Universiteit Leiden

# Master Computer Science

Pre-training World Models to efficiently learn related tasks


| | |
|---|---|
| Name: | Ken S. Voskuil |
| Student ID: | s2553740 |
| Date: | 30/06/2022 |
| Specialisation: | Data Science |
| 1st supervisor: | prof. dr. Aske Plaat |
| 2nd supervisor: | dr. Thomas M. Moerland |


Master's Thesis in Computer Science


Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

# Pre-training World Models to efficiently learn related tasks

K. S. Voskuil

June 2022

**Abstract**

Reinforcement Learning problems are often divided into smaller tasks, and each sub-task is then usually solved by separate instances of the same method. We ask if we could improve efficiency by applying a method to sets of sub-tasks. We show that combining a number of sub-tasks is a simple way to scale the difficulty to match the capacity of a chosen method. We also find that combining sub-tasks is an effective pre-training strategy. Even when the combination of sub-tasks is too hard to solve with a single agent, we see clear efficiency improvements when we fine-tune the pre-trained agent on individual tasks. On the other hand, we did not find improved capacity to generalize and transfer experience to other, unseen sub-tasks.

## 1 Introduction

Many, if not most practical Reinforcement Learning (RL) problems can be divided into smaller, more specific tasks, or are part of a set of related tasks which could be composed together into a single, more complex problem. Deciding on the appropriate problem statement in this regard significantly impacts which solutions can be applied, how well these solutions will perform, and the efficiency with which the task is accomplished. As an illustration, imagine applying Reinforcement Learning to cooking; one approach would be to break cooking up into fine-grained tasks such as 'chopping', 'stir-frying' and 'broiling', which we could then try to accomplish by training a separate policy per task. A higher-level policy would then be needed to stitch these policies together to complete the bigger task of cooking a meal. A different approach could be to train one policy end-to-end per recipe. We can even generalise further, and train a single agent to master the French kitchen or any reasonable recipe.

There are several aspects to consider when choosing the problem scope to which you apply your RL method;

- Parameterized policy algorithms have a limited capacity – or maximum policy complexity that can be expressed with any parameterization. A

very general task like 'mastering the French kitchen' might not be feasible at all with the chosen method and configuration. By splitting a problem up and using multiple instances of the method, each with their own parameterization, we scale the capacity of our solution.

- Learning such a general problem is also often *much* harder. The credit assignment problem is one of the core obstacles in RL, and by introducing sub-problems we can explicitly assign importance (or value) to partial results, using external domain knowledge. In the example of cooking, it might take a very long time to learn the importance of correctly dicing an onion if we only measure the final outcome.

- On the other hand, when we subdivide our problem to an impractical level, it might become infeasible to learn a new policy from scratch for each task. You might have hundreds or thousands of recipes you want to train an agent for, making training an agent from scratch per recipe quickly too computationally expensive.

- Besides an infeasible number of tasks, this subdividing can also introduce clear inefficiencies; *many* recipes will require a diced onion, but if we train a new policy for each recipe, we would need to learn how to dice an onion every time. We may try to avoid this by splitting the problem up further, but there will likely always be some overlap between tasks. In our cooking example, even seemingly dissimilar tasks like 'chopping' and 'stir-frying' require an understanding of how to physically control knives and pans.

  In general a fundamental understanding of the dynamics of the environment can usually be applied to and shared between even the most granular tasks. Neural networks have shown to be very effective in learning features and representations automatically[12], but the continuous effort in transfer learning show that reusing these learned representations is not straightforward.

The somewhat hypothetical problem of cookery aside, we see these considerations come into play in much of RL research. Many benchmark sets, such as the Atari 51 benchmark [3], provide a variety of problems with a shared structure (in the case of Atari 51, the observation structure and action space) which are usually solved by training a separate policy for each problem instance. While in benchmarking this serves its own purpose of making it easy to compare a new method across different scenarios, one may ask what the most efficient way is to solve multiple Atari games. It might be that we could save training time by training one policy for multiple games, or even by finding common sub-tasks between games and training and stitching together partial policies.

Another clear scenario where these considerations can be applied are longer video games such as Super Mario Bros., or Sonic the Hedgehog. It may not be feasible to train a single instance of an algorithm to beat the full game, and it is common practice to take the different levels of such a game as separate tasks. However, these levels usually build off of each other, by reusing challenges,

obstacles and general dynamics seen in previous levels. By learning separate policies for each level, we would have to relearn the same things multiple times.

Several sub-fields of Reinforcement Learning explore different approaches to this problem. Most pertinent to our work is Multi-task RL, which studies how to solve multiple tasks simultaneously. Transfer Learning in RL tries to avoid the inefficiencies of learning separate tasks with shared components, by transferring experience learned in one environment to another, and reusing shareable knowledge instead of relearning it from scratch. Meta Learning shares similar goals, but instead of trying to directly reuse experience or learned knowledge, the focus lies on optimizing the learning process itself. These fields are roughly orthogonal, and methods from one sub-field can often be combined with ideas from another.

For our work we apply DreamerV2[8], an established *world model*-base method, on multiple levels of Super Mario Bros. We find that DreamerV2 is capable of beating individual Mario levels without special tuning, and that a single agent can learn an effective policy for multiple of these levels at the same time. Without changing the model, we adapt the training procedure to experiment with pre-training and fine-tuning, which can be considered meta learning methods, and with transferring learned parameters to unseen levels. We compare a typical configuration where we train a separate agent for each level, to training a single agent on multiple levels, fine-tuning pre-trained agents, and transferring pre-trained agents to new levels. We show that both training on multiple tasks and fine-tuning on specific tasks can significantly improve the efficiency compared to learning separate policies for each level.

More specifically, given a set of environments $\mathcal{E}$ we investigate the following research questions;

1. Is it more efficient to train a single DreamerV2 agent on all environments in this set at once, compared to training a separate agent on each environment separately?
2. Can a DreamerV2 agent trained on all environments in $\mathcal{E}$ be fine-tuned further on specific environments $E \in \mathcal{E}$?
3. Can a DreamerV2 agent trained on all environments in $\mathcal{E}$ effectively transfer information to a new environment $E \notin \mathcal{E}$?

# 2 Background

## 2.1 Reinforcement Learning

In Reinforcement Learning (RL) the goal is to learn a good *policy* $\pi$ by interacting with an *environment* $E$. These environments often take the form of games or (simulations of) physical systems in which some task must be performed. However, the core framework of definitions that RL concerns itself with is general

enough that many other problems can be cast in it. There are several examples of methods initially developed for games that were successfully applied to atypical RL problems, such as chemical synthesis planning[17].

### 2.1.1 Environments

More formally, Reinforcement Learning studies problems that can be modeled as a (Partially Observable) Markov Decision Problem or MDP, expressed as an environment:

$$E = \langle \mathcal{S}, \mathcal{A}, p, r \rangle$$

An environment consists of a state space $\mathcal{S}$, an action space $\mathcal{A}$, a dynamics function $p$ and a reward function $r$. $\mathcal{S}$ is the set of all reachable states, while $\mathcal{A}$ describes the set of actions our policy is allowed to take. The dynamics function $p(s', R|s, a)$ describes the probability distribution of the state transition and reward, after taking action $a \in \mathcal{A}$ from the current state $s \in \mathcal{S}$. Finally, the reward function $r(s, a)$ describes the expected reward $R$ after taking action $a$ from state $s$.

Given such an environment and a starting state $s_0 \in S$, we can generate *trajectories*, by choosing an action $a_t$ at each timestep $t$, given the current state $s_t$, and observing the next state $s_{t+1} \sim p(s|s_t, a_t)$ and reward $R_{t+1} \sim p(R|s_t, a_t)$ from the environment.

As an example, the Atari game Pong could be expressed in this framework by using the game screen as the observable state and the state of the different buttons on the Atari game pad as the action space. While Pong might seem mostly deterministic, we observe that this MDP environment for Pong is not; given just one frame, it is not possible to tell which direction the ball is moving, and $p$ would assign the same probability to each possible direction. Such an MDP is also called a Partially Observable MDP, as (part of) the state transition uncertainty is explained by hidden state variables.

In this example, this could either be resolved by changing the MDP (for example, by making the observable state in the MDP consist of the last two frames), or by trying to capture or model the hidden state with the agent, for example by using a RNN-based method.

