symmetry

Article

The Multi-Dimensional Actions Control Approach for Obstacle Avoidance Based on Reinforcement Learning

Menghao Wu [1,2], Yanbin Gao [1,*], Pengfei Wang [1], Fan Zhang [1] and Zhejun Liu [1]

1 College of Intelligent Systems Science and Engineering, Harbin Engineering University, Harbin 150001, China; wumenghao@hrbeu.edu.cn (M.W.); hbforwpf@hrbeu.edu.cn (P.W.); zhangfan41@hrbeu.edu.cn (F.Z.); lzj1436791649@outlook.com (Z.L.)
2 Department of Computer Science, Aalto University, 02150 Espoo, Finland
* Correspondence: gaoyanbin@hrbeu.edu.cn

Abstract: In robotics, obstacle avoidance is an essential ability for distance sensor-based robots. This type of robot has axisymmetrically distributed distance sensors to acquire obstacle distance, so the state is symmetrical. Training the control policy with a reinforcement learning method is a trend. Considering the complexity of environments, such as narrow paths and right-angle turns, robots will have a better ability if the control policy can control the steering direction and speed simultaneously. This paper proposes the multi-dimensional action control (MDAC) approach based on a reinforcement learning technique, which can be used in multiple continuous action space tasks. It adopts a hierarchical structure, which has high and low-level modules. Low-level policies output concrete actions and the high-level policy determines when to invoke low-level modules according to the environment's features. We design robot navigation experiments with continuous action spaces to test the method's performance. It is an end-to-end approach and can solve complex obstacle avoidance tasks in navigation.

Keywords: continuous control; reinforcement learning; obstacle avoidance

Citation: Wu, M.; Gao, Y.; Wang, P.; Zhang, F.; Liu, Z. The Multi-Dimensional Actions Control Approach for Obstacle Avoidance Based on Reinforcement Learning. *Symmetry* **2021**, *13*, 1335. https://doi.org/10.3390/sym13081335

Academic Editor: Jan Awrejcewicz

Received: 28 June 2021
Accepted: 22 July 2021
Published: 24 July 2021

1. Introduction

For mobile robots, obstacle avoidance is an essential and core research area for safe navigation. A good obstacle avoidance control policy can also improve the local motion control ability in path planning work. However, the classical programming methods can be time-consuming. It needs many iterations to do jobs like fine-tuning to improve the performance. In general, distance measuring sensors like sonar are used to detect the environment and do the collision avoidance. Usually, these sensors are arranged axisymmetrically with the robot's heading direction as the axis. The acquired status is distributed symmetrically, and the sensors with shorter distances are used to judge the orientation of obstacles. To optimize performance, it should take lots of environmental scenarios into account to test the control policy to improve the overall behavior [1].

In the artificial intelligence domain, solving the control problems with a deep reinforcement learning (DRL) algorithm is effective since it does not need too much manual work on parameters tuning and expert knowledge. It can increase the robotics' autonomy, and flexibility for acquiring skills [2]. Reinforcement learning (RL) is a mathematical framework for which agents can learn control policy through trial and error [3]. DRL methods enrich agents with a self-learning ability, so they have a well adaptive ability. In some of the successful works such as [4,5], they trained intelligent agents in an end-to-end manner and completed several goal-directed gaming tasks. Gu, Holly, and Lillicrap [6] use reinforcement learning (RL) algorithms to deal with the robot manipulation problem. For obstacle-avoiding robots, the sensory input can be regarded as the states of the environment, and the operation of velocity and direction can be seen as actions [7]. Using RL methods can map the symmetrically distributed sensors' input to the robot's left and right steering action output and guide the agent to process the state input reasonably and avoid the collision. Using RL algorithms to train robots has become a researching hotspot.

Based on the forms of action spaces in an RL environment, algorithms can be roughly divided into two categories: methods in discrete action space and continuous action space. The discrete-action algorithms are often adopting the value-based RL methods. Their manner is to assign quality values to state–action pairs, update the state–action pairs with the high values in the iterative process, and then form a control strategy. The typical algorithms such as Deep Q-Networks [5], Dueling Networks [8], Double Q-Learning [9], soft actor–critic [10], etc. These can also be named the indirect learning methods because the learning process is driven by value evaluation. The ultimate goal is to improve the control policy via assignment. One drawback of the tasks with discrete action space is that it still suffers from the curse of dimensionality [11].

The RL algorithms in continuous action space usually adopt the policy optimization manner, and the policy outputs action directly instead of action value like value-based methods. These typical methods such as policy gradient methods [12]. To solve the convergence of policy gradient methods, Konda and Tsitsiklis present the Actor–Critic algorithm [13], O'Donoghue, Munos, and Kavukcuoglu present the PGQL algorithm [14]. Both methods take advantage of the value-based approach and policy gradient method. These methods directly optimize the control policy and draw on the benefits of value assignment manner. These methods are more widely used for real-world robotic tasks because the continuous action space corresponds to the robotic mission. Moreover, they can also be applied to solve tasks in discrete action space, but not vice versa.

In most experiments above, researchers applied RL algorithms to train agents to control a single-dimension action, such as controlling the vehicle's driving direction. This method can keep a stable output of policy. In addition, most of the algorithms are designed initially to solve the single action space task. Although policy-based methods can output multiple dimensional actions by adding parallel networks, such a method lacks a high-level policy to manage the outputs and a lack of research. In navigation environments, DRL algorithms are mainly used to output the steering angle, the robot's one-dimensional action. Suppose the robot can control two-dimensional actions simultaneously according to environmental features, such as driving direction and speed. In that case, the tasks that the robot can operate can be at a more challenging level. If it has a high-level strategy to deploy the two-action cooperation, agents can play a good performance over complex tasks.

Hierarchical reinforcement learning (HRL) can decompose a task into several sub-tasks and solve each job with a sub-model which will be more potent than solving the entire task with one model [15]. Compared with the standard RL, HRL has more complex decision-making ability, and its policy has multiple levels that operate each sub-task individually. The hierarchical framework can reduce the computational complexity of solving extensive Markov decision processes (MDP) and improve the exploration [16]. Simultaneously, the prior knowledge can be conveniently incorporated into the agent with a partial program, and the domain-specific information can be used to define low-level rewards. It dramatically decreases the agent's random exploration and guides the agent in advance through high-level policy based on the prior knowledge of the environment. However, many hierarchical reinforcement learning algorithms are designed for solving tasks with sub-tasks, and sub-policies are used independently to solve particular scenarios. In addition, designing a concise high-level control policy is complex for tasks in large domains. An essential goal of reinforcement learning is to minimize the designer's role [17]. Our method uses a hierarchical frame. The policies' combination of sub-modules controls multiple dimensional actions at the same time. The whole training process is consistent with the standard RL setting, the agent explores without knowledge, and the reward fully drives the learning process.

Contribution: We solve the obstacle avoidance task by introducing a hierarchical frame-based multiple actions control algorithm (MDAC). The method can cope with functions with numerous actions in continuous spaces. Moreover, it can efficiently complete the tasks without prior knowledge. We design three navigation environments to test and analyze the performance of the algorithm. Based on the frame, we also extend the structure to explore techniques to improve the performance. In the method, the concepts of manager and worker are introduced. Where the manager assigns work to workers, and the latter are responsible for their actions. During training, the robot can coordinate multiple actions to complete more complex tasks. Compared with the traditional reinforcement learning algorithm, our method can tackle a more complex environment, converge better, and, after convergence, it can maintain a stable output. Our method is more consistent with the classic RL settings with minimized human intervention. Many RL techniques can be directly applied to this method. It is an end-to-end training approach in a partially observable environment.

The rest of the paper is organized as follows: Section 2 gives a review of related approaches in the continuous control area and robotics applications. In Section 3, we provide the background knowledge and problem setup of reinforcement learning. Section 4 proposes our approach's frame and pseudocode. In Section 5, we present the experimental performance in navigation environments. Some general discussions are presented in Sections 6 and 7 concludes the paper with some potential use cases.

2. Related Work

Arsh et al. [18] presented an action branching architecture for control multiple actions. The architecture independently optimizes each dimensional action with its state and reward information by combining multi-dimensional actions to achieve the target. They presented various kinds of branching networks. The critical insight is to combine multiple independent modules. It is used to tackle tasks with a low correlation between actions, and the actions' influences on the state are in different directions. Tampuu et al. [19] proposed the multiple value-based agents' method to explore the high-dimensional environment. These agents have shared input but independent Q-learning modules. The multi-agent system is also a researching focus. It is a valuable idea to complete one task through the cooperation of multiple agents. However, the convergence is an issue to implement them. Luke et al. [20] used a sequence-to-sequence network architecture to tackle high dimensional continuous control problems. The method needs manual work on designing action dimensions, and, once the number of actions increases, the Q-value estimates may become inaccurate to use.

