55th IEEE Conference on Decision and Control (CDC 2016)

Learning State Representation for Deep Actor-Critic Control

Jelle Munk, Jens Kober and Robert Babuška

***Abstract*— Deep Neural Networks (DNNs) can be used as function approximators in Reinforcement Learning (RL). One advantage of DNNs is that they can cope with large input dimensions. Instead of relying on feature engineering to lower the input dimension, DNNs can extract the features from raw observations. The drawback of this *end-to-end learning* is that it usually requires a large amount of data, which for real-world control applications is not always available. In this paper, a new algorithm, Model Learning Deep Deterministic Policy Gradient (ML-DDPG), is proposed that combines RL with state representation learning, i.e., learning a mapping from an input vector to a state *before* solving the RL task. The ML-DDPG algorithm uses a concept we call *predictive priors* to learn a model network which is subsequently used to pre-train the first layer of the actor and critic networks. Simulation results show that the ML-DDPG can learn reasonable continuous control policies from high-dimensional observations that contain also task-irrelevant information. Furthermore, in some cases, this approach significantly improves the final performance in comparison to end-to-end learning.**

I. Introduction

In recent years, there has been a growing interest in the use of Deep Neural Network (DNN) as function approximators in Reinforcement Learning (RL). DNNs were used in [1] and [2] to approximate the Q-function when Q-learning was used to learn a task from raw visual observations. In [3] this technique was combined with the actor-critic approach to form the Deep Deterministic Policy Gradient (DDPG) algorithm, which was able to solve several continuous control tasks, including the cart-pole benchmark [4] and the "cheetah" locomotion task introduced in [5].

One important aspect of learning to control is to learn how to efficiently gather the task-relevant sensory information necessary to make informed decisions [6]. Specifically in the case of a robot designed for a wide variety of tasks, the amount of sensory information needed for one particular task, is often far less than the total amount of information that the robot gathers through its sensors [7]. Furthermore, the sensory information (observations) typically requires preprocessing before it can be used in control as state information.

Instead of relying on an engineer to design a state estimator to reconstruct the state vector from a set of observations, ideally the machine should be able to learn this estimator as well. Learning such an observation-to-state mapping, prior to solving the RL problem, is known in the literature as *state representation learning* [7]. Several examples exist in which this approach has been successfully applied, for instance: by using an auto-encoder network [8], Slow Feature Analysis (SFA) [9], robotic priors [7] or by simultaneously learning the reward and transition function [10]. These examples have, however, all focused on learning from visual observations and none of them integrates these methods with algorithms that combine RL with DNNs.

This paper introduces a new algorithm called Model Learning Deep Deterministic Policy Gradient (ML-DDPG), in which state representation learning is combined with a RL algorithm that uses DNNs as a function approximator. The algorithm is designed to learn a wide range of continuous control policies on a varied range of challenging sets of observations, that are typically unsuitable for other control algorithms. It learns from a high-dimensional stream of data that can include information that is both relevant and irrelevant to the task at hand, and can include a Markov state directly or indirectly, by including sequences of actions and observations.

The ML-DDPG algorithm learns a model network based on the concept of *predictive priors*, which assumes that the next state representation and the reward should both be predictable, given the current state and the action taken in that state. The model network is constructed in such a way that it learns the observation-to-state mapping by back-propagating the prediction errors, which results in a state representation that is inherently predictable. Both the actor and the critic then learn from the state representation instead of the raw observations.

The motivation behind the algorithm is to compare an approach based on end-to-end learning [11] with an approach in which learning a state representation from a set of observations precedes the learning of a good control policy for a given task, based on the learned state representation. Given enough data and learning time, end-to-end learning is believed to reach a performance that is superior to other approaches, since there are no constraints imposed on the network that restrict it in any way [12]. However, end-to-end learning requires a large amount of data [13], which is not always available. The aim of the ML-DDPG is, therefore, to outperform the DDPG when data is scarce without putting a constraint on the performance when data is abundant.

The algorithm is tested on several continuous control tasks, where for each task two challenging observation sets are defined. In one observation set, half of the inputs are "noise-states", which represent sensor measurements that have no relation to the task for which the agent receives rewards. For the second observation set, the system is assumed to be partially observable; the data set is therefore made up of a sequence of actions and measurements, where the sequence

J. Munk, J. Kober and R. Babuška are with the Delft Center for Systems and Control of Delft University of Technology, The Netherlands jellemunk@gmail.com, {j.kober, r.babuska}@tudelft.nl.

is intentionally chosen to be larger than necessary to guarantee the Markov property. We show that our algorithm is able to extract the information that is relevant for the task at hand from the observations, and so to improve the final policy learned by the agent.

The paper is organized as follows. Section II introduces RL, DDPG and state representation learning. Section III details the new algorithm, ML-DDPG. Simulation results are presented in Section IV and Section V concludes the paper.