### 2.1.2 Optimization

Reinforcement Learning generally tries to optimize the parameters $\theta$ of a parameterized policy function $\pi_\theta(a|s)$, to maximize the *value*, or the expected accumulated rewards in a trajectory. Given a policy function $\pi$, we can define the value of a state $s$ as the expected accumulated discounted rewards if we follow $\pi$ from $s$:

$$v_\pi(s_t) = \mathbb{E}[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ...] = \mathbb{E}\left[\sum_{k=0} \gamma^k R_{t+k}\right]$$

The discount factor $\gamma \in [0, 1]$ ensures convergence even for infinite trajectories, as long as $\gamma < 1$ and $R_k$ is bounded. It also allows control over how much to emphasize short-term rewards over long-term rewards.

Thus, given an environment $E$, a discount factor $\gamma$ and a parameterized policy function $\pi_\theta$, we can formalize the optimization problem that Reinforcement Learning tries to solve as finding the optimal policy $\pi^*$, where

$$\pi^* = \max_{\pi_\theta} v_{\pi_\theta}(s), \text{for all } s$$

Reinforcement Learning is an iterative process, where a current parameterization $\theta_t$ is used to obtain experience from the environment for one or multiple episodes, which in turn is used (together with previous experience) to optimize the parameters for $\theta_{t+1}$.

### 2.1.3 Combining MDPs

Besides being able to capture a wide variety of problems, we observe that multiple MDPs can always be combined into one. For example, if we are given two environments $E_A = \langle \mathcal{S}_A, \mathcal{A}_A, p_A, r_A \rangle$ and $E_B = \langle \mathcal{S}_B, \mathcal{A}_B, p_B, r_B \rangle$, then we can construct a new MDP $E_{AB} = \langle \mathcal{S}_{AB}, \mathcal{A}_{AB}, r_{AB}, p_{AB} \rangle$ that combines $E_A$ and $E_B$ with:

$$\mathcal{S}_{AB} = \{(0, s) | s \in \mathcal{S}_A\} \cup \{(1, s) | s \in \mathcal{S}_B\}$$

$$\mathcal{A}_{AB} = \{(0, a) | a \in \mathcal{A}_A\} \cup \{(1, a) | a \in \mathcal{A}_B\}$$

$$p_{AB}((e, s')|(e, s), (e, a)) = \begin{cases} p_A((e, s')|s, a) & \text{if } e = 0 \\ p_B((e, s')|s, a) & \text{if } e = 1 \end{cases}$$

$$r_{AB}((e, s), (e, a)) = \begin{cases} r_A(s, a) & \text{if } e = 0 \\ r_B(s, a) & \text{if } e = 1 \end{cases}$$

As a result, methods that apply to general RL problems can also be applied to a combination of multiple problems. Our work focuses on a special case, where the state and action spaces are shared between the different environments, but the state distributions are almost completely disjoint. This is common in research on multi-task reinforcement learning, which often assumes that the different environments or tasks also share the dynamics.

Besides multiple tasks in the same environment, our work also applies to multiple games on the same platform (such as the Atari 51 benchmark) and multiple levels of the same game. While you could argue that the dynamics in the latter scenario *is* shared, the state distributions in these levels are usually nearly completely disjoint. Effective generalization is thus still needed to be able to transfer knowledge of the dynamics from one level to the next.

Because we assume the observable state distribution is (close to) disjoint, we do not explicitly pass which environment an observation comes from to

our policy; we use $\pi(s)$ where $s$ was observed from one of our environments, instead of $\pi((e, s))$ where $e$ would identify which environment this state came from. This design choice allows us to simplify the implementation of combining environments. Instead of explicitly creating new reward and dynamics functions, we choose between the environments at the start of an episode, and use the corresponding reward and dynamics function within that episode.

As a slight abuse of notation, in the rest of this work we use $\mathcal{E} = \{E_A, E_B\}$ not just to refer to a set of environments, but also for the constructed environment that combines $E_A$ and $E_B$ as we just described.

## 2.2 Model-based RL and World Models

Model-based RL encompasses a group of algorithms that use a model of (part of) the environment while learning a value function or policy. This model can be exact or approximate and can be learned or given. With MuZero, Schrittwieser et al. [15] show that a model does not have to correspond to the environment directly to be beneficial, as long as it is approximately *value equivalent* over a relevant set of policies.

Besides variations in the kind of model or how the model is obtained, model-based algorithms also differ in how the model is applied. For example, the model can be used in online planning or for generating simulated training data.

Recent successes in model based reinforcement learning show that *world models* can be an effective tool to train agents for relatively complex environments. Coined by Schmidhuber et al.[14], this term is used by some to refer to learned recurrent models of at least the transition function $p$, to try to capture the dynamics of the system – as opposed to non-recurrent or given models, or models that capture a different component of the environment, such as the reward function. Such a world model can complement the training data for learning a policy with 'imaginary' observations, or replace the true environment completely[5].

In addition to modeling the transition function, several recent works[6][7] also learn a compact latent representation of the state space, which is obtained through a learned recurrent encoder $q(s_t, h_{t-1}) = h_t$. In the case of Dreamer[7], for example, a dynamics model is then trained within this latent space, that can predict the current latent state $h_t$ given the previous latent state $h_{t-1}$ and action $a_{t-1}$. Similarly, the policy is also applied to these latent states, instead of the full observed state.

Aside from learned feature compression, which has been shown to improve the performance of policies, this also decouples the size of the transition model and policy network from the observation size in the true environment, which is usually very big and sparse. As a result, the authors of Dreamer and DreamerV2 [7][8] note that they can simulate and learn from 'thousands' of parallel imagined trajectories instead of just a few real trajectories.

## 2.3 DreamerV2

Our work builds on DreamerV2 by Hafner et al.[8], a world model-based method which replaces the world model of the original Dreamer with a stochastic model with a (partially) discrete latent representation. The latent state at time step $t$ is composed of a deterministic vector $h_t$ and a stochastic discrete vector $z_t$. The discrete vector consists of 32 categorical values, each with 32 classes. The authors provide several hypotheses to motivate the use of a categorical distribution for the latent state, including that this could be a better inductive bias for non-smooth state changes in video games.

Because our work hinges on DreamerV2, we will discuss its design in more depth.

### 2.3.1 World model

As mentioned earlier, the dynamics of the environment are modeled by DreamerV2 in the latent space, and are learned through autoregression; while the distribution $q_\phi(z \mid h_t, s_t)$ for the latent vector $z_t$ is conditioned on the observed state at timestep $t$, the model also tries to predict $\hat{z}_t \sim p_\phi(\hat{z}_t \mid h_t)$ from just the recurrent state $h_t$. These can be taken as the posterior and prior distribution respectively, and optimized across timesteps as a sequential beta-VAE[10], by optimizing their KL-divergence.

The latent vector $h_t$ is derived recurrently from the last latent state $(h_{t-1}, z_{t-1})$ and the last action $a_{t-1}$. DreamerV2 also models the probability distributions for the observed state $s_t$, reward $r_t$ and discount factor $\gamma_t$, conditioned on the latent state. By learning to model these aspects of the environment based on the latent states, these are optimized to learn a compact representation of the environment. The parameters for all components of the world model are optimized jointly, according to the loss function given in equation 3. As such, we represent the parameterization of the whole system with $\phi$. The components of the world model can be summarized with the following equations, as taken from [8]:

$$
\begin{aligned}
&\text{Recurrent model:} && h_t &&= f_\phi(h_{t-1}, z_{t-1}, a_{t-1}) \\
&\text{Representation model:} && z_t &&\sim q_\phi(z \mid h_t, s_t) \\
&\text{Transition predictor:} && \hat{z}_t &&\sim p_\phi(\hat{z} \mid h_t) \\
&\text{Image predictor:} && \hat{s}_t &&\sim p_\phi(\hat{s} \mid h_t, z_t) \\
&\text{Reward predictor:} && \hat{R}_t &&\sim p_\phi(\hat{R} \mid h_t, z_t) \\
&\text{Discount predictor:} && \hat{\gamma}_t &&\sim p_\phi(\hat{\gamma} \mid h_t, z_t).
\end{aligned}
\tag{1}
$$

### 2.3.2 Policy

The policy for DreamerV2 is trained using an Actor-Critic algorithm[2], operating on top of the latent state representations from the world model. The components of the algorithm can be described by the equations

$$\begin{aligned} &\text{Actor:} \quad a_t \sim p_\psi(a|z_t) \\ &\text{Critic:} \quad v_\xi(z_t) \approx \mathbb{E}_{p_\phi, p_\psi}\left[\sum_{t'=t+1} \gamma^{t'-t} R_t\right] \end{aligned} \tag{2}$$

The actor and critic are both implemented as an MLP with the same size, and with ELU activations. Here, each component has its own parameterization and is optimized separately. The actor is optimized to maximize the critic, while the critic is trained using TD-learning, with a $\lambda$-target, which incorporates the rewards and value approximations of multiple steps.

