sequences) and asked to generate the predicted future latent representations given the first image and the sequence of actions. Then for each timestep, we take all the latent states in that timestep, and reduce their dimensionality using PCA (principle component analysis). We then project each of the points (and its corresponding ground-truth observation) in 3D using the 3 most important dimensions from the PCA. Doing so, we can get a quick visual representation of the learned factoring of the latent space.

The second metric is a measure of the extent of state cluster formation. We would ideally like each group of observations corresponding to a certain real-world state to be grouped together (and easily separable) in latent space. To measure whether this has occurred, we run a clustering algorithm on a set of 1024 128-dimensional latent states generated by passing observations through the encoder E. Specifically, for the MNIST game experiment, we run the KMeans++ [3] algorithm with 10 clusters (because there are 10 expected classes) and 100 random seeds for up to 300 steps each. The best-generated clustering defines the "best" grouping of the latent space into 10 chunks, which ideally would be the 10 classes. We then compare this classification to the true classification of each state by calculating their normalized mutual information score (NMI) [22]. The NMI is a symmetric measure of the similarity of two categorical distributions, which is agnostic to group membership. [7] One can permute the labels of the classes without changing the measure of the metric, which is important because we do not know which kmeans clusters correspond to which real-world classes. This metric in a sense represents both how closely the components of each class have clustered together, and how well the different classes are distinguished in latent space.[8]

The third metric is the goal-value prediction loss. This is simply the average cross-entropy loss of goal-value predictions after k steps.

The last metric is the predictive accuracy of planning using the learned environment model. We utilize the learned model with the following simple planning algorithm. Given an initial environment state s_t and a set of valid actions A, simulate a breadth-first search of all possible actions and their resulting latent states up to a fixed horizon h_p. Then look through all the visited latent states, choose the latent state with the highest estimated goal-achievement accuracy, and then (in the real environment) execute the sequence of actions that resulted in that latent state.

[7]To give some intuition for the metric: say I have 100 points A, split into 10 different classes (0 to 9). I construct B, which has only 2 classes, by taking the class of A modulo 2. A and B are two alternate labellings of the same set of points, and will have an NMI score around 0.5 — that is, if you can correctly segment the points into just two mega-classes (comprised of the original 10), you already get to 0.5. Similarly, if C results from the previous procedure except modulo 3, the NMI score would be around 0.67.

[8]Of course, there exists the problem that the neural network might use a feature representation that is not based on spatial proximity, and thus would not be captured by the kmeans-generated clusters. However, because the spatial-consistency loss specifically penalizes state representations based on how far same-class latent states are from each other, this is not likely. This was borne out empirically (by inspecting the latent visualizations).

Because our test games (both the "linear" and "complex" MNIST variants) can be solved in at most 5 moves from any initial state, we chose the planning horizon to be $h_p = 5$. Note that this metric evaluates a weaker condition than the goal-prediction loss. Goal estimates need not be precise to generate a good plan, only properly ordered. So long as at least one goal-achieving state are assigned higher predicted goal likelihood than all non-goal-achieving states, the plan will succeed. Still, this metric is important as it quickly demonstrates the true usefulness of the learned model.

3.5 Experiments and Results

3.5.1 Visualizing the Learned Representation

We will first dive into the properties of a single learned model. We trained a model with a single recurrent layer and $\lambda_C = \lambda_G = 1$ to predict 3-step subtrajectories in the linear MNIST environment. Note that using only 3-step sequences is significant, because that means that no subtrajectory beginning in state $\{4, 5, 6\}$ will ever net a reward. The resulting latent space is visualized in Fig. 3-4.

We first can observe that in the 0th timestep (after no actions) the model has in fact learned to segment the state space into the different MNIST digits based only on reward. This suggests that there is on some level sufficient signal in this architecture to learn simple feature-spaces. Even the 4s, 5s, and 6s are relatively segmented and separated from each other, implying that the latent consistency loss is propagating signal to states that do not directly encounter rewards. We can also note that this latent structure is mostly preserved for the two subsequent timesteps (after 1 and 2 actions), though some morphing occurs (the 5s spread out across most other classes, suggesting that its transition model is poorly-learned). Finally, in the last timestep, the latent space appears to collapse, with the overall classes shearing into multiple parts (despite the latent-consistency loss that should be pushing them together) and generally melding with other classes. The only classes that are clearly identifiable are the 0s and perhaps 1s, which is what we'd expect — after all, 0s are the goal-

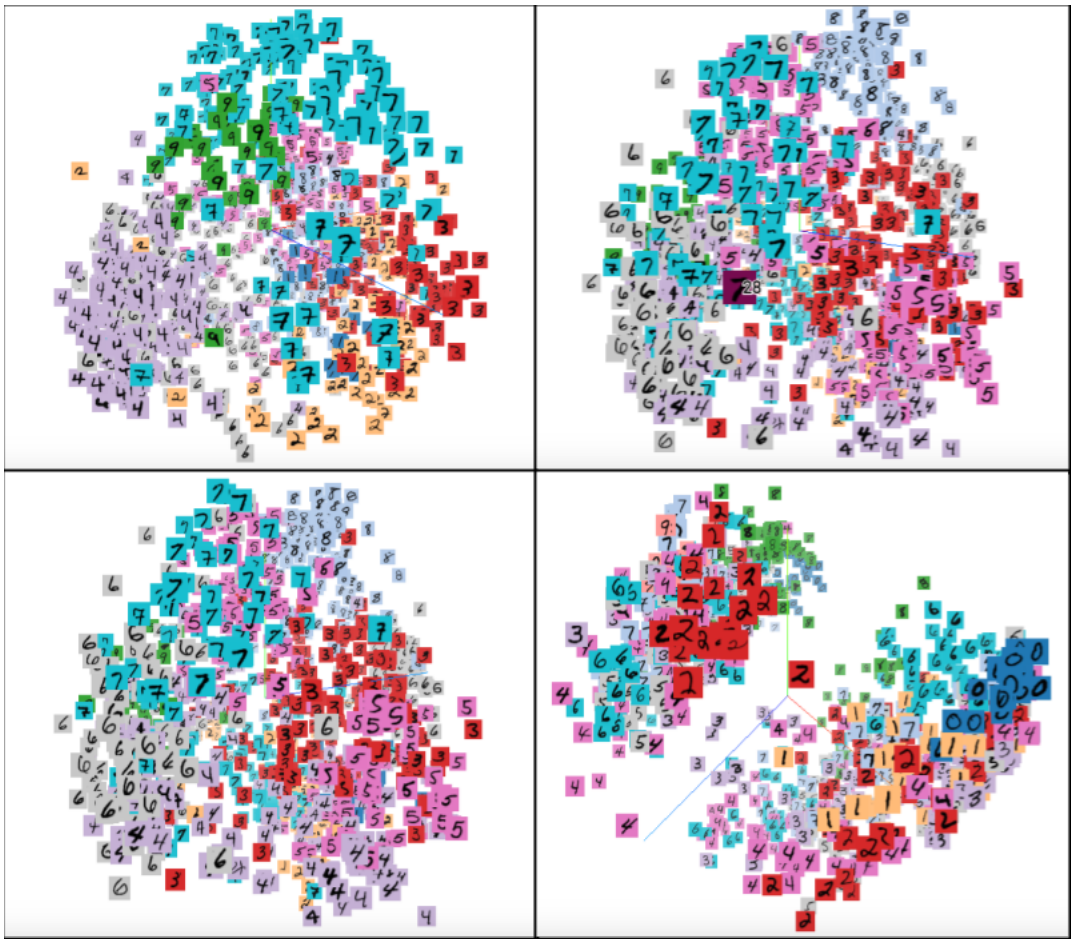


Figure 3-4: The post-PCA embedding visualization of a trained MNIST-linear learner (each class is marked a different color, though the learner is never provided the images' true classes). Top left is the set of initial latent states, top right the set of latent states computed by simulating the latent state forward by one action, bottom left is after simulating forward 2 actions, and bottom right is after 3 actions. (Note that the bottom right's group colorings are shifted by 1, so that e.g. red corresponds to $3 \rightarrow 2$.)

states, and the learner's primary objective is to accurately predict whether a goal has occurred at exactly a 3-step horizon. The latent consistency loss proves secondary at this final timestep.

We can watch this particular evolution more granularly by looking at the embedding space of just two classes, 3 and 4, in Fig. 3-5. We can immediately observe that the classes begin segmented, and slowly become less separated until they merge completely in the final frame. This makes sense — the classes are only separable by reward prediction in the 0th timestep (because three "-1" actions would bring a

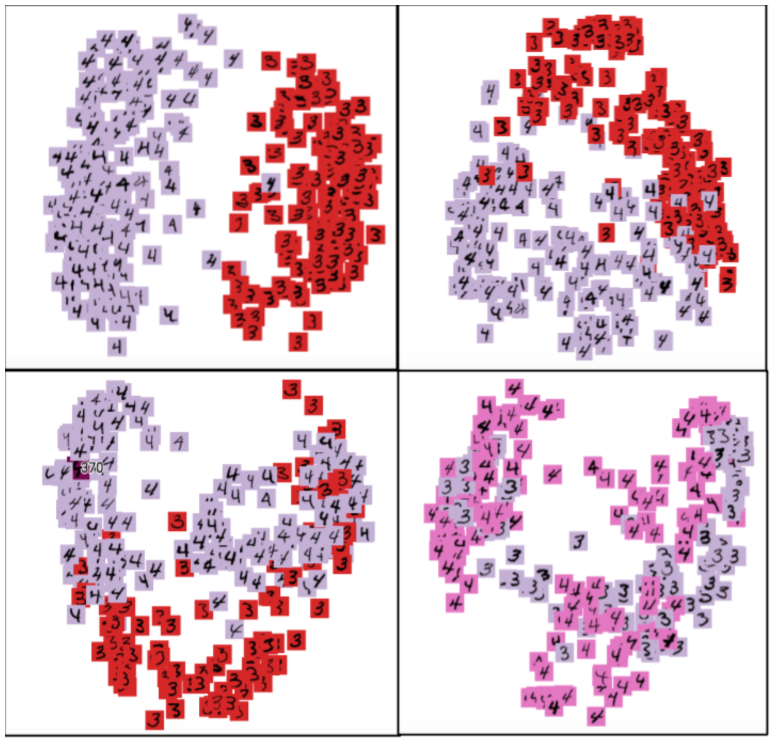
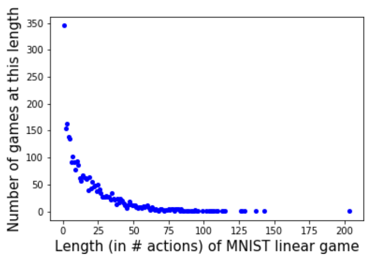


Figure 3-5: The post-PCA embedding visualization of two classes (3 and 4 from a 3-step-subsequence linear-MNIST model. Top left is after 0 timesteps, top right is after 1 timestep, bottom left is after 2 timesteps, and bottom right is after 3 timesteps. (Note that the bottom right's group colorings are shifted.)