HRL is a series of methods which are using different levels of policies to cope with complex tasks. It is a long-standing researching area, and the core idea of HRL tasks [2,21] is to decompose a whole task into several parts and solve them separately. Its target is to discover and exploit the hierarchical frame within a Markov decision problem (MDP) [22]. The hierarchical neural network trains several sub-tasks to find a recursively optimal policy. Similar to this kind of method, MDAC also adopts the hierarchical policy structure. The difference is that we focus on and solve a complete task rather than split a task. As the hierarchical structure is used for reference in the method, the following is a brief introduction of HRL. The frameworks of HRL have a variety of designs according to task requirements. The mainstream design is to set separate sub-networks for solving sub-tasks and apply a leading neural network to do the overall planning. Jun et al. [23] used a two-layered HRL architecture to solve a real pole balancing task. The architecture has high and low levels for planning the entire mission, selecting the next sub-goal, and dealing with the sub-task to control the actuators. Kulkarni et al. [24] proposed the hierarchical-DQN framework, which can efficiently explore the environment and apply it to the classic ATARI series game. They divide a task into several unrelated sub-tasks and complete them separately with low-level policy. The adopted framework integrates DRL with hierarchical value functions, making the agent motivated to solve intrinsic targets to enhance exploration. Thomas et al. [25] decomposed the Markov decision process and value function into hierarchical ones and presented the HRL framework that converges to an optimal policy much faster than the flat Q learning approach. Alexander et al. [21]

introduced a feudal network for HRL. The method introduces the concept of the manager and worker modules. The manager is in charge of lower temporal resolution and sets goals for the worker module.

As for the solution for the tasks in continuous action space, HRL also has its unique advantages, which can further solve the task with harder difficulty. Because in the frame pattern, any algorithm suitable for the task can be placed. It has good expandability. Yang et al. [26] presented a hierarchical DRL algorithm to solve the continuous control problem. Their method can learn several essential skills at the same time. The architecture has a critic network and multi-actor networks for learning different skills. However, the algorithm requires predefined reward functions for all levels of hierarchy and needs a fully observable environment. Bacon et al. [27] propose the gradient-based option-critic architecture, which can learn both the internal policy and the termination functions. Their method does not need to provide additional rewards and subgoals. To improve the training efficiency, Ofir et al. [28] used the off-policy experience for hierarchy training and introduced an off-policy correction to address the dynamic issue of high-level action space. Their HRL method requires fewer environment interactions than on-policy algorithms.

3. The Problem Setup

Reinforcement learning (RL) is an artificial intelligence technique that adopts an agent to gather task solution skills from trial and error. The agent straightforwardly learns knowledge by interaction with environments. Surely, there are certain requirements for RL's environment to realize the application of algorithms. We use $\mathcal{E}$ to indicate the environment. Several elements of RL are time step t, state s_t, action, a_t, and reward r. An RL task can be formulated as a Markov Decision Process (MDP), consisting of state space $\mathcal{S}$, action space $\mathcal{A}$, transition function $\mathcal{P}$, reward space $\mathcal{R}$, and a discount factor γ. Markov property can be defined as $p(s_{t+1}|s_1, a_1, \ldots, s_t, a_t) = p(s_{t+1}|s_t, a_t)$. The Markov property indicates that the future states are conditionally independent of the past given the present. Thus, in an RL task, the decisions and values are assumed to be a function only of the current state [7]. The main purpose of an agent is to train its policy to get the maximum cumulative reward. When it makes a wrong action, it will be punished. Through this mechanism, the agent can achieve the purpose of self-learning through interaction. This process continues until the agent reaches a terminal state. The discount factor $\gamma \in (0, 1]$ trades off the importance of immediate and future rewards [8].

The value function is used to measure the action value by estimating its long-term reward in the RL system. It is a core component in value-based RL algorithms. The quality function or state–action-value function $Q^{\pi}_{s,a}$ is a mainly used indicator to measure the quality of action's selection. It defines the value of choosing action a under a given state s and following a policy π:

$$Q^{\pi}(s, a) = \mathbb{E}[R|s, a, \pi] = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k r_{t+k}|s, a, \pi] \tag{1}$$

The training target is to find the best policy which can lead to the highest quality value $Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a)$.

It is an indirect way to learn policy π based on the estimate of the value function. It is suitable for tasks in discrete action space. For the problem in the continuous action space, we need to optimize the policy directly. The policy-based algorithms parameterize the control policy as $\pi(a|s; \theta)$ and optimize the parameter during training. The policy gradient method [12], one of the policy-based approaches, provides the gradient of object function $J(\pi)$ with respect to the parameters of policy π:

$$\begin{aligned} \nabla_{\theta} J(\pi_{\theta}) &= \int_{\mathcal{S}} \rho^{\pi} \int_{\mathcal{A}} \nabla_{\theta} \pi_{\theta}(a|s) Q^{\pi}(s, a) da ds \\ &= \mathbb{E}_{s \sim \rho^{\pi}, a \sim \pi^{\theta}} \left[\nabla_{\theta} \log \pi^{\theta}(a|s) Q^{\pi}(s, a) \right] \end{aligned} \tag{2}$$

The policy gradient method is difficult to converge without a clear learning signal. By combining the value-based methods, the actor–critic method has been well applied in many continuous tasks [29]. Because the continuous action space is widely used in the robotics field, and the continuous control algorithms can be applied in more areas besides discrete control tasks. Therefore, we aim at the problem in continuous action space. Our method is based on the classical actor–critic algorithm [13], which combines the benefit from value-based methods and policy-based methods. The actor $\pi_\theta : \mathcal{S} \to \mathcal{A}$ maps representation of the states to action-selection probabilities. The critic $Q^\pi_\phi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ estimates the expected return and maps states to the expected cumulative reward.

3.1. Deep Network

The deep neural network has a powerful model fitting ability, and it is widely used in various fields of deep learning such as image processing, video, and audio processing. With the deep neural network, RL can be more effective in learning skills. Deep RL (DRL) applies neural nets for representing the value functions or control policies. There are a series of successful DRL techniques like Deep Q-Network [5], achieving human-level performance on Atari games from video input. It adopts the target network and experience replay skills for stable training; these techniques make the learning process more robust. The parameters of the neural network will be modified iteratively to minimizing the loss function:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}[(y_i^{DQN} - Q(s,a;\theta_i))^2] \tag{3}$$

with

$$y_i^{DQN} = r + \gamma \max_{a'} Q(s',a';\theta^-) \tag{4}$$

RL researchers saw the potential of DRL after the DQN algorithm was presented. A number of researchers have done a lot of work and put forward many excellent algorithms such as double DQN [30], dueling Q-network [8], value iteration network [31], priority experience replay [32], etc.

For the policy-based RL algorithms such as the actor–critic and policy gradient method, researchers also adopted a deep neural network to improve their model's learning ability. In this kind of task, the feature of the environment is taken as the input of the network, and the output of it is taken as the action of the agent. Thus, the trained policy of the neural network, or the agent, outputs right actions by perceiving the change of the environment. The whole process is driven by the promotion of reward. In this way, the classical RL uses the neural network and is able to solve a more complex task than before. Lillicrap et al. [33] presented the deep deterministic policy gradient (DDPG) algorithm, which uses the actor–critic method's framework and some insights from DQN. It has been used in many physics tasks both in a simulator and in a real robot. In the method, both actor and critic components use a neural network to learn a deterministic policy and approximate the Q value of the policy.

3.2. Hierarchical Architecture

With the development of artificial intelligence, the tasks that need to be processed have become complex, such as large-scale planning tasks [34] and long-horizon and sparse reward tasks [35]. They drive the reinforcement learning researchers to explore more complex models. HRL has become a structure to explore in its domain. There are some HRL works [26,36,37] that use a high-level policy to select low-level policy. Each sub-policy is used to process a sub-task. HRL methods rely on hierarchies of features, and different levels of policies control different actions. Since a high-level policy is used, the overall action spaces decrease, and it has a more efficient exploration. The environment state can also be managed hierarchically. The low-level agent can receive the partial state allowed by the high-level agent. Feudal reinforcement learning defines the structure of control layers from top to bottom. High-level managers assign tasks to low-level workers. For the hierarchical frameworks, policies π_h are divided into high-level policy $\pi_h\left(a^h|s^h\right)$

and low-level policies $\pi_l\left(a^l|s^l, a^h\right)$. The high-level policy is responsible for the overall planning and enables some workers at some steps. In addition, it can also assign local goals for each sub-agent to complete and provide rewards. The HRL process follows the Semi-MDP, which includes the state S, options O, transition model P, reward function R, and discount factor γ, in which the transitions can last for several timesteps. The typical objective function in HRL is (5). The action of low-level policy directly brings the return, and its selection depends on the action of high-level policy. Similar to RL, the target is to maximize the objective function and find the best policies:

$$J(\pi) = \iint \sum \pi_h\left(a^h|s^h\right)\pi_l\left(a^l|s^l, a^h\right)Q^\pi(s, a^l)\mathrm{d}a^l\mathrm{d}s \tag{5}$$

Since it is a hierarchical network, it contains the high-level model and low-level models. Each time the agent interacts with the environment, all models jointly determine its actions and feedback to the environment, thereby obtaining rewards. Then, the distribution of rewards is also an area worth studying. For the high-level policy π_h, each time, it needs to decide to execute sub-models according to the environment $\mathcal{E}$. The executed sub-model π_l also selects actions according to the state s. The high and low-level models collaborate output actions a to get a reward r. At the same time, this MDP sequence will be saved to their respective memory unit R. They can find their effective actions through iteration to achieve higher returns. This design can ensure that each execution unit is not interfered with by other factors, and, each time, it is compared with its previous status and improves its policy. The sequences recorded by the memory are also valid in each step.

