DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Multiple Action Prediction in Deep Reinforcement Learning

Sanjeev Kumar

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Multiple Action Prediction in Deep Reinforcement Learning

Tiefes bestärkendes Lernen mit multiplen Aktionen

Author:	Sanjeev Kumar
Supervisor:	Prof. Dr. Nassir Navab
Advisers:	Christian Rupprecht, Dr. Federico Tombari
Submission Date:	15.02.2017

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.02.2017 Sanjeev Kumar

Acknowledgments

It is my pleasure to express my gratitude to all the people who supported me in the successful completion of this Master's Thesis.

First of all, I would like to express my sincere thanks to Christian Rupprecht for his guidance throughout this work. He was always available to provide valuable inputs and support without which this work would not have been possible.

Secondly, I would like to thank Dr. Federico Tombari for his kind supervision and guidance to steer the work in the right direction.

I am also grateful to Prof. Dr. Nassir Navab for allowing me to undertake this Master's Thesis under the supervision of his chair.

Finally, I must express my gratitude to my family for their continuous moral and financial support throughout the duration of my studies.

Abstract

Deep reinforcement learning has been applied successfully to a lot of domains in past few years. This thesis is primarily focused on deep reinforcement learning algorithms for continuous action space problems. Continuous action space problems arise very commonly in robotics such as picking objects with a robotic arm, autonomous driving etc. One common challenge in continuous control problems is the prediction of stochastic actions for a given state. We address this problem by using a formulation which produces M actions at each state. This formulation is general enough to be used with various reinforcement learning algorithms. We demonstrate this by proposing a stochastic version of deterministic policy gradient algorithm (DPG) called multiple action policy gradients (MAPG) and evaluate it on Mujoco continuous control problems and TORCS driving task. Further, we compare the results of our approach with other stochastic policy gradient algorithms.

Contents

1 Introduction

1.1 Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning inspired by behaviorist psychology and is based on the idea that humans learn by interacting with the environment. It considers a software agent situated in an environment making a sequence of decisions such that some cumulative reward is maximized. At each time step of interaction with the environment, the agent takes an action, and it receives an observation and a reward. The changes in the environment after an action, are converted by an interpreter into observation and reward. The interpreter encodes the problem to be solved in such a way that positive reward indicates the action was in the direction of the solution. So, maximizing cumulative reward over multiple interactions with environment corresponds to a solution of the underlying problem. The environment and interpreter are treated as black boxes and collectively referred to as the environment. The interaction between agent, environment, and interpreter is illustrated in Figure 1.1.

The goal of a reinforcement learning algorithm is to maximize the agent's total reward by repeatedly interacting with the environment. In the beginning, the RL agent knows nothing about the dynamics of the environment, but eventually learns to solve the problem by repeated trial and error. Therefore, reinforcement learning is general enough to be applied in a wide variety of problems from manufacturing [**01_manu**], to robot control [**01_control**], to finance [**01_finance**]. A detailed mathematical formulation of reinforcement learning problem is presented in Chapter 2.

1.2 Deep Reinforcement Learning

Today, most machine learning methods pose the learning problem as function learning from given data. Deep learning has been very successful in learning complex functions given enough data. In deep learning, the target function is represented by stacking multiple non-linear layers (a deep neural network) and the network parameters are optimized by gradient descending a problem-specific loss function. The deep neural

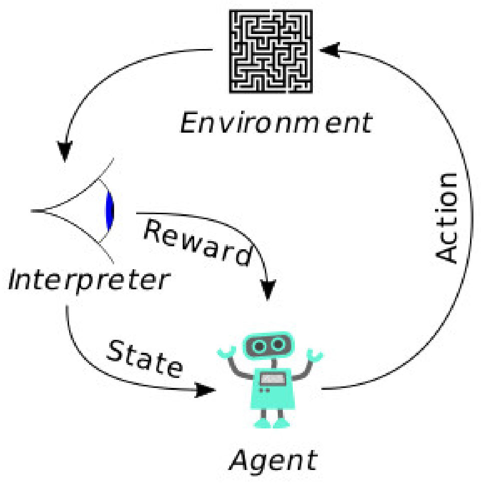


Figure 1.1: A typical reinforcement learning setup. The agent interacts with environment by performing actions, and receives reward and observation from interpreter as dictated by problem. Source: Wikipedia

networks have shown remarkable success in a variety of image recognition tasks [**01_imagenet**].

Deep reinforcement learning (DRL) is the study of reinforcement learning using deep neural networks as function approximators. Though neural networks have been used as function approximators in reinforcement learning for a long time, it recently got a lot of attention after the proposal of the deep Q-network algorithm (DQN) [**01_dqn**]. DQN learns to play a wide range of Atari games from raw pixels and outperforms humans on some tasks. Recently, a new algorithm Alpha Go [**01_alphago**] which also uses deep reinforcement learning, has learned to play Go better than human experts.

Deep neural networks can be directly optimized to learn a mapping from inputs to outputs in supervised learning. However, applying deep neural networks in reinforcement learning problems is not that straightforward as there are many different choices where a function approximator can be used. For example, in case of Atari games, a neural network can be used either to predict the action to take at a particular state (pixel values) or to approximate the future expected reward for a state-action pair. DQN uses the latter formulation and approximates Q-values for all state-action pairs. The neural network used by DQN is illustrated in Figure 1.2. The detailed classification of different deep reinforcement learning algorithms is given in Chapter 3.

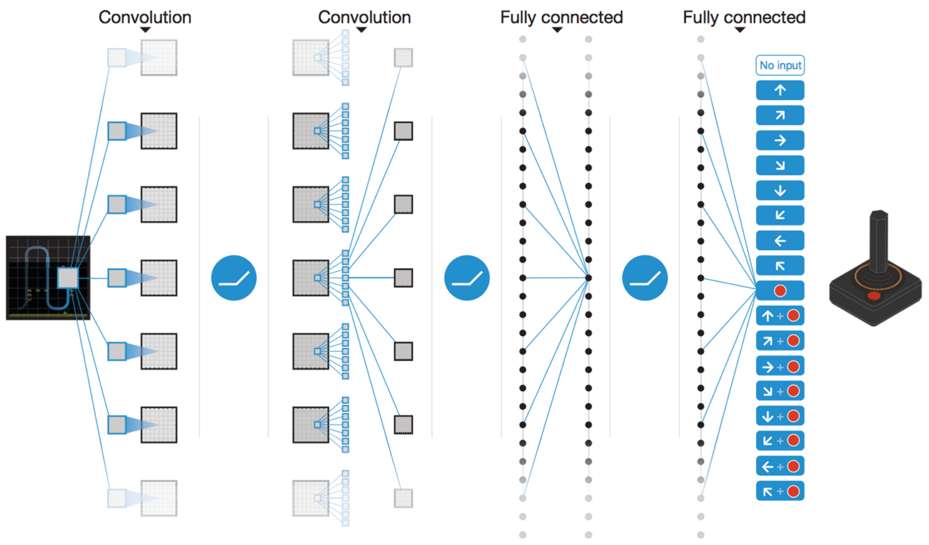


Figure 1.2: The input to the network consists of 4 consecutive frames and it outputs Q-values for all possible actions. The action with highest Q-value is then chosen at this state. Source: [**mnih2015humanlevel**]

.

1.3 Stochastic Continuous Control

Continuous control problems in reinforcement learning arise when the available actions of an agent are continuous valued. Continuous action spaces are usually more challenging than discrete action spaces [**ddpg**]. A naive approach to modifying discrete action space algorithms, such as DQN, to continuous action domain would be to discretize the action space. However, this approach has several drawbacks. If the discretization is coarse, the output actions will not be smooth and will lead to unstable behavior. And if the discretization is fine, the number of discretized actions will be very high and intractable to learn. This situation becomes even worse as the number of different actions increases, due to the curse of dimensionality.

A *policy* determines an RL agent's action selection process at a state. The policy can be either stochastic $\pi(a|s)$, the probability of an action a given a state s or deterministic $\pi(s)$, a deterministic mapping from state s to action a. The stochastic policies are better than deterministic policies for many reasons. The randomness embedded in the stochastic policy lead to a better exploration of the state space which is important to learn a good policy. A stochastic policy is also required to calculate the policy gradients which are needed when policies are represented by neural networks.

1.4 Contributions

In this thesis, we consider continuous action space problems in deep reinforcement learning. We address the shortcomings of existing algorithms for stochastic continuous control and purpose a new formulation for such problems. Our proposed approach to making a policy stochastic is called Multiple Action Policy Gradients (MAPG). MAPG produces M point estimates from the policy network at a state and then samples one action uniformly. The uniform random sampling from M actions renders the resulting policy stochastic.

