university of groningen

faculty of science and engineering

Multi-Source Transfer Learning for Deep Model-Based Reinforcement Learning

Remo Sasso

university of groningen

faculty of science and engineering

University of Groningen

Multi-Source Transfer Learning for Deep Model-Based Reinforcement Learning

Master's Thesis

To fulfill the requirements for the degree of
Master of Science in Artificial Intelligence
at University of Groningen under the supervision of
Dr. Marco Wiering (Artificial Intelligence, University of Groningen)
and
Matthia Sabatelli (Electrical Engineering and Computer Science, University of Liege)

Remo Sasso (s2965917)

July 8, 2021

Contents

Abstract

Reinforcement learning (RL) is well known for requiring large amounts of data for agents to learn tasks. Recent progress in model-based RL (MBRL) allows agents to be much more data-efficient, as it enables agents to learn behaviors in imagination by leveraging an internal World Model of the environment. Improved data-efficiency can also be achieved by reusing knowledge from previously learned tasks, but transfer learning is still an emerging topic for RL. In this research, we propose and investigate novel transfer learning approaches for deep MBRL. Moreover, rather than transferring knowledge from a single source, this work focuses on multi-source transfer learning techniques. First, we propose transferring knowledge of agents that were trained on multiple tasks simultaneously. In addition, where common transfer learning techniques use an all-or-nothing approach for transferral of neural network layers, we present fractional transfer learning as an alternative approach. Next, we introduce meta-model transfer learning, which is a technique that allows the combination and transferral of knowledge for multiple individual sources in a universal feature space. Finally, we present latent task classification, which enables agents to classify previous tasks and detect novel tasks in latent space. We perform experiments for each of the proposed methods by applying them to a state-of-the-art MBRL algorithm (Dreamer). The results verify the abilities of latent task classification, and also show that each of the proposed transfer learning methods can lead to significantly faster and/or better learning performances.

***Keywords*—** Multi-Source Transfer Learning; Model-Based Reinforcement Learning; Deep Learning

1 Introduction

Reinforcement learning (RL) [1, 2] is a branch of artificial intelligence (AI) research that focuses on developing algorithms which enable agents to learn tasks in an environment they are situated in. RL agents learn tasks by means of maximizing the rewards they receive from an environment. Rewards result from transitions in the environment that are initiated by the agent taking actions. This type of learning can be a powerful tool for creating intelligent agents that become able to master complicated tasks, occasionally resulting in superhuman capabilities. For instance, the RL based AlphaZero algorithm learned to master the games of Go, shogi, and chess at a superhuman level [3]. However, in order for RL agents to obtain these capabilities, the amount of interaction with the environment necessary can be enormous. An important issue in RL research is therefore sample efficiency, defined as how much interaction is necessary for the agent to master the task at hand.

In model-based RL (MBRL), the agents have access to an internal model of the world [4]. This model allows the agent to predict transitions in the environment that result from its actions, without having to interact with the environment. The model therefore allows agents to perform planning, as well as generating internal trajectories that can be used for learning, resulting in improved sample efficiency. However, the effectiveness of this type of approach depends on the quality of the model, which often needs to be learned. Better sample efficiency can also be accomplished by reusing knowledge of previously learned tasks. Transfer learning (TL) can therefore be a key component in advancing the field of RL research, but as of writing TL is still an emerging topic in RL [5].

In the past decades, a surge of Deep Learning (DL) has significantly advanced the field of AI. DL is a set of tools that make use of artificial neural networks, which are capable of extracting features from high-dimensional data. The learned feature representations can be used to make predictions about unseen before data, meaning neural networks have the ability to generalize beyond their training data. Moreover, they are able to do so with many types of data, such as images and text. By making use of DL techniques, several of the most successful (deep) RL applications could be developed. Popular examples include RL systems that were able to beat the best human players in the games Go [6], Starcraft II [7], and Dota 2 [8].

Recently, MBRL has also significantly progressed by making use of DL techniques, resulting in state-of-the-art algorithms for both discrete and continuous benchmarks [3, 9, 10]. An important advancement in terms of sample efficiency was that of World Models, which allow MBRL agents to learn behaviours by imagining trajectories of environments with high-dimensional visual observations [11]. This type of approach provides a strong foundation for TL and multi-task learning, as World Models are of generalizable nature for different environments sharing the same physical laws [12, 13]. However, there is very few research that has been carried out for combining MBRL algorithms that make use of World Models with TL and multi-task learning.

In this thesis we aim to provide insights and novel approaches for combining the aforementioned concepts. That is, we investigate the possibilities of combining multi-task learning and TL with a state-of-the-art World Model based MBRL algorithm, Dreamer [9]. In the remainder of this chapter we present the research questions, as well as the structure of this thesis.

1.1 Research Questions

In this study we aim to answer the following questions:

1. Does transferring knowledge of deep model-based reinforcement learning agents trained on multiple tasks simultaneously serve as an effective multi-source transfer learning approach?

2. Can transferring fractions of neural network parameters be used as an alternative transfer learning approach for certain models?

3. How can knowledge of multiple separate deep reinforcement learning models be combined to serve as a multi-source transfer learning technique?

4. How can both detection of novel domains and classification of known domains be accomplished in a multi-task setting for deep model-based reinforcement learning architectures?

1.2 Thesis Outline

The structure of this thesis is as follows. In this chapter we provided a brief introduction to this research, after which we presented the research questions. Chapter 2 details on the theoretical concepts used in this thesis, and discusses relevant related works. In Chapter 3 we propose several new methods and provide the technical details of each of them. Chapter 4 describes the experimental setups that are used for the evaluation of all of the proposed new methods. Next, in Chapter 5 we present and describe the results of the aforementioned experiments. Finally, in Chapter 6 we answer the research questions of this thesis whilst discussing the relevant results, after which we summarize the thesis by stating the main contributions, and finally propose future directions of research.

2 Theoretical Background

In this chapter we detail on all theoretical concepts related to this thesis and discuss relevant related works. We start off by providing a brief introduction to RL (Section 2.1). This is followed by a description of the workings and usages of MBRL (Section 2.2). We then provide a brief introduction to DL, and how this is generally used in the context of RL (Section 2.3). Next, we also provide a brief introduction to TL, after which we discuss how it can be used for RL (Section 2.4). Finally, we discuss the conclusions of this chapter (Section 2.5).

2.1 Reinforcement Learning

RL is a principled mathematical framework that is considered as a sub-field of AI. In this framework autonomous learning agents are used to solve sequential decision-making tasks. The agent interacts with an environment in which the task takes place, such that it can improve its performance in solving the task by means of trial and error (2.1.1). This framework is often theoretically formalized as a Markov Decision Process (2.1.2). The ultimate goal is for the agent to learn an optimal sequence of actions (referred to as optimal policy) that solves the task at hand, by means of a policy learning algorithm (2.1.3). Dependent on the complexity of the problem, different solution methods can be used to allow these algorithms to work (2.1.4).

2.1.1 Environment Interaction

Environment interaction is an essential element of RL. For a given point in time, the current state of the environment is observed by the agent, based on which it takes an action. This results in a transition of environment states, which also yields a reward signal. The reward signal is a metric that represents the performance of solving the problem at hand, which is therefore used by RL agents to learn what action to take in a given state. That is, in order to find an optimal sequence of actions for solving a task, the agent attempts to maximize the expected (cumulative) reward. This means that agents take an action based on what they expect the total reward to be which results from the state transitions that follow from it. However, not all environments are deterministic, and therefore it may be difficult to predict future transitions and rewards.

A crucial component of this learning process is the ability of the agent to optimally balance exploiting actions it currently believes maximize the expected cumulative reward, and exploring actions that may improve the maximization even further. This is known as the infamous exploration-exploitation dilemma. Agents employ exploration methods that guide them in exploring new states and actions of the environment. The sample efficiency of these techniques has a large impact on the ability and swiftness of learning to solve a given task. An example of a popular exploration technique is ε-greedy. Using this method, the agent explores by taking a random action with a probability of ε, otherwise it uses the action that it currently believes maximizes the expected cumulative reward. Even though this type of method will eventually lead to an optimal policy, its random nature causes redundant environment interactions to take place. Despite this sample inefficiency, ε-greedy remains a popular technique due to its simplicity and computational efficiency. In contrast, more sophisticated exploration methods often require more computation, which therefore don't scale well with large and complex problems [14].

2.1.2 Markov Decision Process

RL problems are typically formalized as a Markov Decision Process (MDP) [15], which is a mathematical framework for discrete-time stochastic control processes. Sequential decision making problems, as encountered in RL, can conveniently be defined as an MDP.

MDPs consist of four main elements:

- a set of states S
- a set of actions A
- a transition function $P(s,a,s') = \Pr(s_{t+1} = s' | s_t = s,\, a_t = a)$

- a reward function $R(s,a,s')$

Here $P(s,a,s')$ denotes the probability that taking action $a \in A$ in state $s \in S$ at time step t results in a transition to state $s' \in S$, where a time step is defined as a discrete point in time of the sequential decision making process. When a state transition takes place, the reward function $R(s,a,s')$ yields an immediate reward corresponding to this transition. Though the number of states $|S|$ and actions $|A|$ can be infinite in certain problems, we will limit this discussion to finite MDPs.

The objective in an MDP is to find a so called policy $\pi^*(s_t) = a_t$ that optimally maximizes the expected cumulative reward. That is, for a given state s, a policy π specifies which action a to take in time step t. The expected cumulative reward is defined as:

$$E\left[\sum_{t=0}^{T} \gamma^t R(s_t, a_t, s_{t+1})\right] \tag{1}$$

Here γ represents the discount factor, where $\gamma \in [0,1)$. In order to weight rewards in the distant-future less than rewards in the immediate future, γ takes on values less than 1. In scenarios where $\gamma = 0$ the agent only considers immediate rewards, and is therefore considered myopic. T, the horizon, represents the number of time steps to account for in computing the expected cumulative reward. When dealing with an infinite horizon problem ($T = \infty$), the discount factor allows us to avoid infinite returns.

2.1.3 Policy Learning

There exist different types of algorithms that can be used in order to learn policies. Learning policies requires the usage of state valuations, which is a way of judging how good it is for the agent to be in a certain state. A valuation function $V^\pi(s)$ is expressed as the expected reward of a given state s that results from following a policy π:

$$V^\pi(s) = E_\pi\left[\sum_{t=0}^{T} \gamma^t R(s_t, a_t, s_{t+1}) | s_t = s\right] \tag{2}$$

Similarly, a valuation (or quality) function $Q^\pi(s,a)$ can be defined for state-action pairs. In this case the function is expressed as the expected return of taking action a in state s, and following policy π afterwards:

$$Q^\pi(s,a) = E_\pi\left[\sum_{t=0}^{T} \gamma^t R(s_t, a_t, s_{t+1}) | s_t = s, a_t = a\right] \tag{3}$$

Note that valuation functions are of recursive nature, as:

$$\begin{aligned} V^\pi(s) &= E_\pi\left[R(s_t, a_t, s_{t+1}) + \gamma^{t+1} R(s_{t+1}, a_{t+1}, s_{t+2}) + ... | s_t = s\right] \\ &= E_\pi\left[R(s_t, a_t, s_{t+1}) + \gamma^{t+1} V^\pi(s_{t+1}) | s_t = s\right] \\ &= \sum_{s' \in S} P(s, \pi(s), s')\left[R(s, \pi(s), s') + \gamma V^\pi(s')\right] \end{aligned} \tag{4}$$

```
Initialize $V_0(s)$ to arbitrary values for $s \in S$
$\sigma \leftarrow$ a small positive number
**repeat**
    $\delta \leftarrow 0$
    **for** $s \in S$ **do**
        **for** $a \in A$ **do**
            $Q(s,a) \leftarrow \sum_{s' \in S} P(s,a,s') \left[R(s,a,s') + \gamma V_i(s')\right]$
        **end**
        $V_{i+1}(s) \leftarrow \max_a Q(s,a)$
    **end**
    $\delta \leftarrow \max(\delta, \sum_{s \in S} |V_i(s) - V_{i+1}(s)|)$
**until** $\delta < \sigma$

**Output:** $\pi(s) \leftarrow \text{argmax}_a \sum_{s' \in S} P(s,a,s') \left[R(s,a,s') + \gamma V_i(s')\right]$
```

Algorithm 1: Value iteration

In Equation 4 we eventually defined the valuation function as a Bellmann Equation [16]. This states that the expected valuation of a state is expressed in terms of the immediate reward and discounted valuations of all possible next states weighted by their transition probability.

As stated before, the objective of an MDP is to find an optimal policy π^*(s), being the policy that optimally maximizes the expected reward, and therefore maximizes the valuation function (Equation 2). That is, in order for an optimal policy to provide the optimal actions, an optimal value function needs to be present. An optimal state value function $V^*(s)$, as well the state-action pair variant $Q^*(s,a)$, can be defined as a Bellmann optimality equation:

$$V^*(s) = \max_a \sum_{s' \in S} P(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right] \tag{5}$$

$$Q^*(s,a) = \sum_{s' \in S} P(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')\right] \tag{6}$$

Consequently, we can express the optimal policy in terms of $V^*(s)$:

$$\pi^*(s) = \arg\max_a \sum_{s' \in S} P(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right] \tag{7}$$

For a given state, the optimal policy therefore yields the action $a \in A$ that results in the highest reward according to the optimal value function. When always taking the most rewarding actions, the policy is named greedy.

In order to learn the optimal value function and the optimal policy, policy learning algorithms iteratively update these functions. One such algorithm is called value iteration: we update the value function iteratively from which we can then derive a policy (Algorithm 1). The initial value function is initialized with random valuations for all states. What repeatedly follows, is that for all states the valuation is computed for each possible action using the Bellmann Equation. The valuation resulting from the action with the maximum valuation is then used for updating the corresponding state in the value function of the next iteration. Once the difference between the value functions of two subsequent iterations is small enough (i.e. converged), a policy is derived from the eventual value function.

```
Initialize $V_0(s)$ and $\pi_0(s)$ to arbitrary values for $s \in S$
$\sigma \leftarrow$ a small positive number
repeat
    /* Policy Evaluation                                        */
    repeat
        $\delta \leftarrow 0$
        for $s \in S$ do
            $V_{i+1}(s) \leftarrow \sum_{s' \in S} P(s, \pi_i(s), s') \left[R(s, \pi_i(s), s') + \gamma V_i(s')\right]$
        end
        $\delta \leftarrow \max(\delta, \sum_{s \in S} |V_i(s) - V_{i+1}(s)|)$
    until $\delta < \sigma$

    /* Policy Improvement                                       */
    for $s \in S$ do
        $\pi_{i+1}(s) \leftarrow \text{argmax}_a \sum_{s' \in S} P(s, a, s') \left[R(s, a, s') + \gamma V_i(s')\right]$
    end
until $\pi_{i+1}(s) = \pi_i(s)$

Output: $\pi_i(s)$ and $V_i(s)$
```

Algorithm 2: Policy iteration

An alternative approach is to make use of the policy iteration algorithm (Algorithm 2). This process consists of two steps: policy evaluation and policy improvement. In the policy evaluation step we repeatedly update the value function as was done in Algorithm 1, but instead of updating the valuation function for a state with the value resulting from the best action, we use the valuation resulting from taking an action according to the current policy. We do so until convergence of two subsequent value functions, after which we move to the policy improvement step. Here, we determine for each state the best action to take according to the converged valuation function, with which we then update the policy for the next iteration. These two steps are repeated until the policies of two subsequent iterations are equivalent, from which a converged policy and valuation function result.

2.1.4 Solution Methods

The algorithms described in the previous section can work very well in applications where the state and action spaces $|S|$ and $|A|$ are finite and small. However, computing the valuation or best action for all possible states becomes infeasible when these spaces are very large. For this reason there are two types of solution methods that are leveraged in RL: tabular solutions and function approximation solutions.

Q-learning [17] is an RL algorithm that serves as a good illustration to clarify the distinction between the aforementioned solution methods. Q-learning is a model-free RL algorithm, which means that it does not make use of a model of the environment, i.e. does not take transition probabilities into account. Due to its simplicity and effectiveness, Q-learning has been one of the most frequently researched algorithms in the RL community for the past decades [18, 19, 20, 21, 22, 23]. The algorithm learns state-action pair valuations (Q-values) by leveraging the Bellmann Equation in a temporal difference (TD) based update rule, as seen in Equation 8:

$$Q(s,a) = Q(s,a) + \alpha(R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)) \tag{8}$$

For a given state-action pair, the corresponding Q-value is updated by summing it with the TD that is controlled by a learning rate. The TD is the difference between the current Q-value and the summation of the immediate reward with the discounted estimate of the optimal future Q-value. The learning rate α is a parameter that determines how much a single update impacts the current Q-value, where $\alpha \in [0,1]$.

In applications where the state and action spaces are finite and small, the Q-value of each state-action pair can be stored in a lookup table, hence the name tabular solution. Using a tabular method, the exact valuations of state-action pairs can be accessed at any time, which generally allows for convergence to optimal value functions and optimal policies. However, for very large action and state spaces this will require too much memory and computation to be applicable. To counteract this, the usage of function approximation is necessary. Function approximators can learn to approximate a certain target function. This is most often done using supervised learning, which is a methodology to train function approximators using target data. That is, given a certain input, the function approximator makes a prediction and adjusts itself based on whether the prediction matches the corresponding target. Therefore, we can use a function approximator to learn to approximate a valuation function or policy function. In the case of Q-learning, given state-action pair inputs, the function approximator learns to predict Q-values. Whereas the function approximator may only have learned from a subset of all possible state-action pairs, it can generalize the learned predictions to novel state-action pairs that are similar to the ones encountered in the learning process.

2.2 Model-Based Reinforcement Learning

In the previous section we introduced a model-free algorithm that learns a policy and value function by directly interacting with the environment. The agent needs to take actions in order to be able to predict what the next state and reward will be. Alternatively, model-based algorithms provide agents with access to an internal model of the environment. This allows agents to make predictions about state transitions and corresponding rewards without having to interact with the environment (Section 2.2.1). By using this model as a reference, agents can learn to plan (a sequence of) actions (Section 2.2.2). The main advantages of these methods over their model-free counterparts, are vastly improved sample efficiency [24, 12], zero- and few-shot learning [25], and state-of-the-art performances on both continuous and discrete benchmarks [3, 9, 10]. Additionally, MBRL allows for important TL [5], multi-objective learning [26, 27], and exploration [28, 29, 30] approaches.

2.2.1 Transition Models

In an MDP, the transition function $P(s,a,s')$ and the reward function $R(s,a,s')$ define the model. There are different types of models that are used in MBRL depending on how the transition function is used:

- Forward model: $\mathcal{F}(s_t,a_t) = s_{t+1}$
- Backward model: $\mathcal{F}(s_{t+1}) = s_t,a_t$
- Inverse model: $\mathcal{F}(s_{t+1},a_t) = s_t$

That is, a model can predict what a next state will be, what state-action pair was used to arrive at a certain state, or what state preceded the transition. The prediction of the reward can be added by predicting an additional scalar. The forward model is most commonly used in MBRL and will also be the main focus of this work.

A model may be known or unknown, meaning in the latter case the model needs to be learned. Model learning is done by gathering state transition data from the environment and using it to learn the dynamics of the world. That is, for a small state space we can solve this in a tabular manner by storing the state transitions in a lookup table. However, for larger state spaces we will need to employ a function approximator. The function approximator, with some set of parameters θ, can learn to predict what the next state and reward will be given a state-action pair, i.e. $s_{t+1}, r_{t+1} = \mathcal{F}_\theta(s_t,a_t)$. The quality of the model is an essential factor of the functionality of a MBRL algorithm. If the model does not represent the environment properly, then model dependent functionalities will not be usable, such as planning (Section 2.2.2). Therefore, the model learning process is a crucial step if the model is unknown.

When designing a dynamics model, one has to consider several issues. First, the issue of uncertainty needs to be addressed. Aleatoric uncertainty is the issue of whether or not we are dealing with a stochastic MDP, in which case the state transitions will not be deterministic. Therefore, instead of returning a single next state, the transition model will need to return a probability distribution over all possible next states. We are dealing with epistemic uncertainty if we have a limited amount of data, meaning that the model will not be able to capture the underlying dynamics of the environment sufficiently, thus affecting the quality of the model. In this case it may therefore be useful to explicitly estimate the model uncertainty in order to be able to take it into account [31]. Another issue is that of when dealing with a Partially Observable MDP (POMDP). In this case the observation of a state does not capture the ground-truth state of the MDP. The MDP may also be non-stationary, meaning that the ground-truth transition and/or reward functions change over time. An example scenario where one deals with non-stationary MDPs, which is also the main focus of this thesis, is TL (Section 2.4). The reason for this is that the transition function or reward function change when the agent is presented with a new domain or new task respectively. Finally, the way we represent states for the model to use is a crucial component of being able to learn a good model, known as representation learning. We will discuss the latter issue in more detail in Section 2.3.

2.2.2 Planning

Once a model has been learned, it can be used to predict the total reward of a trajectory that results from using a certain policy or action. This means the highest rewarding policy or action can be selected from a set of candidates without having to interact with the environment. There are two types of planning: background and decision-time planning. With background planning the model is used to optimize for a general policy. For instance, Dyna [2] is a background planning method that uses the model to simulate environment data using which it trains a policy with model-free methods. Decision-time planning instead optimizes for a sequence of actions starting from the current environment state. Model Predictive Control (MPC) [32] is the best-known decision-time planning method, which repeatedly optimizes for a sequence of actions and only executes the first action of the returned plan. That is, background planning methods are useful for applications where long-term thinking is beneficial, whereas decision-time planning is useful in stochastic or non-stationary environments.

An important distinction that needs to be made in planning is that of discrete and continuous action spaces. With discrete action spaces an agent has a finite set of actions from which it can select one to execute. In continuous action spaces the action is represented as a real-valued vector, which can then be used for executing actions such as moving a torque of a robot arm [33]. When using decision-time planning methods this is an important issue to take into account, as different trajectory optimizations methods are used for each of the action spaces. For discrete action spaces Monte-Carlo Tree Search (MCTS) [34] estimates are generally used. Here, for a given state, the model is used to predict different trajectories in a tree search manner. That is, for a given state (root node), different actions are executed that expand the tree with new states. Whilst continuously doing so, a reward value and visit counter is kept track of for each action and state. Eventually the policy resulting in the highest rewarding sequence returned. For continuous action spaces we cannot perform a tree-search to find the optimal trajectory. Instead, we can take one or more randomly initialized action sequences for which we optimize using gradient descent [35]. We will further detail on gradient descent in Section 2.3.1.

2.3 Deep Reinforcement Learning

In this section we will briefly introduce the concept of Deep Learning (DL) (Section 2.3.1). The revolutionary advancements of DL in the past decades has provided powerful tools for RL that are used for representation learning and function approximation, allowing RL algorithms to scale to complex problems (Section 2.3.2). These tools have also allowed significant advancements in MBRL, such as learning and planning in imagination (Section 2.3.3).

2.3.1 Deep Learning

Before we can define DL, we first need to introduce the concept of the perceptron [36]. A perceptron is an information unit that computes a weighted sum of the inputs $[x_1, x_2, ...x_n]$ it receives, after which it produces an output $\hat{y}$:

$$\hat{y} = \sum_{j=1}^{n} x_j w_j + b \tag{9}$$

By having classification or regression as objective, a perceptron can be trained such that it learns to map a certain input to a corresponding output. In order to train, the output of the perceptron is compared to some target y, resulting in an error quantity. For instance, when the objective is regression, we can compute the error, or loss, L using the Mean Squared Error (MSE) as follows:

$$L(y,\hat{y}) = \frac{1}{2}(y-\hat{y})^2 \tag{10}$$

We can then compute how we should update each of the weights by computing the corresponding partial derivatives, or gradient:

$$\frac{\partial L(y,\hat{y})}{\partial w_j} = -(y-\hat{y})x_j \tag{11}$$

Afterwards, we can update the weights by using a learning rate α that determines how influential each update will be:

$$w_j = w_j - \alpha(\frac{\partial L(y,\hat{y})}{\partial w_j}) \tag{12}$$

We can learn more complex and non-linear objective functions by adding so-called hidden layers to the perceptron, in which case we are speaking of a multi-layer perceptron (MLP), see Figure 1. A hidden layer takes as input the output of a preceding layer. However, in order to cope with non-linear problems, so-called non-linear activation functions need to be applied to the output of a given layer. An example activation function σ is the Rectified Linear Unit (ReLu), where the maximum between the output and zero is computed:

$$\sigma(x) = max(0,x) \tag{13}$$

In similar fashion to the single-layer perceptron, we can train the MLP by computing a vector consisting of the partial derivatives of the weights based on the error $L(y,\hat{y})$, which we call the gradient:

$$\nabla_W L(y,\hat{y}) = \left[\frac{\partial L(y,\hat{y})}{\partial w_1}, \frac{\partial L(y,\hat{y})}{\partial w_2}, \ldots, \frac{\partial L(y,\hat{y})}{\partial w_n}\right] \tag{14}$$

However, as we are dealing of weights from multiple layers, we need to compute intermediate weight update values derived from the output layer using the chain rule. This process is called backpropagation, as we propagate the error of the output layer back through the earlier layers. The gradient is then used to update all of the weights using Equation 12 in order to minimize the error. For this reason, this optimization method is named gradient descent. We are speaking of DL when we are using ANNs that consist of more than one layer. Two of the most important DL architectures that were invented in recent years are the Convolutional Neural Network (CNN) [37] and the Recurrent Neural Network (RNN) [38].