4. Proposed Method

To have a separate network for controlling every single action, we adopt a hierarchical structure Figure 1, and it has two-level policies. The top-down hierarchical structure is similar to the FeUdal HRL used to tackle works containing subtasks. In the feudal network structure, the manager is responsible for managing high-level tasks and assigning tasks to low-level workers as needed. Like the concept, the high-level and low-level modules in MDAC can be regarded as manager and workers. Usually, HRL performs tasks with multiple scenarios, and the correlation between each action in the task is relatively small. The difference is that low-level modules in the work are used for cooperation in one application scenario, and they have a higher degree of correlation. In such a scenario, better cooperation will make the task completion better and easier, while insufficient cooperation will significantly reduce the possibility of task completion.

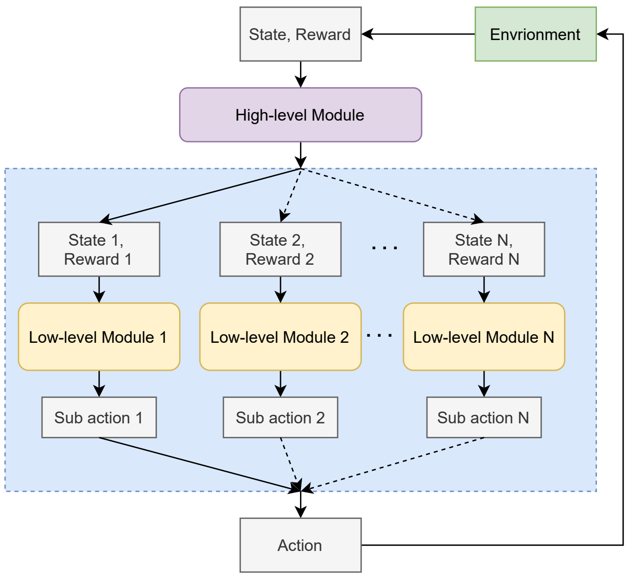

Figure 1. The basic frame of MDAC.

We name the high-level policy module 0 (M0) π_h, which is to used to learn high-level policy. M0 outputs either a discrete value or a real number between –1 to 1, which is converted into several discrete numbers to select sub-policy on the step. Accordingly, we name sub-modules M1, M2, . . . , and MN, and corresponding policies are π_1^1, π_1^2, π_1^N. Each sub-module is used to execute the concrete actions in each dimension. Among them, M1 is the main sub-module, which means it will be implemented in every step. In our application scenario, we use it to control the direction of the robot. Because the direction operation is essential for obstacle avoidance, it outputs actions according to state at each step to improve safety. On the contrary, other sub-modules are not always executed. It depends on the instructions of M0. They are executed when M0 allows. The action set bases on the cooperation policy of high and low-level modules and can be $\{a_l^1, a_l^1 + a_l^2, \ldots, a_l^1 + a_l^N\}$.

M0 will firstly receive the environment's state and rewards information, and then it assigns part of them to the corresponding sub-module. Of course, it can also transfer the same information and feedback to all sub-modules. It depends on which environmental information is essential and which is not needed for the sub-module. Transitions follow the Markov function $P(s,a,s') = Pr(s'|s,a)$. In this process, each state results from its previous state and action. It can be seen that the MDPs are satisfied with the M0 and M1 in the designed frame. For other sub-modules, they satisfy semi-MDPs, and they will not meet all the environment's transitions. For submodules, the output action will continue until the next executed step. The state is drawn from environment state-space $\mathcal{S}$. The state of all modules comes from this space. However, each module has its own action space, and we name them as $\mathcal{A}_0$, $\mathcal{A}_1, \ldots, \mathcal{A}_n$, respectively. In each transition, one or more actions from actions spaces are executed by the agent to interact with the environment. For the sub-module that is not executed, its memory unit will not add the transition in that step. The idea is that each module can make an independent judgment according to the environmental information it can receive. The sub-modules do not interfere with each other directly, and a high-level module assigns their coordination.

With this frame, in principle, any classical reinforcement learning algorithm can be used in each model. We use the actor–critic network based on the policy and value approaches, a well-applied continuous method to robot control. All modules have their replay memory. Experience replay is a technique to improve the training stability in reinforcement learning by efficiently using previous experience. It also leads to a better convergence behavior. Our method is uniformly sampling the tuples in memory units for off-policy training, and this is a conservative but effective way. In the training process, the robot selects high-reward actions at a high frequency in the initial stage. The disadvantage is that there are potentially higher-reward actions that have not been explored and executed. Thus, we added noise during the execution of actions to explore the action space, which is sampled from the Gaussian distribution. This method can increase the exploratory of each module in training. Through rich exploration, multiple experiences will be stored in memory. In the middle and later stages of training, the method will gradually reduce this noise to zero, which is also a way to improve its stability. After the trained policy is mature, the exploration is no longer needed.

For high-level module M0, the actor policy maps state space to high-level action space $\pi_{\theta_0}^a : \mathcal{S} \to \mathcal{A}_0$. The critic branch is based on the value update method. It is used for estimating expected return $Q_{0\omega} : \mathcal{S} \times \mathcal{A}_0$. We use θ and ω to represent actor and critic network parameters. Use the subscript number to represent the module number. The policies of actor and critic are represented by using π^a and π^c. Below, the equations for module 0 are listed. In the critic network, the Q value function is defined as Equation (6). Its value is the expectation of the cumulative discount reward of action a_0:

$$Q_0(s|a_0) = \mathbb{E}[R|s, a_0, \pi_0^c] = \mathbb{E}[\sum \gamma^k r_{t+k}|s, a_0, \pi_0^c] \quad (6)$$

The critic network's parameters are updated by minimizing the loss function (7). In Equation (7), $y_0^{DQN} = r + \gamma \max Q_0'(s', a_0'; \theta_0^-)$. It is the target Q network, and it uses the transitions in memory for off-policy training. The real-time working network Q_0 slowly tracks the target network Q_0' and updates its parameters in fixed iteration times. The Q_0 is like Equation (6), and it is the working critic network, and its parameters are about to be updated:

$$L(\theta_0) = \mathbb{E}[(y_0^{DQN} - Q_0(s, a_0; \theta_0))^2] \tag{7}$$

For the actor branch, it updates parameters with a policy gradient method, and the gradient is calculated by Equation (8). In the equation, Q_0 is the output from the critic's value function, and it gives quality value to the actor's output (action):

$$\begin{aligned}\nabla_{\omega_0} J(\pi^a_{0\omega_0}) &= \int_{\mathcal{S}} \rho^{\pi^a_0} \int_{\mathcal{A}_0} \nabla_{\omega_0} \pi^a_{0\omega_0}(a_0|s) Q_0{}^{\pi^a_0}(s, a_0) da_0 ds \\ &= \mathbb{E}_{s\sim\rho^{\pi^a_0}, a\sim\pi^{a\omega_0}_0}[\nabla_{\omega_0} log \pi^{a\omega_0}_0(a_0|s) Q_0{}^{\pi^a_0}(s, a_0)]\end{aligned} \tag{8}$$

Based on the above equations, we list the update rules of the high-level M0 as pseudocode (Algorithm 1).

Algorithm 1 Update rule of module M0

```
1:  Initialize:
2:  Initialize actor $\mu_0(s|\theta_0)$ and critic $Q_0(s, a_0|\omega_0)$ with weights $\theta_0$ and $\omega_0$
3:  Initialize target actor $\mu_0'$ and target critic $Q_0'$
4:  Initialize replay memory $R_0 = \emptyset$, random process $\mathcal{N}_0$.
5:  Updating Loop:
6:  for Step $t$ = 1, $T$ do
7:      With probability $\epsilon$ select module action $a_{0\_t} = \mu_0(s_t|\theta_0) + \mathcal{N}_{0\_t}$, otherwise select $a_{0\_t} = \mu_0(s_t|\theta_0)$
8:      Execute $a_{0\_t}$ and observe reward $r_t$ and new state $s_{t+1}$
9:      Store transition $(s_t, a_{0\_t}, r_t, s_{t+1})$ in $R_0$
10:     Sample a random minibatch of $N$ transitions $(s_i, a_{0\_i}, r_i, s_{i+1})$ from $R_0$
11:     Implement target actor $a'_{0-i+1} = \mu_0'(s_{i+1}|\theta_0')$
12:     Implement critic $Q'_{0-i+1} = Q_0'(s_{i+1}, a'_{0-i+1}|\omega')$
13:     Set $y_{0\_i} = r_i + \gamma Q'_{0-i+1}$ (set $y_{0\_i} = r_i$ if $s_{t+1}$ is terminal)
14:     Update critic network by minimizing the loss function as Equation (7):
            $L(\theta_0) = \frac{1}{N}\sum_i (y_{0\_i} - Q_0(s_i, a_{0\_i}|\theta_0))^2$
15:     Update actor network using the sampled policy gradient as Equation (8):
            $\nabla_{\omega_0} J \approx \frac{1}{N}\sum_i \nabla_{a_0} Q_0(s, a_0|\omega_0)|_{s=s_i, a_0=\mu_0(s_i)} \nabla_{\theta_0}\mu_0(s|\theta_0))|_{s_i}$
16:     Soft update target networks of critic and actor ($\tau \ll 1$):
            $\theta_0' \leftarrow \tau\theta_0 + (1-\tau)\theta_0'$    $\omega' \leftarrow \tau\omega_0 + (1-\tau)\omega_0'$
17: end for
```

The updating rule above is a typical algorithm for the actor–critic-based method. If other submodules use the same algorithm, the fundamental updating rules are the same as Algorithm 1, except that some submodules' state spaces are subsets of the whole environmental state-space $\mathcal{S}$. In each module, critics estimate Q value according to the action and state, and the actor will select an action to achieve a high Q value. The combination is easy to converge to the best strategy in continuous spaces.