The proposed formulation of the policy has several interesting implications. First, one can study the decision process of RL agent at runtime. A low variance among M actions implies that the model only sees one way to act in a certain situation. A wider or even multi-modal distribution suggests that either there exist several possibilities given the current state or the model is unsure which action is the best. Second, the resulting stochastic policy by MAPG leads to a better exploration of the entire solution space and the final solution is of higher quality. Finally, MAPG can eliminate the external exploration mechanisms required during training e.g. Ornstein-Uhlenbeck process noise [**ounoise**], parameter noise [**paramnoise**] or differential entropy of the normal distribution.

While MAPG helps in optimizing continuous policies, it has few drawbacks. It increases the overall training time of the algorithm. The value of M is an hyper-parameter which needs to be tuned separately for different environments.

In experiments, we evaluate MAPG on DDPG [**ddpg**] algorithm on six continuous control problems of the OpenAI Gym [**greg2016**] as well as a *deep driving* scenario using the TORCS car simulator [**TORCS**]. Then, we compare the performance of MAPG with DDPG [**ddpg**], A3C [**a3c**] and SVG(0) [**svg**] on all tasks. Further, we study the effects of MAPG on exploration during training and the quality of policy by analyzing the variance among different action values.

2 Background

In this chapter, we formalize the reinforcement learning problem and discuss briefly various class of reinforcement learning algorithms.

2.1 Markov Decision Process

Markov Decision Process (MDP) provides a mathematical framework to formalize sequential decision making. MDP describes an agent interacting with a stochastic environment. The decision maker and learner are called the agent, everything else is referred to as the environment. The interaction between agent and environment happen continuously in discrete time steps. At each time step t, the agent selects action and environment respond to the action by presenting a new situation for the agent.

An MDP has following components:

- $\mathcal{S}$ is a finite set of environment states.
- $\mathcal{A}$ is a finite set of actions available for agent at a state.
- $p(s_{t+1}|s_t, a_t)$ is the transition probability distribution. It represents the probability that an action a_t in state s_t at time t will lead to state s_{t+1} at time $t+1$.
- $r(s_t, a_t, s_{t+1}) \in \mathbb{R}$ is the reward received by the agent after taking an action a_t at state s_t and reaching at state s_{t+1}.
- $\gamma \in [0, 1]$ is the discount factor, which determines present value of future rewards.

Sometimes the initial state distribution $\mu(s)$ is also provided, which is used to sample the initial state s_0.

The core problem of MDP is to find a *policy* for the agent, a function $\pi(s)$ which maps a state s to action chosen by the agent. The policy π should be such that it maximizes the cumulative reward received by the agent over multiple interactions with

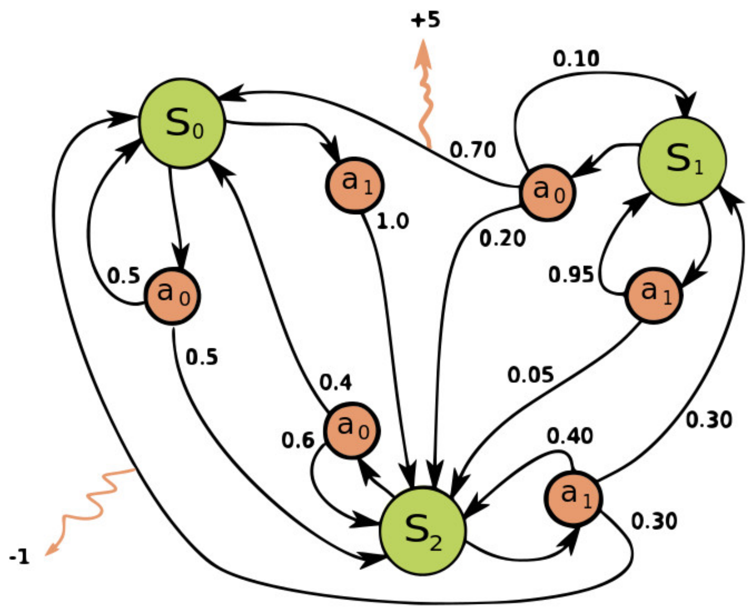


Figure 2.1: An example of MDP with three states and two actions. The orange arrows represent rewards emitted by environment to agent. The labels on edges are transition probabilities. Source: Wikipedia.

the environment. In stochastic policies, π is a conditional distribution $\pi(a|s)$ from which actions are sampled. The state transitions of an MDP satisfy the *Markov property*, i.e. the transition probabilities are independent of previous states and only depend on current state s_t. A simple MDP with three states and two actions is illustrated in Figure 2.1.

2.2 Reinforcement Learning Problem

Reinforcement learning can be studied mathematically as an MDP. An RL problem can be episodic or continuous depending on the task at hand. In episodic RL problems, the agent reaches a terminal state after a finite number of steps; one such finite interaction is termed as an *episode*. Whereas in continuous RL problems, the agent interacts continuously with the environment and there is no terminal state. The notion of cumulative reward breaks in case of continuous RL problem as it can have infinite steps. So, instead of cumulative reward, discounted return is maximized and is defined at time step t with discount factor γ as

$$R_t = \sum_{i=t}^{\infty} \gamma^{i-t} r(s_i, a_i, s_{i+1}). \tag{2.1}$$

The states s_i are generated by sampling actions from some policy $\pi(a_i|s_i)$. The discount factor γ is needed to weight the importance of future rewards. For $\gamma = 0$, the agent is only concerned with maximizing immediate reward, as γ increases the agent becomes far-sighted and acts to maximize future rewards as well. Usually, $\gamma \in [0.9, 0.99]$ is most commonly used. Discount return is also valid for episodic RL problems as those can be reduced to continuous RL problems by adding a self-loop at terminal states with transition probability 1.

The discounted reward can be recursively written as

$$R_t = r(s_t, a_t, s_{t+1}) + \gamma R_{t+1}. \tag{2.2}$$

Thus, the goal of reinforcement learning algorithms is to find a policy that maximizes the expected future reward.

2.3 Value Functions

A value function represents how good it is to be in a particular state. Value functions are important as they provide a framework to optimize agent policies and a lot of reinforcement learning algorithms use value functions in some form.

2.3.1 Definition

Value function is formally defined in terms of expected future rewards at a state. Since future rewards depend on the policy of the agent, therefore the value functions are defined with respect to policies. A value function can be dependent on state only or on state-action pair. A state dependent value function $V_\pi(s_t)$ is called state-value function and is defined as expected future reward starting from s_t and then following the policy $\pi(a_t|s_t)$. It can be formally written as

$$V_\pi(s_t) = \mathbb{E}_\pi[R_t|s_t] = \mathbb{E}_\pi\left[\sum_{i=t}^{\infty}\gamma^{i-t}r(s_i,a_i,s_{i+1})|s_t\right], \tag{2.3}$$

where $\mathbb{E}_\pi[.]$ denotes the expected value of a random variable given that the agent follows policy π, and t is any time step.

The state-action value function is called Q-function and is denoted by $Q_\pi(s_t, a_t)$. The Q-function is defined as the expected return starting from s, taking an action a and then following the policy $\pi(a_t|s_t)$. It can be formally written as

$$Q_\pi(s_t,a_t) = \mathbb{E}_\pi[R_t|s_t,a_t] = \mathbb{E}_\pi\left[\sum_{i=t}^{\infty}\gamma^{i-t}r(s_i,a_i,s_{i+1})|s_t,a_t\right]. \tag{2.4}$$

The value function satisfies a recursive relation which follows from Eqn. 2.2, 2.3 and 2.4, called Bellman equation. For any policy π, the following always hold for value functions

$$V_\pi(s_t) = \mathbb{E}_{s_{t+1}\sim p, a_t\sim\pi}[r(s_t,a_t,s_{t+1}) + \gamma V_\pi(s_{t+1})], \tag{2.5}$$

$$Q_\pi(s_t,a_t) = \mathbb{E}_{s_{t+1}\sim p}[r(s_t,a_t,s_{t+1}) + \gamma\mathbb{E}_{a_{t+1}\sim\pi}[Q_\pi(s_{t+1},a_{t+1})]]. \tag{2.6}$$

2.3.2 Optimal Value Functions

A policy π^1 is better than π^2 if $V_{\pi^1}(s) \geq V_{\pi^2}(s)$ for all possible states s. There exist policies which are always better than all other policies, called optimal policies. The optimal policy π^* for a given reinforcement learning task can be found by finding the optimal value function:

$$V^*(s) = \max_\pi V_\pi(s), \tag{2.7}$$

$$Q^*(s,a) = \max_\pi Q_\pi(s,a). \tag{2.8}$$

Optimal value function gives the best possible total reward. Since Q^* represents the optimal value function, the best value at step $t+1$ is just the action which has highest Q-value. So, the Bellman equation (see Eqn. 2.4) for the optimal Q-value function Q^* can be written without referencing any other policy π. The optimal policy π^* for

Q^* is therefore the greedy policy $\max_{a \in A} Q^*_{\pi^*}(s,a)$. The Bellman equation for optimal Q-value, known as Bellman optimality condition can be written as

$$Q^*_{\pi}(s_t, a_t) = \mathbb{E}_{s_{t+1} \sim p}[r(s_t, a_t, s_{t+1}) + \gamma \max_{a' \in A} Q^*_{\pi^*}(s_{t+1}, a')]. \tag{2.9}$$

A similar optimality condition holds for state-value function V^* as well.