II. Preliminaries

This work builds on earlier work from the RL community, specifically on the DDPG actor-critic algorithm introduced in [3], and on the concept known as state representation learning. The rest of this section explains this prior work in more detail.

A. Reinforcement Learning (RL)

In RL a learning agent interacts with an environment with the aim of maximizing the rewards received from the environment over time. A RL problem is modelled as a Markov Decision Process (MDP) described by the tuple $M = (S, A, f, r)$, where the state space S is a set of states $s \in \mathbb{R}^m$, the action space A is a set of actions $a \in \mathbb{R}^p$, $f : S \times A \to S$ is the state transition function, and $r : S \times A \to \mathbb{R}$ is the reward function. At each timestep t, the agent receives an observation $o_t \in \mathbb{R}^n$ that determines its current state s_t, it chooses an action a_t, receives a scalar reward $r_{t+1} \in \mathbb{R}$ according to the reward function r and transits to state s_{t+1} according to the transition function f.

The goal in RL is to learn a control policy $\pi : S \to A$ that maximizes the discounted sum of future rewards $R_t = \sum_{t=i}^{T} \gamma^{i-t} r(s_t, a_t)$ where $\gamma \in [0, 1]$ is the discount factor and T the number of time steps per learning episode.

The action-value function Q is often used in RL algorithms to denote the expected future reward given an action a_t taken in state s_t and thereafter following the policy π by taking the action $a^\pi = \pi(s)$. The Q function, in the form of a difference equation is given by

$$Q^\pi(s_t, a_t) = r_{t+1} + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1})).$$

B. Actor-Critic

In applications like robotics, where the state and action spaces are continuous, function approximators have to be used to approximate both the action-value function Q and the policy π [14]. Actor-critic algorithms are suitable in these situations since they allow both of these functions to be learned separately. This is in contrast with critic-only methods, which require a complicated optimization at every time step to find the policy.

In actor-critic methods, the critic learns the action-value function Q while the actor learns the policy π. In order to ensure that updates of the actor improve the expected discounted return, the update should follow the policy gradient [15]. The main idea behind actor-critic algorithms is that the critic provides the actor with the policy gradient. In theory, the critic should have converged before it can provide the actor with an unbiased estimate of the policy gradient, in practice however this requirement can be relaxed as long as the actor learns at a slower rate than the critic [15].

C. Deep Deterministic Policy Gradient (DDPG)

The DDPG algorithm is an off-policy actor-critic algorithm, first introduced in [3]. In this algorithm, both the actor and the critic are approximated by a DNN with parameter vectors ζ and ξ, respectively. The critic is trained by minimizing the squared Temporal Difference (TD) error

$$\mathcal{L}(\xi) = \Big(r_{t+1} + \gamma Q(s_{t+1}, \pi(s_{t+1}|\zeta)|\xi) - Q(s_t, a_t|\xi)\Big)^2.$$

The actor is updated in the direction of the policy gradient $\bigtriangledown Q_\zeta$ using the current approximation of the critic. The update of ζ with $\triangle\zeta$ is given by

$$\triangle\zeta = \bigtriangledown_a Q(s_t, \pi(s_t|\zeta)|\xi) \bigtriangledown_\zeta \pi(s_t|\zeta).$$

According to [16] the Q-function should be in the *compatible* form in order for the policy gradient to be unbiased. Although this is generally violated in the DDPG algorithm, with the addition of a few extra stability measures the algorithm has been shown to work well in practice.

A significant problem occurs when minimizing (II-C) [2]. The updates of the parameter ξ not only change the output of the critic network $Q(s_t, a_t|\xi)$, but they also change the target function $r_{t+1} + \gamma Q(s_{t+1}, \pi(s_{t+1}|\zeta)|\xi)$ that the network is learning. This is due to the recursive nature of the action-value function. Similarly, updates to the actor parameter ζ also change the target function. This coupling can lead to unstable behaviour and can cause the learning process of the action-value function approximation to diverge.

A solution, proposed in [3], that reduces the coupling between the target function and the actor and critic networks, is to update the parameters of the target function using "soft" updates. Instead of using ζ and ξ directly, a separate set of weights ζ^- and ξ^- are used, which slowly track the parameters ζ and ξ of the actor and critic networks constituting so called target networks.