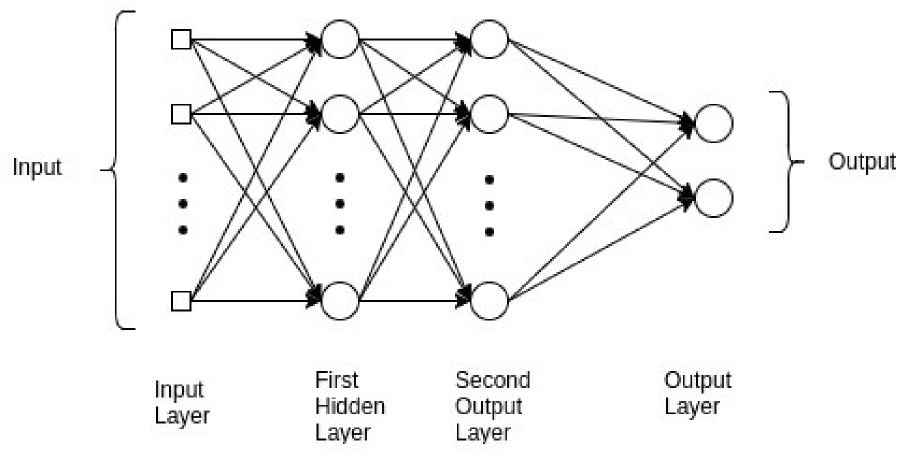


Figure 1: A schematic diagram of a basic multi-layer perceptron architecture. Source: [39].

A CNN is able to process pixel inputs, meaning we can train it to map image inputs to corresponding outputs. The core mechanism of a CNN relies on shifting each of a set of filters (or kernels) along each channel of an input image, see Figure 2. At a given position in the image, a kernel is convoluted with the pixels located in that region, after which the output is stored in a so-called feature map. The objective here can be interpreted as each of the filters learning to identify a certain feature at any location in the input image, such as a shape. These feature maps are usually fed to a pooling layer, which reduces the size of the feature map even further by taking the maximum or average of certain regions. This process repeats itself throughout the network, resulting in feature maps of continuously reduced sizes, meaning more abstract features are extracted later on in the network. Eventually, the extracted features are fed to a fully-connected layer that assembles the final feature maps into a vector space that can be used to make predictions, such as classification. Using Equation 14 the weights of all the layers in the network can be updated, such that earlier layers can learn what features to extract from the input image in order to minimize the error.

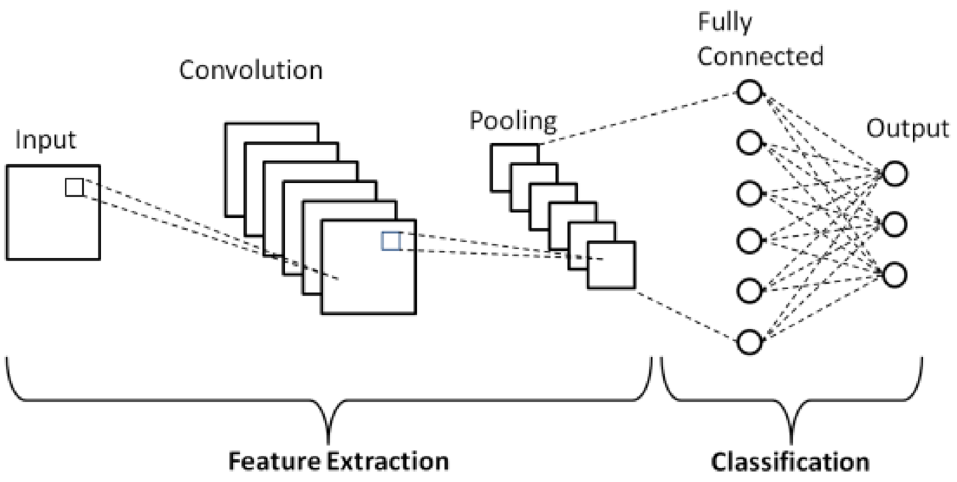


Figure 2: A schematic diagram of a basic convolutional neural network architecture. Source: [40].

Instead of feeding the extracted features to a fully connected layer, it is also possible to feed this information to a CNN inversely, resulting in an autoencoder (AE) [41]. When doing this, the extracted (encoded) features are fed to dilated convolutional layers, which output feature maps of increased dimensions (Figure 3). By having the input equivalent to the target output, the AE can learn to reconstruct the input image from the encoded lower dimensional space. This lower dimensional space, or latent space, will thus be able to represent input images in a compressed manner.

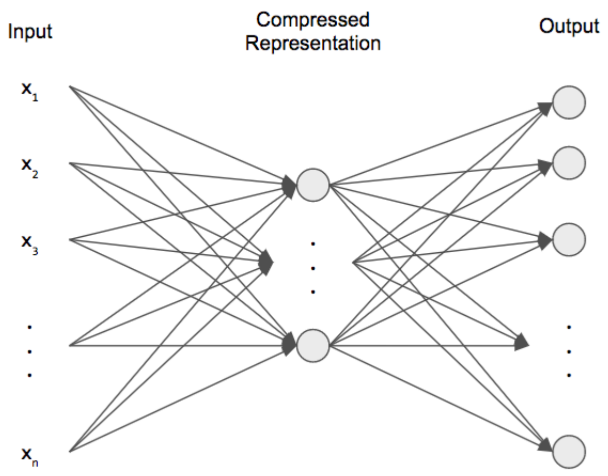


Figure 3: A schematic diagram of a basic autoencoder architecture. Source: [42].

RNNs are known for their ability to process and make predictions on sequential data, such as text. The

RNN architecture is designed such that each layer of the RNN processes a part of the input sequence. In addition to an input element, each layer also receives the hidden state of the previous layer as input (Figure 4). Alternatively, one can pass on the output of a previous layer rather than the hidden state. By combining the input and information from a previous layer the network is able to take temporal relations into account of the input sequence. If we provide hidden states of previous layers as input, the weighted sums can formally be defined as:

$$\begin{aligned} h_t &= \sigma_h(x_t W + h_{t-1} U + b_h) \\ y_t &= \sigma_y(h_t V + b_y) \end{aligned} \tag{15}$$

Here W, V, and U are the shared weights across all layers for input, output, and hidden connections respectively, which is a characteristic of RNNs.

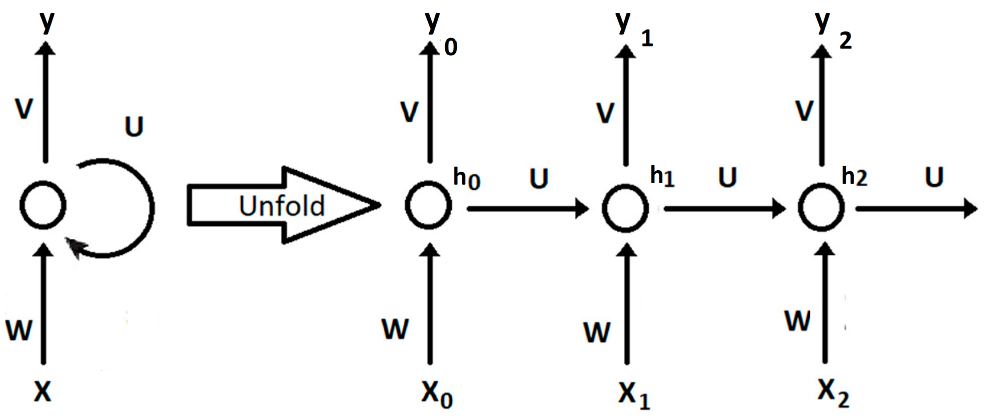


Figure 4: A schematic diagram of a basic recurrent neural network architecture. Adapted from [43].

2.3.2 Deep Learning in Context of Reinforcement Learning

In previous sections we have discussed how value and policy functions for large state spaces can be approximated using function approximation. Neural networks are a very popular means of function approximation, as was shown in the breakthrough paper [44]. The authors used TD learning in combination with self-play in order to develop an agent that could eventually play backgammon at a human expert level, where an an MLP was used as function approximator. In later years DL architectures such as CNNs and RNNs also sparked the interest of the RL community, which resulted in Deep RL (DRL). DRL became particularly popular since the introduction of Deep Q-Networks (DQN) [45], where CNNs were used to process high-dimensional raw image data for a set of 49 different Atari games. This type of representation learning resulted in Q-learning agents that achieved a level comparable to professional human players in each of the games, solely based on a reward signal and image observations. The next big advancement in DRL was AlphaGo [6], which beat the world's best human player in the complex boardgame Go. AlphaGo was a hybrid DRL system that used search heuristics in combination with neural networks that were trained using supervised learning and RL. AlphaGo later evolved into AlphaZero [46], which learned to play Go, Chess and Shogi without any supervised learning from human data. The final iteration of these works, MuZero [3], resulted in state-of-the-art performances in Go, Chess, Shogi, and the Atari benchmarks by using MBRL, meaning the rules of the games were unknown to the agent and had to be learned. DRL has also been shown to be able to master highly complex modern games, such as Starcraft II [7] and Dota 2 [8]. In both of these games the world's best players were beaten by the DRL systems.

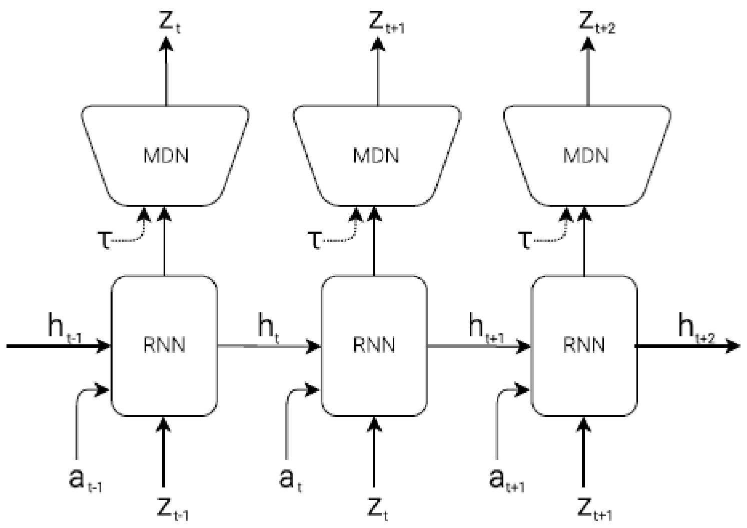


Figure 5: MDN-RNN architecture for World Models. Source: [11].

2.3.3 World Models

Whereas model-free methods could be used to directly learn from high-dimensional pixel images, MBRL faced the difficulty that a transition model is hard to train from such high-dimensional data. MBRL was shown to be efficient and successful for small sets of low-dimensional data, such as PILCO [31], where the authors used Gaussian processes to learn a model of the environment from which trajectories could be sampled. However, methods like Gaussian processes for learning a transition model don't scale to high-dimensional observations. One work showed that by first using an AE to learn a lower-dimensional representation of image data, a controller could be trained to balance a pendulum using the feature vectors provided by the bottle-neck hidden layer of the AE as information [47]. This usage of compressed latent spaces allowed much more sample efficient RL methods to be developed [48, 49].

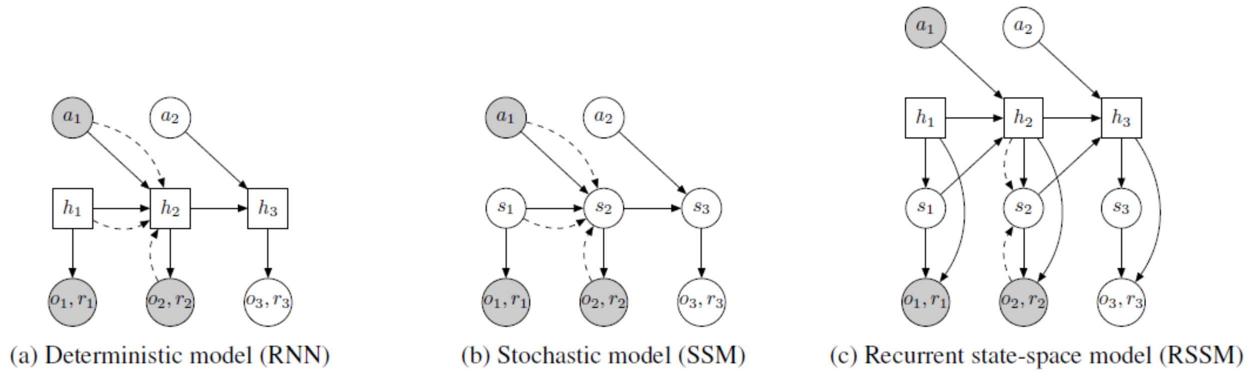


Figure 6: A purely deterministic model (a), a stochastic model (b), and the proposed RSSM model (c). Circles represent stochastic variables and squares deterministic variables. Solid lines denote generation and dashed lines inference. The third time step is predicted. Source: [12].

More recently, World Models [11] were introduced that allowed MBRL agents to do policy learning without interacting with visual environments. The authors first trained a Variational AE (VAE) to learn a compressed representation of randomly gathered image data. Opposed to learning a function to compress the inputs, as is done with vanilla AEs, VAEs learn the parameters of a probability distribution representing the data. Using the VAE produced latent space, an RNN is trained that takes as input latent state z_t, an action a_t, and hidden state h_t to predict a probability distribution of the next latent state z_{t+1}. The RNN is combined with a Mixture Density Network (MDN), such that a temperature parameter can be introduced that allows controlling of the model uncertainty when sampling from the distribution (Figure 5). After training, the MDN-RNN model allowed 'imagined' trajectories to be sampled that represent the actual environment. The reward functions of these environments can be assumed, as, for instance, the VizDoom environment that was used provides rewards for the amount of elapsed time. A linear controller policy could therefore be learned without interacting with the actual environment, after which the policy was able to be transferred to the actual environment. The authors of SimPLe extended this approach, where they showed that a World Model can be used as a simulator to train model-free policies, resulting in impressive sample efficiency [50].

Rather than learning a World Model and policy in separate stages, PlaNet trains these processes jointly and performs latent online planning for visual continuous control tasks [12]. Moreover, instead of assuming the reward function of the environment, PlaNet learns a model of the reward as a function of the latent states. The latter allows planning to be performed completely in imagination. For planning, the authors combine MPC with the cross entropy method (CEM). The latter is a population-based optimizer that iteratively evaluates a set of candidate action sequences under the model that are sampled from a belief distribution, which is repeatedly re-fit with the top K rewarding action sequences. As transition model architecture, the authors introduce the Recurrent State Space Model (RSSM). This model combines a deterministic and stochastic model into one architecture. Purely deterministic models, such as a vanilla RNN, cannot capture multiple futures, and tend to exploit model inaccuracies. Purely stochastic models have difficulties remembering information over multiple time steps, given the randomness in transitions. The authors split the latent state in stochastic and deterministic parts, allowing robust prediction of multiple futures (Figure 6). This work provided the foundation for the state-of-the-art MBRL continuous control algorithm Dreamer [9], as well as its variant that is adapted for discrete problems, DreamerV2 [10]. Dreamer will be the main focus of this work and explained in detail in Section 3.1.

2.4 Transfer Learning

This section introduces the concept of TL. We define TL and detail on how it is often used for AI algorithms (Section 2.4.1), after which we introduce how the concept can be applied in the RL framework (Section 2.4.2). Finally, we cover prior research that have applied TL to MBRL, with a main focus of approaches using World Models (Section 2.4.3).

2.4.1 General Transfer Learning

The idea of TL is to re-use information from a previous (source) task to enhance the learning of a new (target) task. Formally, we can define general TL as follows (adapted from [51, 52]):

Definition 1 (*Transfer learning*) Given a source task T_S taking place in source domain D_S, as well as a target task T_T taking place in target domain D_T, *transfer learning* aims to enhance learning of the objective function $f_T(\cdot)$ in D_T by using knowledge of D_S and T_S, where $D_S \neq D_T$ or $T_S \neq T_T$.

A domain D consists of two parts: a feature space X and a marginal probability distribution $P(X)$. A task T also consists of two parts: a set of labels Y and an objective function $f(\cdot)$. The most frequent type of TL that is used in AI is that of parameter-based TL, meaning we transfer knowledge at a parameter/model level. For instance, we transfer the trained weights of a neural network model that was trained on some source task T_S to a neural network that learns a new task T_T. In principle, we reuse the exact same neural network, but in most cases it is necessary to avoid transferring the weights of the output layer of the network. As the last layer is already fit to produce outputs for a different task, it would be difficult to retrain it for a new task. The last layer is therefore frequently randomly initialized, such that it can more easily adapt to a new task. Parameter-based TL is generally done by either freezing the pre-trained network (not updating its parameters), or retraining the pre-trained parameters along with the new last layer, referred to as fine-tuning [53, 54, 55].

Transferring knowledge from a single domain and task, as described above, is called single-source TL. When we transfer knowledge from multiple domains and tasks, we speak of multi-source TL, which is the main concept that will be used in this work. We are performing inductive TL if the source and target domains are the same, but the source and target tasks are different. When the tasks are the same, but the source and target domains are different (yet somehow related), we instead speak of transductive TL. TL is not to be confused with multi-task learning, as the former transfers knowledge sequentially to related domains, whereas the latter transfers knowledge whilst simultaneously learning two or more related tasks. If the objective function is learned faster and/or better than without TL, we speak of a positive transfer, opposed to negative transfer when the contrary is the case. Alternatively, if the transfer learning results in neither a significant improvement, nor worsening of performance, we speak of a neutral transfer.

2.4.2 Transfer learning in Context of Reinforcement Learning

TL can also be used to speed up the learning process of RL agents, by enhancing the process of learning an optimal policy. We can formally define TL in context of RL by adapting Definition 1 as follows (adapted from [5]):

Definition 2 (Transfer learning for reinforcement learning) Given a source task T_S taking place in source MDP $\mathcal{M}_S$, as well as a target task T_T taking place in target MDP $\mathcal{M}_T$, *transfer learning for reinforcement learning* aims to enhance learning of the optimal policy π^* in $\mathcal{M}_T$ by using knowledge of $\mathcal{M}_S$ and T_S, where $\mathcal{M}_S \neq \mathcal{M}_T$ or $T_S \neq T_T$.

This means that the objective function $f(\cdot)$ can now be considered as the expected cumulative reward obtained by the optimal policy π^* (Equation 7), and we limit the domains to being MDPs. For simplicity we again restricted this definition to single-source TL, but the number of source domains and tasks can be greater than one. If there is no T_S nor $\mathcal{M}_S$ we are dealing with regular RL that only leverages feedback from $\mathcal{M}_T$.

When applying TL to RL, we need to consider what differences exist between the source and target MDPs. The main (partial) differences between domains can be in the state space S, the action space A, the reward function $R(s,a,s')$, and the transition dynamics $P(s,a,s')$. Additionally, two domains may differ in the initial states the agent starts in, or the number of steps an agent is allowed to move. Depending on the differences between the domains, we can consider what components to transfer from source to target domain. For instance, depending on the differences of the action spaces, state spaces, and reward functions, we can transfer policy and value functions [56, 57], as well as reward functions [58].

TL is particularly popular in DRL, as, for example, pre-trained feature extractions of CNNs can often be reused to process high-dimensional image inputs of similar domains, which speeds up the learning process [59]. Distillation techniques are a type of TL approach where a new network learns to predict the mappings of inputs to outputs of a pre-trained network [60]. DRL researches have used this method to transfer policies from large DQNs, or multiple DQNs trained on different tasks, to a single and smaller DQN [61, 62]. Combinations of TL and multi-task learning can result in improvements of sample efficiency and robustness of current DRL algorithms [63]. The usage of neural networks also allows the (parameter-based) transfer of feature representations, such as representations learned for value and policy functions [13, 64].

In order to evaluate the performance of TL approaches, several metrics have been proposed, such as the jumpstart (initial) performance, overall performance, and asymptotic (ultimate) performance [5, 65]. By comparing these metrics of an agent trained with TL and without TL, we can evaluate the effects of a TL approach in different ways: mastery (ultimate performance), generalization (initial performance), and a combination of both (overall performance).

2.4.3 Transfer Learning using World Models

When applying TL to MBRL, we can also consider to transfer the dynamics model to the target domain if it contains similar dynamics to the source domain [66]. When using World Models, state observations are mapped to a compressed latent representation by means of reconstruction. In this latent space reward modules, dynamics models, as well as policy and value functions are trained. The authors in [5] suggest that this can be beneficial for transferring knowledge across tasks that, for instance, share the dynamics, but have different reward functions. The authors of SEER show that for similar domains, the AE trained to map state observations to latent space can be frozen early-on in the training process, as well as transferred to new tasks and domains to save computation [67]. PlaNet [12] was used in a multi-task experiment, where the same PlaNet agent was trained on tasks of six different domains simultaneously. The agent infers which task it is doing based on the visual observations, and the authors show that a single agent becomes able to master these tasks using a single World Model. Due to the same physical laws that are present in physics engine used by the different domains, the transition model is able to transfer this knowledge among the different tasks. Similarly, the authors of [68] also perform a multi-task experiment, where the dynamics model of a MBRL agent is transferred to several novel tasks sequentially, and show that this results in significant gains of sample efficiency. In Plan2Explore [25] the authors show that a single global World Model can be trained task-agnostically, after which a Dreamer [9] agent can use the model to zero-shot or few-shot learn a new task. They show that this can result in state-of-the-art performance for zero-shot and few-shot learning, as well as competitive performance with a vanilla Dreamer agent.

2.5 Conclusion

In this chapter we covered the theoretical background relevant for this work. We first introduced the RL framework by providing formal definitions and covering common approaches. Next, we detailed on MBRL, and how this type of methodology can be beneficial to use in terms of sample efficiency and planning. Afterwards, we introduced the concept of DL and how this has led to significant advances in RL, such as the development of World Models. Finally, we introduced the general concept of TL, after which we show how this can be used in context of RL, and how it has been used for MBRL approaches.

We can conclude that RL methods leveraging World Models are currently the state-of-the-art for several important benchmarks in terms of sample complexity and performance. Additionally, we can conclude that TL is a very powerful concept that can be used to boost the sample efficiency and performance of RL methods. Based on prior works we can also conclude that World Models are suitable for multi-task learning, and provide a promising foundation for TL. In this study we combine the concepts of multi-task learning, TL, and World Models, to develop novel multi-source TL approaches.

3 Methods

In this chapter we provide all relevant information for the proposed methodologies of this study. First, we detail on Dreamer [9], which is the algorithm that was used in all experiments of this work (Section 3.1). Next, we explain how simultaneous multi-task learning can be done using Dreamer, after which we present the first novel approach of this study: transferring fractions of neural network parameters (Section 3.2). Lastly, we present meta-model transfer learning, in which we introduce the concepts of a universal feature space, latent space classification, and a reward meta-model that can be used for multi-source TL (Section 3.3).

3.1 Dreamer

In this section we detail on the core components and processes of Dreamer. For complete details we refer the reader to [9]. Dreamer is the state-of-the-art deep MBRL algorithm for continuous control tasks. Dreamer learns using the following processes:

- A latent dynamics model is learned from a dataset of past experience to predict future rewards and transitions of actions and observations.
- An action and value model are learned from imagined latent trajectories.
- The action model is used in the environment to collect experiences that are added to the growing dataset.

The overall process of Dreamer is that it repeatedly collects experience data, learns latent dynamics using a World Model, and then uses this model to learn value and policy functions by imagining trajectories in latent space (Figure 7). The latent dynamics model is the RSSM architecture that we described in Section 2.3.3, from PlaNet [12]. The World Model consists of three components:

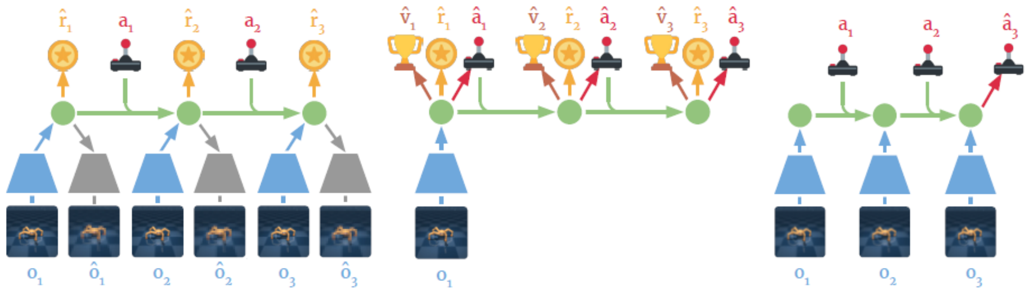
- Representation model $p_\theta(s_t|s_{t-1}, a_{t-1}, o_t)$, that maps state observations and actions to vector-valued latent states.
- Transition model $q_\theta(s_t|s_{t-1}, a_{t-1})$, that predicts the next latent state given a latent state and action, without having to observe or imagine the corresponding images.
- Reward model $q_\theta(r_t|s_t)$, that predicts the reward of a latent state.

Here p denotes distributions generating real environment samples, q denotes Gaussian distributions generating imagined samples, and θ denotes the neural network parameters of the models. The representation model is a VAE that is trained by reconstruction. The transition model learns by comparing its prediction of a latent state-action transition to the corresponding encoded next true state. The reward model learns to predict rewards of latent states by using the true reward of the environment state as a target. The imagined trajectories start at true model states s_t sampled from approximate state posteriors yielded by the representation model $s_t \sim p_\theta(s_t|s_{t-1}, a_{t-1}, o_t)$, which are based on past observations o_t of the dataset of experience. The imagined trajectory then follows predictions from the transition model, reward model, and a policy, by sampling $s_\tau \sim q_\theta(s_\tau|s_{\tau-1}, a_{\tau-1})$, taking the mode $r_\tau \sim q_\theta(r_\tau|s_\tau)$, and sampling $a_\tau \sim q_\phi(a_\tau|s_\tau)$ respectively, where τ denotes a time step in imagination.

