Karlsruhe Institute of Technology

OpenAI

Parameter Space Noise for Exploration in Deep Reinforcement Learning

Master Thesis of

Matthias Plappert

At the Department of Informatics
Institute for Anthropomatics and Robotics (IAR)
High Performance Humanoid Technologies Lab (H^2T)
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

OpenAI
San Francisco, United States

Referee:	Prof. Dr.-Ing. Tamim Asfour
First advisor:	Dr. Marcin Andrychowicz
Second advisor:	Prof. Pieter Abbeel

Duration: June 1st, 2017 – November 30th, 2017

Ich versichere hiermit, dass ich die Arbeit selbstständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht habe und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissentschaftlicher Praxis beachtet habe.

Karlsruhe, den 30.11.2017

. . . . . .

(**Matthias Plappert**)

Kurzzusammenfassung

Reinforcement Learning bietet ein abstraktes Rahmenwerk, um eine Vielzahl von Problemen zu lösen. Beispiele reichen hier von der Steuerung von Produktionsprozessen, dem Handeln von Wertpapieren, bis hin zum Spielen von Brettspielen wie Backgammon und Go auf professionellem Niveau. In der Robotik stellt Reinforcement Learning eine vielversprechende Möglichkeit dar, selbstlernende Robot zu realisieren.

Der Erfolg von Reinforcement Learning-Algorithmen hängt entscheidend von derer Fähigkeit zur Exploration ab. Da der Algorithmus zu Beginn keinerlei Information über seine Umgebung hat, müssen diese zunächst gesammelt werden. Obwohl eine Vielzahl von Explorationsmethoden existieren, finden in der Praxis vor allem immer noch simple Methoden Anwendung, welche die Aktionen, die der Algorithmus wählt, verrauschen. Mithilfe dieser so gesammelten Informationen wird schließlich die Strategie, nach der der Algorithmus handelt, verbessert. Oft wird diese Strategie durch Funktionsapproximatoren realisiert, beispielsweise (tiefe) neuronale Netze.

In der hier vorliegenden Arbeit erforschen wir, ob anstatt Exploration durch Verrauschen der Aktionen auch Exploration durch Verrauschen der Gewichte einer parametrisierten Strategie erreicht werden kann. In einer theoretischen Diskussion zeigen wir auf, dass solch ein Verfahren zu konsistenterer Exploration im Vergleich zu traditionellen Verfahren führt. Anschließend benennen, analysieren, und lösen wir zwei wesentliche Probleme, die beim Verrauschen der Gewichte tiefer neuronaler Netze eine wichtige Rolle spielen.

Die theoretischen Überlegungen werden anschließend experimentell validiert. Zunächst zeigen wir, dass unser Verfahren signifikant anderes Explorationsverhalten aufweist, als solche, welche im Aktionsraum explorieren. Anschließende Experimente zeigen, dass Rauschen im Parameterraum mit modernen Reinforcement Learning-Algorithmen wie Deep Q-Networks (DQN) oder Deep Deterministic Policy Gradient (DDPG) erfolgreich kombiniert werden können um anspruchsvolle Probleme erfolgreich zu erlernen. Weitere Experimente zeigen, dass unser hier vorgestelltes Verfahren vor allem bei Problemen, welche nur wenige Lernsignale absondern, traditionellen Methoden deutlich überlegen ist. Zuletzt evaluieren wir das Verhalten unseres Verfahren mithilfe dreier realistischer Roboter-Manipulations-Experimente mit komplizierten Kontaktdynamiken und dünnbesetzten Belohnungen. In diesen Experimenten zeigt sich, dass allein durch Exploration durch Parameterrauschen in einem Teil dieser extrem fordernden Probleme gelernt werden kann. Weiterhin lässt sich Parameterrauschen mit anderen Explorationsmethoden wie Hindsight Experience Replay (HER) kombinieren und führt damit insgesamt zu verbessertem Explorationsverhalten im Vergleich zu HER mit traditionellem Aktionsrauschen.