The "soft" updates are performed after each learning step, using the following update rule

$$\zeta^- \leftarrow (1-\tau)\zeta^- + \tau\zeta, \qquad \xi^- \leftarrow (1-\tau)\xi^- + \tau\xi$$

where $\tau \in (0, 1]$ represents the trade-off between the learning speed and stability. Using these new parameters, the squared TD error becomes

$$\mathcal{L}_c(\xi) = \Big(r_{t+1} + \gamma Q(s_{t+1}, \pi(s_{t+1}|\xi^-)|\zeta^-) - Q(s_t, a_t|\xi)\Big)^2.$$

A second measure to ensure stable learning is to use an Experience Replay Database (ERD) [1]. Samples collected from the system are stored in this database such that they can be reused at a later stage. This is necessary because DNN are global approximators and are prone to catastrophic forgetting, i.e., the network forgets what it has learned in some part when updating some other part. In order to prevent catastrophic forgetting, a DNN should be trained with mini-batches, where the samples in a mini-batch are independently

Algorithm 1 DDPG

{Actor-Critic Learning}
Randomly initialize network weights ζ and ξ
$\zeta^- \leftarrow \zeta$ and $\xi^- \leftarrow \xi$ {set weights of target network}
for learning step = 1 **to** N **do**
  Sample random mini-batch from database
  Calculate $\triangle\zeta$ and $\mathcal{L}_c$ over mini-batch
  $\zeta \leftarrow \zeta - \alpha_a \triangle \zeta$ and $\xi \leftarrow \xi - \alpha_c \frac{\partial \mathcal{L}_c}{\partial \xi}$
  $\zeta^- \leftarrow (1-\tau)\zeta^- + \tau\zeta$ and $\xi^- \leftarrow (1-\tau)\xi^- + \tau\xi$ {update target network}
end for

and identically distributed. The ERD is necessary to create such mini-batches, although care should be taken to keep the data within the database varied enough to prevent it from over-fitting [17]. See Algorithm 1 for an overview of DDPG.

D. State Representation Learning

In RL, the state s is not always directly accessible, but needs to be constructed from a set of observations o. Such an observation-to-state map $f : O \rightarrow S$ can be the result of feature engineering, in which an engineer selects the observations and designs the mapping, but this can also be learned from data. The process of learning the observation-to-state mapping is called state representation learning [7].

State representation learning is a form of unsupervised learning, i.e., there are no training examples available since it is not known a priori what the most suitable state representation is to solve the problem. Learning an observation-to-state mapping therefore involves either making assumptions about the structure of the state representation or learning the mapping as part of learning some other function.

In [8], [18], [19], an auto-encoder is used to find an observation-to-state mapping in which the observations are compressed into a low-dimensional state vector. The objective, during training, is to find states from which the original observations can be reconstructed. The auto-encoder subsequently learns a state representation that captures only the unique features of the observation, i.e., how they differ from other observations.

Another unsupervised method is Slow Feature Analysis (SFA) [9], which is based on the idea that most phenomena in the world change slowly over time. In [20], [18] this assumption is used to learn a mapping between visual observations and a state representation that gradually changes over time.

In [7], these and several other assumptions about the structure of a good state representation are combined into the so-called Robotic Priors. They are divided into the simplicity prior, the temporal coherence prior, the proportionality prior, the causality prior and the repeatability prior. For each of these priors, a loss function is defined. An observation-to-state mapping is subsequently trained to minimize the combined loss functions of the individual priors. The paper then shows a performance increase when using the learned state representation instead of the raw observations as input to the Neural Fitted Q-iteration algorithm [1].

III. Model Learning Deep Deterministic Policy Gradient (ML-DDPG)

A DNN approximates a function by learning a transformation from an input vector to a feature representation, that can be linearly combined in the output layer, to a target function [21]. Viewed in this way, an internal signal, between two layers of a DNN represents some intermediate representation that is somewhere between the original input and the final feature representation. The main idea behind the ML-DDPG approach is that that up to a certain point, an actor and critic network, can benefit from sharing their intermediate representation. Furthermore, we argue that this intermediate representation can be learned more effectively by a third (model learning) network. In the rest of this paper, we refer to this intermediate representation as the state and the transformation of the input to this state, as an observation-to-state mapping.

The ML-DDPG architecture consist of three DNNs, a model network, a critic network and an actor network. The model network is trained by using a new concept that we call *predictive priors* and is integrated with the actor and critic networks by copying some of its weights. In the experiments, the creation of the ERD, learning the model network and training the actor and critic is done in separate steps. This allows us to train both the DDPG and the ML-DDPG on exactly the same dataset and it simplifies the training of the individual layers of a DNN. Ultimately, however, the goal is to train the model network simultaneously with the actor and critic networks and to create the ERD while learning, as in the original DDPG.

A. Predictive priors

The *predictive priors* consist of two separate priors. The first prior is the *predictable transition prior* which states that, given a certain state s_t and an action a_t taken in that state, one can predict[1] the next state $\hat{s}_{t+1}$. An important difference with other methods like [22], [10], is that we do not predict the next observation $\hat{o}_{t+1}$ but the next state $\hat{s}_{t+1}$. This becomes important if the observation o_t contains task-irrelevant information. A state that needs to be able to predict the next observation still has to contain this task-irrelevant information to make the prediction, whereas in the proposed case this information can be ignored altogether. The second prior is the *predictable reward prior* which states that, given a certain state s_t and an action a_t taken in that state, one can predict the next reward $\hat{r}_{t+1}$. This prior enforces that all information relevant to the task is available in the state, which helps the *predictable transition prior* to converge to a meaningful representation for the given task.