In order to learn policies, Dreamer uses an actor-critic approach, which is a popular policy iteration approach for continuous spaces. An actor-critic consists of an action model and a value model, where the action model implements the policy, and the value model estimates the expected reward that the action model achieves from a given state:

- Actor model: $q_\phi(a_\tau|s_\tau)$
- Value model: $v_\psi(s_\tau) \approx E_{q(\cdot|s_\tau)}(\sum_{\tau=t}^{t+H} \gamma^{\tau-t} r_\tau)$

Here $H(= 15)$ denotes the finite horizon of an imagined trajectory, ϕ and ψ the neural network parameters for the action and value model respectively, and $\gamma(= 0.99)$ is the discount factor. In order to learn these models,



(a) Learn dynamics from experience (b) Learn behavior in imagination (c) Act in the environment

Figure 7: (a) From the dataset of past experience, the Dreamer learns to encode observations and actions into compact latent states via reconstruction, and predicts environment rewards. (b) In the compact latent space, Dreamer predicts state values and actions that maximize future value predictions by propagating gradients back through imagined trajectories. (c) The agent encodes the history of the episode to compute the current model state and predict the next action to execute in the environment. Source: [9]

estimates of state values for the imagined trajectories $\{s_\tau, a_\tau, r_\tau\}_{\tau=t}^{t+H}$ are used. Dreamer acquires these by using an exponentially-weighted average of the estimates V_λ:

$$\begin{aligned}
\mathrm{V}_{\mathrm{N}}^{k}(s_\tau) &\doteq \mathrm{E}_{q_\theta,q_\phi}\left(\sum_{n=\tau}^{h-1}\gamma^{n-\tau}r_n+\gamma^{h-\tau}v_\psi(s_h)\right) \quad \text{with} \quad h=\min(\tau+k,t+H)\\
\mathrm{V}_\lambda(s_\tau) &\doteq (1-\lambda)\sum_{n=1}^{H-1}\lambda^{n-1}\,\mathrm{V}_{\mathrm{N}}^{n}(s_\tau)+\lambda^{H-1}\,\mathrm{V}_{\mathrm{N}}^{H}(s_\tau)
\end{aligned} \tag{16}$$

That is, V_N^k estimates rewards beyond k imagination steps with the current value model. This is used in V_λ, where $\lambda(=0.95)$ is a parameter that controls the exponentially-weighted average of the estimates for different k. This value estimate is then used in the objective functions of the action and value models:

$$\begin{aligned}
&\max_\phi \mathrm{E}_{q_\theta,q_\phi}\left(\sum_{\tau=t}^{t+H}\mathrm{V}_\lambda(s_\tau)\right)\\
&\min_\psi \mathrm{E}_{q_\theta,q_\phi}\left(\sum_{\tau=t}^{t+H}\frac{1}{2}\|v_\psi(s_\tau)-\mathrm{V}_\lambda(s_\tau)\|^2\right)
\end{aligned} \tag{17}$$

That is, the objective function for the action model $q_\phi(a_\tau|s_\tau)$ is to predict actions that result in state trajectories with high value estimates. The objective function for the value model $v_\psi(s_\tau)$ is to regress these value estimates. As can be inferred from these definitions, the value model and reward model are crucial components in the training process, as they determine the value estimation. The learning procedure of the action model is therefore completely dependent on the predictions of these models. Moreover, as the reward model and value model are dependent on imagined states of a fixed World Model that result from actions, they are also dependent on the action model. Each of the models mentioned are implemented as neural networks, and are optimized using stochastic backpropagation of multi-step returns. Pseudocode of the complete process a Dreamer agent undertakes using the aforementioned definitions can be found in Algorithm 3.

```
Initialize dataset $D$ with $S$ random seeds.
Randomly initialize neural network parameters of World Model $\theta$, action model $\phi$, and
  value model $\psi$.
**repeat**
    **for** *update step c=1..C* **do**
        /* Dynamics learning */
        Draw $B$ data sequences of sequence length $L$ $\{a_t, o_t, r_t\}_{t=k}^{k+L} \sim D$.
        Compute model states $s_t \sim p_\theta(s_t | s_{t-1}, a_{t-1}, o_t)$.
        Update $\theta$ using representation learning.

        /* Behaviour learning */
        Imagine trajectories $\{s_\tau, a_\tau\}_{\tau=t}^{t+H}$ from each $s_t$.
        Predict rewards $\mathrm{E}(q_\theta(r_\tau . s_\tau))$ and values $v_\psi(s_\tau)$.
        Compute value estimates $V_\lambda$.
        Update $\phi \leftarrow \phi + \alpha \nabla_\phi \sum_{\tau=t}^{t+H} \mathrm{V}_\lambda(s_\tau)$.
        Update $\psi \leftarrow \psi - \alpha \nabla_\psi \sum_{\tau=t}^{t+H} \frac{1}{2} \left\| v_\psi(s_\tau) - \mathrm{V}_\lambda(s_\tau) \right\|^2$.
    **end**

    /* Environment interaction */
    $o_1 \leftarrow env.reset()$
    **for** *time step t=1..T* **do**
        Compute $s_t \sim p_\theta(s_t \mid s_{t-1}, a_{t-1}, o_t)$ from history.
        Compute $a_t \sim q_\phi(a_t \mid s_t)$ with the action model.
        Add exploration noise to action.
        $r_t, o_{t+1} \leftarrow env.step(a_t)$.
    **end**
    Add collected experience to dataset $\mathcal{D} \leftarrow \mathcal{D} \cup \left\{ (o_t, a_t, r_t)_{t=1}^{T} \right\}$.
**until** *converged*
```

Algorithm 3: Dreamer (adapted from [9]).

Important to note for following sections, is that the reward, value, and action models are each implemented as three fully-connected layers, the encoder and decoder are four-layer CNNs, and the dynamics model is the RSSM as described in Section 2.3.3. Dreamer initializes with 5 randomly collected episodes, after which it iterates between 100 training steps and collection of 1 episode using the current action model with Gaussian exploration noise. The exploration strategy is ε-greedy, where ε is linearly decaying from 0.4 to 0.1 over the first 200.000 gradient steps.

3.2 Fractional Transfer Learning

In this section we introduce the first proposed methods, being transfer of simultaneous multi-task agents, and fractional transfer learning (FTL). The idea here is that we train a single agent on multiple tasks simultaneously (Section 3.2.1), after which we transfer its parameters to a novel task. Moreover, unlike common TL approaches, we only transfer fractions of the parameters for certain components (Section 3.2.2). Finally, we detail for each component of Dreamer whether it should receive no transfer, fractional transfer, or full transfer (Section 3.2.3).

3.2.1 Simultaneous Multi-Task Learning

As was shown in PlaNet [12], a single agent using the RSSM World Model is able to learn multiple tasks from different domains simultaneously if the dynamics across the domains are similar, and visual observations are used to be able to infer which domain is being faced. In this work we investigate whether these simultaneous multi-task agents can serve as a new type of multi-source TL for RL. In order to train agents on multiple tasks from different domains simultaneously, some adjustments need to be made to Dreamer, as was done in the PlaNet paper. First, when dealing with multiple tasks simultaneously, the action dimension $|A|$ of each domain may be different. Components like the action model in Dreamer require $|A|$ to be fixed in order to provide compatible outputs for each domain. To overcome this issue, we first compute which task has the largest action space dimensions, which is then used as the target dimension for padding the action spaces of all other tasks with unused elements:

$$|A|_D = \max(|A|_{\mathbf{D}}) \tag{18}$$

Where $D \in \mathbf{D}$ is domain D from the set of domains $\mathbf{D}$ that the agent will be trained in. This means the action space for each domain will be equivalent, but for tasks with originally smaller action spaces there are elements that have no effect on the environment and can therefore be ignored by the agent. Next, in order to acquire a balanced amount of gathered experience in the replay dataset, we apply the following modifications. Instead of randomly collecting 5 episodes for initialization, we randomly collect 1 episode of each task. Additionally, instead of collecting 1 episode of a single task after 100 training steps, we collect 1 episode for each of the training tasks.

3.2.2 Fractional Transfer

As discussed in Section 2.4, when transferring representations of neural networks the common approach is to re-use all weights of each layer. The final layer often needs to be randomly initialized for learning new tasks, as it otherwise needs to unlearn the previous task. The other layers are either frozen, or retrained together with the final layer. That is, layers are either fully transferred, or randomly initialized when TL is applied to neural networks. However, the drawback of re-initializing a layer randomly is that all knowledge contained in that layer is lost. Therefore, we present a novel TL approach for neural networks: FTL. The idea is to transfer a fraction of the weights of a source layer to a target layer. This can be done adding a fraction of the source layer's weights to the randomly initialized weights of a target layer. Rather than discarding all knowledge by random initialization, this approach allows the transfer of portions of potentially useful knowledge. Additionally, we can avoid overfitting scenarios with this approach as we don't necessarily fully transfer the parameters.

Formally, we can simply define FTL as:

$$W_T = W_T + \omega W_S \tag{19}$$

Where W_T is the randomly initialized weights of a target layer, W_S the trained weights of a source layer, and ω the fraction parameter that quantifies what proportion of knowledge is transferred, where $\omega \in [0, 1]$. That is, for $\omega = 0$ we have no transfer (i.e. random initialization), and for $\omega = 1$ we have a full transfer of the weights.

3.2.3 Application to Dreamer

The goal of this part of this thesis is to transfer knowledge from agents that were simultaneously trained on multiple tasks to novel tasks. We are interested in observing whether this provides for a positive transfer, and additionally whether FTL can improve these transfers. However, as Dreamer consists of several different components, we need to identify what type of TL is most suitable for transferring each component's learned representation. In Figure 8 a schematic illustration of all components of Dreamer and what type of transfer is applied to them can be observed. All of the following described parameters that are transferred are kept trainable afterwards.

First, we identify the components of Dreamer that require random initialization, which are the components that involve processing actions. This is due to the fact that when transferring to a new domain, the action dimensions are likely incompatible with the action dimensions from the multi-source agents. Moreover, even if the action dimensions are equivalent, when the source and target domains are different, the learnt behaviours won't apply to the target domain and have to be unlearned when fully transferred. Therefore, we chose to randomly initialize (i.e. $\omega = 0$) the last layer of the action model, and the weights that connect the input actions to the forward dynamics model.

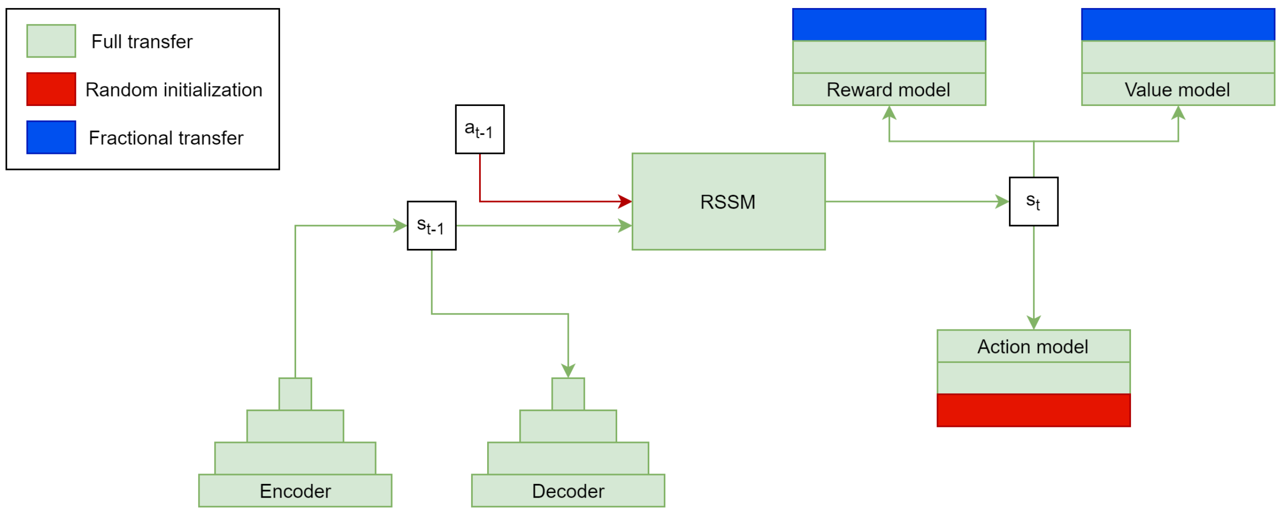


Figure 8: A schematic diagram of Dreamer's components and which type of TL is applied to each of the components. Green represents full transfer, red represents random initialization, blue represents FTL, arrows represent relevant input/output weights, and rectangles represent layers or models. For instance, the input weights of the RSSM model for an action a_{t-1} are randomly initialized.

Next, we identify the components that are suitable for FTL, being the value model and reward model. For these neural networks we are not dealing with dimension mismatches, meaning random initialization is not a necessity. Additionally, full transfers are not applicable, as reward and value functions are generally different between the source and target domains, which would mean the weights of the source task would have to be unlearned. However, though these functions may not be equivalent across domains, the learned weights may still consist of useful information that can be transferred with FTL. For these reasons we perform FTL for the last layer of the reward model and the last layer of the value model.

For the action model, value model, and reward model we only perform FTL and random initialization for the last layer. For the previous layers of these models we apply full transfer, as is generally done in TL for supervised learning, where only the last layer is randomly initialized. As the previous layers solely learn feature extractions of the latent state inputs, they can be re-used across domains. Preliminary empirical results showed that the best performance was obtained when only treating the last layer differently.

Finally, we identify which components of Dreamer are suitable for full transfers, which are the forward dynamics model and the VAE. The forward dynamics model can be fully transferred across domains if, for instance, the physical laws across the domains are equivalent, meaning this type of knowledge can be re-used [65]. The domains that will be used in this study share the same physics engine (Section 4.1), and therefore we will completely transfer the dynamics model parameters. We can also fully transfer the parameters of the representation model, as the generality of convolutional features allow reconstruction of observations for visually similar domains, and can therefore quickly adapt to novel domains [67].

3.3 Meta-Model Transfer Learning

In this section we propose several methods, being a universal AE (UAE), meta-model TL, and latent task classification (LTC). Meta-model TL is a TL approach that combines information from multiple separate sources to serve as input for a meta-model (Section 3.3.2). LTC introduces a classifier to World Model architectures that can be used to classify different tasks, as well as detecting novel tasks (3.3.3). However, in order for both of these methods to work, we require a single feature space for different agents and domains, meaning a UAE is necessary (Section 3.3.1).

3.3.1 Universal Autoencoder

The concept of a UAE is inspired by of SEER [67], where the authors show that trained encoders can be frozen early on in training to save computation, as well as be transferred to visually similar domains. We use this idea

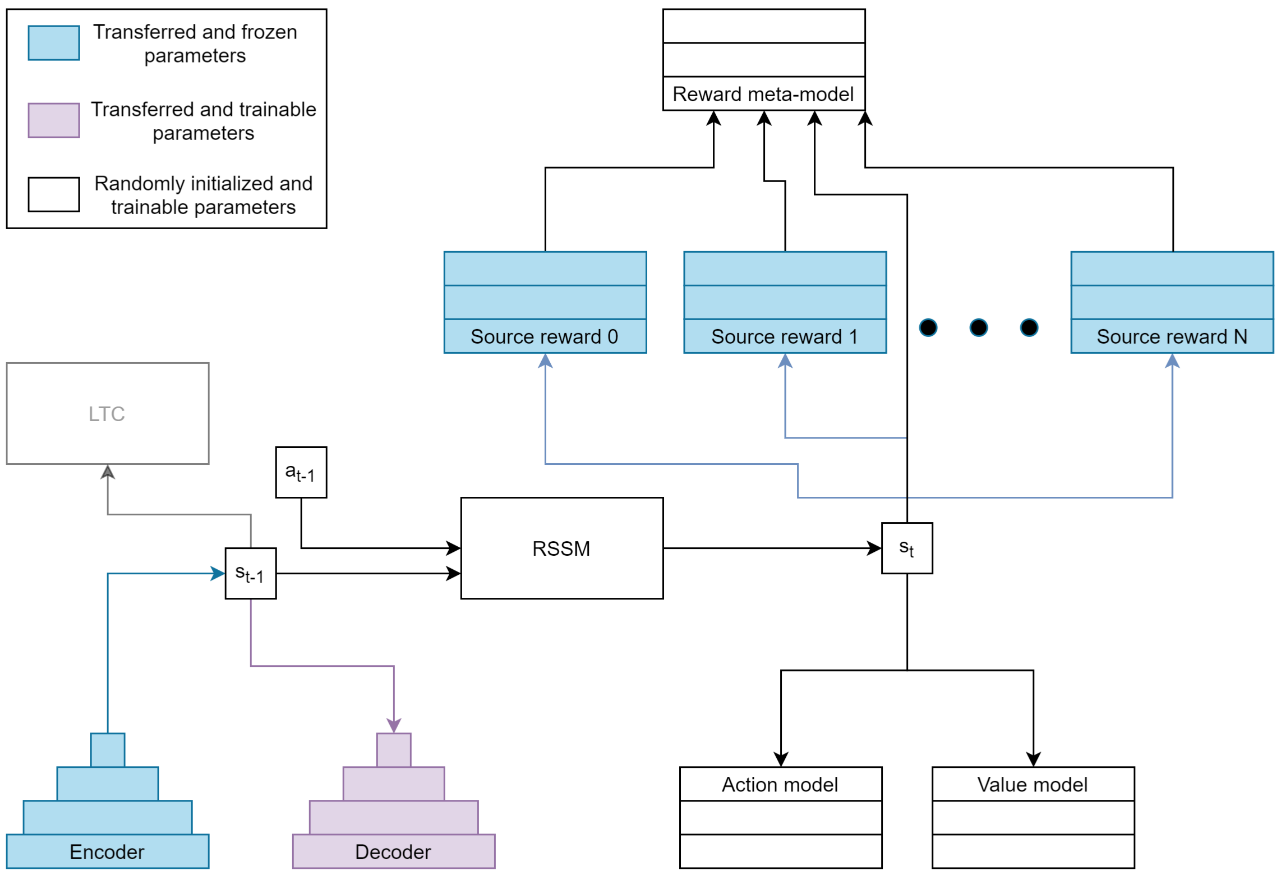


Figure 9: A schematic overview of the reward meta-model and UAE concepts, where blue indicates frozen parameters, and white indicates randomly initialized and trainable parameters. The frozen encoder yields latent states for a universal feature space, which is fed to each of the source reward models that are also frozen. The outputs of each frozen reward model are concatenated together with the latent state, and then fed to the reward meta-model.

and extend it to training a single VAE used by a Dreamer agent on two domains simultaneously as described in Section 3.2.1. The motivation here, is that the learned reconstruction of multiple different domains allows more diverse feature extraction to be learned, and therefore enhances generalizability of reconstructing images of novel domains. We can transfer this VAE and freeze its encoder in order to have a universal feature space across different agents and domains.

3.3.2 Meta-Model

Rather than performing multi-source TL by transferring parameters of an agent trained on multiple tasks simultaneously, we propose another multi-source TL approach which uses knowledge of several individually trained agents on different tasks. The idea is to combine information from multiple source agents by concatenating the predictions of several frozen pre-trained models to serve as input for a new model: the meta-model. As could be inferred from Equation 16, the reward and value model are crucial components for the policy training process of Dreamer, as they determine the value estimates that are used in training the actor-critic. Moreover, the reward model is the only connection to the real environment for learning representative value estimates, and therefore the value model is also fully dependent on the quality of the reward model. For this reason, we limit the application of the meta-model approach to solely the reward model for this study, though the meta-model approach could also be applied to other models.

First, we train N individual agents, each on a single task of a unique domain, where each of the agents make use of the same UAE's frozen encoder as described in Section 3.3.1. For a given agent i, where $i \in N$, its reward model $q_{\theta i}(r_t|s_t)$ is stored. The knowledge of the collection of stored reward models $\mathbf{G}_N$ is transferred to a novel agent by, for a given timestep t, concatenating the predictions of each $q_{\theta i}(r_t|s_t) \in \mathbf{G}_N$, after which this assembly of feature representations is concatenated with the latent state s_t:

$$\text{Meta-model: } q_\theta(r_t | s_t, \mathbf{G}_N) \tag{20}$$

That is, the reward meta-model of the target agent takes as input both the latent state s_t, and N predictions for s_t from frozen stored reward models. For an schematic illustration of this approach applied to Dreamer see Figure 9.

Rather than replacing the weights of a new agent, this approach simply provides additional information to the new reward model that could potentially be useful for the new task. We believe this approach will allow predictions from irrelevant models to be ignored, and therefore prevent interference that could result in negative transfers. On the other hand, if the information is useful, the meta-model can learn to more heavily weigh the output of the reward model that provides relevant information for the novel task in computing its predictions. Moreover, as we retain knowledge in frozen reward models from different tasks, the agent could leverage a stored reward model when it encounters the corresponding task in a multi-task learning setting.

3.3.3 Latent Task Classification

In the previous section we finalized with the possibility of switching between reward models, dependent on which task is being performed. However, in order for the agent to switch to a corresponding reward model, it needs to know which task it is performing. In simultaneous multi-task learning the agent learns to infer this from the visual observations (Section 3.2.1). One could argue that the meta-model would eventually learn which reward model prediction to solely use for a certain task. However, we believe this would require a long time to accomplish, and therefore we instead propose LTC.

The idea is to first train a classifier on latent states of each of the tasks on which the stored reward models were trained. For a given latent state s_t, the classifier can therefore learn to predict which task the agent is facing, who can consequently switch to the corresponding reward model. However, the agent needs to know when it is facing a novel task in order to know when to use the reward meta-model. For the latter reason, the classifier we propose to use for LTC is K-Nearest Neighbors (KNN) [69].

KNN is a supervised learning algorithm that stores a set of training data points $\mathbf{X}$, where each datapoint is assigned to a certain class $y \in \mathbf{Y}$ (or task y in our case). Using $\mathbf{X}$, KNN predicts to which class a new data point $x \notin \mathbf{X}$ belongs to by computing the K nearest data points to the new sample, and determining to which class y the majority of these K data points belong to. However, in order to compute the nearest data points, KNN makes use of a distance metric, often being the Euclidean distance or the Manhattan distance. The former is generally used in low-dimensional spaces, whereas the latter is more suitable for high-dimensional spaces [70]. As we are dealing with high-dimensional latent spaces, we will be making use of the Manhattan distance. For an M-dimensional space, the Manhattan distance between a data point $\mathbf{p} = (p_1, p_2, ..., p_M)$ and a data point $\mathbf{q} = (q_1, q_2, ..., q_M)$ is defined as:

$$d_{\text{Manh}}(\mathbf{p}, \mathbf{q}) = |p_1 - q_1| + |p_2 - q_2| + ... + |p_M - q_M| \tag{21}$$

In our case we train a KNN on N tasks with M-dimensional latent states, where $M = 230$. As we are training on low-dimensional representations of image observations, the distance metric represents the similarity of compressed image observations of real environment domains. For this reason, we propose to determine whether a latent state belongs to a novel tasks by evaluating the distance to the nearest neighbor latent state the KNN was trained on. That is, we can introduce a threshold ξ that determines whether a given latent state s corresponds to a novel task or a previously seen task, if the Manhattan distance to the nearest neighbor exceeds the threshold. Like so, we can formulate LTC with a KNN trained on dataset $\mathbf{X}$ with task labels $\mathbf{Y}$ for a given latent state s as:

$$y(s) = \begin{cases} \text{Novel}, & \text{if } \min_{x \in \mathbf{X}} d_{\text{Manh}}(x, s) > \xi \\ \text{KNN(x)}, & \text{otherwise} \end{cases} \tag{22}$$

That is, if the distance between the datapoint nearest to the new sample exceeds the threshold, it is regarded as a novel task. If this is not the case the sample is classified by the KNN as one of the previous tasks.

4 Experimental Setup

In this chapter we will describe the continuous control tasks that were used in the experiments provided the PyBullet framework [71] (Section 4.1). We then detail the description, hyperparameter setup, and evaluation metrics for the FTL (Section 4.2), LTC (Section 4.3), and meta-model transfer learning (Section 4.4) experiments. In the following experiment setups we only mention the hyperparameters that are introduced with each method, all other hyperparameters are as they were originally designed for Dreamer (see Section 3.1).

4.1 Continuous Control Simulated Environments

In order to perform experiments of the proposed methodologies, we chose to use several continuous control tasks from the environments that are provided by the PyBullet physics engine. These environments are very similar to the continuous control environments provided by the OpenAI Gym [72], yet are open-source and harder to solve [73]. For the experiments we chose a set of 4 locomotion tasks (Section 4.1.1) and 2 pendula swing-up tasks (Section 4.1.2). For each of the environments and experiments we provide image observations from the simulation to the algorithms, and return task-compatible action vectors to perform actions.

4.1.1 Locomotion Tasks

The used locomotion tasks are the Ant, HalfCheetah, Hopper, and Walker2D environments (Figure 10). These environments have action spaces of 8, 6, 3, and 6 respectively, corresponding to the number of controllable joints the entity has. The objective in each of the environments is identical: a fixed target is set 1km away from the initial position of the creatures, which they need to reach as efficiently as possible. The reward signal for a given frame f is composed of a few different terms:

$$r_f = AliveBonus_f + Progress_f + ElectricityCost_f + JointLimitCost_f \tag{23}$$

Here the $AliveBonus$ denotes whether or not the agent has fallen to the ground, meaning whether or not the z-coordinate is above a certain threshold that differs per creature. $Progress$ is defined as the difference between the current potential and the previous potential, where potential is the $\frac{m}{s}$ relative to the target for a given frame. For a given executed action, the $ElectricityCost$ indicates the amount of electricity that was used to move the joints with a certain speed. Finally, $JointLimitCost$ is a term that discourages joints getting stuck, based on whether or not the joint is used.