2.4 Policy Optimization

As discussed in the last section, the goal of a reinforcement learning algorithm is to find a policy that maximizes the expected future rewards. There are various approaches to find such a policy and could be broadly classified in following two types:

2.4.1 Value Iteration

Value iteration algorithms search for an optimal value function. The value function is initialized randomly in the beginning and then improved iteratively until optimal value function is found. The value function is improved at each iteration using the Bellman equation (see Eqn.2.6). Once optimal value function has been found, the optimal policy can be extracted from value function by choosing a greedy actions at each state.

2.4.2 Policy Iteration

Policy iteration algorithms find the optimal policy by alternating between policy evaluation and policy improvement. The agent starts with a random policy, finds the value function for the policy (policy evaluation), then improves the policy based on previous value function. The value function is found using Eqn. 2.4. In many problems, policy iteration converges faster than value iteration because policy function is easier to represent than the value function. For example, in case of Atari games, it is easier to represent a policy which predicts the next action at an image frame rather than representing a function which predicts the expected reward for all actions at that frame.

2.5 Function Approximation

In real-world problems, finding an optimal policy is computationally impossible as the problem state space increases exponentially with increase in state dimensionality. In such cases, only an approximation to the optimal policy can be found. There are many choices for function approximators such as decision trees, nearest neighbor, neural networks. Out of these, neural networks had a lot of success recently in a variety of reinforcement learning problems and are commonly used nowadays.

Depending on which part of the problem is modeled with neural networks, there exist a variety of algorithms. The different categories of model-free algorithms are listed below.

- **Value Based Methods** learn a value function and have an implicit greedy policy. The value function is represented by a neural network as $Q(s,a|\theta^Q)$, where θ^Q are network parameters. The value function network can either take state s as input and output Q-values for all (s,a) pairs or take (s,a) as input and output a single Q-value. DQN [**01_dqn**] is a value based algorithm which learns Q-values for all (s,a) pairs.

- **Policy Based Methods** learn the policy directly and no value function is used. The policy is represented by a neural network as $\pi(a|s,\theta^{\pi})$, where θ^{π} are network parameters. Alpha Go [**Silver_2016**] uses a policy based method which trains a policy network with Monte-Carlo tree search.

- **Actor-Critic Methods** are a mix of both value-based and policy-based algorithms. These have two networks, an actor network which represents the policy and a critic network which represents the value function. Sometimes actor and critic share some part of the network. The critic network is used to estimate the policy gradients for optimizing the policy network. Actor-critic methods are more common in continuous action space problems. A3C [**a3c**] uses an actor-critic architecture, where multiple actor-critic networks are used for asynchronous training.

2.6 Model-based and Model-free Reinforcement Learning

Model based reinforcement learning algorithms build a model of the environment dynamics. The model contains the knowledge about the task at hand, such as transition probabilities and different possible immediate outcomes. These algorithms usually roll-out the entire trajectory by taking actions using the environment model and estimate the total reward for each action. The action with the highest total reward is then chosen. [**Nagabandi2017NeuralND**], SVG [**svg**], NAF [**gu2016**] are some recently proposed model-based algorithms which use neural networks to the model environment dynamics.

Model-free reinforcement learning algorithms, on the other hand, are based on estimating the expected returns at any state by learning a value function or learning a policy which maximizes expected returns. Model-free algorithms are easier to use but require a lot of trial and error to get a good estimate of the value function or the policy.

DQN [**01_dqn**], DDPG [**ddpg**], A3C [**a3c**] and TRPO [**schulman2015trust**] are some recent model-free algorithms which have shown to perform well on variety of Atari games and robotic simulation tasks.

2.7 On-policy and off-policy Algorithms

A reinforcement learning algorithm can have two policies during training. One is used to generate the experience and explore the environment, while the second is the target optimal policy which is being learned by the agent. Depending on whether these policies are same or different it gives rise to two different class of algorithms.

On-policy algorithms optimize the same policy which is used by the agent to explore the environment. SARSA [**sutton1998reinforcement**] is an on-policy algorithm.

Off-policy algorithms use a different policy to explore the environment and optimize a different policy based on the generated experience. DQN is an off-policy learning algorithm.

2.8 Training Techniques

Using reinforcement learning directly with neural networks is often unstable and sample inefficient. The agent has to play millions of episodes to learn a policy and it is not guaranteed that the agent will learn a useful policy even after that. So, there are some common techniques to alleviate these problems. These techniques are described below.

2.8.1 Experience Replay

Experience replay [**Lin1992**] is a technique in which the learning algorithm proceeds in two phases. First, it gathers experience and store transitions (s_t, a_t, s_{t+1}, r_t) at each time step t in a replay memory. In the second step, the algorithm randomly samples batches from the replay memory and applies the Bellman optimality equation (see Eqn. 2.8) to update the target Q-network. Experience replay has following advantages:

- It efficiently uses the agent's past experience by learning through it multiple times. This is important because generating experience in real-world is costly. Value function learning with neural networks does not converge quickly, so it is better to use multiple passes of same experience to train the network.

- Most gradient-based algorithms assume data to be independent identically distributed (i.i.d.) which is achieved by randomly sampling from replay buffer. This leads to better convergence behavior.

- Neural networks does not handle training with sequential data very well, which leads to *catastrophic forgetting* [**French1999CatastrophicFI**]. Catastrophic forgetting is common in RL problems because when new data is introduced (from experience with the environment), new updates to the network overwrite the previously acquired behaviour. Sampling old experience from the replay memory reduces this problem.

2.8.2 Soft Target Updates

Training neural networks can be unstable with reinforcement learning algorithms on many environments. This could be because the same network that is being optimized is also used for calculation of the target term in the loss function. This often leads to divergent behaviour [**ddpg**] [**01_dqn**].

The problem is solved by maintaining a target network in addition to a local network (the network being optimized directly). The target network is used for calculation of the target term in the loss function and the local network parameters are updated using this loss. In the beginning of training, the target network starts with the same parameters as the local network. During training, the parameters of the target network (θ') are then slowly updated from the local network parameters (θ). The update equation for target network parameters is given by

$$\theta' = (1 - \tau)\theta' + \tau\theta, \tag{2.10}$$

where τ is usually $\ll 1$.

3 Related Work

In this chapter, we discuss related work on continuous action space reinforcement learning problems which use neural networks as function approximators.

3.1 Continuous Q-learning

Q-learning has been very successful in discrete action space problems. In Q-learning, the optimal policy is found by learning an action-value function $Q(s, a)$. At each time step t, the algorithm interacts with environment by taking action a_t at states s_t observing next state s_{t+1} and reward $r(s_t, a_t)$. The Q-values for state-action pair (s_t, a_t) are then updated by the following equation

$$Q(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha[r(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a_r)], \tag{3.1}$$

where $\alpha \in (0, 1]$ is the learning rate of the algorithm.

The maximization $\max_a Q(s_{t+1}, a_t)$ is easy to calculate when actions are discrete as there are finite Q-values at each state. However, it becomes intractable when actions are continuous.

3.1.1 Normalized Advantage Function

Normalized advantage function (NAF) **[gu2016]** addresses the problem of estimating $\max_a Q(s_{t+1}, a_t)$ for continuous actions. It represents the Q-function such that the maximization can be analytically calculated from the state-value function $V(s_t)$ and advantage term $A(s_t, a_t)$. The state-value function and advantage are estimated by a neural network. The Q-function is calculated as

$$Q(s_t, a_t) = A(s_t, a_t|\theta^A) + V(s_t|\theta^V), \tag{3.2}$$

$$A(s_t, a_t|\theta^A) = -\frac{1}{2}(a_t - \mu(s_t|\theta_\mu))^T P(s_t|\theta^P)(a_t - \mu(s_t|\theta_\mu)), \tag{3.3}$$

where μ is the deterministic policy used to generate the experience. $P(s_t|\theta^P)$ and $V(s_t|\theta^V)$ are estimated by the neural network. Eqn. 3.3 is quadratic in μ, so μ maximizes the Q-function.