### 2.3.3 Training

Experience is gathered by generating trajectories in the environment using the current policy, modulated by $\alpha$-greedy exploration. For each time step, we record a tuple containing $\langle s_t, R_t, \gamma_t, a_t \rangle$, where $\gamma_t$ is 0 for absorbing states, and 1 otherwise. At the end of each episode, this data is added to the replay buffer. The replay buffer is initially seeded using a random policy for $T_{init}$ steps. These steps are counted towards the training budget $T$.

Every $n$ steps, DreamerV2 runs the training procedure, which can be divided in roughly two steps; after collecting a batch of training data, it first trains the world model, then the policy. Crucially, the training data is only used to train the world model, while the policy is trained on *imaginary* trajectories, generated by the world model. The flow of data during training is displayed in the diagram in figure 1.

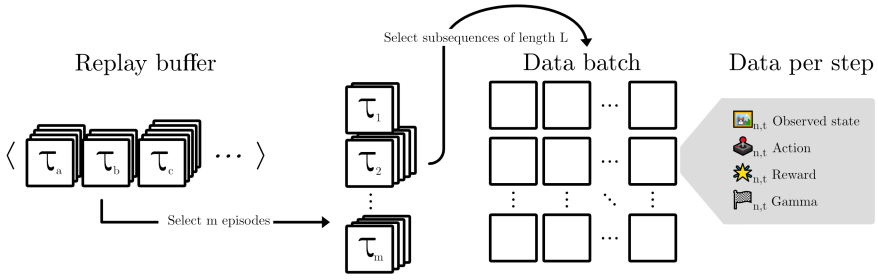
First, training data is sampled from the replay buffer, consisting of $m$ episode subsequences of length $L$. The world model is used to obtain the latent representations and predicted latent representations for these subsequences. These are used together with the environment signals recorded in these subsequences to optimize the world model according to the following loss function:

$$\begin{aligned} &\mathcal{L}(\phi; s_{1:L}, R_{1:L}, \gamma_{1:L}, a_{1:L}) = \\ &\mathbb{E}_{q_\phi(z_{1:L} \mid a_{1:L}, s_{1:L})}\Big[\sum_{t=1}^{T} \underbrace{-\ln p_\phi(s_t \mid z_t, h_t) - \ln p_\phi(R_t \mid z_t, h_t) - \ln p_\phi(\gamma_t \mid z_t, h_t)}_{\text{representation loss}} \\ &\qquad + \underbrace{\beta \mathrm{KL}[q_\phi(z_t \mid h_t, s_t) \mid\mid p_\phi(\hat{z}_t \mid h_t)]}_{\text{dynamics loss}}\Big] \end{aligned} \tag{3}$$

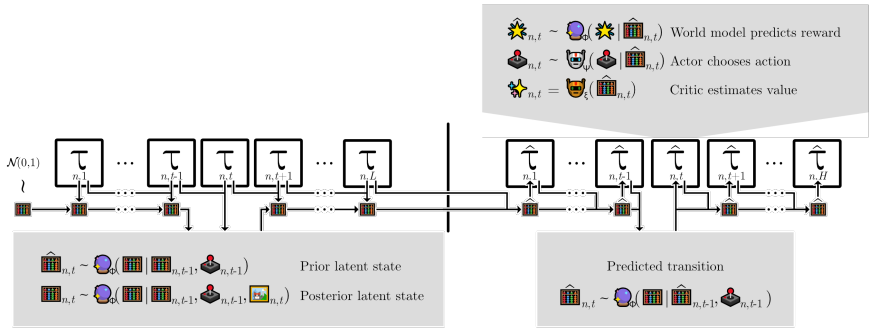
The representation loss maximizes the log likelihood of the recorded environment signals, conditioned on the latent state, while the dynamics loss minimizes the divergence between $p_\phi$ and $q_\phi$, bringing the two distributions closer together.

After optimizing the world model, the built up latent representations of the trajectories are reused as the starting point for $m$ imaginary trajectories of length $H$. This is done by recursively sampling an action from the policy

conditioned on the current imagined latent state $a_t \sim p_\psi(a|\hat{z}_t)$, to obtain the next recurrent state $h_{t+1} = f_\phi(h_t, \hat{z}_t, a_t)$ and predict the next latent state $\hat{z}_{t+1} \sim p_\phi(\hat{z} \mid h_{t+1})$.




(a) Each training step, DreamerV2 samples a batch of subsequences of length $L$ from $m$ trajectories.




Collect prior and posterior latent states to train world model

Generate imaginary trajectory to train actor and critic

(b) For each subsequence DreamerV2 recurrently builds the prior and posterior latent state using $q_\phi$ and $p_\phi$ respectively. After constructing this latent representation of the subsequence, the trajectory is continued by applying $p_\phi$ recurrently to generate new latent states. For each step in this imaginary trajectory DreamerV2 predicts the reward and value using the world model and critic. This data is then used to train the critic.

Figure 1: The flow of data during a training step of DreamerV2.

For each imaginary step $t$, the world model is used to predict the reward $\hat{r}_t \sim p_\phi(\hat{r} \mid h_t, \hat{z}_t)$ and discount value $\hat{\gamma}_t \sim p_\phi(\hat{\gamma} \mid h_t, \hat{z}_t)$. These are then used to calculated the value targets, to train the critic with. For this, DreamerV2 uses $\lambda$-targets[16]:

$$V_t^\lambda = \hat{R}_t + \hat{\gamma}_t \begin{cases} (1-\lambda) v_\xi(\hat{z}_{t+1}) + \lambda V_{t+1}^\lambda & \text{if } t < H, \\ v_\xi(\hat{z}_H) & \text{otherwise.} \end{cases} \tag{4}$$

The critic is optimized by minimizing the squared error:

$$\mathcal{L}(\xi; \hat{z}_{1:H-1}, V_{1:H-1}^\lambda) = \mathbb{E}_{p_\phi, p_\psi} \sum_{t=1}^{H-1} \frac{1}{2} \left( v_\xi(\hat{z}_t) - \text{sg}(V_t^\lambda) \right)^2 \tag{5}$$

Where $\text{sg}(\cdot)$ denotes *stop gradients*. The actor is then optimized using Reinforce gradients[19], which maximizes the probability of the chosen actions, weighed by the centered action value:

$$\mathcal{L}(\psi; a_{1:H}, V_{1:H}^\lambda) = \mathbb{E}_{p_\phi, p_\psi} \sum_{t=1}^{H-1} - \ln p_\psi(a_t | \hat{z}_t) \text{sg}(V_t^\lambda - v_\xi(\hat{z}_t)) \underbrace{- \eta \text{H}[a_t | \hat{z}_t]}_{\text{entropy regularization}} \tag{6}$$

We observe that while the policy and world model are trained separately, this training procedure introduces a dependency between the two training steps by using the latent state produced while training the world model as the starting point for the imaginary trajectories generated to train the policy. The length of the subsequences $L$ not only controls how many environment steps the world model trains on in each training step, but also limits how much historic information can be used to generate the imagined trajectories that the policy is trained on.

The implementation of DreamerV2 is open source, and is available on github[1], including the Atari environments and hyperparameters. We add minor extensions to their implementation; we add an environment wrapper that instantiates multiple sub-environments and chooses one randomly for each episode, we integrate the gym-mario environment and we extend the main script to help manage our configurations and experiments. We also implement support to sample past the horizon of an episode, which is described and motivated further below.

# 3 Related Work

Our work is not the first to explore the idea of (pre-)training on multiple related tasks. One of the earliest works to try this with Deep RL methods is "Actor-Mimic Deep Multitask and Transfer Reinforcement Learning" by Parisotto, Ba and Salakhutdinov[13]. In their work, they modify DQN[12] and show that with a single model of similar size, they could learn eight Atari games at the same time, and reach a comparable performance for seven of those games to normal DQN agents that were trained on each game separately. They also show that an agent trained like this could learn new, related environments faster.