3 to a goal state, but not a 4). It is interesting that in later timesteps the clusters become multimodal, for example in the second timestep there are two clusterings of 4s. While this might be exacerbated by the particular coordinate-shift of the PCA, it is still notable that the latent consistency loss — which should be pushing all 4s to the same location in latent space — is not bridging between these two groups.

In subsequent sections, we will explore the performance of the model across multiple timesteps. It is useful to understand, on average, how many timesteps appear in a given episode. This graph for the linear-MNIST environment can be seen in Fig. 3-6. We can see that the "linear" variant tends to have slightly shorter games than the "complex" variant.



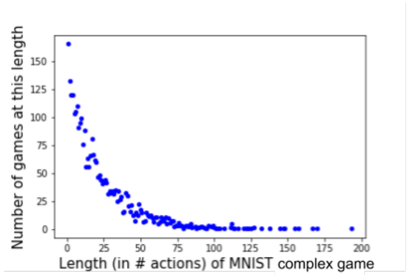


Figure 3-6: Lengths of the 3000 episodes of a random agent acting in the **(left)** linear-MNIST and **(right)** complex-MNIST environments that were used to evaluate the models.

3.5.2 Varying the Latent Consistency Loss

To probe the usefulness of the latent consistency loss, we retrained the model with all the same parameters, except this time removing that loss ($\lambda_C = 0$). The embeddings for the first few timesteps are shown in Fig. 3-7. We can observe the latent states increasingly collapse into each other almost immediately after the first timestep, reflecting the fact that the model has failed to learn a transition model for the states which cannot reach the goal within the remaining steps (out of 3). However, remarkably, the initial latent space (at the 0th timestep) appears much better segmented than the one with the latent consistency loss. That might make sense for the classes distinguishable by differing reward: there is no longer a latent-consistency loss pushing various latent representations together, so the model optimizes the latent space more directly for reward prediction, and thus identifies the classes better. However, the truly strange result is that the 4s, 5s, and 6s are almost perfectly segmented from each other as well, despite having no observable difference in reward. This seems to be a property of the MNIST dataset: sufficient training to identify most of the MNIST digits, without signal regarding the other digits, causes the neural network to automatically segment the other digits. [9]

To further explore the usefulness of the latent consistency loss (or lack thereof), we ran a series of experiments varying the latent consistency parameter λ_C between

[9]That is, the network learns that "the best way to identify which digit isn't a 4, 5, or 6 is to identify 4s, 5s, and 6s, and then rule them out. We separately validated this phenomenon by training a simple MNIST classifier with the labels for $4, 5$, and 6 all set to 4. In the resulting embedding space, the 5s and 6s separated from the 4s despite no labels separating them.

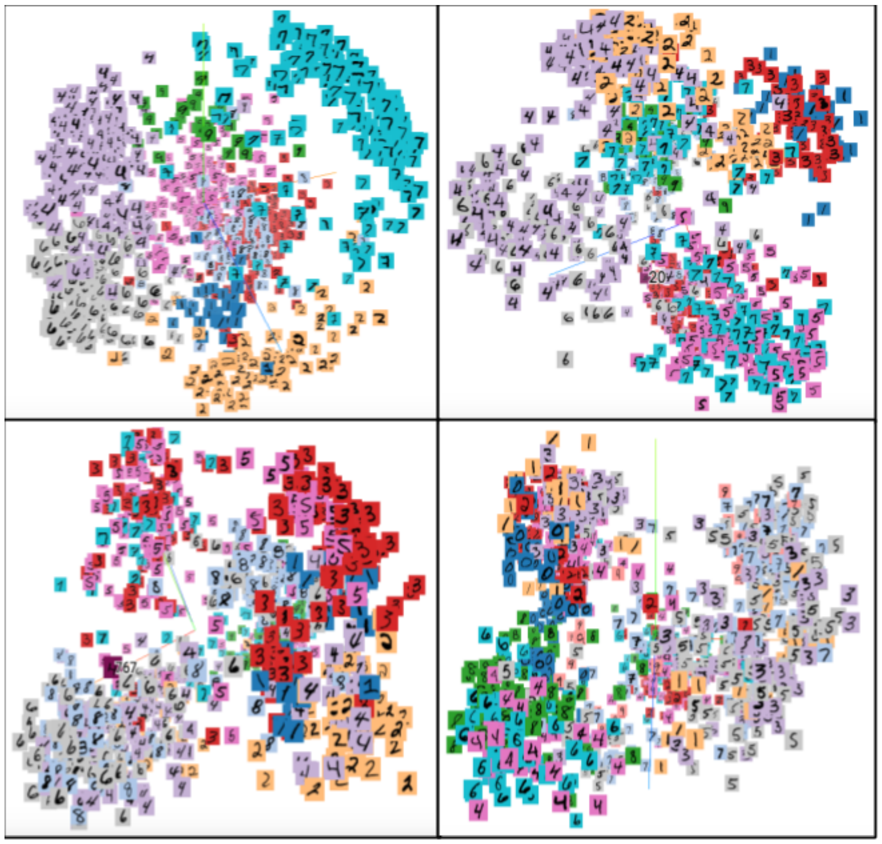
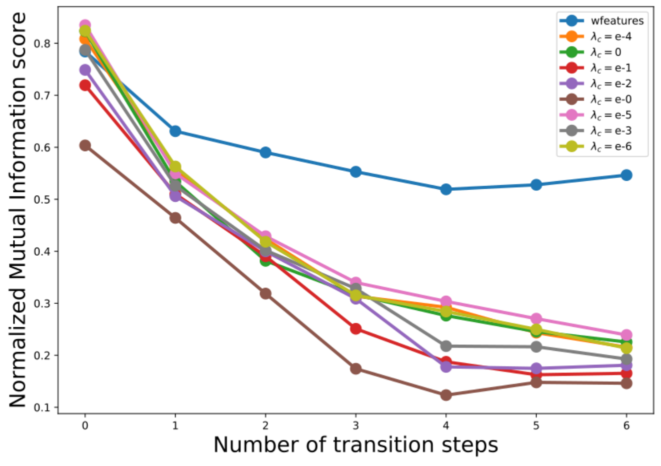


Figure 3-7: The post-PCA embedding visualization from a 3-step-subsequence linear-MNIST model with no latent consistency loss. Top left is after 0 timesteps, top right is after 1 timestep, bottom left is after 2 timesteps, and bottom right is after 3 timesteps. (Note that the bottom right's group colorings are shifted.)

1 and 10^{-6}. Each run learned from 3-step action sequences, and used goal-boosting with $\lambda_{GB} = 0.5$. The results are displayed in Fig. 3-8. From the bottom figure, we can observe that all of our models have very accurate reward-consistency losses for the first 3 steps (corresponding to the length of sequences they were trained on). Afterwards, however, the quality of the reward predictions decreases substantially — with the models with higher latent consistency showing slightly better generalization to more timesteps.

On the top of Fig 3-8, as we'd expect, the quality of the clustering of the data decreases as the number of transitions increases. Interestingly, as the latent consistency parameter decreases, the quality of the clusters increases. This makes sense if "latent consistency" and "goal-prediction" are competing objectives, and improving



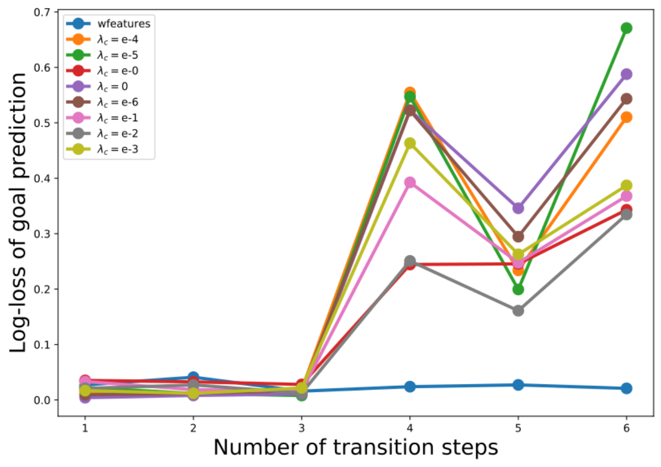


Figure 3-8: **(Top)** The Normalized Mutual Information score for the clusterings of the latent state representations for each timestep in a 3-step 1-transition-layer model of the linear-MNIST environment, varying the latent consistency scalar λ_C. **(Bottom)** The goal-prediction cross entropy for the same models. We compare these models with the "wfeatures" model, which can be thought of as a benchmark on the upper limit of this architecture's representation capacity. It is trained for 8 timesteps and is asked during training to predict the true state label; for further details, see Chapter 4.

on latent consistency comes at the cost of short-term latent space structure. However, increasing the latent consistency loss fails to improve the overall latent segmentation even in later timesteps. We repeated the experiment with a larger transition model (3 stacked layers in the RNN rather than 1) and only 2 transitions per training step, and observed the same results (see Fig. 3-9).