4.1.2 Pendula Tasks

The pendula swing-up tasks that were used are the InvertedPendulum and the InvertedDoublePendulum environments (Figure 11). These environments both have an action space of 1. The objective of these environments

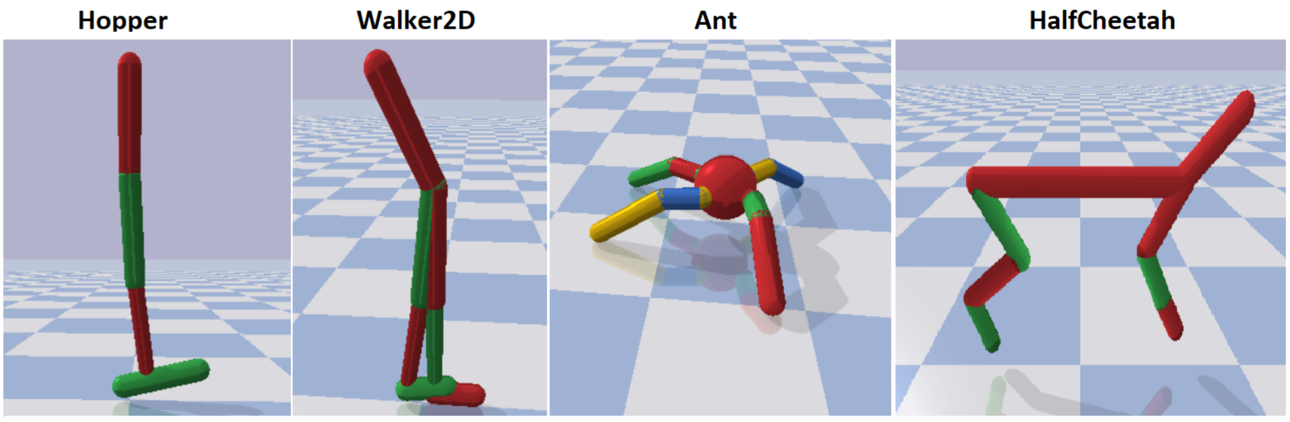


Figure 10: Images of the locomotion environments that were used in this work along with their name displayed at the top of each image.

is to swing the pendulum upwards and then balancing it. The reward signal for the InvertedPendulum for a given frame f is:

$$r_f = \cos\Theta \tag{24}$$

where Θ is the current position of the joint. For the InvertedDoublePendulum a swing-up task did not exist yet, therefore we have created this ourselves by simply adding the cosine of the position of the second joint Γ to Equation 24:

$$r_f = \cos\Theta + \cos\Gamma \tag{25}$$

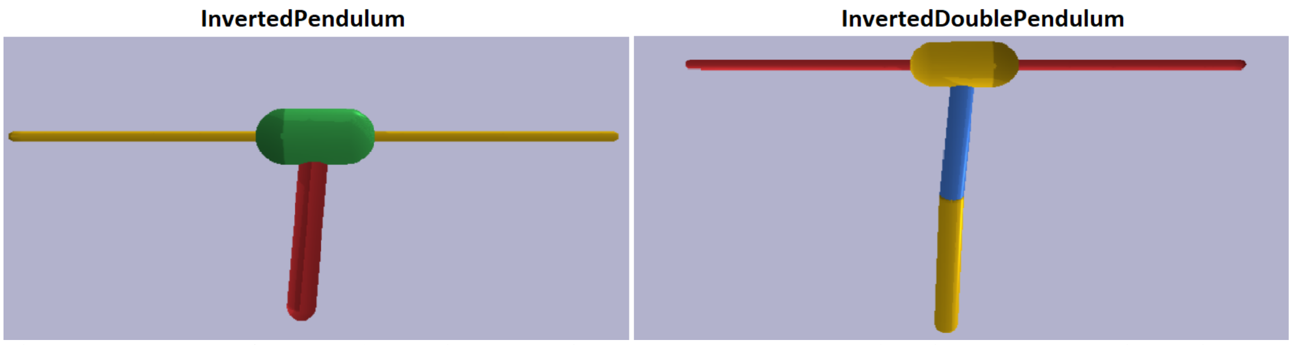


Figure 11: Images of the pendula swing-up environments that were used in this work along with their name displayed at the top of each image.

4.2 Fractional Transfer Learning

The experimental setup for FTL methodologies described in Section 3.2 will be detailed on in this section.

4.2.1 Experiment Description

The set of source tasks and the set of target tasks used in this experiment consist of the Hopper, Ant, Walker2D, HalfCheetah, InvertedPendulum, and InvertedDoublePendulum environments. First, a single agent is trained on multiple source tasks as described in Section 3.2.1. For each target task we train simultaneous multi-task agents of 2, 3, and 4 source tasks.
The HalfCheetah multi-source agents are:

- Hopper + Ant
- Hopper + Ant + Walker2D
- Hopper + Ant + Walker2D + InvertedPendulum

The Hopper multi-source agents are:

- HalfCheetah + Walker2D
- HalfCheetah + Walker2D + Ant
- HalfCheetah + Walker2D + Ant + InvertedPendulum

The Walker2D multi-source agents are:

- HalfCheetah + Hopper
- HalfCheetah + Hopper + Ant
- HalfCheetah + Hopper + Ant + InvertedPendulum

The InvertedPendulum multi-source agents are:

- HalfCheetah + InvertedDoublePendulum
- HalfCheetah + InvertedDoublePendulum + Hopper
- HalfCheetah + InvertedDoublePendulum + Hopper + Ant

The InvertedDoublePendulum multi-source agents are:

- Hopper + InvertedPendulum
- Hopper + InvertedPendulum + Walker2D
- Hopper + InvertedPendulum + Walker2D + Ant

The Ant multi-source agents are:

- HalfCheetah + Walker2D
- HalfCheetah + Walker2D + Hopper
- HalfCheetah + Walker2D + Hopper + InvertedPendulum

4.2.2 Hyperparameters

Each of the source agents is trained for 2e6 environment steps for a single run, where 1 episode of each task is collected when gathering data. For each source agent we transfer the weights of all components to the corresponding target agent, except for the weights of the last layer of the reward, value, actor networks, as well as the action-to-model weights. We add weight fractions of $\omega = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5]$ from the source to the reinitialized weights of the last layer of the reward and value networks, where $\omega = 0.0$ indicates complete re-initialization. Each source-target combination trains for 1e6 environment steps for three random seeds.

4.2.3 Evaluation

We evaluate the target agents by obtaining an episode return every 1e4 environment steps of the training process. We compare the overall average episode return, as well as the average episode return of the final 1e5 environment steps, to a baseline Dreamer agent that was trained from scratch for three seeds on each of the target tasks.

4.3 Latent Task Classification

The experimental setup for the LTC method described in Section 3.3.3 will be detailed on in this section. The reason for development of this approach is for it to be used in switching between models belonging to different tasks for multi-task learning. However, in this study we focus on the classification performance of this approach, rather than deploying it in a multi-task setting.

4.3.1 Experiment Description

For this experiment we use the UAE concept described in Section 3.3.1. We take the frozen encoder of the VAE of the source agent that was trained simultaneously on the Ant and Hopper tasks for 2e6 environment steps. The encoder provides the latent states on which we train the KNN algorithm for four source tasks, after which we present it with two novel tasks.

The combinations are as follows:

- With HalfCheetah and InvertedDoublePendulum as target tasks, the source tasks are Hopper, Walker2D, Ant, and InvertedPendulum.
- With Hopper and Walker2D as target tasks, the source tasks are HalfCheetah, InvertedPendulum, InvertedDoublePendulum, and Ant.

- With the Ant and InvertedPendulum as target tasks, the source tasks are Hopper, Walker2D, Inverted-DoublePendulum, and HalfCheetah.

Latent test samples are fed to the KNN, and based on the nearest neighbor's Manhattan distance, we classify whether or not the sample belongs to a novel task. If the sample is not classified as a novel task, it is classified by the KNN as one of the previous tasks.

4.3.2 Hyperparameters

The KNN classifier is trained on 100 random samples of each of the four training tasks. Afterwards, the model is evaluated on 100 samples randomly drawn from the dataset of experience created by a random agent for both the previous tasks and the novel tasks. We use Manhattan distance nearest neighbor thresholds for novel task classification of $\xi = [0.5, 0.7, 0.9]$. The classification of previous tasks is done with the 5 nearest neighbors.

4.3.3 Evaluation

We evaluate the classifier by taking the average accuracy for each set of random 100 samples. That is, for the novel tasks this accuracy is defined as whether or not it was classified as a novel task with the used threshold. For known tasks the accuracy is defined as both being classified as a previous task by the threshold, as well as being classified as the correct previous task.

4.4 Meta-Model Transfer Learning

The experimental setup for the meta-model transfer learning method described in Section 3.3 will be detailed on in this section.

4.4.1 Experiment Description

For these experiments we again use the UAE of the source agent that was trained on the Hopper and Ant task simultaneously for 2e6 environment steps. Using the frozen encoder of this VAE, we train single Dreamer agents on a single unique source tasks, after which we store their reward models. For each target task, we attach a set of 2, 3, and 4 frozen reward models to the reward meta-model as described in Section 3.3.2. We use the same source-target combinations as described in Section 4.2.1.

4.4.2 Hyperparameters

Each of the single-source agents is trained for 1e6 environment steps for a single run. Each source-target combination trains for 1e6 environment steps for three random seeds.

4.4.3 Evaluation

We evaluate the target agents by obtaining an episode return every 1e4 environment steps of the training process. We compare the overall average episode return, as well as the average episode return of the final 1e5 environment steps, to a baseline Dreamer agent that was trained from scratch on each of the target tasks for three random seeds.

5 Results

In this chapter we present the results of the experiments described in Section 4. In Section 5.1 the results for the FTL experiments can be found. Section 5.2 shows the results for the LTC experiments using KNN. Finally, the meta-model experiment results can be found in Section 5.3. The baseline performance of Dreamer on each of the tasks can be found in Appendix A. Example performances of simultaneous multi-task Dreamer agents for each of the trained tasks can be found in Appendix B.

5.1 Fractional Transfer Learning

In this section we present the results for FTL. We refer the reader to the results corresponding to each of the tasks in the list below as follows. The first, second, and third figures refer to 2, 3, and 4 source tasks respectively. In each figure the overall performance of the transfer learning agent (blue) can be seen compared to the overall performance of a baseline agent (green), where for both the standard deviation of three runs is represented by the shaded area. The first and second table refer to the overall average episode return, and the average episode return of the final 1e5 environment steps respectively.

- HalfCheetah: Figure 12, Figure 13, Figure 14, Table 1, and Table 2.
- Hopper: Figure 15, Figure 16, Figure 17, Table 3, Table 4.
- Walker2D: Figure 18, Figure 19, Figure 20, Table 5, and Table 6.
- InvertedPendulum: Figure 21, Figure 22, Figure 23, Table 7, and Table 8.
- InvertedDoublePendulum: Figure 24, Figure 25, Figure 26, Table 9, and Table 10.
- Ant: Figure 27, Figure 28, Figure 29, Table 11, and Table 12.

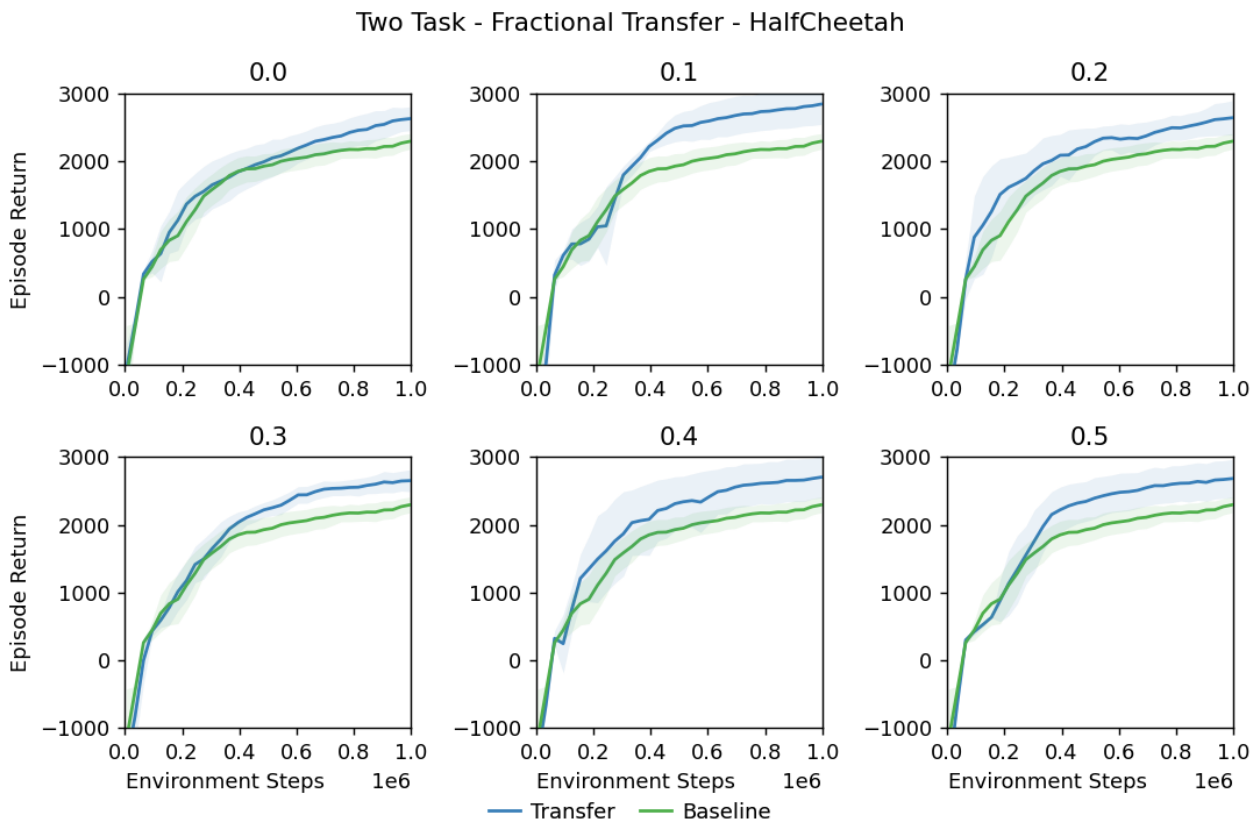


Figure 12: Episode return of having simultaneously trained on Hopper and Ant, after which fractional transfer took place for the HalfCheetah task with fractions of 0.0 up to 0.5.

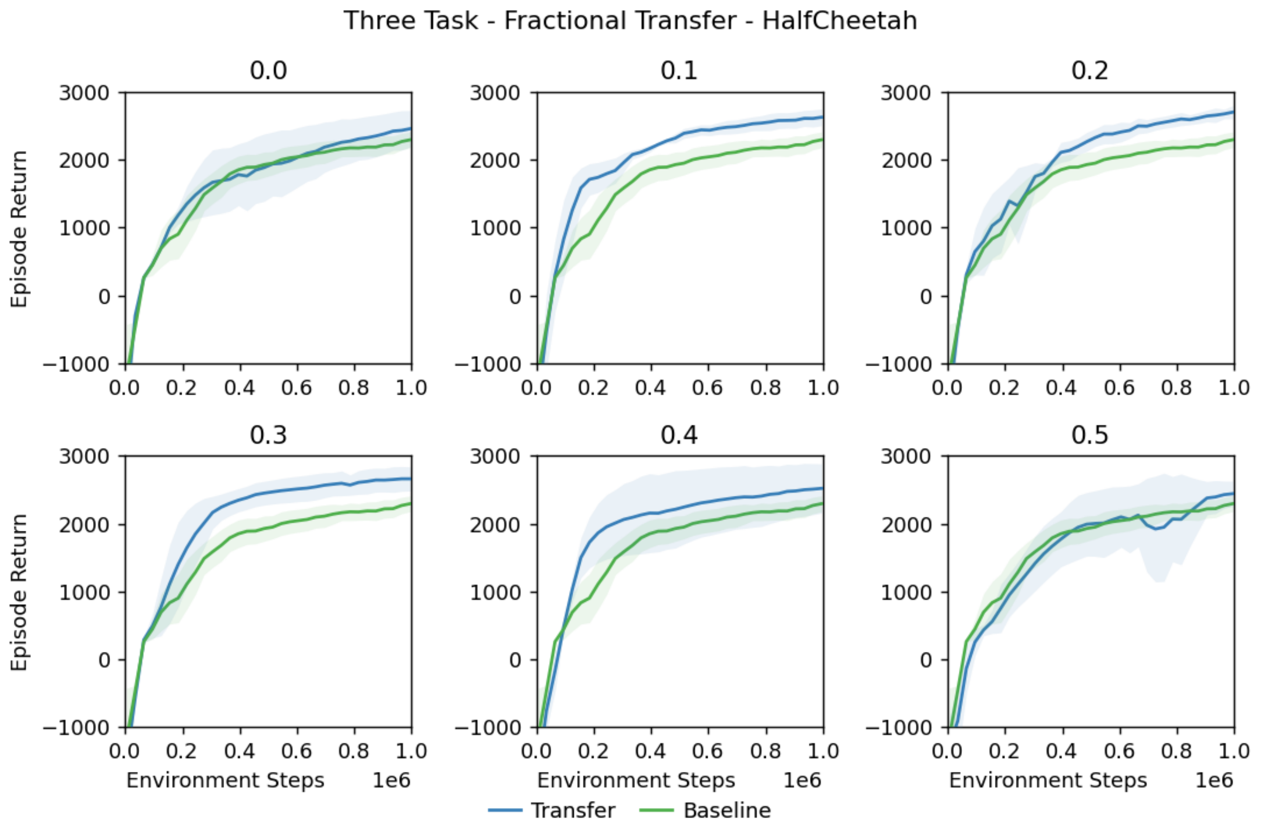


Figure 13: Episode return of having simultaneously trained on Hopper, Ant and Walker2D, after which fractional transfer was done for the HalfCheetah task with fractions of 0.0 up to 0.5.

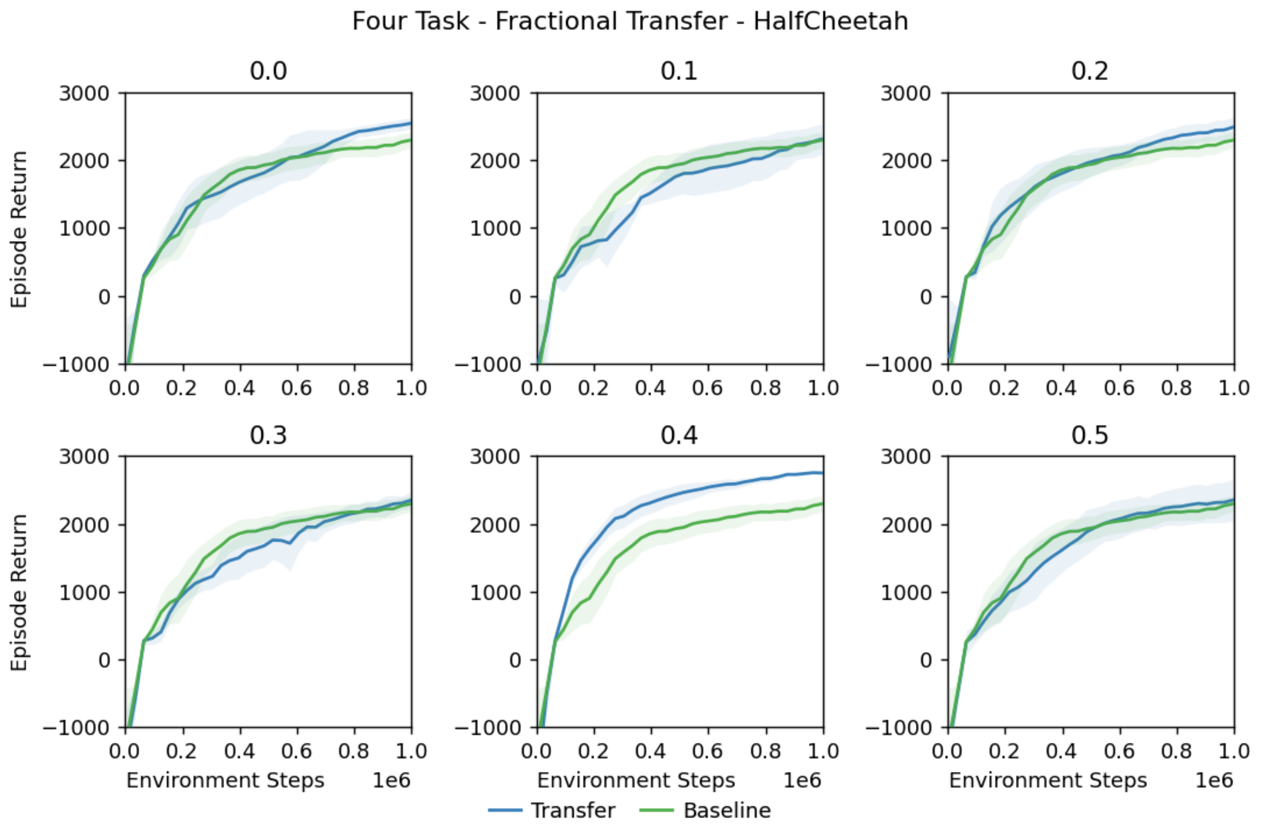


Figure 14: Episode return of having simultaneously trained on Hopper, Ant, Walker2D and InvertedPendulum, after which fractional transfer was done for the HalfCheetah task with fractions of 0.0 up to 0.5.

	2 Tasks	3 Tasks	4 Tasks
0.0	1841 ± 806	1752 ± 783	1742 ± 777
0.1	**2028** ± **993**	2074 ± 762	1500 ± 781
0.2	1982 ± 838	1967 ± 862	1773 ± 748
0.3	1899 ± 911	**2094** ± **859**	1544 ± 771
0.4	2008 ± 943	2015 ± 873	**2162** ± **789**
0.5	1961 ± 944	1647 ± 896	1635 ± 809
Baseline		1681 ± 726	

Table 1: Average return for fraction transfer of 2, 3, and 4 source tasks for the HalfCheetah task with fractions of 0.0 up to 0.5.

	2 Tasks	3 Tasks	4 Tasks
0.0	2614 ± 208	2435 ± 287	2520 ± 103
0.1	**2820** ± **297**	2615 ± 132	2276 ± 240
0.2	2628 ± 256	**2676** ± **119**	2455 ± 160
0.3	2642 ± 175	2658 ± 190	2313 ± 149
0.4	2682 ± 320	2513 ± 383	**2749** ± **90**
0.5	2668 ± 286	2424 ± 228	2327 ± 305
Baseline		2264 ± 160	

Table 2: Average return for the final 1e5 environment steps of the training process for fraction transfer of 2, 3, and 4 source tasks for the HalfCheetah task with fractions of 0.0 up to 0.5.

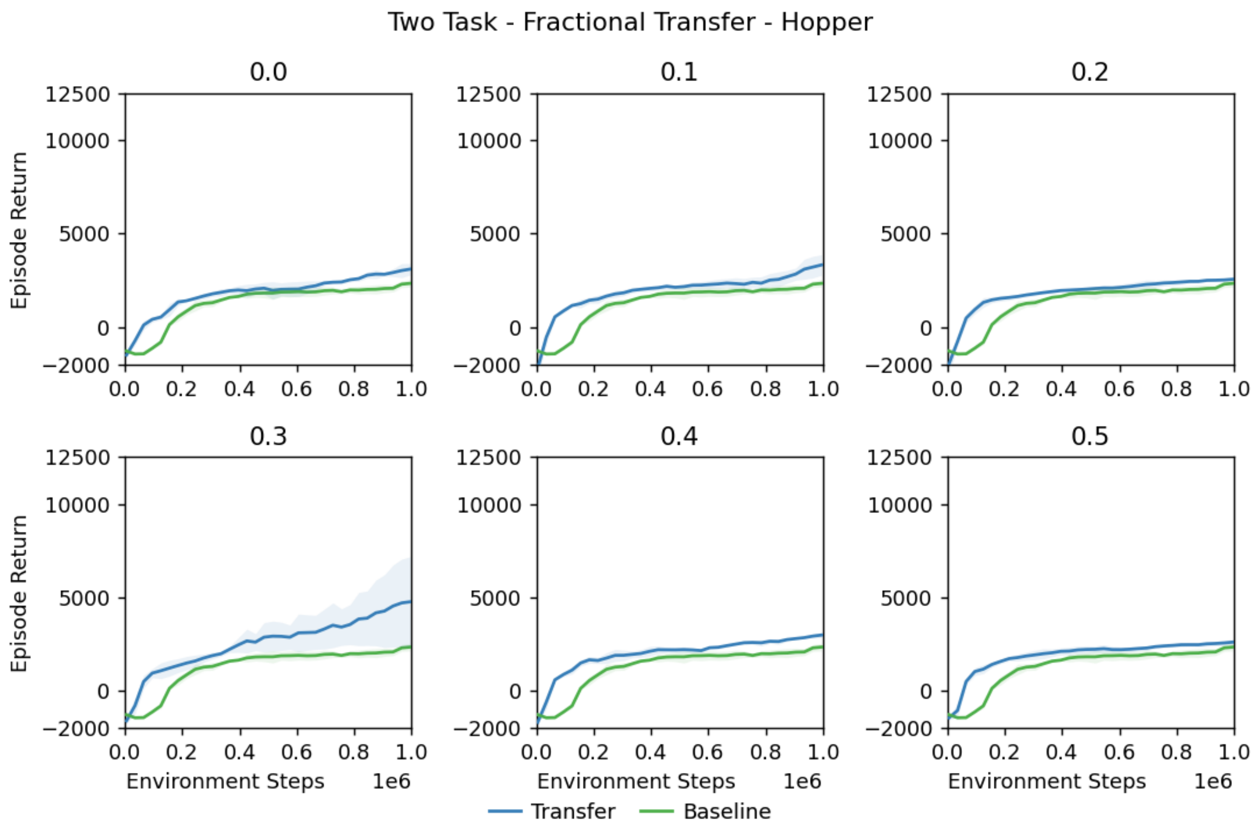


Figure 15: Episode return of having simultaneously trained on Cheetah and Walker2D, after which fractional transfer was done for the Hopper task with fractions of 0.0 up to 0.5.

