Masaryk University
Faculty of Informatics

Model-Based Deep Reinforcement Learning with Continuous Actions

Master's Thesis

Bc. Karol Kuna

Brno, Spring 2017

MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Model-Based Deep Reinforcement Learning with Continuous Actions

MASTER'S THESIS

Bc. Karol Kuna

Brno, Spring 2017

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Karol Kuna

Advisor: doc. RNDr. Tomáš Brázdil, Ph.D.

Acknowledgement

I would like to thank my advisor, doc. RNDr. Tomáš Brázdil, Ph.D., for all his help and for pointing me in the right direction.

Abstract

We present a novel model-based actor-critic off-policy deep reinforcement learning algorithm for learning deterministic policies in environments with a continuous action domain. We examine the benefits of using a learned model of the environment and compare it to a contemporary model-free approach. Reinforcement learning agents are tested on several robotic control tasks from OpenAI Gym. Methods are compared in terms of their sample efficiency and ability to transfer knowledge from one problem to another. Finally, we provide a working implementation of the tested methods.

Keywords

deep reinforcement learning, model-based reinforcement learning, actor-critic, deep learning, OpenAI gym, control, continuous actions

Contents

List of Figures

1 Introduction

1.1 Motivation

Reinforcement learning [1] is a branch of artificial intelligence that deals with learning agent's behavior solely from interaction with an environment. We use a formulation of the reinforcement learning problem where agent interacts with the environment sequentially in discrete time steps. In each step, the agent observes current state of the environment and decides to take an action according to its policy. The environment then transitions to next state and provides feedback in a form of reward signal. Agent's objective is to maximize cumulative reward received during the entire interaction. The goal of reinforcement learning is finding an action-selection policy that achieves this objective.

Reinforcement learning has been combined with deep neural networks into "deep reinforcement learning" [2]. Using advances in deep learning [3], these methods can learn policies directly from raw input without feature engineering. Recently, this approach has been successfully applied to many challenging domains, including playing Atari 2600 games at human level [4] and defeating human champions in the game Go [5]. These games are typically played with discrete actions from a finite domain. On the other hand, many robot control tasks require continuous actions, e.g. for modulating precise amount of torque applied to joint actuators. In this more general setting, classical optimization techniques are intractable due to the infinite number of actions. Commonly, these tasks are solved using actor-critic methods, which learn a continuous policy (actor) using feedback from the critic. Deep Deterministic Policy Gradient (DDPG) [6] is an actor-critic method based on the policy gradient [7] which was successfully applied to multiple challenging control tasks.

Model-free methods, such as DDPG, don't learn an explicit model of the environment and tend to require huge amounts of training data to learn successful policies. On the contrary, model-based techniques may often be more sample efficient by utilizing learned environment model [8, 9]. We are primarily concerned with sample efficiency of the algorithms, i.e. the number of interactions with an environment before a successful policy is learned. This is often a limiting factor for practical use of these algorithms.

We developed a novel model-based deep reinforcement learning algorithm called Deep Model Learning Actor-Critic (DMLAC). Unlike other model-based approaches [9, 8, 10] which use a model to generate additional training data, we make the model an integral part of the learning process and actor-critic architecture. We show that incorporation of a model leads to an increase in sample efficiency when learning new tasks and transferring knowledge from one task to another.

1.2 Thesis Organization

In the second chapter, we introduce the reinforcement learning formalisms and define the common notation. This chapter serves as a preliminary for further chapters which extend the discussed concepts.

The third chapter gives an overview of deep reinforcement learning and how neural networks are applied to it. Additionally, we describe DDPG algorithm which is later experimentally compared to our method. This chapter can be too considered as a preliminary.

In the fourth chapter, we present our contribution in the form of the novel DMLAC method. We propose theoretical model-based actor-critic learning process and give pseudocode of the algorithm.

In the fifth chapter, we describe our implementation of DMLAC and DDPG using the machine learning library TensorFlow [11].

The sixth chapter presents performed experiments and gives experimental evaluation of our model-based method compared to common model-free alternative of DDPG. We test the methods in environments with continuous actions selected from OpenAI Gym [12].

We conclude the results in the final seventh chapter.

2 Reinforcement Learning

2.1 Setting

We consider a standard reinforcement learning (RL) setting [1, 12] where an agent interacts with an environment sequentially in discrete time steps $t = 0, 1, 2, \ldots$ We model the reinforcement learning problem as an extension of a deterministic Markov Decision Process with a continuous state space $S \subseteq \mathbb{R}^N$ and an action space $A \subseteq \mathbb{R}^M$. An arbitrary state $s \in S$ and action $a \in A$ are real-valued vectors of dimensions N and M respectively.

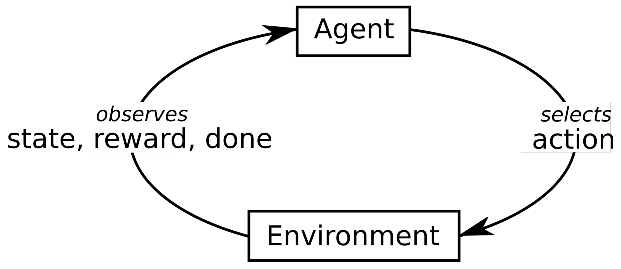

Figure 2.1: Interaction of agent and environment

Environment In general, environments may be stochastic, but we limit ourselves to the deterministic case which is more suitable for our purposes. We assume that the environment satisfies the Markov property and is fully observable [1]. Therefore, its one-step dynamics can be described by a state transition function $T : S \times A \to S$, a reward function $r : S \times A \to \mathbb{R}$, and a terminal condition $done : S \to \{0, 1\}$. An initial state s_0 of the environment is selected randomly according to a probability distribution $p(s_0)$. In time step t, the agent observes current state s_t of the environment and selects action a_t to perform. The environment then transitions to next state $s_{t+1} = T(s_t, a_t)$ and awards the agent with reward $r_{t+1} = r(s_t, a_t)$. The interaction between the agent and environment ends after reaching a terminal state s where the terminal condition $done(s) = 1$ is satisfied.

Agent The goal of an agent is to select actions so as to maximize the cumulative reward received during the entire interaction with an environment [1]. Agent's behavior is prescribed by a *policy* which selects an action from the action space based on the current state. In general, the policy may be stochastic. Stochastic policy $\pi : S \rightarrow \mathrm{P}(A)$ maps agent's state space to a probability distribution over actions. In an arbitrary state s, the agent selects action a according to the distribution density $\pi(s)$. In this thesis, we are mainly concerned with learning deterministic policies, a special case of stochastic policies. Deterministic policy $\mu : S \rightarrow A$ is a function that maps states directly to actions. In this case, the agent selects action $a = \mu(s)$ in an arbitrary state s. Since many control tasks don't require stochastic policies to be solved, deterministic policies are particularly useful due to their desirable properties in policy gradient estimation (section 2.6.2) [7].

2.2 Episodic and Continuing Tasks

In addition to the current state and reward, environments also provide a binary signal $done_t = done(s_t)$ informing the agent whether time step t is final [1]. Upon reaching a terminal state s_t in the final step t, the interaction between the agent and environment is concluded. We call such interaction an *episode* with duration t and the task *episodic*. On the other hand, tasks that have no terminal states and continue forever are called *continuing* tasks.

In order to unify notation between both types of tasks, we transform episodic tasks into continuing [1]. We add a special *absorbing state* into the state space. After reaching a terminal state, all further transitions lead to the absorbing state no matter what action is taken. The agent receives zero reward for these transitions. Since the sum of all rewards in the original and the modified task is the same, they can be treated as equivalent for the purposes of reinforcement learning.

2.3 Expected Discounted Return

Reinforcement learning is the process of finding a policy that maximizes some notion of a cumulative reward. Commonly, the concept of discounted return is introduced [1]. Discount factor $\gamma \in \langle 0, 1)$ is used to scale down the importance of future rewards. Typical values of γ are approaching 1. We define discounted return R_t as the discounted sum of all rewards from time step t till infinity:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \ldots = \sum_{i=t}^{\infty} \gamma^{i-t} r_{i+1} \tag{2.1}$$

We assume that the rewards are bounded and discount factor $\gamma < 1$, therefore R_t exists and is finite [1]. Notably, the discounted return depends on the environment and the policy used to select actions. In general, they may be stochastic and for this reason, RL typically searches for policies that maximize the expected discounted return $\mathbb{E}_{E,\pi}[R_t]$ dependent on the environment E and the policy π.

2.4 Value Functions

To an agent that maximizes its cumulative reward, some states are more valuable than others. We formulate this value of a state as the expected discounted return that the agent expects to get by following some policy from this state onwards [1]. We define this relationship as a value function $V : S \to \mathbb{R}$. Value function maps each state s_t to the expected discounted return R_t, assuming that the agent follows a policy μ in an environment E:

$$V^{\mu}(s_t) = \mathbb{E}_{E,\mu}[R_t | s_t] \tag{2.2}$$

Alternatively, value function can be defined recursively as a sum of the immediate reward r_{t+1} and discounted value of the next state s_{t+1}, also known as the Bellman equation [13]

$$V^{\mu}(s_t) = \mathbb{E}_{E,\mu}\big[r_{t+1} + \gamma V^{\mu}(s_{t+1})\big] \tag{2.3}$$

Similarly, we can estimate value of an arbitrary state-action pair (s_t, a_t) as the expected discounted return of taking the action a_t (independent of agent's policy) in the state s_t and then following the agent's policy μ from the next state s_{t+1} onward. Formally, action-value function $Q : S \times A \to \mathbb{R}$ is defined as

$$Q^{\mu}(s_t, a_t) = \mathbb{E}_{E,\mu}[R_t | s_t, a_t] \tag{2.4}$$

Analogously to the value function, action-value function Q^μ can be defined recursively, too:

$$Q^\mu(s_t, a_t) = \mathbb{E}_{E,\mu}\big[r(s_t, a_t) + \gamma Q^\mu(s_{t+1}, \mu(s_{t+1}))\big] \tag{2.5}$$

Optimal value function V^* maps states to maximum expected discounted return obtainable by any policy [1]

$$V^*(s_t) = \max_\mu V^\mu(s_t) \tag{2.6}$$

Optimal action-value function Q^* assigns maximum expected discounted return of any policy to state-action pairs

$$Q^*(s_t, a_t) = \max_\mu Q^\mu(s_t, a_t) \tag{2.7}$$

$$Q^*(s_t, a_t) = \mathbb{E}_E\big[r(s_t, a_t) + \gamma \max_a Q^*(s_{t+1}, a)\big] \tag{2.8}$$

2.5 Temporal-Difference Learning

In reinforcement learning, an agent is trying to optimize its policy to maximize the expected discounted return. Typically, the agent also learns a value or an action-value function which are used to improve its policy. Since we are dealing with continuous state and action spaces, we use function approximators to estimate these functions. We parametrize such approximators with parameters Θ which determine the function that the approximator computes. The goal of the learning process is to find parameters of the approximator that closely estimate the real function [1].

Considering that we are dealing with continuing tasks, sampling the real value function from an infinite sequence of future rewards is infeasible. Temporal-difference (TD) learning [14] is a method that solves this problem by bootstrapping a previously learned estimate to learn a new better estimate of a function. TD can be used to incrementally improve the estimate of a value function from past experience of interactions with the environment.