[1] https://github.com/danijar/dreamerv2

Their multi-task agent is not trained to optimize its task performance directly, but to mimic aspects of expert agents, a process which is also called distillation. While their work shows that a single DQN agent has the capacity to learn multiple complex games, their use of expert networks means that their method needs strictly more resources than training a separate agent per task. Similarly, their transfer learning method requires a very high initial cost, with modest gains in most games. As a result, it would take transferring to many games before this method becomes more efficient.

More recent work that focuses on the multi-task RL scenario includes Distral[18], which trains a separate policy per task, but aims to transfer knowledge between tasks by distilling each task policy into one shared policy, in turn using this shared policy to regularize the optimization of the task policies. Hessel et al. extend PopArt[9], which learns a normalized value function, to the muti-task setting, to learn tasks with very different reward scales. Liu et al.[11] propose another specialized model architecture for multi-task learning, with a shared convolutional encoder-decoder network, and task-specific attention networks. The idea here is that the convolutional network learns global features that can be shared, while the attention networks can select the features that are relevant for a specific task.

A common thread through these works is that they introduce a new architecture or modifications to existing models to have both task-specific components and a shared component. In this regard, our work is simpler; we apply an existing model to a multi-tasking scenario without modifications, and without task-specific components. We hope that the world model of DreamerV2 is able to facilitate transfer learning between tasks by finding efficient representations for common features. Because we do not modify the model, we can easily switch from a multi-tasking scenario to single tasks. This leads us naturally to fine-tuning agents on specific tasks after being trained on $N$ tasks simultaneously.

Besides research from the multi-task learning field, our work also bears similarities to MAML, by Finn et al.[4]. Like in our pre-training set-up, in MAML a model is trained on multiple tasks during a *meta-learning* phase, and the parameterization obtained at the end of this phase are then used for initialization in future tasks. Instead of optimizing the parameters for the tasks in the meta-learning phase directly, MAML instead optimizes for maximal sensitivity to the loss functions of the task. In other words, MAML tries to find the parameterization that gives the biggest improvement after one gradient descent step. As a result, it should be possible to fine-tune this parameterization for a new task in just a few optimization steps.

Where MAML focuses on few-shot transfer scenario's to tasks outside of the meta-learning, we also look at the performance on the pre-training tasks, both before and after fine-tuning.

# 4 Methods

To investigate the muti-tasking and generalization capabilities of Dreamer, we propose three experiment set-ups, outlined below.

## 4.1 Multiple environments

We train a single DreamerV2 agent on a set of environments $\mathcal{E}^{pre}$ for $T$ steps. The performance of this agent will be compared with the performance of $N = |\mathcal{E}^{pre}|$ separate baseline DreamerV2 agents, trained for $T$ steps on each of the environments. We also compare our performance against the baseline performance after $\frac{T}{N}$ steps, as at this point, the baseline agents have used an equivalent amount of resources (in terms of computation and data) as our agents.

To train an agent on multiple environments, we follow the normal training procedure of DreamerV2, but during the collection of data, we choose one of the $N$ environments at the start of each episode, using a uniform distribution. Trajectory fragments used to train the world model are sampled by first choosing $m$ episodes from which the fragments are taken. Thus, the agent trains on each environment approximately as often, even when there are discrepancies in episode lengths. If DreamerV2 is able to efficiently generalize between the seen environments, we expect to see our agent reach a similar or better performance after $T$ steps as what the baseline agents achieve after $\frac{T}{N}$ steps.

The complexity of this task greatly depends on how many environments we choose for $\mathcal{E}^{pre}$, and which ones specifically. The larger the similarities between the environments, the easier it should be to generalize between them. To test the effect of the number of environments, we run our experiments for two sizes of $\mathcal{E}^{pre}$.

This experimental set-up can be viewed as a *pre-training* procedure. In the next two experiment set-ups, we use the parameterizations obtained in this experiment for initialization. Because of this, we will also refer to this set-up as the pre-training set-up.

## 4.2 Fine-tuning on seen environments

In the first of these two set-ups, we use the learned parameters to train $M \leq N$ new agents, each on one of the environments $E \in \mathcal{E}^f \subseteq \mathcal{E}^{pre}$. These agents are trained for $T$ steps and compared with the $M$ corresponding baseline agents trained on the same environments. We also compare the results averaged over the $M$ agents at step $t$ to the averaged returns of the baseline agents at step $t + \frac{T}{M}$, to take the overhead of pre-training into account.

In combination with the previous experiment, these results may show if there is a bottleneck in the generalization capacity of DreamerV2. For example, if we find that the fine-tuned agents improve on the baseline results, and that this improvement is larger than in the previous result, this could be an indication that some components of DreamerV2 were able to generalize well, while other components could not.

### 4.3 Transfer learning with new environments

With the last experiment set-up, we investigate if pre-training is useful for transfer learning. This time we train $K$ new agents, initialized using the parameters obtained during pre-training, on $K$ environments $\mathcal{E}^{\tau}$, with $\mathcal{E}^{\tau} \cap \mathcal{E}^{pre} = \emptyset$. We also train $K$ baseline agents on these environments, using the original initialization scheme of DreamerV2. Both our agents and the baseline agents are trained for $T$ steps, and their performance are compared directly.

## 5 Experiments

We ran our experiment on Nintendo's Super Mario Bros., where we treat each level as a separate environment. We found during early exploration that DreamerV2 learns Mario levels much quicker than most Atari games, which we attribute in part to the dense reward function used for the Mario games. Because of this, we ran our experiments for $T = 10M$ frames, compared to the $T = 100M$ frames Hafner et al. used for Atari games. Another factor for choosing Super Mario Bros. is that, compared to the Atari games, different levels are much more similar to each other, both in terms of visual features and dynamics, though there is still enough variation to learn about the generalization capabilities of our method.

Besides the types and placement of obstacles within each level, there are five distinct visual themes[2]: *Overworld*, *Underground*, *Athletic*, *Castle* and *Underwater*, each with their own texture sets. The *Overworld* and *Athletic* levels can also be set either at night (with a black background) or during the day (with a blue background). Levels are grouped into sets of four, called *worlds*, with repeating themes and increasing difficulty between worlds. Levels are referenced to by their world number and level number within their world. For example, the seventh level is referenced by `2-3`, as it is the third level of the second world.

### 5.1 Environment details

To run the game, we used the `gym-super-mario-bros` package[3], which is built on top of the open source NES-py emulator[4]. By default, this package comes with several control schemes. The control scheme used for our experiment allows for the following button combinations, as twelve discrete options: `'None'`, `'Right'`, `'Right, A'`, `'Right, B'`, `'Right, A, B'`, `'A'`, `'Left'`, `'Left, A'`, `'Left, B'`, `'Left, A, B'`, `'Down'`, and `'Up'`.

The package also introduces its own reward function. This function rewards the agent to move to the right as fast as possible. The reward for a given action is given by the horizontal movement (positive for moving to the right, negative for moving to the left) and elapsed time ($-1$ for each tick of the internal game

[2]https://www.mariowiki.com/Super_Mario_Bros.
[3]https://github.com/Kautenja/gym-super-mario-bros
[4]https://github.com/Kautenja/nes-py

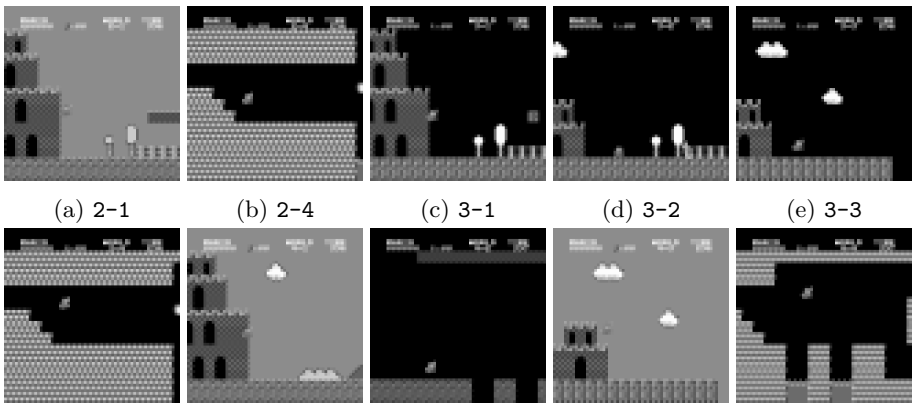
clock), with a negative bonus of $-15$ when the agent dies. The reward is clipped to stay between $(-15, 15)$.