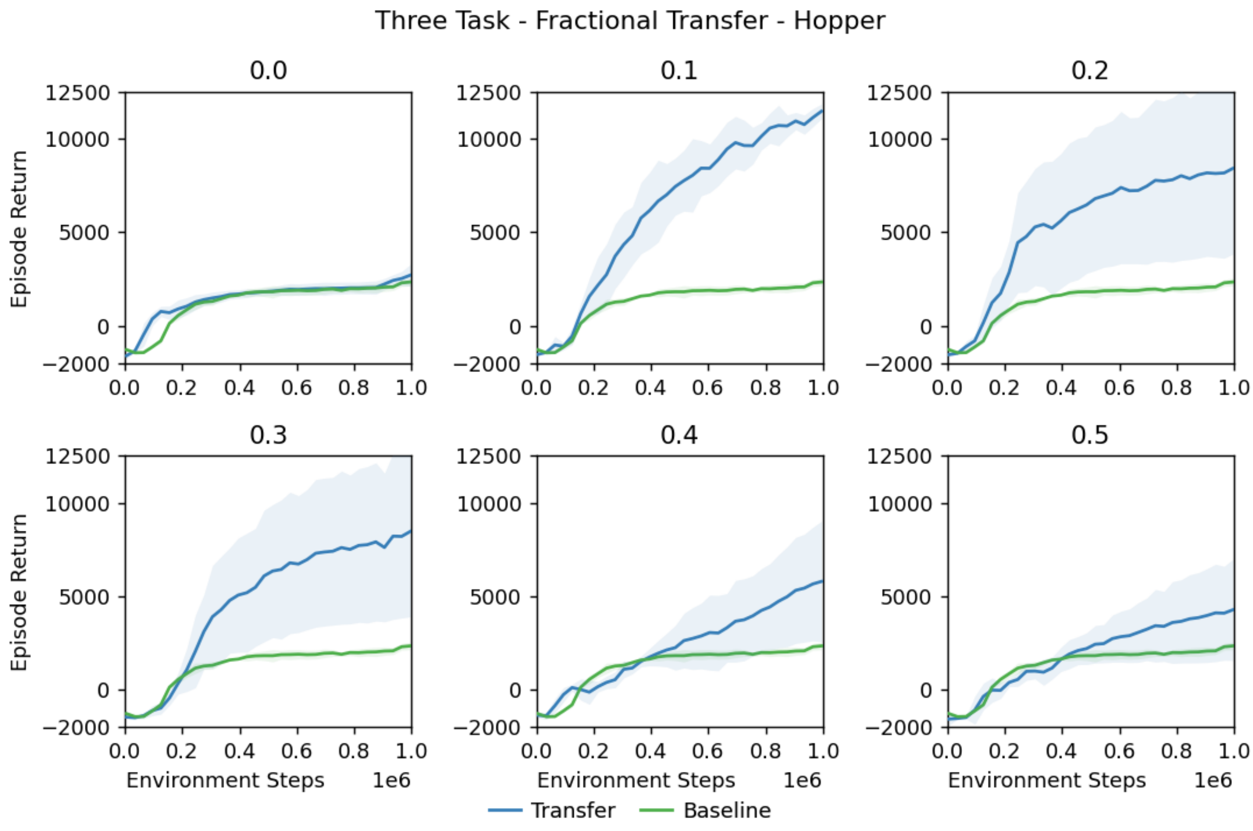


Figure 16: Episode return of having simultaneously trained on Cheetah, Walker2D and Ant, after which fractional transfer was done for the Hopper task with fractions of 0.0 up to 0.5.

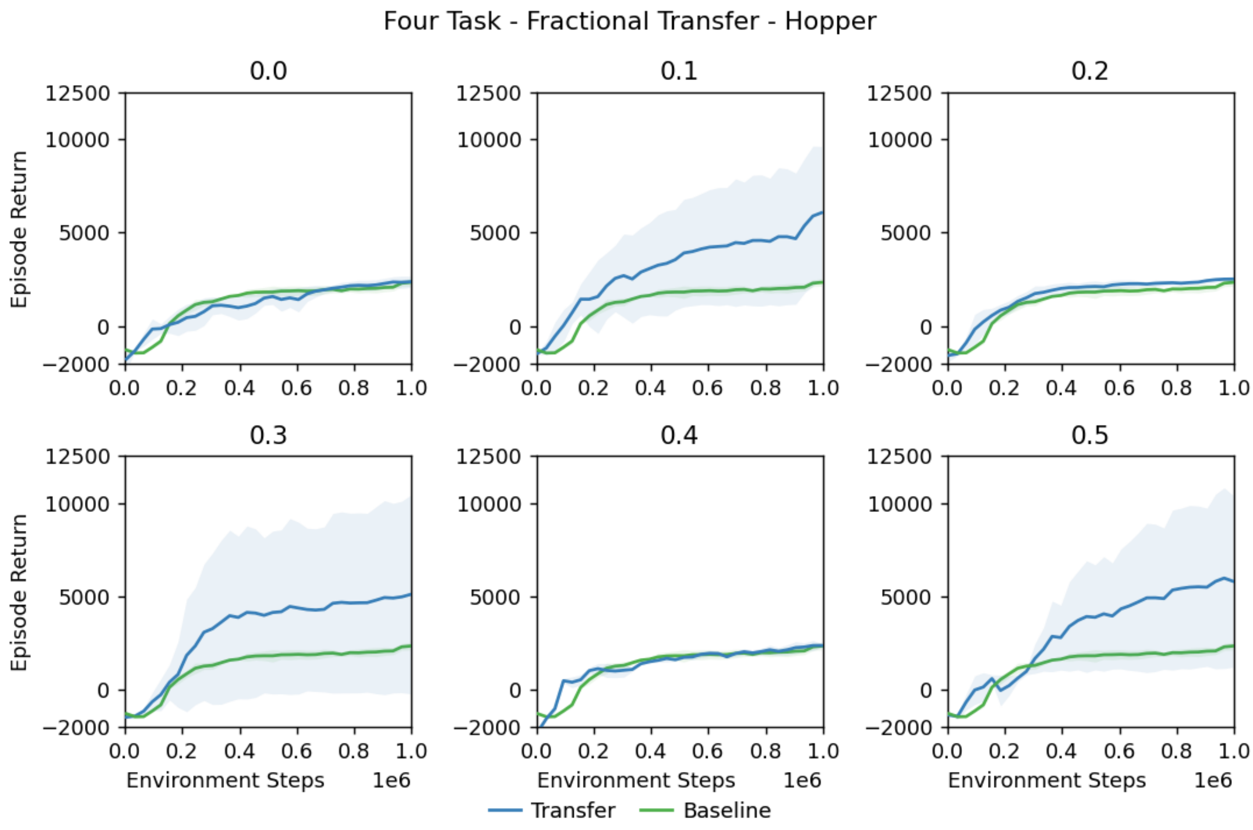


Figure 17: Episode return of having simultaneously trained on Cheetah, Walker2D, Ant and InvertedPendulum, after which fractional transfer was done for the Hopper task with fractions of 0.0 up to 0.5.

	2 Tasks	3 Tasks	4 Tasks
0.0	1917 ± 960	1585 ± 941	1263 ± 1113
0.1	2041 ± 887	**6542** ± **4469**	3300 ± 3342
0.2	1911 ± 712	5538 ± 4720	1702 ± 1078
0.3	**2670** ± **1789**	4925 ± 4695	**3341** ± **4609**
0.4	2076 ± 803	2437 ± 2731	1451 ± 991
0.5	1975 ± 772	2014 ± 2328	3246 ± 3828
Baseline		1340 ± 1112	

Table 3: Average return for fraction transfer of 2, 3, and 4 source tasks for the Hopper task with fractions of 0.0 up to 0.5.

	2 Tasks	3 Tasks	4 Tasks
0.0	3006 ± 960	2560 ± 591	2357 ± 351
0.1	3210 ± 887	**11124** ± **1645**	5749 ± 3747
0.2	2535 ± 712	8274 ± 4649	2507 ± 240
0.3	**4670** ± **1789**	8282 ± 4602	5019 ± 5284
0.4	2922 ± 803	5624 ± 3125	2355 ± 325
0.5	2571 ± 772	4149 ± 2652	**5886** ± **4774**
Baseline		2241 ± 502	

Table 4: Average return for the final 1e5 environment steps of the training process for fraction transfer of 2, 3, and 4 source tasks for the Hopper task with fractions of 0.0 up to 0.5.

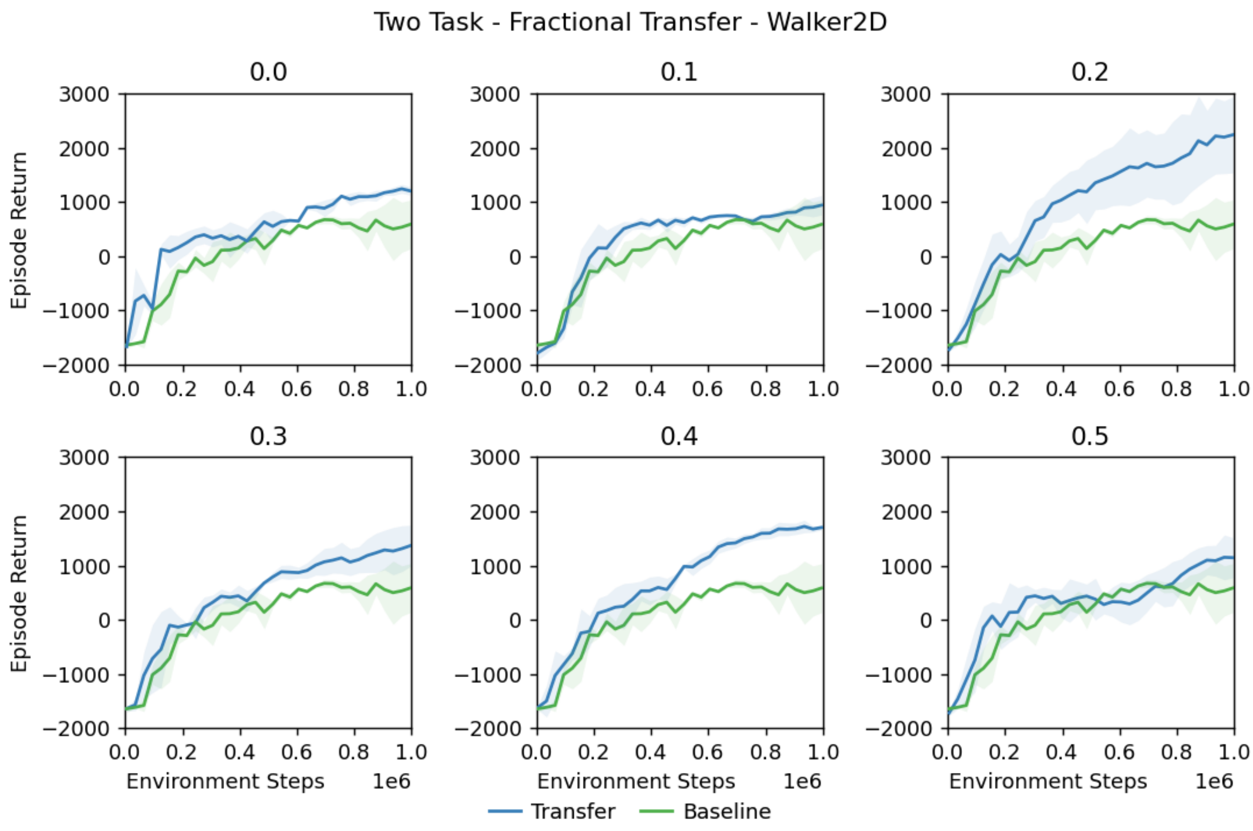


Figure 18: Episode return of having simultaneously trained on Cheetah and Hopper, after which fractional transfer was done for the Walker2D task with fractions of 0.0 up to 0.5.

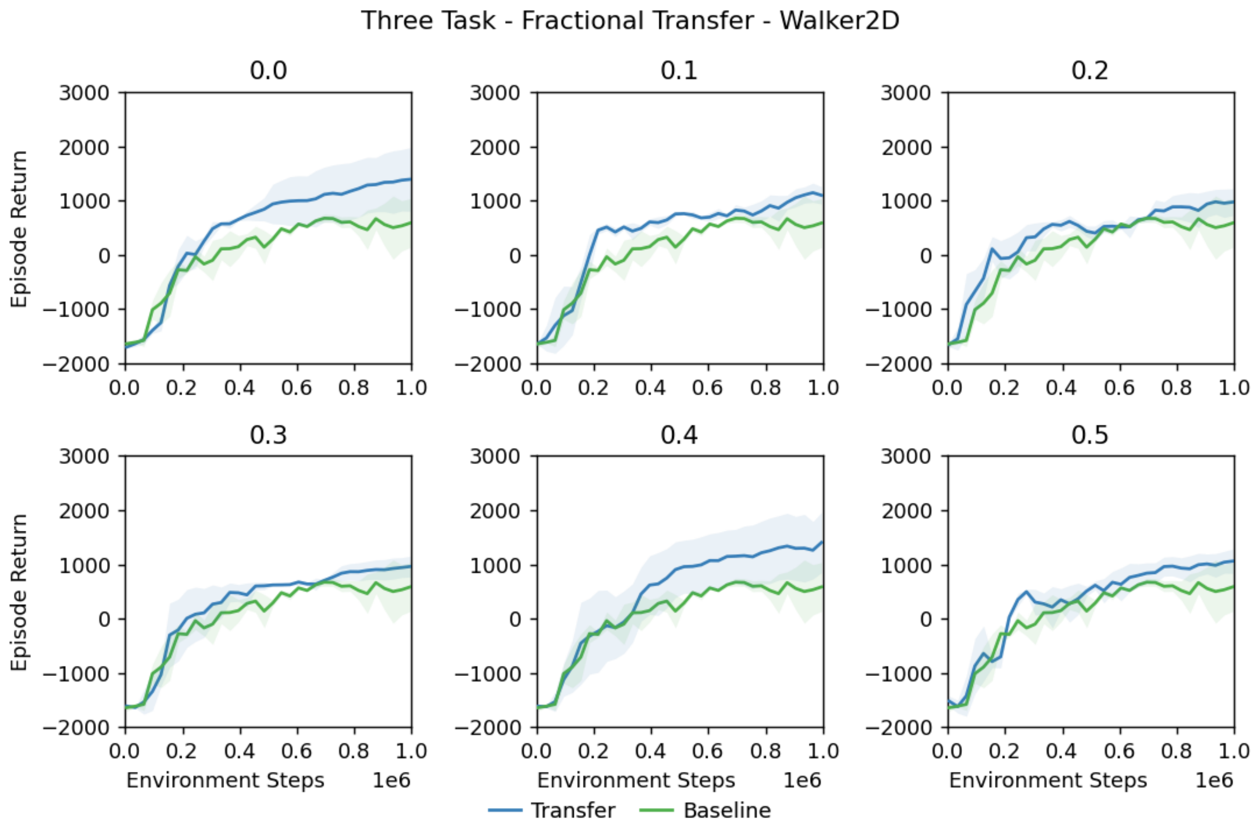


Figure 19: Episode return of having simultaneously trained on Cheetah, Hopper and Ant, after which fractional transfer was done for the Walker2D task with fractions of 0.0 up to 0.5.

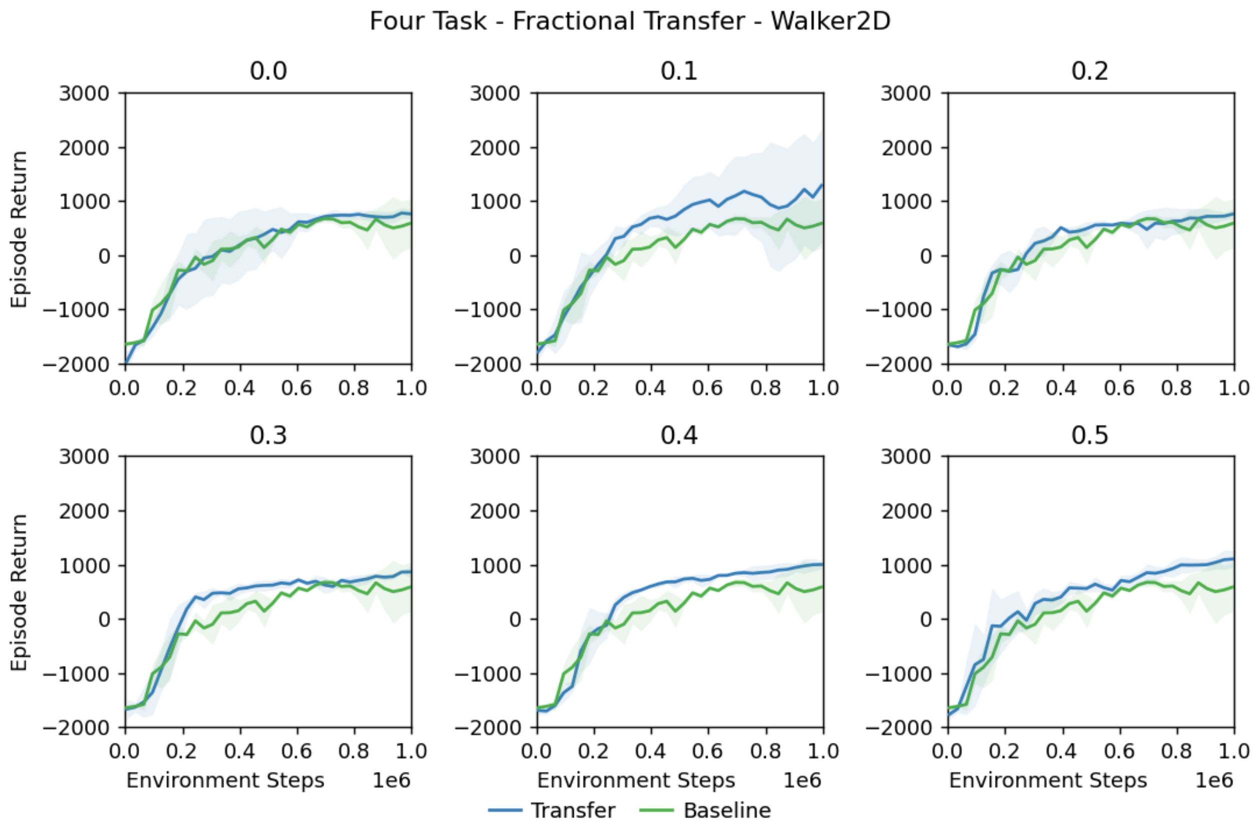


Figure 20: Episode return of having simultaneously trained on Cheetah, Hopper, Ant and InvertedPendulum, after which fractional transfer was done for the Walker2D task with fractions of 0.0 up to 0.5.

	2 Tasks	3 Tasks	4 Tasks
0.0	546 ± 776	**545 ± 1012**	157 ± 891
0.1	360 ± 803	441 ± 852	**478 ± 1113**
0.2	**1009 ± 1254**	393 ± 813	200 ± 807
0.3	535 ± 914	325 ± 812	323 ± 818
0.4	742 ± 992	516 ± 1074	354 ± 862
0.5	352 ± 853	355 ± 903	396 ± 829
Baseline		116 ± 885	

Table 5: Average return for fraction transfer of 2, 3, and 4 source tasks for the Walker2D task with fractions of 0.0 up to 0.5.

	2 Tasks	3 Tasks	4 Tasks
0.0	1212 ± 776	**1368 ± 613**	744 ± 256
0.1	911 ± 803	1104 ± 297	**1191 ± 1146**
0.2	**2214 ± 1254**	963 ± 314	733 ± 183
0.3	1321 ± 914	944 ± 199	841 ± 299
0.4	1705 ± 992	1325 ± 631	990 ± 199
0.5	1127 ± 853	1033 ± 301	1085 ± 315
Baseline		547 ± 710	

Table 6: Average return for the final 1e5 environment steps of the training process for fraction transfer of 2, 3, and 4 source tasks for the Walker2D task with fractions of 0.0 up to 0.5.

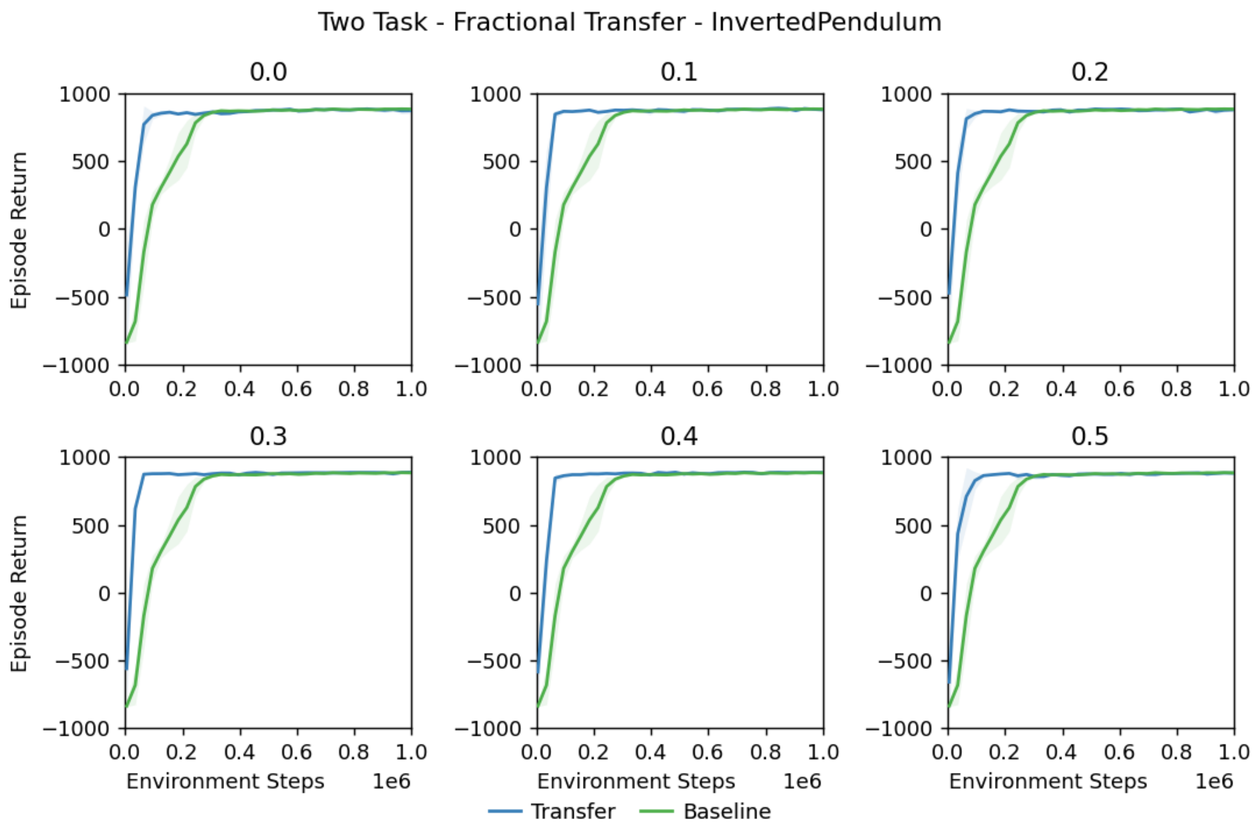


Figure 21: Episode return of having simultaneously trained on HalfCheetah and InvertedDoublePendulum, after which fractional transfer was done for the InvertedPendulum task with fractions of 0.0 up to 0.5.

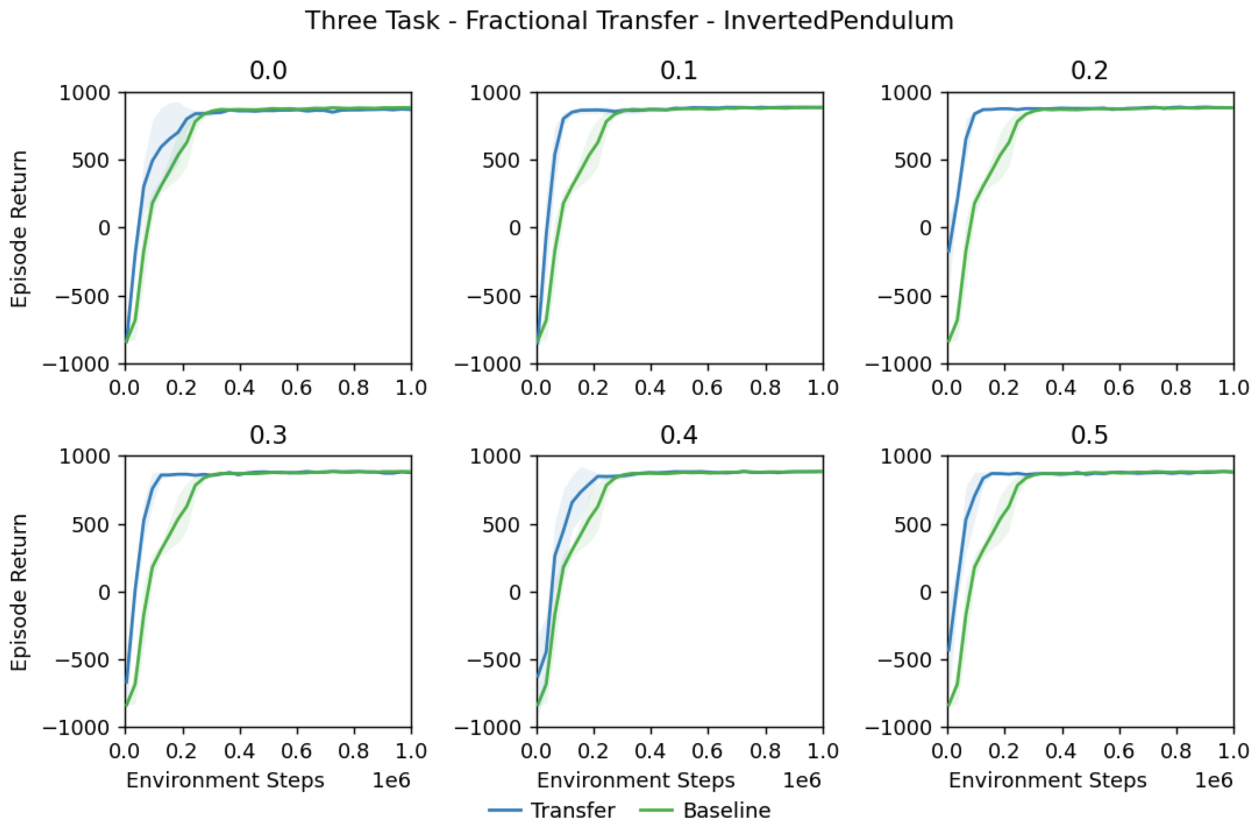


Figure 22: Episode return of having simultaneously trained on HalfCheetah, InvertedDoublePendulum and Hopper, after which fractional transfer was done for the InvertedPendulum task with fractions of 0.0 up to 0.5.