2.5.1 One-Step Temporal-Difference

TD learning [1] requires experience of a policy μ interacting with an environment E. In the simplest one-step version called TD(0), the experiences are one-step transitions between two consecutive steps $t, t+1$. We would like the estimated value $V^{\mu}_{\Theta}(s_t)$ of state s_t to approach the true expected discounted return $\mathbb{E}_{E,\mu}[R_t|s_t]$ from this state. As was already discussed in the previous section, this return is unavailable. Instead, TD(0) approximates it with one-step target return backup R^1_t consisting of the immediate reward r_{t+1} and the bootstrapped value estimate $V^{\mu}_{\Theta}(s_{t+1})$ of the next state's value.

$$R^1_t = r_{t+1} + \gamma V^{\mu}_{\Theta}(s_{t+1}) \tag{2.9}$$

TD error δ_t is the difference between the one-step return backup and the old value estimate.

$$\delta_t = R^1_t - V^{\mu}_{\Theta}(s_t) \tag{2.10}$$

The same ideas can be applied to estimation of an action-value function Q^{μ} with a parametrized approximator Q^{μ}_{Θ}

$$R^1_t = r_{t+1} + \gamma Q^{\mu}_{\Theta}(s_{t+1}, a_{t+1}) \tag{2.11}$$

$$\delta_t = R^1_t - Q^{\mu}_{\Theta}(s_t, a_t) \tag{2.12}$$

Once the TD error is known, we can change the parameters of the approximator to reduce it. We define a loss function L_t for the current transition as a mapping from the approximator's parameters to squared value of TD error.

$$L_t(\Theta) = \frac{1}{2}\delta^2_t \tag{2.13}$$

Assuming that the approximator is differentiable, we can use gradient descent methods to update its parameters in the direction that minimizes loss $L_t(\Theta)$. We ignore the fact that the target return R^1_t in TD error also depends on parameters Θ and treat it as a constant [6]. Gradient descent is discussed in more detail in section 3.1.1.

2.5.2 Multi-Step Temporal-Difference

TD learning is not limited to its one-step variant, which updates the value estimate based on only one transition. Let's consider an experience of n transitions from steps $t \dots t+n$. Discounted sequence of received rewards $r_t \dots r_{t+n}$ together with the bootstrapped value estimate $V^{\mu}_{\Theta}(s_{t+n})$ of the last state s_{t+n} gives us a better estimate of the true value function [1]. We define n-step target return R^n_t as

$$R^n_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \gamma^{n-1} r_{t+n} + \gamma^n V^{\mu}_{\Theta}(s_{t+n}) \qquad (2.14)$$

$$R^n_t = \sum_{i=1}^{n} \gamma^{i-1} r_{t+i} + \gamma^n V^{\mu}_{\Theta}(s_{t+n}) \qquad (2.15)$$

Analogously to one-step TD error, n-step TD error δ^n_t is defined as the difference between the target return and the old value estimate

$$\delta^n_t = R^n_t - V^{\mu}_{\theta}(s_t) \qquad (2.16)$$

In the most general case, a return backup can be an average of many n-step returns [1]. Algorithm TD(λ) uses λ-return R^{λ}_t defined as the average of all n-step backups discounted by factor $\lambda \in \langle 0, 1 \rangle$

$$R^{\lambda}_t = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R^n_t \qquad (2.17)$$

2.5.3 On-Policy, Off-Policy Methods

There is an important distinction between how different RL methods learn from experience. We distinguish *on-policy* and *off-policy* methods. Let's suppose that we use a policy π to interact with an environment and generate one-step transitions $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ from two consecutive steps. Off-policy methods are able to use these transitions to learn an approximator of value (or action-value) function of any policy, while on-policy methods are limited to the policy π.

Algorithm SARSA [15] is an example of on-policy methods. It learns an estimate Q^{π}_{Θ} of action-value function Q^{π} using TD learning.

$$\delta_t = r_{t+1} + \gamma Q^{\pi}_{\Theta}(s_{t+1}, a_{t+1}) - Q^{\pi}_{\Theta}(s_t, a_t) \qquad (2.18)$$

In TD error computation, SARSA bootstraps action-value estimate $Q^{\pi}_{\Theta}(s_{t+1}, a_{t+1})$ using action a_{t+1} sampled from the policy π. Therefore, it learns estimated action-value function of the policy π, the same policy that was used to generate the transition.

On the other hand, off-policy methods are capable of learning an approximator of value (or action-value) function of an arbitrary policy μ using the transitions generated by the policy π. Off-PAC [16] algorithm is an example of off-policy methods that learns an action-value approximator Q^{μ}_{Θ} of a policy μ

$$\delta_t = r_{t+1} + \gamma Q^{\mu}_{\Theta}(s_{t+1}, \mu(s_t)) - Q^{\mu}_{\Theta}(s_t, a_t) \tag{2.19}$$

In TD error computation, Off-PAC uses bootstrapped action-value estimate $Q^{\mu}_{\Theta}(s_{t+1}, \mu(s_t))$ of the next state and the action $\mu(s_t)$ generated by an arbitrary deterministic policy μ independently of π.

The property of off-policy methods to learn from experiences generated by other policies is highly useful, as it allows them to learn more efficiently. Firstly, it allows the use of stochastic exploratory policies to generate a wide range of experience for learning. Secondly, the policies typically change in time as we are optimizing them. On-policy methods are only able to learn from the most recent transitions generated by the current policy, while off-policy methods can take advantage of the entire history of transitions. This higher sample efficiency makes off-policy methods of particular interest for this thesis.

2.6 Actor-Critic Methods

2.6.1 Optimal Policy

In reinforcement learning, we are trying to find a policy that maximizes the expected discounted return. Policy μ^* is optimal [1] if value function V^{μ^*} of following this policy has the same values as optimal value function $V^{\mu^*}(s) = V^*(s)$ in every state $s \in S$. Let's suppose that the optimal action-value Q^* was known, then the optimal policy could be acquired by greedily choosing action that maximizes action-value in current state s_t.

$$\mu^*(s_t) = arg \max_{a} Q^*(s_t, a) \tag{2.20}$$

This is a central idea of RL techniques for solving tasks with discrete actions from finite action spaces, such as Q-learning [17]. These techniques typically don't require explicit representation of a policy, since it is implicitly specified by Q function. However, a naive application of Q-learning to continuous action domain by discretization of actions suffers from combinatorial explosion [6] and doesn't yield good results. To solve this issue, actor-critic methods are typically used.

2.6.2 Deterministic Actor-Critic

An actor-critic architecture [1] consists of two components: an actor that selects actions and a critic which criticizes them. These methods maintain an explicit parametrized representation of a deterministic policy μ_θ, called actor, and an action-value function Q^μ_Θ, called critic.

After the actor selected an action $a_t = \mu_\theta(s_t)$ in a state s_t, we let the critic Q^μ_Θ criticize it by assigning it an action-value $Q^\mu_\Theta(s_t, \mu_\theta(s_t))$. Since finding the optimal action $a^* = arg \max_a Q^\mu_\Theta(s_t, a)$ w.r.t. current action-value approximator Q^μ_Θ is intractable in a continuous action space, we modify actor's parameters in the direction that increases the action-value. We use deterministic policy gradient theorem [7] to modify actor's parameters θ alongside gradient $\nabla_\theta Q^\mu_\Theta(s, \mu_\theta(s_t))$

$$\theta \leftarrow \theta + \alpha \nabla \theta Q_\Theta(s_t, \mu_\theta(s_t)) \tag{2.21}$$

where constant α is a step size (also called learning rate) in the direction of the gradient. By applying chain rule to the above equation, we can split this gradient into the gradient of policy w.r.t. its parameters and the gradient of action-value w.r.t. to the action

$$\theta \leftarrow \theta + \alpha \nabla_\theta \mu_\theta(s_t) \nabla_{\mu_\theta(s_t)} Q^\mu_\Theta(s_t, \mu_\theta(s_t)) \tag{2.22}$$

After actor's parameters θ have been updated, the action-value function is no longer up-to-date with the latest policy and its parameters Θ need to be updated using TD learning (section 2.5) with TD error

$$\delta_t = r_{t+1} + \gamma Q^\mu_\Theta(s_{t+1}, \mu_\theta(s_t)) - Q^\mu_\Theta(s_t, a_t) \tag{2.23}$$

Usually, the process of policy improvement and learning its value function approximator alternates in a so called policy iteration [1].

2.7 Model-Based Reinforcement Learning

Model-based RL is a family of RL methods that utilize a model of the environment to learn policies. Model-free methods, on the other hand, learn without knowing or learning a model of the environment [1]. Model is an approximator of how the environment reacts to agent's actions. We assume that this is not known a priori. We distinguish a model of reward function M^r and a model of transition function M^T. Reward model M^r approximates the environment's reward function r

$$M^r(s_t, a_t) \approx r_{t+1} = r(s_t, a_t) \tag{2.24}$$

Transition model M^T approximates transitions to next states according to the transition function T of the environment

$$M^T(s_t, a_t) \approx s_{t+1} = T(s_t, a_t) \tag{2.25}$$

Learning the model is typically constructed as a standard supervised learning task, where the goal is to minimize some notion of a distance between the predicted output and the target output.

2.7.1 Model Learning Actor-Critic

Model learning actor-critic (MLAC) [18] is a model-based actor-critic method. MLAC assumes that the environment is deterministic and learns a transition model M^T, a value function V^μ_Θ, and a deterministic policy μ_θ using local linear regression for function approximation.

In an arbitrary state s_t, policy parameters θ are optimized using modified policy gradient method in the direction that maximizes value $V^\mu_\Theta(s_{t+1})$ of the next state. Note that this is an approximative method, since an optimal policy must also maximize immediate reward, which is ignored here. Even though the next state s_{t+1} is not known, it can be inferred by model M^T from the state s_t and agent's action $\mu_\theta(s_t)$. The resulting gradient of the next state's value w.r.t. policy parameters is

$$\nabla\theta V^\mu_\Theta\Big(M^T(s_t, \mu_\theta(s_t))\Big) \tag{2.26}$$

By applying the chain rule, we can decompose the gradient into

$$\nabla_\theta \mu_\theta(s_t) \nabla_{a_t} M^T(s_t, \mu_\theta(s_t)) \nabla_{M^T(s_t, \mu_\theta(s_t))} V^\mu_\Theta\Big(M^T(s_t, \mu_\theta(s_t))\Big) \tag{2.27}$$

Intuitively, the value function gradient tells us in which direction to change the next state to increase its value. Then, we use the model to find out how to change the action in order to transition closer to that better next state. Finally, we modify the policy parameters to produce an action closer to this.

Value function approximator V^{μ}_{Θ} is learned using TD learning with eligibility traces [1]. One-step return backup R^1_t is estimated using the value of the next state retrieved from the transition model and the immediate reward r_{t+1} sampled from the environment with an exploratory policy, which makes the method on-policy

$$R^1_t = r_{t+1} + \gamma V^{\mu}_{\Theta}\Big(M^T\left(s_t, \mu_\theta(s_t)\right)\Big) \tag{2.28}$$

2.7.2 Dyna-MLAC

Dyna framework [8] is a method of accelerating RL algorithms with the help of an environment model. While classical RL algorithms are trained only from the interactions sampled from an environment, Dyna offers a simple way to generate hypothetical experiences using a learned model. RL methods can train on these hypothetical transitions without any special treatment.

