technische universität dortmund

RUHR-UNIVERSITÄT BOCHUM
INSTITUT FÜR NEUROINFORMATIK

Master Thesis

A Comparison of Auxiliary Tasks for Low-Dimensional Representation Learning for Reinforcement Learning

Noah Krystiniak
noah.krystiniak@tu-dortmund.de

2022

Institute for Neural Computation
Faculty of Computer Science
Ruhr-Universität Bochum

Primary Supervisor:	Prof. Dr. Laurenz Wiskott
Secondary Supervisor:	M. Sc. Moritz Lange
Submission:	25th Juli 2022

Abstract

Recently, Deep Reinforcement Learning has proven to be very potent for a growing number of problems. However, there are still some obstacles to overcome in order to extend its applicability to a large number of real-world problems. One of these obstacles is sample efficiency, which limits its applicability in areas such as Robotic and Continuous Control with low-dimensional data. Integrating representation learning with deep reinforcement learning can significantly improve sample efficiency as well as overall performance. A proven tool to achieve this are auxiliary tasks, which represent alternative learning goals to the actual learning goal of reinforcement learning and can be used to learn useful representations. This work compares various auxiliary tasks for representation learning for deep reinforcement learning in low-dimensional continuous control problems in MuJoCo in terms of sample efficiency and maximum returns. It is shown that auxiliary tasks based on observation prediction perform better than those based on reward prediction. Furthermore, it turns out to be more profitable to predict differences between observations than to predict only observations.

Kurzfassung

In jüngster Zeit hat sich Deep Reinforcement Learning für eine wachsende Anzahl an Problemen als sehr potent erwiesen. Jedoch gilt es noch einige Hürden zu überwinden, um die Anwendbarkeit auf eine Großzahl von real-world Problemen auszuweiten. Eine dieser Hürden ist Sample-Effizienz, die vor allem die Anwendbarkeit in den Bereichen Robotic und Continuous Control mit niedrig-dimensionalen Daten beschränkt. Die Integration von Representation Learning in Deep Reinforcement Learning kann sowohl die Sample-Effizienz als auch die Gesamtperformance deutlich verbessern. Hierbei sind Auxiliary Tasks ein probates Mittel, welche alternative Lernziele zu dem eigentlichen Lernziel von Reinforcement Learning darstellen, und zum Lernen von Representationen genutzt werden können. Diese Arbeit vergleicht diverse Auxiliary Tasks für Deep Representation Learning für Reinforcement Learning bei niedrig-dimensionalen Continuous Control Problemen in MuJoCo. Es wird gezeigt, dass Auxiliary Tasks, die auf dem Vorhersagen von Observations basieren, besser performen, also solche, die auf dem Vorhersagen von Rewards basieren. Weiterhin erweist es sich als profitabler, Differenzen zwischen Observations vorherzusagen, als lediglich observations.

Contents

1 Introduction

Reinforcement learning is a set of techniques to find optimal behaviour in a certain environment to reach a certain goal. It uses algorithms that receive an observation of the environment as their input, e.g. an image, and learn to perform an action that maximizes the reward, a scalar whose maximization is linked to reaching the desired goal.

In 2013, Mnih et al. set an important milestone by combining deep learning and reinforcement learning - resulting in deep reinforcement learning - in their DQN algorithm and demonstrating impressive performances in various Atari games [26] [25]. With the introduction of neural networks, reinforcement learning algorithms now were able to learn more complex behaviour, thus extending the research to more and more real world problems. While reinforcement learning is especially well-known for its successful applications in games like backgammon [40], chess [36], Atari [25] and Dota 2 [29] it is becoming increasingly viable for certain real-world control problems like traffic light control [21], smart building energy management [46] and the cooling of data centres [19].

However, the introduction of increased complexity usually goes with an increased need for sample data that the algorithms need to learn. This is a problem, which still severely limits the applicability of deep reinforcement learning in many domains, especially in robotic and continuous control problems [11]. One very promising option to improve sample efficiency as well as performance in this area is the integration of representation learning (more specifically state representation learning). Lesort et al. [20] provide a comprehensive introduction and overview of state representation learning. They distinguish between an observation, which is the raw information data of some sensors, and a corresponding state, which is a compact and usually abstract representation of the observation but still contains the necessary information to solve the actual task of the reinforcement learning problem.

State representation learning means to learn a mapping from observations (and possibly also actions or rewards) to states, while fulfilling one or more learning objectives, also called auxiliary tasks. An example for an auxiliary task is a so-called forward model. Here, observation and action are used to learn a representation, which is able to predict the resulting next observation. Consequently, these representations contain valuable information about the dynamics of the environment, which can accelerate and improve the learning of optimal behavior in deep reinforcement learning.

The usual assumption with state representation learning is that the state has a sometimes significantly lower dimension than the observation. Most settings for which Lesort et al. list representation learning methods work with raw image data as observations, i.e. high dimensional data. Here it is true that, for example, for navigating in an environment not every single pixel is important, and the task-relevant information, such as position and movement of certain objects, can be displayed in a much lower dimensionality.

In their paper "Can Increasing Dimensionality Improve Deep Reinforcement Learning?" Ota et al. [31] introduced the OFENet, a network that learns higher dimensional representations of low dimensional data in continuous control tasks by using a forward model as an auxiliary task. They used these representations with state-of-the-art algorithms like TD3 and SAC on several MuJoCo continuous control tasks and demonstrated significant improvements of sample efficiency and performance.

With the information about the different learning objectives from Lesort et al. the question arises whether other auxiliary tasks can be used to learn representations with the OFENet and how they differ in terms of sample efficiency and performance. In this work, I use - besides the forward model - two additional auxiliary tasks to learn representations with the OFENet, which are used with the state-of-the-art algorithms TD3 and SAC for different MuJoCo continuous control tasks. The first of the two auxiliary tasks is based on the approach of Anderson et al. [2], who used the current state and action to predict the difference between the current state and the next state. I have taken this approach and applied it to observations. The second auxiliary task is a reward prediction model taken from Lesort et al., which takes the current observation and action to predict the next reward. I compare the auxiliary tasks w.r.t. their performance and sample efficiency and vary two hyperparameters of the OFENet, of which one controls how much the OFENet is pretrained and the other determines the dimensionality of the learned representation.

It is shown that the auxiliary task of Anderson et al. which is based on predicting the difference of two successive observations, in comparison across several environments and the algorithms TD3 and SAC, offers the best performance. The prediction of rewards, on the other hand, proves to be inferior in the overall picture. Increased pretraining of the OFENet is able to increase sample efficiency in some cases, whereas no pretraining resulted in the comparably worst performance overall. It was found that a higher dimensionality of the representations for the investigated environment with comparatively high dimensional observation space (Humanoid-v2) is better suited than a lower dimensionality. For the environment with a comparatively low-dimensional observation space (Hopper-v2), on the other hand, the representations with a lower dimensionality performed better.

2 Background

In this chapter I will provide relevant background information about different aspects of reinforcement learning and representation learning.

2.1 Reinforcement Learning

This section is based on the book *Reinforcement Learning: An Introduction* from Sutton and Barto [38] as well as on lecture notes from Wiskott & Schüler [44] (that are based on Sutton and Barto [37]).

2.1.1 Markov Decision Processes (MDPs) and the Policy

Reinforcement learning is about goal-oriented interaction of an agent (the learning algorithm) with an environment. The environment is modelled by a Markov Decision Process (MDP) which is described by a tuple $\{\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, d_0\}$. $\mathcal{S} = \{s_0, s_1, ...\}$ is called state-space and contains all possible states and $\mathcal{A} = \{a_0, a_1, ...\}$ is called action-space and contains all possible actions. The reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ returns a reward for any transition $s_t \to a_t \to s_{t+1}$ while the transition function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ describes the dynamics, i.e. the next state s_{t+1} when taking an action a_t while observing a state s_t. Lastly, d_0 is the initial state of the system. The reward function as well as the transition function can be deterministic or probabilistic depending on the environment that is to be modelled. A defining characteristic of MDPs is the Markov Property which states that the dynamics described by the transition function $\mathcal{P}$ only depend on the current state s_t and action a_t and not on previous states and actions.

The agents behaviour, i.e. what action a_t to take given some state s_t, is defined in a policy $\pi : \mathcal{S} \to \mathcal{A}$. A policy can be deterministic or probabilistic, just like the transition function and reward function. Together the MDP and the policy completely describe the behaviour of a system and are able to generate episodes $\{(s_t, a_t, r_{t+1}, s_{t+1})\}_{t=0,1,...,\infty}$. Figure (2.1) illustrates the interaction between agent and environment as a cyclic process.

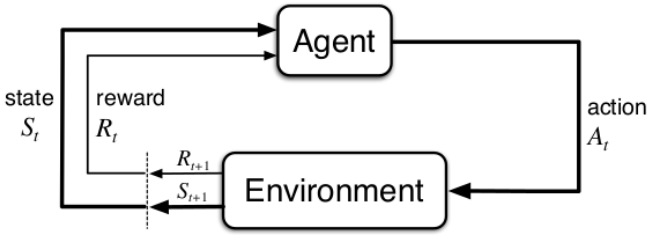


Figure 2.1: The interaction between agent and environment. [38]

Mathematically, the goal of reinforcement learning then is to find a policy that maximizes the expected discounted return

$$\mathbb{E}\left[G_0\right] = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1}\right] \tag{2.1}$$

for any starting state s_0. γ is called the discount rate and if $\gamma < 1$ the policy will prioritize earlier rewards over later ones.

2.1.2 Value Functions

To find an optimal policy it is crucial to be able to not only evaluate any policy but also to improve upon it. For this two important functions are essential, namely the state-value function and the action-value function. The state-value function

$$v_\pi(s) = \mathbb{E}_\pi\left[G_t \mid s_t = s\right] \tag{2.2}$$

evaluates the expected return given a policy π and a starting state s. The action-value function

$$q_\pi(s,a) = \mathbb{E}_\pi\left[G_t \mid s_t = s, a_t = a\right] \tag{2.3}$$

on the other hand evaluates the expected return when taking a certain action a in a given state s and only then following the policy π. It is important to note that

$$v_\pi(s) = \mathbb{E}_{a\sim\pi}\left[q_\pi(s,a)\right]. \tag{2.4}$$

Both functions have a corresponding Bellman Equation, i.e. they fulfill the following recursive conditions:

$$v_\pi(s) = \mathbb{E}_\pi\left[r_{t+1} + \gamma G_{t+1} \mid s_t = s\right] \tag{2.5}$$

$$q_\pi(s,a) = \mathbb{E}_\pi\left[r_{t+1} + \gamma G_{t+1} \mid s_t = s, a_t = a\right] \tag{2.6}$$

Now the goal of maximizing the expected discounted return can also be formulated as finding the optimal action-value function $q^*(s, a)$ because from this we can extract the optimal action

$$a^*(s) = \underset{a}{\operatorname{argmax}}\ q^*(s, a). \tag{2.7}$$

for every state and therefore construct an optimal policy π^*. This formulation can be split up even further into finding the optimal state-value function $v^*_\pi(s)$ (called the *prediction* problem) and finding the optimal policy π^* (called the *control* problem).

For discrete finite MDPs with known dynamics (transition function and reward function) these problems can be solved by using iterative methods. However, for most environments where one wants to apply reinforcement learning the dynamics are unknown and the optimal value functions and the optimal policy can only be approximated by using samples (meaning trajectories) generated from the environment. When trying to get a reliable approximation of π^* one has to ensure that the action-space is explored properly to not miss out on any actions that might be better than the current one. At the same time too much exploration might be harmful because the ultimate goal is to maximize the return which means exploiting the currently best action. This problem is known as the *exploration-exploitation dilemma* and keeping exploration and exploitation in balance is an important aspect for an RL algorithm. For example, the off-policy algorithm TD3 adds noise to actions in the training phase for exploration while omitting it in the evaluation phase for exploitation.

2.1.3 Temporal Difference Learning for Off-Policy Methods

Temporal-Difference Learning is an iterative method and uses single trajectories to incrementally optimize an estimation $V(s_t)$ of the state-value function through a TD learning rule

$$V(s_t) \leftarrow V(s_t) + \alpha \underbrace{[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]}_{\delta_t} \tag{2.8}$$

with a learning rate α and the TD error δ_t. Using an estimate as part of its update is called bootstrapping.

Q-learning was introduced in 1989 by Watkins [42] and is an off-policy TD control algorithm that uses a TD learning rule in an off-policy way to optimize an estimation $Q(S_t, A_t)$ of the action-value function. It is defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{2.9}$$

and serves as a basis for current state-of-the-art RL algorithms such as TD3 or SAC. Off-policy means that the samples used for training the *target policy* (the learning agent)

are generated by a different *behavior policy*. One benefit of off-policy learning is that samples in the form of single transitions (also called *experiences*) can be stored in a replay buffer and used at a later time.

Actor-critic methods introduce a framework to tackle the *prediction problem* and the *control problem* separately. The *actor* represents the actual policy, i.e. it takes a state as its input and selects an action accordingly (→*control problem*). The *critic* represents the state-value or action-value function, i.e. it evaluates the input-state w.r.t. the policy represented by the actor (→*prediction problem*). The driving force for training both the actor and the critic is the TD error. Figure (2.2) illustrates the actor-critic architecture.

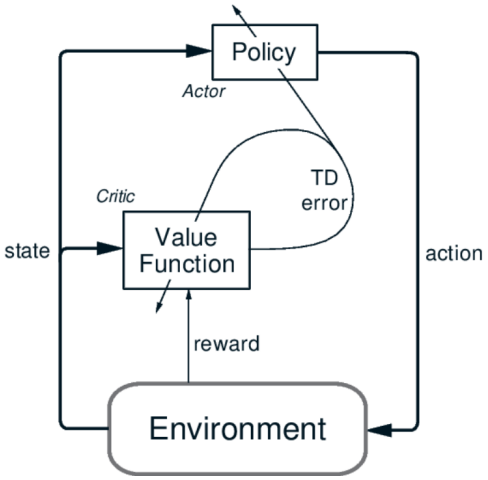


Figure 2.2: The actor-critic architecture. The TD error is the driving force for learning both actor and critic. [37]

2.1.4 TD3

The following section is based on lecture slides from Olivier Sigaud [35] and algorithm documentations for TD3 from SpinningUp [1].

TD3 by Fujimoto et al. [7] is a current state-of-the-art RL algorithm and is an optimized variant of the DDPG algorithm by Lillicrap et al. [22].

In original Q-Learning the current Q-values for any state-action pair are stored in a Q-table. By using (deep) neural networks as function approximators with Q-Learning Deep Q-Networks (DQNs) were introduced by Mnih et al. [26] which made it possible to omit the Q-table and use a neural network as a parametrized representation of the critic $\widehat{Q}_{\phi}^{\pi_\theta}(s_t, a_t)$. The DQN is a neural network that takes a state as its input and has one output neuron for the Q-value for every possible action.

When trying to use DQNs with continuous action spaces the problem of finding $\left(\max_a Q(s_t, a)\right)$ and $\left(\max_a Q(s_{t+1}, a)\right)$ for the TD error arises because it is impossible to have one output neuron for each (s, a)-pair. DDPG solves this problem by training an estimator ($\rightarrow$ actor) of the best action which introduces an actor-critic framework. The actor returns its estimate for the best action given an input-state. Now the critic takes $(s + a)$ as its input and has only one output neuron which returns $\hat{Q}_\phi^{\pi_\theta}(s, a)$. Figure (2.3) illustrates both architectures for DQN and DDPG.

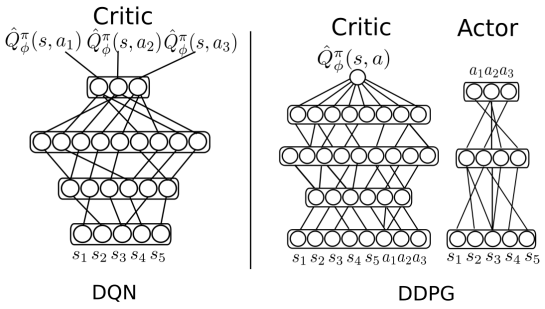


Figure 2.3: Exemplary network architectures for DQN and DDPG. [35]

To reduce instabilities that arise when applying all components of the so-called deadly triad (off-policy learning, bootstrapping, function approximation, see [38]) a separate *target critic* $\hat{Q}'^{\pi_\theta}_{\phi'}$ is used to compute the target

$$y_t = r_{t+1} + \gamma \hat{Q}'^{\pi_\theta}_{\phi'}(s_{t+1}, \pi(s_{t+1})). \tag{2.10}$$

Compared to the actual network this target network changes slowly, resulting in a stable bootstrapping target. This weakens the instabilities introduced by the bootstrapping component and stabilizes the training (see Zhang et al.[48]).

The critic is then updated by minimizing the loss function

$$L = 1/N \sum_t (y_t - \hat{Q}_\phi^{\pi_\theta}(s_t, a_t))^2. \tag{2.11}$$

To elaborate this process, the actual critic returns a Q-value for the pair (s_t, a_t). The target critic takes the resulting s_{t+1} and the corresponding action $a_{t+1} = \pi(s_{t+1})$ estimated by the actor and returns a target Q-value for the pair (s_{t+1}, a_{t+1}). Within the loss function the mean squared error between these two Q-values (over N samples) is computed and its gradient ∇L is used to update the actual critic.

The parameters of the target critic ϕ' slowly follow the parameters of the actual critic ϕ and are updated according to

$$\phi' \leftarrow (1 - \tau)\phi' + \tau\phi \tag{2.12}$$

with the target update coefficient τ.

At last the actor is updated by using the gradient over actions $\nabla_a \hat{Q}_\phi^{\pi_\theta}(s_t, a_t)$ as an error signal to update the actor weights.

A drawback of DDPG is that the Q-function tends to overestimate Q-values. TD3 addresses this and enhances DDPG with three tricks. Firstly, it learns two Q-functions and uses their minimum for computing the target. Secondly, it updates the policy less frequently, e.g. every two Q-function updates. Lastly, it adds noise to the target action so that errors in the Q-function can't be exploited that easily. (For clarity, DDPG adds noise to the sampled actions during training to ensure exploitation but not to the target action.)

2.1.5 SAC

As with TD3, the following section is based on lecture slides from Olivier Sigaud [34] and algorithm documentations for SAC from SpinningUp [1].