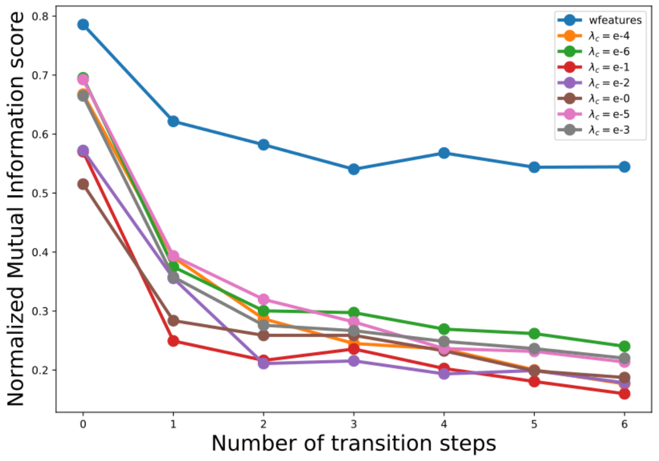


Figure 3-9: The Normalized Mutual Information score of the clusterings of the latent state representations for each timestep in a 2-step 3-transition-layer model of the linear-MNIST environment. We compare these methods with a benchmark "wfeatures" algorithm, which can be thought of as a benchmark on the upper limit of this architecture's representation capacity.

To better understand this phenomenon, we ran a different experiment, again in the MNIST linear environment and with a 3-layer RNN transition model trained on 2-step training sequences. This time, we evaluated the latent-space of the models (NMI score) only on sequences that began at state 5. In this case, because 5 is too far away from the goal 0 and the algorithm was only ever trained on 2-step sequences, the algorithm must construct latent representations from the latent consistency loss (as 3, 4, 5, 6, and 7 appear the same using the reward prediction loss alone). The results are in Fig. 3-10. Exactly as we'd expect, the runs with higher latent consistency scalars perform dramatically better at identifying and segmenting the latent states far from the goal. The high-latent-consistency model's performance does seem to decay beyond two timesteps for which the model was trained, suggesting a lack of generalization to longer horizons (which is similarly observable in Figs. 3-8 and 3-9). Overall, this experiment does seem to offer an explanation for the worse latent-segmentation performance of models with higher latent-consistency. The models optimizing only reward-prediction had good latent-space segmentation for a few of the states, and

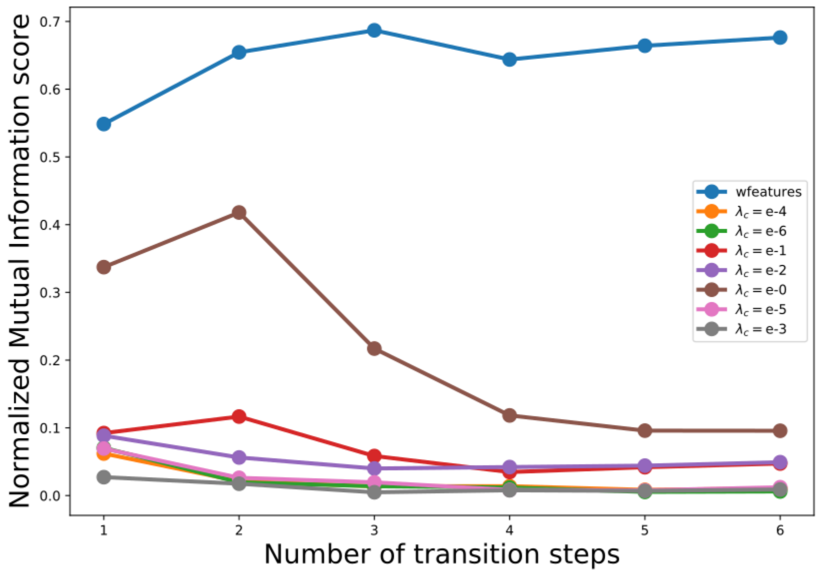


Figure 3-10: The Normalized Mutual Information score of the clusterings of the latent state representations for each timestep, with sequences always starting at state 5, for a 2-step training sequence 3-transition-layer model of the linear-MNIST environment with varying λ_C. Note that step 0 is omitted because the class is always the same. We compare these methods with a benchmark "wfeatures" algorithm, which can be thought of as a benchmark on the upper limit of this architecture's representation capacity. It is trained for 8 timesteps with perfect state features; for further details, see Chapter 4.

very poor segmentation for others (e.g. state 5), while the models with higher latent-consistency had moderately good latent segmentation across all states.

3.5.3 Varying the Training Sequence Length

In a separate set of experiments on the both the linear and complex MNIST environments, we varied the length of action sequences provided to the model during training. Each model had a 1-layer RNN transition model, a latent consistency scalar $\lambda_C = 1$, and no goal-boosting $\lambda_{GB} = 0$. The results are shown in Fig. 3-11. From the top figures, we can clearly observe that increasing the training horizon length improves the quality of the latent-space structure, even far beyond the training horizon. In the linear MNIST case (left) there is notably a substantial improvement after 5 timesteps

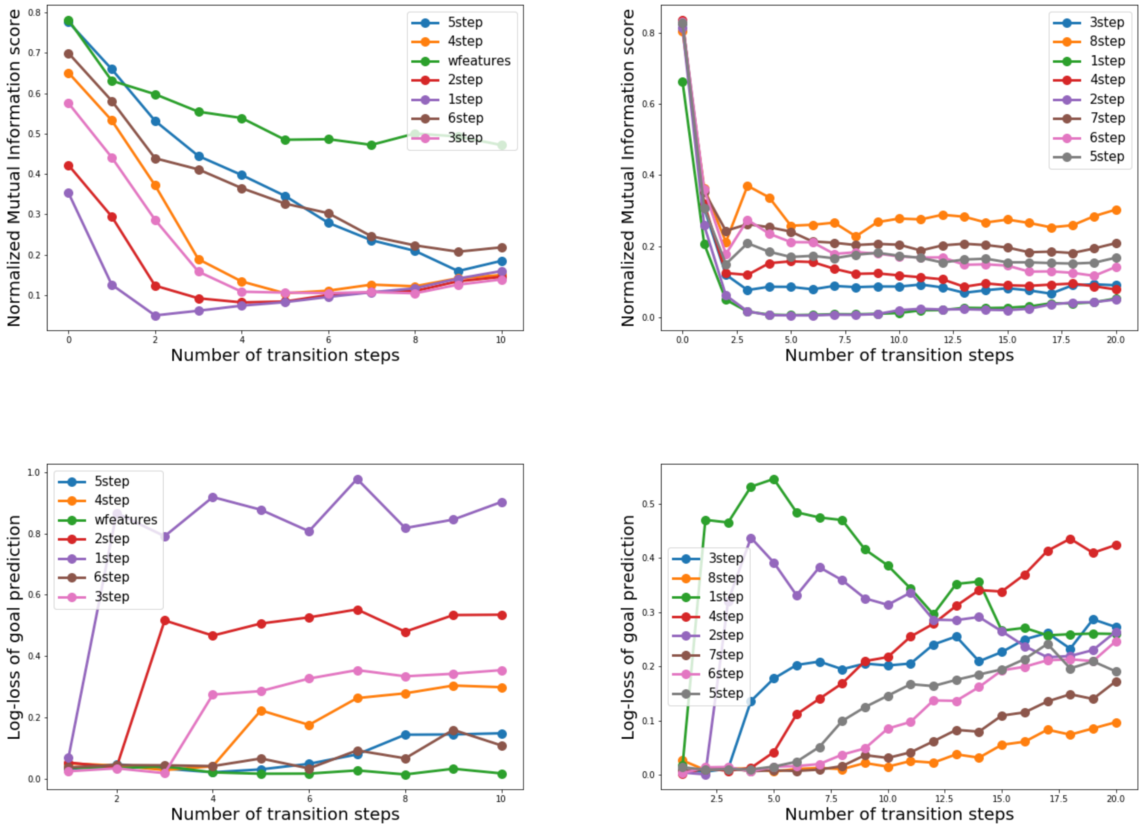


Figure 3-11: The **(top)** latent clusterings' Normalized Mutual Information scores and **(bottom)** goal-prediction cross entropy for each timestep in a 1-transition-layer model of the **(left)** linear-MNIST environment and **(right)** complex-MNIST environment, varying the length of training sequences shown to the algorithm.

or greater. This is precisely the sequence length necessary to possibly reach a goal from every state, which means that the latent consistency loss is no longer principally to segment the space. Oddly, over time, all the linear-MNIST models appear to converge to roughly the same quality of latent space (though this does not imply they have the same predictive quality, per the bottom of the figure, so perhaps this is a shortcoming of our clustering + NMI-score measurement approach).

In the bottom of Fig. 3-11, we can see that the models' ability to generalize to longer-horizon reward prediction substantially improves with each extra training transition. This makes sense: as the model is forced to predict latent representations and goals farther away from the original image, it must improve its transition model (which is repeated a greater number of times). It seems that for the linear environment, the goal-prediction quality saturates around 5- or 6-step sequences, which as previously suggested is the the length at which all states can reach a goal. In the complex-transitions environment, adding longer sequences continues to yield benefits for longer horizons even up to 8 transition steps, implying that none of these models have completely "solved" a stable structure for the environment. The fact that most of the models have a converging long-term reward loss might be a result of these models' eventually breaking down and predicting a constant likelihood for the goal.