## 5.2 Integration and hyperparameters

To integrate this game into our training set-up, we chose to preprocess the input and output into the same structure as used by DreamerV2 for Atari games. We transform the input frames into grayscale, and resize them to a resolution of $64 \times 64$ pixels. We also repeat each action for 4 frames, and implement sticky actions, with a probability of 0.2 of repeating the previous action. We reuse all hyperparameters from the Atari experiments.

In certain Mario levels, bad policies (such as a random policy) lead to very short episodes. In the original implementation of DreamerV2, episodes shorter than the training trajectory subsequence length $L$ were ignored during training, leading to pathological cases where our agent collects very little usable training data during initialization, and never improves. To combat this, instead of throwing away short episodes, we allow DreamerV2 to sample past the end of an episode. For the time steps past the end of the episode, we treat the last step of the episode as an absorbing state, so that $p(s_t = s_{t+1}|s_t, a_t) = 1$, no matter the action. To this end, we repeat the last observation, set the reward and discount targets to 0 and uniformly sample a random action. We do not expect to generate many of these absorbing states in practice, as policies quickly improve to generate episodes much longer than $L$ steps.

## 5.3 Experiment configurations



(a) `2-1` (b) `2-4` (c) `3-1` (d) `3-2` (e) `3-3`

(f) `3-4` (g) `4-1` (h) `4-2` (i) `4-3` (j) `4-4`

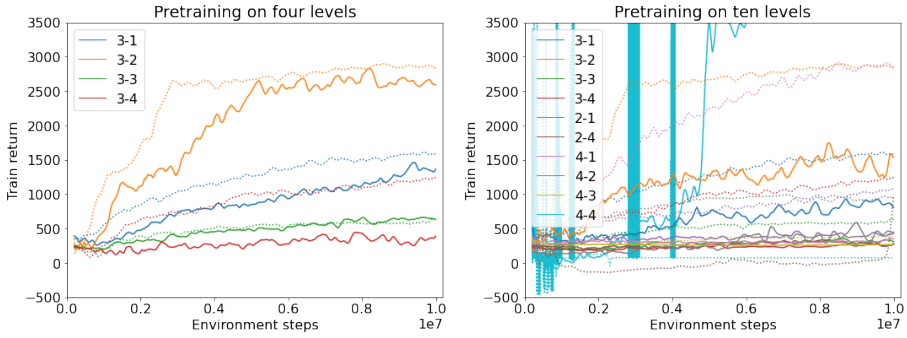
Figure 2: Observed frames from the 10 levels used in the pre-training and fine-tuning experiments, near the start of a trajectory in each level. For fine-tuning and for pre-training with the $\mathcal{E}_4^{pre}$ configuration, only the levels of world three are used.

We executed two experiment configurations, one with a small set of levels of $|\mathcal{E}_4^{pre}| = 4$ and one with a larger set $|\mathcal{E}_{10}^{pre}| = 10$. We excluded all *Underwater* levels from our experiments, as there are only two, and they have very different dynamics from the other levels; Mario is less affected by gravity in these levels, but slowed down while walking. Similarly, we excluded level `2-3`, as this level almost exclusively repeats one obstacle, which rarely appears in other levels. For both experiment configurations, we use the levels from the first world for transfer learning and levels from the third world for fine-tuning: $\mathcal{E}^{\tau} = \{\texttt{1-1}, \texttt{1-2}, \texttt{1-3}, \texttt{1-4}, \}$ and $\mathcal{E}^{f} = \{\texttt{3-1}, \texttt{3-2}, \texttt{3-3}, \texttt{3-4}, \}$. Both worlds include an instance of each level theme. For the small experiment, we set $\mathcal{E}_4^{pre} = \mathcal{E}^{f}$. For the larger experiment we use the first ten levels after world one; $\mathcal{E}_{10}^{pre} = \{\texttt{2-1}, \texttt{2-4}, \texttt{3-1}, \texttt{3-2}, \texttt{3-3}, \texttt{3-4}, \texttt{4-1}, \texttt{4-2}, \texttt{4-3}, \texttt{4-4}\}$.

As with the original Atari experiments, we repeat each configuration (including the baselines) with $R = 5$ different seeds. Summing up, we train our baseline agents for $5 * (10 + 4) * 1e7$ frames, we fine-tune these agents for another $5 * (4 + 4) * 1e7$) frames, and retrain them on unseen environments for $5 * (4 + 4) * 1e7$ frames, leaving us with a total of $150e7$ training frames. We ran this experiment on multiple machines with RTX 2080s and RTX 2070s. We found that training one agent for $10M$frames took about 36 hours, giving us a total computation time of about 5400 hours, or 225 days.

# 6 Results

## 6.1 Multi-task training




(a) $\mathcal{E}_4^{pre}$ (b) $\mathcal{E}_{10}^{pre}$

Figure 3: Train returns collected during pre-training DreamerV2 on $N = 4$ (left) and $N = 10$ (right) levels. The dashed lines present the performance of the $N$ baseline agents, averaged over five runs, while the solid lines show the performance of the (single) pre-training agent. The cost of training a pre-training agents for $T$ steps is $\frac{1}{N}$ the cost of training $N$ baseline agents. The erratic behavior seen in `4-4` is an artifact of the exploitable reward function.

Figure 3 shows the train returns of DreamerV2 applied to $\mathcal{E}_4^{pre}$ and $\mathcal{E}_{10}^{pre}$, plotted per level. We also plot the returns of the baseline agents.

When training on four levels at the same time, we see that DreamerV2 is able to achieve comparable performance to the baseline in three out of the four levels, even though this agent only sees roughly $\frac{1}{4}$ times the data for each level. For the levels `3-1` and `3-2`, we see that DreamerV2 learns a bit slower than the baseline agents, but they reach a similar performance ceiling, and the difference in learning rate is not quite four times slower. For `3-3`, the baseline and muti-tasking agent both perform similarly poorly; it seems both agents are not able to get past an early obstacle. Finally, `3-4` is the only level where the muti-tasking agent is not able to perform close to the baseline.

When training on ten levels, the results look very different. We first address the line for level `4-4`; both the baseline and muti-tasking agent perform very erratically, with large spikes throughout the training process. This is an artifact of the reward function, which the agent is able to exploit in this level. As described, the agent is rewarded (or penalized) for horizontal movement. This gives the agent a dense incentive to move towards the end of the level, which is always at the right end of the level. However, in level `4-4`, taking a wrong turn halfway through the level will transport the player to near the beginning of the level. By taking this turn on purpose, the agent can move to the right continuously and accumulate rewards until the timer runs out, while avoiding the hard obstacles at the end of the level. Comparing the baseline agent in this level with the muti-tasking agent, we do see that the muti-tasking agent outperforms the baseline significantly. The muti-tasking agent seems to be able to exploit the reward function much more consistently. However, because the reward in this level is nearly unbounded with a relatively easy policy (at least compared to actually completing the level), we see the results for this level as outliers, and have clipped the peaks out of this graph.

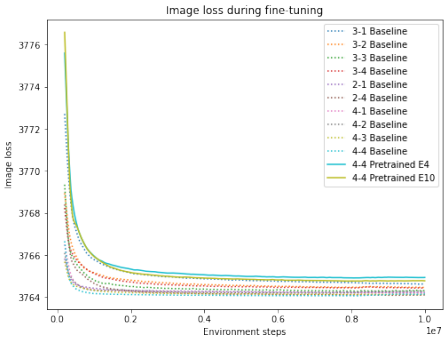


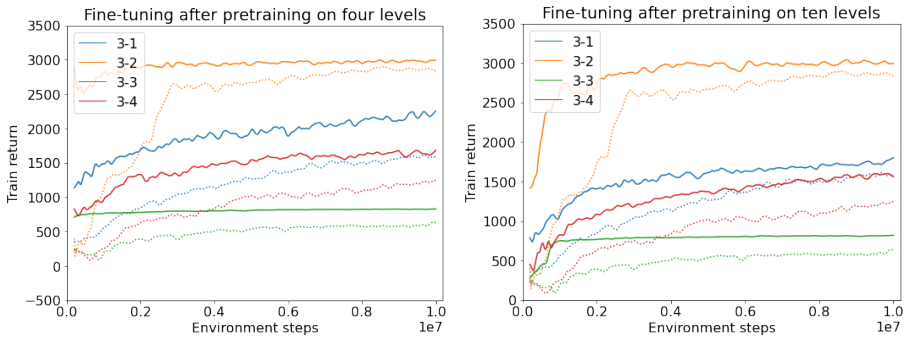

Figure 4: The image prediction loss for ten baseline configurations and two pre-training configurations.