Dyna-MLAC [10] combines MLAC method with Dyna framework. It trains a policy and a value function with MLAC update (eq. 2.27, 2.28) using the real transitions (s_t, r_{t+1}) generated an arbitrary exploratory policy in the real environment. In addition to that, it simulates N interaction steps inside an environment model with the same exploratory policy. Model's state $\tilde{s}_{\tilde{t}}$ and time step $\tilde{t}$ are independent of the real environment. MLAC update is then also performed on these generated hypothetical transitions $(\tilde{s}_{\tilde{t}}, \tilde{r}_{\tilde{t}+1}), \ldots, (\tilde{s}_{\tilde{t}+N}, \tilde{r}_{\tilde{t}+N+1})$.

3 Deep Reinforcement Learning

Deep reinforcement learning [2] is the combination of reinforcement learning and deep learning [3]. Deep neural networks are typically used as approximators of a policy and an action-value function. Recently, deep RL has successfully learned to play Atari 2600 games directly from raw input [4] and beaten human champions in the game Go [5]. Deep neural networks are particularly interesting nonlinear function approximators for their universal approximation ability [19]. However, using deep neural networks brings its own set of challenges [20] and requires modifications of the standard RL algorithms.

3.1 Neural Networks

Artificial neural networks (ANN) are a set of machine learning models loosely similar to biological neural networks [3]. The basic processing unit of an ANN is a neuron. Neuron processes its input vector x by multiplying each input element x_i by its respective weight w_i and adding bias term b

$$z = \sum_i w_i x_i + b \tag{3.1}$$

A differentiable activation function f is then applied to the weighted sum z to calculate activation y of the neuron

$$y = f(z) \tag{3.2}$$

3.1.1 Multilayer Perceptron

Multilayer perceptron (MLP), sometimes referred to as deep feedforward network or deep neural network, is a type of a feedforward ANN, whose neurons form an acyclic directed graph and are organized into layers [3]. MLPs are often used as nonlinear function approximators.

A layer is a set of neurons with the same inputs. MLP consists of an input layer, one or more hidden layers, and an output layer. The input layer is a special kind of layer that has no inputs, but rather stores the network's input in its activations. We say that a layer is fully-connected if the layer, its input, and the connections between them form a complete bipartite graph. Hidden layers and output layer are all fully-connected.

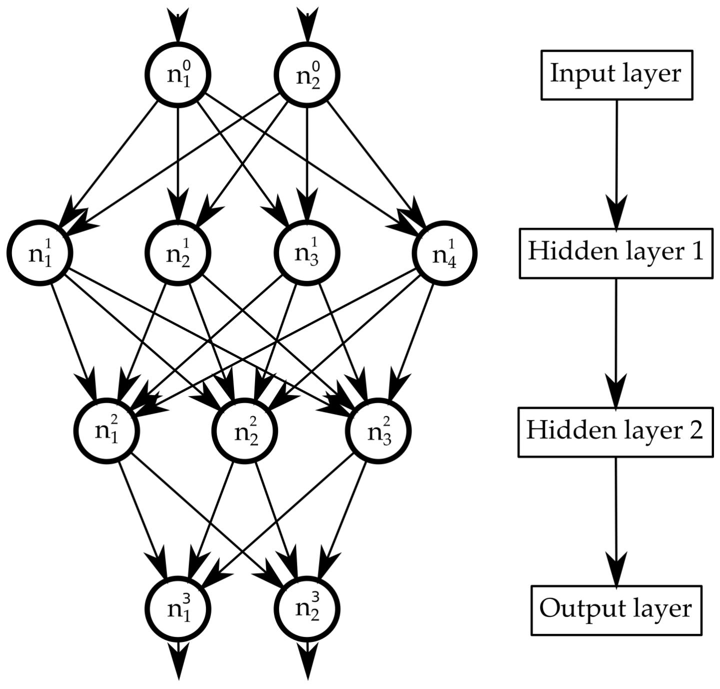

Figure 3.1: MLP consisting of an input layer with two neurons, two hidden layers with four and three neurons respectively, and an output layer with two neurons. Schematic diagram of the layers on the right.

Forward propagation Let's suppose we have an MLP with M hidden layers. Instead of describing behavior of individual neurons, we will use more convenient vector notation to describe entire layer [3]. We denote inputs of the network as vector y_0, activations of neurons in ith hidden layers as vector y_i, and output layer activations as y_{M+1}. We also denote bias terms as vector b_i and connection weights as weight matrix W_i, where element in jth row and kth column is a weight of connection from input j to neuron k of the respective layer.

Each non-input layer $i = 1, \dots, M+1$ computes its weighted sum of inputs z_i and activation y_i based on the activation of previous layer y_{i-1} according to

$$z_i = y_{i-1} W_i + b_i \tag{3.3}$$

$$y_i = f(z_i) \tag{3.4}$$

The output of a network is computed in a process called forward propagation. Firstly, activations of the input layer y_0 are set to the input of the network. Then, activations are propagated from the input layer through hidden layers to the output layer in a cascading fashion (figure 3.1) following equations 3.3 and 3.4. Finally, activations of the output layer become the output of the network.

Backpropagation ANNs are usually trained using gradient descent methods [3]. These methods are trying to minimize some loss function L of network parameters θ (weights and biases) by modifying parameters in the direction of gradient $\nabla_\theta L(\theta)$. To compute this gradient for parameters in every layer, we use backpropagation, or backward propagation of errors.

We start the process by computing $\nabla_{y_{M+1}} L(\theta)$, the gradient of the loss function w.r.t. to the network's output [3]. This gradient will depend on the chosen loss function. As an example, the squared error loss $L(\theta) = \frac{1}{2}\|y_{M+1} - target\|^2$ is commonly used in supervised learning where the goal is to bring the network's output closer to the *target* vector. By differentiating this loss w.r.t. the network's output, we get $\nabla_{y_{M+1}} L(\theta) = y_{M+1} - target$.

Each layer then computes its error vector $\delta_i = \nabla_{z_i} L(\theta)$, which is the gradient of the loss function w.r.t. the layer's weighted sum of inputs. With the gradient $\nabla_{y_{M+1}} L(\theta)$ that we already calculated, we can directly compute the error vector of the output layer δ_{M+1} as

$$\delta_{M+1} = \nabla_{y_{M+1}} L(\theta) \odot f'(z_i) \tag{3.5}$$

where $\odot$ is element-wise multiplication operator and f' is derivative of the activation function. We require activation function to be differentiable for this reason. For every hidden layer $i = M \dots 1$, we use the chain rule to propagate the errors backward according to equation

$$\delta_i = (W_{i+1}^T \delta_{i+1}) \odot f'(z_i) \tag{3.6}$$

where W_{i+1}^T is transposed weight matrix of the next layer. Similar to forward propagation, backpropagation propagates the error in a cascading fashion. However, propagation happens in the opposite direction from the output layer to the first hidden layer. After that, we compute the desired gradient of the loss w.r.t. the parameters $\nabla_\theta L(\theta)$. Since parameters of layer i consist of both weight matrix W_i and bias b_i, we compute their gradients as follows

$$\nabla_{W_i} L(\theta) = y_{i-1}^T \delta_i \tag{3.7}$$

$$\nabla_{b_i} L(\theta) = \delta_i \tag{3.8}$$

Finally, parameters θ are updated alongside their respective gradient depending on our gradient descent algorithm of choice. In case of the standard stochastic gradient descent (SGD) [3] with learning rate parameter α, the update looks as follows

$$\theta_i \leftarrow \theta_i + \alpha \nabla_\theta L(\theta) \tag{3.9}$$

One may also choose a more advanced technique such as momentum [21], root mean square propagation (RMSProp) [22], or Adaptive Moment Estimation (Adam) [23]. They typically use information about gradients from past updates to heuristically improve training speed.

The same technique described here can be used for the gradient ascent, instead of descent, by simply flipping the sign of the loss.

3.1.2 Regularization

When searching for parameters of a neural network, it is not only important to find ones that minimize the loss on the training data, but ones that also generalize well to new unseen data. One approach to this problem is limiting network's capacity using parameter norms. L^2 parameter regularization is an approach to regularization, that introduces weight decay [3]. We modify the original loss function L of parameters θ to also minimize the magnitude of the weights W (biases are unchanged), the assumption being that smaller weights generalize better. This gives us a new loss function L_R

$$L_R(\theta) = L(\theta) + \frac{1}{2}\|W\|^2 \tag{3.10}$$

3.1.3 Mini-Batch Methods

In the previous sections, we have described online (or stochastic) gradient descent algorithm that uses single input sample to form an estimate of the gradient. In general, this doesn't have to be the case. We distinguish a spectrum of methods ranging from online methods to batch methods, that use the entire dataset to estimate the gradient. Mini-batch methods are in the middle and estimate gradient from multiple samples, called mini-batch [3].

Justification for using mini-batches is achieving more accurate gradient estimate by reducing the variance of individual samples, which may improve training time. Thanks to the parallelism of modern graphics processing units (GPUs), the performance penalty of using mini-batches is sublinear w.r.t. batch size.

Mini-batch is an input matrix whose rows are individual input vectors. Equations for forward propagation (section 3.1.1) and backpropagation (section 3.1.1) require no modification thanks to our notation. One change that's typically done is scaling the gradients by factor $\frac{1}{m}$, where m is batch size, in order to average the gradients instead of summing them.

3.1.4 Batch Normalization

Different dimensions of network's input may have different scales or be shifted, which unnecessarily slows down learning. Generally, it is a good practice to normalize each dimension of network's input vector to have zero mean and unit variance across the whole dataset. However, this only helps in training the first hidden layer of an MLP. Input distributions of layers that are deeper in the network may still change uncontrollably as the network is trained.

Batch normalization [24] is a technique for accelerating training of neural networks. It performs normalization per each mini-batch and can be applied to inputs of all layers. During training, inputs are normalized by mean and variance computed from the mini-batch. Moving averages of mean and variance of inputs are maintained and used for normalization during testing. Batch normalization is differentiable and backpropagates errors through itself. For exact derivations of how it is trained, we refer the reader to the original paper [24].

3.2 Experience Replay

An experience $x_t = (s_t, a_t, r_{t+1}, s_{t+1})$ is created by observing interaction of an agent and an environment in two consecutive steps [4, 25]. Deep neural networks typically require a lot of training data to learn. Training them with online stochastic gradient descent using only the most recent experience doesn't lead to satisfactory results for two reasons. Firstly, it is wasteful to train on each experience only once. Secondly, consecutive experiences are typically highly correlated. This breaks assumptions of gradient descent methods and causes catastrophic forgetting [26] of old experiences in neural networks.

One solution to this problem is experience replay. After every interaction with the environment, we store the current experience x_t in a replay memory. Replay buffer $\Re$ is a data structure used as replay memory that can store a limited number of experiences N. When it's full, the oldest sample is removed to make space for a new one. When a neural network is trained with experience replay, it uses mini-batches of experiences sampled uniformly from the replay buffer. This helps to both reuse experiences and to uncorrelate consecutive samples.

Experience replay has demonstrated its usefulness in Deep Q-learning [2, 4], a variant of Q-learning with a neural network approximator of the action-value function, which has successfully learned to play Atari 2600 games.

3.3 Prioritized Experience Replay

Experience replay samples experiences uniformly from the replay memory, which means they are replayed with the same frequency as they are experienced. However, not all experiences are equally important for learning, especially if rewards are very sparse, e.g. they are non-zero only at the end of an episode.