3.5.4 Varying the Goal-Boosting Parameter

Finally, we ran an experiment on the linear-MNIST environment in which we varied the training sequence length with and without the goal-boosting loss ($\lambda_{BG} = \{0, \frac{1}{2}\}$). The results are visible in Fig. 3-12. The primary takeaway from the different models'

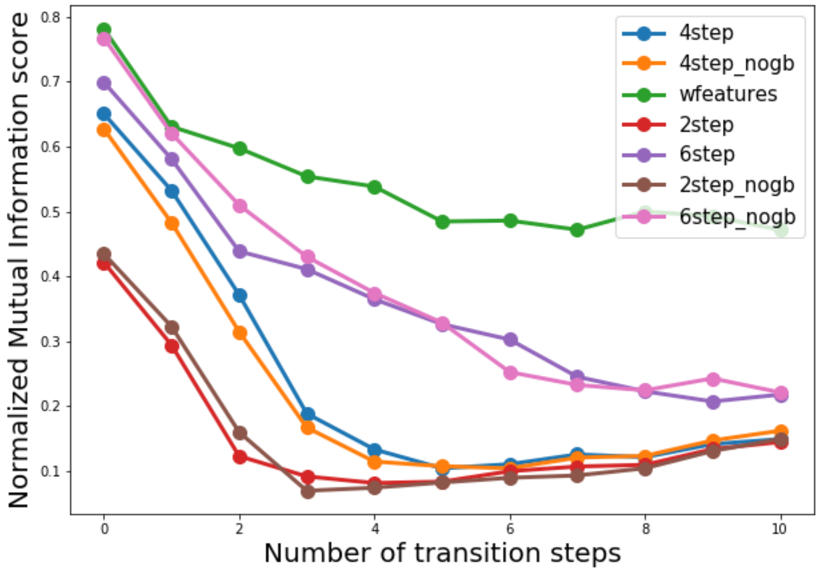


Figure 3-12: The Normalized Mutual Information score for the clusterings of the latent state representations for each timestep in a 1-transition-layer model of the linear-MNIST environment, varying the gradient-boosting as well as the length of training sequences. "wfeatures" is trained separately, and can be seen as a benchmark for the optimal model performance given this architecture.

SMI scores seems to be that empirically, goal-boosting does not appreciably affect the performance of the model. This implies that despite the rareness of goal occurrences, at least for short-horizon problems, goal-boosting is not necessary to properly predict goal-states.

3.5.5 Planning Performance

Finally, we tested our models, with varying training sequence lengths, no goal-boosting, and a single transition-model layer, by using them to plan for 500 real episodes (following the procedure from section 3.4.4). The results of this evaluation can be seen in Fig. 3-13. We can see that, for both environments, our models require relatively

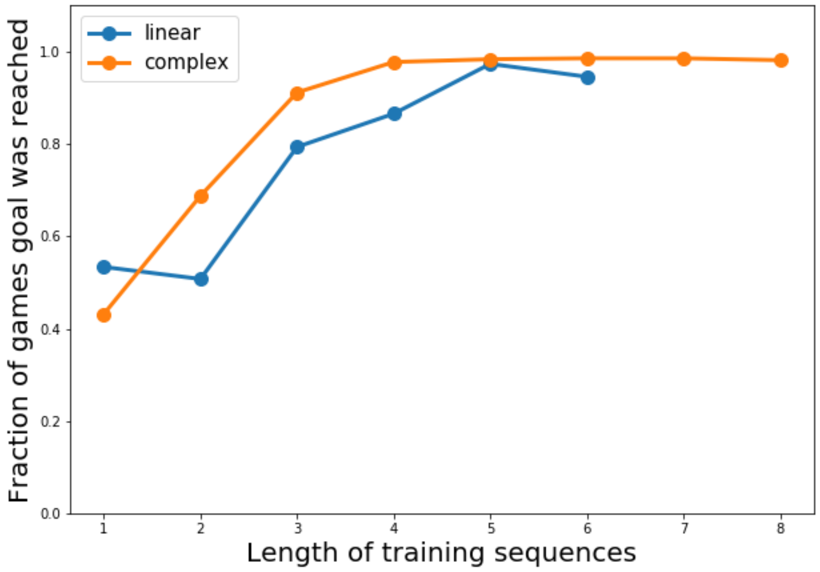


Figure 3-13: The fraction of games (out of 500) won by a simple planning agent using a 5-step BFS to choose actions according to models trained with varying sequence lengths.

short training sequences to consistently generate successful 5-step plans. This seems to conflict with the earlier experiments showing poor reward-prediction loss for all but the longest horizons. However, in planning, the important thing is that "goal" states be called goal states more often than non-goal states. Even from short training trajectories, the models seem to be able to predict this quality well enough to plan. It's also interesting that the models seem to have an *easier* time planning in the "complex" environment than in the "linear" one, despite the former's more complicated transition dynamics. This might be because no state in the "complex" variant is never more than 3 steps away from a goal, whereas the much simpler linear environment has states 4 and 5 steps away.

3.6 Discussion

We have described a method for learning a model to predict reward given historical environment experiences. As our experiments have shown, the model is capable of accurately learning state representations and reward dynamics for simple games. The performance of the model degrades in states farther from the goal state, but improves when training sequences get longer. The latent consistency loss also helps to learn representations for states that are farther away from goals, at the cost of accurate reward-prediction closer to goals.

One major problem appears to be the sparsity of goals, and the extent to which the model breaks down farther from them. This problem, of signal sparsity, is inherent to the "reward-prediction-only" model. However, one possible step forward is to relax the problem setting. Rather than predicting exclusively the reward, predict the reward as well as a small set of "features", which are provided by the trainer and occur more frequently than the goals themselves. For more on this approach, see Ch. 4.

One might consider whether mean-squared error is the wrong form for the latent consistency loss. This entire approach has modelled latent states as points in $\mathcal{R}^n$ (where n is the latent-space dimensionality), but real-world planning problems often contain more discrete attributes. We briefly experimented with forcing the latent states to take values between -1 and 1, but the loss in representational capacity substantially hindered our model. Changing the means of state representation, and the accompanying (also currently continuous) transition function structure, may be a promising direction to improve on the model.

Another quirk of the model as it currently exists is that the transition model does not inherently learn the structure of its predictions. That is, if we were to learn a simple physics simulator environment for 2 bodies, and then change the environment to 3 bodies, no knowledge would carry over. This is because the weights in the transition model are not tied, so the transition dynamics are learned entirely separately for each component of the latent space. To address this issue, one can replace the fully-connected layers of the transition model with convolutional layers (like in the ConvLSTM framework [35]). As an example, Oh et al. [26] use a convolutional transition model to learn the dynamics of an agent moving through a maze.

Though we have restricted ourselves to fully-observable environments, we can without too much complication extend our approach to certain partially-observable envronments. If we replace the encoder E, which is currently a CNN, with an RNN with convolutional layers before each timestep's input, we could use multiple observations at different timesteps to synthesize the initial latent state x_t. Similarly, when predicting future latent states x_{t+k}, the network can use the transition model T to propagate partially-observed information forward. However, if a sufficient state representation cannot be obtained from the preceding few observations, the problem becomes harder. In that case, we cannot apply the latent consistency loss to equate two states if there is uncertainty about whether they're the same state. Similarly, we could not assume determinism in the results of our actions if we were unsure of our current state.

Similarly, relaxing the assumption of deterministic transitions requires a more complicated model. The transition model itself must be stochastic in that case, yielding multiple future latent representations given a single original latent representation and action. Using a generative adversarial framework [13] could work, but would be difficult to train. GANs are, as of the time of writing, notoriously unstable, and this GAN would need to synthesize latent states which are themselves shifting and being learned — not to mention the fact that even minor discrepancies in learned probabilities of transitions would propagate problematically to future states. Recent work by Esteban et al. [12] on recurrent conditional GANs shows that there is hope that such models might become tractable, but also serves to illustrate the difficulty of training such a complicated network.

Chapter 4

Learning Environment Models with Partial Features

4.1 Motivation

As we saw in Chapter 3, learning environment dynamics using only reward prediction can be useful for simple environments. However, as rewards grow sparser and environments more complicated, the usefulness of the pure-reward approach decreases, and we must seek alternatives. That said, we would still like to maintain our earlier paradigm of training a model to learn a latent representation of the environment by predicting sparse, known-to-be-important signals. Until now, the only such signal we've considered is the reward. But often times we can find many real-world metrics that are relevant to our desired latent state representation, and which we could use as signals in training. Consider that a robot has available its task-space configuration, or rangefinding data for its proximity to other objects. In both cases, a proper model of the environment should be able to predict those quantities, even if they are only tangentially related to the final goal (e.g. picking up a particular object).

To this end, we introduce a new framework for environment learning. We use these external metrics, hereby referred to as "features", as a supervision signal which our model should predict in addition to reward, so as to construct our latent space.

This setup can be understood as a sort of multi-task learning problem. As has been shown extensively [2], training a system to predict multiple related tasks jointly can yield better predictions for each task than if each task were being predicted separately. This is because by optimizing across multiple related objectives, the learner is less likely to get stuck at a local minimum for a particular task. It will instead tend to find a latent representation that is consistent with all the tasks (and thus for related tasks more general). In our setup, the only task that truly matters is reward prediction. But by jointly learning to predict a set of important, task-relevant features, our hope is to impose beneficial structure on the latent space and thus improve our models' reward prediction performance. At the same time, our model can still learn a simplified state representation, so long as the environment is "reward-and-features-reducible" (an extension of "reward-reducibility" from Chapter 1).