Die hier vorliegende Arbeit zeigt, dass Parameterrauschen erfolgreich zur Exploration in Anwendungen des Reinforcement Learnings verwendet werden kann und in der Regel traditionellen Methoden, welche Rauschen im Aktionsraum verwenden, überlegen ist. Unsere Experimente zeigen weiterhin, dass Parameterrauschen sowohl in Problemen mit diskreten wie auch kontinuierlichen Aktionsräumen Anwendung findet und sich erfolgreich mit komplexeren und anwendungsspezifischeren Explorationsmethoden kombinieren lässt.

Abstract

The success of any reinforcement learning algorithm hinges on its ability to explore efficiently. While many sophisticated approaches have been proposed in recent years, today's state-of-the-art algorithms still rely on traditional action space noise due to its simplicity. In this thesis, we propose a novel method: *parameter space noise*. Parameter space noise is conceptually simple yet allows state-of-the-art algorithms to learn in environments in which traditional action space noise exploration fails. We theoretically justify why we expect this approach to outperform traditional action space noise, analyze potential pitfalls when applying it to policies realized by deep neural networks, and propose approaches that mitigate these problems. Next, we evaluate our proposed approach on a large variety of different environments where we show that parameter space noise indeed outperforms action space noise in the majority of cases. We also demonstrate that parameter space noise can be combined with other exploration methods and show that this combination is capable of learning successful strategies on a set of complex and realistic robot manipulation tasks.

Acknowledgement

I would like to thank the entire OpenAI team. In particular, I would like to thank Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Pieter Abbeel, Alex Ray, Bob McGrew, Jonas Schneider, Josh Tobin, Peter Welinder, Wojciech Zaremba, Filip Wolski, Vikash Kumar, Ankur Handa, Greg Brockman, Ilya Sutskever, Jack Clark, Ryan Lowe, Smitha Milli, Avital Oliver, Jean Harb, and Vicki Cheung, who have all helped to make my time at OpenAI a success.

Marcin Andrychowicz has been an amazing advisor during my time at OpenAI and I would like to thank him in particular for many insightful discussions, for always providing very thoughtful advice, and for always asking the right questions to reveal the not yet fully understood parts of my research.

A very special thank you to Christian Mandery and Tamim Asfour, who I have met during my Bachelor thesis and who have both been amazing advisors since. Without your continued support, I would simply not be where I am today.

And finally, last but by no means least, to Laura Tessin, who has always supported me no matter what. Thank you for everything!

Contents

1. Introduction

In recent years, reinforcement learning has achieved impressive results on a wide variety of challenging tasks like learning to play Atari video games directly from pixels [Mnih et al., 2015], learning policies for complex continuous control problems that involve locomotion and manipulation [Schulman et al., 2015a,c, Lillicrap et al., 2015, Gu et al., 2016, Heess et al., 2017], and even beating professional Go players by training a system that only plays against itself [Silver et al., 2017]. In robotics in particular, deep learning and deep reinforcement learning have similarly achieved impressive results. Robots are now capable of learning complex manipulation tasks like opening a bottle, putting cloths on a hanger, and precisely fitting small pieces into a larger structures [Levine et al., 2015, 2016, Pinto and Gupta, 2016].

However, the success of reinforcement learning hinges on the agent's capabilities to effectively *explore* its environment. This is necessary because, initially, the agent has no knowledge about its environment and has to try different strategies in order to find successful ones. Even as learning progresses, it becomes important to strike a balance between exploring and exploiting learned behavior, which is the long-standing *exploration vs. exploitation dilemma* [Kumar and Varaiya, 1986, Bertsekas, 1987, Thrun, 1992, Kearns and Singh, 2002, Brafman and Tennenholtz, 2002].