3.2 Policy Gradient Methods

Policy gradient algorithms directly optimize the policy parameters by estimating the gradient of the policy's performance with respect to the policy parameters. The policy is represented as a parametrized probability distribution $\pi[a|s,\theta]$, where θ is the policy parameter vector, represented by a neural network. The policy performance is usually measured by the expected value of future rewards and therefore the goal of a policy gradient algorithm is the following optimization problem

$$\max_{\theta} \mathbb{E}[R|\pi_\theta]. \tag{3.4}$$

This optimization problem can be solved by gradient ascent on the gradient of $\mathbb{E}[R|\pi_\theta]$ with respect to the policy parameters θ. To find the policy gradient, we use the following equality which finds the gradient of expectation $\mathbb{E}_{x\sim p(x|\theta)}[f(x)]$ with respect to distribution parameters θ,

$$\nabla_\theta \mathbb{E}_x[f(x)] = \mathbb{E}_x[\nabla_\theta \log p(x|\theta) f(x)], \tag{3.5}$$

where $f(x)$ is a scalar valued function and x is sampled from the probability distribution $p(x|\theta)$. The proof of this identity is given in **[schulman_policy_opt]**. We can approximate the gradient of expectation by sampling N values $x_1, x_2, \ldots, x_N$ form $p(x|\theta)$ and taking the average

$$\nabla_\theta \mathbb{E}_x[f(x)] \approx \frac{1}{N} \sum_{n=1}^{N} \nabla_\theta \log p(x_i|\theta) f(x_i). \tag{3.6}$$

Now, to apply the above idea to optimize Eqn. 3.4, the policy should be *stochastic*. Let us consider a trajectory τ generated by the policy $\pi(a|s,\theta)$.

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T).$$

Let probability distribution of the trajectory be $p(\tau|\theta)$, parameterized by θ. $p(\tau|\theta)$ can be expanded in terms of $\pi(a|s,\theta)$, transition probability distribution $p(s_{t+1}|s_t, a_t)$ and the initial state distribution $\mu(s)$ as

$$p(\tau|\theta) = \mu(s_0)\pi(a_0|s_0,\theta)p(s_1|s_0,a_0)$$
$$\pi(a_1|s_1,\theta)\dots\pi(a_{T-1}|s_{T-1},\theta)p(s_T|s_{T-1},a_{T-1}). \tag{3.7}$$

Taking logarithm of Eqn. 3.7 and differentiating with respect to θ, the transition probability terms $p(s_{t+1}|s_t,a_t)$ and initial state distribution $\mu(s_0)$ will become zero as these are independent of θ. Therefore, we get

$$\nabla_\theta \log p(\tau|\theta) = \sum_{t=0}^{T-1} \log \pi(a_t|s_t,\theta).$$

Using Eqn. 3.5 the policy gradient for the expected reward with respect to θ on experience τ becomes

$$\nabla_\theta \mathbb{E}_\tau[R(\tau)] = \mathbb{E}_\tau\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t,\theta)R(\tau)\right]. \tag{3.8}$$

Other policy performance criteria can be used in place of discounted reward $R(\tau)$. One common policy performance criterion is advantage estimate $A(s_t)$ which is given as

$$A(s_t) = \hat{R}(\tau) - V(s_t|\theta^V), \tag{3.9}$$

where $\hat{R}(\tau)$ is the discounted return for some experience τ. $V(s_t|\theta^V)$ is the state-value function at s_t which is represented by a critic network. The advantage estimate represents the increase in expected return by taking a particular action at state s_t. Here, the advantage signifies how good the actor network is at predicting an action at time step t with respect to the critic network.

There exist a variety of policy gradient algorithms which use Eqn. 3.8 to calculate policy gradients. We will discuss the most popular of these algorithms here.

3.2.1 Deterministic Policy Gradients

The deterministic policy gradients (DPG) [**silver2014deterministic**] is an off-policy, actor-critic algorithm which uses deterministic policies. In the last section, we proved that a stochastic policy is required to calculate the policy gradients. However, Silver et al. proved that the policy gradients do exist for deterministic policies and are a limiting case of stochastic policy gradients.

Since the policy is deterministic, external process noise is added for exploration. In the original DPG paper, they use Ornstein-Uhlenbeck process noise [**ounoise**], but any other noise such as parameter noise [**paramnoise**] could be used instead. DPG uses experience replay and soft updates to stabilize the training.

3.2.2 Stochastic Value Gradients

Stochastic value gradients (SVG) proposed by [**svg**] provides a family of reinforcement learning algorithms ranging from model-based to model-free. SVG(0) is the model-free variant of stochastic value gradients. SVG(0) is very similar to DDPG and uses a stochastic policy by parameterizing the policy with a noise variable. At each training step, the agent collects the experience and uses it to update the policy and value network.

3.2.3 Trust Region Policy Optimization

Trust region policy optimization (TRPO) [**schulman2015trust**] makes vanilla policy gradient algorithms stable and sample efficient. Vanilla policy gradient algorithms use an estimator of the reward $R(\tau)$ (calculated using Eqn. 2.1 for T steps) obtained from the generated trajectory τ. Since, vanilla policy algorithms update policy from the sampled batch which is generated using the same policy, if some gradient update leads to a bad policy, the bad policy will then generate bad trajectories leading to further drop in the performance and it becomes hard for algorithm to recover.

TRPO addresses this problem by using a *surrogate objective* which is given as

$$L_{\pi_{old}}(\pi) = \frac{1}{N}\sum_{i=1}^{N}\frac{\pi(a_i|s_i)}{\pi_{old}(a_i|s_i)}\hat{A}_i, \tag{3.10}$$

where π_{old} is the old policy to which the N samples were generated, π is the current policy and $\hat{A}_i$ is the advantage estimate (same as $R(\tau)$) calculated on π_{old}. The policy gradient can be calculated by differentiating the surrogate objective loss. TRPO is an on-policy policy gradient algorithm.

3.2.4 Asynchronous Advantage Actor Critic

Asynchronous advantage actor-critic (A3C) [**a3c**] is an on-policy, actor-critic algorithm. The policy gradient in A3C is calculated using Eqn. 3.8 and Eqn. 3.9. A3C trains multiple actor-critic networks asynchronously by running multiple agent threads. Each agent thread uses its local copy of actor-critic network. The local network parameters are synchronized with a global network at the start of each iteration. Each agent accumulates local gradients and updates the global network at end of each training step.

The A3C agent uses entropy based exploration. A3C works well on many tasks because multiple actors make the updates independent and identically distributed which result in faster convergence of networks. A3C typically uses 4, 8 or 16 parallel threads.

3.3 Continuous Distribution Estimation

Continuous probability distributions when represented using neural networks predict either a point estimate or parameters of a Gaussian distribution. This representation is limited when a rich representation of probability distribution is required. Two approaches to deal with this problem are described below.

3.3.1 Mixture Density Networks

Mixture density networks (MDN) [**bishop_mdn**] predicts a mixture model as the output of the neural network. So, MDNs can represent any probability distribution completely. This is based on the observation that when estimating a continuous probability distribution using a neural network, only a point estimate can be produced which is the conditional average of target data. This point estimate provides a very limited description of the target variable.

3.3.2 Multiple Hypothesis Prediction

Multiple hypothesis prediction (MHP) [**rupprecht2017iccv**] proposes a framework to train and reformulate single prediction neural networks as multiple output networks. This formulation shows performance improvement in a variety of tasks such as image classification, pose estimation, semantic segmentation etc.

4 Multiple Action Policy Gradients

In this section, we describe the Multiple Action Policy Gradient algorithm and give a simple proof why MAPG converges to a better policy. Further, we propose the MAPG version of Deep Deterministic Policy Gradient (DDPG).

4.1 Multiple Action Prediction

Multiple action prediction idea is inspired by multiple hypothesis prediction [**rupprecht2017iccv**] for supervised learning. The basic idea behind multiple action prediction is to propose M actions at each time step for a reinforcement learning agent. The agent uniformly samples one action out of these M proposed actions. This formulation makes the policy stochastic and is useful in continuous reinforcement learning tasks. We call the class of policy gradient algorithms using this formulation as Multiple Action Policy Gradients (MAPG). Although, multiple action prediction is general enough and can be used with any existing reinforcement learning algorithm, in this work we only consider policy gradient methods with multiple action prediction.

MAPG is based on the intuition that the resulting policy is stochastic which leads to a better exploration of the entire solution space as well as a final solution of potentially higher quality. It can also help in eliminating the external exploration mechanisms required during training e.g. Ornstein-Uhlenbeck process noise [**ounoise**], parameter noise [**paramnoise**] or differential entropy of the normal distribution.