Prioritized experience replay [27] addresses this issue by assigning a priority p_t to each experience sample x_t. Experience x_t is sampled from a prioritized replay buffer $\Re$ stochastically according to their priority $p_t > 0$. Probability of selecting sample x_t is defined as

$$P(t) = \frac{p_t^\alpha}{\sum\limits_{i \in \Re} p_i^\alpha} \tag{3.11}$$

where exponent α determines the amount of prioritization. In general, priorities may be arbitrary. In RL tasks, it is quite suitable to prioritize samples by magnitude of their TD error, which describes how surprising, or informative, the sample is. For experience x_t we set its priority $p_t = |\delta_t| + \epsilon$, where ϵ is a small positive constant to make sure $p_t > 0$. Priorities of samples may be updated at any time. A convenient time to update them is right after the experiences are replayed, as we can use TD errors that have already been computed.

Prioritized replay changes the distribution of experiences away from the real distribution [27]. This negatively affects the learning of expected values depending on this distribution. In order to correct for this bias, importance-sampling weight w_t of experience x_t is used

$$w_t = \left(\frac{1}{|\Re|} \cdot \frac{1}{P(t)} \right)^\beta \tag{3.12}$$

Exponent $\beta \in \langle 0, 1 \rangle$ is a constant determining amount of compensation for this bias ($\beta = 1$ is full compensation, $\beta = 0$ none). Weights are normalized by $1/\max_i w_i$ for stability reasons. TD error δ_t is scaled using the normalized weight to $w_t \delta_t$ and then used for training.

3.4 Deep Deterministic Policy Gradient

Deep deterministic policy gradient (DDPG) [6] is a model-free off-policy actor-critic RL algorithm applicable to problems with continuous action spaces. It is based on the deterministic policy gradient method [7] (section 2.6.2) and uses neural networks as function approximators of a deterministic policy μ_θ with parameters θ and an action-value function Q^μ_Θ with parameters Θ.

Similar to Deep Q-learning [2, 4], DDPG also uses experience replay to train its neural networks by replaying random mini-batches of uncorrelated transitions (section 3.2).

3.4.1 Target Networks

Straightforward application of neural networks to deterministic policy gradient (section 2.6.2) doesn't lead to stable training, as action-value approximator is trained recursively using bootstrapped action-value estimated by the same approximator. Nonlinear function approximators such as neural networks are known to diverge under such conditions in TD learning [20].

DDPG solves this problem by using another set of networks for creating bootstrapped backups [6]. Target policy network $\mu_{\theta'}$ is initialized with parameters θ', a copy of original policy parameters θ. Likewise, target action-value network $Q^{\mu}_{\Theta'}$ is initialized with parameters Θ', a copy of original action-value parameters Θ. Target networks then slowly track original parameters. Every time the original parameters are updated, target parameters move closer to them

$$\begin{aligned} \theta' &\leftarrow \tau\theta + (1-\tau)\theta' \\ \Theta' &\leftarrow \tau\Theta + (1-\tau)\Theta' \end{aligned} \tag{3.13}$$

according to parameter $\tau \in \langle 0, 1\rangle$. Smaller τ leads to slower tracking of original parameters, but more stable learning.

Action-value parameters Θ are updated according to one-step TD error δ_t with bootstrapped return backup estimated by target networks

$$\delta_t = r_{t+1} + \gamma Q^{\mu}_{\Theta'}(s_{t+1}, \mu_{\theta'}(s_t)) - Q^{\mu}_{\Theta}(s_t, a_t) \tag{3.14}$$

3.4.2 Deterministic Policy Gradient

Policy parameters θ are updated in line with the deterministic policy gradient [7] (section 2.6.2)

$$\nabla_{\theta} Q^{\mu}_{\Theta}(s, \mu_{\theta}(s_t)) \tag{3.15}$$

Although, the use of neural networks trained by backpropagation offers a more intuitive view of the policy gradient computation. Let us consider an actor-critic architecture as shown in figure 3.2 on the next page. Policy and action-value networks are interlinked [28] so as to compute action-value $Q^{\mu}_{\Theta}(s_t, \mu_{\theta}(s_t))$, where s_t is an arbitrary state. The goal of the training process is to increase the action-value

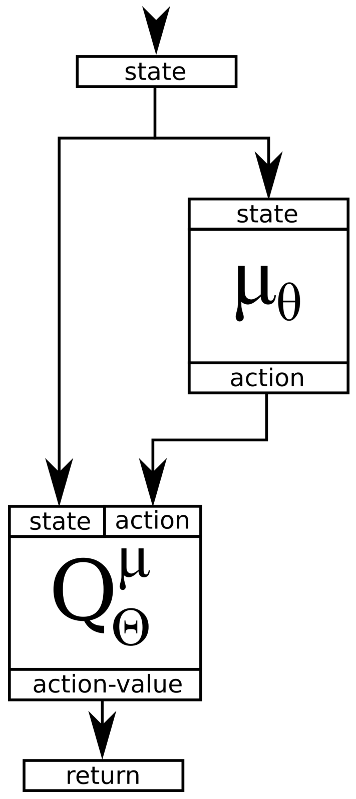

Figure 3.2: Actor-critic architecture of DDPG.

$Q^\mu_\Theta(s_t, \mu_\theta(s_t))$ by modifying policy parameters θ. This leads to a straightforward definition of a loss function

$$L_t(\theta) = Q^\mu_\Theta(s_t, \mu_\theta(s_t)) \tag{3.16}$$

Since the approximators Q^μ_Θ and μ_θ are differentiable, so is the loss function. We use gradient ascent to modify policy parameters in the direction of gradient $\nabla_\theta L_t(\theta)$. Application of chain rule to this gradient and computation of parameter updates is taken care of by the backpropagation algorithm (section 3.1.1). In this process, parameters of action-value approximator remain constant and only the policy parameters are updated.

4 Deep Model Learning Actor-Critic

Deep model learning actor-critic (DMLAC) is a novel model-based off-policy actor-critic algorithm that we developed. DMLAC adapts ideas of Dyna-MLAC [10] method for use with neural networks using recent advancements in deep reinforcement learning. DMLAC learns a model of the environment from experiencing interactions with it. A deterministic policy and its value function are trained by simulating interactions in the learned model. Policy, value function, and model are all trained simultaneously in every step, without requiring a "motor babbling" phase to identify the model. We discuss learning of individual components in the following sections.

4.1 Model Learning

We assume that the environment's transition function and reward function are not known a priori. DMLAC learns a transition model which approximates environment's state transition function. Unlike some other model-based methods, which only learn the transition model [9, 10, 18, 29], DMLAC also learns a reward model which approximates environment's reward function.

We represent the transition model M^T_ϕ using a neural network with parameters ϕ. Analogously, reward model M^r_ψ is represented using a neural network with parameters ψ. We impose no limitation on the architecture of the neural networks other than differentiability w.r.t. their respective parameters and action input.

Reward model We want the reward model M^r_ψ to approximate the environment's true reward function $M^r_\psi(s, a) \approx r(s, a)$ for all states and actions. Given a particular input consisting of a state s_t and an action a_t, the reward model should predict reward $\tilde{r}_{t+1} = M^r_\psi(s_t, a_t)$ that approximates the true reward r_{t+1} which would be received after taking action a_t in state s_t.

A neural network representing the reward model is trained by gradient descent (section 3.1.1). We update reward model's parameters ψ in the direction that minimizes reward model error. For an arbitrary transition (s_t, a_t, r_{t+1}), we define loss function L_t^r as a squared error between the predicted and observed reward

$$L_t^r(\psi) = \frac{1}{2}\Big(M_\psi^r(s_t, a_t) - r_{t+1}\Big)^2 \tag{4.1}$$

Then, we use backpropagation to update parameters ψ in the direction of gradient $\nabla_\psi L_t^r(\psi)$.

Transition model Similarly, we want the transition model M_ϕ^T to approximate the environment's transition function $M_\psi^T(s, a) \approx T(s, a)$ for all states and actions. Given an arbitrary state s_t and action a_t, the transition model should predict next state $\tilde{s}_{t+1} = M_\phi^T(s_t, a_t)$ that approximates the true next state s_{t+1} into which the agent would transition by taking action a_t in state s_t.

We train the neural network representing transition model by gradient descent (section 3.1.1). We modify policy parameters ϕ in the direction that decreases prediction error. We define loss function L_t^T for an arbitrary transition (s_t, a_t, s_{t+1}) as squared error between the predicted and observed next state

$$L_t^T(\phi) = \frac{1}{2}\Big\|M_\phi^T(s_t, a_t) - s_{t+1}\Big\|^2 \tag{4.2}$$

Parameters ϕ are then updated in the direction of gradient $\nabla_\phi L_t^T(\phi)$ using backpropagation algorithm.

4.2 Policy Learning

Deterministic policy μ_θ with parameters θ is represented with a neural network. We train the policy using a modified policy gradient method similar to MLAC [18] and deterministic value gradients [29]. Given an arbitrary state s_t, we want to adjust the policy parameters in the direction that increases expected discounted return R_t of following policy μ_θ from this state onward. Since this infinite-horizon return is not available (section 2.3), we use the Bellman equation [13] to estimate one-step return R_t^1 by bootstrapping a value estimate from a value approximator V_Θ^μ

$$R_t^1 = \mathbb{E}_{\mu,E}\big[r_{t+1} + \gamma V_\Theta^\mu(s_{t+1})\big] \tag{4.3}$$

We can remove the expectation of following the policy μ_θ inside the environment E by rewriting the equation using the agent's policy function and the environment's reward and transition functions.

$$R_t^1 = r(s_t, \mu_\theta(s_t)) + \gamma V_\Theta^\mu(T(s_t, \mu_\theta(s_t))) \tag{4.4}$$

The environment's reward and transition functions are not known to us. However, we can utilize reward and transition model, which approximate them. We perform one-step interaction of the agent and model. Agent takes action $a_t = \mu_\theta(s_t)$ according to its policy inside the model and receives reward $\tilde{r}_{t+1} = M_\psi^r(s_t, a_t)$. Model then transitions to next state $\tilde{s}_{t+1} = M_\phi^T(s_t, a_t)$ whose estimated value is $V_\Theta^\mu(\tilde{s}_{t+1})$. After assembling these terms together, we get the expected discounted return $\tilde{R}_t^1$ which is dependent on the current model

$$\tilde{R}_t^1 = M_\psi^r(s_t, \mu_\theta(s_t)) + \gamma V_\Theta^\mu(M_\phi^T(s_t, \mu_\theta(s_t))) \tag{4.5}$$

Assuming that the model is a good approximation of the real environment, then the model-dependent return $\tilde{R}_t^1$ approximates the real return R_t^1. We optimize the policy by gradient descent so as to increase the model-dependent expected discounted return. Gradient of model-dependent return $\tilde{R}_t^1$ w.r.t. policy parameters θ is

$$\nabla_\theta \tilde{R}_t^1 = \nabla_\theta \Big[M_\psi^r(s_t, \mu_\theta(s_t)) + \gamma V_\Theta^\mu(M_\phi^T(s_t, \mu_\theta(s_t))) \Big] \tag{4.6}$$

By applying chain rule, we decompose the gradient $\nabla_\theta \tilde{R}_t^1$ into

$$\nabla_\theta \mu_\theta(s_t) \nabla_{a_t} M_\psi^r(s_t, a_t) + \gamma \nabla_\theta \mu_\theta(s_t) \nabla_{a_t} M_\phi^T(s_t, a_t) \nabla_{\tilde{s}_{t+1}} V_\Theta^\mu(\tilde{s}_{t+1}) \tag{4.7}$$