The *predictive priors* approach essentially enforces the two elementary properties of an MDP. The relevant information is, however, already present in the original observation. The

[1] Values predicted by models are denoted with $\hat{\cdot}$.

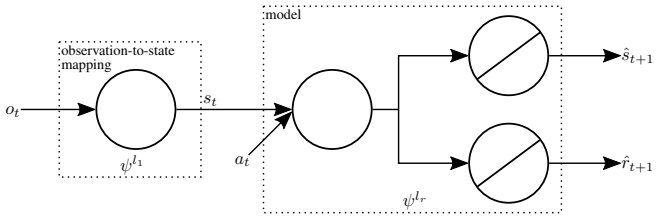


Fig. 1: Network architecture of the model network learning an observation-to-state mapping using the predictive priors.

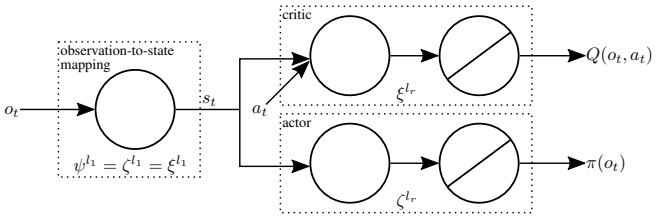


Fig. 2: Actor and critic networks integrating the observation-to-state mapping learned in the model network.

point of using the predictive priors is to find a state representation from which it is easier, i.e., fewer transformations are necessary, to actually learn the transition and reward function.

The advantage of using the *predictive priors* is that state representation learning is transformed from an unsupervised learning problem to a supervised learning problem. A single interaction with the system produces a sample $\{o_t, a_t, r_{t+1}, o_{t+1}\}$. The observation o_{t+1} can be mapped to s_{t+1} using the current approximation of the observation-to-state mapping. This gives us both the inputs o_t and a_t as well as the target values r_{t+1} and s_{t+1}. Given a set of such samples and some function approximator, supervised learning can be used to find these mappings.

Another advantage of this approach is that the state representation that is learned is goal directed. Observations that do not correlate with the reward or are inherently unpredictable will not be encoded in the state representation. This is in contrast to methods like an auto-encoder or SFA since these methods do not differentiate between observations that are useful for solving a particular task and observations that are not.

B. Model network

The *predictive priors* are implemented by a model network that learns to predict the next state and reward $\{\hat{s}_{t+1}, \hat{r}_{t+1}\}$ from the observation-action tuple $\{o_t, a_t\}$. The architecture is shown in Figure 1. Each circle in the image represent a single layer containing multiple neurons, the lines are n-dimensional signals. The observation-to-state mapping is encoded in the first layer of the network. It has observation o_t as input and state s_t as output. This state, together with the action a_t form the input to the second layer. Finally, two parallel linear output layers produce a prediction of the next state $\hat{s}_{t+1}$ and the reward $\hat{r}_{t+1}$ respectively.

The network is trained by minimizing the following objective function

$$\mathcal{L}_m = \|s_{t+1} - \hat{s}_{t+1}\|_2^2 + \lambda_m \|r_{t+1} - \hat{r}_{t+1}\|_2^2$$

where λ_m represents the trade-off between predicting the reward and the next state. Note that, to obtain the target s_{t+1}, the current approximation of the observation-to-state mapping is used, to map the next observation o_{t+1} to the next state s_{t+1}. This could potentially lead to convergence problems, since the target depends on the current approximation. In practice, however, these problems did not occur. In all experiments, which all started from different random initial conditions, the learning converged to similar local optima.

Algorithm 2 ML-DDPG

{Model learning}
Randomly initialize network weights ψ, ζ and ξ
for pre-training step = 1 **to** M **do**
  Sample random mini-batch from database
  Calculate $\mathcal{L}_m$ over mini-batch
  $\psi \leftarrow \psi - \alpha_m \frac{\partial \mathcal{L}_m}{\partial \psi}$
end for
{Actor-Critic Learning}
$\zeta^{l_1} \leftarrow \psi^{l_1}$ and $\xi^{l_1} \leftarrow \psi^{l_1}$ {copy weights to actor and critic}
$\zeta^- \leftarrow \zeta$ and $\xi^- \leftarrow \xi$ {set weights of target network}
for learning step = 1 **to** N **do**
  Sample random mini-batch from database
  Calculate $\mathcal{L}_a$ and $\mathcal{L}_c$ over mini-batch
  $\zeta^{l_r} \leftarrow \zeta^{l_r} - \alpha_a \frac{\partial \mathcal{L}_a}{\partial \zeta^{l_r}}$ and $\xi^{l_r} \leftarrow \xi^{l_r} - \alpha_c \frac{\partial \mathcal{L}_c}{\partial \xi^{l_r}}$
  $\zeta^- \leftarrow (1-\tau)\zeta^- + \tau\zeta$ and $\xi^- \leftarrow (1-\tau)\xi^- + \tau\xi$ {update target network}
end for