A simple example of inverted pendulum cart balancing task where MAPG is useful is illustrated in Figure 4.1. In this task, the goal is to balance the pendulum upside down. The agent can control the movement by moving the cart to left or right. The pendulum is currently hanging vertically and has two equally promising actions to reach final goal: either moving left or right. A deterministic agent chooses a single action for every state, this breaks the inherent symmetry of the task. Other distribution parameter estimation methods (e.g. A3C, NAF) might work better than deterministic policies in this case as there are only two good options, but in cases when there are

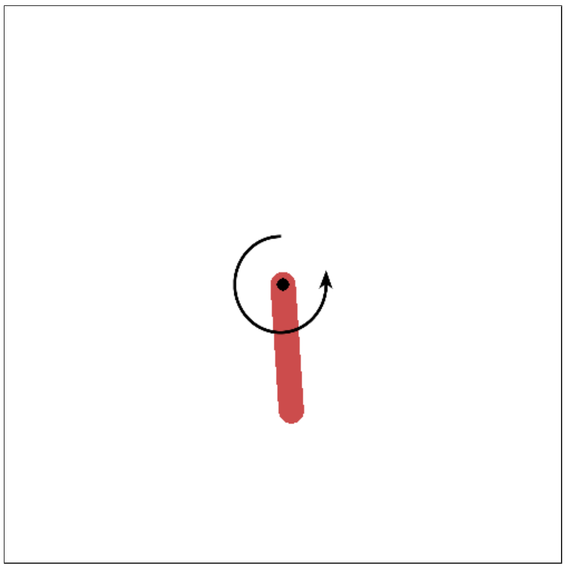


Figure 4.1: Screenshot of `Pendulum` environment. In this position the pendulum has two equally good actions, it can either move left or right to reach the target vertically upward position.

more than two good actions to select, this will not be optimal. MAPG allows the agent to suggest multiple actions, which enables it to resolve cases like this easily.

4.2 Optimal Policy for MAPG

MAPG produces multiple actions which can be viewed as M different policies. Here, we show that MAPG learns an optimal policy and all learned M policies are equally good. We investigate a typical reinforcement learning setup introduced in Chapter 2.

We consider the Bellman equation for Q-value function from Eqn. 2.6. Methods such as (D)DPG use a deterministic policy where each state is deterministically mapped to an action using a function $\pi : \mathcal{S} \rightarrow \mathcal{A}$ which simplifies Eqn. 2.6 to

$$Q_\pi(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_t) + \gamma Q_\pi(s_{t+1}, \pi(s_{t+1}))]. \tag{4.1}$$

The Q value of an action is approximated by a critic network which estimates $Q_\pi(s_t, a_t)$ for the action chosen by the actor network.

The key idea behind predicting multiple actions is that it is possible to learn a stochastic policy as long as the inner expectation remains tractable. Multiple action prediction achieves this by predicting a fixed number M of actions $\rho : \mathcal{S} \to \mathcal{A}^M$ and uniformly sampling from them. The expected value is then the mean over all M state-action pairs. The state-action value can then be defined as

$$\begin{aligned} Q_\rho(s_t, a_t) =& \mathbb{E}_{r_t, s_{t+1} \sim E} \\ & \left[r(s_t, a_t) + \gamma \frac{1}{M} \sum_{m=1}^{M} Q_\rho(s_{t+1}, \rho_m(s_{t+1})) \right]. \end{aligned} \tag{4.2}$$

This is beneficial since we not only enable the agent to employ a stochastic policy when necessary, but we also approximate the action distribution of the policy with multiple samples instead of one.

There exists an intuitive proof that the outer expectation in Eqn. 4.2 will be maximal if and only if the inner Q_ρ are all equal. The idea is based on the following argument: let us assume ρ as an optimal policy maximizing Eqn. 3.4. Further, one of the M actions $\rho_j(s_{t+1})$ for a state s_{t+1} has a lower expected return than another action k,

$$Q_\rho(s_{t+1}, \rho_j(s_{t+1})) < Q_\rho(s_{t+1}, \rho_k(s_{t+1})). \tag{4.3}$$

Then there exists a policy ρ^* that would score higher than ρ that is exactly the same as ρ except that it predicts action k instead of j: $\rho^*_j(s_{t+1}) := \rho_k(s_{t+1})$. However, this contradicts the assumption that we had learned an optimal policy beforehand. Thus in an optimal policy all M action proposals will have the same expected return. More informal, this can also be seen as a derivation from the training procedure. If we always select a random action from the M proposals, they should all be equally good since the actor cannot decide which action will be executed.

From the proof, it directly follows that it is possible - and sometimes necessary - that all proposed actions are identical. This is the case in situations where there is just one single right action to take. When the action proposals do not collapse into one, there are two possibilities: either it does not matter what action is currently performed, or all proposed actions lead to the desired outcome.

Naturally, the set of stochastic policies includes all deterministic policies, since a deterministic policy is a stochastic policy with a single action having probability density equal to one. This means that in theory, we expect the multiple action version of a deterministic algorithm to perform better or equally well since it could always learn a deterministic policy by predicting M identical actions for every state.

In the experimental section, we will investigate if the predicted value for each of the M actions is indeed similar during an episode.

4.3 DDPG with MAPG

Here we modify deep deterministic policy gradient (DDPG) algorithm to use MAPG. DDPG is based on actor-critic architecture, where actor network produces a deterministic policy $\pi(s)$ for a state s. To make DDPG use MAPG, we modify the last layer of actor network and output M action vectors. During training of the algorithm, one out of M action vectors is uniformly sampled and index j of the chosen action vector is saved along with transition $(s_t, a_t^i, s_{t+1}, r_t)$. Now, during optimization of the actor network, the policy gradient is only propagated through the branch connected by j at the last layer. Over time each branch in the last layer will be selected equally often, thus every branch will be updated and learned during training.

Algorithm 1 outlines the MAPG algorithm in detail.

Algorithm 1 DDPG with MAPG algorithm

```
Modify actor network $\pi(s|\theta_\pi)$ to output $M$ actions, $A_t = \{\rho_1(s_t), \dots, \rho_M(s_t)\}$.
Randomly initialize actor $\pi(s|\theta_\pi)$ and critic $Q(s, a|\theta_Q)$ network weights.
Initialize target actor $\pi'$ and critic $Q'$ networks, $\theta'_\pi \leftarrow \theta_\pi$ and $\theta'_Q \leftarrow \theta_Q$.
**for** episode $= 1$ **to** $N$ **do**
  Initialize random process $\mathcal{N}$ for exploration.
  Receive initial observation/state $s_1$.
  **for** $t = 1$ **to** $T$ **do**
    Predict $M$ action proposals $A_t = \{\rho_1(s_t), \dots, \rho_M(s_t)\} = \pi'(s_t|\theta_\pi)$.
    Uniformly sample an action $j$ from $A_t$: $a_t^j = \rho_j(s_t) + \mathcal{N}_t$.
    Execute action $a_t^j$ and observe reward $r_t$ and state $s_{t+1}$.
    Store transition $(s_t, a_t^j, r_t, s_{t+1})$ to replay buffer $R$.
    Sample a random batch of size $B$ from $R$.
    Set $y_i = r_i + Q'(s_{i+1}, \pi'(s_{i+1}|\theta'_\pi)|\theta'_Q)$.
    Update critic by minimizing the loss,
        $L = \frac{1}{B}\sum_i (y_i - Q(s_i, a_i^j|\theta_Q))^2$
    Update all actor weights connected to $a_t^j$.
        $\nabla_{\theta^\pi} J \approx \frac{1}{B}\sum_i \nabla_{a_i^j} Q(s, a|\theta^Q)|_{s=s_i, a=a_i^j} \nabla_{\theta_\pi - \theta_\pi^{\{1\dots M\}} + \theta_\pi^j} \pi(s|\theta^\pi)|_{s_i}$
    Update the target networks:
        $\theta'_\pi \leftarrow \tau\theta'_\pi + (1-\tau)\theta_\pi$
        $\theta'_Q \leftarrow \tau\theta'_Q + (1-\tau)\theta_Q$
  **end for**
**end for**
```

5 Experiments

In this section, we evaluate the MAPG algorithm on six different Mujuco classic control tasks. We also compare the performance of our method with other popular continuous control algorithms.

5.1 Environments

5.1.1 Mujoco

MuJoCo [**todorov2012mujoco**] is a physics engine which provides simulations of various robotic environments involving contact dynamics. These environments range from simple pendulum on cart balancing problem to complex bipedal movement of a humanoid. The different Mujoco environments we used for our experiments are listed in Table 5.1. Screenshots of different environments are given in Figure 5.1.