We will still learn a latent environment representation x, and will now also learn a "featurer" $F(x)$ which aims to predict features f given the latent state. One might reasonably ask why we are not simply using the features f as our latent state (instead of using an intermediate representation x), and instead having x be sufficient to predict f. The feature-as-latent-space approach is used by Villegas et al. [31], which uses human body pose as the latent space, simulates it forward in time, and then converts it into an image. (Their task is video prediction; in our case, we would use it to compute the reward.) However, using the feature space as the latent space has a couple fundamental limitations. On the one hand, the hand-designed features f might not be sufficient to predict reward, making it important for the latent state to contain other information related to reward. On the other hand, hand-chosen features f_i might not even contain enough information content to predict the value of those same features f_{t+j} in the future. By allowing for a separate latent state that must contain both sufficient information to predict features, as well as any other information the learner determines is necessary to predict future rewards/features, the system can be applied to a broader range of environments with less effort to hand-design features.

4.2 Reward-and-Feature Prediction

4.2.1 Problem Statement

Our goal is to learn the reward dynamics for a particular environment. We are given n historical episodes, including observations $\{o_0, \dots, o_T\}_n$, actions $\{a_0, \dots, a_T\}_n$, and rewards $\{g_1, \dots, g_T\}_n$. We are also given one of the following: either corresponding lists of features $\{f_0, \dots, f_T\}_n$, or a "feature-extractor" function $H(o_i) \to f_i$ that can extract features from observations on demand, and can be used to generate the lists of features. Each f is a vector containing multiple individual features, and each feature can take one of a number of discrete classes (though this can easily be extended to the continuous-feature case).

The environment in question must be fully observable and have deterministic transitions (see Sec. 1.2), but can have stochastic observations and partial-state rewards. We must also place a certain restriction on the type of features f used in supervision to make the problem tractable. Features must be one of the following:

- The feature is fully computable from the observation itself (there exists a function $H(o)$ that can compute the feature)

- The feature is partially computable from the current observation, and with the incomputable part not excessively noisy, or

- The feature's initial value is incomputable from the initial state, but the feature's evolution is fully computable from the current observation. For example, given a robot's starting GPS location and heading, one can predict from a video-feed how that location-signal is evolving, even though the robot could not determine the GPS location from a video feed alone. (This requires changing the training architecture slightly to accept initial feature-values at the beginning of a training sequence, and we will not analyze this case.)

4.2.2 Model

As in Chapter 3, our model has an encoder E, a transition model T, and a goal-extractor G, whose function is detailed in Eq. 3.1, and reproduced below:

$$x_t^i = E(o_t^i)$$

$$\widehat{x_{t+j|t}}^i = T(x_t^i, a_t^i, a_{t+1}^i, \ldots, a_{t+j}^i) \quad , \quad \widehat{x_{t|t}}^i = x_t^i$$

$$\widehat{g_{t+j|t}}^i = G(\widehat{x_{t+j|t}}^i)$$

We now add a new model component, the "featurer" F, which extracts features from the latent state:

$$\widehat{f_{t+j|t}}^i = F(\widehat{x_{t+j|j}}^i) \tag{4.1}$$

We also define a new loss:

$$\mathcal{L}_F = \frac{1}{Z_3} \sum_{i=1}^{n} \sum_{h=1}^{h_{max}} \sum_{t=0}^{T^{(i)}-h} \sum_{j=0}^{h} \sum_{m=0}^{M} \sum_{c=0}^{c_{max}} -I\left(f_{m,c,t+j}^{(i)}\right) \log \widehat{f_{m,c,t+j|t}}^{(i)} \tag{4.2}$$

where $f_{m,c,t+j}^{(i)}$ is the cth possible class of the mth component of the $(t+j)$th-timestep feature vector from the ith episode, and $I(x)$ is the one-hot encoding of x (along the axis of c). M is the total number of feature components in each feature vector f, c_{max} is the total number of classes each feature could be in, h_{max} is the maximum training sequence length, $T^{(i)}$ is the length of the ith episode, and Z_3 is the number of terms in the loss (where each value of the f vector is considered a separate term). We combine this feature loss with the previously defined losses to create a new overall objective:

$$\mathcal{L} = \lambda_G \mathcal{L}_G + \lambda_C \mathcal{L}_C + \lambda_F \mathcal{L}_F \tag{4.3}$$

where λ_C, λ_G, and λ_F are scalars modifying the relative importance of the different losses.

Note that once training is complete, and the model is being executed, we no longer use its feature prediction capability F — and only compute the estimated reward g.

4.3 Methodology

4.3.1 Architecture

The models for E, T, and G are identical to those presented in Section 3.4.1. The new component, the "featurer" F, is a two-layer neural network. The first layer contains 256 hidden nodes followed by an Exponential Linear Unit nonlinearity [9]. The second layer has as many output nodes as the number of distinct features f times the number of different classes each of those features could take (we assume that features are in a discrete class, though it is simple to relax this constraint). Each set of outputs corresponding to classes of a feature component is then passed through a softmax function to normalize the probabilities. These output probabilities are the estimated likelihood of each feature's true class.

4.3.2 Environments

One set of experiments will be done on a slightly-modified version of the complex variant of the MNIST game described in Chapters 1 and 3. The difference is that now we will also provide the model-training algorithm with different types of features relating to the environment's true state. Specifically, we create variants of the game that provide the trainer with the current true digit of the game $\mod n$, where $n \in \{2, 3, 4, 5, 10\}$. This allows us to probe the effect of different amounts of feature information and their effect on the model's ability to learn an accurate latent space.

Our second set of environments is a pair of new, more complicated environments, called the "increment game" and the "rotation game" (visualized in Fig. 4-1). Both consist of 3-by-2 grids of MNIST digit images ranging from 0 to 2, and have one action for each column and each row. However, their transition mechanics and goal-states are entirely different. In the increment game, taking an action a corresponding to a row/column increments each digit in that row/column by 1 ($\mod 2$). The initial state is generated by applying 5 random actions to a gamestate with all 0s. The game's objective is to make every tile display a 0.

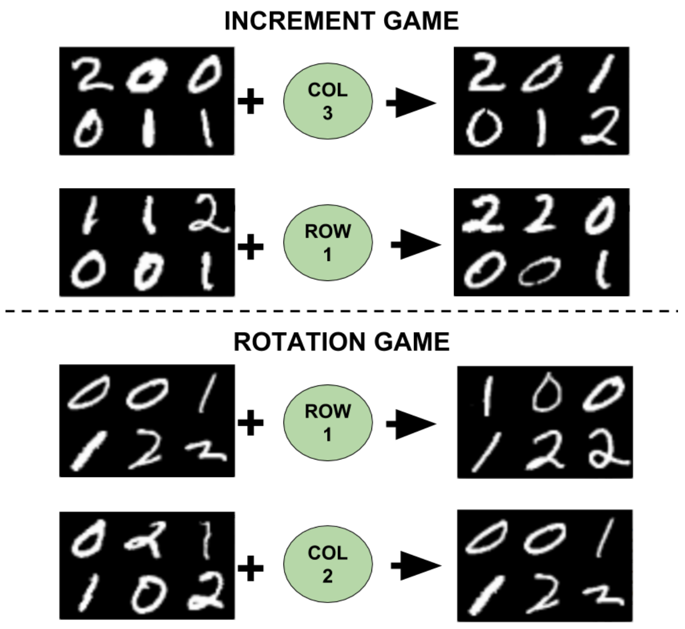


Figure 4-1: A graph demonstrating example transitions from a pair of environments. On the top, an example of increment game transitions, in which the 3rd column is incremented and in which the 1st row is incremented. On the bottom, an example of rotation game transitions, in which the 1st row is rotated and in which the 2nd column is rotated.

In the rotation game, taking an action a corresponding to a row/column rotates the digits in that row/column down/to the right by 1. For examples, see Fig. 4-1. The goal is to have the 0s on the left, the 1s in the center, and the 2s on the right. The initial state is generated by applying 5 random actions to a goal state (reversed, so that rows are left-shifted instead of right-shifted).

These environments provide interesting new attributes. Their state-spaces are much more complicated than the MNIST game's, with about 100 different states rather than 10, making memorizing the entire transition graph harder. They also have more-structured transition functions, in which subcomponents of the latent state are separately manipulated in numerous ways. The two environments also differ from each other in significant ways. Each subcomponent (digit) of the increment game is only dependent on that same component in the previous timestep, whereas in the rotation game the value of a digit in the next timestep depends on a different digit elsewhere in the state. In the increment game, every achievable state (including the goal) is reachable from every other state in at most 10 moves — whereas in the rotation game, the true gamestate can get substantially farther from a goal state.

We create variants of the increment and rotation games with different amounts of feature info. For each of the two environments, we include one variant with every digit as a separate feature, one variant with only the 4 rightmost digits as features, one environment with only the 2 rightmost digits as features, and one environment with no features at all. Comparing the ability to learn these variants can provide us insight on how important the density of features is to learning more complicated environments.

4.3.3 Metrics

We use the same metrics as in Chapter 3, with a couple modifications and one addition. First, for the increment and rotation games, the clustering-metric is no longer relevant since there are too many distinct states, and they are too complex, to expect KMeans to automatically segment them all. Similarly, when evaluating the planning algorithms' performance, we set up the initial states to have a guaranteed solution that is exactly 5 states long. We also introduce a new metric, the mean cross-entropy loss for feature prediction, that is the average cross-entropy loss for the prediction of each feature in f for a given group of predictions.