The Soft Actor Critic algorithm (SAC) by Haarnoja et al. [9] combines the stability of on-policy methods that use a stochastic policy (like the well-known PPO by Schulman et al. [33]) with the sample efficiency of off-policy methods with a deterministic policy like TD3 [7]. It enhances the original optimization problem of finding a policy that maximizes the expected discounted return by introducing an additional entropy term. The resulting RL problem is

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \underset{(s_t,a_t)\sim\rho_\pi}{\mathbb{E}} \left[\sum_{t=0}^{\infty} \gamma^t \left(r(s_t, a_t) + \alpha H(\pi(\cdot|s_t)) \right) \right] \tag{2.13}$$

with the entropy

$$H(P) = \underset{x\sim P}{\mathbb{E}}[-\log P(x)] \tag{2.14}$$

where x is a random variable (here the actions) with a density function P. $(s_t, a_t) \sim \rho_\pi$ denotes the state-action marginal of the trajectory distribution induced by the policy $\pi(a_t|s_t)$ [9].

The trade-off coefficient α, sometimes called temperature, scales the impact of the entropy term and is an important hyperparameter of SAC. According to the authors, the entropy serves as a tool for exploration and by maximizing the sum of expected discounted return *and* entropy the algorithm is encouraged "to succeed at the task while acting as randomly as possible" [9].

The actor returns parameters for the density function P, usually mean μ_ϕ and standard deviation σ_ϕ of a Gaussian distribution. This introduces stochasticity which increases

exploration. An action is then obtained by sampling from P and applying a squashing function, usually a tanh-function

$$a = \tanh(n) \ , \ n \sim \mathcal{N}(\mu_\phi, \sigma_\phi). \tag{2.15}$$

Figure 2.4 illustrates the network architecture of SAC.

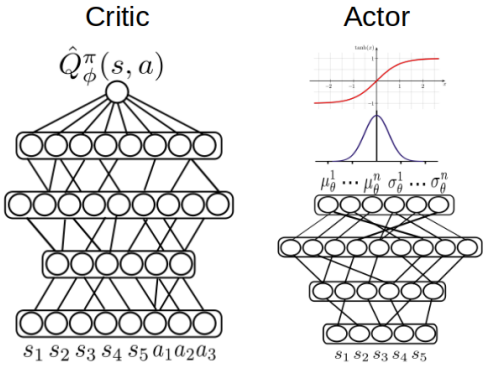


Figure 2.4: The actor-critic architecture for SAC. Note that the actor returns the mean and standard deviation of a gaussian distribution. [34]

Updating the critic is similar to TD3 but with an additional entropy term. The actor update is similar as well but includes stochasticity from the density function, the entropy term and a min-double-Q trick where the minimum of two critics is used for the actor loss as well and not just for the critic loss. The detailed formulas and their derivations are too extensive to include them here but can be found in the original paper (Haarnoja et al., [9]) or at SpinningUp [1].

2.2 Representation Learning

According to Bengio et al. [3] "representation learning [is about] learning representations of the data that make it easier to extract useful information when building classifiers or other predictors". Other definitions describe the term "useful information" more precisely, e.g. Sutton et al. state that part of the problem of representation learning is "[...] to learn inductive biases such that future learning generalizes better and is thus faster". Le-Khac et al. [15] gives a similar definition for the goal of representation learning, i.e. that it should "[capture] and [extract] more abstract and useful concepts that can improve performance on a range of downstream tasks".

In addition to the the definitions above which answer the question of what a representation is supposed to do Lange [18] gives a possible definition of what a representation is, namely "[...] a feature space S_{latent} that is connected through some mapping f: $S_{original} \rightarrow S_{latent}$ to a feature space $S_{original}$." Lange describes the feature space as the space the data lives in. It is called feature space since it is characterised by features of the data. *Learning* a representation then implies to learn a mapping from one feature space to another one in a goal directed manner.

2.2.1 Representation Learning for Reinforcement Learning

Lesort et al. [20] use the term *state representation learning* for a certain kind of representation learning "where learned features are in low dimension, evolve through time, and are influenced by actions of an agent". This description emphasizes the usage of representation learning for control scenarios like reinforcement learning. Lesort et al. further state that the low dimensional characteristic of the learned features helps with problems like the curse of dimensionality and interpretability of a model and improve performance and speed of reinforcement learning algorithms. The reason for this is the assumption that an *observation* o_t of a system, i.e. raw sensory data, usually contains lots of information that is irrelevant for learning a certain behavior, while the *state* s_t is of a much lower dimension and only contains the essential information for the learning task.

2.2.2 Auxiliary Tasks

As mentioned in section 2.2 representation learning usually involves a goal-directed mapping from one feature space to another feature space. This can be done by using an auxiliary task. For example, a neural network can be trained to minimize the loss function of an auxiliary task to learn representations that contain prior knowledge of the systems dynamics and or the task domain [14].

The paper "State Representation Learning for Control: An Overview" by Lesort et al. [20] provides an excellent basis of possible learning objectives and auxiliary tasks that can be used for state representation learning. Therefore, their paper was used as a basis to find auxiliary tasks that will be used for the experiments in this work. In addition to this, Anderson et al. [2] provide another auxiliary task that was used in this work which will be explained in the following. The four auxiliary tasks that were used within the scope of this work are:

The *forward state prediction task* (*fsp*) is the auxiliary task of predicting the next state s_{t+1} from the current state s_t and the current action a_t. This encourages to learn

representations that contain the necessary informations about the dynamics of the system to predict the next state given a current state and action. This is also known as a forward model [20].

The *forward state difference prediction task* (*fsdp*) works in a similar way but uses the difference between the current state and the next state, i.e. $(s_{t+1} - s_t)$, as a prediction target. According to Anderson et al. [2] this avoids learning just an identity map, as consecutive states are often very similar, and more training is required to learn the small variations needed to accurately predict the next state.

The *reward prediction task* (*rwp*) takes the current state s_t and current action a_t and tries to predict the next reward r_{t+1}. According to Munk et al. [28] using the reward as a prediction target enforces the learned representation to only include task relevant information which might not be given for the *fsp* or *fsdp* task.

The *inverse prediction task* (*inv*) tries to predict the action a_t given the two corresponding states s_t and s_{t+1} [20]. This approach is similar to the forward model as it uses the same components to learn representations that contain informations about the dynamics of the system but in a different way. The key difference to the forward model is that the prediction target is an action and not a state.

Figure 2.5 shows the auxiliary tasks with their corresponding components.

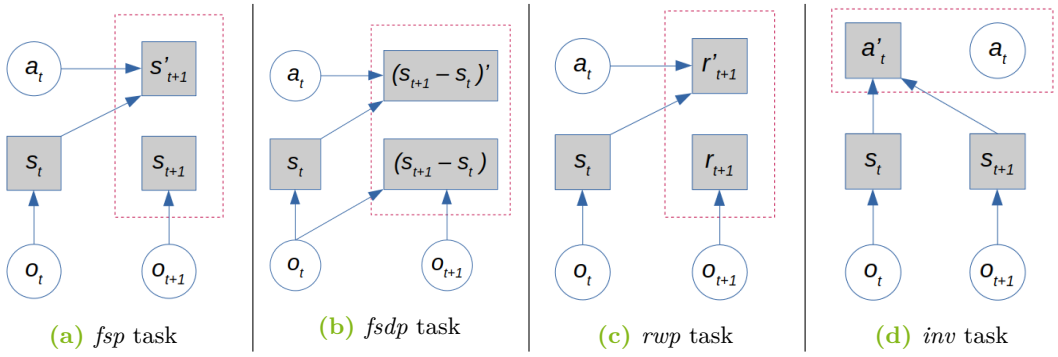


Figure 2.5: The auxiliary tasks *fsp*, *fsdp*, *rwp* and *inv*. White components are input data and grey components are output data. Components in the red dashed rectangle are used for the models loss function where s'_{t+1}, r'_{t+1} and $(s_{t+1} - s_t)'$ are the predicted values and s_{t+1}, r_{t+1} and $(s_{t+1} - s_t)$ are the prediction targets. The graphs are adapted from [20].

3 Related Works

While most representation learning methods deal with high dimensional data such as images, there are significantly fewer approaches for low dimensional data. For example, Lesort et al. [20] give an overview of state representation learning methods, of which the vast majority of methods deal with high dimensional image data. Zhao et al. [49] present a method for state representation learning for low dimensional data, which is based on the idea that the features of adjacent states should be more similar than features of far apart states. This approach seemed promising, so that I originally wanted to conduct further experiments based on their work. However, I was not able to replicate their results from their paper, which is why I did not pursue this approach further.

Ota et al. [31] present another approach by introducing the OFENet, a neural network used for learning higher dimensional representations of low dimensional data for deep reinforcement learning in continuous control tasks. To learn these representations it uses the auxiliary task of predicting the next observation given the current observation and current action which is called a *forward model*. As Jaderberg et al. [13] state, environments offer a lot of possible training signals that can be exploited in addition to reward maximization for reinforcement learning or consecutively for representation learning. Lesort et al. [20] give an overview over various methods and alternative auxiliary tasks for state representation learning, such as an inverse dynamics model and a reward prediction model. This inspired me to try to combine and analyze different auxiliary tasks with the OFENet. Munk et al. [28] introduced their ML-DDPG algorithm that learns state representations before solving the actual RL task. For this it uses two predictive priors, a predictable transition prior that predicts the next state (and not the next observation like Ota et al.) and a predictable reward prior (may also be called reward prediction model). However, these representations had a lower dimension than the original observation and their goal was to learn to extract compact states from noisy observations. Zhang [47] provide a similar approach to Ota et al. and use multiple auxiliary tasks (forward model, inverse dynamics model, reward model) to learn higher dimensional representations but with a different network architecture.

4 Methods

In this chapter I am going to give a detailed description of the Online Feature Extractor Network (OFENet) including how the three auxiliary tasks *forward state prediction* (*fsp*), *forward state difference prediction* (*fsdp*) and *reward prediction* (*rwp*) can be used to learn representations with OFENet. Two important hyperparameters that will the basis of the experiments are explained, namely the *total units* that determine the size of the learned representation and the *pretraining volume* that determines how much the OFENet is pretrained before the actual agent training. Additionally, I will give information on the implementations of TD3 and SAC and the used MuJoCo environments. Observation in the following means the observation of the systems state. For the information and formulas presented in the background section these observations can be used just as well as states.

4.1 Online Feature Extractor Network

The Online Feature Extractor Network (OFENet) was proposed by Ota et al. [31] and learns a higher dimensional representation of its input data by using the auxiliary task of predicting the next observation o_{t+1} from the current observation o_t and the current action a_t ($\rightarrow$*fsp* task). *Online* means it is trained simultaneously with the RL agent that uses the learned representations. The reason for learning a *higher* dimensional representation is that it allows the RL algorithm to learn more complex functions. In other words, the representation is able to contain more information and as a consequence the RL algorithm can (in theory) learn a more optimal behaviour (=reach a higher maximum return) and learn more sample efficiently (=faster). However, it is important to not just increase the dimensionality naively but to use an auxiliary task to learn a *good* higher dimensional representation as Ota et al. showed in the section *5.3 Ablation study* of their work [31].

An actor-critic algorithm for environments with continuous action and observation space (e.g. TD3) takes two different types of inputs, the first is a single observation o_t which is the actor input, the second is an observation-action-pair (o_t, a_t) which is the critic input. OFENet learns the mappings

$$z_{o_t} = \phi_o(o_t) \tag{4.1}$$

$$z_{o_t,a_t} = \phi_{o,a}(z_{o_t}, a_t) \tag{4.2}$$

where functions $\phi_o(o_t)$ and $\phi_{o,a}(z_{o_t}, a_t)$ are represented as neural networks with parameters θ_{ϕ_o} and $\theta_{\phi_{o,a}}$. The representation z_{o_t,a_t} is input to the prediction module f_{pred}, a fully connected layer with parameters θ_{pred} generating the predictions of o_{t+1}. This set of parameters $\theta_{aux} = \{\theta_{pred}, \theta_{\phi_o}, \theta_{\phi_{o,a}}\}$ is then optimized by minimizing the loss function

$$L_{aux} = \mathrm{E}_{(o_t,a_t)\sim p,\pi}\left[||f_{pred}(z_{o_t,a_t}) - o_{t+1}||^2\right] \tag{4.3}$$

where π represents the policy and p the transition function of the environment.

As for the network architecture, OFENet uses a combination of a multilayer perceptron (MLP) and DenseNet (Huang et al. [10]) for the functions $\phi_o(o_t)$ and $\phi_{o,a}(z_{o_t}, a_t)$, called MLP-DenseNet. Each of these functions may also be called a *block* of the OFENet. Essentially, one "normal" layer of the MLP-DenseNet is made up of two different types of layers, a simple fully-connected-layer (fc-layer) with an activation function and a concat-layer. The concat-layer concatenates the output of the latest previous layer with the output of all other previous layers. Figure 4.1 shows an exemplary configuration of an OFENet.

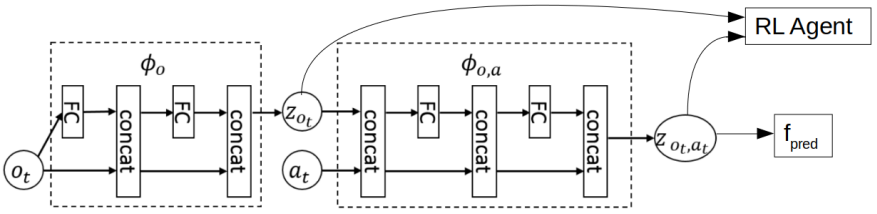


Figure 4.1: Training of OFENet. Adapted from [31].

As a consequence of the concatenation of the concat-layers the number of units of every fc-layer are added up to the overall dimensionality of the resulting representation z_{o_t} or z_{o_t,a_t}. Therefore, the dimensionalities of the representations are given by the dimension of the observation $\dim(o_t)$, the number of fc-layers *num layers* and the number of units per fc-layer *num units* as

$$\dim(z_{o_t}) = \dim(o_t) + num\ layers \times num\ units \tag{4.4}$$

$$\dim(z_{o_t,a_t}) = \dim(z_{o_t}) + dim(a_t) + num\ layers \times num\ units \tag{4.5}$$

with

$$num\ layer \times num\ units = total\ units\ . \tag{4.6}$$

The quantity *total units* is one of the two key hyperparameters for the experiments.

OFENet uses batches of samples for training which are sampled from a replay buffer, usually shared with the RL algorithm. The training of OFENet and training of the RL algorithm happens simultaneously with different sample batches from the same replay buffer. During the training of OFENet the RL algorithm is not trained and vice versa. Figure 4.2 illustrates the training process.

Algorithm 1 Training of OFENet

Initialize parameters $\theta_{aux} = \{\theta_{pred}, \theta_{\phi_o}, \theta_{\phi_{o,a}}\}$
Initialize experience replay buffer $\mathcal{B}$

for each environment step **do**
  $a_t \sim \pi(a_t|o_t)$
  $o_{t+1} \sim p(o_{t+1}|o_t, a_t)$
  $\mathcal{B} \Leftarrow \mathcal{B} \cup \{(o_t, a_t, o_{t+1}, r_{t+1})\}$
  sampling mini-batch $\{o_{t,\mathcal{B}}, a_{t,\mathcal{B}}, o_{t+1,\mathcal{B}}\}$ from $\mathcal{B}$
  $\theta_{aux} \Leftarrow \theta_{aux} - \lambda_{\theta_{aux}} \nabla_{\theta_{aux}} L_{aux}$
  resampling mini-batch $\{o_{t,\mathcal{B}}, a_{t,\mathcal{B}}, o_{t+1,\mathcal{B}}\}$ from $\mathcal{B}$
  $z_o \Leftarrow \phi_o(o_{t,\mathcal{B}})$
  $z_{o,a} \Leftarrow \phi_{o,a}(z_o, a_{t,\mathcal{B}})$
  Update the agent (e.g., SAC) parameters with the representations $z_o, z_{o,a}$
end for

Figure 4.2: Training of OFENet excluding the pretraining. [31]

To stabilize the input to the RL algorithm the OFENet can be pretrained with random samples that are collected before the agent training. The amount of this pretraining - called *pretrain steps* - is the second key hyperparameter for the experiments.

Because the simultaneous training might impact the distribution of the inputs to the RL algorithm (also called *internal covariate shift*) Batch Normalization (see Ioffe & Szegedy [12]) is used to normalize the fc-layers inputs.

For RL algorithms that naturally only use observations as their input and not observation-action pairs (e.g. PPO) the training process of the OFENet is the same but only z_{o_t} would be passed to the algorithm.

4.2 Auxiliary Tasks

To learn higher dimensional representations that are able to improve the performance and sample efficiency of the RL agent it is important to use an auxiliary task when training the OFENet. Ota et al. used the auxiliary task of predicting the next observation o_{t+1} from the current observation o_t and the current action a_t. While this is generally known

as a *forward model* [20] it will be called *forward state prediction task* (or *fsp* task) within the scope of this work, to emphasize the focus on the auxiliary task.

Ota et al. could already show that using OFENet with the *fsp* task generally increased the performance and sample efficiency for the used RL algorithms (TD3, SAC, PPO) but other auxiliary tasks might be more suitable in general, or at least more suitable depending on the task (e.g. dimensionality of the observation/action space).