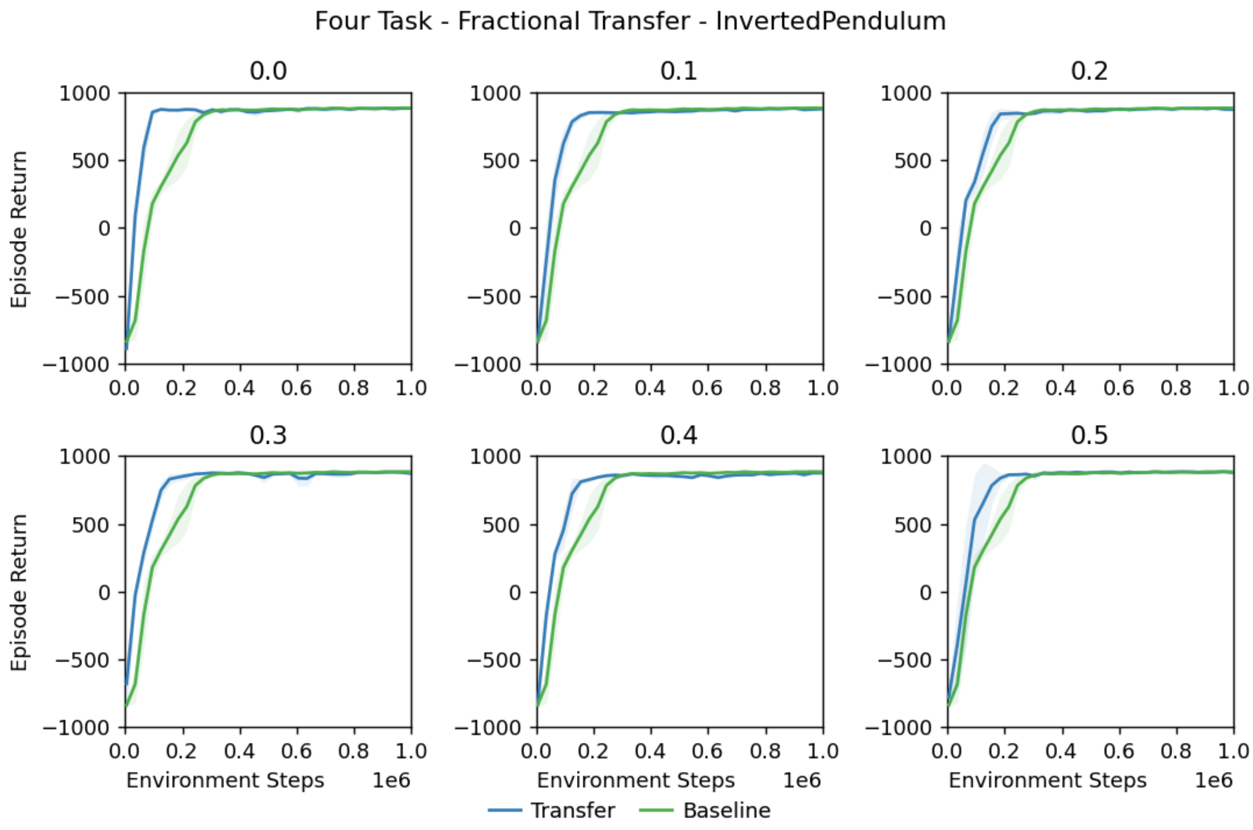


Figure 23: Episode return of having simultaneously trained on HalfCheetah, InvertedDoublePendulum, Hopper and Ant, after which fractional transfer was done for the InvertedPendulum task with fractions of 0.0 up to 0.5.

	2 Tasks	3 Tasks	4 Tasks
0.0	847 ± 135	779 ± 259	**838 ± 169**
0.1	857 ± 145	834 ± 198	803 ± 239
0.2	856 ± 121	**850 ± 139**	782 ± 269
0.3	**871 ± 93**	832 ± 182	806 ± 213
0.4	858 ± 150	789 ± 285	790 ± 237
0.5	852 ± 126	830 ± 181	791 ± 304
Baseline		723 ± 364	

Table 7: Average return for fraction transfer of 2, 3, and 4 source tasks for the InvertedPendulum task with fractions of 0.0 up to 0.5.

	2 Tasks	3 Tasks	4 Tasks
0.0	875 ± 135	871 ± 38	**881 ± 21**
0.1	884 ± 145	**887 ± 11**	874 ± 30
0.2	874 ± 121	884 ± 20	878 ± 34
0.3	884 ± 93	877 ± 36	877 ± 27
0.4	**885 ± 150**	883 ± 17	870 ± 40
0.5	881 ± 126	881 ± 20	881 ± 35
Baseline		883 ± 17	

Table 8: Average return for the final 1e5 environment steps of the training process for fraction transfer of 2, 3, and 4 source tasks for the InvertedPendulum task with fractions of 0.0 up to 0.5.

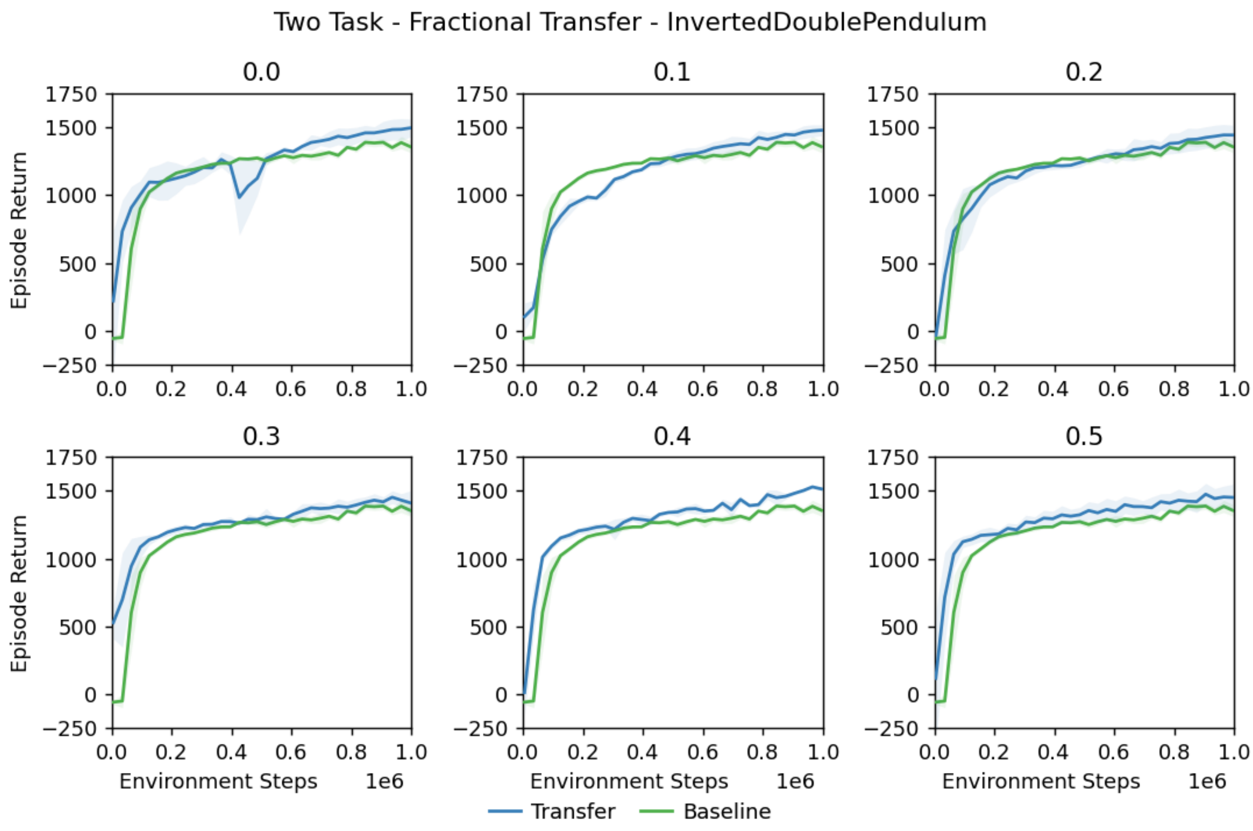


Figure 24: Episode return of having simultaneously trained on Hopper and InvertedPendulum, after which fractional transfer was done for the InvertedDoublePendulum task with fractions of 0.0 up to 0.5.

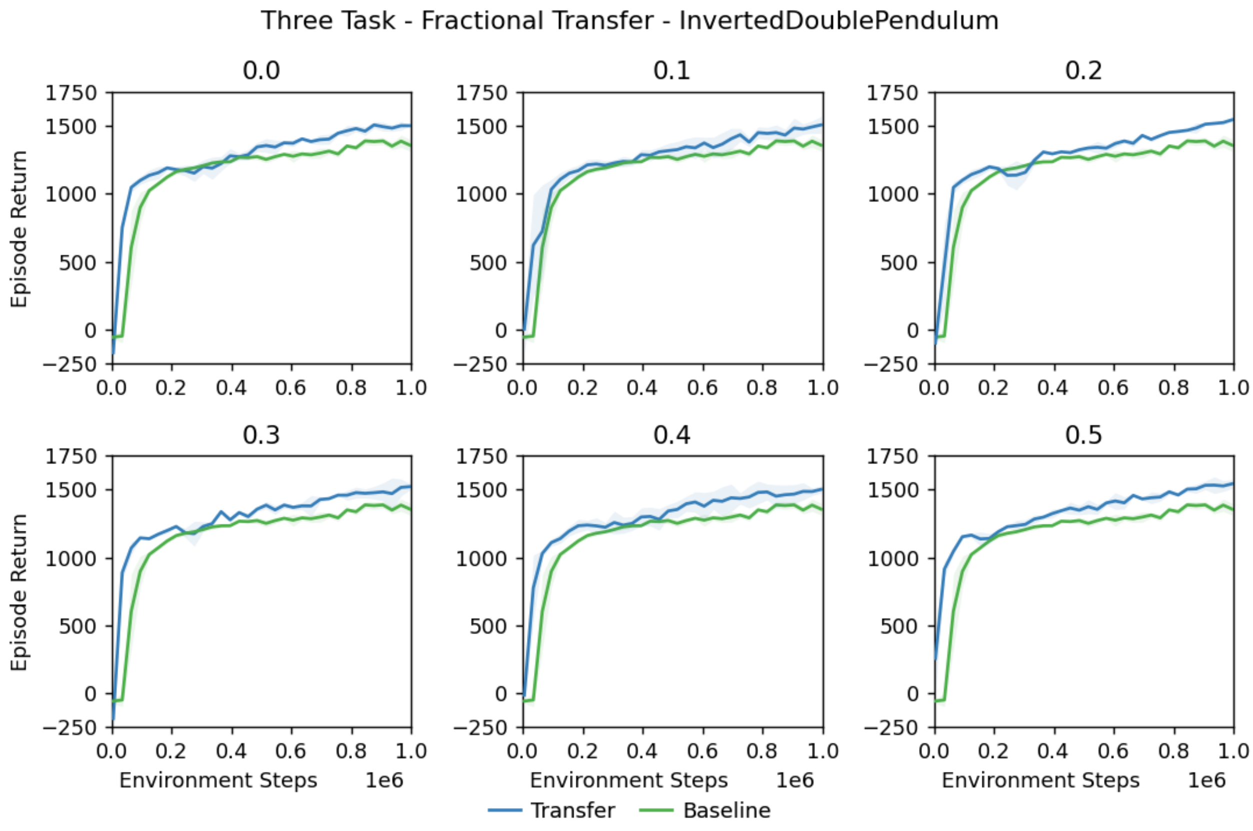


Figure 25: Episode return of having simultaneously trained on Hopper, InvertedPendulum and Walker2D, after which fractional transfer was done for the InvertedDoublePendulum task with fractions of 0.0 up to 0.5.

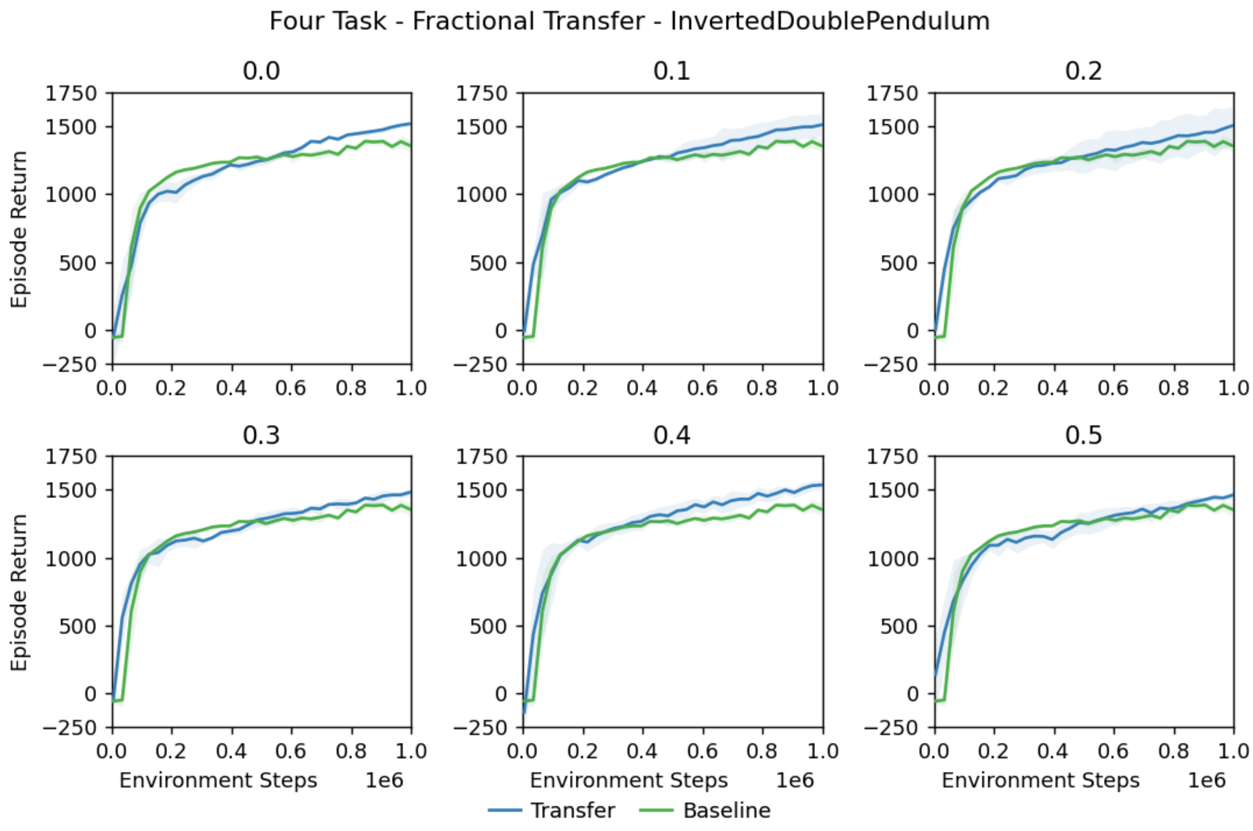


Figure 26: Episode return of having simultaneously trained on Hopper, InvertedPendulum, Walker2D and Ant, after which fractional transfer was done for the InvertedDoublePendulum task with fractions of 0.0 up to 0.5.

	2 Tasks	3 Tasks	4 Tasks
0.0	1255 ± 270	1303 ± 248	1208 ± 322
0.1	1185 ± 316	1283 ± 272	1251 ± 272
0.2	1209 ± 280	1299 ± 254	1235 ± 284
0.3	1279 ± 216	1325 ± 225	1239 ± 259
0.4	**1307** ± **227**	1323 ± 243	**1278** ± **287**
0.5	1301 ± 216	**1340** ± **230**	1206 ± 278
Baseline		1194 ± 306	

Table 9: Average return for fraction transfer of 2, 3, and 4 source tasks for the InvertedDoublePendulum task with fractions of 0.0 up to 0.5.

	2 Tasks	3 Tasks	4 Tasks
0.0	1489 ± 133	1494 ± 142	1506 ± 74
0.1	1473 ± 76	1488 ± 169	1502 ± 112
0.2	1438 ± 116	1531 ± 93	1481 ± 188
0.3	1428 ± 150	1506 ± 190	1469 ± 101
0.4	**1512** ± **126**	1493 ± 154	**1521** ± **116**
0.5	1450 ± 161	**1535** ± **159**	1449 ± 82
Baseline		1366 ± 179	

Table 10: Average return for the final 1e5 environment steps of the training process for fraction transfer of 2, 3, and 4 source tasks for the InvertedDoublePendulum task with fractions of 0.0 up to 0.5.

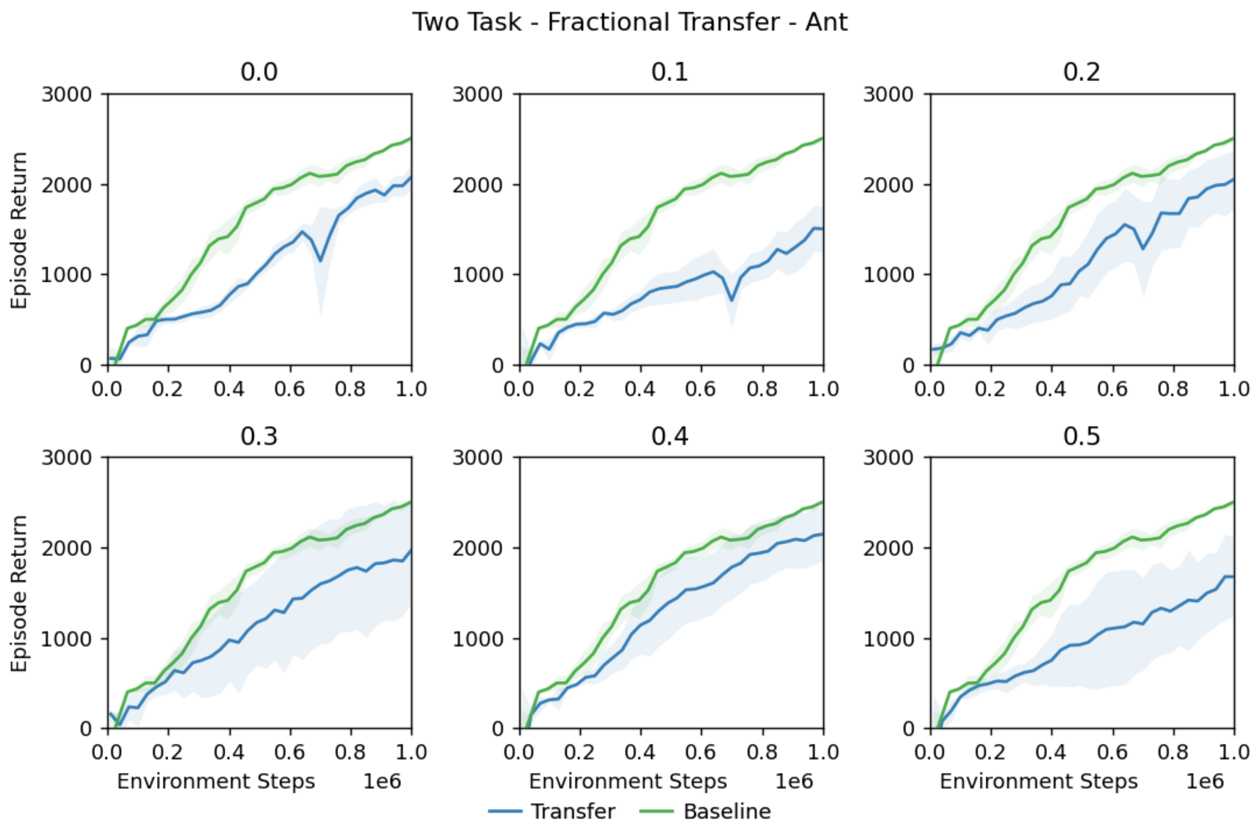


Figure 27: Episode return of having simultaneously trained on HalfCheetah and Walker2D, after which fractional transfer was done for the Ant task with fractions of 0.0 up to 0.5.

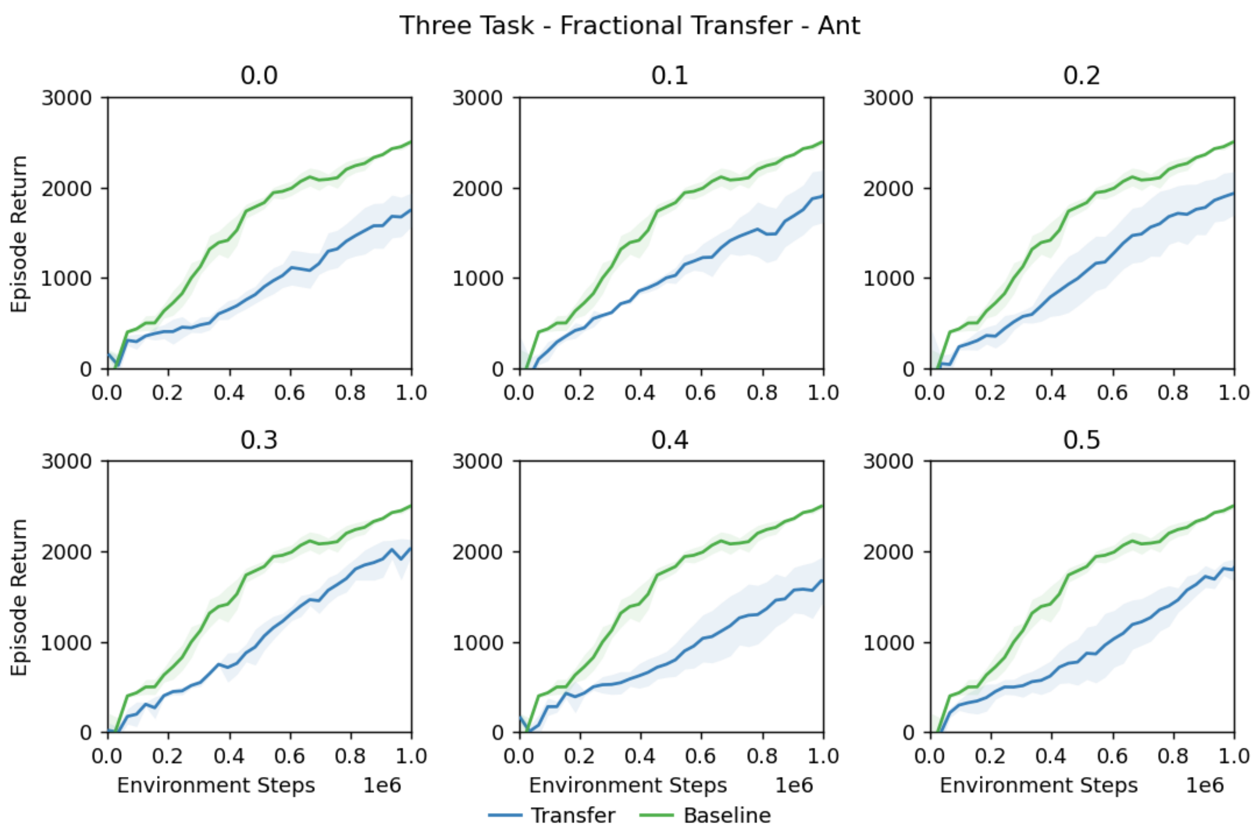


Figure 28: Episode return of having simultaneously trained on HalfCheetah, Walker2D, and Hopper, after which fractional transfer was done for the Ant task with fractions of 0.0 up to 0.5.