C. Integrating the model network

The model network is trained first, before the other two networks. Afterwards the observation-to-state mapping, that is encoded in the first layer of the model network, is copied to the first layer of, both the actor and the critic network. The parameter vectors ψ, ζ and ξ of the model, actor and critic respectively are therefore split in two parts where, ψ^{l_1}, ζ^{l_1} and ξ^{l_1} represent the weights of the first layer and ψ^{l_r}, ζ^{l_r} and ξ^{l_r} the weights of the remaining layers. Figure 2 shows how the actor and critic networks use the same observation-to-state mapping that is learned by the model network.

After pre-training, i.e., the model learning, the weights of the first layers ζ^{l_1} and ξ^{l_1} are fixed. Subsequently, standard actor-critic is used to learn the weights in the remaining layers of the actor ζ^{l_r} and critic ξ^{l_r}, see Algorithm 2.

D. Saturation penalty

One specific problem we encountered, with both the DDPG and the ML-DDPG, was the fact that the actor sometimes learned actions that lay outside the saturation limits of the actuator. This is caused in part because all the samples from which the networks learn are collected prior to learning. If an agent learns actions outside the range,

TABLE I: Dimension table for the two benchmarks

	Action	Internal state	$o^{\text{unrelated}}$	$o^{\text{redundant}}$
2-link arm	2	6	18	24
Octopus	36	96	192	308

in which data was originally collected, the policy gradient, evaluated at these actions, is based on extrapolating the critic network, which for large deviations is very unreliable. This creates instability issues in both networks and hampers the convergence of the algorithm.

The loss function of the actor corresponds to the Q-function. In order to restrict the action space a *saturation penalty* is added and the loss function becomes

$$\mathcal{L}_a(\zeta) = -Q(s, \pi(s|\zeta)) + \lambda_a \Big(\max(\pi(s|\zeta) - 5, 0) + \max(-\pi(s|\zeta) - 5, 0) \Big)^2$$

where λ_a represents the trade-off between maximizing the reward and minimizing the saturation penalty. The actions are scaled such that they have zero mean and a standard deviation of 1, which puts the saturation limit at 5 times the standard deviation of the original exploration policy.

IV. Simulation Results

In order to compare the performance of the ML-DDPG algorithm with the DDPG algorithm, they are both applied to the 2-link arm problem from [17] and the octopus problem from [23]. In both benchmarks the state is not directly available, instead there are two types of observations:

1) $o^{\text{redundant}}$ - Includes a sequence of actions and measurements of a *partially observable* system. It is called redundant since the length of the sequence is chosen larger than necessary to give the observation vector the Markov property.
2) $o^{\text{unrelated}}$ - Includes the full state extended by a vector of white noise inputs.

Table I shows the dimensions of the action space, the internal state and the two observation types for both benchmarks. For both ML-DDPG and DDPG an ERD, batch-normalization, an L2 penalty $\lambda_c = 0.002$ on the critic weights and "soft" updates of the target networks with $\tau = 10^{-3}$ are used to stabilize the learning. Adam [24] is used for learning the weights of all three DNNs with a base learning rate of $\alpha_m = 10^{-3}$, $\alpha_a = 10^{-4}$ and $\alpha_c = 10^{-3}$ for the model, actor and critic respectively. The hidden layers of all three networks contain 100 neurons each. $\lambda_a = 50$ and $\lambda_m = 10$ and $\gamma = 0.99$.

Before the algorithms are run, data is collected by following a random policy based on the Ohrnstein-Uhlenbeck process [25]. The algorithms are then trained off-policy, on the same database. A single learning step consists of a single update of the weights of the actor and the critic, each experiment consists of 40000 learning steps for the 2-link arm problem and 30000 learning steps for the octopus. Every 100 learning steps the policy π is evaluated using a pre-defined reference signal. The performance of the two algorithms is evaluated for both types of observations. Furthermore, the experiments are repeated for different sizes of the ERD, to compare how the algorithms perform when data is either scarce or abundant.

In order to make a quantitative comparison between the learning curves, the settling time τ_s, rise time τ_{rise} and the average performance $\bar{R}$ are calculated, see Appendix. In all experiments, the saturation penalty described in Section III-D, was a necessary condition for the algorithms to converge.