This gradient computation can be grasped more intuitively by looking at it from the perspective of an actor-critic. Let's consider an actor-critic network created by interlinking policy, model and value networks as shown in figure 4.1 on the next page. For an arbitrary state s_t, this actor-critic network computes the model-dependent expected discounted return $\tilde{R}_t^1$. The goal of the training process is to increase $\tilde{R}_t^1$ by modifying the policy parameters θ. Similar to DDPG (section

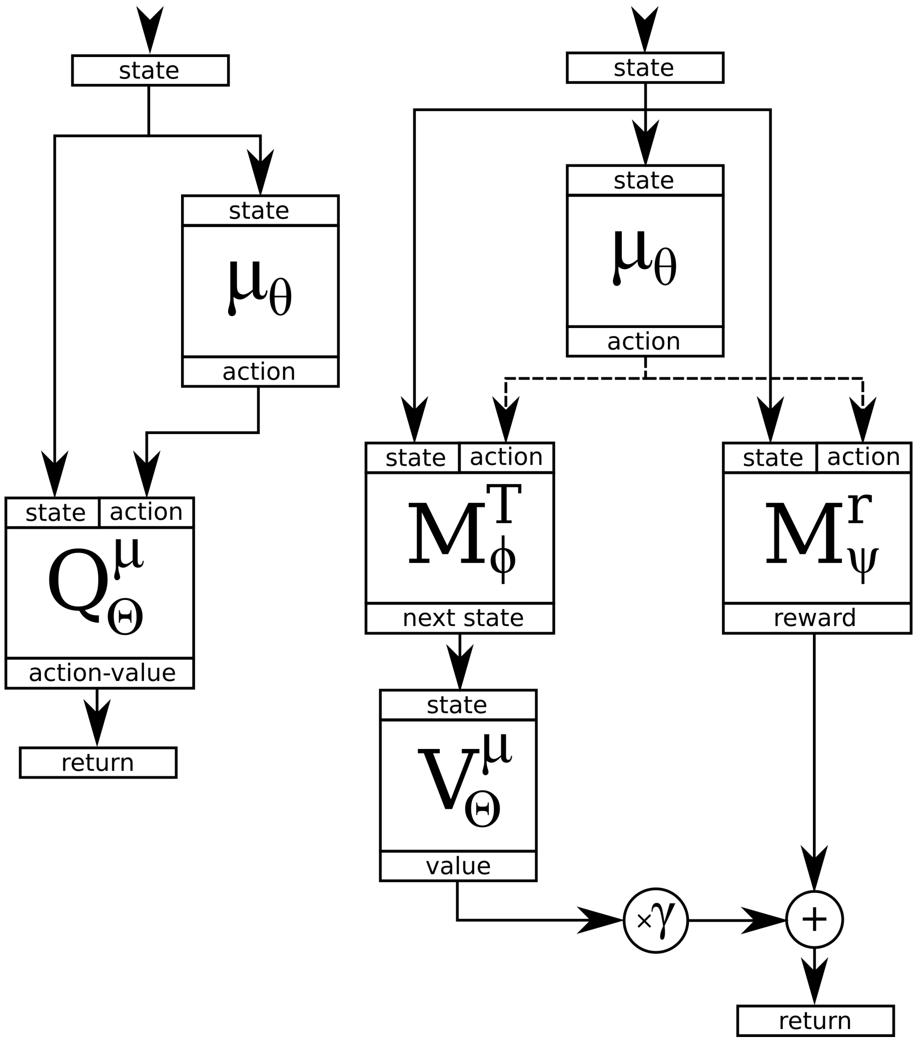

Figure 4.1: Actor-critic architecture of DDPG (left) compared to "actor-model-critic" architecture of DMLAC (right).

3.4.2), we train this actor-critic network by gradient ascent. We define a loss function of policy parameters $L_t(\theta)$ as the model-dependent return $\tilde{R}^1_t$

$$L_t(\theta) = M^r_\psi(s_t, \mu_\theta(s_t)) + \gamma V^\mu_\Theta(M^T_\phi(s_t, \mu_\theta(s_t))) \tag{4.8}$$

Since model, value, and policy networks are differentiable, so is the loss function. We can update policy parameters θ in the direction of gradient $\nabla_\theta L_t(\theta)$. We compute this gradient using backpropagation algorithm (section 3.1.1). Only the policy parameters are updated.

4.3 Value Learning

We train a value function approximator using n-step TD(λ) learning [1] with experience replay [25]. Firstly, we sample a random mini-batch of B states from the replay memory. From each of these states, we simulate n interaction steps forward inside the learned model. Interactions inside the model are executed in parallel using mini-batches. This comes at little extra computation cost thanks to advances of modern hardware. The simulated interactions can be created using any policy, which opens up possibilities of using various on-policy and multi-step TD methods, while keeping the overall algorithm off-policy.

Our approach is related to asynchronous actor-critic methods [30] which perform interactions in multiple parallel environments. The use of learned model reduces number of interactions with the real environment, which may be costly. Additionally, parallel environments may not always be available (e.g. in real-world tasks).

Let V^μ_Θ be a value function approximator of a deterministic policy μ_θ. We use target networks $V_{\Theta'}$, $\mu_{\theta'}$, whose parameters slowly track the original ones, to stabilize training (section 3.4.1) [6]. Given an arbitrary starting state s_t from the mini-batch, we simulate n interaction steps using policy $\mu_{\theta'}$ inside the model, starting from the state $\tilde{s}_0 = s_t$. From the sequence of observed states $\tilde{s}_0 \dots \tilde{s}_n$ and rewards $\tilde{r}_1 \dots \tilde{r}_n$, we can calculate any i-step return $\tilde{R}^i$ for $i \in \{1 \dots n\}$

$$\tilde{R}^i = \sum_{k=1}^{i} \gamma^{k-1}\tilde{r}_k + \gamma^i V_{\Theta'}(\tilde{s}_i) \tag{4.9}$$

Model is only an approximation of the environment. Typically, the longer the interaction, the higher the discrepancy between the real environment and the model. To counteract this model error, we use a weighted average of all i-step returns, denoted as λ-return $\tilde{R}^\lambda$. Higher weights are given to shorter interactions according to decay constant

λ. Weights are normalized so that their sum equals 1.

$$\tilde{R}^{\lambda} = \frac{1-\lambda}{1-\lambda^n} \sum_{i=1}^{n} \lambda^{i-1} \tilde{R}^i \tag{4.10}$$

After computing the λ-return $\tilde{R}^{\lambda}$, we use it as a return backup for TD error computation. Parameters Θ of value function approximator V_{Θ} are updated so as to decrease the TD error by gradient descent of loss function $L_t(\Theta)$

$$L_t(\Theta) = (\tilde{R}^{\lambda} - V_{\Theta}(s_t))^2 \tag{4.11}$$

Notably, the return $\tilde{R}^{\lambda}$ is model-dependent and only an approximation of the true return R_t from state s_t. The accuracy of the return depends on the quality of the learned model. Nevertheless, we train model and value approximator simultaneously without needing a motor-babbling phase to learn the model first. In the tested environments, this is not a limitting factor since learning the model is typically easier task than learning the value function.

Episodic tasks A naive application of DMLAC to episodic tasks raises some isssues, as the model is unaware of episode ends. In particular, even perfectly trained model may simulate interactions past terminal state into invalid fictious states.

It is worth reiterating that we use the unified view of tasks described in section 2.2, where episodic tasks are converted to continuing tasks by adding an absorbing state. We solve the above mentioned problem by storing several transitions to absorbing state in replay memory before skipping to the next episode. In theory, a model trained on these transitions can properly deal with episode ends without any further modifications.

4.4 Algorithm

The DMLAC algorithm consists of exploring environment with an arbitrary stochastic exploratory policy π_{exp}, storing the observed transitions in replay memory, and interleaving model, value, and policy learning. Training steps don't need to be synchronized with environment's time steps [9]. We introduce parameter U which denotes number of training updates per one environment time step. The algorithm is presented on the following page.

Algorithm 1: DMLAC algorithm

```
Require: $\gamma, \lambda, \tau, n, B, U$
Initialize random $M^T_\phi$, $M^r_\psi$, $\mu_\theta$, $V_\Theta$, $\pi_{exp}$
$\theta' \leftarrow \theta, \Theta' \leftarrow \Theta, \Re \leftarrow \emptyset$
for $t = 0, \infty$ do
    Observe current state $s_t$
    Take action $a_t$ according to exploratory policy $\pi_{exp}(s_t)$
    Observe next state $s_{t+1}$ and reward $r_{t+1}$
    Add transition $(s_t, a_t, r_{t+1}, s_{t+1})$ into $\Re$
    for $u = 1, U$ do
        Sample a mini-batch of $B$ samples $(s_i, a_i, r_{i+1}, s_{i+1})$ from $\Re$
        Update $M^T_\phi$ by gradient descent of loss:
            $L(\phi) = \frac{1}{B} \sum_{i=1}^{B} \frac{1}{2} \big(M^T_\phi(s_i, a_i) - s_{i+1}\big)^2$
        Update $M^r_\phi$ by gradient descent of loss:
            $L(\psi) = \frac{1}{B} \sum_{i=1}^{B} \frac{1}{2} \big(M^r_\psi(s_i, a_i) - r_{i+1}\big)^2$
        Update $\mu_\theta$ according to policy gradient:
            $\nabla_\theta \tilde{R}^1_t = \frac{1}{B} \sum_{i=1}^{B} \nabla_\theta \Big[M^r_\psi(s_i, \mu_\theta(s_i)) + \gamma V_\Theta(M^T_\phi(s_i, \mu_\theta(s_i)))\Big]$
        for $i = 1, B$ do
            Set starting state $\tilde{s}_0 = s_i$
            for $j = 1, n$ do
                Take action $\tilde{a} = \mu_{\theta'}(\tilde{s}_{j-1})$ in model
                Observe $\tilde{s}_j = M^T_\phi(\tilde{s}_{j-1}, \tilde{a})$, $\tilde{r}_{j-1} = M^r_\psi(\tilde{s}_{j-1}, \tilde{a})$
                $\tilde{R}^j \leftarrow \sum_{k=1}^{j} \gamma^{k-1} \tilde{r}_k + \gamma V_{\Theta'}(\tilde{s}_j)$
            $\tilde{R}^\lambda_i = \frac{1-\lambda}{1-\lambda^n} \sum_{j=1}^{n} \lambda^{j-1} \tilde{R}^j$
        Update $V_\Theta$ by gradient descent of loss:
            $L(\Theta) = \frac{1}{B} \sum_{i=1}^{B} \big(\tilde{R}^\lambda_i - V_\Theta(s_i)\big)^2$
        Update target policy and value parameters
```

4.5 Extensions

Prioritized experience replay Prioritized experience replay 3.3 can be applied to DMLAC in a straightforward fashion. Since, there are multiple components being trained, each of them may assign different priorities to individual samples. We prioritize experience for training the policy and value function according to the absolute value of TD error. Experience for training transition and reward model is prioritized according to their respective squared error.

Stochastic environments Our method is suitable for environments that are deterministic or have low enough variance of transitions. However, the same approach can be extended to stochastic environments by learning a stochastic transition model approximating transition probabilities.

5 Implementation

As a part of this thesis, we provide an efficient implementation of the tested deep RL methods (DDPG and DMLAC) written in Python using TensorFlow library [11]. The agents are tested in environments from OpenAI Gym [12], a collection of RL tasks. Additionally, we provide interactive IPython notebooks for viewing the results of our experiments and for running them.