4.4 Experiments

4.4.1 MNIST with features

The first experiment explores the complex MNIST game environments with varying feature types. We train models with 3-action sequences, with $\lambda_F = \lambda_G = \lambda_C = 1$. The results of the training are visualized in Fig. 4-2. First, we can clearly see from the bottom left figure that providing partial state information improves the model's reward prediction ability in nearly all cases. In the top left, we see that the latent spaces learned by the models using $3, 4, 5$, and 10 classes are more-or-less equivalently

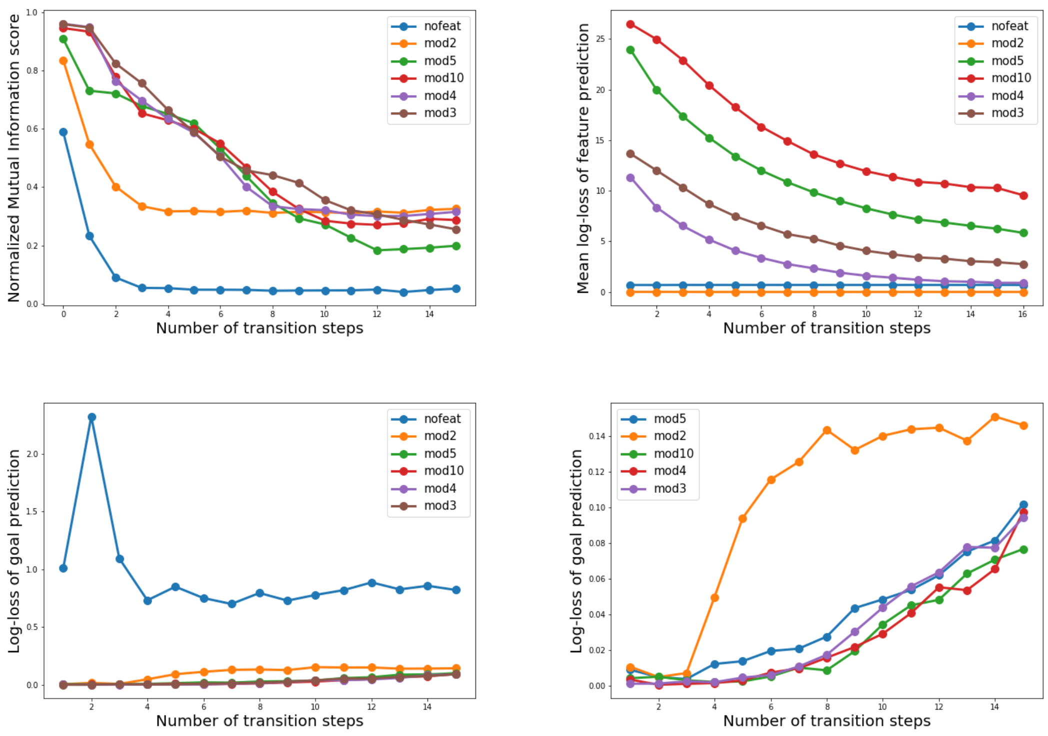


Figure 4-2: Results of the experiments learning models for the complex MNIST game environment, with 3-step training sequences and $\lambda_C = \lambda_F = \lambda_G = 1$. We vary the auxiliary features provided to the learner, which is always the true digit $\mod n$, where $n \in \{2, 3, 4, 5, 10\}$ (with the feature-blind model provided for comparison). **Top left:** NMI score of latent-space clusters generated by KMeans. **Top right:** Mean cross-entropy loss for feature prediction. **Bottom left:** Log-loss of reward prediction. **Bottom right:** Same as previous, but with featureless version omitted for clarity.

good, with 2 seemingly being not enough information to create a good a latent space structure, though still better than with no features at all. This suggests that with partial (though incomplete) information about the state (e.g. in the $\mod 3$ feature case), the model is capable of inferring the rest of the state and segmenting the classes as well as if it were given the full state.

We also see, in the top right, that the feature-prediction accuracy decreases as the feature complexity increases. This makes sense, as there are a greater number of features to accurately predict. On the other hand, from the bottom two charts, we can see that increasing feature complexity improves reward prediction up to a point

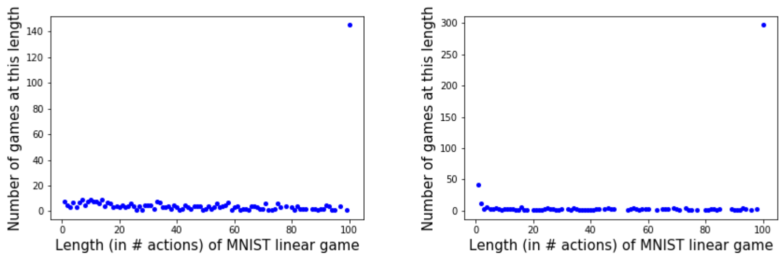


Figure 4-3: Frequency of game lengths out of 500 games used to evaluate the (**left**) increment and (**right**) rotation games.

(except for mod 5)[1]. This suggests that when choosing a set of features f, there is a tradeoff between increasing feature-complexity to induce more structure on the latent space, and decreasing feature complexity or scaling its relative importance via λ_F to reduce its importance relative to reward-prediction in the overall loss.

4.4.2 More complicated games

In our second set of experiments, we trained models on the variants of the increment and rotation game environments. We can see from Fig. 4-3, unlike in the MNIST game, these games tend to be considerably longer, to the point where we cut games off if they did not conclude after 100 actions. Each model was again given 3-action training sequences and $\lambda_C = \lambda_F = \lambda_G = 1$. The predictive results are evaluated in Fig. 4-4.

The first finding, in the top two figures, is that the models were much more capable of learning the feature representations for the increment game than they were for the rotation game, especially in the short term. However, the marginal usefulness of additional features in predicting those features (as shown in the variance in the feature losses) is much greater for the rotation game than the increment game. This makes sense, as the increment game's features are all independent of each other (and thus

[1]The feature-space mod 5 has an odd effect on the state space. It is particularly difficult to separate between e.g. a 1 and a 6, because adding 1 or multiplying by 2 would yield the same state mod 5. There is only one action, $x3$, that ever differentiates the two, and so the optimization is unsurprisingly prone to suboptimal local minima in a way the other partial feature representations were not. This raises the point that sometimes feature-choices can actually be misleading and trap the model in local optima themselves — though in this case it still outperforms the featureless model

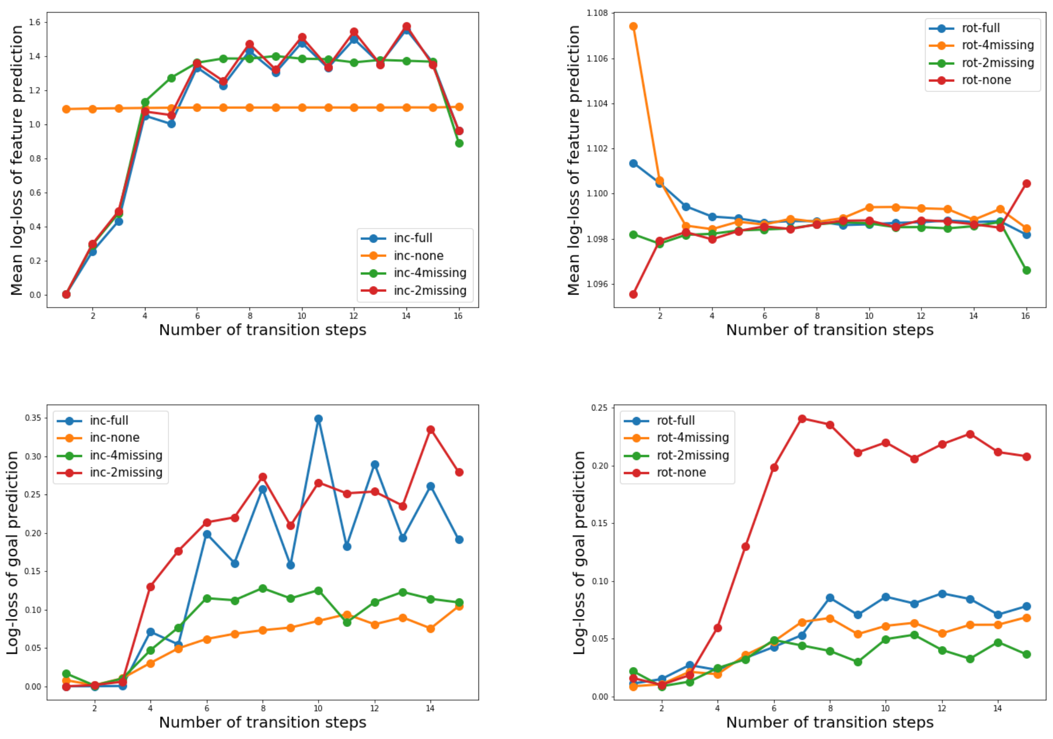


Figure 4-4: Results of experimenting with feature-trained models of the increment game and rotation game environments. Each was provided with 3-timestep training sequences, $\lambda_C = \lambda_F = \lambda_G = 1$, and with varying amounts of feature information: 'full' meaning all 6 state digits, '2missing' meaning only the 4 rightmost digits, '4missing' meaning only the 2 rightmost digits, or 'none' meaning no feature information. The **left** column shows results for the "increment game", while the **right** shows results for the "rotation game". **Top:** Mean cross-entropy loss of feature prediction. **Bottom:** Log loss of goal prediction.

it is no easier to predict previous features given new features) whereas the rotation game's features are interdependent and learning a more complete state representation helps in future feature prediction.

The other finding, in the bottom two figures, is that the goal prediction quality actually suffers with more features in the increment game, while using features significantly improved performance in the rotation game. This may again be a case of the earlier discussed tradeoff: improving the latent space to reduce the feature loss $\mathcal{L}_F$ can come at the cost of reducing the goal loss $\mathcal{L}_G$. (Perhaps fiddling with λ_F would fix this problem, though some tradeoff, so long as the model is not perfect, is inevitable) However, as can be seen in the rotation-game results, even if the model isn't able to learn to accurately predict features, the incorporation of features can still improve the model's ability to predict future goal-values.