A. 2-link arm

The 2-link arm consists of two links, each controlled by a motorized joint. The angle of the first link $\theta_1 \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ is measured with respect to the downward position and the angle of the second link $\theta_2 \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ with respect to the first link. The two motorized joints can be controlled by setting a_1 and a_2, which are (a scaled version of) the motor voltages. Finally the reference position $p^{\text{ref}} = \begin{bmatrix} x^{\text{ref}} & y^{\text{ref}} \end{bmatrix}$ determines the desired position of the tip of the second link.

The reward from the environment is based on the Euclidian distance D between the reference position and the current position of the tip of the second link and on the angular velocities $\dot{\theta}$ of the two links. The reward function is given by

$$r(D, \dot{\theta}) = -\Big(D + w|\dot{\theta}|_2 \Big)$$

where w represent the trade-off between the two terms.

The observation vector $o^{\text{redundant}}$ contains the current angles θ and the angles and actions from the previous 5 timesteps as well as the reference:

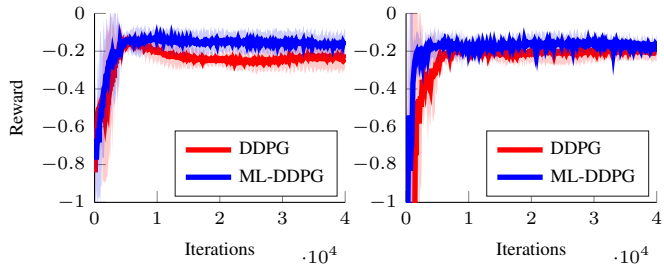
$$o_t^{\text{redundant}} = \begin{bmatrix} \theta_t & .. & \theta_{t-5} & a_{t-1} & .. & a_{t-5} & p_t^{\text{ref}} \end{bmatrix}$$

where θ_t is a vector of the two angles at time instance t, a_t a vector of the two control actions and p_t^{ref} the Cartesian reference position at timestep t. The observation vector $o^{\text{unrelated}}$ contains the current angles θ, the angular velocity $\dot{\theta}$, a vector of unrelated white noise inputs e_t and the reference position p_t^{ref}:

$$o_t^{\text{unrelated}} = \begin{bmatrix} \theta_t & \dot{\theta}_t & e_t & p_t^{\text{ref}} \end{bmatrix}.$$

Figure 3 shows the mean (thick line) and standard deviation (shaded area) of the learning curve for both observation vectors using an ERD of 30K samples. The results from the other experiments are presented in Table II. For the 2-link arm benchmark, the ML-DDPG outperforms the DDPG algorithm in final performance $\bar{R}$ ($+37.8\%$ on $o^{\text{unrelated}}$ and $+8.1\%$ on $o^{\text{redundant}}$) and in rise time τ_{rise} -29% and settling time τ_s -36.2% on the $o^{\text{unrelated}}$ observation type. It does have slower convergence on the $o^{\text{redundant}}$ type (rise time τ_{rise} $+28.7\%$ and settling time τ_s $+39\%$). Both algorithms perform better, in terms of the final performance, if more data is available. The advantage of the ML-DDPG over the DDPG seems relatively constant and does not degrade when data becomes abundant as was expected.

Figure 4 shows a time-domain plot of the x-coordinate of the tip of the second link and (one of the) accompanying control actions. It is clear that the performance is incomparable



(a) $o^{\text{unrelated}}$ (b) $o^{\text{redundant}}$

Fig. 3: Learning curve 2-link arm using 60K samples

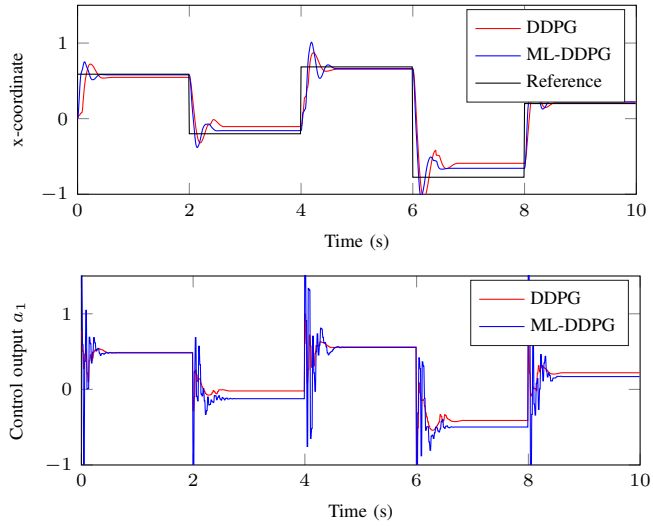


Fig. 4: Time domain plot of the final policy on the 2-link arm benchmark, showing one of the states (top) and control actions (bottom).

to other optimal control methods, the controlled system has a steady-state error and a significant overshoot. It is important to note, however, that the controller does not use a separate observer, although the system is partially observable and that the reference is given in Cartesian coordinates whereas the position of the 2 links are given in joint angles. The controller, therefore, needs to learn the non-linear mapping between the two, while also learning the unobservable states from a sequence of measurements. Seen in this light, we think the performance is actually quite good. We also believe the performance can be further improved by tuning the reward function and/or the architecture of the DNNs which has not been done extensively to get these results.