5.1 Neural Networks

We provide a light-weight neural network library for creating feed-forward neural networks (section 3.1) for the purposes of deep reinforcement learning (chapter 3). The key features necessary for deep RL are target networks and interlinking networks into an actor-critic architecture. The implementation of neural networks uses the library TensorFlow [11] as the backend for most of its computations.

TensorFlow TensorFlow [11] is a machine learning library developed by Google. It allows easy creation of complex mathematical models by combining built-in tensor operations (e.g. addition, multiplication, application of a function). The computation is represented by a dataflow graph, where each node of the graph is an operation and each edge is a tensor. TensorFlow supports automatic gradient computation and many popular gradient optimizers such as stochastic gradient descent [3] and Adam [23]. The operations are run on GPU and parallelized whenever possible.

NeuralNetwork NeuralNetwork class implements a feedforward neural network assembled from layers. The network consists of one or more inputs and a single output. The intermediary hidden layers may be linked into an arbitrary acyclic graph connecting the inputs to the output. We offer common layer types such as fully connected layers, batch normalization and layers performing various miscellaneous operations (addition, subtraction, multiplication, concatenation, bounding). Inputs of the network may be either set externally or from an output of another network, which provides functionality for interconnecting networks into actor-critic architectures. The network supports processing its inputs in mini-batches. NeuralNetwork construct a computation graph of the neural network and leaves the computation to the TensorFlow [11] backend.

Gradient optimizers We provide two optimizer classes for training neural networks using gradient methods (section 3.1.1) with mini-batches (section 3.1.3). *SquaredLossOptimizer* performs gradient descent of a squared error loss function. It is particularly suited for supervised learning tasks, where the target is known. *MaxOutputOptimizer* performs gradient ascent of a loss function equal to the network's output. This optimizer is suited for actor-critic networks in RL. Both optimizers use TensorFlow's built-in gradient computation and optimizers.

TargetNeuralNetwork TargetNeuralNetwork class implements a target network which slowly tracks parameters of the original network (section 3.4.1). Other than that, it has the same functionality as *NeuralNetwork* and can be used interchangeably, as they both have the same interface.

5.2 Deep reinforcement learning

Experience replay We provide *ExperienceReplay* class for training RL agents using experience replay (section 3.2). *ReplayBuffer* class handles storing and random sampling of mini-batches for training.

Prioritized experience replay We use *PrioritizedReplayBuffer* to store samples with their respective priorities and sample mini-batches according to the priorities. Similar to experience replay, we implemented prioritized experience replay (section 3.3) for model-free methods with class *PrioritizedExperienceReplay*. Model-based methods may sample mini-batches for model learning differently (section 4.5), which is handled by *ModelBasedPrioritizedExperienceReplay* class.

DDPG We represent the policy and the action-value function of DDPG algorithm (section 3.4) using *NeuralNetwork* objects. We also maintain their target networks for learning the action-value function. The policy network is trained by gradient ascent in an actor-critic network created by interlinking the policy and action-value networks (figure 3.2).

DMLAC DMLAC (chapter 4) represents policy, value function, reward model, and transition model using *NeuralNetwork* objects. We maintain target networks of the policy and value network for learning the value function using n-step TD learning. The model is trained by gradient descent using *SquaredLossMinimizer*. The policy network is trained by gradient ascent in an "actor-model-critic" architecture created by interlinking the networks (figure 4.1).

Exploration strategies For exploring the environment, we use two different exploratory stochastic policies. First, ϵ-greedy policy [1] samples a random action from the action space with probability ϵ and otherwise acts according to the agent's policy. The second exploration policy adds noise to the action selected according to the agent's policy. We use Ornstein-Uhlenbeck process to generate temporally correlated noise [6].

Bounded action space Environments that we tested typically have bounded action spaces, where each dimension of an action vector has an upper and a lower bound. This poses a question how to ensure that the actions generated by the policy honor these bounds. Even more importantly, this causes serious issues for training the policy using gradient methods. It is quite common that the action-value approximator assigns higher action-value to actions that exceed the bounds. The policy gradient methods then aggressively increase the actions beyond bounds to extremely high values, which is not the desired behavior. We solve this problem similarly to the inverting gradient method [28]. Briefly, policy gradient methods compute gradient of the expected return w.r.t. the policy's action. We check whether any action dimension a_d is out of bound. When it is and the gradient suggests moving the action further off the bound, we invert the gradient.

State-action embedding In RL, we are often dealing with functions that have two inputs, state and action (e.g. an action-value function). Approximating such functions with neural networks raises the question of how to combine these inputs. We can simply concatenate the two inputs together into one vector and make no further modification to the network. However, this has performed poorly in our tests where the state dimension was significantly higher than the action dimension. We speculate that it is caused by proportionally smaller contribution of the action input to the weighted sum of the first hidden layer's inputs. We solve this issue by processing the inputs separately by hidden layers of the same size (figure 5.1). Then, we concatenate their outputs and process it by the rest of the network. We call this approach *state-action embedding*. Both of these approaches are implemented.

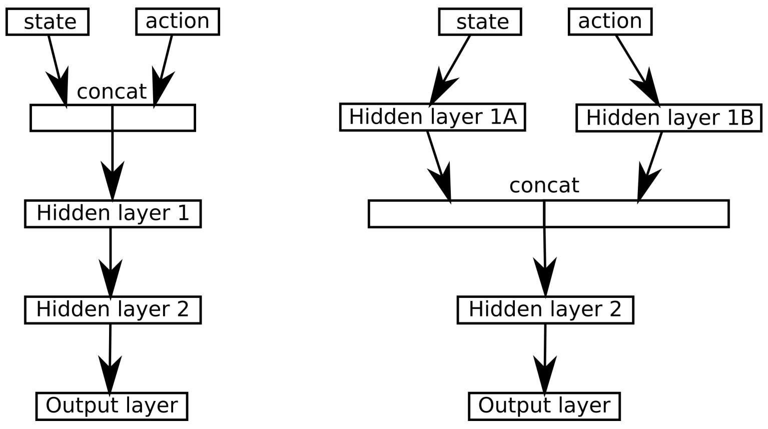

Figure 5.1: Concatenation of a state input and an action input (left) compared to the state-action embedding (right)

5.3 OpenAI Gym

OpenAI Gym [12] is a reinforcement learning toolkit. It includes a collection of various environments for comparing performance of different RL algorithms. Each environment has the same interface for interacting with the agent. We are concerned only with a subset of the environments that have continuous action domain.

Experiment Each interaction between an agent and an environment is run inside an IPython notebook. This notebook manages creation of the deep RL method and its neural networks, interacting and exploring the environment, saving the transitions to a replay memory, and training the agent. It also measures various performance metrics of the agent. After completion, the performance of the agent is visualized and saved. We provide notebooks for all of our experiments. The results are presented in the next chapter.

6 Experiments

6.1 Methodology

We compare DDPG (section 3.4) and DMLAC (chapter 4) in three different environments in terms of sample efficiency. We also study their ability to do transfer learning.

In all experiments, the neural networks have the same architecture. We use exponential linear unit (ELU) [31] activation function in hidden layers and linear activation in the output layer. The policy and value network have two hidden layers (256 and 128 neurons). Action-value, reward model, and transition model, use state-action embedding (section 5.2) with 128 neurons for each input and a second hidden layer with 128 neurons. We train the neural networks with Adam optimizer [23] with learning rate 10^{-4} in policy network and 10^{-3} in the other networks. We initialize the OpenAI Gym environments and exploratory policies with the same random seed. We use prioritized experience replay and perform 10 training updates in one time step. Further details can be found in IPython notebooks of the experiments.

In the graphs that we resent in further sections, we show the average result of 5 runs of the algorithms. Whenever practical, we also show the worst and the best result marked by a filled area around the curve.

6.2 Sample efficiency

In this set of experiments, we are testing the methods w.r.t. their sample efficiency [1]. We are concerned with the number of interactions that it takes to learn a satisfactory policy. As a proxy to sample efficiency, we use the measure of cumulative reward. Cumulative reward is the sum of all rewards received from the beginning (time step 0). We suppose that a method with higher cumulative reward learns a better policy faster and hence is more efficient. In episodic tasks, we also measure cumulative reward per episode.

6.2.1 Pendulum

Pendulum (technically *Pendulum-v0*) [32] is an environment from the OpenAI Gym. An agent manipulates the pendulum by applying torque (a continuous action) in the joint actuator (figure 6.1). The goal is to swing the pendulum up in the vertical position. The maximum obtainable reward is zero when the pendulum is balanced, otherwise the reward is negative.

Pendulum is a continuing task without an end. However, we also consider a modified episodic version, where the episode ends are marked by successfully balancing the pendulum for 100 consecutive steps. We will denote this task as *Pendulum-v0-episodic*. Since this task doesn't require very long term planning, we use discount factor $\gamma = 0.9$. Agents explore the environment with ϵ-greedy exploration strategy, where ϵ decreases from 1 at the beginning of training to 0 at the end polynomially (degree 4).

Figure 6.1: Pendulum environment

We observe that DMLAC learned to balance the pendulum faster than DDPG in all runs (figure 6.2). Additionally, DMLAC's spread of the final cumulative rewards of is smaller. We also tested multi-step DMLAC variants, however they performed almost exactly the same as the 1-step variant. For clarity, we have omitted them from the figures.

In the episodic variant, we observe that both methods learn comparable policies after just one episode (figure 6.3 and 6.4). However, it takes DMLAC on average roughly 400 fewer steps to solve the first episode. Due to this fact, DMLAC achieves better cumulative reward in all runs.