In recent years, many sophisticated exploration strategies have been proposed that rely on complex additional structures such as counting tables [Tang et al., 2016], density modeling of the state space [Ostrovski et al., 2017], learned dynamics models [Houthooft et al., 2016, Achiam and Sastry, 2017, Stadie et al., 2015], or self-supervised curiosity [Pathak et al., 2017b] (see Chapter 3 for a more in-depth coverage). However, since these approaches are often quite complicated and computationally expensive, simple action space exploration like ϵ-greedy and additive Gaussian noise is still commonly found in almost all reinforcement learning papers that are not specifically concerned with exploration. Additionally, even these advanced exploration methods often require some form of "inner exploration" method, by which we mean that they often extend the reward function but still rely on traditional action space noise to find high reward states.

Given the importance of exploration in reinforcement learning and the fact that action space noise is still omnipresent in today's state-of-the-art algorithms, we believe it is worthwhile to reconsider the basic assumption that we should use action space noise [Rückstieß et al., 2008]. In this thesis, we propose a novel method called *parameter space noise*, which is designed to be a drop-in replacement for action space noise in current state-of-the-art

deep reinforcement learning algorithms. Due to its conceptual simplicity and almost trivial implementation, we believe that it is a promising candidate to replace traditional action space noise.

This thesis is organized as follows. We first introduce and review the most important concepts of reinforcement learning and deep learning in Chapter 2. In this section, we also review important deep learning algorithms that are used throughout this thesis. Next, We discuss related work, especially recent exploration methods in the context of deep reinforcement learning, in Chapter 3. In the following section, Chapter 4, we introduce parameter space noise for exploration, analyze it theoretically, discuss potential problems when applying it to policies realized through deep neural networks, and introduce approaches to mitigate these problems. Chapter 5 evaluates the performance of the proposed parameter space noise exploration. We show that our proposed method explores more efficiently in a wide range of challenging environments. Our approach is applicable to both on- and off-policy algorithms and works with discrete and continuous action spaces. We further show that parameter space noise can learn successful strategies on environments with extremely sparse rewards where exploration is truly important. A final set of experiments demonstrates that parameter space noise can be combined with advanced techniques for exploration like Hindsight Experience Replay (HER, Andrychowicz et al. [2017]) by applying it to a set of challenging robot manipulation tasks with realistic real-world physics. Finally, we summarize our work in Chapter 6 and point out promising directions for future work.

2. Background

2.1 Reinforcement Learning

In reinforcement learning [Sutton and Barto, 1998], an agent interacts with an environment by executing actions. The environment in return yields its new state as well as a scalar reward. Crucially, the agent can only interact with the environment through actions; it cannot modify the way the environment works or the way it yields rewards. In this interaction, the agent aims to maximize the total return (i.e. the sum of rewards) it obtains. This fundamental interaction between agent and environment is depicted in Figure 2.1.

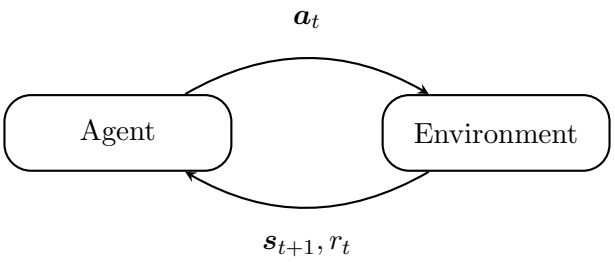


Figure 2.1: *The fundamental interaction in reinforcement learning between agent and environment. The agent choses an action $\boldsymbol{a}_t$ and sends it to the environment. The environment then yields the new state $\boldsymbol{s}_{t+1}$ as well as the reward that was obtained $r_t := r(\boldsymbol{s}_t, \boldsymbol{a}_t)$. This process is repeated for all time steps.*

We now formalize the aforementioned description of agent and environment and its interaction. A Markov Decision Process (MDP) is defined as $M = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \rho_0, r)$ where $\mathcal{S}$ is the state space and $\mathcal{A}$ is the action space. We define the reward function as $r : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ and sometimes write $r_t := r(\boldsymbol{s}_t, \boldsymbol{a}_t)$ for the reward obtained in time step t. $\rho_0 : \mathcal{S} \mapsto [0, 1]$ is the probability distribution over initial states. Finally, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1]$ denotes the transition probability distribution, i.e. given state $\boldsymbol{s}_t$ and $\boldsymbol{a}_t$, how likely do we end up in state $\boldsymbol{s}_{t+1}$. Notice the Markov property: The next state only depends on the current state and action.