B. Octopus

The octopus arm [23] consists of 13 segments, each of which contains multiple muscles. The muscles can be controlled by specifying their stiffness $a \in [0, 1]$. The arm is attached to a base at one side and moves in a 2-dimensional plane. The task is to reach to a randomly placed food target. The task is completed if it touches the food target with any part of the arm.

The reward from the environment is based on the Euclidean distance D between the food and the segment that is closest to the food. Whenever the goal is reached an extra bonus B is given. The reward function is given by

$$r(D, B) = (B - 2) - D$$

where $B = 2$ whenever the goal is reached and $B = 0$ otherwise.

As in the 2-link arm benchmark two observation vectors are defined $o^{\text{redundant}}$, in which the velocity information is replaced by positions and actions at previous time instances, and $o^{\text{unrelated}}$, in which the state is extended with a vector of white noise inputs.

The results on the Octopus benchmark are not as conclusive as with the 2-link arm. Both algorithms have a similar rise and settling time and are able to learn the task in about 2000 learning steps. Both successfully learn to reach for the food, which takes them around 1.5s from their starting position. In order to see if the learned policy also generalized to other initial positions, the octopus arm was randomly excited for 2s before testing the learned policy again. Also, in these cases, the octopus was successful in reaching the food. Perhaps in spite of the high dimensionality of the problem, the Octopus problem is relatively easy since it does not require a very precise control action, like in the case of the 2-link arm.

TABLE II: Learning curve characteristics on the 2-link arm for different sizes of the ERD. The rise time τ_{rise} and settling time τ_s are denoted in $\times 1000$ learning steps.

Input type	DB size	DDPG			ML-DDPG		
		τ_{rise}	τ_s	$\bar{R}$	τ_{rise}	τ_s	$\bar{R}$
$o^{\text{unrelated}}$	15K	3	**3.2**	−0.22	**2**	3.5	**−0.18**
	30K	3	8.6	−0.24	**2.8**	**3.8**	**−0.16**
	90K	3.4	3.7	−0.19	**1.9**	**2.6**	**−0.13**
$o^{\text{redundant}}$	15K	**2.5**	**11.6**	**−0.26**	3.7	17.2	−0.28
	30K	4.8	5.1	−0.21	**1.9**	**2.5**	**−0.18**
	90K	**5.9**	**6.2**	−0.19	11.3	12.2	**−0.15**

V. Conclusion

This paper introduces a new algorithm, the ML-DDPG, that trains a model network using the *predictive priors* before learning the RL task. To the best of our knowledge, this was the first time that the DNNs of a RL algorithm were trained in two-stages, using the prediction error as a training signal. A big advantages of using the *predictive priors* is that it does not require the agent to follow a specific policy and/or exploration strategy. Hence the agent can learn from any previously created ERD. Furthermore, the DNN is trained using supervised learning as opposed to the approach used in [7], where unsupervised learning was used.

Learning an observation-to-state mapping, before training the actor and critic networks, has been shown to increase the final performance on a 2-link arm benchmark, primarily when the observations contain a large amount of unrelated inputs, for the task at hand. Even on a relatively large dataset it has been experimentally shown that this approach

outperforms an approach in which the DNNs are trained end-to-end, which suggests that the latter method is not always preferable as is claimed in [12].

Results in this paper confirm that using DNNs in actor-critic algorithms, is a very promising field of research, especially for cases in which the state and action dimensions of the problem are very high. More work is necessary to visualise what kind of state representation the ML-DDPG is actually learning and how it performs on other benchmarks. Other future work will try to answer the more general question of how DNNs seem to escape the curse of dimensionality.

APPENDIX

To define the settling time and the rise time of the learning curve, first introduce the undiscounted return after j learning steps averaged over the number N_e of learning experiments:

$$\bar{R}_j = \frac{1}{N_e} \sum_{l=1}^{N_e} \sum_{t=0}^{T} r(s_t, a_t)$$

where T is the duration of the evaluation, referring to the jth learning step within the lth learning experiment. For each reward, the sequence $\bar{R}_1, \bar{R}_2, \ldots, \bar{R}_{N_t}$ is normalized so that the minimum value of this sequence is -1.