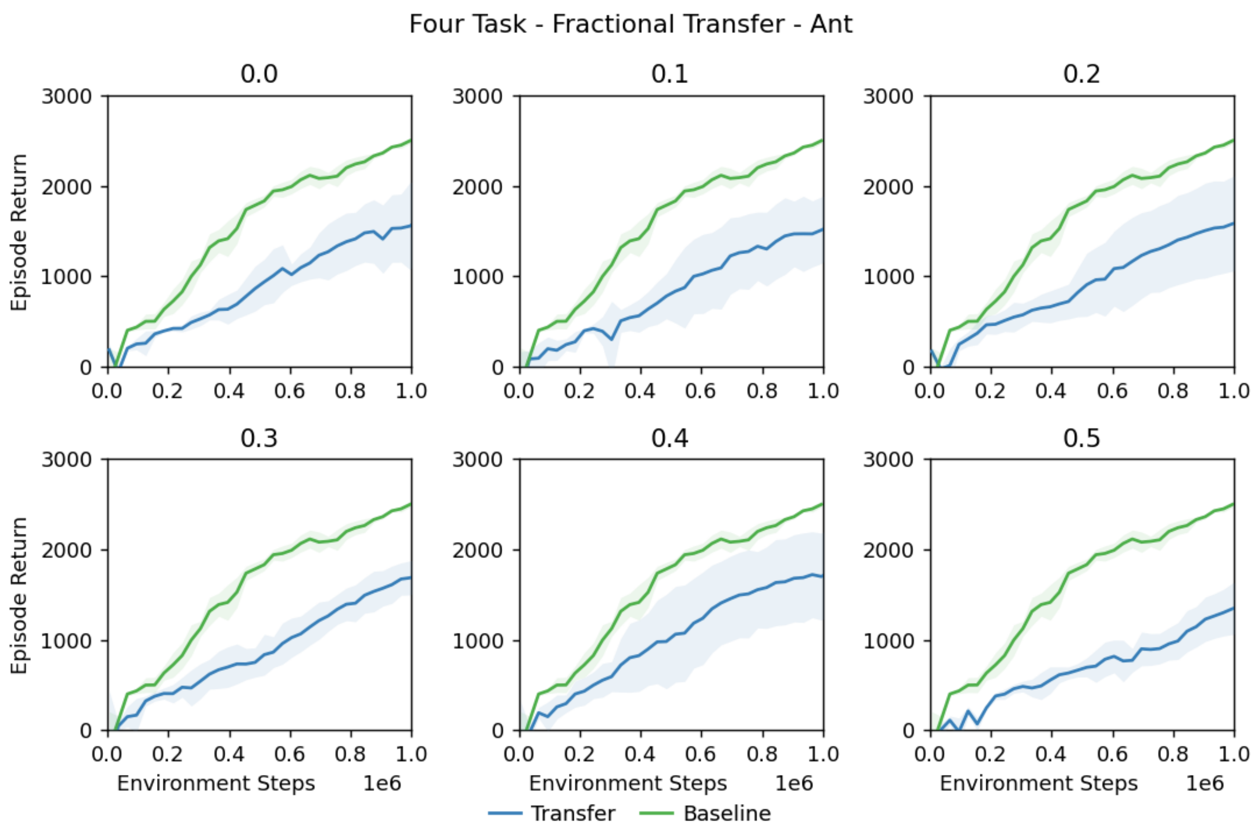


Figure 29: Episode return of having simultaneously trained on HalfCheetah, Walker2D, Hopper, and Inverted-DoublePendulum, after which fractional transfer was done for the Ant task with fractions of 0.0 up to 0.5.

	2 Tasks	3 Tasks	4 Tasks
0.0	1070 ± 659	923 ± 559	897 ± 586
0.1	806 ± 469	1022 ± 645	834 ± 652
0.2	1124 ± 722	1052 ± 687	898 ± 616
0.3	1162 ± 824	1080 ± 701	905 ± 554
0.4	1308 ± 735	885 ± 571	1022 ± 709
0.5	951 ± 651	932 ± 609	683 ± 498
Baseline		**1636 ± 895**	

Table 11: Average return for fraction transfer of 2, 3, and 4 source tasks for the Ant task with fractions of 0.0 up to 0.5.

	2 Tasks	3 Tasks	4 Tasks
0.0	1963 ± 659	1703 ± 256	1555 ± 515
0.1	1415 ± 469	1857 ± 380	1484 ± 471
0.2	2021 ± 722	1899 ± 292	1556 ± 520
0.3	1903 ± 824	1979 ± 346	1660 ± 248
0.4	2127 ± 735	1614 ± 371	1713 ± 545
0.5	1645 ± 651	1785 ± 394	1314 ± 293
Baseline		**2619 ± 261**	

Table 12: Average return for the final 1e5 environment steps of the training process for fraction transfer of 2, 3, and 4 source tasks for the Ant task with fractions of 0.0 up to 0.5.

5.2 Latent Task Classification

The results for the LTC with the HalfCheeetah and InvertedDoublePendulum as target tasks can be found in Table 13. All misclassifications of being a novel task for the InvertedDoublePendulum resulted from confusing it with the InvertedPendulum task. The classification errors for the Walker2D tasks were due to classifying it as the Hopper task.

The results for the LTC with the Hopper and Walker2D as target tasks can be found in Table 14. The classification errors for the InvertedPendulum and the InvertedDoublePendulum were due to confusions among these two tasks.

The results for the LTC with the Ant and InvertedPendulum as target tasks can be found in Table 15. The classification errors for the Walker2D and Hopper tasks were due to confusions among these two tasks.

	Novel tasks		Previous tasks			
Threshold	HalfCheetah	InvertedDoublePendulum	Hopper	Walker2D	Ant	InvertedPendulum
50	1.00	1.00	0.74/1.00	1.00/0.70	1.00/1.00	1.00/1.00
70	0.00	0.97	0.86/1.00	1.00/0.61	1.00/1.00	1.00/1.00
90	0.00	0.00	1.00/1.00	1.00/0.83	1.00/1.00	1.00/1.00

Table 13: Results for LTC for novel tasks HalfCheetah and InvertedDoublePendulum, where the task is classified as unseen before if it exceeds the threshold, otherwise classification of one of the known tasks is performed. The latter can be seen on the right side of the table: whether the samples of a pre-trained class were classified correctly, and not as a novel (left side of each entry) or different training task (right side of each entry).

	Novel tasks		Previous tasks			
Threshold	Hopper	Walker2D	HalfCheetah	InvertedPendulum	InvertedDoublePendulum	Ant
50	1.00	1.00	1.00/1.00	0.73/0.71	0.70/0.78	0.87/1.00
70	0.00	0.00	1.00/1.00	1.00/0.80	1.00/0.65	1.00/1.00
90	0.00	0.00	1.00/1.00	1.00/0.92	1.00/0.68	1.00/1.00

Table 14: Results for LTC for novel tasks Hopper and Walker2D, where the task is classified as unseen before if it exceeds the threshold, otherwise classification of one of the known tasks is performed. The latter can be seen on the right side of the table: whether the samples of a pre-trained class were classified correctly, and not as a novel (left side of each entry) or different training task (right side of each entry).

	Novel tasks		Previous tasks			
Threshold	Ant	InvertedPendulum	Hopper	Walker2D	InvertedDoublePendulum	HalfCheetah
50	1.00	1.00	1.00/0.60	1.00/0.65	0.92/1.00	1.00/1.00
70	0.00	0.00	1.00/0.70	1.00/1.00	1.00/1.00	1.00/1.00
90	0.00	0.00	1.00/0.72	1.00/0.85	1.00/1.00	1.00/1.00

Table 15: Results for LTC for novel tasks Hopper and Walker2D, where the task is classified as unseen before if it exceeds the threshold, otherwise classification of one of the known tasks is performed. The latter can be seen on the right side of the table: whether the samples of a pre-trained class were classified correctly, and not as a novel (left side of each entry) or different training task (right side of each entry).

5.3 Meta-Model Transfer Learning

In this section the results for the reward meta-model can be found, where each target task used 2, 3, and 4 source reward models. The results of the HalfCheetah, Hopper, Walker2D, InvertedPendulum, InvertedDoublePendulum, and Ant tasks can be found in Figure 30, Figure 31, Figure 32, Figure 33, Figure 34, and Figure 35 respectively. In each figure the overall performance of the transfer learning agent (blue) can be seen compared to the overall performance of a baseline agent (green), where for both the standard deviation of three runs is represented by the shaded area.

The overall average episode return, and episode return of the final 1e5 environment steps for all target tasks can be found in Table 16 and Table 17 respectively.

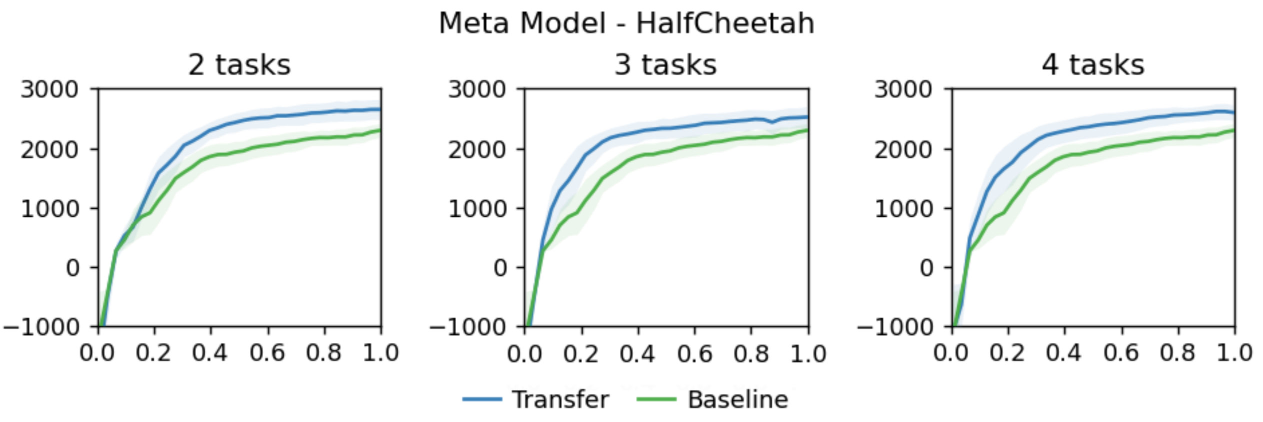


Figure 30: Episode return of using an UAE and reward meta-model for the HalfCheetah target task. For 2 source tasks Hopper and Ant were used, for 3 tasks Walker2D was added, and finally for 4 tasks InvertedPendulum was added.

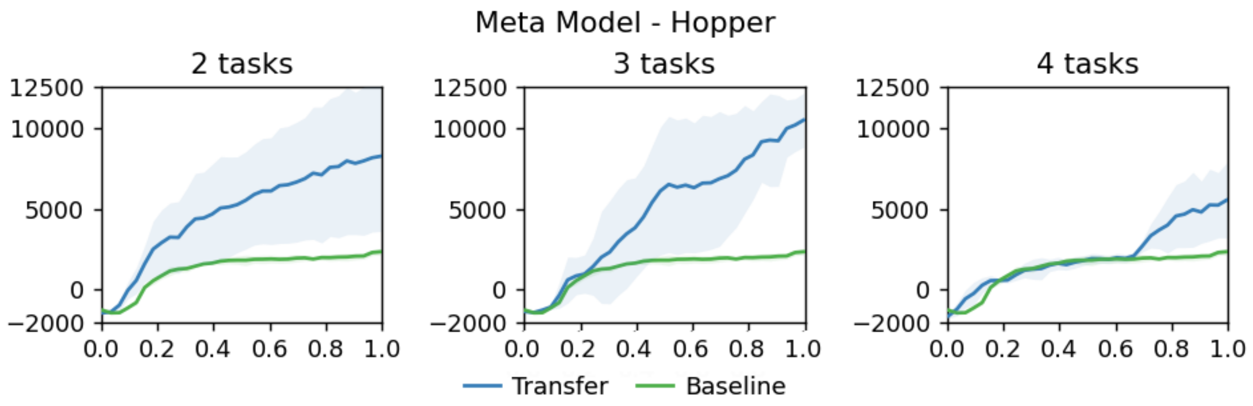


Figure 31: Episode return of using an UAE and reward meta-model for the Hopper target task. For 2 source tasks HalfCheetah and Walker2D were used, for 3 tasks Ant was added, and finally for 4 tasks InvertedPendulum was added.

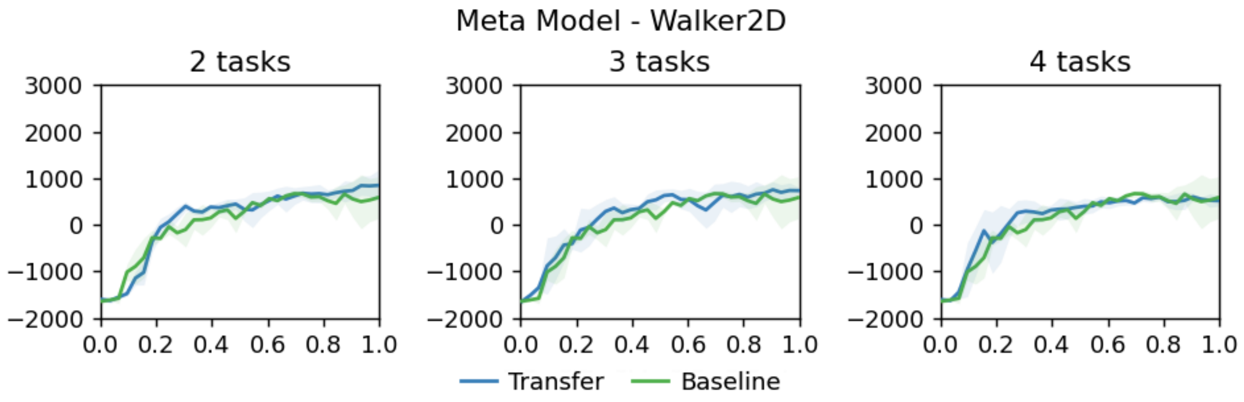


Figure 32: Episode return of using an UAE and reward meta-model for the Walker2D target task. For 2 source tasks HalfCheetah and Hopper were used, for 3 tasks Ant was added, and finally for 4 tasks InvertedPendulum was added.

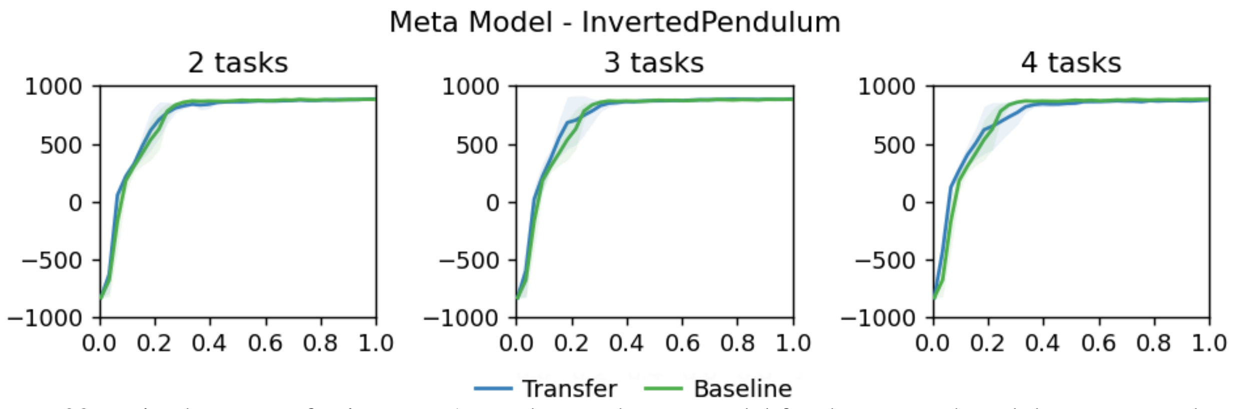


Figure 33: Episode return of using an UAE and reward meta-model for the InvertedPendulum target task. For 2 source tasks HalfCheetah and InvertedDoublePendulum were used, for 3 tasks Walker2D was added, and finally for 4 tasks Ant was added.

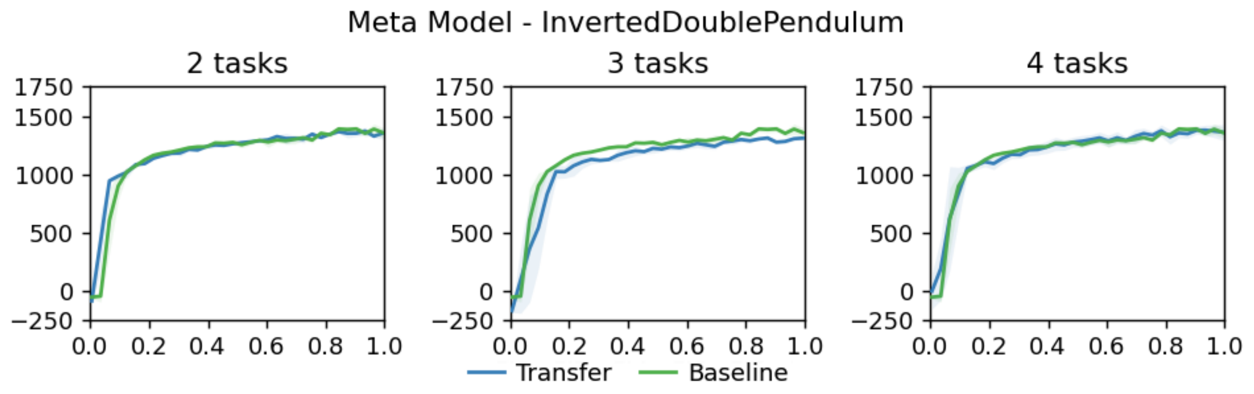


Figure 34: Episode return of using an UAE and reward meta-model for the InvertedDoublePendulum target task. For 2 source tasks Hopper and InvertedPendulum were used, for 3 tasks Walker2D was added, and finally for 4 tasks Ant was added.

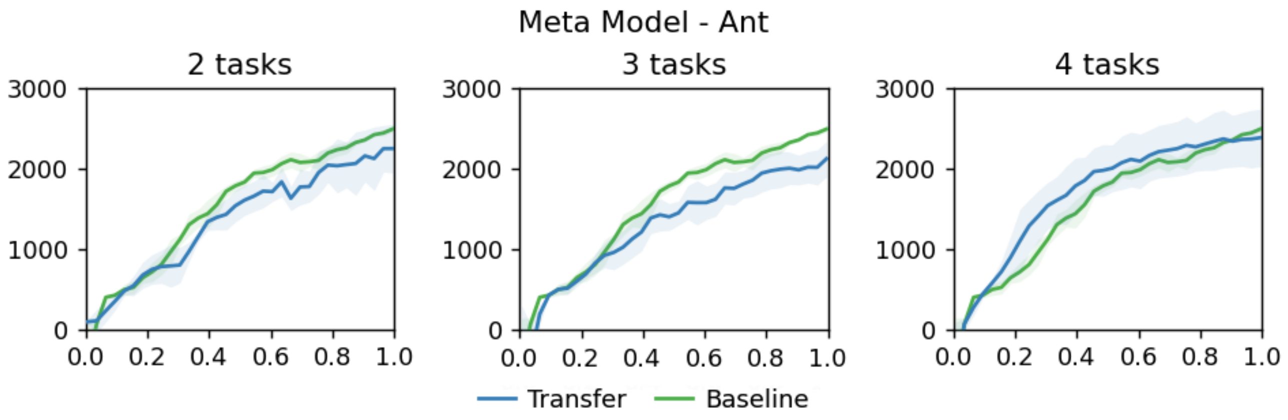


Figure 35: Episode return of using an UAE and reward meta-model for the Ant target task. For 2 source tasks HalfCheetah and Walker2D were used, for 3 tasks Hopper was added, and finally for 4 tasks InvertedPendulum was added.

	2 Tasks	3 Tasks	4 Tasks	Baseline
HalfCheetah	2057 ± 851	**2078 ± 721**	2018 ± 762	1681 ± 726
Hopper	**5085 ± 4277**	5019 ± 4813	2233 ± 2227	1340 ± 1112
InvertedPendulum	731 ± 332	**740 ± 333**	731 ± 302	723 ± 364
Walker2D	196 ± 860	**233 ± 823**	181 ± 799	116 ± 885
InvertedDoublePendulum	**1214 ± 230**	1120 ± 327	1198 ± 295	1194 ± 306
Ant	1423 ± 788	1366 ± 687	**1765 ± 764**	1589 ± 771

Table 16: Average episode return of using a UAE and reward meta-model, using 2, 3, and 4 source tasks transferred to the HalfCheetah, Hopper, Walker2D, InvertedPendulum, InvertedDoublePendulum, and Ant tasks.

	2 Tasks	3 Tasks	4 Tasks	Baseline
HalfCheetah	**2618** ± **362**	2514 ± 183	2614 ± 160	2264 ± 160
Hopper	8080 ± 4704	**10178** ± **2241**	5356 ± 2390	2241 ± 502
InvertedPendulum	**885** ± **14**	884 ± 13	875 ± 21	883 ± 17
Walker2D	**846** ± **286**	730 ± 343	543 ± 430	547 ± 710
InvertedDoublePendulum	1348 ± 171	1302 ± 152	**1368** ± **156**	1366 ± 179
Ant	2212 ± 607	2064 ± 386	2366 ± 427	**2463** ± **208**

Table 17: Final averaged episode return of the last 1e5 environment steps using a UAE and reward meta-model, using 2, 3, and 4 source tasks transferred to HalfCheetah, Hopper, Walker2D, InvertedPendulum, InvertedDoublePendulum, and Ant tasks.

6 Discussion

In this chapter we summarize, discuss and conclude the thesis. We first answer the research questions whilst discussing the relevant results of the previous chapter in Section 6.1. Next, we summarize this thesis by concisely providing the main contributions in Section 6.2. Finally, in Section 6.3 we propose potential future directions of research that can follow from the proposed methodologies, and we conclude the thesis in Section 6.4.

6.1 Answers to Research Questions

Here we answer and discuss the research questions that were presented in Section 1.1:

1. **Does transferring knowledge of deep model-based reinforcement learning agents trained on multiple tasks simultaneously serve as an effective multi-source transfer learning approach?**

   We found that using the state-of-the-art deep MBRL algorithm for continuous control tasks, simultaneously training on multiple tasks does indeed result in a positive transfer in most cases, both in terms of overall average performance and ultimate performance. However, depending on the number of relevant source tasks, this approach may also result in a neutral or negative transfer. For instance, for the HalfCheetah, Hopper, and Walker2D locomotion testing environments, we generally observe a decrease in both overall and ultimate performance as we add more tasks that provide irrelevant information to the target domain, such as the pendula environments (Table 1, Table 2, Table 3, Table 4, Table 5, Table 6). Note that the number of source tasks, regardless of relevance, should not influence the performance, even though for any number of source tasks we trained each multi-task agent for the same number of environment steps. Results of simultaneous multi-task agents show that for the same tasks there are similar performances across 2, 3, and 4 tasks (see Appendix B). This assumption reflects in the results of the pendula testing environments, where the third and fourth added tasks are both irrelevant, yet 3 source tasks does not necessarily outperform 4 source tasks (Table 7, Table 8, Table 9, Table 10). These results also suggest, that as long as there is at least one relevant source task in the multi-source collection, a positive transfer is likely to occur. This can be observed for the InvertedPendulum and InverterdDoublePendulum test environments, where these two environments are the only relevant sources for each other. Lastly, for the Ant testing environment we observe significant negative transfers for any set of source tasks, which is likely due to the movement dynamics and reward function being too different from the other domains, meaning all source tasks provide irrelevant (and therefore interfering) information (Table 11, Table 12).

2. **Can transferring fractions of neural network parameters be used as an alternative transfer learning approach for certain models?**

   We showed that it is indeed possible to transfer fractions of parameters of neural network layers, rather than transferring all parameters or random initialization. This can be done by adding a fraction of the source parameters to randomly initialized parameters. We identified that this type of TL is beneficial for DRL components such as the reward model and value model, as these components suffer from overfitting when fully transferred, and do not require to be re-initialized as they are unrelated to actions. We found that compared to randomly initializing the last layer of components (as done for the previous research question), FTL results in significantly better overall and ultimate performance for almost all testing environments, except for the Ant environment as argued for earlier. The magnitude of the performance improvement over the baseline and randomly initializing differs for different sizes of the fraction parameter. For instance, the performance of transferring 50% of the parameters seems to result in the worst performance improvements, and sometimes even negative transfers. This is likely due to the fact that too much information is being transferred, which may therefore need to be unlearned, and consequently results in interference with the learning process. In general, we observe that fractions between 10% and 40% result in the largest magnitudes of performance improvement, though there is no clear favourite amongst the different testing environments, nor a clear correlation with the number of (relevant) source tasks. Therefore we advise experimentation of a couple different fraction values in the range $[0.1, 0.4]$, as certain fractions may unlock massive performance improvements (see $\omega = 0.1$ in Figure 16, and $\omega = 0.2$ in Figure 18).

3. **How can knowledge of multiple separate deep reinforcement learning models be combined to serve as a multi-source transfer learning technique?**

   We presented an approach that allows the combination of information from several separate DRL models, by means of a meta-model. However, in order to proceed with this approach, a pre-trained UAE was assumed to be necessary, of which the encoder was frozen for training both the source models and meta-model. By doing so, a universal feature space could be realized with which each of the different models were trained on, allowing us to combine them with the meta-model. We accomplished multi-source TL of the like by feeding latent input states to all of the frozen source models, of which the corresponding predictions were concatenated with the same latent state to serve as input for the meta-model. The experiments showed that this type of multi-source TL results in a positive transfer for the majority of test cases when only applying the method to the reward model (Table 16, Table 17). The results suggest that, compared to (fractionally) transferring parameters of simultaneous multi-task learning agents, the meta-model approach is more robust in terms of avoiding significant negative transfers, regardless of the relevance of the source domains. However, this avoidance may also have resulted from the fact that we did not transfer the dynamics model in meta-model transfer learning, whereas with FTL we fully transferred the dynamics model, which likely interferes with the learning process of the Ant testing environment. Moreover, this approach does not introduce a tunable hyperparameter, and retains knowledge of source tasks. However, the results also show that the magnitude of the positive transfers is often much lower than those of FTL. With FTL the knowledge is transferred more directly as it is parameter-based, which reflects in the results as both negative and positive transfers are of significantly larger magnitude compared to the meta-model results. The less direct impact of meta-model TL therefore also results in more frequent neutral transfers.