5.1.2 TORCS

The open racing car simulator (TORCS) is a 3D, multi-player car racing environment. `TORCS` provide an interface for agents to drive cars. It provides 29 sensors readings at each time-step of simulation, these sensor readings include car speed, angle with the road, distance from track etc. which are enough to make driving decisions. The car can be controlled using three actions, steer $\in [-1, 1]$, brake $\in [0, 1]$ and accelerate $\in [0, 1]$. In our experiments, we use 0.04 seconds long discrete time steps during simulation. During training, the reward was set proportional to the component of car velocity along the direction of road $v * \cos(\alpha)$, where α is the angle between the velocity vector and the center line of the track. This reward encourages forward motion.

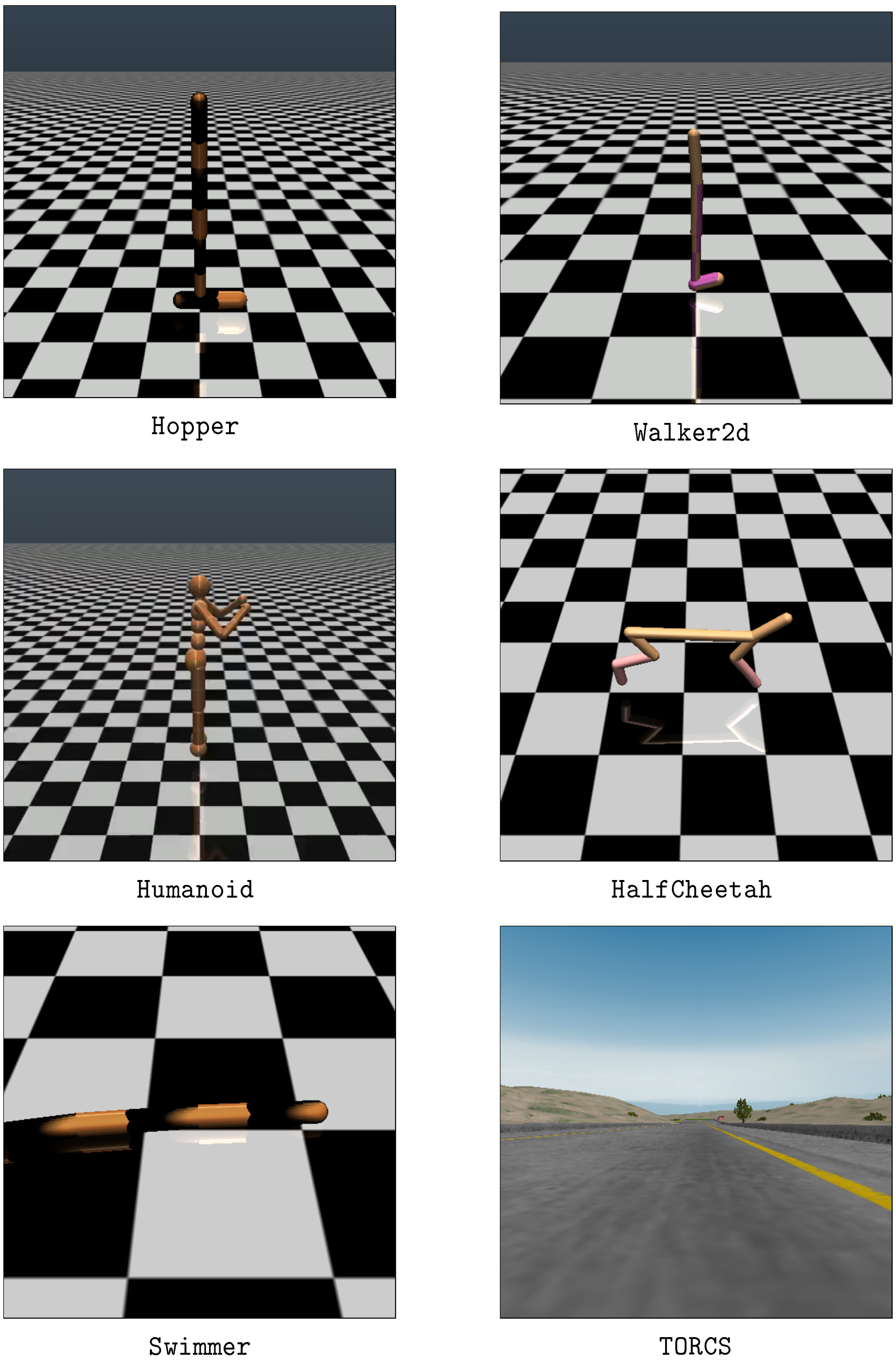


Figure 5.1: Screenshots of different environments used for evaluation.

Table 5.1: A detailed description of the tasks used for evaluation. We have chosen a variety in tasks with continuous action space that differ in both state and action dimensionality.

Task	Action Dimension	State Dimension	Description
Pendulum	1	3	Pendulum on a cart.
Hopper	3	11	One legged robot.
Walker2d	6	17	Two dimensional bipedal robot.
Humanoid	17	376	Three dimensional bipedal robot.
HalfCheetah	6	17	Two leg robot.
Swimmer	2	6	Three joint swimming robot.
TORCS	3	29	Control car in 3D simulation.

5.2 Training Setup

Here we describe the setup used for training different algorithms.

5.2.1 MAPG and DDPG

We use the OpenAI Gym [**greg2016**] and the OpenAI baselines [**openaibase**] for evaluating our experiments. The base actor and critic network architectures are fixed in all experiments. Each network has two fully connected hidden layers with 64 units each. Each fully-connected layer is followed by a ReLU non-linearity. The actor network takes the current observed state s_t as input and produces M normalized actions $a_t^{(m)} \in [-1, 1]^d$ by applying tanh. We then scale the predicted actions to the actual range specific to the current environment. This allows us to have the same learning rate even when dealing with different action ranges. From the M actions $a_t^{(m)}$, a single action a_t is randomly chosen with equal probability.

The critic network uses the current state s_t and action a_t as input and outputs a scalar value (Q-value). The action value is concatenated with the output of the first layer followed by one hidden layer and an output layer with one unit. The critic network is trained by minimizing the mean square loss between the calculated discounted reward and the computed Q value.

The actor network is trained by computing the policy gradient from the Q-value of the chosen action. The network weights of the last layer are only updated for the selected action. This is achieved by setting a zero loss/gradient for the remaining $M - 1$ actions. Ornstein-Uhlenbeck process noise is added to the action values from the actor for exploration. The parameters for the OU process are fixed for all experiments as $\mu = 0$, $\sigma = 0.2$ and $\theta = .15$. The training is carried out for a total of two million steps in all tasks.

5.2.2 A3C

For A3C training, we use same actor-critic networks as for the earlier experiment. The output of actor network is a mean vector (μ_a) (one for each action value) and a scalar standard deviation (σ^2, shared for all actions). The actions values are sampled from the normal distribution $\mathcal{N}(\mu_a, \sigma^2)$. We used differential entropy of a normal distribution to encourage exploration with weight 10^{-4}. The networks are trained with four asynchronous threads and the Adam optimizer is used to train the networks.

5.2.3 SVG

SVG(0) is the model-free variant of stochastic value gradients. SVG(0) is very similar to DDPG, but the policy is made stochastic by parameterizing the actor with a random variable sampled from $\mathcal{N}(0, 1)$. We used off-policy training with experience replay for training SVG(0).

5.3 Results

For more meaningful quantitative results, we report the average reward over 100 episodes with different values of M for various tasks in Table 5.2. In Addition to the scores of MAPG, we report the scores of DDPG, A3C and SVG(0) which are the closest related methods from the literature. For almost all environments we score higher with $M = 5$. The lower performance in the `Humanoid` task for $M = 5$ might be explained by the drastically higher dimensionality of the world state in this task which makes it more difficult to observe.

The scores of policy based reinforcement learning algorithms can vary a lot depending on network hyper-parameters, reward function and codebase/framework as outlined in [**henderson2017deep**]. To minimize the variation in score due to these factors, we fixed all parameters of different algorithms and only studied changes on the score by varying M. Our metric for performance in each task is average reward over 100 episodes by an agent trained for 2 million steps. This evaluation hinders actors with high M since in every training step only a single out of the M actions will be updated per state. Thus, in general actors with the higher number of action proposals, will need a longer time to learn a meaningful distribution of action. This also means that an actor with higher M will take longer to train all heads sufficiently long. This is partially remedied by the fact that we only split the last layer of the actor into the different actions. With this, most of the actor is updated at every training step and the additional number of training iterations scales slower than linear with increased M.

In Figure 5.4, we studied the variance in action values for $M = 10$ during training together with the achieved reward. The standard deviation of actions generated by MAPG decreases with time. As the network converges to a good policy (increase in expected reward) the variation in action values is reduced. However, there are some spikes in standard deviation even when the network has converged to a better policy. It shows that there are situations in which the policy sees multiple good actions (with high Q-value) which can be exploited using MAPG.