In section 2.2.2 alternative auxiliary tasks to the *fsp* task were listed, namely the *fsdp* task, *rwp* task and *inv* task. Here are the corresponding loss functions to train the OFENet for all auxiliary tasks used for the experiments:

$$L_{fsp} = \mathrm{E}_{(o_t,a_t)\sim p,\pi}\left[||f_{pred}(z_{o_t,a_t}) - o_{t+1}||^2\right] \tag{4.7}$$

$$L_{fsdp} = \mathrm{E}_{(o_t,a_t)\sim p,\pi}\left[||f_{pred}(z_{o_t,a_t}) - (o_{t+1} - o_t)||^2\right] \tag{4.8}$$

$$L_{rwp} = \mathrm{E}_{(o_t,a_t)\sim p,\pi}\left[||f_{pred}(z_{o_t,a_t}) - r_{t+1}||^2\right] \tag{4.9}$$

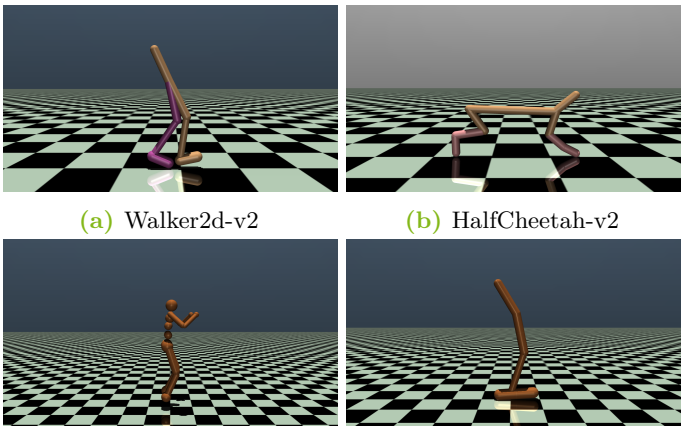
$$L_{inv} = \mathrm{E}_{(o_t,o_{t+1})\sim p,\pi}\left[||f_{pred}(z_{o_t,o_{t+1}}) - a_t||^2\right] \tag{4.10}$$

The main difference between these auxiliary tasks is the prediction target. The *fsp* and *fsdp* tasks uses the next observation or the difference between two successive observations, which is information independent of the actual task of the RL problem. The *inv* task has the action as prediction target, which is also independent of the actual task. In addidtion, the *inv* task also has a different input. The prediction target of the *rwp* task is the reward, which in contrast depends on the actual task and thus contains fundamentally different and task-dependent information compared the other auxiliary tasks. This has the consequence that the representations learned with the *rwp* task are expected to learn fundamentally different information than the other auxiliary tasks. As the representations with *fsp/fsdp* learn task-independent information they could be considered more general, which might have a positive effect on exploration resulting in a possibly improved performance on the actual RL task.

When using representations learned with the *inv* task with TD3 or SAC (or any RL algorithm with a similar structure) it is important to explicitly include the action a_t as the critic input because otherwise it is impossible to compute the gradient of the critic w.r.t. the actions which is necessary for the actor update (see section 2.1.4). With the other tasks this is not a problem as the action is explicitly included in z_{o_t,a_t}. Consequently, the critic input when using represenations by the OFENet learned with the *inv* task is $(z_{o_t,o_{t+1}} + a_t)$ while the actor input is the same as with the other tasks, i.e. z_{o_t}.

4.3 MuJoCo

For the experiments the *MuJoCo* [41] simulation environments - provided by *OpenAI Gym* [6] - were used. Figure 4.3 shows exemplary screenshots of the four *MuJoCo* environments Walker2d-v2, HalfCheetah-v2, Humanoid-v2 and Hopper-v2.



(a) Walker2d-v2 **(b)** HalfCheetah-v2

(c) Humanoid-v2 **(d)** Hopper-v2

Figure 4.3: Exemplary screenshots of the *MuJoCo* simulation environments used for the experiments.

These environments simulate low dimensional continuous control problems where the observation space contains the position (x-, y-, z-coordinate) and velocity of different body parts of the agent. In addition to that, the observation space of Humanoid-v2 contains terms for inertia, constraint forces and external forces on the body [8]. The external forces are excluded from the target and the prediction when training OFENet as they are too hard to predict, according to Ota et al. [31]. The action space contains torques that are applied to the joints. The reward function and transition function are deterministic [27]. For all environments the starting state (i.e. the initial distribution d_0) is a fixed state with small uniform noise added [8].

Table 4.1: *MuJoCo* environments used for the experiments. The observation space contains the position and velocity of different body parts. Humanoid-v2 has additional terms for inertia and constraint forces as well as external forces on the body. The external forces (in brackets) are excluded from the target and prediction when training OFENet. The action space contains torques that are applied to the joints.

	observation space	**action space**
Walker2d-v2	17-D	6-D
Humanoid-v2	292(+84)-D	17-D
HalfCheetah-v2	17-D	6-D
Hopper-v2	11-D	3-D

The ultimate goal for all environments is to learn to control the joints so that the agent gets as far as possible in a given amount of timesteps. Taking actions that are too large is penalized in the reward function which encourages the agent to learn a more controlled and continuous behaviour [8].

5 Experiments

For the sake of readability, instead of saying "the performance of the RL algorithm using the representations learned by OFENet with the *fsp*/*fsdp*/*rwp* task" I will simply write "the performance of *fsp*/*fsdp*/*rwp*" or "the performance of the auxiliary task *fsp*/*fsdp*/*rwp*".

The following experiments try to answer the following questions:

- Can I reproduce the results from Ota et al. [31] and implement the auxiliary tasks *fsdp* and *rwp* in addition to *fsp* used by Ota?
- Can I then transfer the code to PyTorch in order combine it with PyTorch-versions of TD3 and SAC?
- How do the three auxiliary tasks *fsp*, *fsdp* and *rwp* compare in terms of sample efficiency and maximum return?
- How does varying the amount of pretraining of the OFENet affect sample efficiency and maximum return?
- How does varying the amount of *total units* of the representations z_{o_t} and z_{o_t,a_t} affect sample efficiency and maximum return?
- How to explain differences and similarities of the performance between the auxiliary tasks?

I will call the experiments required to answer the first two questions *preparatory experiments* as they do not contribute directly to the aim of this work - comparing and analyzing auxiliary tasks for representation learning for reinforcement learning. However, they were still necessary to get to the point where the *comparative experiments*, which contain the essential experiments of this work, could be done. The *comparative experiments* then try to answer the last four questions. Additionally, the preparatory experiments served as an excellent excercise to get hands-on experience with representation learning and reinforcement learning and feel confident enough to execute the main experiments. Another benefit of these preparatory experiments is that they provide additional validation for the OFENet by Ota et al. [31] by decoupling it from the authors code, transfering it from TensorFlow to PyTorch and combining it with other (non-author) implementations of TD3 and SAC.

The used implementations for TD3 and SAC in combination with OFENet can be found at my GitHub [17]. The implementation for TD3 is based on the original code by Fujimoto et al. [7]. For SAC the implementation by Yarats et al. [45] was used as a basis. The original code for the OFENet by Ota et al. can be found in their paper [31]. While the original OFENet was implemented with TensorFlow [24] I transfered it to PyTorch [32] within the scope of my work.

5.1 Preparatory Experiments

In this section, some results of Ota et al. are reproduced for validation. Subsequently, the auxiliary tasks *fsdp*, *rwp* and *inv* are implemented and their performance is examined. Finally, the OFENet with the various auxiliary tasks is implemented in PyTorch to use it with PyTorch versions of TD3 and SAC to be used for the following *comparative experiments*.

5.1.1 Reproducing Results from Ota et al.

The original code provided by Ota et al. was used to reproduce the results for all algorithms for the MuJoCo-environment Walker2d-v2. Figure 5.1 shows the comparison of the resulting plots.

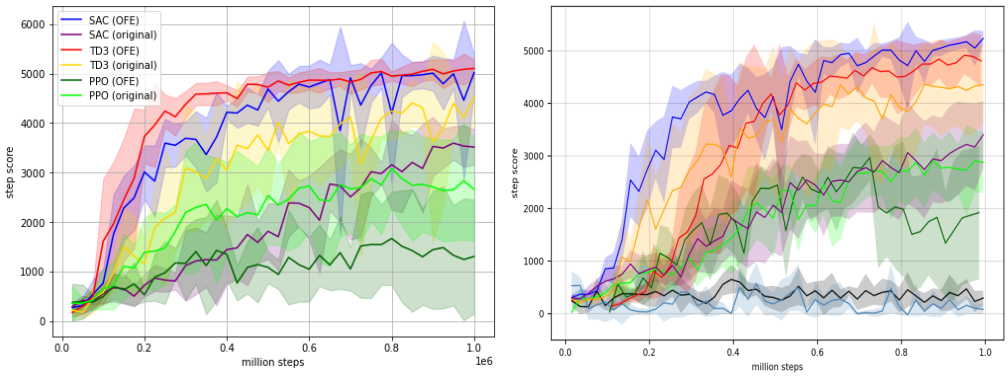


Figure 5.1: My experiments (a) using the code from Ota et al. to validate their results (b) for the MuJoCo-environment Walker2d-v2. Represented are mean and corresponding standard deviation over five different random seeds. Besides TD3 (OFE) and PPO (OFE) my results are in good agreement with the ones by Ota et al. [31]. The blue and black curve in (b) are other algorithms used by Ota et al. which are not relevant for this work.

The hyperparameters used for OFENet and the algorithms are identical to the ones Ota et al. used except for the amount of collected random samples and therefore the pretraining volume of the OFENet. For PPO (OFE) and TD3 (OFE) Ota et al. collected 100k random samples before starting the agent training and pretrained OFENet with these samples while I used 10k random samples for all algorithms as this was the default value in the code provided by Ota et al..

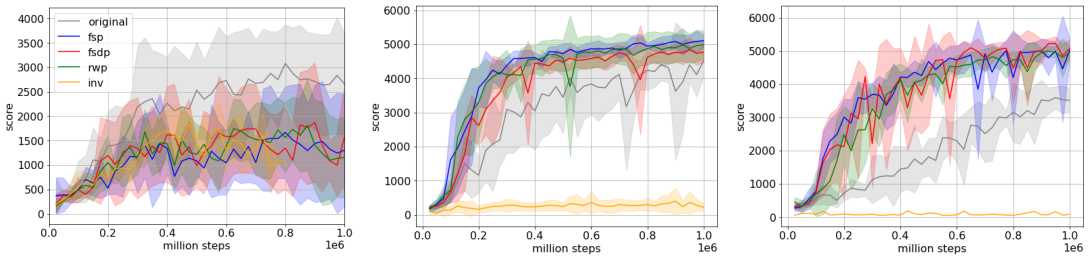
TD3 (OFE) learned much faster in my experiments and converged earlier to a slightly higher end score, compared to the results from Ota et al.. As within the *comparative experiments* in section 5.2.4 in Figure 5.17d it could be shown that more pretraining can possibly slow down the learning, at least for the Hopper-v2 environment. Therefore, the difference in pretraining may be partly responsible for the different performances. Additionally, the probabilistic nature of reinforcement learning algorithms and the (random) choice of seeds may also partly explain differences in performance.

For PPO (OFE) we have the same difference of random samples collected and used to pretrain the OFENet but with a different result. With 10k collected random samples (see (a)) PPO on average does not reach the 2000-score-mark while with 100k collected random samples (see (b)) it nearly surpassed the 3000-score-mark even though it declined afterwards. This is in contrast to the interpretation for TD3 (OFE) without an intuitive explanation except for possibly variations due to the stochasticity of PPO and random seeds.

Besides these two aspects the other performances are in good agreement. The blue and black curve in 5.1 (b) are variations of the ML-DDPG algorithm (see Munk et al. [28]) in combination with SAC examined by Ota et al. but are not relevant for the rest of this work.

5.1.2 Implementing the Auxiliary Tasks *fsdp*, *rwp* and *inv*

As a next step the auxiliary tasks *fsdp*, *rwp* and *inv* are implemented - in addition to the already used *fsp* task - into the Code by Ota et al. by changing the loss function of the OFENet, as described in chapter 4. Figure 5.2 shows the results for PPO, TD3 and SAC for the MuJoCo environment Walker2d-v2.



(a) PPO (b) TD3 (c) SAC

Figure 5.2: Results for PPO, TD3 and SAC with OFENet using the auxiliary tasks *fsp*, *fsdp*, *rwp* and *inv* for the MuJoCo environment Walker2d-v2, implemented into the code provided by Ota et al.. Represented is the average over five different random seeds with the corresponding standard deviation. For TD3 and SAC the *fsp*, *fsdp* and *rwp* tasks perform similar and better than the original versions (without OFENet). For PPO they perform worse than PPO (original). The *inv* task could not learn anything for TD3 and SAC and performed similar to the other tasks for PPO.

The performance of the auxiliary tasks *fsp*, *fsdp* and *rwp* is similar for all three RL algorithms. Using the *inv* task with TD3 and SAC results in a near-zero performance, so the agent does not learn anything at all. However, for PPO the *inv* task performs similar to the other tasks. The reasons for this will be further examined in the next subsection.

TD3 and SAC significantly profit from using OFENet with the *fsp*, *fsdp* and *rwp* task. For PPO using the representations learned by OFENet seems to harm the learning process as the maximum score is significantly below PPO (original) for all auxiliary tasks. One major difference between PPO and SAC/TD3 is that PPO is an on-policy algorithm and does not use Q-learning methods, i.e. it does not approximate a Q-function. Therefore, it only uses z_{o_t} and not z_{o_t,a_t} (or $z_{o_t,o_{t+1}}$ for the *inv* task) which could partly explain why PPO can not use OFENet as efficiently as TD3 and SAC. However, because PPO with OFENet performs worse than PPO (original) only TD3 and SAC are used for the remaining experiments.

5.1.3 Problems with the *inv* Task

As the *inv* task performs similar to the other tasks for the on-policy algorithm PPO the problem is suspected to be linked to some aspect regarding off-policy algorithms. To investigate the reason of why the representations learned with the *inv* task are unusable for TD3 and SAC and prohibit any learning, further experiments are done with TD3. In addition to using z_{o_t} as the actor-input and $(z_{o_t,o_{t+1}} + a_t)$ for the critic-input the following combinations of actor- and critic-input of the *inv* task are examined:

- Version 0: actor-input = z_{o_t} , critic-input = $(z_{o_t, o_{t+1}+a_t})$
- Version 1: actor-input = z_{o_t} , critic-input = $(z_{o_t} + a_t)$
- Version 2: actor-input = z_{o_t} , critic-input = $(z_{o_{t+1}} + a_t)$
- Version 3: actor-input = o_t , critic-input = $(z_{o_t} + a_t)$

The versions including o_{t+1} in the critic-input result in near-zero performance and completely prohibit any learning while the other versions lead to an acceptable performance, e.g. reaching the 10.000-return-mark for HalfCheetah-v2 in 1M steps. It appears that the presence of o_{t+1} within the representation used for the critic-input causes problems which prevent the learning of any goal-oriented behaviour.

A closer look at the actor critic structure of TD3 reveals, that the presence of o_{t+1} must inevitably lead to problems. The observation o_{t+1} results from the input of (o_t, a_t) into the environment and thus depends on a_t. However, since the environment is not differentiable, the gradient of the critic w.r.t. the action $\nabla_a \hat{Q}_\phi^{\pi_\theta}(o_t, a_t, o_{t+1})$, which is needed for the actor update, cannot be calculated correctly. This explains why including o_{t+1} in the critic input hinders the learning of any useful behaviour with an actor-critic algorithm whose updates are based on differentiating a Q-function. Figure 5.3 illustrates this problem.

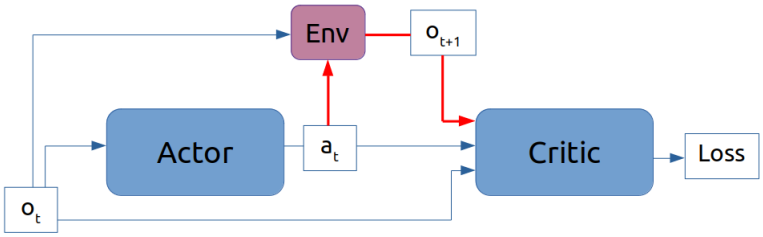


Figure 5.3: Illustrating the problem of using the inverse model with an actor-critic algorithm whose updates are based on differentiating a Q-function. When differentiating the critic loss w.r.t. the action - needed for the actor update - the gradient along the red lines can not be calculated since the environment is not differentiable. This results in incorrect actor updates and hinders learning.

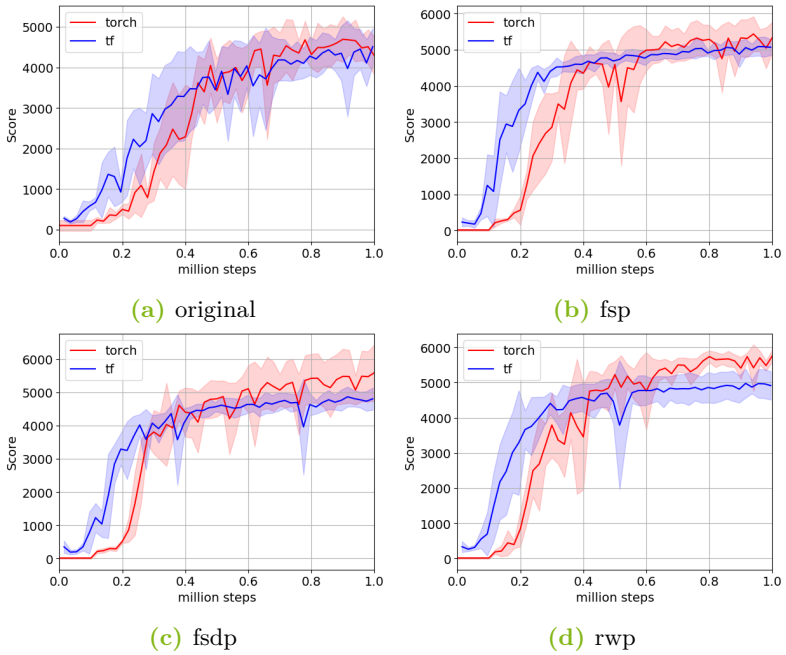
Consequently, using representations learned by OFENet with the *inv* task leads to the same problem. Additionally, this explains why the inverse model works comparatively well with PPO. PPO also has an actor-critic structure, but it only uses the representation z_{o_t} since no Q-values are calculated, so the problem described above does not occur.

5.1.4 Transferring OFENet to PyTorch

To be more independent of the code from Ota et al. and because several discussions and articles (e.g. [43] [23] [5]) suggest that PyTorch is more accessible for research and faster than TensorFlow I implemented OFENet in PyTorch and used it with PyTorch-versions of TD3 and SAC.

In Figure 5.4 is a comparison of TD3 with OFENet in PyTorch (torch) and TensorFlow (tf) for the three auxiliary tasks *fsp*, *fsdp* and *rwp* using the MuJoCo environment Walker2d-v2. These experiments were done to validate the functionality of the PyTorch-version of OFENet.

For the PyTorch-version 100k random samples were collected before the agent training and used to pretrain OFENet, for the TensorFlow-version only 10k. OFENet (torch) used a learning rate of 0.001 (default value for Adam optimizer [16]) and OFENet (tf) 0.0003 (learning rate used by Ota et al.). These differing values happened to be those used during development of the PyTorch version. In fact, the difference between values only became apparent later when comparing these particular details of the two code versions. However, despite those differences the results of initial runs were already sufficient to verify that the PyTorch implementation behaves similar to the TensorFlow version. So in the interest of time experiments were not repeated with the exact same parameters for both versions.