Looking at the results for the other levels, we see that DreamerV2 is unable to keep up with the baseline results in *most* levels, or even get close to the baseline performance at $t = \frac{T}{10}$. For seven out of the ten levels, the muti-tasking agent does not seem to significantly improve over its initial, random policy. In levels `4-1` and `4-2`, the agent does improve, but still performs significantly worse than in the $\mathcal{E}_4^{pre}$ configuration. These results show clearly that Dreamer does not have the capacity to master 10 Super Mario Bros. levels at the same time.

Besides looking at the returns of our agents, we can also look at the performance of the world model. We plot the image loss for both pre-trained agents

and the baseline agents in figure 4. We see that the pre-training agents perform very similarly to the worst performing baseline agent for level `3-1`. Since this level is also part of the pre-training environments, it would be surprising if the pre-trained agents performed much better than this agent. We would expect the pre-training image loss to be strictly worse, as we have a much higher diversity of data to learn from. Of course, measuring the direct model loss can be deceptive in model based RL, as two agents might just see different data from its environment. Still, we believe these results are a signal that the world model has significant overcapacity for this domain, and that the bottleneck lies in the policy.

## 6.2 Fine-tuning



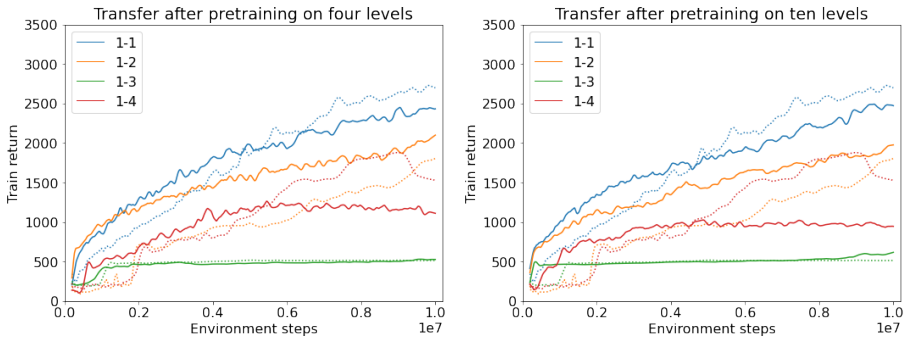

(a) $\mathcal{E}_{10}^{pre}$ (b) $\mathcal{E}_{10}^{pre}$

Figure 5: The train return of our pre-trained agents, after fine-tuning them for another $1e7$ steps on four tasks from the pre-training task set. The solid lines represent the train return of the pre-trained agents, while the dashed lines show the train return of the four baseline agents trained on these tasks.

In figure 5 we plot the train returns of fine-tuning the agents obtained in the previous experiment, on all four levels of world 3.

Looking at the results of the agents pre-trained on $\mathcal{E}_4^{pre}$, we see that fine-tuning is able to continue improving the agents from where we stopped pre-training. In fact, if we compare the performance of the fine-tuned models at $t = T - \frac{T}{4} = 0.25e7$, which is the point at which the fine-tuning experiment has used as many computational resources and samples as the baseline agent, we see that the fine-tuned models clearly outperform the baseline in every level. Perhaps surprisingly, we can see very similar behavior for our $\mathcal{E}_{10}^{pre}$ fine-tuning experiment. Even though our DreamerV2 agent was not able to learn all 10 levels at the same time, it seems that DreamerV2 was still able to gain relevant information from pre-training.

### 6.3 Transfer learning




(a) $\mathcal{E}_4^{pre}$ (b) $\mathcal{E}_{10}^{pre}$

Figure 6: Train returns of our pre-trained agents in four new levels. The solid lines represent the train return of the pre-trained agents, while the dashed lines show the train returns of the four baseline agents trained on these levels.

The results of our final experiment configuration are shown in figure 6. Here we see the results of using the parameters of the pre-trained agents on four levels from world 1, which were not part of either pre-training environment. In this scenario, our agents have seen *more* data than the baseline. We had expected that this previous experience would translate to learning new levels of the same game faster. However, the train returns show a mixed result. We do see that for every level, the pre-trained agent starts off better than the baseline agents. However, this advantage quickly diminishes for levels `1-1`, `1-4` and especially `1-3`. Both the agents pre-trained on $\mathcal{E}_4^{pre}$ and $\mathcal{E}_{10}^{pre}$ are outperformed by the baseline agents before $t = \frac{T}{2} = 0.5e7$.

While the pre-trained agents do start off better, and seem to perform better in one of the levels, on average, the performance is similar to the baseline, while obtaining the initial parameters costed a large amount of data and computation.

## 7 Discussion and future work

Given a problem divided in several related sub-tasks, instead of applying tabula rasa learning on each separate sub-task or on the whole problem at once, we explored a middle road. We trained a single agent on a varying number of these sub-tasks and fine-tuning the agent on specific sub-tasks both from the pre-training set and from outside this set. We now discuss our results, relate them to our original research questions, and point out avenues for future research.

## 7.1 Difficulty scaling

If we compare our pre-training results, we see that without tuning, DreamerV2 is fairly capable of solving four levels with one agent, while ten is clearly too much. These results answers our first research question; it *can* be more efficient to train a single DreamerV2 instance on multiple problems, as we see for $\mathcal{E}_4^{pre}$, though this effect breaks down if the set of combined problems is too big or too difficult. Once this point is reached, we see that DreamerV2 does not just become less efficient, but might not be able to learn a policy at all for most problems in the set.

This method introduces a new hyperparameter $|\mathcal{E}|$, for which we chose an arbitrary small and large value. To apply this in a practical setting, we should have some strategy for choosing this value. Of course, it is also possible to dynamically choose sets of sub-problems. For example, by starting with the set of all $N$ sub-problems, and halving a set if the agent for that set is not learning after $t$ steps. After splitting a set in two, the process is repeated with a separate agent on each half. In the worst case, where each set contains just one sub-problem and you are training $N$ separate agents, the overhead of this process amounts to $t(\frac{N}{2} + \frac{N}{4} + ...)$ steps. Taking $N$ to be a power of 2 for simplicity, this comes out to $t\sum_{i=1}^{log_2(N)} \frac{N}{2^i} = t(N-1)$.

On the other hand, if we expect that a single agent can solve on average two of the sub-tasks, we would only need to train $\frac{N}{2}$ agents, saving $T\frac{N}{2}$ steps. In this scenario, the overhead would come out to $t\sum_{i=2}^{log_2(N)} \frac{N}{2^i} = t\frac{N-2}{2}$. Thus, in this scenario, even if we only checked and dynamically split the task set after $T$ steps, this method would be more efficient compared to training $N$ agents for $T$ steps directly. This dynamic algorithm thus can quickly improve the efficiency of solving a series of sub-tasks, by scaling the difficulty of the problem to the capacity of the chosen method.

This proposed method is naive, in the sense that it does not use any gained knowledge about the sub-tasks to group them. If we could group tasks based on similarity, this would have a large impact on how easy it is to learn a group of tasks. Recent work in multi-task RL, such as as [1], show that task clustering can be done unsupervised, and can have a significant positive effect on multi-task learning efficiency.

Finally, our fine-tuning results show a dramatic improvement in performance early in their learning curve. Most strikingly, we see these improvements even when fine-tuning the $\mathcal{E}_{10}^{pre}$ agent. While this agent was not able to improve over a random policy for levels `3-3` and `3-4` during pre-training, with fine-tuning we were able to improve from random policy to baseline at $t = T$ in less than $t = 0.2 * T$ steps. This suggests that even while the agent pre-trained on $\mathcal{E}_{10}^{pre}$ was not able to learn an effective policy for those 10 levels, it did learn useful information for those levels.