This is an interesting aspect of the algorithm. In many tasks, there exist a good amount of states where only one single action is optimal. In these cases we expect all actions of MAPG to collapse into a single (or very similar) action. However, in certain situations, it might be beneficial to act non-deterministically. We will analyze this observation further in the next section.

In our experiments, A3C performed poorly than DDPG in all tasks and was not able to learn a good policy for `Humanoid` task. SVG(0) performed rather poorly on almost all tasks.

5.3.1 Mujoco

We show the box plots for the scores of Mujoco environments in Figure 5.2, where we can see that the overall score usually increases with M. If we set $M = 1$ our algorithm is equal to DDPG, so its performance can also be seen in Figure 5.2. `HalfCheetah` has the most unstable training procedure for all algorithms, often a good policy is lost and the algorithms degenerate back to a zero reward policy where they begin to learn again during training.

5.3.2 TORCS

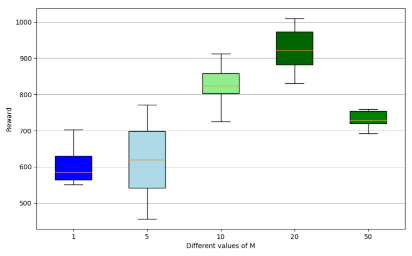
In `TORCS` experiments, MAPG with $M = 10$ was able to complete multiple laps of the track, whereas the DDPG based agent could not complete even one lap of the track. The average distance traveled over 100 episodes by DDPG is 807m and 5882m (both in meters) for our MAPG agent.

Table 5.2: Comparison of average score $\pm 3\sigma$ over 100 episodes for Mujoco tasks between DDPG and MAPG.

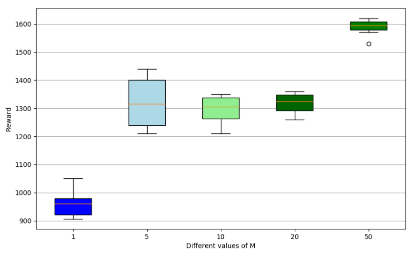
Environment	DDPG	$M = 5$	$M = 10$	$M = 20$	$M = 50$
Hopper-v1	603 ± 76	619 ± 157	824 ± 94	**923** ± 90	732 ± 34
Walker2d-v1	960 ± 72	1318 ± 115	1297 ± 70	1319 ± 50	**1589** ± 45
Humanoid-v1	1091 ± 65	809 ± 72	**1248** ± 115	1112 ± 75	1212 ± 110
HalfCheetah-v1	4687 ± 455	5052 ± 125	**6659** ± 570	4116 ± 85	4333 ± 70
Swimmer-v1	38 ± 7	45 ± 8	**51** ± 6	41 ± 4	40 ± 2

Table 5.3: Comparison of average score $\pm 3\sigma$ over 100 episodes for Mujoco tasks between DDPG, A3C, SVG(0) and MAPG with $M = 10$.

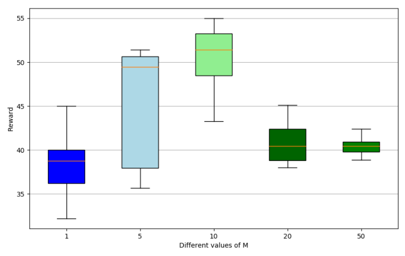
Environment	DDPG	A3C	SVG(0)	$M = 10$
Hopper-v1	603 ± 76	532 ± 105	181 ± 62	**824** ± 94
Walker2d-v1	960 ± 72	764 ± 112	613 ± 118	**1297** ± 70
Humanoid-v1	1091 ± 65	281 ± 40	412 ± 82	**1248** ± 115
HalfCheetah-v1	4687 ± 455	3803 ± 125	429 ± 76	**6659** ± 570
Swimmer-v1	38 ± 7	33 ± 10	41 ± 5	**51** ± 6



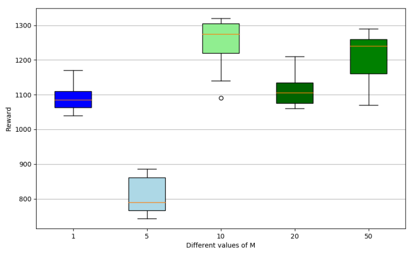
(a) `Hopper`.



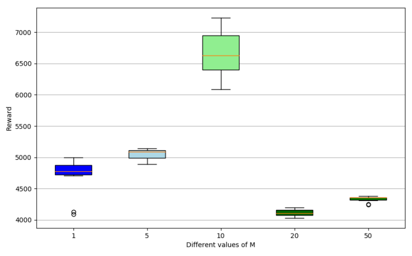
(b) `Walker2d`.



(c) `Swimmer`.



(d) `Humanoid`.



(e) `HalfCheetah`.

Figure 5.2: Variation in score of (from top left) `Hopper`, `Walker2d`, `Swimmer`, `Humanoid` and `HalfCheetah` with different values of M.

5.4 Action Variance Analysis

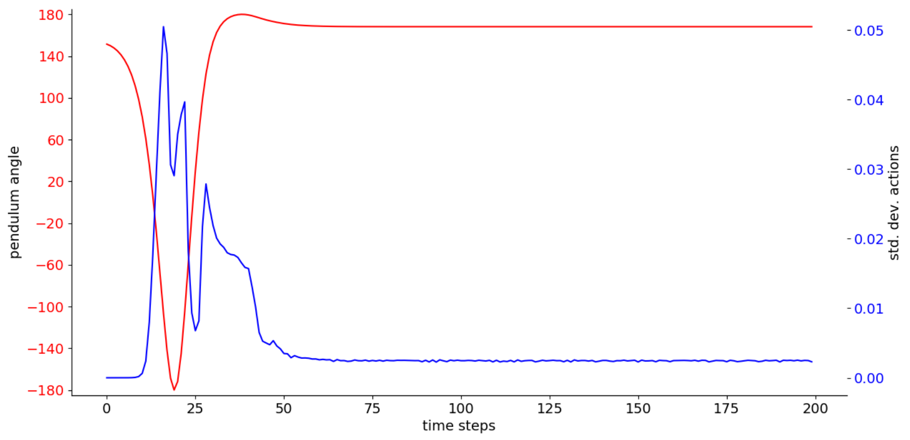


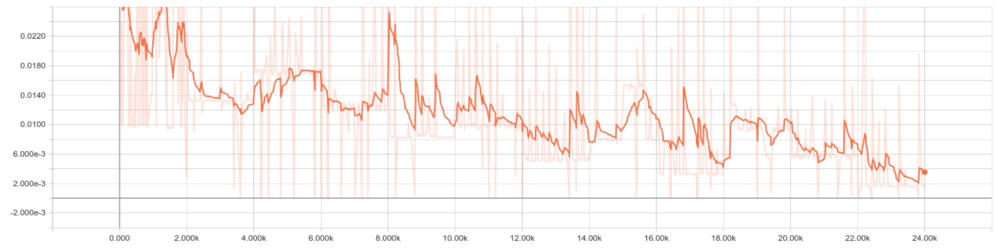
Figure 5.3: Standard deviation and angle during one episode of the `Pendulum` environment. An angle of ± 180 is the target inverted pose. 0 is hanging downwards.

We use the simple `Pendulum` environment to analyze the variance during one episode. The task is the typical inverted pendulum task, where a cart has to be moved such that it balances a pendulum in an inverted position.

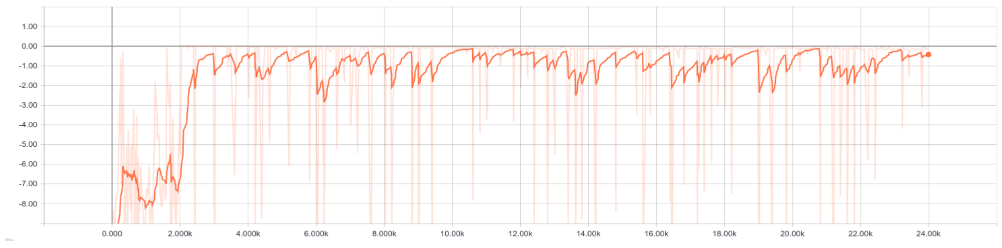
Figure 5.3 plots standard deviation and the angle of the pendulum. Some interesting relationships can be observed. The variance exhibits two strong spikes that coincide with an angle of 0 degrees. This indicates that the agent has learned that there are two ways it can swing up the pole: either by swinging it clockwise or counterclockwise. A deterministic agent would need to pick one over the other instead of deciding randomly. Further, once the target inverted pose (at 180 degrees) is reached the variance does not go down to 0. This means that for the agent a slight jitter seems to be the best way to keep the pendulum from gaining momentum in one or the other direction.