The performance $\bar{R}_f$ at the end of learning is defined as the average normalized undiscounted return in the last c learning steps:

$$\bar{R}_f = \frac{1}{c} \sum_{j=N_t-c+1}^{N_t} \bar{R}_j$$

The settling time τ_s of the learning curve is then defined as the number of learning steps after which the learning curve enters and remains within a band ϵ of the final value $\bar{R}_f$:

$$\tau_s = T_t \cdot \arg\max_j (|\bar{R}_f - \bar{R}_j| \geq \epsilon \bar{R}_f)$$

In this paper c and ϵ are set to 1000 and 0.05 respectively. The rise time is defined as the number of learning steps required to climb from the 10% performance level to the 90% performance level:

$$\tau_{\text{rise}} = \tau_{90} - \tau_{10}$$

with τ_p defined as:

$$\tau_p = T_t \cdot \arg\max_j \left(\frac{\bar{R}_j - \bar{R}_1}{\bar{R}_f - \bar{R}_1} \geq \frac{p}{100} \right)$$

for $p = 10\%$ and 90%.

REFERENCES

[1] M. Riedmiller, "Neural fitted Q iteration - first experiences with a data efficient neural reinforcement learning method," in *European Conf. on Machine Learning (ECML)*, 2005.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[3] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[4] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Trans. on Systems, Man and Cybernetics*, no. 5, pp. 834–846, 1983.

[5] P. Wawrzyński, "Real-time reinforcement learning by sequential actor–critics and experience replay," *Neural Networks*, vol. 22, no. 10, pp. 1484–1497, 2009.

[6] D. M. Wolpert, J. Diedrichsen, and J. R. Flanagan, "Principles of sensorimotor learning," *Nature Reviews Neuroscience*, vol. 12, no. 12, pp. 739–751, 2011.

[7] R. Jonschkowski and O. Brock, "State representation learning in robotics: Using prior knowledge about physical interaction," in *Robotics: Science and Systems (R:SS)*, 2014.

[8] S. Lange, M. Riedmiller, and A. Voigtlander, "Autonomous reinforcement learning on raw visual input data in a real world application," in *Int. Joint Conf. on Neural Networks (IJCNN)*, 2012.

[9] L. Wiskott and T. J. Sejnowski, "Slow feature analysis: Unsupervised learning of invariances," *Neural computation*, vol. 14, no. 4, pp. 715–770, 2002.

[10] N. Jetchev, T. Lang, and M. Toussaint, "Learning grounded relational symbols from continuous data for abstract reasoning," in *Workshop on Autonomous Learning, Int. Conf. on Robotics and Automation (ICRA)*, 2013.

[11] U. Muller, J. Ben, E. Cosatto, B. Flepp, and Y. L. Cun, "Off-road obstacle avoidance through end-to-end learning," in *Advances in Neural Information Processing Systems (NIPS)*, 2005.

[12] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *Journal of Machine Learning Research*, vol. 17, no. 39, pp. 1–40, 2016.

[13] W. Böhmer, J. T. Springenberg, J. Boedecker, M. Riedmiller, and K. Obermayer, "Autonomous learning of state representations for control: An emerging field aims to autonomously learn state representations for reinforcement learning agents from their real-world sensor observations," *KI-Künstliche Intelligenz*, vol. 29, no. 4, pp. 353–362, 2015.

[14] L. Buşoniu, R. Babuška, B. De Schutter, and D. Ernst, *Reinforcement learning and dynamic programming using function approximators*, vol. 39. CRC Press, 2010.

[15] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems (NIPS)*, 2000.

[16] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *Int. Conf. on Machine Learning (ICML)*, 2014.

[17] T. de Bruin, J. Kober, K. Tuyls, and R. Babuška, "The importance of experience replay database composition in deep reinforcement learning," in *Deep Reinforcement Learning Workshop, Advances in Neural Information Processing Systems (NIPS)*, 2015.

[18] C. Finn, X. Y. Tan, Y. Duan, T. Darrell, S. Levine, and P. Abbeel, "Learning visual feature spaces for robotic manipulation with deep spatial autoencoders," *arXiv preprint arXiv:1509.06113*, 2015.

[19] N. Wahlström, T. B. Schön, and M. P. Deisenroth, "Learning deep dynamical models from image pixels," in *IFAC-PapersOnLine*, 2015.

[20] R. Legenstein, N. Wilbert, and L. Wiskott, "Reinforcement learning on slow features of high-dimensional input streams," *PLoS Comput Biol*, vol. 6, no. 8, p. e1000894, 2010.

[21] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[22] I. Grondman, M. Vaandrager, L. Buşoniu, R. Babuška, and E. Schuitema, "Efficient model learning methods for actor-critic control," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 42, no. 3, pp. 591–602, 2012.

[23] Y. Engel, P. Szabo, and D. Volkinshtein, "Learning to control an octopus arm with Gaussian process temporal difference methods," in *Advances in Neural Information Processing Systems (NIPS)*, 2005.

[24] D. P. Kingma and J. L. Ba, "Adam: a method for stochastic optimization," in *Int. Conf. on Learning Representations (ICLR)*, 2015.

[25] G. E. Uhlenbeck and L. S. Ornstein, "On the theory of the Brownian motion," *Physical review*, vol. 36, no. 5, pp. 823–841, 1930.