(a) original (b) fsp

(c) fsdp (d) rwp

Figure 5.4: Comparison of PyTorch and TensorFlow Versions of TD3 with the auxiliary tasks *fsp*, *fsdp*, *rwp* and original TD3 for the MuJoCo environment Walker2d-v2. The solid line represents the average over 5 different random seeds with the corresponding standard deviation. Because of different learning rates and different amounts of random samples collected and used for pretraining the OFENet the results are not expected to be identical but still validate the functionality of the PyTorch-version of OFENet.

5.2 Comparative Experiments

In the first three subsections 5.2.1 Humanoid-v2, 5.2.2 HalfCheetah-v2 and 5.2.3 Hopper-v2 I am going to compare the performance of the auxiliary tasks *fsp*, *fsdp* and *rwp* with different values for *pretrain steps* and *total units* for the particular MuJoCo environment. To get an overview of the general performance between the auxiliary tasks, the max return and the "% steps of TD3/SAC (original)" are compared in a scatter plot. "% steps of TD3/SAC (original)" describes, how many steps (or samples) of TD3/SAC (original, meaning without OFENet) are needed to reach the max return of TD3/SAC (original). So this value represents how many samples can be saved with OFENet, compared to the original algorithm, and represents the sample efficiency in the context of this work.

The following two subsections 5.2.4 *Pretrain Steps* and 5.2.5 *Total Units* examine the impact of the hyperparameters for each auxiliary task in particular. Varying the amount

of *pretrain steps* corresponds to varying the amount of random samples collected before the agent training and using these samples to pretrain the OFENet. At 10k/100k *pretrain steps* 10k/100k random samples are collected, OFENet is pretrained with these 10k/100k samples and then the agent training starts. This means that for 10k *pretrain steps* at the 100k-step-mark OFENet has been trained with 10k random samples and 90k agent-generated samples. And, consequently, for 100k *pretrain steps* at the 100k-step-mark OFENet has been trained with 100k random samples only. To explicitly inspect the impact of the varying pretraining amounts on the learned representations, the plots in section 5.2.4 *Pretrain Steps* were aligned such that the pretraining of OFENet is finished at the 0-step-mark.

Different values for *total units* result in a different size (i.e. dimensionality) of the learned representations z_{o_t} and z_{o_t,a_t} as mentioned in the equations 4.4 and 4.5. A larger dimensionality of the representations can also be seen as a larger complexity of the representations or a potentially increased expressiveness.

In the last two subsections 5.2.4 *Pretrain Steps* and 5.2.5 *Total Units* I will use the same runs from the first three subsections but instead of comparing the auxiliary tasks with each other, I will examine the effect of the varied *pretrain steps* and *total units* within each auxiliary task. For the plots with the varied *pretrain steps* the curves have been aligned so that the agent training always starts at step 0. After these effects have been pointed out for each auxiliary task individually they can be compared with each other.

The used values for the varied hyperparameters are *total units* $= \{48, 240, 960\}$ and *pretrain steps* $= \{0, 10000, 100000\}$. The *total units* are constrained by the number of layers and number of units per layer of the OFENet and equation 4.6. Using the same values as Ota et al. [31] the *total units* are constrained to be multiples of 48. Therefore, 48 covers the smallest possible value, 240 is the default value used by Ota et al. and 960 was chosen to represent a high value. For the experiments that vary the *total units* the *pretrain steps* are fixed to 10k as this is the default value used by Ota et al.. When varying the *pretrain steps* the *total units* are fixed to 240, again the default value by Ota et al.. The runs with HalfCheetah-v2 and Humanoid-v2 were trained with 3 million steps/samples and for Hopper-v2 1 million steps/samples were used, the same as Ota et al.. All other hyperparameters for TD3, SAC and OFENet can be found in the appendix. Every run was executed five times with five different random seeds. The seed determines the pseudorandom processes in the code, for example the random samples collected in the beginning or the noise that is added to the actions of TD3.

I want to note that many papers either use the standard deviation as their dispersion measure for their plots (e.g. Ota et al. [31], Fujimoto et al. [7]) or do not indicate the type of dispersion measure at all (e.g. Haarnoja et al. [9], Schulman et al. [33]). From my experience, taking the min/max-range as the dispersion measure better represents the characteristic and the behaviour of the data collected within the experiments. One reason

against using the standard deviation is that using the average of five runs to represent the performance for a given configuration (environment, RL algorithm, hyperparameters, ...) might be not enough for the standard deviation to be a meaningful dispersion measure. Therefore, the min/max-range was used as a dispersion measure for the remaining experiments.

The experiments were done with the help of GNU parallel [39] for job execution and Weights & Biases [4] for experiment tracking and visualization.

5.2.1 Humanoid-v2

In this section the values for *pretrain steps* and *total units* were varied for the MuJoCo task Humanoid-v2. The Humanoid-v2 environment is the most complex one of the used environments with an observation/action space of (376/17). Therefore, here we have the largest difference of the dimensionality of the prediction target for the OFENet, 292 for the *fsp* and *fsdp* task (see section 4.3 why not 376) and 1 for the *rwp* task.

Figure 5.5 shows the results for using the auxiliary tasks *fsp*, *fsdp* and *rwp* to learn representations with the OFENet for TD3/SAC with various amounts of *pretrain steps*.

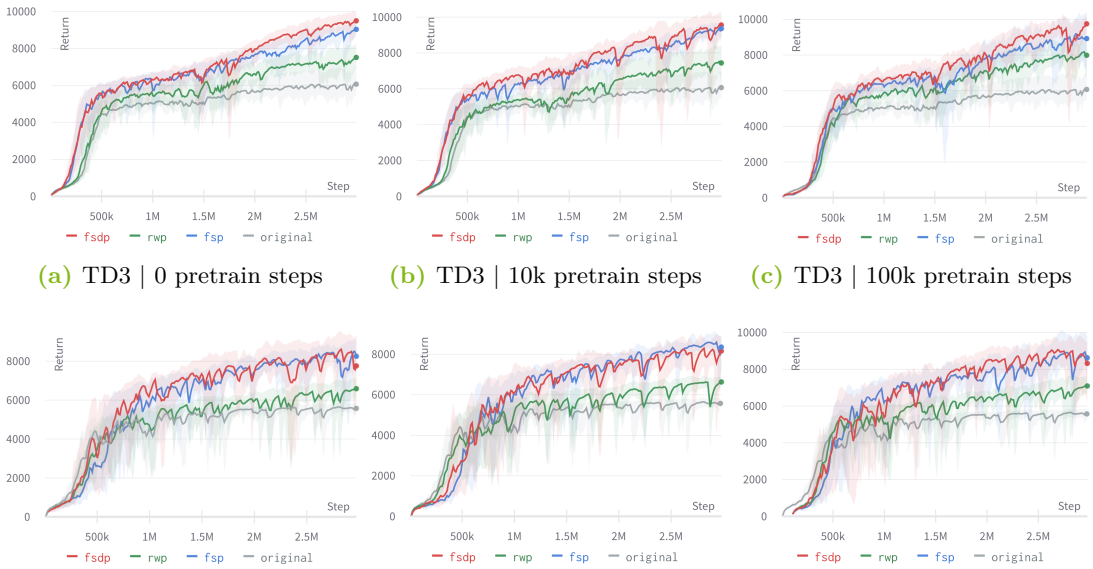
For 0 *pretrain steps* with TD3 the *fsp* and *fsdp* task learn significantly faster then the *rwp* task and TD3 (original). Additionally, their end returns are significantly higher with the *fsdp* task slightly surpassing the *fsp* task after 2 million steps. The *rwp* task learns just as fast as TD3 (original) and only slowly builds up a return advantage towards the end of about 7500 vs. 6000. With SAC we have a similar picture where *fsp* and *fsdp* notably outperform *rwp* although the initial learning speed is the same for all three auxiliary tasks. Additionally, *fsdp* has no advantage over *fsp* and their performance is similar.

Increasing the *pretrain steps* to 10k leads to the *fsdp* task performing a little bit unstable with TD3 but still having the advantage over the *fsp* task. The *rwp* tasks now interestingly performs slightly worse up until about 1.5 million steps but performing equally well from then on compared to 0 *pretrain steps*. For SAC, again, there is no difference between *fsp* and *fsdp* and the pretraining seems to rather harm their performance towards the end than to improve it. The performance of *rwp* is unchanged.

Further increasing the *pretrain steps* to 100k slows down the initial training for *fsp* and *fsdp* with TD3 so that the initial performance is now on a par with *rwp* and TD3 (original). The slower training can be explained by the agent training starting later. *fsp* and *fsdp* quickly surpass *rwp* and build up their advantage towards the end. The *rwp* task seems to profit from the increased pretraining as it now learns faster and is able to reach higher scores towards the end, surpassing the 8000-return-mark on average. The other two tasks are still superior with *fsdp* seemingly gaining a slight performance

increase towards the end compared to 10k *pretrain steps*. The overall performance of *fsp* remains unchanged. With SAC the performance here is very compared to 0 *pretrain steps*. However, this time *fsdp* has slightly higher returns in the end than *fsp*.

As for the effect of the pretraining, only *rwp* seems to profit here at 100k *pretrain steps* for TD3 in terms of learning speed and overall return reached compared to the lower values. Observing the curve of *rwp* and TD3 (original) for 0 and 10k *pretrain steps*, it becomes apparent that *rwp* only really parts from TD3 (original) after about 1.5M steps. This indicates that only then the learned representations start to provide substantial value to the RL algorithm. Using 100k *pretrain steps* seems to accelerate the learning of more valuable representations so that the RL algorithm profits from them much faster. A possible explanation for this might be that with the *rwp* task the representations learn information that is relevant for the actual RL task instead of learning about the dynamics of the system, as with the *fsp* or *fsdp* task. Increasing the pretrain amount consequently increases the task relevant information contained in the representation which would then accelerate the initial learning.



(a) TD3 | 0 pretrain steps **(b)** TD3 | 10k pretrain steps **(c)** TD3 | 100k pretrain steps

(d) SAC | 0 pretrain steps **(e)** SAC | 10k pretrain steps **(f)** SAC | 100k pretrain steps

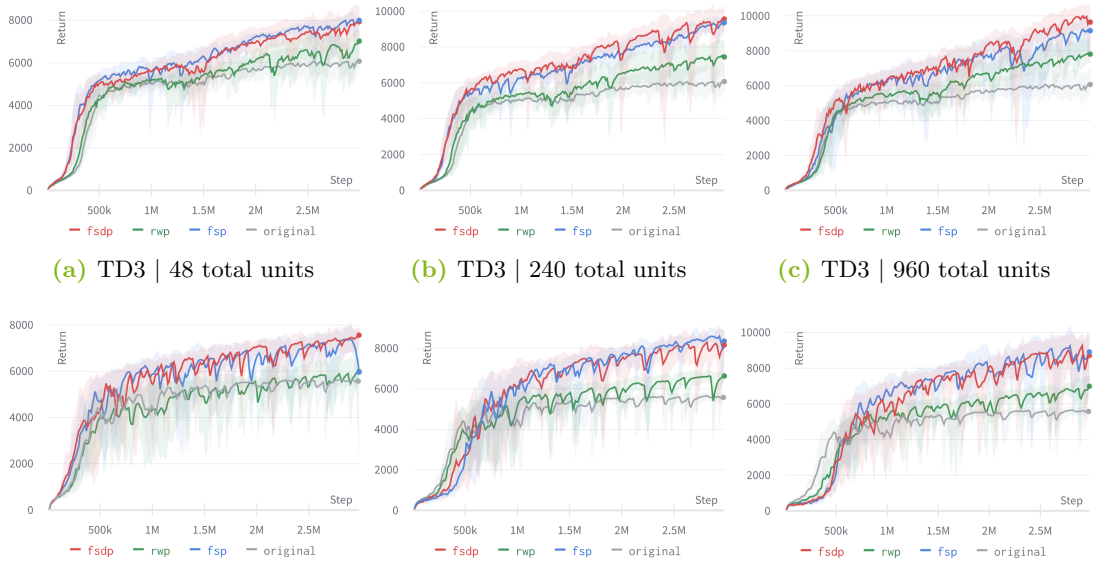
Figure 5.5: Comparison of using the auxiliary tasks *fsp*, *fsdp* and *rwp* to learn representations with OFENet for TD3/SAC on the Humanoid-v2 task. The OFENet has been pretrained with 0, 10k or 100k random samples before the agent training. TD3/SAC (original) shows the performance of using the regular algorithm without OFENet where 10k random samples are collected before the agent training as well. The solid lines and the shaded area represent the average return and the corresponding min/max for five different random seeds.

In Figure 5.6 are the results for using the auxiliary tasks *fsp*, *fsdp* and *rwp* to learn representations with OFENet for TD3/SAC with varying values for *total units*.

For 48 *total units* the *fsp* and *fsdp* task perform very similar for TD3 and SAC with *fsp* having a rather insignificant advantage for TD3. Both learn notably faster and reach higher scores than *rwp* and TD3/SAC (original). The *rwp* task provides a small advantage over TD3/SAC (original) only towards the end.

For TD3, increasing to 240 *total units* results in a substantial performance gain for *fsp* and *fsdp* while *rwp* only receives a minor increase in the second half of its performance. Here, *fsdp* now has a slight advantage over *fsp*. With SAC the performance of *fsp* and *fsdp* shows a slightly decreased initial learning speed but does not change otherwise while *rwp* improves slightly towards the end.

Further increasing to 960 *total units* with TD3 slows the initial training for *fsp* and *fsdp* so that now they perform closer to *rwp* and TD3 (original). The average score towards the end receives a slight increase for *fsdp* and *rwp* while the performance of *fsp* is pretty much unchanged. However, *rwp* still performs significantly worse than the other two tasks and its advantage over TD3 (original) only shows after about 1.5M steps. For SAC the initial learning speed of *fsp* and *fsdp* decreased as well while they both reach significantly higher returns. The returns of *rwp* improved slightly as well.



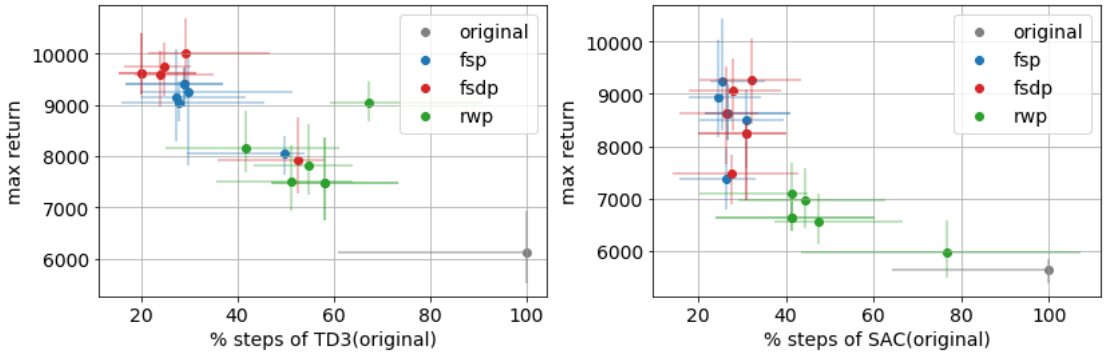
(a) TD3 | 48 total units **(b)** TD3 | 240 total units **(c)** TD3 | 960 total units

(d) SAC | 48 total units **(e)** SAC | 240 total units **(f)** SAC | 960 total units

Figure 5.6: Comparison of using the auxiliary tasks *fsp*, *fsdp* and *rwp* to learn representations with OFENet for TD3/SAC on the Humanoid-v2 task. The representations have a dimensionality corresponding to 48, 240 or 960 *total units* (see equations 4.4 and 4.5 in section 4.1). TD3/SAC (original) shows the performance of using the regular algorithm without OFENet. The solid lines and the shaded area represent the average return and the corresponding min/max for five different random seeds.

To combine the results of both experiments with different *pretrain steps* and *total units*, Figure 5.7 compares the max return and *% steps of TD3/SAC (original)*, which is a measure of the sample efficiency, With TD3 the *fsdp* clearly outperforms in both regards, closely followed by *fsp*. They both need on average 70-80% less samples than TD3 (original) to reach the max return of TD3 (original). *rwp* performs comparably poor but still saves about 40-60% samples on average. With SAC the *fsp* and *fsdp* are not as separated as with TD3 and perform similar overall, saving about 70-80% samples compared to SAC (original). *rwp* with SAC is a little bit more sample efficient compared to TD3.

The overall better performance of the *fsdp* task is in agreement with the assumption made by Anderson et al. in section 5.2. The worse performance of *rwp* is in agreement with the assumption stated in the beginning of this section, that the difference in the dimensionality of the prediction target (1 vs. 292) may result in less valuable representations learned with the *rwp* task compared to the other two, resulting in a worse performance.



(a) TD3 | pretrain steps & total units

(b) SAC | pretrain steps & total units

Figure 5.7: Comparison of max return and *% steps of TD3/SAC (original)* of all runs in this section with varying *pretrain steps* and *total units* for Humanoid-v2. % steps of TD3/SAC (original) describes how many steps (or samples) of TD3/SAC (original) are needed to reach the max return of TD3/SAC (original). Represented are averages over five runs with different random seeds and corresponding min/max values.

5.2.2 HalfCheetah-v2

In this section the *pretrain steps* and *total units* were varied for the MuJoCo environment HalfCheetah-v2. This environment has an observation/action space of dimensionality (17/6) and, therefore, contains significantly fewer dimensions than Humanoid-v2 (376/17) but a little bit more than Hopper-v2 (11/3) which will be discussed in the next section.