The rewards' environment feedback to the agent every time can allocate the same reward value for the modules that participate in this step. This design is because each unit is an independent individual, and all modules need to find control policies to get higher rewards compared to their history. In each memory unit, the relative level of reward is the basis for measuring the transition quality. In the critic networks, we adopt the soft update technique to update parameters, which means the network parameters θ' slowly track the learned target networks θ: $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$ with $\tau \ll 1$. This method can ensure that the policy is steadily improved and prevent it from changing much to affect the entire learning process. Our method's overall process can be summarized in the form of pseudocode (Algorithm 2). Here, we take two sub-modules into account.

Algorithm 2 Multi-dimensional actions control approach

```
1: Initialize:
   M0, M1, and M2 actor–critic networks
2: for episode = 0 to M do
3:     for step $t$ = 0 to N do
4:         M0 receives state and outputs action $a_0$, pre-processes state for sub modules(optional)
5:         M1 receives state and outputs action $a_1$
6:         If M2 is enabled by M0:
7:             M2 receives state and outputs action $a_2$
8:             Agent implements action $a_1$ and $a_2$
9:         Else
10:            Agent implements action $a_1$
11:        M0 receives reward, assigns rewards to sub modules
12:        Store transitions to each individual's memory unit
13:        Train each networks by off-policy training like Algorithm 1, update parameters
14:    end for
15: end for
```

5. Experiments

We design three environments to simulate the process of obstacle avoidance and navigation based on Pyglet library. The state space is derived from the robot's sensors' information. The action spaces of the sub-modules are the speed and direction of the robot. The high-level module's action space is several discrete values that indicate to select whether to execute sub-modules or not. After the robot executes an action, the environment gives reward feedback and transit to the next state. The simulation environments have Markov properties. In the design of this experiment, the sub-actions work together to change the robot situation. The main sub-module M1 is executed every step. The other sub-module that controls the speed is not executed every time. It is mainly responsible for changing speed. After being executed, the speed will be kept until the subsequent execution. The width of the robot is 20 units. The robot has 25 distance sensors that are distributed in front of the robot. The farthest sensing distance is 50 (Figure 2). The state space is composed of these 25 distances.

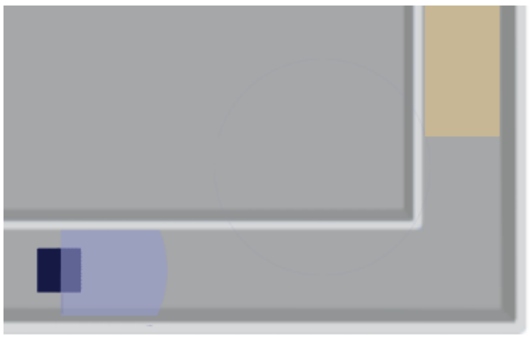

Figure 2. Maze 1.

5.1. Maze 1

The first scenario is a task to make a turn and reach the destination. As Figure 2 shows, the target area is in yellow. The robot starts each time from the lower-left corner, and the purpose is to reach the target without collision. The robot detects the distance from the obstacle through a 25-dimensional radar. Since the robot's width (20) is not very different from that of the road (40), the turning point is a difficult place to drive. It is easy to get collision. Thus, how to effectively operate the robot to steer perfectly on the corner becomes the key to complete the task.

In this environment, the driving speed and direction are in two continuous action spaces. The steering range contains 90 degrees to the left and right of the forward direction. Because DDPG is used to control one dimension (direction), we set several different speeds to explore the limit of the method in this scenario. The speed range is adjustable in the environment, and the range will be shown with specific experiments.

The starting point of each episode is at the bottom left corner, as Figure 2 shows, the robot's position. When the robot has a collision, the reward is -1, and a new episode starts from the starting point. When the robot is running safely, the reward is 0.001 for each step. When the robot moves to the target area, the reward is $0 - steps\_used/700$, and the new episode starts again. To accelerate the robot's speed and uses fewer whole steps per episode than running at a plodding speed to obtain each step's reward. We use the $-steps\_used/700$ in reward as a punishment. We set 1000 steps for each episode and train 1000 episodes in total.

Experimental Details

In the experiment, we test the performance of deep deterministic policy gradient (DDPG) [33], double DDPGs, and MDAC methods. The current mainstream for controlling multiple actions simultaneously is to parallel multiple modules to execute them, so the double-DDPG model is used for comparison. The double DDPGs method uses two DDPG modules and one for steering direction and speed adjusting. To control two actions at the same time, use DDPGs in parallel to explore the performance.

DDPG: For the DDPG method, the actor's network structure is 25 neurons in the input layer, 100 neurons in the first hidden layer, 20 neurons in the second hidden layer, and one neuron in the output layer. The activate functions of input and hidden layers are ReLU. The output layer's activate function is Tanh. The critic network structure is: the input layer has 26 neurons, the first hidden layer has 100 neurons, the second hidden layer has 20 neurons, and one neuron in the output layer. Table 1 lists detailed hyperparameters values of DDPG.

Table 1. Hyperparameter.

actor learning rate	1×10^{-4}	batch size	16
critic learning rate	1×10^{-4}	soft update factor τ	0.1
discounted factor γ	0.9	memory capacity	1000

The training results are shown in Figure 3. As the improvement of speed, convergence takes more time. When the speed is 37, the agent cannot complete the task. For the other speeds, the policies trained by DDPG can complete the task, and the robot can avoid obstacles and reach the target area. When the speed is higher than 37, the robot will hit the wall on the way. The improvement of speed increases the difficulty of the task. To show the paths at different speeds, we plot the routes map as Figure 4. For speed 37, it is impossible to make a good turn before hitting the obstacle. The best route is that, when the speed is 20, it is closest to the inner turning point and reaches the destination with the least number of steps. It also shows the advantage if the robot can control the speed. The robot will slow down at the right corner to improve the possibility of steering success.

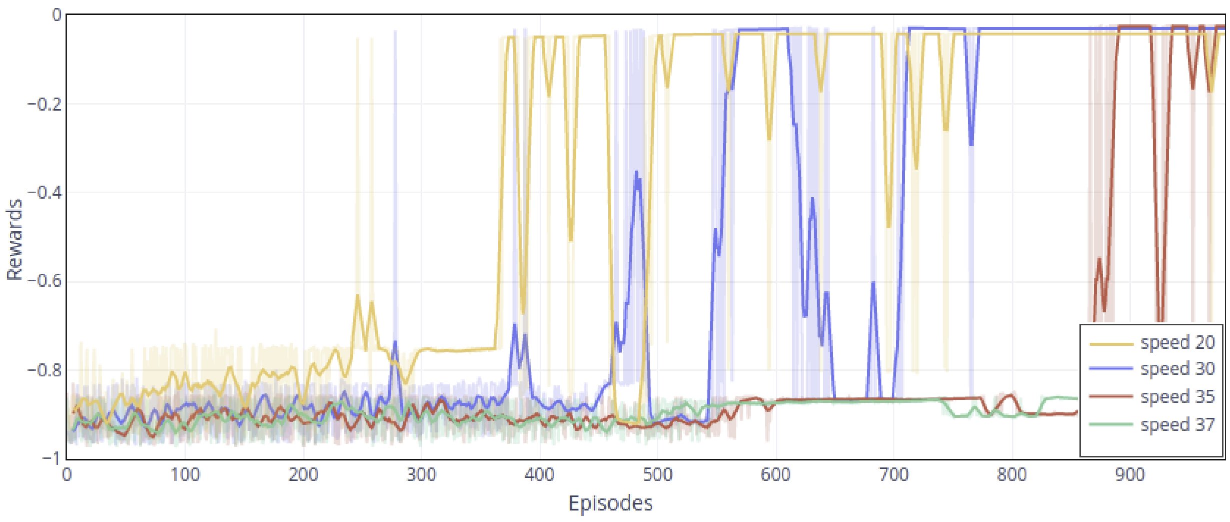

Figure 3. The training results using the DDPG method.

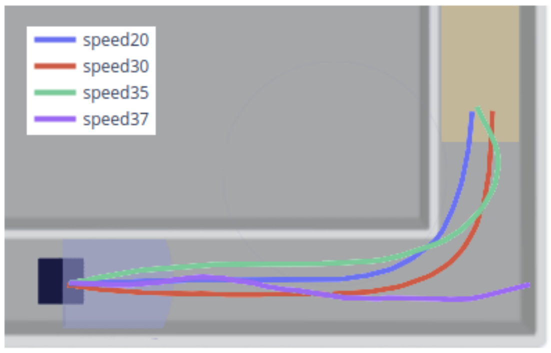

Figure 4. Routes of different speeds.