In reinforcement learning, we typically wish to find a policy $\pi : \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$ that

maximizes the expected discounted return:

$$\eta(\pi) := \mathbb{E}_{\boldsymbol{s}_0 \sim \rho_0, \boldsymbol{a}_t \sim \pi(\cdot|\boldsymbol{s}_t), \boldsymbol{s}_{t+1} \sim \mathcal{P}(\cdot|\boldsymbol{s}_t, \boldsymbol{a}_t)} \left[\sum_{t=0}^{T-1} \gamma^t r(\boldsymbol{s}_t, \boldsymbol{a}_t) \right]. \tag{2.1}$$

To simplify notation, we sometimes use $\tau^\pi = (\boldsymbol{s}_0, \boldsymbol{a}_0, \boldsymbol{s}_1, \boldsymbol{a}_1, \ldots, \boldsymbol{s}_{t-1}, \boldsymbol{a}_{t-1}, \boldsymbol{s}_T)$ to denote a trajectory with $\boldsymbol{s}_0 \sim \rho_0$, $\boldsymbol{a}_t \sim \pi(\cdot \mid \boldsymbol{s}_t)$, and $\boldsymbol{s}_{t+1} \sim \mathcal{P}(\cdot \mid \boldsymbol{s}_t, \boldsymbol{a}_t)$. Additionally, we write $r(\boldsymbol{s}_T, \boldsymbol{a}_T)$ for the terminal reward although it has no dependence on $\boldsymbol{a}_T$. $\gamma \in [0, 1)$ is a discount factor which ensures that the return is bound even if $t \to \infty$ and can be used to select a time horizon over which we optimize.[1] T denotes the time horizon of the MDP. Finally, we sometimes write $\pi : \mathcal{S} \mapsto \mathcal{A}$ to denote a deterministic policy.

An important problem in reinforcement learning is that of *exploration vs. exploitation*. More concretely, we generally do not have any information about $\rho_0(\boldsymbol{s}_0)$ and $\mathcal{P}(\boldsymbol{s}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{a}_t)$ initially. The agent thus has to explore its environment in order to find states that yield high rewards. On the other hand, the agent also has to exploit what it has already learned in order to achieve high rewards. To complicate things further, the state space can often be extremely large and may even be infinite.

2.2 Deep Learning

Deep learning [LeCun et al., 2015, Goodfellow et al., 2016] has been exceedingly successful in recent years in a variety of domains like image classification [Krizhevsky et al., 2012, Szegedy et al., 2017], speech recognition [Hinton et al., 2012, Graves et al., 2013], and machine translation [Sutskever et al., 2014, Bahdanau et al., 2014]. Deep learning utilizes artificial neural networks (ANNs or NNs) that have been around for centuries [McCulloch and Pitts, 1943, Rosenblatt, 1958]. However, instead of using only shallow networks, deep learning uses networks that consist of many layer. This allows networks to learn feature representations themselves in contrast to more traditional approaches that required cumbersome and inefficient feature engineering. While making networks deeper seems trivial at first, it actually took significant effort to overcome problems like vanishing gradients [Hochreiter, 1998, Maas et al., 2013], instability in training due to covariate shift [Ioffe and Szegedy, 2015], and flaky optimization [Sutskever et al., 2013]. New architectures like convolutional neural networks (CNNs, Sermanet et al. [2012]) and long-short term memory (LSTM, Hochreiter and Schmidhuber [1997]) further pushed the field forward.