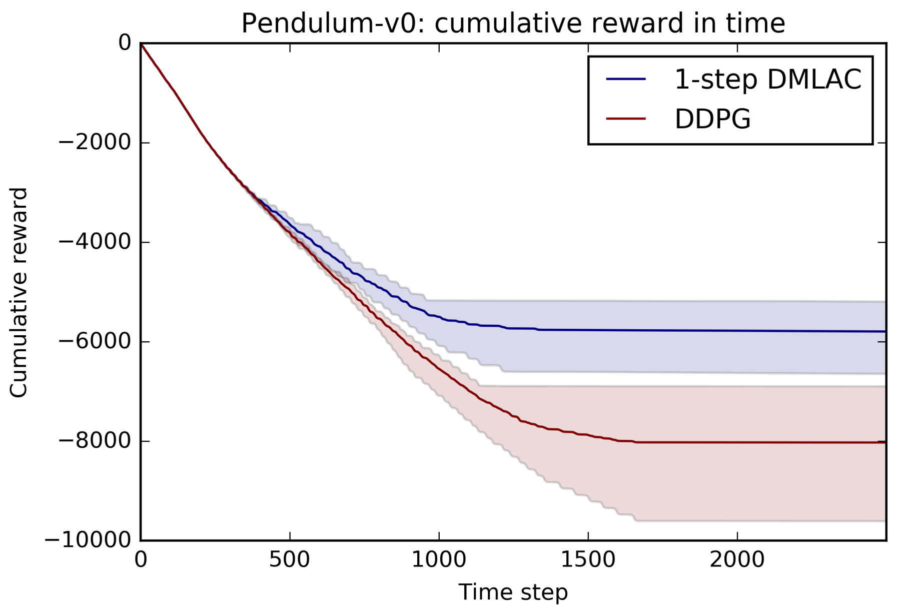

Figure 6.2: Pendulum-v0: cumulative reward in time. DMLAC compared to DDPG

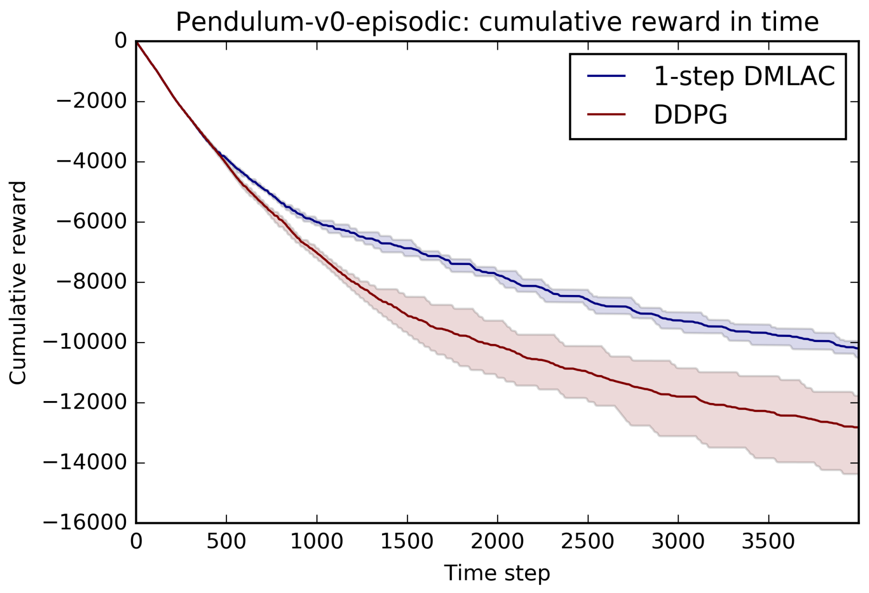

Figure 6.3: Pendulum-v0-episodic: cumulative reward in time. DMLAC compared to DDPG

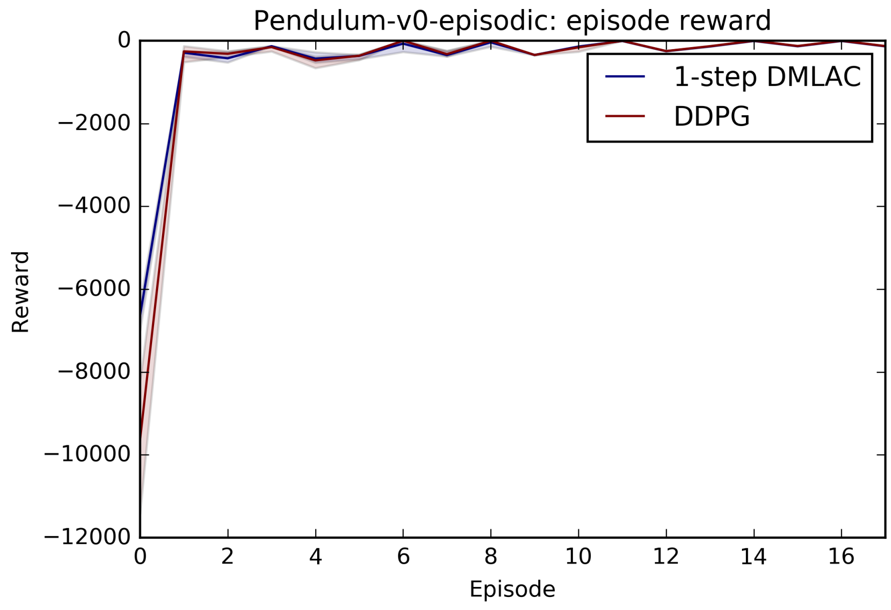

Figure 6.4: Pendulum-episodic: episode reward in time. DMLAC compared to DDPG

6.2.2 Mountain Car

Mountain Car (technically *MountainCarContinuous-v0*) [33] is an environment from the OpenAI gym. The goal is to drive a car up the mountain. However, in order to do that, the car must at first reverse to build up enough momentum to get on top of the hill. Because of this, the task requires longer term planning and thus we use higher discount factor $\gamma = 0.99$. Agents explore this environment with temporally correlated Ornstein-Uhlenbeck noise [6] for one episode. After that, there is no more exploration.

We observe in the figure 6.6 that DMLAC achieves a higher cumulative reward than DDPG, again. Interestingly, we weren't able to achieve a better score with multi-step DMLAC. Using longer than 3-step interactions or higher decay factors λ than 0.5 has proved to be too unstable. For clarity, we omit the upper and lower bounds.

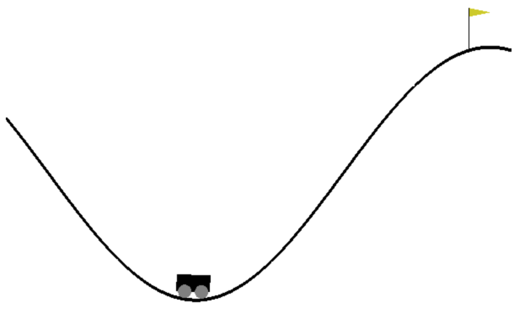

Figure 6.5: Mountain car environment

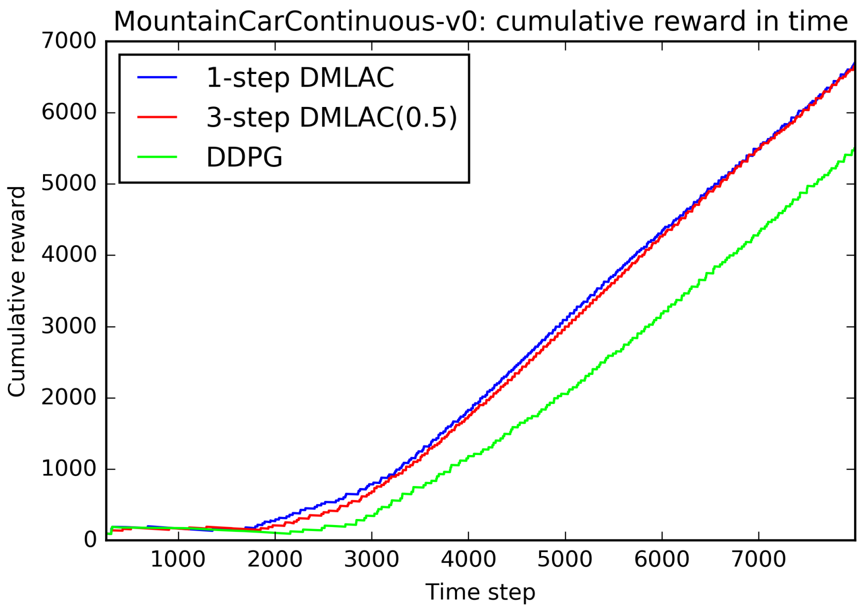

Figure 6.6: Mountain car: cumulative reward in time. DMLAC compared to DDPG

6.2.3 Reacher

Reacher (technically *Reacher-v1*) [34] is an environment with continuous actions from the OpenAI Gym. An agent manipulates a robotic arm by applying torque in two actuated joints (figure 6.7). The goal is to touch a point in a 2-dimensional space. The maximum obtainable reward, when the tip of the arm touches the marked point, is zero.

Otherwise, the reward is negative proportionally to the distance to the point. Agents explore the environment with ϵ-greedy exploratory policy, with ϵ decreasing from 1 at the beginning of training to 0 at the end polynomially (degree 4).

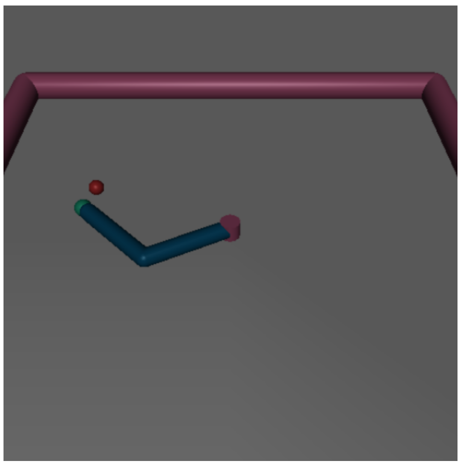

Figure 6.7: Reacher environment

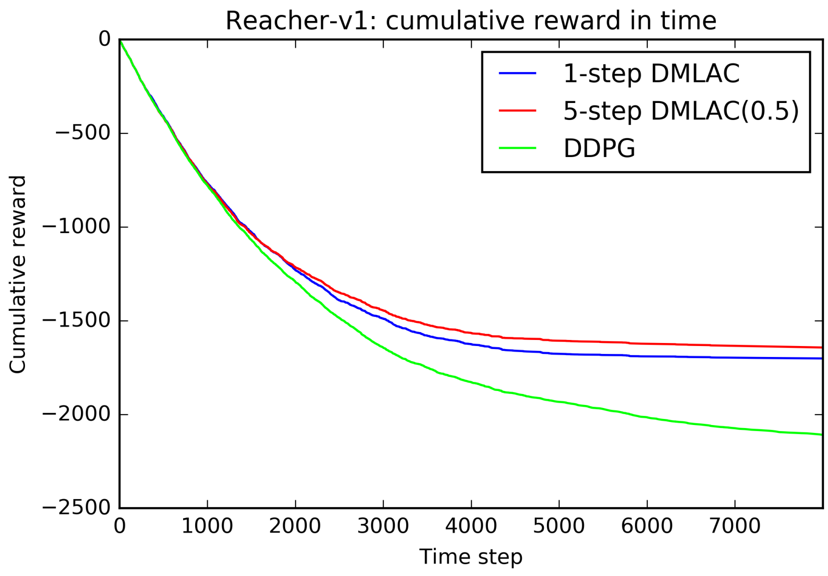

Figure 6.8: Reacher-v1: cumulative reward in time. DMLAC compared to DDPG

In this environment, DMLAC again achieves higher cumulative reward (figure 6.8). The learned policy of DMLAC is qualitatively different. While DDPG's policy oscillates the tip of the arm around the point, DMLAC remains relatively steady on the point. Additionally, we have noticed a marginal improvement when using multi-step DMLAC. Why this is the case only in this environment remains unclear. For clarity, we omit the upper and lower bounds.

6.3 Transfer Learning

In the previous experiments, we have evaluated sample efficiency when learning new tasks from scratch. A more interesting scenario for real-world tasks is transferring already learned knowledge to solve a new task. While OpenAI Gym doesn't provide benchmarks for such use case yet [12], we can simulate a transfer learning task by modifying the environment.

We use the Reacher environment from the previous section. Firstly, we train an agent to reach a point in the bottom right corner of its environment for 5000 training steps, when the policy is able to solve the task. Then, we move the point to the upper left corner, mimicking the same task described in the previous section. Importantly, the agent already has experience of interacting with the environment before starting the new task. We purge the contents of the replay memory in order not to affect the learning of a new task with old experience.

We observe in figure 6.9 on the next page, that DDPG completely failed to transfer any knowledge to the new task. In fact DDPG wasn't able to learn the new task at all while it could learn it from scratch in the previous experiment. We advise the reader to compare the figure 6.8 with the figure 6.9. On the other hand, both 1-step and 5-step versions of DMLAC learned a successful policy much faster by transfer learning than from scratch. By a small margin, 5-step DMLAC learned a qualitatively better policy that touches the point a little closer than 1-step DMLAC. We think this is a successful demonstration of transfer learning.