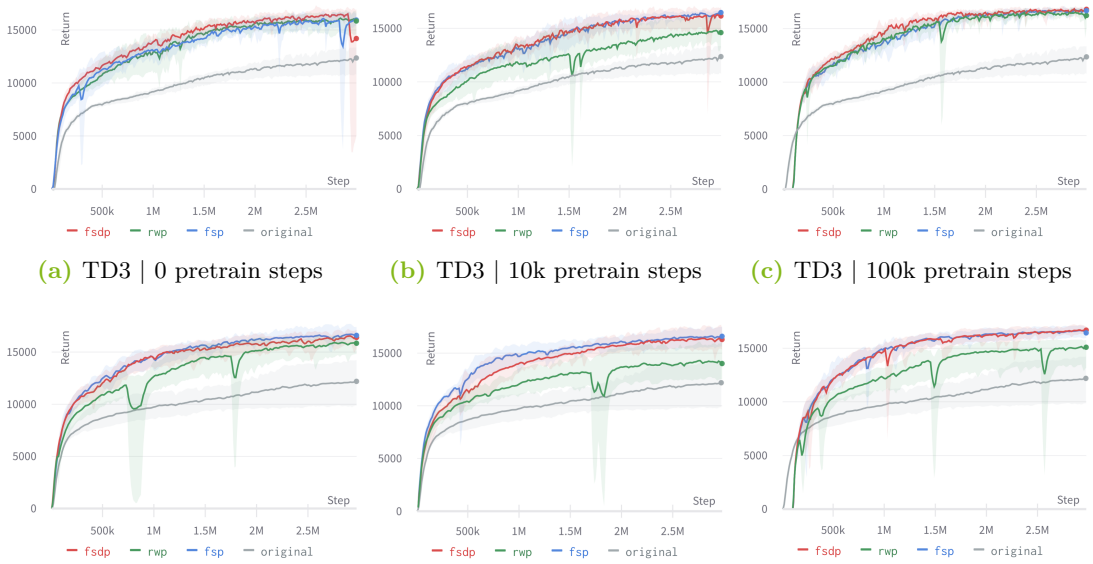
In Figure 5.8 are the results for using the auxiliary tasks to learn representations with OFENet for TD3/SAC with various amounts of for *pretrain steps*.

For 0 *pretrain steps* all three auxiliary tasks perform very similar with *fsdp* having a slight lead in terms of maximum return and learning speed for TD3. With SAC *rwp* performs notably worse and *fsp* and *fsdp* are now on a par.

Increasing the *pretrain steps* to 10k causes *fsp* to catch up to *fsdp* so that they now perform equally good with TD3. With SAC the *fsp* task has a significant advantage over *fsdp*. For *rwp* however the performance and learning speed dropped significantly for both TD3 and SAC. There is no intuitive reasoning for this, especially considering that the performance is significantly better for 0 and 100k *pretrain steps* for TD3 and SAC.

For 100k *pretrain steps* all tasks perform quite similar with TD3. As with 0 *pretrain steps* *fsdp* has a slight advantage over the other two auxiliary tasks up until 1.5M steps. *rwp* with SAC, however, performs notably worse than the other two.

All in all varying the *pretrain steps* does not reveal any meaningful differences between the auxiliary tasks except for the odd behaviour of *rwp* at 10k *pretrain steps* and a very minor lead of *fsdp* at 0 and 100k *pretrain steps* with TD3. However, it is noteworthy that *rwp* with SAC overall performed significantly worse compared to TD3. As this did not occur with Humanoid-v2 the difference of TD3 and SAC, *rwp* does not perform worse with SAC in general. Either, these performances are just randomly much worse than with TD3, or there is a more complex interaction between algorithm, environment and auxiliary task resulting in the worse performance.



(a) TD3 | 0 pretrain steps **(b)** TD3 | 10k pretrain steps **(c)** TD3 | 100k pretrain steps

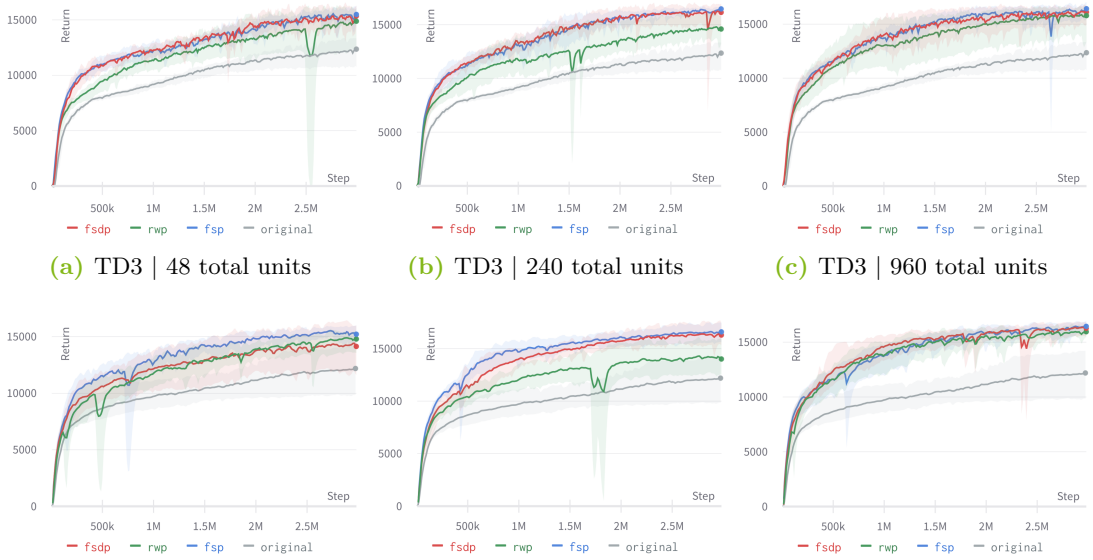
(d) SAC | 0 pretrain steps **(e)** SAC | 10k pretrain steps **(f)** SAC | 100k pretrain steps

Figure 5.8: Comparison of using the auxiliary tasks *fsp*, *fsdp* and *rwp* to learn representations with OFENet for TD3/SAC on the HalfCheetah-v2 task. The OFENet has been pretrained with 0, 10k or 100k random samples before the agent training. TD3/SAC (original) shows the performance of using the regular algorithm without OFENet where 10k random samples are collected before the agent training as well. The solid lines and the shaded area represent the average return and the corresponding min/max for five different random seeds.

Figure 5.8 shows the results for using the auxiliary tasks to learn representations with OFENet for TD3/SAC with varying values of *total units*.

For 48 *total units* the *fsp* and *fsdp* task perform basically identically whereas the *rwp* task learns significantly slower and reaches a slightly lower maximum return with TD3. With SAC al three tasks perform very similar. The runs of *rwp* for 240 *total units* are the same as with 10k *pretrain steps* so we have the same odd behaviour for both TD3

and SAC. The learning speed and maximum return for the *fsp* and *fsdp* task improved and both tasks still perform equally well for TD3. For SAC, however, *fsp* now has a notable advantage over *fsdp*. Increasing the *total units* further to 960 does not increase the performance and learning speed any further for *fsp* and *fsdp* with TD3. Comparing the runs for 48 *total units* and 960 *total units* for *rwp* (ignoring the odd runs with 240 *total units* out) there is a clear increase in performance and learning speed. Although the learning speed is a little bit slower compared to *fsp* and *fsdp* the end score is almost identical. With SAC all three tasks now perform nearly identical with *fsdp* having a slightly faster learning speed and *rwp* performing slightly worse at the end.



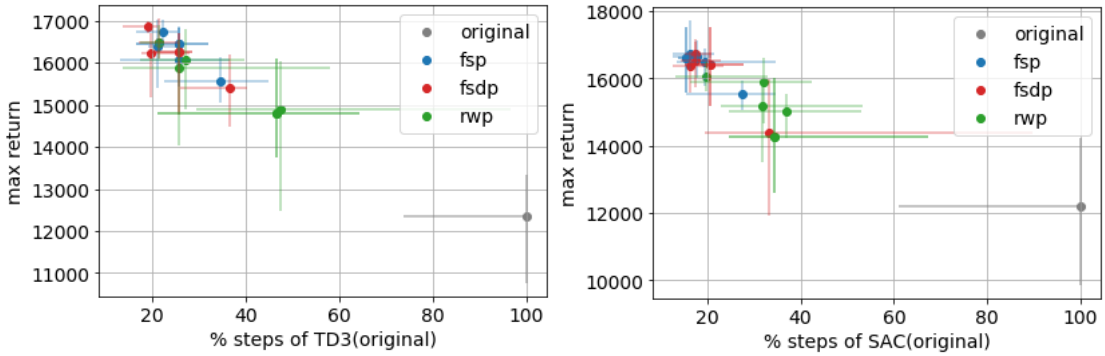
(a) TD3 | 48 total units **(b)** TD3 | 240 total units **(c)** TD3 | 960 total units

(d) SAC | 48 total units **(e)** SAC | 240 total units **(f)** SAC | 960 total units

Figure 5.9: Comparison of using the auxiliary tasks *fsp*, *fsdp* and *rwp* to learn representations with OFENet for TD3/SAC on the HalfCheetah-v2 task. The representations have a dimensionality corresponding to 48, 240 or 960 *total units* (see equations 4.4 and 4.5 in section 4.1). TD3/SAC (original) shows the performance of using the regular algorithm without OFENet. The solid lines and the shaded area represent the average return and the corresponding min/max for five different random seeds.

Figure 5.10 shows the max return and *% steps of TD3/SAC (original)* for all runs of both experiments with different *pretrain steps* and *total units*. For TD3 most runs are close together and save about 70-80% samples compared to TD3 (original). *fsdp* is slightly ahead in terms of both max return and sample efficiency while *rwp* performs worse overall. However, all three auxiliary tasks are able to reach very similar performances, depending on the *pretrain steps* and *total units*. With SAC the sample efficiency is even larger as many runs are able to save >80% samples compared to SAC (original). Here, the gap

between *rwp* and the other two auxiliary tasks is more distinctive. The reasoning for the worse performance of *rwp*, compared to *fsp* and *fsdp*, is the same as with Humanoid-v2, namely that the lower dimensionality of the prediction target (1 vs. 17) allows to learn less valuable representations resulting in a weaker overall performance.



(a) TD3 | pretrain steps & total units

(b) SAC | pretrain steps & total units

Figure 5.10: Comparison of max return and *% steps of TD3/SAC (original)* of all runs in this section with varying *pretrain steps* and *total units* for Humanoid-v2. % steps of TD3/SAC (original) describes how many steps (or samples) of TD3/SAC (original) are needed to reach the max return of TD3/SAC (original). Represented are averages over five runs with different random seeds and corresponding min/max values.

5.2.3 Hopper-v2

In this section the values for *pretrain steps* and *total* units were varied for the MuJoCo task Hopper-v2. This environment has the observation and action space with the least dimensionality (11/3) of the used environments and may be thus be considered the least complex one. This leads to the expectation that the difference between the two more complex auxiliary tasks, the *fsp* and *fsdp* task, and the less complex *rwp* task is smaller than with the other environments Humanoid-v2 and HalfCheetah-v2.

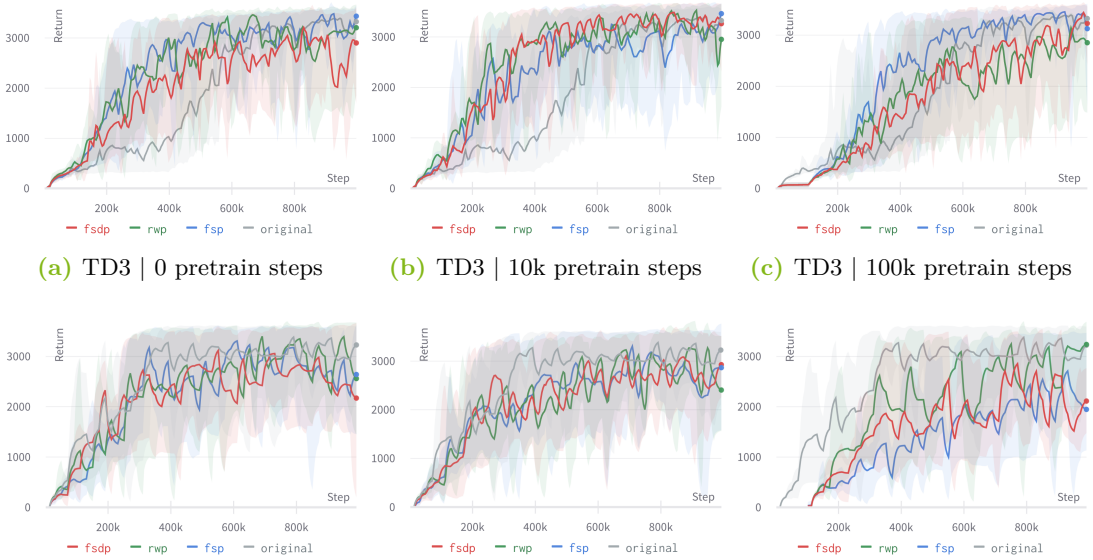
In Figure 5.11 are the results for using the auxiliary tasks to learn representations with OFENet for TD3/SAC with various amounts of *pretrain steps*.

With TD3 for 0 *pretrain steps* in the beginning, all three tasks learn significantly faster than TD3 (original). At about 600k steps TD3 (original) catches up and now performs similar to *fsp* and *rwp*. *fsdp* learns a little bit slower than the other two tasks and on average reaches lower returns overall. However, the min/max-range indicates that *fsdp* at least is able to reach similar scores but at the same time shows the least stable behaviour.

With SAC there is no notable difference between the task and they perform the same as SAC (original).

Increasing the *pretrain steps* to 10k slightly improves the learning speed and performance of *fsdp* for TD3, although this may not be significant as the plots are quite volatile behaviour. The other two auxiliary tasks do not show any significant change. The performances with SAC are the same as with 0 *pretrain steps*.

Further increasing the *pretrain steps* to 100k slows down the learning for *fsdp* and *fsdp* so that they now perform much closer to TD3 (original) in the beginning, and even a little bit worse towards the end. However, they still reach similar high end returns as with the other amounts of *pretrain steps*. The *fsp* task performs at least equally well, if not better, than with 0k or 10k *pretrain steps*. For SAC the increased *pretrain steps* are harmful for all auxiliary tasks and especially *fsp* and *fsdp* now perform much worse than SAC (original) whereas *rwp* learns slower but reaches similar end returns. The slower learning speed with 100k *pretrain steps* can be explained by the agent training starting later and more random samples in the replay buffer which seem to hinder the learning of optimal behaviour for this environment.



(a) TD3 | 0 pretrain steps (b) TD3 | 10k pretrain steps (c) TD3 | 100k pretrain steps

(d) SAC | 0 pretrain steps (e) SAC | 10k pretrain steps (f) SAC | 100k pretrain steps

Figure 5.11: Comparison of using the auxiliary tasks *fsp*, *fsdp* and *rwp* to learn representations with OFENet for TD3/SAC on the Hopper-v2 task. The OFENet has been pretrained with 0, 10k or 100k random samples before the agent training. TD3/SAC (original) shows the performance of using the regular algorithm without OFENet where 10k random samples are collected before the agent training as well. The solid lines and the shaded area represent the average return and the corresponding min/max for five different random seeds.

Figure 5.11 shows the results for using the auxiliary tasks to learn representations with OFENet for TD3/SAC with varying values of *total units*.

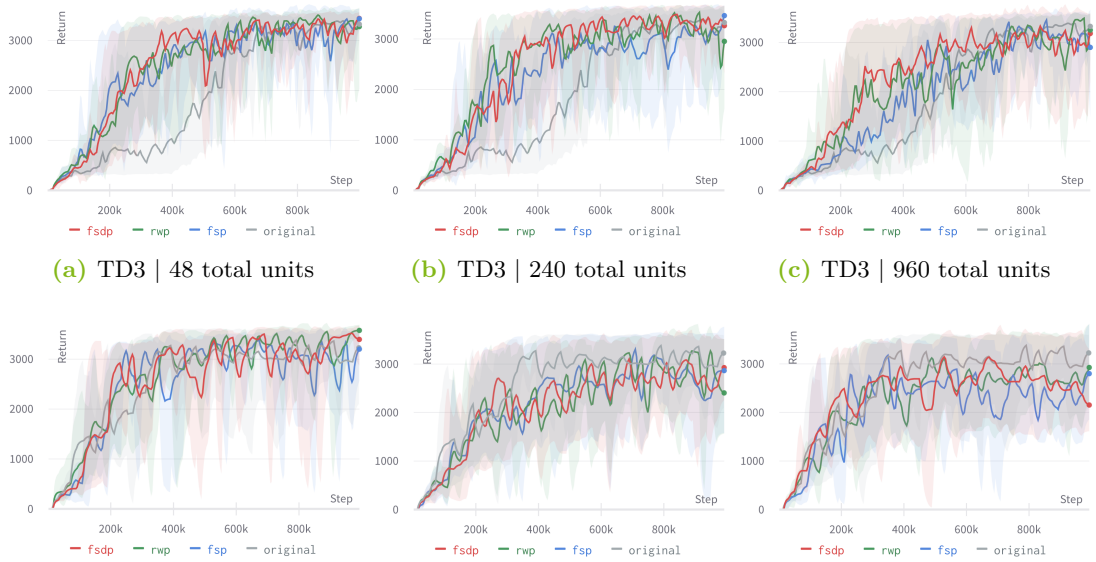
With 48 *total units* all three auxiliary tasks perform similar and learn much quicker than TD3 (original). At about 700k steps TD3 (original) catches up to the auxiliary tasks and performs equally well from there on. With SAC all tasks and SAC (original) perform the same.

Increasing the *total units* to 240 seems to weaken the performance of *fsp* with TD3 a little bit by making it less stable although this effect may not be significant enough to link it to the increased *total units*. In general, the behaviour of the three auxiliary tasks did not change significantly compared to 48 *total units*. With SAC all tasks now perform slightly worse as with 48 *total units*.

Further increasing the *total units* to 960 significantly slows down the learning for all three tasks with TD3 and slightly reduces the scores reached in general. As with SAC

the learning speed with 960 *total units* is not decreased compared to 240 *total units* the general volatile behaviour of Hopper-v2 is the most likely explanation for this.

To conclude the experiments for varying the *total units* for the Hopper-v2 environment, all three auxiliary tasks performed similar for TD3 and SAC throughout. For all three tasks and both RL algorithms the higher values for *total units* 240 and 960 rather harm the agent learning in terms of learning speed and overall scores reached. This indicates that the larger complexity of the learned representations with larger dimensionality is too much for this rather simple environment so that lower amounts of *total units* should be preferred for all auxiliary tasks.