However, the basic building block of a neural network is still relatively simple. A fully connected layer consists of N input units that are, well, fully connected to M output units:

$$\boldsymbol{h} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}, \tag{2.2}$$

where $\boldsymbol{W} \in \mathbb{R}^{M \times N}$ is the *weight matrix*, $\boldsymbol{b} \in \mathbb{R}^M$ is the *bias vector*, $\boldsymbol{x} \in \mathbb{R}^N$ is the *input vector* and $\boldsymbol{h} \in \mathbb{R}^M$ is the *hidden* (or output) *vector*. In essence, Equation (2.2) describes an affine transformation, with learnable parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. Figure 2.2 depicts the schematics of a neural network with two hidden layers.

In order to learn non-linear functions and in order to meaningfully combine many layers,[2] a non-linearity is added on top of the affine transformation:

$$\boldsymbol{h} = f\left(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}\right), \tag{2.3}$$

[1] For example, if we set $\gamma = 0$, we effectively only consider the immediate reward and greedily optimize for that.

[2] In theory, combining multiple linear layers does not make sense since it could be re-written as a single linear layer. However, and interestingly, computers only approximate floating point numbers, which can actually be seen as an implicit non-linearity. See `https://blog.openai.com/nonlinear-computation-in-linear-networks/` for details on this matter.

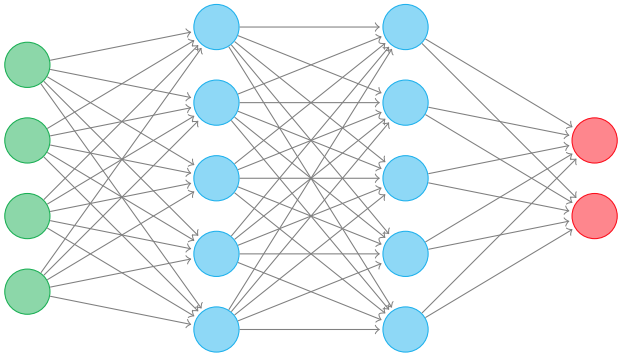


Figure 2.2: *A neural network with an input layer (in green), two fully connected hidden layers (in blue), and an output layer (in red). The bias vectors are omitted for readability.*

with $f(\cdot)$ being some non-linear and differentiable function, sometimes also referred to as the *activation function*. Common choices are the hyperbolic tangent, the rectifier linear unit, and the softmax activation function:

$$f(x_i) = \tanh(x_i) \qquad f(x_i) = \begin{cases} x_i & \text{if } x_i \geq 0, \\ 0 & \text{otherwise} \end{cases} \qquad f(\boldsymbol{x}) = \frac{\exp \boldsymbol{x}}{\sum_i \exp x_i}. \tag{2.4}$$

In order to train such a network, a *loss function* has to be defined. The concrete loss function depends on the use case. For a regression, a common choice is the squared error between prediction (denoted as $\widehat{\boldsymbol{y}}$) and ground truth (denoted as $\boldsymbol{y}$):

$$\mathcal{L} = \|\boldsymbol{y} - \widehat{\boldsymbol{y}}\|^2 . \tag{2.5}$$

Since $\widehat{\boldsymbol{y}} = g_\theta(\boldsymbol{x})$, where $g(\cdot)$ denotes the entire network and $\boldsymbol{\theta}$ denotes all learnable parameters (e.g. $\boldsymbol{W}$ and $\boldsymbol{b}$), we can differentiate $\mathcal{L}$ w.r.t. $\boldsymbol{\theta}$ to obtain the gradient $\nabla_{\boldsymbol{\theta}}\mathcal{L}$.[3] Since the gradient points in the direction of the largest increase, we can take a small step in the opposite direction, thus iteratively minimizing the loss function and therefore the prediction error:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha \nabla_{\boldsymbol{\theta}_k} \mathcal{L}, \tag{2.6}$$

where $\alpha \in \mathbb{R}_{>0}$ is the learning rate and $\boldsymbol{\theta}_k$ denotes the parameters of the network at the k-th iteration of the optimization. The gradient computation itself can efficiently be carried out using reverse-mode automatic differentiation [Linnainmaa, 1970], also commonly referred to as back-propagation in the special case of neural networks [Rumelhart et al., 1988].