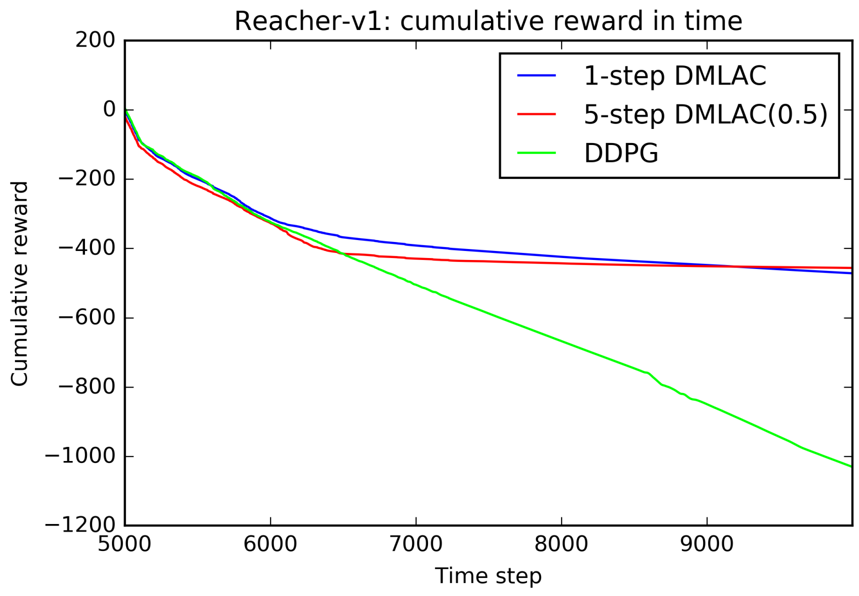

Figure 6.9: Reacher transfer learning: cumulative reward in time. DMLAC compared to DDPG

7 Conclusion

We studied the use of an environment model in reinforcement learning tasks with continuous action spaces. We presented a novel model-based actor-critic deep reinforcement learning method that we call Deep Model Learning Actor-Critic (DMLAC). We gave a theoretical description of our algorithm and contextualized it with older model-based techniques and a contemporary model-free method, Deep Deterministic Policy Gradient (DDPG).

We gave a theoretical description of out algorithm and provided a working implementation of DMLAC and DDPG. We experimentally evaluated both of these methods in environments from OpenAI Gym. Our results suggest that model-based deep reinforcement learning algorithms may indeed improve sample efficiency and show a promising path to solving transfer learning.

Our algorithm is suitable for deterministic environments and policies. A natural direction for future work is extending the algorithm to stochastic environments and policies. We also recommend further tests of the algorithm on more challenging tasks, such as learning directly from a visual input.

8 An appendix

The implementation is available in the attachment, but also online at https://github.com/karolkuna/reinforcement-learning
Installation instructions are provided in *Readme.md*. Due to the hardware requirements (nVidia GPU) and complexity of installation, we recommend viewing the provided interactive IPython notebooks.

Bibliography

1. SUTTON, Richard S.; BARTO, Andrew G. *Introduction to Reinforcement Learning*. 1st. Cambridge, MA, USA: MIT Press, 1998. ISBN 0262193981.
2. MNIH, Volodymyr; KAVUKCUOGLU, Koray; SILVER, David; GRAVES, Alex; ANTONOGLOU, Ioannis; WIERSTRA, Daan; RIEDMILLER, Martin A. Playing Atari with Deep Reinforcement Learning. *CoRR*. 2013, vol. abs/1312.5602. Available also from: `http://arxiv.org/abs/1312.5602`.
3. GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.
4. MNIH, Volodymyr et al. Human-level control through deep reinforcement learning. *Nature*. 2015, vol. 518, no. 7540, pp. 529–533. ISBN 0028-0836. Available also from: `http://dx.doi.org/10.1038/nature14236`.
5. SILVER, David et al. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*. 2016, vol. 529, no. 7587, pp. 484–489. Available from DOI: `10.1038/nature16961`.
6. LILLICRAP, Timothy P.; HUNT, Jonathan J.; PRITZEL, Alexander; HEESS, Nicolas; EREZ, Tom; TASSA, Yuval; SILVER, David; WIERSTRA, Daan. Continuous control with deep reinforcement learning. *CoRR*. 2015, vol. abs/1509.02971. Available also from: `http://arxiv.org/abs/1509.02971`.
7. SILVER, David; LEVER, Guy; HEESS, Nicolas; DEGRIS, Thomas; WIERSTRA, Daan; RIEDMILLER, Martin. Deterministic Policy Gradient Algorithms. In: JEBARA, Tony; XING, Eric P. (eds.). *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*. JMLR Workshop and Conference Proceedings, 2014, pp. 387–395. Available also from: `http://jmlr.org/proceedings/papers/v32/silver14.pdf`.
8. SUTTON, Richard S. Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming. In: *In Proceedings of the Seventh International Conference on Machine Learning*. Morgan Kaufmann, 1990, pp. 216–224.

9. GU, Shixiang; LILLICRAP, Timothy P.; SUTSKEVER, Ilya; LEVINE, Sergey. Continuous Deep Q-Learning with Model-based Acceleration. *CoRR*. 2016, vol. abs/1603.00748. Available also from: `http://arxiv.org/abs/1603.00748`.

10. COSTA, B.; CAARLS, W.; MENASCHÉ, D. S. Dyna-MLAC: Trading Computational and Sample Complexities in Actor-Critic Reinforcement Learning. In: *2015 Brazilian Conference on Intelligent Systems (BRACIS)*. 2015, pp. 37–42. Available from DOI: `10.1109/BRACIS.2015.62`.

11. MARTÍN ABADI et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. Available also from: `http://tensorflow.org/`. Software available from tensorflow.org.

12. BROCKMAN, Greg; CHEUNG, Vicki; PETTERSSON, Ludwig; SCHNEIDER, Jonas; SCHULMAN, John; TANG, Jie; ZAREMBA, Wojciech. *OpenAI Gym*. 2016. Available from eprint: `arXiv:1606.01540`.

13. BELLMAN, Richard. *Dynamic Programming*. 1st ed. Princeton, NJ, USA: Princeton University Press, 1957. Available also from: `http://books.google.com/books?id=fyVtp3EMxasC&pg=PR5&dq=dynamic+programming+richard+e+bellman&client=firefox-a#v=onepage&q=dynamic%20programming%20richard%20e%20bellman&f=false`.

14. SUTTON, Richard S. Learning to predict by the methods of temporal differences. *Machine Learning*. 1988, vol. 3, no. 1, pp. 9–44. ISSN 1573-0565. Available from DOI: `10.1007/BF00115009`.

15. RUMMERY, G. A.; NIRANJAN, M. *On-Line Q-Learning Using Connectionist Systems*. 1994. Technical report.

16. DEGRIS, Thomas; WHITE, Martha; SUTTON, Richard S. Off-Policy Actor-Critic. *CoRR*. 2012, vol. abs/1205.4839. Available also from: `http://arxiv.org/abs/1205.4839`.

17. WATKINS, Christopher J. C. H.; DAYAN, Peter. Q-learning. *Machine Learning*. 1992, vol. 8, no. 3, pp. 279–292. ISSN 1573-0565. Available from DOI: `10.1007/BF00992698`.

18. GRONDMAN, Ivo; VAANDRAGER, Maarten; BUSONIU, Lucian; BABUSKA, Robert; SCHUITEMA, Erik. Efficient Model Learning Methods for Actor-Critic Control. *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics : a publication of the IEEE Systems, Man, and Cybernetics Society*. 2012, vol. 42 3, pp. 591–602.

19. HORNIK, K.; STINCHCOMBE, M.; WHITE, H. Multilayer Feedforward Networks Are Universal Approximators. *Neural Netw.* 1989, vol. 2, no. 5, pp. 359–366. ISSN 0893-6080. Available from DOI: `10.1016/0893-6080(89)90020-8`.

20. TSITSIKLIS, J. N.; ROY, B. Van. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*. 1997, vol. 42, no. 5, pp. 674–690. ISSN 0018-9286. Available from DOI: `10.1109/9.580874`.

21. RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1. In: RUMELHART, David E.; MCCLELLAND, James L.; PDP RESEARCH GROUP, CORPORATE (eds.). Cambridge, MA, USA: MIT Press, 1986, chap. Learning Internal Representations by Error Propagation, pp. 318–362. ISBN 0-262-68053-X. Available also from: `http://dl.acm.org/citation.cfm?id=104279.104293`.

22. TIELEMAN, T.; HINTON, G. RMSprop Gradient Optimization. Available also from: `http://www.cs.toronto.edu/%5C~%7B%7Dtijmen/csc321/slides/lecture%5C_slides%5C_lec6.pdf`.

23. KINGMA, Diederik P.; BA, Jimmy. Adam: A Method for Stochastic Optimization. *CoRR*. 2014, vol. abs/1412.6980. Available also from: `http://arxiv.org/abs/1412.6980`.

24. IOFFE, Sergey; SZEGEDY, Christian. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*. 2015, vol. abs/1502.03167. Available also from: `http://arxiv.org/abs/1502.03167`.

25. LIN, Long-Ji. *Reinforcement Learning for Robots Using Neural Networks*. Pittsburgh, PA, USA: Carnegie Mellon University, 1992. PhD thesis. UMI Order No. GAX93-22750.

26. GOODFELLOW, I. J.; MIRZA, M.; XIAO, D.; COURVILLE, A.; BENGIO, Y. An Empirical Investigation of Catastrophic Forgetting in Gradient-Based Neural Networks. *ArXiv e-prints*. 2013. Available from arXiv: `1312.6211 [stat.ML]`.

27. SCHAUL, Tom; QUAN, John; ANTONOGLOU, Ioannis; SILVER, David. Prioritized Experience Replay. *CoRR*. 2015, vol. abs/1511.05952. Available also from: `http://arxiv.org/abs/1511.05952`.

28. HAUSKNECHT, Matthew J.; STONE, Peter. Deep Reinforcement Learning in Parameterized Action Space. *CoRR*. 2015, vol. abs/1511.04143. Available also from: `http://arxiv.org/abs/1511.04143`.

29. HEESS, Nicolas; WAYNE, Greg; SILVER, David; LILLICRAP, Timothy P.; TASSA, Yuval; EREZ, Tom. Learning Continuous Control Policies by Stochastic Value Gradients. *CoRR*. 2015, vol. abs/1510.09142. Available also from: `http://arxiv.org/abs/1510.09142`.

30. MNIH, Volodymyr; BADIA, Adrià Puigdomènech; MIRZA, Mehdi; GRAVES, Alex; LILLICRAP, Timothy P.; HARLEY, Tim; SILVER, David; KAVUKCUOGLU, Koray. Asynchronous Methods for Deep Reinforcement Learning. *CoRR*. 2016, vol. abs/1602.01783. Available also from: `http://arxiv.org/abs/1602.01783`.

31. CLEVERT, Djork-Arné; UNTERTHINER, Thomas; HOCHREITER, Sepp. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *CoRR*. 2015, vol. abs/1511.07289. Available also from: `http://arxiv.org/abs/1511.07289`.

32. OPENAI. *Pendulum-v0* [online]. 2017 [visited on 2016-01-01]. Available from: `https://gym.openai.com/envs/Pendulum-v0`.

33. OPENAI. *Reacher-v1* [online]. 2017 [visited on 2016-01-01]. Available from: `https://gym.openai.com/envs/Reacher-v1`.

34. OPENAI. *MountainCar-v0* [online]. 2017 [visited on 2016-01-01]. Available from: `https://gym.openai.com/envs/MountainCar-v0`.