Based on this result, it seems a top-down clustering algorithm that splits larger task groups up into smaller ones would fit better than bottom-up approach; in our example, even if we find after $t$ steps that the current set of sub-tasks is too hard, the agent might still have gained valuable experience.

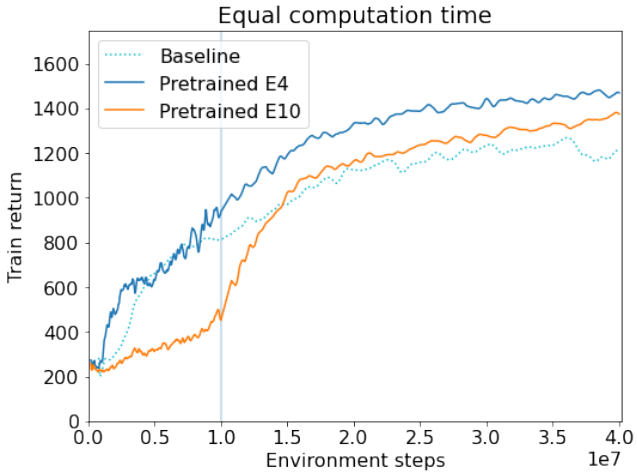



Figure 7: Train returns of our pre-training and fine-tuning configurations, compared to the baseline, averaged over the levels in $\mathcal{E}^f$. The first $1e7$ steps of the solid lines represent the results from pre-training, while the remaining $3e7$ steps represent the results from fine-tuning the agents.

After splitting up the current task set into smaller sets, we can initialize new agents using the parameterization of the last iteration, to apply to the new subsets, keeping the relevant experience gained in the previous $t$ steps.

## 7.2 Efficiency

Our results show that both combining tasks and especially fine-tuning after pre-training can be more efficient than tabula rasa learning on each task separately, confirming our second research question. This increase in efficiency is shown more directly in 7. In this figure we combine the results of our pre-trained and fine-tuned agents and compare them with the baseline agents after an equal number of environments steps. We show the train return in levels $\mathcal{E}^f = \{\texttt{3-1}, \texttt{3-2}, \texttt{3-3}, \texttt{3-4}\}$. With one agent per level, each trained for $T = 1e7$ steps, it takes a total of $4 * 1e7$ steps to obtain the baseline results. With pre-training, we only need one agent to train on the four levels. Since we use the parameters of the pre-trained model directly during fine-tuning, we can extend the pre-train learning curve with the returns of the four agents fine-tuned on these levels. We append the learning curve of the first $\frac{3}{4} * 1e7$ time steps of each agent, to roughly match the total computation time of the baseline agents.

Focusing on the first $1e7$ time steps, we see that a single DreamerV2 agent performs fairly close, and even slightly better than the baseline agents, while pre-training on four levels. While the train return curve of the baseline already starts to flatten at $1e7$ time steps, our fine-tuning agents are able to keep improving for longer, creating a clear gap in the train return.

When pre-training on ten levels the results seem a bit worse. Especially during pre-training, we see that the efficiency is much lower than the baseline results. Of course, it would be more fair to compare these pre-training results against ten agents instead of four, but we can see that even in that comparison, the baseline would likely outperform our pre-training agent. After pre-training, we see the performance improve dramatically. In just $0.5e7$ steps, the fine-tuning agents have caught up with the baseline agents. A different way to interpret this result is that with this configuration, the overhead of pre-training on ten levels for $T$ steps was won back after fine-tuning on four of those ten levels. Assuming the results are similar for the other six levels, fine-tuning on more levels would proportionally shrink the pre-training part of the graph, so that it would display an efficiency advantage over the baseline.

### 7.3 Overfitting

Besides directly improving the efficiency of solving a large problem, we expected that learning multiple sub-tasks at the same time would also help against overfitting. Though additions such as sticky actions already try to prevent that the agent optimizes a trajectory directly – instead of learning to react on its inputs, we find that in practice, overfitting to a specific task is still a big problem. We demonstrate this with practical examples we observed during our experiments. Figure 8 shows five frames from the start of a trajectory in level `1-3`, and the reconstruction of those frames by three agents which were not trained on this level. The figure reveals clear signs of overfitting; all of the reconstructed frames contain artifacts resembling observations from levels the agent was trained on.

We note that the reconstructions by the baseline agent pretty closely resemble full frames that appear in level `4-1`, on which the agent is trained. We hypothesize that the encoder of this agent learns to recognize *where* in the level a frame is, which would be enough for the image predictor to predict most of the features on that frame. If this is the case, we should be able to significantly reduce the latent state size without impacting the performance much. This behavior would clearly be bad for generalization, as in this case, the agent would not learn to react to different features like obstacles or enemies. In unseen scenario's, the encoder would be unable to encode much useful information into the embedding, and in turn, very little of the learned policy would apply.

The reconstructions by the agent pre-trained on $\mathcal{E}_4^{pre}$ are the worst. This is likely due to the big difference between `1-3` and the levels in $\mathcal{E}_4^{pre}$. All four levels in the pre-training set are set on a black background, which means that at least half of each frame consists of inputs that the agent has never seen before. We still see clear evidence of overfitting of the world model. For example, we attribute the quality of the reconstruction of the level information at the top to overfitting, instead of an ability to transfer these detailed features from the input onto a new background. However, the reconstructed frames do not clearly correspond with specific frames from observed levels. This could be a sign that its latent space is less dependent on the location within a level, and tries to encode the appearance and location of features more directly. A generous eye may even see

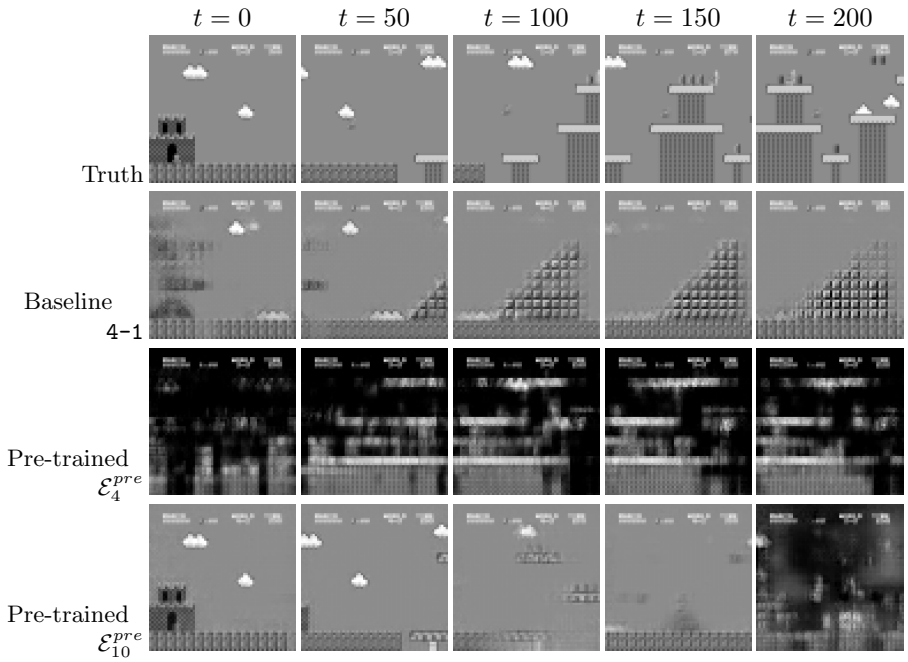



Figure 8: Five frames taken from a trajectory in `1-3`, and their reconstructions by agents trained on one level (`4-1`), 4 levels ($\mathcal{E}_4^{pre}$) and 10 levels ($\mathcal{E}_{10}^{pre}$)

the model trying to predict platforms (though not in the right places), a feature that is not especially frequent in the levels in $\mathcal{E}_4^{pre}$. An alternative explanation is that the encoded state just looks too different from any seen frame, resulting in a noisy prediction.