In Fig. 4-5, we show the performance of the models for a 5-step planning problem. Interestingly, the rotation and increment games have completely opposite improvement patterns with increased number of features provided to the learner. The rotation game does appear to be substantially harder than the increment game. One reason for this might be that the order of actions in the increment game doesn't matter, whereas the order in the rotation game matters a great deal, and so there are fewer acceptable paths to take. Using our analysis of Fig. 4-4, we can make certain predictions about the cause of these results. The rotation game's goal-predictive capabilities improved with more features, leading to the overall increase in planning success, with the final "full-features" model slightly distracted with feature-prediction. The results from the other model are harder to explain, and further research is needed. Still, it may be partially attributable to the fact that the pure-reward-prediction model was good enough that additional features did not provide substantial leverage (until the final step, which learned a complete latent state, and thus was able to outperform the other models. This theory is borne out in a subjective analysis of the learned latent spaces).

4.5 Discussion

We have demonstrated an approach that can incorporate external features into model-learning for reward prediction. Incorporating external features can make certain reward prediction problems more tractable to learn, though it provides less benefit for other problems and can even worsen reward-prediction performance when it forms a competing objective. In many experiments, just a few features were sufficient to interpolate the rest of the latent state.

One particularly nice aspect of this approach is that it allows human practitioners direct influence over the model's learned state representation. Much time is spent proverbially shaking our fists at neural network architectures that do not extract

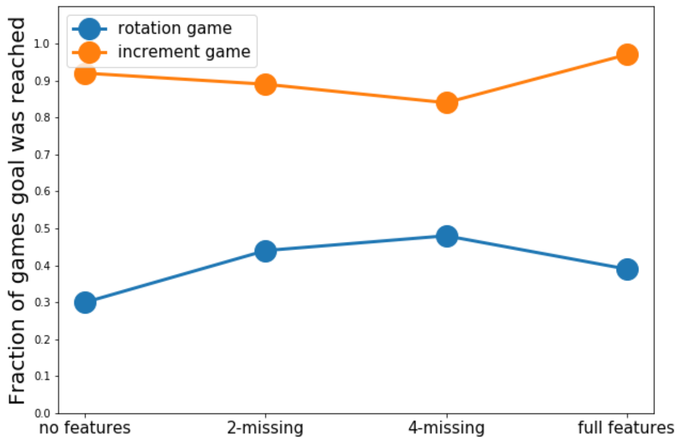


Figure 4-5: The goal-achievement frequency (evaluated on 100 games) for a 5-step BFS planner using models trained to imitate the "rotation" and "increment" environments. There is guaranteed to be a 5-step path to the goal for each initial state. Each model was trained with 3-timestep sequences, $\lambda_C = \lambda_F = \lambda_G = 1$, and with varying amounts of feature information: 'full features' meaning all 6 state digits, '2missing' meaning only the 4 rightmost digits, '4missing' meaning only the 2 rightmost digits, or 'no features' meaning no feature information at all.

the latent representations we were hoping, because we lack the ability to incorporate information into our networks that traditional programmers have over their programs. This approach allows us to both directly incentivize and directly inform the model to learn latent representations that contain information we deem important — while also leveraging the adaptive strength of machine learning to plug the holes in the featureset to improve the accuracy of the model.

One challenging question is how to properly evaluate models that are provided with external features. Clearly, extremely hard problems (like those in robotics) are sufficiently difficult under any setting that solving them with the aid of certain external features is meaningful. However, for simpler problems, it is hard to know whether success at a task is "impressive" or "trivial". The quantity of features that one can provide to the trainer is a sliding scale, and any problem becomes trivial when nearly the entire true state space is revealed to the algorithm. The relationship between feature-complexity and environment-learning-tractability is ultimately environment-dependent, but further study on the characterization of environments is needed.

Finally, generating the features themselves can be done in a number of ways. The most costly but most general way is through explicit annotation of each timestep in each episode, e.g. by a human. Alternatively, one can create a feature-extractor function (either using a neural network, or any traditional non-differentiable algorithm) to extract the features dynamically. A third approach is to generate the features automatically from the data in an unsupervised manner. We briefly discuss a number of methods for unsupervised feature generation in Appendix B.

Appendix A

Reward Prediction with Valid/Invalid Transition Classification

A.1 Encoding structure through classification

This work has centered around the question of how best to learn a latent representation of the world. As has previously been discussed, this boils down to the question: "what questions should the latent state be asked to encode?" In Chapter 3, the latent representation was required to encode enough information to guess the reward, and sufficient information to predict subsequent reward. However, in sparse-reward environments such an approach proves inadequate, and so in Chapter 4 we require the latent state to encode enough information to guess the reward *and other important info*, now and in the future. Here, we pose a potential new requirement: that the latent state encode enough information for a classifier to discern whether a proposed state transition is "valid" and truly occurred in the environment, or whether it is made up and thus invalid.

There are a few ways to set up such a classifier. In the first, the classifier $C_a(x_0, x_1)$ takes as input two states x_0 and x_1, and for each action $a_i \in A$ outputs the probability that $T(x_0, a_i) \to x_1$ is a valid transition.[1] This approach was used by Pathak et

[1]In the case of a multi-state transition, it may generate the probabilities for different sequences of actions.

al. [28] for feature extraction, but is fundamentally hobbled because it scales poorly with the number of actions/length of action sequences. The other approach is to train a classifier $C_b(x_1, \widehat{x_1})$ that is given an initial state x_0 and action a_0, simulates the state forward $\widehat{x_1} = T(x_0, a_0)$, and then predicts whether the resulting latent states are the same or different. An important ingredient here is that the environment be deterministic: the original "real" resulting state x_1 must be the only possible latent state resulting from the action a_0, so that simulating x_0 forward can be assumed to yield the same state. Otherwise, this approach easily scales with longer action sequences and can be readily incorporated into the previous chapters' prediction architectures. While it ultimately suffers from a fundamental flaw, we believe that it is worth examining it as an interesting alternative approach to environment learning.

A.2 Model

Here we formalize the transition classifier model. We are given a set of historical episodes of observations $o_1, \ldots, o_T$, actions $a_1, \ldots, a_{T-1}$, and goal-values $g_1, \ldots, g_T$, which we alternatively refer to as "valid" transition sequences S_t. We construct a set of "invalid" transition sequences $\{o_t, a_t, a_{t+1} \ldots, a_{t+k-1}, o_{t+k}\} = S_f$ by one of several methods which we'll discuss later. We define the latent state prediction architecture as in Eq. 3.1, reproduced here for convenience:

$$x_t^i = E(o_t^i) \tag{A.1}$$

$$\widehat{x_{t+j|t}}^i = T(x_t^i, a_t^i, a_{t+1}^i, \ldots, a_{t+j}^i) \quad , \quad \widehat{x_{t|t}}^i = x_t^i \tag{A.2}$$

We propose training a neural network classifier C to differentiate between valid and invalid transitions using the following loss function:

$$\mathcal{L}_V = \frac{1}{Z_4} \sum_{i \in \{S_t, S_f\}}^{n} \sum_{h=1}^{h_{max}} \sum_{t=0}^{T^{(i)}-h} \sum_{j=0}^{h} -I(i) \log C\left(x_{t+j}^{(i)}, \widehat{x_{t+j|t}}^{(i)}\right) \tag{A.3}$$

where $I(i)$ is an indicator variable which is 1 if the ith trajectory is "valid" and 0 if it is invalid, and Z_3 is equal to the number of log-loss terms in the sum. This loss can be added to e.g. the goal-prediction loss and latent consistency loss to yield an overall objective:

$$\mathcal{L} = \lambda_G \mathcal{L}_G + \lambda_C \mathcal{L}_C + \lambda_V \mathcal{L}_V \tag{A.4}$$

where λ_G, λ_C and λ_V are scalar constants. This allows the transition validity loss to impose a structure on the latent space, and hopefully in so doing improve the system's ability to predict rewards.

Note that this formulation is different from adversarial learning, e.g. as used in Mathieu et al. [21] to generate video. In the adversarial version, the classifier C serves as the opponent of the latent predictor. The classifier is shown "valid" transitions and predicted transitions, and is tasked with ferreting out the predicted (and thus fake) transitions. The latent predictor's adversarial objective is to fool the classifier, and in so doing it learns to predict latent states similar to the distribution of true latent states.

In this case, however, the classifier is actually cooperating with the latent predictor. The latent predicted state is **assumed to be true**, and is compared to the "valid" and "invalid" transitions. The classifier aims to differentiate the two groups, and in part does this by changing the latent representation (via the encoder) to contain more information that signifies the state as valid. Thus this approach does not require optimizing adversarial objectives or finding saddle points, and consists of a single fully differentiable objective.

The drawback of this approach lies in the one detail we've glossed over: how does one construct the set of invalid transition sequences S_f? This turns out to be the method's shortcoming. There appears to be no good general purpose scheme, but below we will discuss a few.

A.2.1 Mechanisms for Generating Invalid Transitions

We will describe each of the following mechanisms for the single-transition case, though they can be easily extended to multi-action sequences. The following approaches can also be combined, constructing subsets of the negative transition set S_f from the different approaches.