Further, it seems to have learned that slowing the pendulum down close to the upright position, is possible with some noisy torque which explains the second peak in Figure 5.3 at timesteps 30-40.

With this analysis, we could show that a MAPG agent can learn meaningful policies. The variance over predicted actions can give additional insight into the learned policy and results in a more diverse agent that can, for example, swing up the pole in two different directions instead of picking one.



(a) Standard deviation in action values.



(b) Reward.

Figure 5.4: Standard deviation and reward with $M = 10$ for the `Pendulum` task during training.

5.5 Value Variance Analysis

To validate the proof that an optimal policy consists of M actions with equal value at each timestep we compute the average relative deviation from the mean action value as

$$\frac{\max_m Q^\rho(s_t, \rho_m(s_t)) - \min_m Q^\rho(s_t, \rho_m(s_t))}{\frac{1}{m}\sum_m |Q^\rho(s_t, \rho_m(s_t))|}. \tag{5.1}$$

This estimates how much the maximal difference between Q values fluctuates in percent over an episode. For the `Pendulum` environment with $M = 10$ we compute an average, relative deviation of 0.2%. This means that indeed the predicted value for all actions is very close to equal.

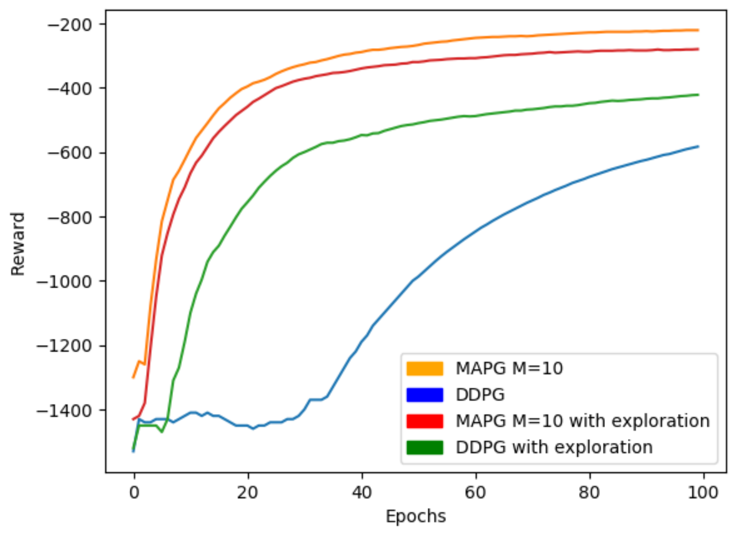
We have further evaluated different strategies for choosing an action during evaluation of a policy. Even though values are very close we have tried choosing the highest value action instead of uniform random choice and observed identical performance. Further, also selecting a fixed action - for example always the first action - did not change the performance of the method. This further validates that the predicted actions are indeed equal in performance as expected from our proof in Section 4.2.

5.6 Effect on Exploration

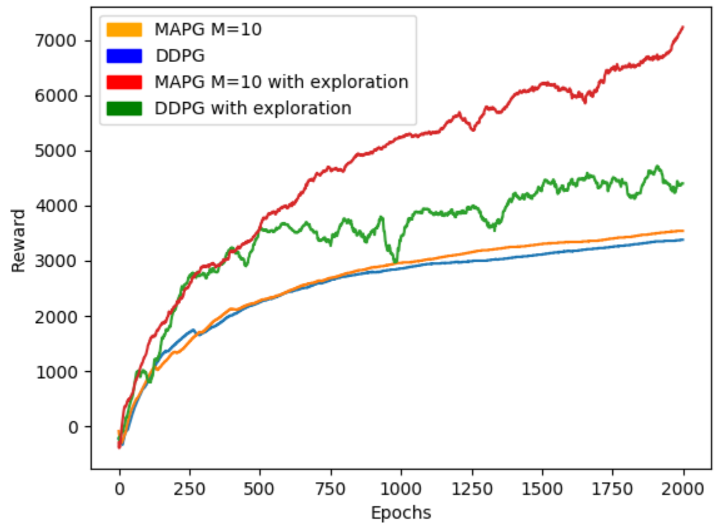
Here, we study the effect of MAPG on exploration during training. We compare the performance of DDPG and MAPG during training with and without any external noise on `Pendulum`, `HalfCheetah` and `Walker2d` environments. Figure 5.5 shows the average reward during training with DDPG and MAPG $M = 10$. The policy trained using MAPG converges to better average reward than DDPG in all cases. Moreover, the performance of MAPG without any external exploration is comparable to DDPG with added exploration noise. This means MAPG can explore the state space enough to find a good policy.

In the `HalfCheetah` environment we can see that using exploration creates a much bigger performance difference between DDPG and MAPG than without. The difference sets in after about 500 epochs. This is an indication that in the beginning of training the actions predicted by MAPG are similar to the one from DDPG. The noise later helps to pull the M actions apart, leading to a more diverse policy with better reward. In `Walker2d` environment, the difference in performance with MAPG and DDPG is not wide as `HalfCheetah`. This could be because $M = 10$ is not optimal for this environment.

Similar to our other experiments we find that MAPG agents explore more possibilities due to their stochastic nature and can then learn more stable and better policies.



(a) `Pendulum`.



(b) `HalfCheetah`.

Figure 5.5: Performance curves for two environments with and without external exploration noise on DDPG and MAPG: original DDPG with OU process noise (green), DDPG without any exploration noise (blue), MAPG (M=10) with OU process noise (red) and MAPG (M=10) without OU process noise (orange).

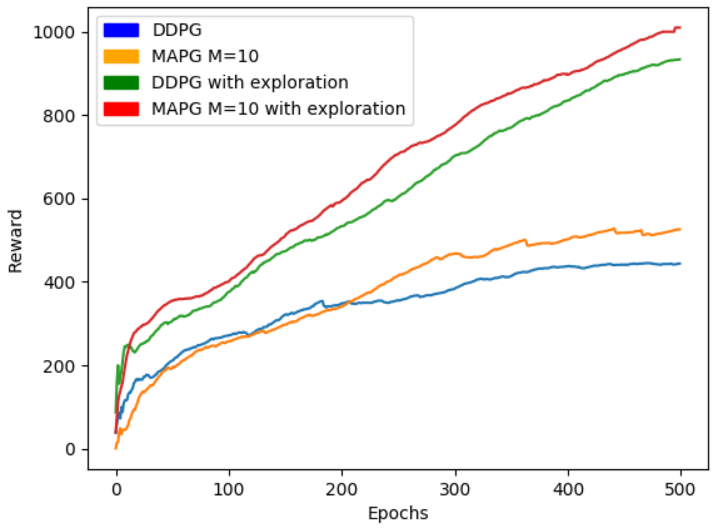


Figure 5.6: Performance of `Walker2d` on DDPG and MAPG with and without external noise.

6 Conclusion and Future Work

As a part of the thesis work, we investigated stochastic continuous action space algorithms. First, we proposed a stochastic continuous control algorithm (MAPG). In experiments, we compared the performance of MAPG on various continuous control tasks with other stochastic control algorithms. The proposed method enabled better exploration of the state space and showed improved performance over DDPG. As indicated by exploration experiments, it can also be a used as a standalone exploration technique, although more work needs to be done in this direction. We also experimented with training from pixels directly, but the result were not comparable to sensor input based training.

6.1 Future Work

Although, our methods show good results on Mujoco environments as compared with DDPG, A3C, and SVG, there is further scope for improvement in the following areas:

6.1.1 On-policy Algorithms

We evaluated our method on a variation of DDPG which is off-policy. There are a lot of on-policy algorithms for continuous action space problems such as Proximal Policy Optimization (PPO) [**SchulmanWDRK17**], Trust Region Policy Optimization (TRPO) which show good performance on several control environments. The evaluation of MAPG on on-policy algorithms will enable studying the generality of the approach.

6.1.2 Visual Environments

In our experiments, we only did evaluations with sensor-based input data and didn't have much success with visual input. Further experiments with visual environments will give us more insights into training from raw pixel data and increase the applicability of the method.

6.1.3 Exploration

In exploration experiments, MAPG was able to learn a good policy without any external policy. Though, it wasn't comparable to policies learned with external exploration noise. Using an additional loss term which encourages the M actions to differ initially and then decays over time could improve the exploration of the algorithm.

6.1.4 Model Based Reinforcement Learning

The idea of multiple action prediction can be extended to model-based reinforcement learning where the model predicts multiple future states at each time step. It will provide a rich representation for transition probability function in continuous environments as there are usually a large number of valid future states at every time step.

List of Figures

List of Tables