Double DDPG: The double DDPGs method uses two DDPG modules for control speed and direction, respectively, and the structures and parameters of each one are the same as the DDPG's method. The default speeds are 40 and 60, and the speed rate can be adjusted from 0.4 to 1.4. The training results are as shown in Figure 5. For the speed 60, the whole process behaves terribly, and it is hard to converge, only reaching the target area a few times. Still, it is difficult for the overall policy to maintain stability. For the default speed 40 trial, it tends to converge at around the 800th episode. The policy can reach the target area, but the later stage is not stable. Form the original light rewards, and the robot often hits obstacle before reaching the target area.

Since the speed 60 trial does not converge and behaves badly, here we only plot the overall speed curve at the default speed 40 as the left side in Figure 6. It can be seen that the trained policy plans to reduce speeds near the environment's corner (at step 31) to avoid an obstacle. However, each step's velocity is changing, which introduces an unstable robot's state and makes it easy to crash. It also does difficult policy training. Similarly, we plot the typical route as the right side in Figure 6 with the default speed of 40. The path can reach the target area safely.

MDAC: The MDAC method can control the direction and driving speed simultaneously. It uses three actor–critic-based structures, two sub-module control driving direction and speed, and the high-level module selects to execute the sub-module for the current state. The design of the network structure and the setting of the three modules' hyperparameters is the same as the DDPG method. The training results are shown in Figure 7.

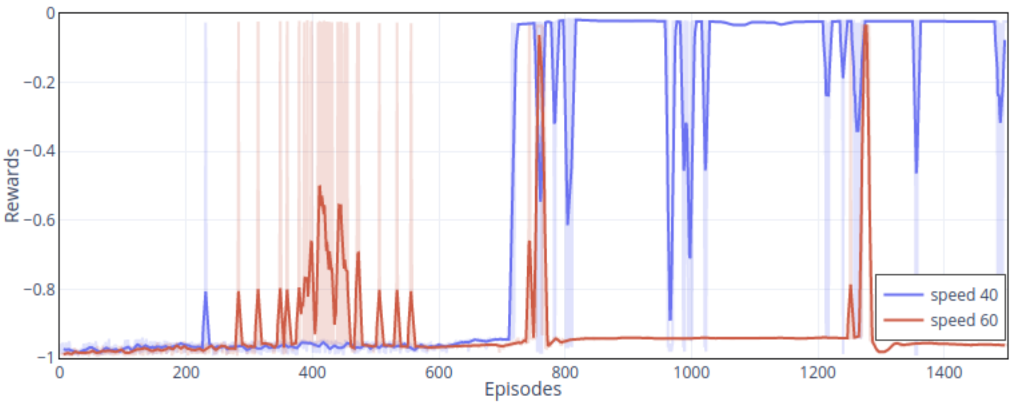

Figure 5. The training rewards using the double DDPGs method.

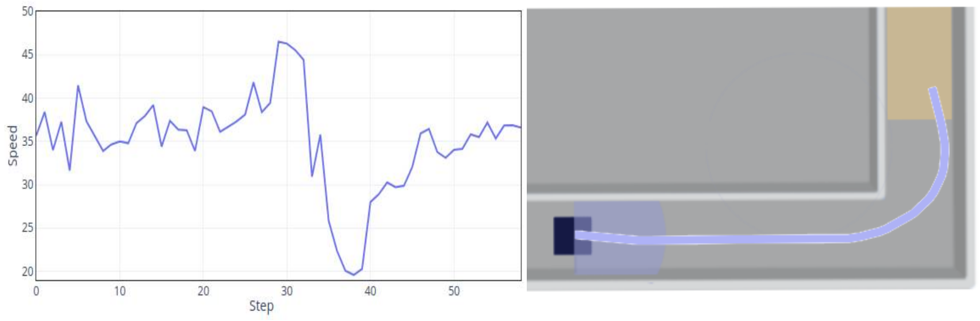

Figure 6. The speed (**left**) and route (**right**) record using double DDPGs with default speed 40.

The light lines in Figure 7 are the original data, and we smooth them for the convenience of comparison. In this experiment, we set the speed range as 0.4 to 1.4 times of default speeds, and default speeds are 40 and 60 separately. It shows that both of them tend to converge during the first 600 and 700 episodes. The higher speed one uses more time than speed 40. In terms of the stability after convergence, the lower speed one is better. To intuitively show their speed adjustment strategies, we choose a representative speed record from two experiments as the left in Figure 8. Because they have different speeds, the steps used in one episode are varied. For speed 40, the adjustment strategy is that the initial velocity is about 50, and it tends to increase, but it quickly decreases speed to near 20 at the right corner (step 29). After the turn, it gradually accelerates to reach the target area. For speed 60, its adjustment strategy during this episode is pronounced. The high-level policy only enables speed adjustment at the corner (step 21) and decelerates it to 25. After the turn, it slightly speeds up and maintains this until completing the task—both policies take less time and steps to complete the task than policy trained by DDPG or double DDPGs. Moreover, compared with double DDPGs, the speed adjustment module is not be enabled when not needed. Thus, the speed of the robot will not change Unstably. It increases the difficulty of training and the uncertainty of states.

Figure 8 right shows the routes with two speeds, 40 and 60. Both paths trained by MDAC are very similar. The curves are also identical to the route with speed 20 trained with DDPG. The path is time-saving to the destination and avoids obstacles around the turning area.

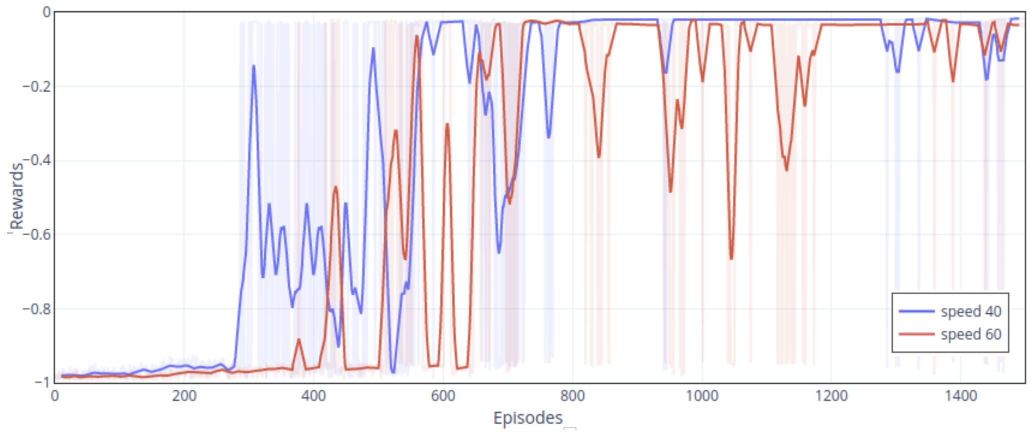

Figure 7. The training result using MDAC.

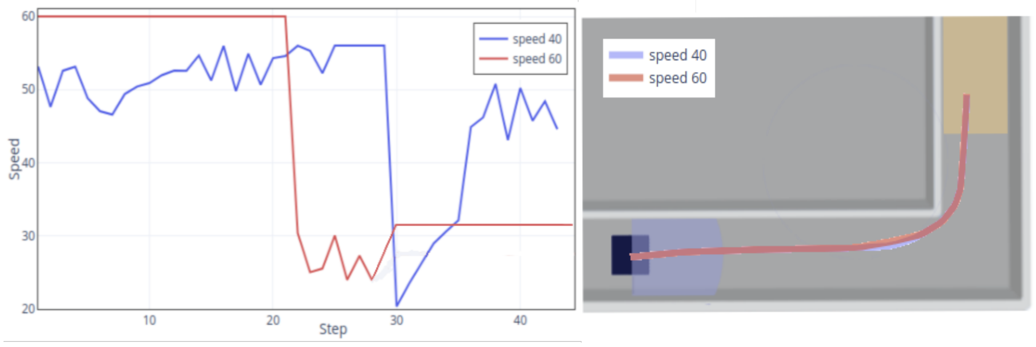

Figure 8. The speeds (**left**) and routes (**right**) records using the MDAC method.

5.2. Maze 2

In this environment (Figure 9), we test the training stability, path, and speed planning ability in a long-path maze. The width of the road gradually changes, and the yellow part is the targeted area. The starting point of the path is in the lower-left position. The turning angles are more diverse. The path becomes narrow from the upper right corner, which makes the robot's turning difficult. On the upper left corner, the turning angle is a right-angle. Finally, reach the targeted area to complete a mission round. The environment contains more scenarios and conditions, so it has higher requirements for policy training.

The robot has two action spaces, driving speed and forward direction. The adjustable range of speed is 0.4 times to 1.4 times the default speed. We set the default speed as 50. The control range of the forward direction is 90 degrees left and right. When the robot hits the boundary of the maze, it gets a penalty of -1. Every step the robot takes, it gets a small reward of 0.0001. When the robot completes a circle and reaches the specified number of steps, the reward is $0 - stepused/600$. When the robot hits or completes a round, it achieves an episode. The cumulative reward will be recorded.