4. **How can both detection of novel domains and classification of known domains be accomplished in a multi-task setting for deep model-based reinforcement learning architectures?**

   Given the usage of separate models trained on different tasks in the meta-model approach, we decided it would be interesting to investigate whether we can perform classification of previous tasks and detection of novel tasks. This would allow such a system to be able to switch between stored reward models and the meta-model depending on which task it is facing. We showed that this is indeed possible by performing LTC using a KNN classifier. That is, using the same UAE as before, a KNN was trained on latent samples for four different source tasks. Afterwards, we performed experiments with several different sets of source tasks and novel tasks, in order to evaluate whether this approach allows classification and detection respectively. By computing the Manhattan distance to the nearest neighbor of a given sample, we showed that a novel task can be detected if the distance exceeds a certain threshold. For distances below this threshold, we simply classified the sample using the KNN. The results show that detection of novel tasks appears to work almost perfectly for a threshold of 50. They also show that for samples of previous tasks, it may occur that the sample is classified as novel, or as a different known task, depending on the similarity between the domains (Table 13, Table 14, Table 15). However, the number of misclassifications is never less than 50% of the testing samples, meaning that classification can be done almost perfectly when determining the classification based on the average of 100 samples.

6.2 Main Contributions

To summarize this thesis, the key contributions are as follows:

- **Simultaneous multi-task transfer learning** We showed that simultaneously training a single deep model-based reinforcement learning agent on multiple tasks provides for a multi-source TL approach that results in positive transfers. However, this is only the case in the presence of at least one relevant source task, where the sole number of source tasks does not necessarily influence the performance.

- **Fractional transfer learning** A novel TL approach was introduced which, rather than fully transferring or randomly initializing neural network weights, allows fractions of the parameters to be transferred. The results showed that applying this method to relevant models of deep reinforcement learning architectures, such as the value model and reward model, significant performance improvements can be expected. Hence, we introduced a promising tunable hyperparameter for transfer learning in deep reinforcement learning, given at least one source task is relevant.

- **Meta-model transfer learning** We presented a novel multi-source TL approach that allows the combination of multiple separate sources for components of deep model-based reinforcement learning architectures. After having trained each model in a universal feature space provided by a universal autoencoder, the outputs of the source models are combined and fed to a meta-model. The results show that also this multi-source TL approach provides for positive transfers in most cases, whilst, regardless of source task domain relevance, never resulting in significant negative transfers.

- **Latent task classification** By training a K-Nearest Neighbor classifier on a set of source tasks, we introduced a novel latent task classification method that allows detection of novel tasks, as well as classification of previously seen tasks. This was accomplished by introducing a threshold parameter that, given the Manhattan distance to the nearest neighbor for a new sample, determines whether a task is novel or not. This approach can allow reinforcement learning agents to switch between separate models that were trained on different tasks, or train a new model when facing a novel task.

6.3 Future Work

First of all, all methods proposed in this study were applied to continuous control tasks that share the same physical laws. Therefore, one direction for future work could be to evaluate the effectiveness of the proposed approaches in discrete domains. For instance, one could train a single agent on multiple discrete tasks simultaneously, and transfer its knowledge to novel discrete tasks using FTL. Moreover, the proposed methodologies were all applied to Dreamer, meaning it would be interesting to evaluate these approaches on different algorithms as well.

In this work we applied meta-model TL to solely the reward model. Another future direction would therefore be to apply this method to other components, such as the value model or dynamics model. Similarly, FTL could perhaps be applied to other components as well, such as the last layer of the forward dynamics model.

LTC was evaluated using samples from single random agents, meaning the observations were often of unnatural nature. An interesting future direction would therefore be to evaluate this approach on samples of agents that have learned to (almost) solve the tasks at hand, as supposedly the samples of a given task would be more similar to each other, and more different to random samples from a novel task. Additionally, the recent surge of contrastive learning for supervised learning allows learning of similar feature representations for samples of the same class, which could therefore be particularly useful in the case of LTC [54, 74]. By increasing the similarity between samples of the same class, the threshold for novel tasks could be increased, which would allow for more robust classifications.

Another interesting direction for future work would be to combine meta-model TL with LTC in a multi-task learning setting. This combination could allow RL agents to switch between frozen pre-trained models for known tasks, and a meta-model for novel tasks. That is, order to autonomously know which task the agent is facing, LTC can be leveraged. Finally, one could investigate to what extent pre-training a UAE on multiple different domains benefits the generalizability for reconstructing novel domains.

6.4 Conclusion

In this thesis we investigated multi-source transfer learning for deep model-based reinforcement learning. The main conclusion we arrived at is that multi-source transfer learning approaches can result in significant and robust positive transfers, dependent on what type of approach is used. We introduced several novel approaches that can be used for transferring knowledge from multiple sources for reinforcement learning agents that leverage World Models. Moreover, we introduced a method that allows agents to classify previously seen domains, and to detect novel domains.

The purpose of this thesis was to provide insights of combining model-based reinforcement learning with multi-task learning and transfer learning. These are still emerging, yet essential topics for advancing the field of reinforcement learning. Therefore, we hope that this work has provided a promising foundation for future research in this area to build upon.

Bibliography

[1] M. A. Wiering and M. Van Otterlo, "Reinforcement learning," *Adaptation, learning, and optimization*, vol. 12, no. 3, 2012.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[3] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[4] T. M. Moerland, J. Broekens, and C. M. Jonker, "Model-based reinforcement learning: A survey," *arXiv preprint arXiv:2006.16712*, 2020.

[5] Z. Zhu, K. Lin, and J. Zhou, "Transfer learning in deep reinforcement learning: A survey," *arXiv preprint arXiv:2009.07888*, 2020.

[6] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan. 2016.

[7] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, vol. 575, pp. 350–354, Oct. 2019.

[8] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, *et al.*, "Dota 2 with large scale deep reinforcement learning," *arXiv preprint arXiv:1912.06680*, 2019.

[9] D. Hafner, T. Lillicrap, J. Ba, and M. Norouzi, "Dream to control: Learning behaviors by latent imagination," *arXiv preprint arXiv:1912.01603*, 2019.

[10] D. Hafner, T. Lillicrap, M. Norouzi, and J. Ba, "Mastering atari with discrete world models," *arXiv preprint arXiv:2010.02193*, 2020.

[11] D. Ha and J. Schmidhuber, "World models," *arXiv preprint arXiv:1803.10122*, 2018.

[12] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson, "Learning latent dynamics for planning from pixels," in *International Conference on Machine Learning*, pp. 2555–2565, PMLR, 2019.

[13] A. Zhang, H. Satija, and J. Pineau, "Decoupling dynamics and reward for transfer learning," *arXiv preprint arXiv:1804.10689*, 2018.

[14] I. Osband, C. Blundell, A. Pritzel, and B. Van Roy, "Deep exploration via bootstrapped dqn," *Advances in neural information processing systems*, vol. 29, pp. 4026–4034, 2016.

[15] R. Bellman, "A markovian decision process," *Journal of Mathematics and Mechanics*, vol. 6, no. 5, pp. 679–684, 1957.

[16] R. Bellman, "Dynamic programming and stochastic control processes," *Information and Control*, vol. 1, pp. 228–239, Sept. 1958.

[17] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, May 1992.

[18] M. Wiering and J. Schmidhuber, "HQ-learning," *Adaptive Behavior*, vol. 6, no. 2, pp. 219–246, 1997.

[19] M. Wiering and J. Schmidhuber, "Fast online Q (λ)," *Machine Learning*, vol. 33, no. 1, pp. 105–115, 1998.

[20] M. A. Wiering, "QV (lambda)-learning: A new on-policy reinforcement learning algrithm," *Proceedings of the 7th european workshop on*, 2005.

[21] M. Sabatelli, G. Louppe, P. Geurts, and others, "Deep quality-value (DQV) learning," *arXiv preprint arXiv*, 2018.

[22] H. Hasselt, "Double Q-learning," in *Advances in Neural Information Processing Systems* (J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, eds.), vol. 23, Curran Associates, Inc., 2010.

[23] S. Z. Gou and Y. Liu, "DQN with model-based exploration: efficient learning on environments with sparse rewards," *arXiv preprint arXiv:1903.09295*, 2019.

[24] K. Chua, R. Calandra, R. McAllister, and S. Levine, "Deep reinforcement learning in a handful of trials using probabilistic dynamics models," in *Advances in Neural Information Processing Systems* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), vol. 31, Curran Associates, Inc., 2018.

[25] R. Sekar, O. Rybkin, K. Daniilidis, P. Abbeel, D. Hafner, and D. Pathak, "Planning to explore via self-supervised world models," in *International Conference on Machine Learning*, pp. 8583–8592, PMLR, 2020.

[26] M. A. Wiering, M. Withagen, and others, "Model-based multi-objective reinforcement learning," *2014 IEEE Symposium on*, 2014.

[27] M. Zhao, Z. Liu, S. Luan, S. Zhang, D. Precup, and Y. Bengio, "A consciousness-inspired planning agent for model-based reinforcement learning," *arXiv preprint arXiv:2106.02097*, 2021.

[28] A. A. Taïga, W. Fedus, M. C. Machado, A. Courville, and M. G. Bellemare, "Benchmarking bonus-based exploration methods on the arcade learning environment," *arXiv preprint arXiv:1908.02388*, 2019.

[29] B. C. Stadie, S. Levine, and P. Abbeel, "Incentivizing exploration in reinforcement learning with deep predictive models," *arXiv preprint arXiv:1507.00814*, 2015.

[30] M. Wiering and J. Schmidhuber, "Efficient model-based exploration," in *PROCEEDINGS OF THE SIXTH INTERNATIONAL CONFERENCE ON SIMULATION OF ADAPTIVE BEHAVIOR: FROM ANIMALS TO ANIMATS 6*, pp. 223–228, MIT Press/Bradford Books, 1998.

[31] M. P. Deisenroth, D. Fox, and C. E. Rasmussen, "Gaussian processes for data-efficient learning in robotics and control," *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 2, pp. 408–423, 2013.

[32] E. F. Camacho and C. Bordons, *Model Predictive Control.* Springer, London, 1999.

[33] H. Van Hasselt and M. A. Wiering, "Reinforcement learning in continuous action spaces," *2007 IEEE International*, 2007.

[34] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *Computers and Games*, pp. 72–83, Springer Berlin Heidelberg, 2007.

[35] M. Henaff, W. F. Whitney, and Y. LeCun, "Model-based planning with discrete and continuous actions," *arXiv preprint arXiv:1705.07177*, 2017.

[36] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain," *Psychol. Rev.*, vol. 65, pp. 386–408, Nov. 1958.

[37] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[38] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[39] "A simple overview of multilayer perceptron." https://www.analyticsvidhya.com/blog/2020/12/mlp-multilayer-perceptron-simple-overview/. Accessed: 2019-12-13.

[40] V. H. Phung, E. J. Rhee, *et al.*, "A high-accuracy model average ensemble of convolutional neural networks for classification of cloud image patches on small datasets," *Applied Sciences*, vol. 9, no. 21, p. 4500, 2019.

[41] D. H. Ballard, "Modular learning in neural networks.," in *AAAI*, pp. 279–284, 1987.

[42] "All about autoencoders." https://pythonmachinelearning.pro/all-about-autoencoders/. Accessed: 2017-10-30.

[43] R. Silva and A. Araújo, "A novel approach to condition monitoring of the cutting process using recurrent neural networks," *Sensors*, vol. 20, no. 16, p. 4493, 2020.

[44] G. Tesauro, "TD-Gammon: A Self-Teaching backgammon program," in *Applications of Neural Networks* (A. F. Murray, ed.), pp. 267–285, Boston, MA: Springer US, 1995.

[45] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[46] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[47] N. Wahlström, T. B. Schön, and M. P. Deisenroth, "Learning deep dynamical models from image pixels," *IFAC-PapersOnLine*, vol. 48, no. 28, pp. 1059–1064, 2015.

[48] C. Finn, X. Y. Tan, Y. Duan, T. Darrell, S. Levine, and P. Abbeel, "Deep spatial autoencoders for visuomotor learning," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 512–519, IEEE, 2016.

[49] M. Watter, J. T. Springenberg, J. Boedecker, and M. Riedmiller, "Embed to control: A locally linear latent dynamics model for control from raw images," *arXiv preprint arXiv:1506.07365*, 2015.

[50] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine, *et al.*, "Model-based reinforcement learning for atari," *arXiv preprint arXiv:1903.00374*, 2019.

[51] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Trans. Knowl. Data Eng.*, vol. 22, pp. 1345–1359, Oct. 2010.

[52] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, "A comprehensive survey on transfer learning," *Proceedings of the IEEE*, vol. 109, no. 1, pp. 43–76, 2020.

[53] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?," *arXiv preprint arXiv:1411.1792*, 2014.

[54] Z. Cao, M. Kwon, and D. Sadigh, "Transfer reinforcement learning across homotopy classes," *CoRR*, vol. abs/2102.05207, 2021.

[55] M. Sabatelli, M. Kestemont, W. Daelemans, and P. Geurts, "Deep transfer learning for art classification problems," in *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, pp. 0–0, 2018.

[56] J. L. Carroll and T. S. Peterson, "Fixed vs. dynamic Sub-Transfer in reinforcement learning," *ICMLA*, 2002.

[57] F. Fernández and M. M. Veloso, "Reusing and building a policy library," *ICAPS*, 2006.

[58] S. Schaal, A. J. Ijspeert, A. Billard, S. Vijayakumar, and J.-A. Meyer, "Estimating future reward in reinforcement learning animats using associative learning," in *From animals to animats 8: Proceedings of the Eighth International Conference on the Simulation of Adaptive Behavior*, pp. 297–304, MIT Press, 2004.

[59] Y. Zhu, R. Mottaghi, E. Kolve, J. J. Lim, A. Gupta, L. Fei-Fei, and A. Farhadi, "Target-driven visual navigation in indoor scenes using deep reinforcement learning," in *2017 IEEE international conference on robotics and automation (ICRA)*, pp. 3357–3364, IEEE, 2017.

[60] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[61] E. Parisotto, J. L. Ba, and R. Salakhutdinov, "Actor-mimic: Deep multitask and transfer reinforcement learning," *arXiv preprint arXiv:1511.06342*, 2015.

[62] A. A. Rusu, S. G. Colmenarejo, C. Gulcehre, G. Desjardins, J. Kirkpatrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, and R. Hadsell, "Policy distillation," *arXiv preprint arXiv:1511.06295*, 2015.

[63] Y. W. Teh, V. Bapst, W. M. Czarnecki, J. Quan, J. Kirkpatrick, R. Hadsell, N. Heess, and R. Pascanu, "Distral: Robust multitask reinforcement learning," *arXiv preprint arXiv:1707.04175*, 2017.

[64] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell, "Progressive neural networks," *arXiv preprint arXiv:1606.04671*, 2016.

[65] M. E. Taylor and P. Stone, "Transfer learning for reinforcement learning domains: A survey.," *Journal of Machine Learning Research*, vol. 10, no. 7, 2009.

[66] B. Eysenbach, S. Asawa, S. Chaudhari, S. Levine, and R. Salakhutdinov, "Off-dynamics reinforcement learning: Training for transfer with domain classifiers," *arXiv preprint arXiv:2006.13916*, 2020.

[67] L. Chen, K. Lee, A. Srinivas, and P. Abbeel, "Improving computational efficiency in visual reinforcement learning via stored embeddings," *arXiv preprint arXiv:2103.02886*, 2021.

[68] N. C. Landolfi, G. Thomas, and T. Ma, "A model-based approach for sample-efficient multi-task reinforcement learning," *arXiv preprint arXiv:1907.04964*, 2019.

[69] E. Fix and J. L. Hodges Jr, "Discriminatory analysis-nonparametric discrimination: Small sample performance," tech. rep., California Univ Berkeley, 1952.

[70] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, "On the surprising behavior of distance metrics in high dimensional space," in *International conference on database theory*, pp. 420–434, Springer, 2001.

[71] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning." `http://pybullet.org`, 2016–2021.

[72] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[73] A. Raffin and F. Stulp, “Generalized state-dependent exploration for deep reinforcement learning in robotics,” *arXiv preprint arXiv:2005.05719*, 2020.

[74] O. Henaff, “Data-Efficient image recognition with contrastive predictive coding,” in *Proceedings of the 37th International Conference on Machine Learning* (H. D. Iii and A. Singh, eds.), vol. 119 of *Proceedings of Machine Learning Research*, pp. 4182–4192, PMLR, 2020.

Appendices

A Baselines

In Figure 36 the baseline performances of Dreamer can be seen for each of the environments for three random seeds.

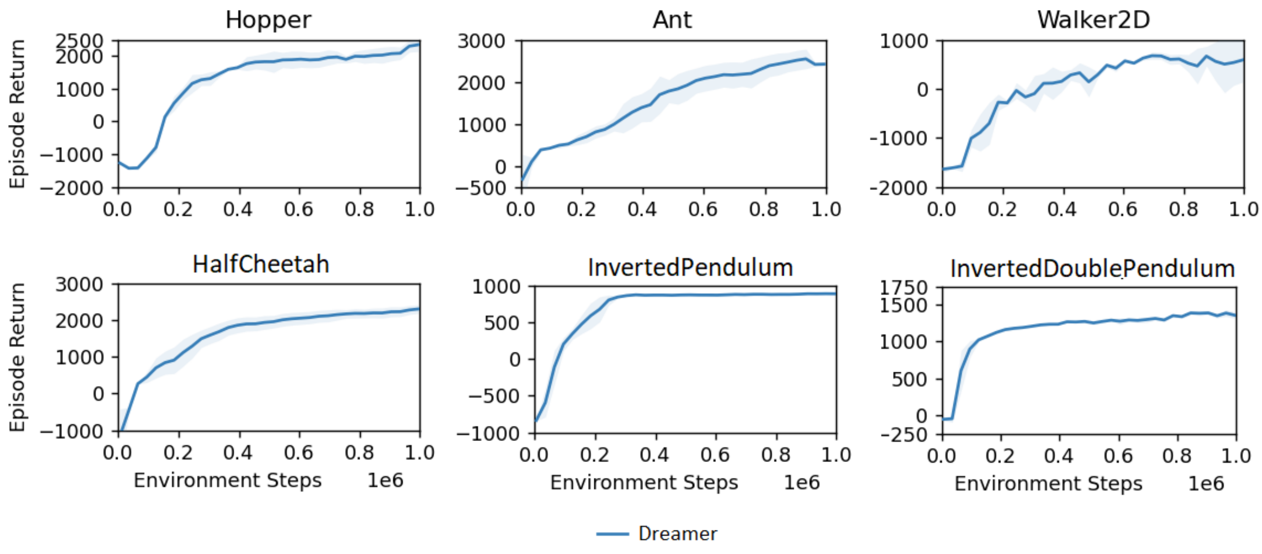


Figure 36: Performance of the base Dreamer model on the 6 tasks used in this work, averaged performance over 3 seeds.

B Simultaneous Multi-Task Learning

Example performances of Dreamer agents simultaneously trained on 2, 3, and 4 tasks for 2e6 environment steps can be found in Figure 37, Figure 38, and Figure 39 respectively.

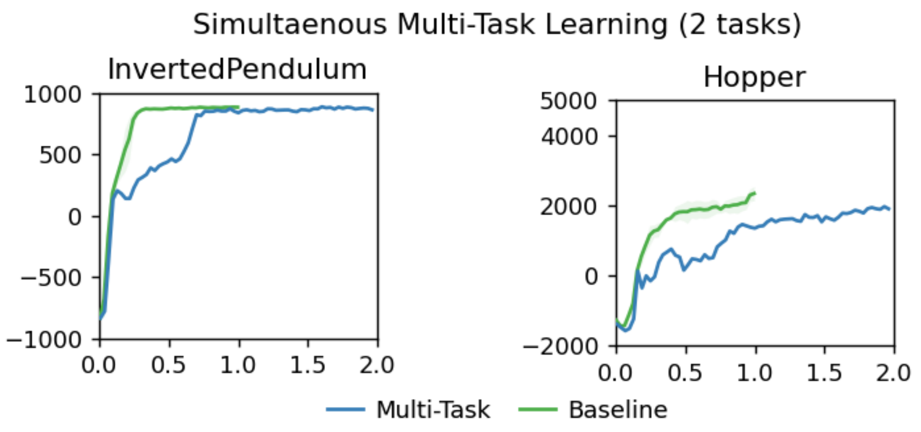


Figure 37: Dreamer simultaneously trained on the InvertedPendulum and Hopper tasks for 2e6 environment steps, along with the corresponding 1e6 environment steps baselines.

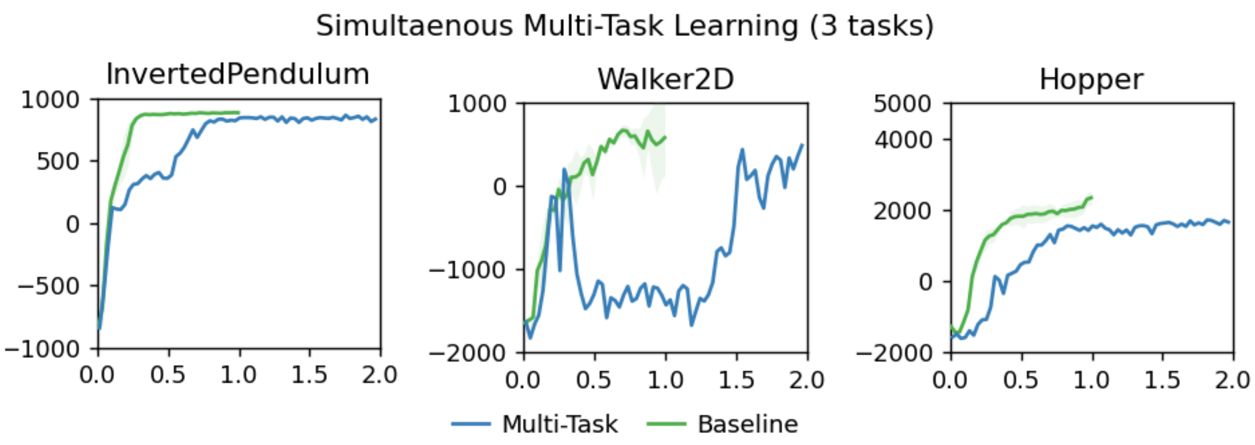


Figure 38: Dreamer simultaneously trained on the InvertedPendulum, Walker2D, Hopper tasks for 2e6 environment steps, along with the corresponding 1e6 environment steps baselines.

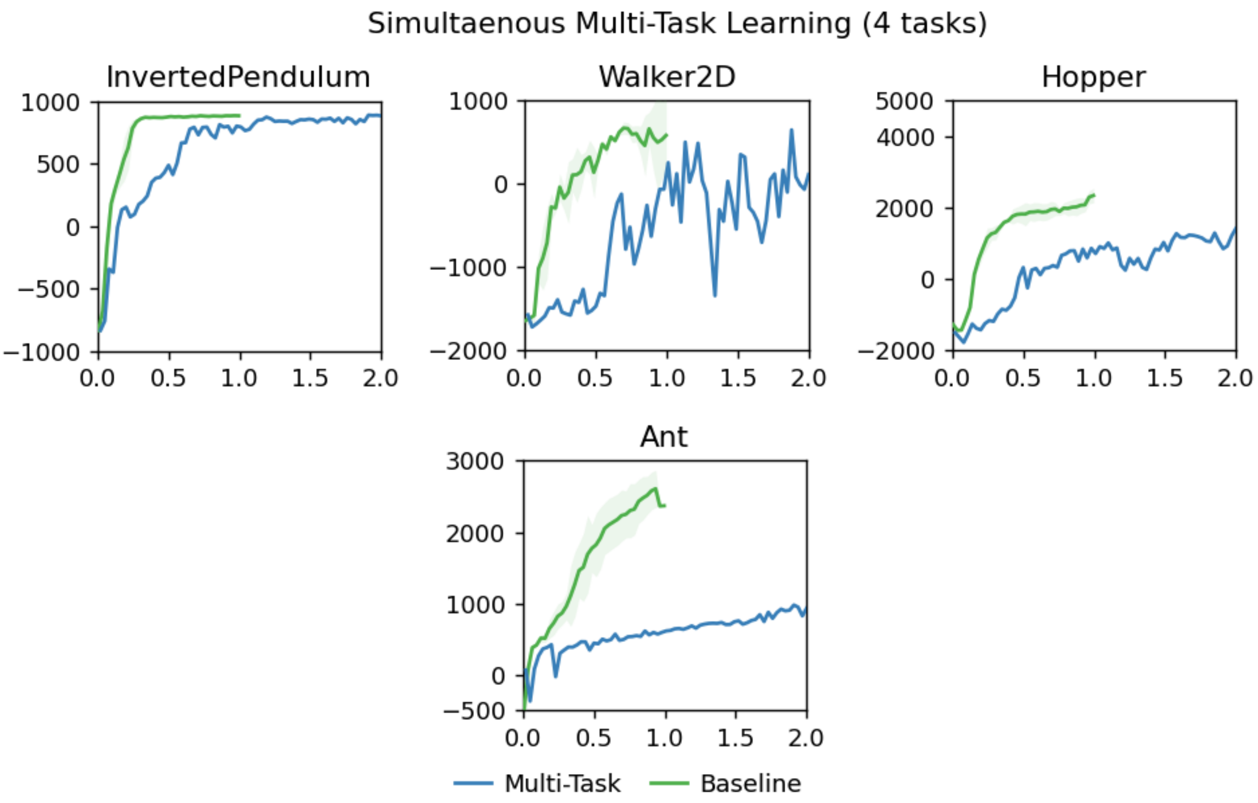


Figure 39: Dreamer simultaneously trained on the InvertedPendulum, Walker2D, Hopper and Ant tasks for 2e6 environment steps, along with the corresponding 1e6 environment steps baselines.