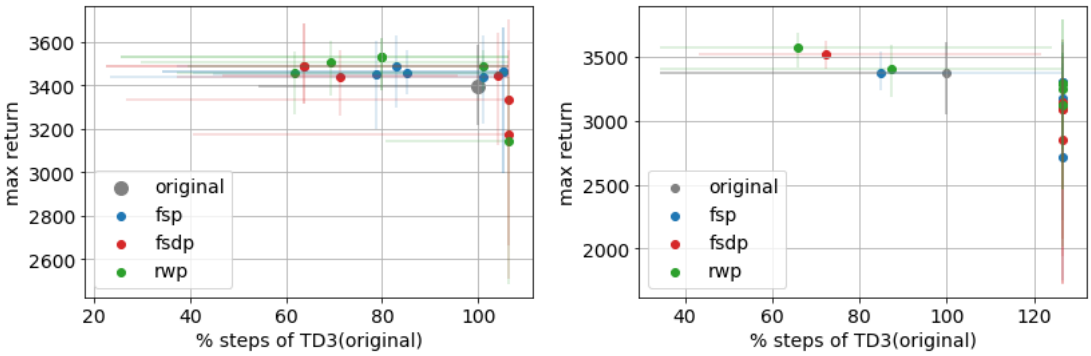
(a) TD3 | 48 total units **(b)** TD3 | 240 total units **(c)** TD3 | 960 total units

(d) SAC | 48 total units **(e)** SAC | 240 total units **(f)** SAC | 960 total units

Figure 5.12: Comparison of using the auxiliary tasks *fsp*, *fsdp* and *rwp* to learn representations with OFENet for TD3/SAC on the HalfCheetah-v2 task. The representations have a dimensionality corresponding to 48, 240 or 960 *total units* (see equations 4.4 and 4.5 in section 4.1). TD3/SAC (original) shows the performance of using the regular algorithm without OFENet. The solid lines and the shaded area represent the average return and the corresponding min/max for five different random seeds.

To combine the results of both experiments with different *pretrain steps* and *total units*, Figure 5.13 compares the max return and *% steps of TD3/SAC (original)*, a measure of the sample efficiency. For TD3 three runs do not reach the max return of TD3 (original) and four runs take longer to reach it than TD3 (original). The other runs only provide a very minor increase of the max return but are able to reach the max return of TD3 (original) with about 60-85% of the samples. *rwp* performs best in terms of max return

and sample efficiency, although just very slightly and possibly insignificant. With SAC only four runs were able to reach higher max returns than SAC (original). Among these runs *rwp* again shows the best performance overall, providing similar improvements as with TD3. The comparatively low benefit achieved with all three auxiliary tasks. A probable reason given for this was that the upper limit of the max returns might generally be reached at the 3.500-return-mark in the Hopper-v2 environment and any further improvement might not be possible. The relatively low added value for the max return could be due to the fact, that for Hopper-v2 the possible max return is probably already reached at the 3,500-return-mark. Since already TD3/SAC (original) can basically reach this value, no significant improvement can be achieved with the representations of OFENet. Considering the stochasticity of the algorithms and the dispersion of the runs, no significant differences between the auxiliary tasks could be observed.



(a) TD3 | pretrain steps & total units

(b) SAC | pretrain steps & total units

Figure 5.13: Comparison of max return and *% steps of TD3/SAC (original)* of all runs in this section with varying *pretrain steps* and *total units* for Hopper-v2. % steps of TD3/SAC (original) describes how many steps (or samples) of TD3/SAC (original) are needed to reach the max return of TD3/SAC (original). Represented are averages over five runs with different random seeds and corresponding min/max values. The rightmost values did not reach the max return of the original algorithm within 1M steps.

5.2.4 *Pretrain Steps*

In this section the auxiliary tasks *fsp*, *fsdp* and *rwp* are inspected individually w.r.t. varying amounts of *pretrain steps*={0,10k,100k} which correspond to different amounts of random samples collected before the agent training that are used for pretraining the OFENet. To better compare the impact of the additional pretraining the plots were aligned so that the agent training always starts at step 0. It is assumed that the influence of the pretraining amount decreases in the course of the agent training as the volume of pretraining becomes smaller and smaller in comparison to the total training. This would

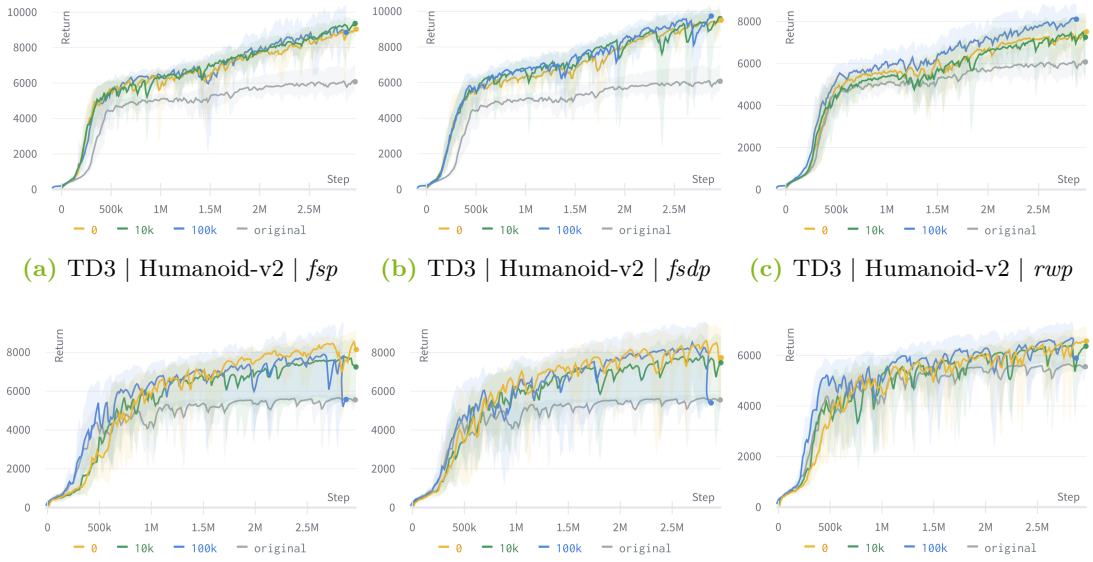
result in using different volumes of pretraining eventually converging towards the same max return. However, it is possible that the pretraining - depending on the volume - accelerates or slows down this convergence.

Figure 5.14 shows the results for all three auxiliary tasks for the Humanoid-v2 task.

For *fsp* there is no significant difference between the three amounts of *pretrain steps* for TD3. With SAC *fsp* initially learns faster up to about 1M steps with 100k *pretrain steps*.

With *fsdp* the performance is fairly equal among all amounts of *pretrain steps* for TD3 and SAC although using 10k *pretrain steps* seems to be slightly disadvantageous for SAC.

rwp with TD3 seems to profit noticeably from 100k *pretrain steps* in terms of learning speed and overall returns, compared to the other two values which perform very similar. This indicates that pretraining the OFENet with the *rwp* task before the agent training with 100k random samples allows to learn more valuable representations resulting in faster learning, compared to training OFENet and RL agent simultaneously right from the start. As already mentioned in section 5.2.1 a reasonable explanation for this is that with the *rwp* task the representations learn information relevant to the actual RL task. More pretraining with the *rwp* task consequently increases the task relevant information in the representation, which can increase initial learning speed and max returns reached. With SAC *rwp* also learns faster and reaches slightly higher scores with 100k *pretrain steps* compared to 0 and 10k *pretrain steps*. Considering the min/max range as well, where *rwp* with 100k *pretrain steps* is significantly ahead for TD3 and SAC, further supports the given explanation.



(a) TD3 | Humanoid-v2 | *fsp* (b) TD3 | Humanoid-v2 | *fsdp* (c) TD3 | Humanoid-v2 | *rwp*

(d) SAC | Humanoid-v2 | *fsp* (e) SAC | Humanoid-v2 | *fsdp* (f) SAC | Humanoid-v2 | *rwp*

Figure 5.14: Examining the impact of varying *pretrain steps* individually for each auxiliary task *fsp*, *fsdp* and *rwp* on the Humanoid-v2 task for TD3 and SAC. The OFENet has been pretrained with 0, 10k or 100k random samples before the agent training. TD3/SAC (original) shows the performance of using the regular algorithm without OFENet where 10k random samples are collected before the agent training as well. The solid lines and the shaded area represent the average return and the corresponding min/max for five different random seeds.

In Figure 5.15 are the results for all three auxiliary tasks for the HalfCheetah-v2 task.

For *fsp* using 100k *pretrain steps* shows a slight but notable improvement of the initial learning speed with TD3, compared to 0 and 10k *pretrain steps*. The max returns seem to converge to the same value towards the end.

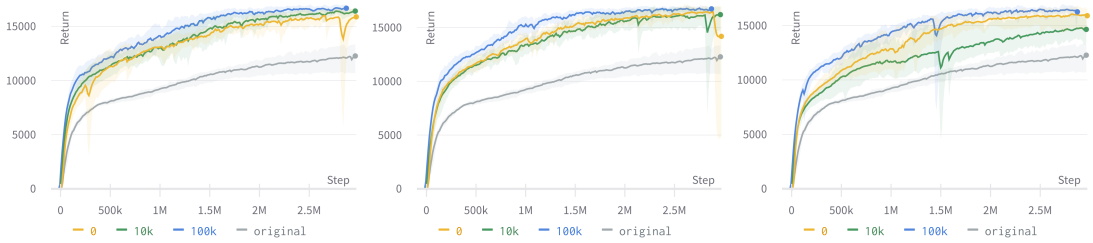
The performance of 0 and 10k *pretrain steps* is quite similar. For SAC there is no difference between the amounts of *pretrain steps*.

For *fsdp* for TD3 and SAC the performance of 0 and 10k *pretrain steps* is approximately the same whereas 100k *pretrain steps* has a notable lead.

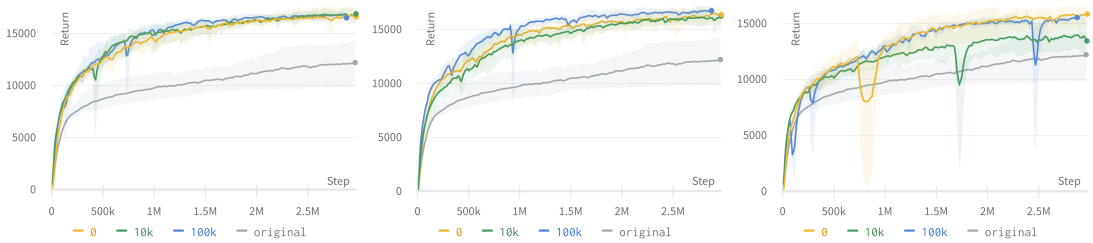
For *rwp* with TD3 the performances of 0 and 100k *pretrain steps* are similar. The runs for 10k *pretrain steps* with TD3 and SAC again show an odd behaviour and it is unclear why using 10k *pretrain steps* dramatically harms the performance of *rwp* but using 0 and 100k *pretrain steps* does not.

For *fsp* (TD3), *fsdp* (TD3&SAC) and *rwp* the improvement in learning speed when using 100k *pretrain steps* can be explained by better and more valuable representations learned by OFENet. This results in faster convergence of the max return, compared to 0 and 10k *pretrain steps*. The question arises whether it is worthwhile to use more samples for pretraining and less for simultaneous training of OFENet and agent. To investigate this further, in Figure 5.16 the runs are compared again, where now the pretraining is included in the total steps. In TD3 it turns out that it is indeed worthwhile and with more pretraining the max return converges faster, whereas with SAC it makes no difference. Consequently, it can increase the overall learning speed if at the beginning more samples are used only for the pretraining of OFENet and thus less samples are left for the simultaneous training of OFENet and agent.

A possible explanation for the overall faster learning with more pretraining is that richer representations can be learned with the random samples than with the agent-generated samples. Since the representations of the OFENet also influences the behavior of the agent - i.e. the agent-generated samples - these richer representations could have a positive effect on the exploration of the agent. This improved exploration at the expense of more pretraining would then be more profitable in this case than to start earlier with the simultaneous training of OFENet and agent with agent-generated. Since this occurs only for TD3 and not for SAC, it is assumed that the exploration in SAC is already good enough and therefore does not benefit from this effect. In contrast to TD3, SAC has a stochastic policy and uses entropy regularization for exploration.

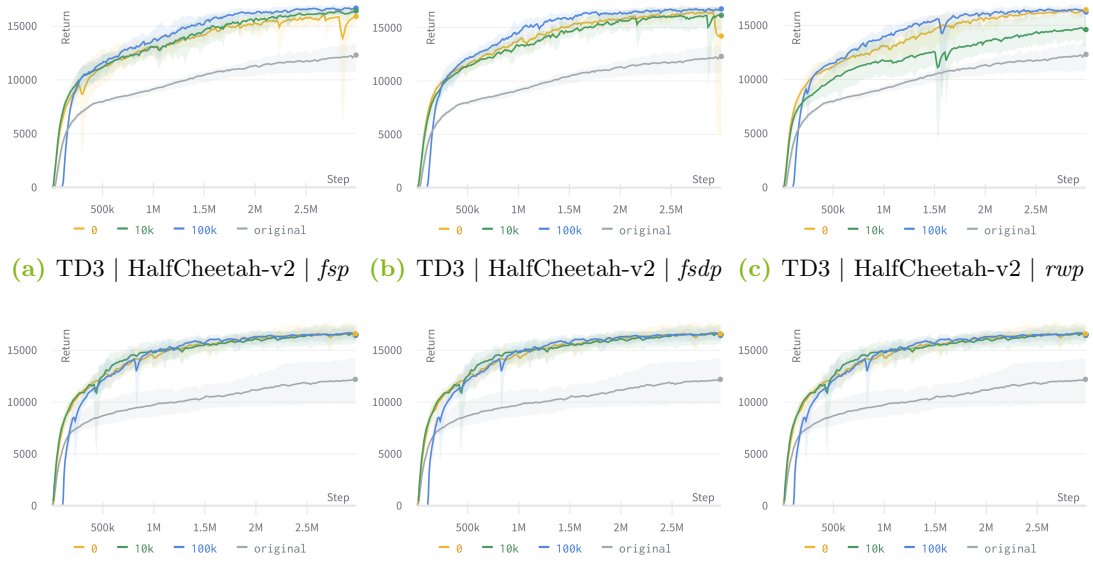


(a) TD3 | HalfCheetah-v2 | *fsp* (b) TD3 | HalfCheetah-v2 | *fsdp* (c) TD3 | HalfCheetah-v2 | *rwp*



(d) SAC | HalfCheetah-v2 | *fsp* (e) SAC | HalfCheetah-v2 | *fsdp* (f) SAC | HalfCheetah-v2 | *rwp*

Figure 5.15: Examining the impact of varying *pretrain steps* individually for each auxiliary task *fsp*, *fsdp* and *rwp* on the HalfCheetah-v2 task for TD3 and SAC. The OFENet has been pretrained with 0, 10k or 100k random samples before the agent training. TD3/SAC (original) shows the performance of using the regular algorithm without OFENet where 10k random samples are collected before the agent training as well. The solid lines and the shaded area represent the average return and the corresponding min/max for five different random seeds.

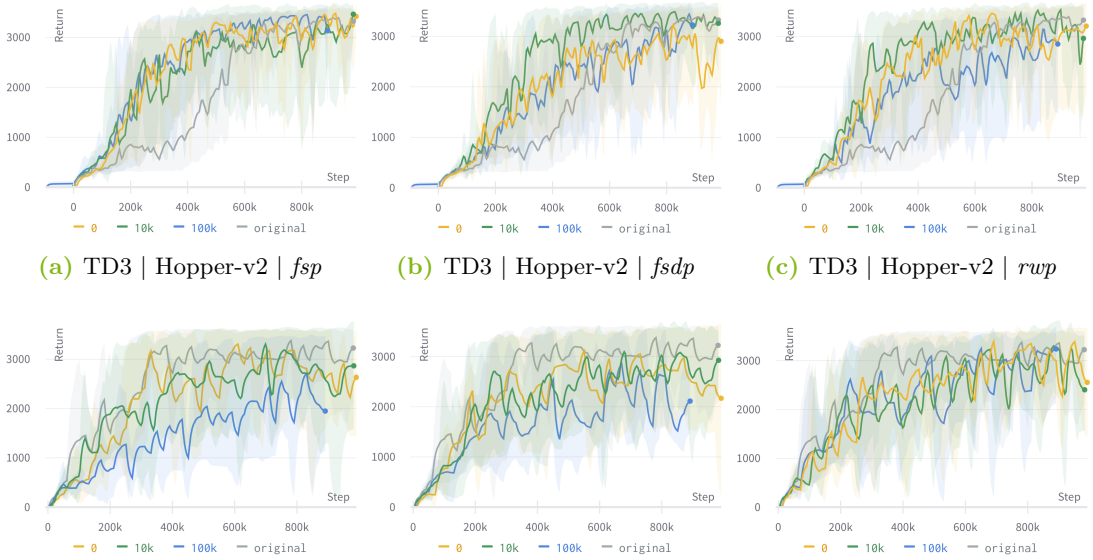


(a) TD3 | HalfCheetah-v2 | *fsp* **(b)** TD3 | HalfCheetah-v2 | *fsdp* **(c)** TD3 | HalfCheetah-v2 | *rwp*

(d) SAC | HalfCheetah-v2 | *fsp* **(e)** SAC | HalfCheetah-v2 | *fsdp* **(f)** SAC | HalfCheetah-v2 | *rwp*

Figure 5.16: Examining the impact of varying *pretrain steps* individually for each auxiliary task *fsp*, *fsdp* and *rwp* on the HalfCheetah-v2 task for TD3 and SAC. The OFENet has been pretrained with 0, 10k or 100k random samples before the agent training. TD3/SAC (original) shows the performance of using the regular algorithm without OFENet where 10k random samples are collected before the agent training as well. This time, the pretraining of OFENet is included within the overall steps, unlike Figure 5.15. With TD3 it is clearly visible that more pretraining can improve the learning speed. The solid lines and the shaded area represent the average return and the corresponding min/max for five different random seeds.

Figure 5.17 shows the results for the Hopper-v2 task. For *fsp* with TD3 the performances with all three amounts of *pretrain steps* are very similar. With SAC increasing the *pretrain steps* significantly harms the learning speed and overall return. For *fsdp* with TD3 the runs with 10k *pretrain steps* performed the best while the other two performed similar and only slightly worse. With SAC the picture is similar as with *fsp*. Lastly, for *rwp* with TD3 and SAC there is no notable difference in performance between the three amounts of *pretrain steps*. Again, it should be noted that the Hopper-v2 environment generally shows a more dispersing behaviour than the Humanoid-v2 and HalfCheetah-v2 and the better performance of *fsp* with 10k *pretrain steps* might not be significant enough to be explained by the *pretrain steps*.