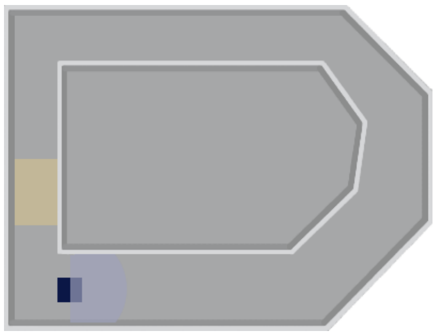

Figure 9. Maze 2.

Experiment Details

In the experiment, we use double DDPGs and MDAC methods for training. The neural network and the number of neurons used are also the same. Hyperparameters are as Table 1. The training results are shown in Figure 10.

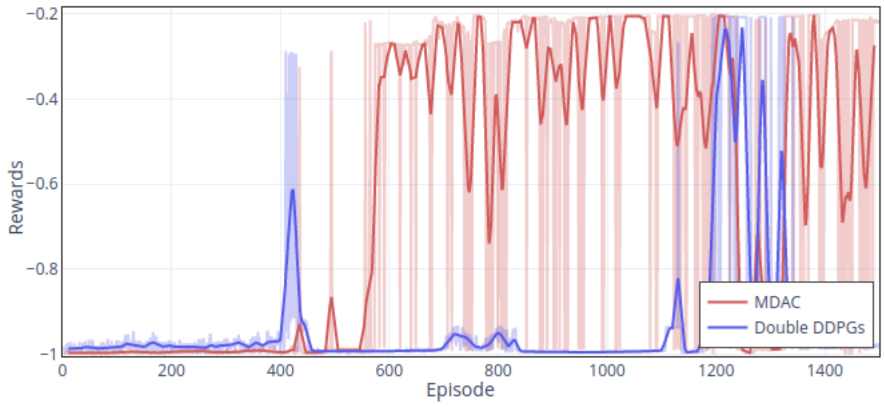

Figure 10. Training results using MDAC and double DDPGs.

To show the overall trends of cumulative rewards, we smoothed the results. The darker lines are the result of smoothing. The light-colored line is the original result. The frequency of reaching the target with the MDAC method is significantly higher than that of double DDPGs. Both policies reach the target area in the initial stage but used more steps, and the total rewards are about -0.3 (episode 400 and 500). Then, policies gradually increase rewards to about -0.2, indicating that their policies are intended to improve the overall speed to acquire higher rewards. Starting from the 600th episode, the MDAC's agent can reach the target area more stably, and the rewards gradually increase. For the double DDPGs, although it reaches the target in some episodes (episode 400, 1200, etc.), it is precarious. In most episodes, it is difficult to get to the end and collide with obstacles. For each trained policy, we plot the speed curves and trajectories.

Figure 11 shows the speed and routes' records in default 50 speed. From the speed figure (Left), two speeds' control policies are similar, and the speeds appropriately reduce when turning, especially at the last turn (Step 110). The upper left's final turn is the narrowest, so both policies reduce the speed as much as possible. The difference between the two is also apparent. The MDAC's policy can maintain a speed when its adjustment is not required, making the whole training process and states more stable. The double DDPGs' speed keeps changing, which is not conducive to training stability. The instability can also be seen in the rewards' result figures. The right part of Figure 11 is the trajectories of two methods, not much difference.

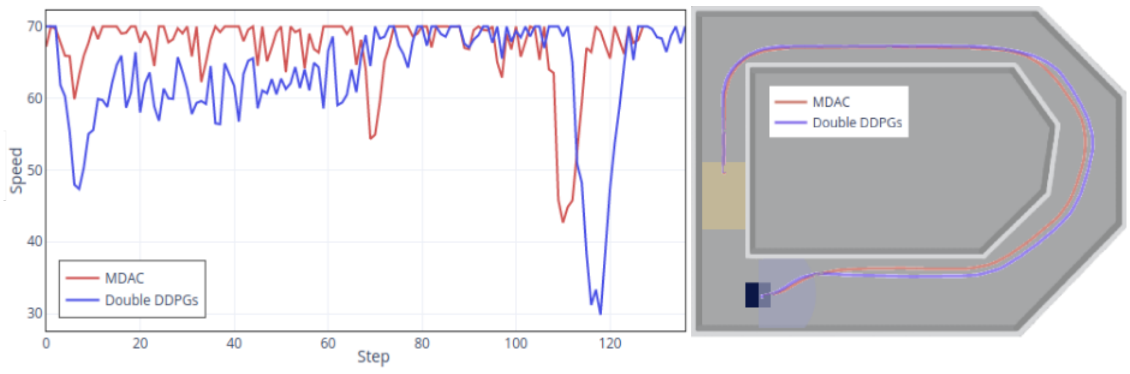

Figure 11. The speeds and routes' records using MDAC and double DDPGs.

In this environment, the agent encounters more state information and scenarios that require policy learning. It is more complex than the turning trial. We tested two methods that can control speed and direction. The advantage of MDAC is it can enable the sub-policy as needed and maintain its action when not required, and this method avoids unnecessary interference. The high-level module helps improve the overall training stability.

5.3. Maze 3

The above experiments compared the methods for single scenarios. This experiment (Figure 12) combines various scenes, and it has a longer path to the target area. In the map, we increase the difficulty by narrowing the width of the road, adding obstacles (darker blocks) and sharp turns. These features enrich the state of the environment and mainly affect the policy's upgrade in the training stage. A robot can easily collide with those and end the episode. We add more robot information for the state space, including the robot's position, heading direction, and speed. Thus, the training network can obtain more robot motion attributes. All state data are normalized to the range between 0 to 1.

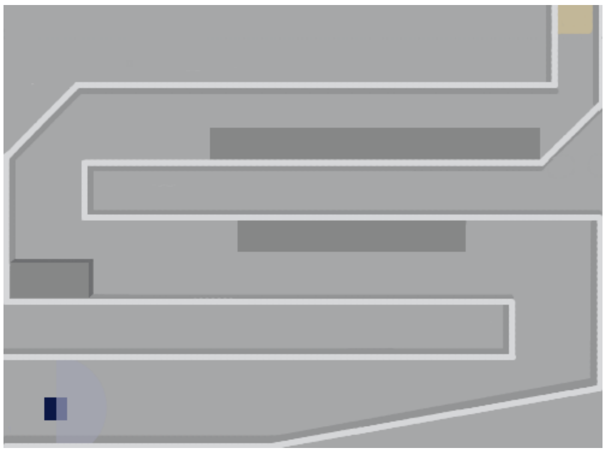

Figure 12. Maze 3: longer path, narrow road and challenging obstacles.

We use this environment to test three methods. The specific parameters of each network model are the same as those in the above experiments. Due to the longer distance, the reward settings are also slightly adjusted: 0.0002 reward for one safe step, -4 reward for a collision, -ep_step/800 for reaching the target area safely.

Experiment Details

The experiment has 3000 episodes in total, and each episode has 800 steps at the maximum. Figure 13 shows the episodes' cumulative rewards in the training period. The DDPG has not reached the target area throughout the process, and the rewards are around -4. We smoothed the original data for double DDPGs and MDAC to facilitate comparison, and darker lines are the smoothed data. The double DDPGs method only reaches the target area a few times around the 500th episode and has never reached the target area after the 600th episode. The MDAC reaches the target stably, starting from the 500th episode to the end of the training.

Next, Figure 14 shows the typical routes of three methods. The DDPG cannot complete the first turn and thus finds a path to maximize the steps to obtain more rewards. The double DDPGs method has reached the target area in some episodes, and we pick one route in this period. The route is relatively close to the one by MDAC, which is more stable during training. Both paths can accurately avoid obstacles and complete turns.

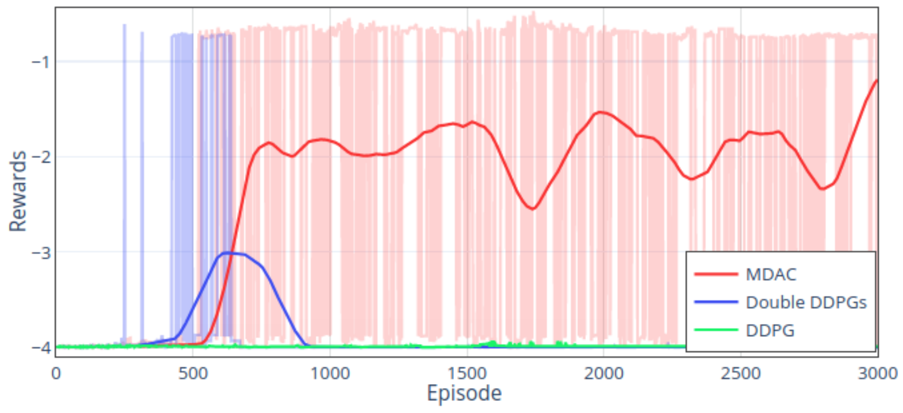

Figure 13. Training rewards of methods MDAC, double DDPGs and DDPG.