Finally, the frames predicted by the agent pre-trained on $\mathcal{E}_{10}^{pre}$ seem the most convincing at first sight. The first frame is a very close reconstruction, though in this case this reconstruction does seem to be recalled from an existing frame. This is evidenced by the floor gap in the bottom right of the frame – making it a closer reconstruction of the first frame of level `4-3` from the pre-training set than the actual observed level. The second frame also seems to be a recalled from a frame from that level. Interestingly, the gap in the floor aligns exactly between the observed and predicted frame, hinting that this gap must be a feature that the agent recognized, and prioritized over (for example) the position of the clouds. In the last three frames we see prediction quality drop dramatically. In the third frame we can barely recognize the platform in the bottom left and the cloud in the top center of the frame, but on the whole the frame is much noisier than the previous ones. This shows that the agent was more uncertain identifying this frame. The last two predicted frames do not clearly correspond to frames of seen levels. We observe that in the last predicted frame, the agent even becomes uncertain about the background color.

This is somewhat surprising, as it would be easy to construct a binary feature to encode this in the latent state.

Overall, we see clear evidence of overfitting of the world model, both in the baseline agents and our pre-trained agents. We hoped that the increased input diversity would make the pre-trained agents more robust against overfitting. Based on these reconstructions of frames from unseen levels, this seems at best only slightly so. It is of course possible that DreamerV2 would overfit less *much more* data diversity, but we would not expect to see dramatically different results even with all 32 Super Mario Bros. levels. Increasing the data diversity beyond that is possible through for example artificial data augmentation, but falls out of the scope of this work.

Overfitting does not only lead to more fragile policies, but also makes reusing experience in new situations, through meta learning or transfer learning, much harder. This may also explain why our agent is not able to effectively transfer its experience to new levels. This answers our third research question; we cannot improve the transfer learning capacity of DreamerV2 by pre-training on a range of related tasks. Even when the number of tasks is too large for DreamerV2 to learn, we see that some components still overfit. Perhaps this would be different if the capacity of the different components were balanced better, or if we had used a model with fewer components, and thus fewer possibilities for bottlenecks.

However, we observe that, since the train returns closely correspond to how much of a level the agent has seen, we know that the agent pre-trained on $\mathcal{E}_{10}^{pre}$ only saw a small part of most of the levels in its pre-training set. So even though the model has seen the starts of levels `3-3` and `3-4`, the relatively good performance during fine-tuning seem to show that our agents were able to quickly adapt to new parts of these levels.

# 8 Generalization of results

As discussed, we focused on Super Mario Bros. in our work because of the computational cost of training Atari games, and because Super Mario Bros. levels are much more similar. We thought this was needed especially in the transfer learning setting. As we saw, even when the tasks are this closely related, transfer learning is not a straight-forward task. However, for our pre-training/fine-tuning method, we can still ask how much our results rely on the close relations between the different levels.

In our initial explorations, we also tried our method on the Atari games, with four and ten different games from the Atari 51 benchmark. The games we tried varied much more than the Super Mario Bros. levels. Though our exploration lacked robustness, our initial results seemed in line with our pre-training and fine-tuning results in Super Mario Bros.

If our method indeed shows to improve efficiency in Atari, this would indicate that our method does not heavily rely on generalizing between the different environments. Most Atari games share only very coarse features. An alternative

explanation that fits with our intuition, is that the effectiveness of fine-tuning stems from a mismatch in model capacity; the world model seems to have a much larger capacity to learn different environments than the actor-critic network does. This may not be entirely surprising, as the individual components such as the image predictor have proven themselves in problem domains with a much higher data variety than games like Super Mario Bros.

Following this intuition, we suggest another extension to methods like DreamerV2 in settings with multiple tasks that share an MDP structure; instead of training one agent on ten tasks, we could also modify DreamerV2 to have one world model shared between ten actor/critic networks. This would be a relatively minor change to the implementation of DreamerV2, and we could use a similar training method as used in this work. Since the world model makes up the largest part of DreamerV2, we expect that this should still lead to significant performance improvements over training ten full DreamerV2 agents.

# 9 Conclusion

In our work we looked at RL problems which can be divided in clear, adjacent sub-problems. Instead of solving the whole problem end-to-end, or training a separate agent per sub-problem, we explored if grouping tasks can improve efficiency, and reduce overfitting for better generalization and transfer learning. Because our sub-problems are episodic and share the same state space structure and action space, we can combine sub-problems together easily, by randomly choosing one of the sub-problems at the start of each episode.

We found that a single DreamerV2 agent can keep up with four separate agents when trained on four Super Mario Bros. levels in this fashion, while ten levels was clearly beyond the capacity of the method we used. This is an indication that grouping problems like this can be used to scale the complexity of a problem to most effectively use the capacity of a method.

Secondly, we found that grouping sub-problems together is an effective basis for pre-training an agent. Even when pre-trained on ten levels, which was too much to learn directly, we see a clear increase in performance and data efficiency when fine tuning. These results are directly applicable to many real world scenario's.

Finally, we found that this pre-training strategy does not directly help in unseen adjacent sub-problems. A closer look shows that even after pre-training we still see clear signs of overfitting, both of the world model and the policy.

Compared to other multi-task RL literature, our work shows that even without multi-task-specific modifications, it can be fruitful to group sub-tasks and learn tasks simultaneously. The pre-training and fine-tuning scheme is easy to implement in practice, can be used with any method, and can significantly improve efficiency over training separate instances for multiple tasks.

# References


[1] Johannes Ackermann, Oliver Richter, and Roger Wattenhofer. "Unsupervised Task Clustering for Multi-task Reinforcement Learning". In: *Machine Learning and Knowledge Discovery in Databases. Research Track*. Ed. by Nuria Oliver et al. Cham: Springer International Publishing, 2021, pp. 222–237. ISBN: 978-3-030-86486-6.

[2] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. "Neuronlike adaptive elements that can solve difficult learning control problems". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.5 (1983), pp. 834–846. DOI: 10.1109/TSMC.1983.6313077.

[3] Marc G. Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *J. Artif. Int. Res.* 47.1 (May 2013), pp. 253–279. ISSN: 1076-9757.

[4] Chelsea Finn, Pieter Abbeel, and Sergey Levine. "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks". In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, June 2017, pp. 1126–1135. URL: https://proceedings.mlr.press/v70/finn17a.html.

[5] C. Daniel Freeman, Luke Metz, and David Ha. "Learning to Predict Without Looking Ahead: World Models Without Forward Prediction". In: *NeurIPS*. 2019.

[6] David Ha and Jürgen Schmidhuber. "World Models". In: (2018). DOI: 10.5281/ZENODO.1207631. URL: https://zenodo.org/record/1207631.

[7] Danijar Hafner et al. "Dream to Control: Learning Behaviors by Latent Imagination". In: *arXiv preprint arXiv:1912.01603* (2019).

[8] Danijar Hafner et al. "Mastering Atari with Discrete World Models". In: *arXiv preprint arXiv:2010.02193* (2020).

[9] Matteo Hessel et al. "Multi-task Deep Reinforcement Learning with PopArt". In: *AAAI*. 2019.

[10] Irina Higgins et al. "beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework". In: *ICLR*. 2017.

[11] Shikun Liu, Edward Johns, and Andrew J. Davison. "End-To-End Multi-Task Learning With Attention". In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019), pp. 1871–1880.

[12] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (2015), pp. 529–533.

[13] Emilio Parisotto, Jimmy Ba, and Ruslan Salakhutdinov. "Actor-Mimic: Deep Multitask and Transfer Reinforcement Learning". In: *ICLR*. 2016.

[14] J. Schmidhuber. "An On-Line Algorithm for Dynamic Reinforcement Learning and Planning in Reactive Environments". In: *Proc. IEEE/INNS International Joint Conference on Neural Networks, San Diego*. Vol. 2. 1990, pp. 253–258.

[15] Julian Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (Dec. 2020), pp. 604–609. DOI: 10.1038/s41586-020-03051-4. URL: https://doi.org/10.1038/s41586-020-03051-4.

[16] John Schulman et al. "High-Dimensional Continuous Control Using Generalized Advantage Estimation". In: *CoRR* abs/1506.02438 (2016).

[17] Marwin HS Segler, Mike Preuss, and Mark P Waller. "Planning chemical syntheses with deep neural networks and symbolic AI". In: *Nature* 555.7698 (2018), pp. 604–610.

[18] Yee Whye Teh et al. "Distral: Robust multitask reinforcement learning". In: *ArXiv* abs/1707.04175 (2017).

[19] Ronald J. Williams. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Machine Learning* 8 (2004), pp. 229–256.