(a) TD3 | Hopper-v2 | *fsp* (b) TD3 | Hopper-v2 | *fsdp* (c) TD3 | Hopper-v2 | *rwp*

(d) SAC | Hopper-v2 | *fsp* (e) SAC | Hopper-v2 | *fsdp* (f) SAC | Hopper-v2 | *rwp*

Figure 5.17: Examining the impact of varying *pretrain steps* individually for each auxiliary task *fsp*, *fsdp* and *rwp* on the Hopper-v2 task for TD3 and SAC. The OFENet has been pretrained with 0, 10k or 100k random samples before the agent training. TD3/SAC (original) shows the performance of using the regular algorithm without OFENet where 10k random samples are collected before the agent training as well. The solid lines and the shaded area represent the average return and the corresponding min/max for five different random seeds.

Summarizing the results for this section, there are some cases in which increased *pretrain steps* were able to increase the learning speed, allowing faster convergence to the max return. As this effect appears only for certain scenarios, the amount of *pretrain steps* is a hyperparameter that can improve learning speed, depending on the scenario.

5.2.5 *Total Units*

In this section the auxiliary tasks *fsp*, *fsdp* and *rwp* are inspected individually w.r.t. varying amounts of *total units*={48,240,960} which correspond to different dimensionalities of the representations z_{o_t} and z_{o_t,a_t} learned by the OFENet.

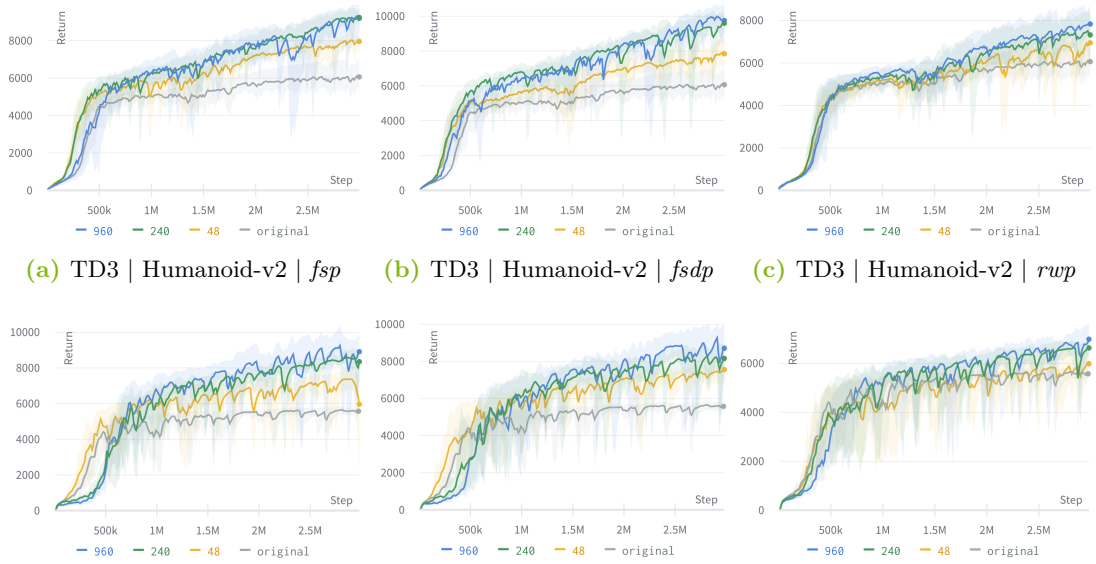
In Figure 5.18 are the results for all three auxiliary tasks for the Humanoid-v2 task.

For TD3 *fsp* has the worst performance with 48 and the best with 240 *total units*. This indicates that the representations with the lower dimensionality can't reach their full potential leading to a lower overall return. Using 960 *total units* results in end returns

similar to 240 *total units* but decreases the initial learning speed. This can be linked to the significantly increased number of parameters when using 960 *total units* which results in an increased learning time not only for the representations but for the RL algorithm as well. (For a given environment the number of *total units* is proportional to the number of parameters of the OFENet and the RL algorithm.) For SAC the higher values for *total units* significantly improve the performance while decreasing the initial learning speed at the same time. Unlike with TD3, increasing the *total units* from 240 to 960 further increases the overall return markedly.

For *fsdp* with TD3 the initial learning speed of all *total units* is very similar with 240 *total units* having a slight lead. From there on, the performances of 240 and 960 *total units* are almost on a par with 960 *total units* reaching slightly higher returns towards the end, even reaching the 10000-score-mark on average. This indicates slightly increased value of the representations with 960 *total units*, probably a result of the increased complexity. For SAC the picture is similar as with *fsp*.

rwp with TD3 task does not profit as much from the increased *total units* as the other tasks and the performance increase only shows after about 1.5M steps for all values of *total units*. From then on, 960 *total units* perform the best closely followed by 240 and then 48 *total units*. This suggests that the comparatively complex Humanoid-v2 environment profits from more complex representations. *rwp* with SAC shows a similar behaviour but performs more dispersive up to 1.5M steps.



(a) TD3 | Humanoid-v2 | *fsp* (b) TD3 | Humanoid-v2 | *fsdp* (c) TD3 | Humanoid-v2 | *rwp*

(d) SAC | Humanoid-v2 | *fsp* (e) SAC | Humanoid-v2 | *fsdp* (f) SAC | Humanoid-v2 | *rwp*

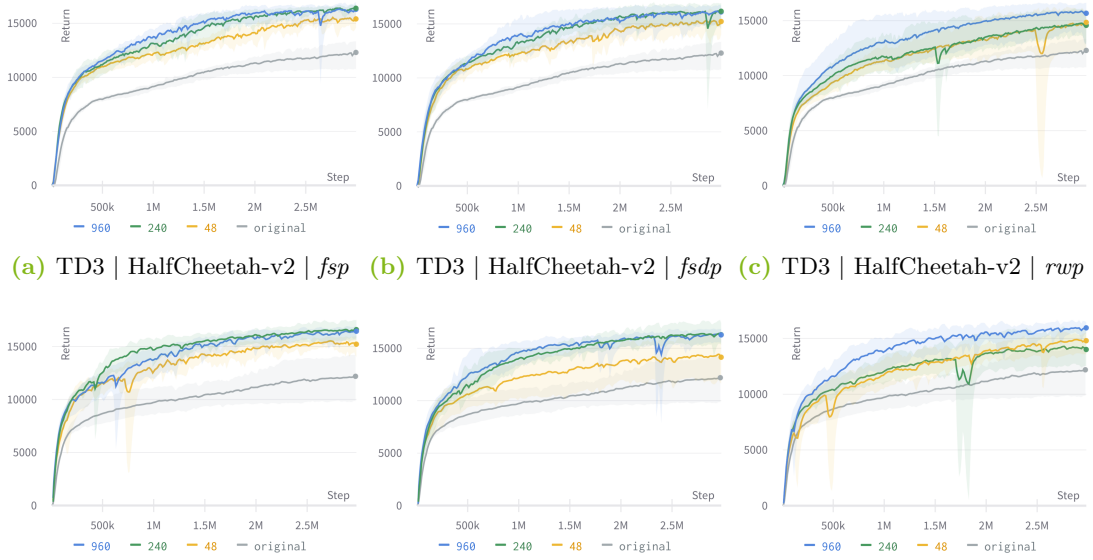
Figure 5.18: Examining the impact of varying *total units* individually on each auxiliary task *fsp*, *fsdp* and *rwp* on the Humanoid-v2 task for TD3 and SAC. The representations have a dimensionality corresponding to 48, 240 or 960 *total units* (see equation 4.6 in section 4.1). TD3/SAC (original) shows the performance of using the regular algorithm without OFENet. The solid lines and the shaded area represent the average return and the corresponding min/max for five different random seeds.

In Figure 5.19 are the results for all three auxiliary tasks for the HalfCheetah-v2 task.

For *fsp* with TD3 and SAC using 48 *total units* results in the worst performance but only after the initial learning phase. This is the same effect as with Humanoid-v2, indicating that the lower *total units* do not allow the representations to reach their full potential. 240 and 960 *total units* perform nearly identical with 960 *total units* for TD3 and 240 *total units* for SAC having a slight lead in the middle section between 500k and 2 million steps but reaching the same returns towards the end as with 240 *total units*.

The performances of *fsdp* with TD3 are identical with the performances of the *fsp* task. With SAC the 960 *total units* have a slight advantage.

For *rwp* with 240 *total units* we have the same odd behaviour already stated in section 5.2.1 for TD3 and SAC. The expectation would have been that 240 *total units* perform better than 48 *total units* and closer to 960 *total units* (with the *pretrain steps* fixed to 10k). Assuming this would be a representative behaviour, the same reasoning of 48 *total units* being inferior to the other values leading to lower overall return would apply.



(a) TD3 | HalfCheetah-v2 | *fsp* **(b)** TD3 | HalfCheetah-v2 | *fsdp* **(c)** TD3 | HalfCheetah-v2 | *rwp*

(d) SAC | HalfCheetah-v2 | *fsp* **(e)** SAC | HalfCheetah-v2 | *fsdp* **(f)** SAC | HalfCheetah-v2 | *rwp*

Figure 5.19: Examining the impact of varying *total units* individually on each auxiliary task *fsp*, *fsdp* and *rwp* on the HalfCheetah-v2 task for TD3 and SAC. The representations have a dimensionality corresponding to 48, 240 or 960 *total units* (see equation 4.6 in section 4.1). TD3/SAC (original) shows the performance of using the regular algorithm without OFENet. The solid lines and the shaded area represent the average return and the corresponding min/max for five different random seeds.

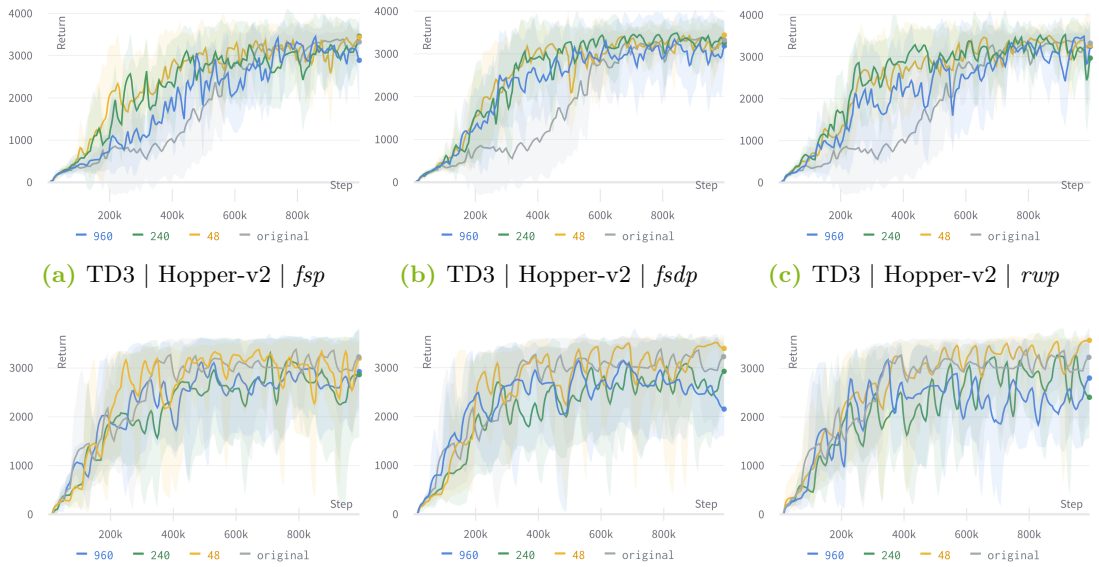
In Figure 5.20 are the results for all three auxiliary tasks for the Hopper-v2 task.

fsp with 240 and 48 *total units* with TD3 performs very similar whereas with 960 *total units* the learning speed is decreased although similar returns are reached towards the end. Again, an indication for 960 *total units* resulting in too complex representations that harm the learning process.

However, with the *fsdp* task this seems to be no problem and all three amounts of *total units* result in an overall equal performance. This suggests that *fsdp* is less sensitive for the choice of *total units*, at least for the rather uncomplex Hopper-v2 environment.

rwp shows a similar behaviour as *fsp* with 48 and 240 *total units* performing equally well and 960 *total units* slightly decreasing the learning speed.

For SAC the increased *total units* rather harm the learning in general so that 48 *total units* perform the best throughout. Using representations learned with OFENet provides an improvement over SAC (original) only with 48 *total units*. Again, an indication that too complex representations are not useful for the Hopper-v2 environment.



(a) TD3 | Hopper-v2 | *fsp* (b) TD3 | Hopper-v2 | *fsdp* (c) TD3 | Hopper-v2 | *rwp*

(d) SAC | Hopper-v2 | *fsp* (e) SAC | Hopper-v2 | *fsdp* (f) SAC | Hopper-v2 | *rwp*

Figure 5.20: Examining the impact of varying *total units* individually on each auxiliary task *fsp*, *fsdp* and *rwp* on the Hopper-v2 task for TD3 and SAC. The representations have a dimensionality corresponding to 48, 240 or 960 *total units* (see equation 4.6 in section 4.1). TD3/SAC (original) shows the performance of using the regular algorithm without OFENet. The solid lines and the shaded area represent the average return and the corresponding min/max for five different random seeds.

Summarizing the results for this section, for the more complex environment Humanoid-v2 *fsp* and *fsdp* profit from the increased *total units* much more than *rwp* but increasing them reduces the initial learning speed. However, using 960 *total units* significantly increases the overall returns for SAC in most cases and very slightly for *fsdp* with TD3 towards the end. For the less complex environment HalfCheetah-v2 the performances of all auxiliary tasks for all values of *total units* are generally closer to each other, for both TD3 and SAC. Reducing the complexity further, with the Hopper-v2 environment the 48 and 240 *total units* generally perform better than 960 *total units*. While the performance of 48 and 240 *total units* with TD3 is similar, using 48 *total units* performs notably the best and is the only configuration that provides an advantage over SAC (original). As a consequence, this suggests a connection between the complexity of the environment (i.e. the dimensionalites of observation and action space) and the optimal complexity of the representations learned by OFENet (i.e. the *total units*) for all auxiliary tasks.

In summary, in more complex environments (Humanoid-v2) 960 *total units*, compared to 48 and 240, in most cases slow down the initial learning but increase the max return. Especially SAC benefits from the 960 *total units*. One reason for this could be that the

return of SAC seems to flatten out with 240 *total units*, whereas it increases continuously with TD3. This means that the return for SAC with 240 *total units* is already converging to its potential maximum, which can be raised with the additional expressivity of 960 *total units*. For environments with medium dimensionality (HalfCheetah-v2), the 960 *total units* increase the learning speed in most cases. *rwp* benefits significantly from the 960 *total units*. One possible explanation for the significant improvement of *rwp* is, that with *rwp*, due to the lower complexity of the target in comparison to the other auxiliary tasks, the increased complexity of the representations can be better exploited overall. For less complex environments (Hopper-v2), fewer *total units* (48) perform best. This shows a trend that with increasing complexity of the environment the the complexity should increase to reach the maximum return. However, it is possible that after a certain point the initial learning speed decreases slightly.

6 Conclusion

The aim of this work was to compare auxiliary tasks for low-dimensional representation learning for reinforcement learning. As a method for representation learning the OFENet by Ota et al. [31] was used, which learns higher-dimensional representations of low-dimensional data by using an auxiliary task. For this I compared two auxiliary tasks that are based on predicting observations (*fsp, fsdp*), one that is based on predicting rewards (*rwp*) and one that is based on predicting actions between two successive observations (*inv*). The latter proved to be not usable for actor-ciritc algorithms whose updates are based on differentiating a Q-function, like TD3 and SAC.

The remaining three auxiliary tasks were investigated in the comparative experiments with regard to sample efficiency and max return. In addition, the influence of different values of the hyperparameters *pretrain steps* and *total units* of the OFENet, the first of which determines the amount of pretraining and the second the dimensionality of the representations, was investigated.

In order to compare the auxiliary tasks comprehensively, the best performance of each auxiliary task in terms of max return and sample efficiency for each for each environment is taken. This comparison is intended to illustrate the potential that can be achieved with the auxiliary tasks if the hyperparameters *pretrain steps* and *total units* are chosen optimally (with respect to the values investigated in this work). To be able to quantify the sample efficiency, the value *% steps of TD3/SAC (original)* was introduced, which describes what proportion of samples in comparison to TD3/SAC (original) is needed with the respective auxiliary task, to reach the max return of TD3/SAC (original).

Figure 6.1 shows the best performance of each auxiliary task for the three environments Humanoid-v2, HalfCheetah-v2 and Hopper-v2 with respect to max return and sample efficiency. In order to be able to show the max return across environments, it is scaled w.r.t. the max return of TD3/SAC (original) and displayed as a percentage value. For better illustration, the reciprocal value of the proportion of *% steps of TD3/SAC (original)* is used, which consequently describes how much faster the respective auxiliary task can reach the max return of TD3/SAC (original) compared to TD3/SAC (original). This value is called "Sample Efficiency Increase". As an example, a "*% steps of TD3/SAC (original)*"-value of 20% corresponds to a sample efficiency increase of 5. Table 6.1 shows the corresponding values as well as a comparison of the auxiliary tasks across the environments and algorithms.