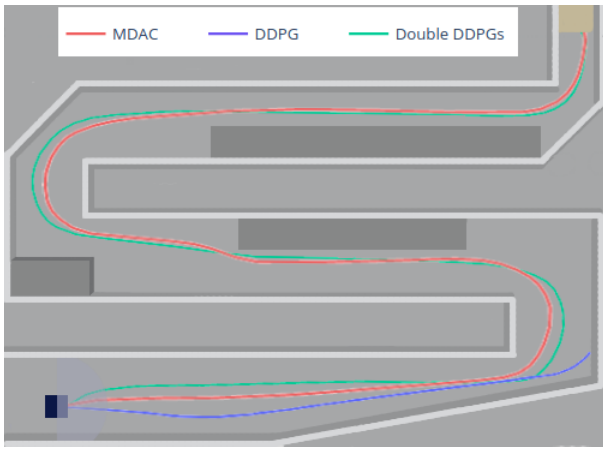

Figure 14. Routes of MDAC, double DDPGs and DDPG methods.

Figure 15 shows the steps used under each method in the whole process. Firstly, the DDPG is also trying to increase the number of steps to obtain more step rewards. However, overall, it is less than 200 steps except for some episodes between the 1500th and 2000th. The double DDPGs method reached the endpoint in the initial stage and took about 700 steps. Still, the performance of this method is unstable, resulting in inferior performance afterward, with each episode less than 100 steps. After the 500th episode, the MDAC method can stabilize the steps between 600 and 700 and behave more stable than double DDPGs. Even though it sometimes does not reach the target area, it collides more on the road around 400 steps. By comparison, the training of double DDPGs leads to the instability of the network, and the initially trained parameters cause worse performance in the subsequent training process. The MDAC can complete the task stably.

Finally, Figure 16 compares the speed adjustment strategies of the MDAC and double DDPGs. Both methods drive the robot at a lower speed due to the challenging environment. However, the speed of double DDPG behaves with uncertainty, causing the agent to be unable to optimize to a stable policy. The MDAC's speed behaves stably, starting with 40 units of speed and dropping to between 20 and 25 at the first turn. After that, it maintains a constant speed at around 26. In the subsequent turns, due to the narrow roads, the speed gradually decreased and finally stabilized at 20. Compared to double DDPGs, a significant difference is that the speed adjustment is relatively stable. The high-level module controls whether to enable the speed adjusting sub-module based on state. Thus, it behaves stably and can reach the destination stably in the later period.

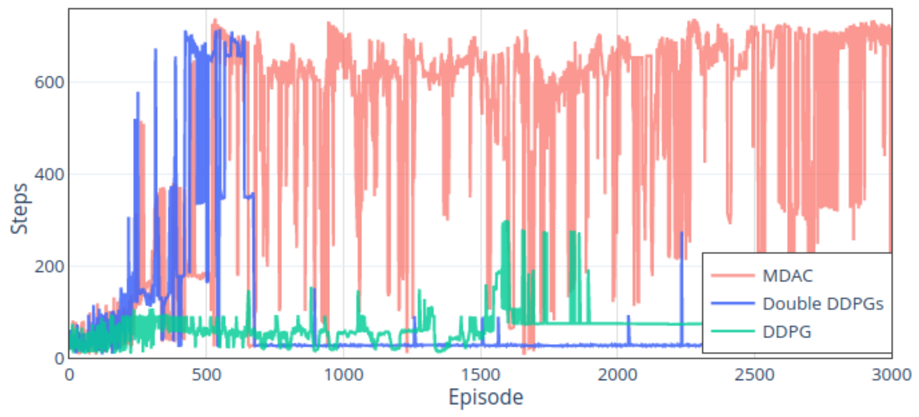

Figure 15. Episode's steps of MDAC, double DDPGs and DDPG methods.

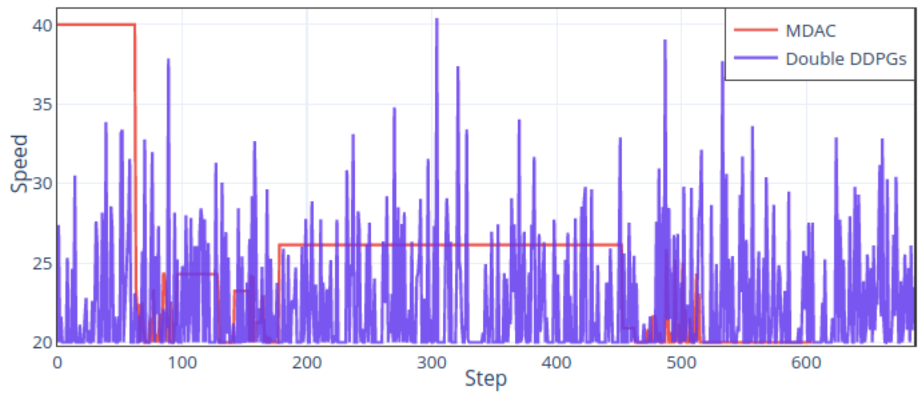

Figure 16. The speeds using MDAC and double DDPGs.

6. Discussion

The control problem of continuous space is a hot research topic in the RL domain. In terms of collision avoidance strategies, compared with traditional control methods, RL methods have unique advantages. It automatically trains to find the best control strategy and trains the model more directly and conveniently. With the RL training mode, the robot can learn a control policy and complete a defined task exploratorily. The whole training process agent acquires the control ability from trial and error using the offline transitions in memory.

In this work, we try to train the unmanned robot to have obstacle avoidance ability in a complex environment. In the experiment setting, the speed and heading direction adjustments are both considered. Many research works often ignore the speed change and assume that the robot runs constantly. A robot can better deal with complex scenes when facing obstacles with an adjustable speed. It can shorten the turning radius by decelerating and avoiding collision smartly.

If the robot has multiple action spaces in robotics applications, then a control policy should consider the mutual influence between actions. As the experiment in the paper, the changes of speed forward direction will simultaneously change the robot's state. In this case, a good collaboration will allow the robot to combine both actions to complete more complex tasks. Conversely, treating them as independent actions will decrease their ability to complete tasks.

Our approach uses a hierarchical frame to expand the number of action control spaces so that the robot can gain more capabilities. Furthermore, the MDAC method has high and low levels strategies, and the high-level one is used for planning and execution. Combined with low-level policies, it can solve the multi-dimensional actions' control tasks. The MDAC integrates the actor–critic approach so that it can control multiple continuous actions. In the experimental environment, the agent knows nothing about the environment at the beginning. The agent knows environment and state information through sensors. It can be seen from the experimental part that different strategies show different convergence conditions during the training process, and the stability after convergence is also different. In practical applications, excellent algorithms will make it easier for the network to converge to the global optimal state, which will show earlier convergence and stability after convergence.

The MDAC extends the scene of RL. It can be potentially applied in the field of robot control. Like most RL approaches, the method does not require prior knowledge of the environmental information and expert knowledge, and it is an end-to-end method. Our work aims to solve the obstacle avoidance tasks of the multi-dimensional-action robot. The trained system can output reasonable driving direction and speed based on the distance sensor, including speed, positioning, and driving direction information. When the network converges to the optimal state and remains relatively stable, the network model of the checkpoint in the stage can be directly applied to the same environment and maintain the best performance. For obstacle avoidance policies, if the map has slight changes, the model can be used since most states (feature) have been exposed during the training process. For maps with significant changes in environmental characteristics, transfer learning technology is needed to use a pre-trained network. This technique allows the model to better adapt to the new environment without long-time training from the beginning. In addition, the network can be used by local planners in path planning tasks. The method considers the kinematics of the robot, the planned trajectory is more in line with the actual situation, and the speed factor is considered.

Regarding the proposed approach, there are some aspects to improve in future work. Firstly, the reward assigning rules used by the high-level and low-level networks are the same. If their reward rules are different and individual, and each rule is designed according to its task, it will better guide the update of each network to reach the optimal condition. Of course, this requires a better simulator's design to enrich the source of rewards and state information; Secondly, the exploration strategy of RL generally uses one manner: an initialize exploration value gradually decreases to the minimum value as the training episodes increase. It can solve most tasks and reasonably balance exploration and exploitation issues. However, in the training of a relatively large-scope environment, the agent perhaps has not explored enough environmental scenarios before the exploration reaches the minimum value. This will affect the overall convergence in the training process. How to intelligently and dynamically adjust the exploration value according to the actual situation is worth studying; lastly, reward sparsity is the issue. In our experiment, the robot has enough reward information about obstacle avoidance, that is, the penalty of a collision and the minor benefit of each safe step. However, It lacks sufficient rewards for the target area's guidance. The agent will have a higher target reward in the memory after the robot has reached the target area. Increasing the target reward for navigating and reasonably guiding the robot to track the destination while avoiding obstacles is worth studying, and it has practical meaning.