Drawing random states

The first and simplest way of creating invalid transitions is to simply take a real o_t and a_t, and then let the resulting state be a random state $\bar{o}_t$ drawn from the set of all previously observed states O_A. Jonschkowski et al. [17] used this approach to force far-apart states to have different representations. The approach will generate a false transition in many cases, although there is the possibility of randomly drawing a state which happened to be true. The likelihood of this occurring is directly proportional to the frequency that we've previously visited that state — so in the linear MNIST game it may be nearly 1 in 10 incorrect, while in environments with millions of states it is almost always correct). The major shortcoming of this actually the distribution of states that are likely to occur. In any environment where states are similar across time, a randomly drawn state taken from any moment in the past will likely be much farther away from the initial state x_0 than x_1 would be (e.g. video frames in Mario). The classifier therefore may fail to learn to differentiate between two potential future states x_1 and $\bar{x_1}$ which are both close to x_0, because it was rarely forced to identify almost-right-yet-wrong transitions.

Drawing negative examples from the same episode

We can generate a distribution of invalid result states x_1 that is more similar to the true state by specifically using the other states from the episode $S = S_{episode} \setminus \{x_1\}$. The rationale is that in many environments, states in the same episode are all mostly similar, but often not identical. A major drawback of this approach is that it assumes that state occurrences are to some extent order-independent. That is, states that are before or long after x_0 are similar to and representative of states that might occur immediately after (like x_1). As a simple counterexample environment, take checkers. Given a state of the board x_t and a proposed negative state $\bar{x_{t+1}}$ you can immediately rule out any state before x_t because it would have more pieces on the board/the pieces would be farther backward, without understanding anything about which pieces are moved by what action. Thus this approach only works for environments where the distribution of states is time-independent.

Drawing negative examples by sampling other actions

The last and most direct approach is to generate invalid transitions by sampling actions $a_i \in A \setminus a_0$ and using the results $\bar{x_1} = T_{real}(x_0, a_i)$ as the set of negative transitions. The distribution of resulting states is likely to be as similar to x_0 as x_1 would be, largely solving the shortcoming of the previous approaches. This does assume that two actions do not yield the same resulting state, though that is true in many cases. More problematically, however, this approaches requires the ability to simulate the environment, repeatedly, from an identical initial condition, and observing the results. All of our previously discussed methods for model-learning could be trained using historical real-world data, but this approach cannot. It is thus limited to the world of perfectly-simulatable games, for which learning an approximate simulator is somewhat less interesting.

Appendix B

Types of Features

In this appendix, we will briefly go over some of the types of semi-supervised features f_t that can be used to force the model to learn a non-trivial state representation s.

B.1 Rewards/Goals

If f is the reward/goal function of the relevant task, this in principle means that the system will learn the simplest state representation which correctly predicts the reward for a given task at any future timestep. In practice, rewards/goals are often sparse, and are insufficient to alone learn rich feature-spaces.

B.2 Observations as Features

Much of the work to-date done on environment simulators aims to learn states s_t that can fully reproduce that state's observation o_t. This feature is appealing because the signal is strong (successfully reproducing the observation almost certainly means capturing the true underlying state x), and is available to the agent's system using no further effort. Chiappa et al. [7] and Oh et al. [25] use the pixel-wise MSE loss to precisely predict future frames, while Mathieu et al. [21] proposes to predict frames that an adversarial learner cannot distinguish from a true frame. In general, learning to fully predict video is training-resource intensive, and current probabilistic generative modeling techniques are often unstable. This makes video prediction a challenge for environments with observation stochasticity. Further, because s must contain virtually all of x, such a procedure is overkill for environments with partial-state rewards, in which only a small portion of the true state is salient.

B.3 Action-predictive Features

A novel approach proposed by Pathak et al. [28] suggests self-supervised feature learning by predicting the action a_t from successive observations o_t and o_{t+1}. In other words, to train a classifier $A(\phi(o_t), \phi(o_{t+1}))$ to predict a_t, and then to use the learned embedding $\phi(\cdot)$ as the action-relevant observation features. This approach extracts only features of the environment that are affected by the agent's actions (if not, they would not be helpful in determining which action was executed). On the other hand, this approach fails to extract features not relevant to predicting the action which may nevertheless be important for modeling the environment (e.g. a long-term process which will only affect the agent many frames into the future). This can be at least partially remedied by forcing the representation to predict the reward r_t (e.g. as $R(\phi(o_t), \phi(o_{t+1})$ as well, to learn short-term features salient to the reward.

B.4 Reusing existing filters

One approach for representation-learning used in Neverova et al. [24] is to train a neural network to segment an image into different objects using supervised examples. Such segmentations can also be done by modifying simple trained CNN classifiers to be fully convolutional networks [20]. These scene segmentations can be used as "high level" features which still maintain much of the spatial structure of the original representation, and can be used as features for nearly-unsupervised learning.

Bibliography

[1] Pulkit Agrawal, Ashvin V Nair, Pieter Abbeel, Jitendra Malik, and Sergey Levine. Learning to poke by poking: Experiential learning of intuitive physics. In *Advances in Neural Information Processing Systems*, pages 5074–5082, 2016.

[2] Andreas Argyriou, Theodoros Evgeniou, and Massimiliano Pontil. Multi-task feature learning. In *Advances in neural information processing systems*, pages 41–48, 2007.

[3] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[4] Masataro Asai and Alex Fukunaga. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. *CoRR*, abs/1705.00154, 2017.

[5] Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, et al. Interaction networks for learning about objects, relations and physics. In *Advances in Neural Information Processing Systems*, pages 4502–4510, 2016.

[6] Michael B Chang, Tomer Ullman, Antonio Torralba, and Joshua B Tenenbaum. A compositional object-based approach to learning physical dynamics. *arXiv preprint arXiv:1612.00341*, 2016.

[7] Silvia Chiappa, Sébastien Racanière, Daan Wierstra, and Shakir Mohamed. Recurrent environment simulators. *CoRR*, abs/1704.02254, 2017.

[8] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[9] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[10] Emily Denton and Vighnesh Birodkar. Unsupervised learning of disentangled representations from video. *CoRR*, abs/1705.10915, 2017.

[11] Alexey Dosovitskiy and Vladlen Koltun. Learning to act by predicting the future. *CoRR*, abs/1611.01779, 2016.

[12] Cristóbal Esteban, Stephanie Hyland, and Gunnar Rätsch. Real-valued (medical) time series generation with recurrent conditional gans. 06 2017.

[13] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[14] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. In *International Conference on Machine Learning*, pages 2829–2838, 2016.

[15] Matthew Guzdial, Boyang Li, and Mark O. Riedl. Game engine learning from video. *International Joint Conference on Artificial Intelligence*, 2017.

[16] Rico Jonschkowski and Oliver Brock. State representation learning in robotics: Using prior knowledge about physical interaction.

[17] Rico Jonschkowski, Roland Hafner, Jonathan Scholz, and Martin A. Riedmiller. Pves: Position-velocity encoders for unsupervised learning of structured state representations. *CoRR*, abs/1705.09805, 2017.

[18] Peter Karkus, David Hsu, and Wee Sun Lee. Qmdp-net: Deep learning for planning under partial observability. *arXiv preprint arXiv:1703.06692*, 2017.

[19] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[20] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

[21] Michael Mathieu, Camille Couprie, and Yann LeCun. Deep multi-scale video prediction beyond mean square error. *arXiv preprint arXiv:1511.05440*, 2015.

[22] Aaron F McDaid, Derek Greene, and Neil Hurley. Normalized mutual information to evaluate overlapping community finding algorithms. *arXiv preprint arXiv:1110.2515*, 2011.

[23] Jelle Munk, Jens Kober, and Robert Babuška. Learning state representation for deep actor-critic control. In *Decision and Control (CDC), 2016 IEEE 55th Conference on*, pages 4667–4673. IEEE, 2016.

[24] Natalia Neverova, Pauline Luc, Camille Couprie, Jakob Verbeek, and Yann LeCun. Predicting deeper into the future of semantic segmentation. *arXiv preprint arXiv:1703.07684*, 2017.

[25] Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard L Lewis, and Satinder Singh. Action-conditional video prediction using deep networks in atari games. In *Advances in Neural Information Processing Systems*, pages 2863–2871, 2015.

[26] Junhyuk Oh, Satinder Singh, and Honglak Lee. Value prediction network. *arXiv preprint arXiv:1707.03497*, 2017.

[27] Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*, 2013.

[28] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. *CoRR*, abs/1705.05363, 2017.

[29] Ethan Perez, Harm de Vries, Florian Strub, Vincent Dumoulin, and Aaron Courville. Learning visual reasoning without strong priors. *arXiv preprint arXiv:1707.03017*, 2017.

[30] Nitish Srivastava, Elman Mansimov, and Ruslan Salakhudinov. Unsupervised learning of video representations using lstms. In *International Conference on Machine Learning*, pages 843–852, 2015.

[31] Ruben Villegas, Jimei Yang, Yuliang Zou, Sungryull Sohn, Xunyu Lin, and Honglak Lee. Learning to generate long-term future via hierarchical prediction. *CoRR*, abs/1704.05831, 2017.

[32] Manuel Watter, Jost Springenberg, Joschka Boedecker, and Martin Riedmiller. Embed to control: A locally linear latent dynamics model for control from raw images. In *Advances in neural information processing systems*, pages 2746–2754, 2015.

[33] T. Weber, S. Racanière, D. P. Reichert, L. Buesing, A. Guez, D. Jimenez Rezende, A. Puigdomènech Badia, O. Vinyals, N. Heess, Y. Li, R. Pascanu, P. Battaglia, D. Silver, and D. Wierstra. Imagination-Augmented Agents for Deep Reinforcement Learning. *ArXiv e-prints*, July 2017.

[34] Jiajun Wu, Joshua B Tenenbaum, and Pushmeet Kohli. Neural scene de-rendering. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[35] SHI Xingjian, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wang-chun Woo. Convolutional lstm network: A machine learning approach for precipitation nowcasting. In *Advances in neural information processing systems*, pages 802–810, 2015.