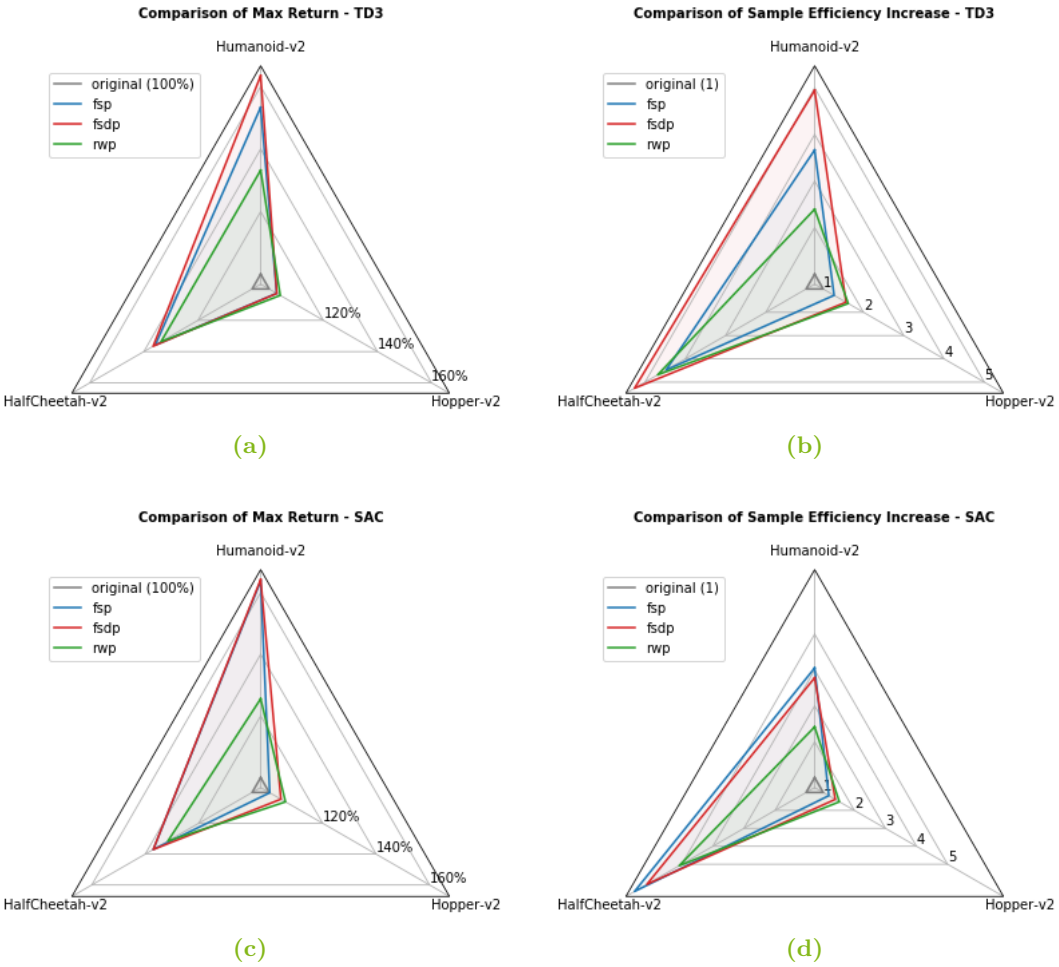


Figure 6.1: Comparison of the max return und sample efficiency increase of the auxiliary tasks across the environments Hopper-v2, Humanoid-v2 and HalfCheetah-v2 for TD3 and SAC. For each auxiliary task the run with the configuration of *pretrain steps* and *total units* is represented, which performed best for max return or sample efficiency increase. The max return is shown as a percentage and scaled w.r.t. the respective max return of TD3/SAC (original). The sample efficiency increase describes how much faster (in terms of samples used) the respective auxiliary task can reach the max return of TD3/SAC (original) compared to TD3/SAC (original). Corresponding values are represented in Table 6.1.

Overall, *fsdp* performs best with *fsp* being only slightly inferior. *rwp* performs overall significantly worse than the other two auxiliary tasks, with the difference being greatest in the most complex Humanoid-v2 environment. In Hopper-v2, on the other hand,

only a comparatively low benefit could be achieved with all three auxiliary tasks. A probable reason given for this was that the upper limit of the max returns might generally be reached at the 3.500-return-mark in the Hopper-v2 environment and any further improvement might not be possible.

As the comparison across environments and algorithms in table 6.1b shows, the *fsdp* task generally performs best in terms of max return and sample efficiency. While, as already stated, *fsp* is only slightly inferior to *fsdp*, *rwp* performs by far the worst. Therefore, if one has to decide in advance for an auxiliary task for the learning of representations and does not know which one is best for a specific scenario, it is recommended to take the *fsdp* task.

Two main reasons were considered for the significantly poorer performance of *rwp*. First, *rwp* has a prediction target with a fixed dimensionality of 1, whereas the dimensionality of the prediction target of *fsp/fsdp* is proportional to the observation space of the environment. This results in decreased potential expressiveness of the representations learned with *rwp* and consequently in a worse performance. For example, increasing the dimensionality of the prediction target (compare HalfCheetah-v2 and Humanoid-v2) resulted in an increasing distance of the performances of *fsp/fsdp* and *rwp*. Secondly, the type of information contained in the representations is expected to be fundamentally different. While in *rwp* with the reward as prediction target task-dependent information is learned, in *fsp/fsdp* this information is task-independent and aims at learning the dynamics of the system. The representations with *fsp/fsdp* could thus be considered more general, which might have a positive effect on exploration and consequently on performance. Regarding the difference between *fsp* and *fsdp*, Anderson et al. [2] state, that the *fsdp* task prevents that an identity is being learned since successive observations are often very similar, which they assume to happen for *fsp*. Consequently, more training is needed to learn the small variations needed to accurately predict the next observation. The results of this work confirm that the representations using the *fsdp* task overall lead to a better performance.

Large amounts of *pretrain steps* (100k) for the low-complex task (Hopper-v2) for all auxiliary tasks has been shown to be harmful to performance in most cases. However, for the tasks with larger complexity (HalfCheetah-v2 and Humanoid-v2) 100k *pretrain steps* are able to increase the learning speed in some cases, resulting in a faster convergence to the same max return compared to 0 or 10k *pretrain steps*. It turns out that 0 pretraining overall is worse than 10k or 100k *pretrain steps*. There are no major differences between the auxiliary tasks as far as the influence of pretraining is concerned.

More *total units* (960) can increase the max return especially in complex environments, like Humanoid-v2, but slow down the initial learning speed. This can be explained by the fact that more *total units* result in a higher expressivity of the representations, which makes it possible to learn more complex and better behavior. Since with more total units

the neuronal networks also have more parameters, this can slow down initial learning. For less complex environments (Hopper-v2), the higher expressivity seems to be rather hindering and the corresponding representations are too complex for the environment, which is why 48 or 240 *total units* should rather be used.

In order to further substantiate the results of this work, the auxiliary tasks should be investigated with further environments and algorithms. The MuJoCo environment Ant lies between HalfCheetah and Humanoid in terms of the dimensionality of the observations. It would therefore be interesting to further investigate possible relations of the improvements of the max return and the sample efficiency w.r.t. the complexity of the environment. Furthermore, the experiments of TD3 with the Humanoid task could be performed with more than 3M steps to see if the use of different auxiliary tasks converges to a different max return. The Walker2d environment might be an interesting comparison to the HalfCheetah environment, as they have an identical dimensionality of the observation/action space. Here it could be investigated whether the behavior of the auxiliary tasks depends only on the dimensionality of the environment or if other factors play a role. Last but not least, it would be very interesting to use the extended version of OFENet by Ota et al. [30] with the *fsdp* task, as it also uses the *fsp* task like the original OFENet. The extended version of OFENet uses even larger networks and a distributed training method to further increase the performance for low-dimensional control problems.

Table 6.1: Comparison of max return and sample efficiency for TD3 (a) and SAC (b) and a summary over both algorithms and all environments (c)). Sample efficiency represents how much faster - in terms of samples used - the respective auxiliary task is able to reach the max return of TD3 (original). The values are based on averages over five runs using different random seeds for the respective configuration (*pretrain steps*, *total units*) with the best performance.

(a) TD3

	Humanoid-v2		HalfCheetah-v2		Hopper-v2	
	Max Return	**Sample Efficiency**	**Max Return**	**Sample Efficiency**	**Max Return**	**Sample Efficiency**
original	6127	1	12 337	1	3392	1
fsp	9394 (10k, 240)	3.7 (100k, 240)	16 750 (100k, 240)	4.5 (100k, 240)	3485 (0k, 240)	1.3 (10k, 960)
fsdp	**10013** (10k, 960)	**5.0** (10k, 240)	**16848** (100k, 240)	**5.3** (100k, 240)	3487 (10k, 240)	**1.6** (10k, 240)
rwp	8162 (100k, 240)	2.4 (100k, 240)	16 487 (100k, 240)	4.7 (100k, 240)	**3530** (10k, 240)	**1.6** (0k, 240)

(b) SAC

	Humanoid-v2		HalfCheetah-v2		Hopper-v2	
	Max Return	**Sample Efficiency**	**Max Return**	**Sample Efficiency**	**Max Return**	**Sample Efficiency**
original	5650	1	12 192	1	3370	1
fsp	9254 (10k, 960)	**4.1** (100k, 240)	16 699 (0k, 240)	**6.5** (10k, 240)	3376 (10k, 48)	1.2 (10k,48)
fsdp	**9279** (10k, 960)	3.8 (0k, 240)	**16703** (100k, 240)	6.1 (10k, 960)	3517 (10k, 48)	1.4 (10k, 48)
rwp	7094 (100k, 240)	2.4 (100k, 240)	16 034 (100k, 240)	5.1 (10k, 960)	**3575** (10k, 48)	**1.5** (10k, 48)

(c) Summarized averages over TD3+SAC and all environments. To compare the max return across the environments, the max return values from a) and b) are divided by the original max return and then represented as an environment independent percentage value. Averaged over TD3+SAC and all environments the *fsdp* task outperforms the *fsp* and *rwp* task in terms of max return and sample efficiency.

	SAC+TD3	
	∅ Max Return (in %)	**∅ Sample Efficiency**
original	100	1
fsp	132	3.6
fsdp	**135**	**3.9**
rwp	123	3.0

Bibliography

[1] Joshua Achiam. "Spinning Up in Deep Reinforcement Learning." In: (2018).

[2] Charles W. Anderson, Minwoo Lee, and Daniel L. Elliott. "Faster reinforcement learning after pretraining deep networks to predict state dynamics." In: *2015 International Joint Conference on Neural Networks (IJCNN)*. 2015, pp. 1–7. DOI: 10.1109/IJCNN.2015.7280824.

[3] Yoshua Bengio, Aaron Courville, and Pascal Vincent. *Representation Learning: A Review and New Perspectives*. 2012. DOI: 10.48550/ARXIV.1206.5538. URL: https://arxiv.org/abs/1206.5538.

[4] Lukas Biewald. *Experiment Tracking with Weights and Biases*. Software available from wandb.com. 2020. URL: https://www.wandb.com/.

[5] Gaudenz Boesch. *Pytorch vs Tensorflow: A Head-to-Head Comparison*. URL: https://viso.ai/deep-learning/pytorch-vs-tensorflow/ (visited on 06/13/2022).

[6] Greg Brockman et al. "OpenAI Gym." In: 2016. eprint: arXiv:1606.01540.

[7] Scott Fujimoto, Herke Hoof, and David Meger. "Addressing Function Approximation Error in Actor-Critic Methods." In: *International Conference on Machine Learning*. 2018, pp. 1582–1591.

[8] *Gym Documentation - MuJoCo*. URL: https://www.gymlibrary.ml/environments/mujoco/ (visited on 06/21/2022).

[9] Tuomas Haarnoja et al. *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. 2018. DOI: 10.48550/ARXIV.1801.01290. URL: https://arxiv.org/abs/1801.01290.

[10] Gao Huang et al. *Densely Connected Convolutional Networks*. 2016. DOI: 10.48550/ARXIV.1608.06993. URL: https://arxiv.org/abs/1608.06993.

[11] Julian Ibarz et al. "How to train your robot with deep reinforcement learning: lessons we have learned." In: *The International Journal of Robotics Research* 40.4-5 (Jan. 2021), pp. 698–721. DOI: 10.1177/0278364920987859. URL: https://doi.org/10.1177%2F0278364920987859.

[12] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. DOI: 10.48550/ARXIV.1502.03167. URL: https://arxiv.org/abs/1502.03167.

[13] Max Jaderberg et al. *Reinforcement Learning with Unsupervised Auxiliary Tasks*. 2016. DOI: 10.48550/ARXIV.1611.05397. URL: https://arxiv.org/abs/1611.05397.

[14] Rico Jonschkowski and Oliver Brock. "State Representation Learning in Robotics: Using Prior Knowledge about Physical Interaction." In: *Robotics: Science and Systems*. 2014.

[15] Phuc H. Le-Khac, Graham Healy, and Alan F. Smeaton. "Contrastive Representation Learning: A Framework and Review." In: *IEEE Access* 8 (2020), pp. 193907–193934. DOI: 10.1109/access.2020.3031549. URL: https://doi.org/10.1109%2Faccess.2020.3031549.

[16] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[17] Noah Krystiniak. *NoKryst13@GitHub*. 2022. URL: https://github.com/NoKryst13/Master_thesis_code.

[18] Moritz Lange. *What Is Representation Learning?* 2022. URL: https://moritzlange.github.io/knowledge/what-is-representation-learning/.

[19] Nevena Lazic et al. "Data Center Cooling using Model-predictive Control." In: *Proceedings of the Thirty-second Conference on Neural Information Processing Systems (NeurIPS-18)*. Montreal, QC, 2018, pp. 3818–3827. URL: https://papers.nips.cc/paper/7638-data-center-cooling-using-model-predictive-control.

[20] Timothée Lesort et al. "State representation learning for control: An overview." In: *Neural Networks* 108 (Dec. 2018), pp. 379–392. DOI: 10.1016/j.neunet.2018.07.006. URL: https://doi.org/10.1016%2Fj.neunet.2018.07.006.

[21] Xiaoyuan Liang et al. "A Deep Reinforcement Learning Network for Traffic Light Cycle Control." In: *IEEE Transactions on Vehicular Technology* 68.2 (Feb. 2019), pp. 1243–1253. DOI: 10.1109/tvt.2018.2890726. URL: https://doi.org/10.1109%2Ftvt.2018.2890726.

[22] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2015. DOI: 10.48550/ARXIV.1509.02971. URL: https://arxiv.org/abs/1509.02971.

[23] Devashree Madhugiri. *A Comparison of TensorFlow vs PyTorch: Deep Learning Frameworks 2022*. URL: https://www.knowledgehut.com/blog/data-science/pytorch-vs-tensorflow (visited on 06/13/2022).

[24] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

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

[26] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. DOI: 10.48550/ARXIV.1312.5602. URL: https://arxiv.org/abs/1312.5602.

[27] *MuJoCo.readthedocs*. URL: https://mujoco.readthedocs.io/en/latest/programming.html (visited on 06/21/2022).

[28] Jelle Munk, Jens Kober, and Robert Babuška. "Learning state representation for deep actor-critic control." In: *2016 IEEE 55th Conference on Decision and Control (CDC)*. 2016, pp. 4667–4673. DOI: 10.1109/CDC.2016.7798980.

[29] OpenAI et al. *Dota 2 with Large Scale Deep Reinforcement Learning*. 2019. DOI: 10.48550/ARXIV.1912.06680. URL: https://arxiv.org/abs/1912.06680.

[30] Kei Ota, Devesh K. Jha, and Asako Kanezaki. *Training Larger Networks for Deep Reinforcement Learning*. 2021. DOI: 10.48550/ARXIV.2102.07920. URL: https://arxiv.org/abs/2102.07920.

[31] Kei Ota et al. *Can Increasing Input Dimensionality Improve Deep Reinforcement Learning?* 2020. DOI: 10.48550/ARXIV.2003.01629. URL: https://arxiv.org/abs/2003.01629.

[32] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[33] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. DOI: 10.48550/ARXIV.1707.06347. URL: https://arxiv.org/abs/1707.06347.

[34] Olivier Sigaud. *Lecture Slides: From Policy Gradient to Actor-Critic methods: From Policy Gradient to Actor-Critic methods Soft Actor Critic*. 2021. URL: http://chronos.isir.upmc.fr/~sigaud/teach/ps/12_sac.pdf.

[35] Olivier Sigaud. *Lecture Slides: From Policy Gradient to Actor-Critic methods. Deep Deterministic Policy Gradient (and td3)*. 2021. URL: http://chronos.isir.upmc.fr/~sigaud/teach/ps/11_ddpg.pdf.

[36] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. DOI: 10.48550/ARXIV.1712.01815. URL: https://arxiv.org/abs/1712.01815.

[37] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. first. MIT press, 1998.

[38] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. second. MIT press, 2018.

[39] O. Tange. "GNU Parallel - The Command-Line Power Tool." In: *;login: The USENIX Magazine* 36.1 (Feb. 2011), pp. 42–47. URL: http://www.gnu.org/s/parallel.

[40] Gerald Tesauro. "Temporal Difference Learning and TD-Gammon." In: *Commun. ACM* 38.3 (Mar. 1995), pp. 58–68. ISSN: 0001-0782. DOI: 10.1145/203330.203343. URL: https://doi.org/10.1145/203330.203343.

[41] Emanuel Todorov, Tom Erez, and Yuval Tassa. "MuJoCo: A physics engine for model-based control." In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[42] Christopher Watkins. "Learning From Delayed Rewards." In: (Jan. 1989).

[43] *Why most researchers are shifting from TensorFlow to Pytorch?* URL: https://www.researchgate.net/post/Why_most_researchers_are_shifting_from_tensorFlow_to_Pytorch (visited on 06/13/2022).

[44] Laurenz Wiskott and Merlin Schüler. *Reinforcement Learning — Lecture Notes —*. Jan. 2019.

[45] Denis Yarats and Ilya Kostrikov. *Soft Actor-Critic (SAC) implementation in PyTorch*. https://github.com/denisyarats/pytorch_sac. 2020.

[46] Liang Yu et al. "A Review of Deep Reinforcement Learning for Smart Building Energy Management." In: *IEEE Internet of Things Journal* 8.15 (2021), pp. 12046–12063. DOI: 10.1109/JIOT.2021.3078462.

[47] Amy Zhang, Harsh Satija, and Joelle Pineau. *Decoupling Dynamics and Reward for Transfer Learning*. 2018. DOI: 10.48550/ARXIV.1804.10689. URL: https://arxiv.org/abs/1804.10689.

[48] Shangtong Zhang, Hengshuai Yao, and Shimon Whiteson. *Breaking the Deadly Triad with a Target Network*. 2021. DOI: 10.48550/ARXIV.2101.08862. URL: https://arxiv.org/abs/2101.08862.

[49] Jian Zhao et al. "State Representation Learning For Effective Deep Reinforcement Learning." In: *2020 IEEE International Conference on Multimedia and Expo (ICME)*. 2020, pp. 1–6. DOI: 10.1109/ICME46284.2020.9102924.

Eidesstattliche Versicherung

(Affidavit)

Krystiniak, Noah
Name, Vorname
(surname, first name)

188906
Matrikelnummer
(student ID number)

☐ Bachelorarbeit
(Bachelor's thesis)

☒ Masterarbeit
(Master's thesis)

Titel
(Title)

A Comparison of Auxiliary Tasks for Low-Dimensional Representation Learning for Reinforcement Learning

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

Ober-Erkenschwick, den 25.07.2022
Ort, Datum
(place, date)

N. Krystiniak
Unterschrift
(signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

Ober-Erkenschwick, den 25.07.2022
Ort, Datum
(place, date)

N. Krystiniak
Unterschrift
(signature)

*Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.