7. Conclusions

In this paper, we presented a hierarchical RL-based method for solving obstacle avoidance tasks, and we designed experimental environments to test the performance. The method's structure is based on a hierarchical structure, and the design of each module network is similar to the actor–critic based methods. Some techniques are used in our method such as target network, soft update, greedy policy, etc. These techniques can make the training effect more stable and increase the exploration degree of the agent. We designed three experiments to compare this method with other classic DDPG-based methods. The advantage of this method is to solve more complex obstacle avoidance problems, consider speed as another action, and maintain good stability during training. Convergence is also relatively fast. In addition, since the method has a common RL interface, other techniques can also be integrated such as priority experience replay, multi-agent system, etc. This method has some room for improvement. For example, all modules in the method use the same source of reward. If there is corresponding reward information for each module, the networks can be better trained with more informative rewards. Of course, it depends on the simulation environment, and how to set them appropriately is worth studying further. In addition, the exploration strategy in the method is relatively standard. For each module, adopting a more targeted exploration strategy can also improve its stability. Finally, a single target-area reward is used as guidance information in the experiment. The robot lacks navigation information, and the reward is relatively sparse. Letting the robot track the target area while completing the essential function (obstacle avoidance) is worth studying. This method allows multiple actions to cooperate in obtaining rewards through the allocation of the high-level module. It can potentially be applied to robot obstacle avoidance tasks in multi-action spaces. In addition, it can be used as a local planner in path planning problems. The planned path first conforms to the kinematics of the robot, so it is reasonable. The route also has speed information, so it is more informative than a typical path planning algorithm.

Author Contributions: Conceptualization, M.W.; Funding acquisition, Y.G.; Investigation, M.W., Y.G., P.W., F.Z. and Z.L.; Methodology, M.W. and P.W.; Project administration, Y.G.; Software, M.W. and Z.L.; Validation, M.W. and F.Z.; Writing—original draft, M.W.; Writing—review and editing, Y.G., P.W., F.Z. and Z.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Khatib, O. Real-time obstacle avoidance for manipulators and mobile robots. In *Autonomous Robot Vehicles*; Springer: New York, NY, USA, 1986; pp. 396–404.
2. Stulp, F.; Schaal, S. Hierarchical reinforcement learning with movement primitives. In Proceedings of the 2011 11th IEEE-RAS International Conference on Humanoid Robots, Bled, Slovenia, 26–28 October 2011; pp. 231–238.
3. Sutton, R.S.; Barto, A.G. *[Draft-2] Reinforcement Learning: An Introduction*; The MIT Press: Cambridge, MA, USA; London, UK, 2013.
4. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv* **2013**, arXiv:1312.5602.
5. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.A.; Veness, J.; Bellemare, M.G.; Graves, A.; Riedmiller, M.; Fidjeland, A.K.; Ostrovski, G.; et al. Human-level control through deep reinforcement learning. *Nature* **2015**, *518*, 529. [CrossRef] [PubMed]
6. Gu, S.; Holly, E.; Lillicrap, T.; Levine, S. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In Proceedings of the 2017 IEEE international conference on robotics and automation (ICRA), Singapore, 29 May–3 June 2017; pp. 3389–3396.
7. Wu, M.; Gao, Y.; Jung, A.; Zhang, Q.; Du, S. The Actor-Dueling-Critic Method for Reinforcement Learning. *Sensors* **2019**, *19*, 1547. [CrossRef]
8. Wang, Z.; Schaul, T.; Hessel, M.; Hasselt, H.; Lanctot, M.; Freitas, N. Dueling network architectures for deep reinforcement learning. *arXiv* **2015**, arXiv:1511.06581
9. Hasselt, H.V. Double Q-learning. *Adv. Neural Inf. Process. Syst.* **2010**, *23*, 2613–2621.
10. Haarnoja, T.; Zhou, A.; Hartikainen, K.; Tucker, G.; Ha, S.; Tan, J.; Kumar, V.; Zhu, H.; Gupta, A.; Abbeel, P.; et al. Soft actor–critic algorithms and applications. *arXiv* **2018**, arXiv:1812.05905.
11. Barto, A.G.; Mahadevan, S. Recent advances in hierarchical reinforcement learning. *Discret. Event Dyn. Syst.* **2003**, *13*, 41–77. [CrossRef]
12. Sutton, R.S.; McAllester, D.A.; Singh, S.P.; Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. *Adv. Neural Inf. Process. Syst.* **1999**, *12*, 1057–1063.
13. Bhatnagar, S.; Sutton, R.; Ghavamzadeh, M.; Lee, M. Natural actor–critic algorithms. *Automatica* **2009**, *45*, 2471–2482. [CrossRef]
14. O'Donoghue, B.; Munos, R.; Kavukcuoglu, K.; Mnih, V. Combining policy gradient and Q-learning. *arXiv* **2016**, arXiv:1611.01626.
15. Al-Emran, M. Hierarchical reinforcement learning: A survey. *Int. J. Comput. Digit. Syst.* **2015**, *4*. [CrossRef]
16. Marthi, B.; Russell, S.J.; Latham, D.; Guestrin, C. Concurrent hierarchical reinforcement learning. In Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30–August 5, 2005; pp. 779–785.

17. Bakker, B.; Schmidhuber, J. Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. In Proceedings of the 8-th Conference on Intelligent Autonomous Systems, Amsterdam, The Netherlands, 24 August 2010; pp. 438–445
18. Tavakoli, A.; Pardo, F.; Kormushev, P. Action branching architectures for deep reinforcement learning. In Proceedings of the AAAI Conference on Artificial Intelligence, Hilton New Orleans Riverside, New Orleans, LA, USA, 2–7 Februray 2018; Volume 32
19. Tampuu, A.; Matiisen, T.; Kodelja, D.; Kuzovkin, I.; Korjus, K.; Aru, J.; Aru, J.; Vicente, R. Multiagent cooperation and competition with deep reinforcement learning. *PLoS ONE* **2017**, *12*, e0172395. [CrossRef] [PubMed]
20. Metz, L.; Ibarz, J.; Jaitly, N.; Davidson, J. Discrete sequential prediction of continuous actions for deep rl. *arXiv* **2017**, arXiv:1705.05035.
21. Vezhnevets, A.S.; Osindero, S.; Schaul, T.; Heess, N.; Jaderberg, M.; Silver, D.; Kavukcuoglu, K. Feudal networks for hierarchical reinforcement learning. In Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia, 6–11 August 2017; Volume 70, pp. 3540–3549.
22. Dietterich, T.G. An Overview of MAXQ Hierarchical Reinforcement Learning. In *International Symposium on Abstraction*; Springer: Berlin/Heidelberg, Germany, 2000.
23. Morimoto, J.; Doya, K. Acquisition of stand-up behavior by a real robot using hierarchical reinforcement learning. *Robot. Auton. Syst.* **2001**, *36*, 37–51. [CrossRef]
24. Kulkarni, T.D.; Narasimhan, K.; Saeedi, A.; Tenenbaum, J. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *Adv. Neural Inf. Process. Syst.* **2016**, *29*, 3675–3683.
25. Dietterich, T.G. Hierarchical reinforcement learning with the MAXQ value function decomposition. *J. Artif. Intell. Res.* **2000**, *13*, 227–303. [CrossRef]
26. Yang, Z.; Merrick, K.; Jin, L.; Abbass, H.A. Hierarchical deep reinforcement learning for continuous action control. *IEEE Trans. Neural Netw. Learn. Syst.* **2018**, *29*, 5174–5184. [CrossRef] [PubMed]
27. Bacon, P.L.; Harb, J.; Precup, D. The option-critic architecture. In Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, San Francisco, CA, USA, 4–9 February 2017
28. Nachum, O.; Gu, S.; Lee, H.; Levine, S. Data-efficient hierarchical reinforcement learning. *Adv. Neural Inf. Process. Syst.* **2018**, *31*, 3303–3313
29. Haarnoja, T.; Zhou, A.; Abbeel, P.; Levine, S. Soft actor–critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv* **2018**, arXiv:1801.01290.
30. Van Hasselt, H.; Guez, A.; Silver, D. Deep reinforcement learning with double q-learning. *arXiv* **2015**, arXiv:1509.06461.
31. Tamar, A.; Wu, Y.; Thomas, G.; Levine, S.; Abbeel, P. Value iteration networks. *Adv. Neural Inf. Process. Syst.* **2016**, *29*, 2154–2162.
32. Schaul, T.; Quan, J.; Antonoglou, I.; Silver, D. Prioritized experience replay. *arXiv* **2015**, arXiv:1511.05952.
33. Lillicrap, T.P.; Hunt, J.J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; Wierstra, D. Continuous control with deep reinforcement learning. *arXiv* **2015**, arXiv:1509.02971.
34. Diuk, C.; Strehl, A.L.; Littman, M.L. A hierarchical approach to efficient reinforcement learning in deterministic domains. In Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems, Future University, Hakodate, Japan, 8–12 May 2006; Association for Computing Machinery: New York, NY, USA, 2006; pp. 313–319
35. Eysenbach, B.; Gupta, A.; Ibarz, J.; Levine, S. Diversity is all you need: Learning skills without a reward function. *arXiv* **2018**, arXiv:1802.06070.
36. Florensa, C.; Duan, Y.; Abbeel, P. Stochastic neural networks for hierarchical reinforcement learning. *arXiv* **2017**, arXiv:1704.03012
37. Heess, N.; Wayne, G.; Tassa, Y.; Lillicrap, T.; Riedmiller, M.; Silver, D. Learning and transfer of modulated locomotor controllers. *arXiv* **2016**, arXiv:1610.05182.