2.3 Deep Reinforcement Learning

While the previous section introduced the basic concepts behind reinforcement learning, this section will discuss recent state-of-the-art algorithms that combine deep learning and reinforcement learning. We will briefly review four concrete algorithms: Deep Q-Networks (DQN, Section 2.3.1), Bootstrapped DQN (Section 2.3.2), Deep Deterministic Policy Gradient (DDPG, Section 2.3.3), and Hindsight Experience Replay (HER, Section 2.3.4).

2.3.1 Deep Q-Networks (DQN)

Mnih et al. [2013, 2015] proposed Deep Q-Networks, which successfully learns to play Atari games directly from pixels. In essence, DQN learns the Q-function, which is defined as:

$$Q^\pi(\boldsymbol{s}_t, \boldsymbol{a}_t) := r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \mathbb{E}_{\tau^\pi}\left[\sum_{i=1}^{T-t} \gamma^{t+i} r(\boldsymbol{s}_{t+i}, \boldsymbol{a}_{t+i})\right]. \tag{2.7}$$

[3] This is why we require that the activation function is differentiable.

Algorithm 1 Deep Q-learning with Experience Replay [Mnih et al., 2013, 2015]

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function $Q_{\boldsymbol{\theta}}$ with random weights
for episode $= 1, \ldots, M$ **do**
  Receive initial state $\boldsymbol{s}_0$
  for $t = 0, \ldots, T-1$ **do**
    With probability ϵ select a random action $\boldsymbol{a}_t$
    otherwise select $\boldsymbol{a}_t = \max_{\boldsymbol{a}} Q_{\boldsymbol{\theta}}(\boldsymbol{s}_t, \boldsymbol{a})$
    Execute action $\boldsymbol{a}_t$ and observe reward r_t and next state $\boldsymbol{s}_{t+1}$
    Store transition $(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1}, r_t,)$ in $\mathcal{D}$
    Sample random mini-batch of transitions $(\boldsymbol{s}_i, \boldsymbol{a}_i, \boldsymbol{s}_{i+1}, r_i,)$ from $\mathcal{D}$
    Set $y_i = \begin{cases} r_i & \text{for terminal } \boldsymbol{s}_{i+1} \\ r_i + \gamma \max_{\boldsymbol{a}'} Q_{\boldsymbol{\theta}}(\boldsymbol{s}_{i+1}, \boldsymbol{a}') & \text{for non-terminal } \boldsymbol{s}_{i+1} \end{cases}$
    Perform a gradient descent step on $(y_i - Q_{\boldsymbol{\theta}}(\boldsymbol{s}_i, \boldsymbol{a}_i))^2$

In other words, the Q-function gives us the expected discounted reward if we take action $\boldsymbol{a}_t$ in state $\boldsymbol{s}_t$ and follow policy π therein after. As it turns out, the Q-function can be written recursively, in which form it is currently referred to as the *Bellman equation* [Sutton and Barto, 1998]:

$$Q^{\pi}(\boldsymbol{s}_t, \boldsymbol{a}_t) := r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \gamma \mathbb{E}_{\boldsymbol{s}_{t+1} \sim \mathcal{P}(\cdot|\boldsymbol{s}_t, \boldsymbol{a}_t), \boldsymbol{a}_{t+1} \sim \pi(\boldsymbol{s}_{t+1})} \left[Q^{\pi}(\boldsymbol{s}_{t+1}, \boldsymbol{a}_{t+1}). \right]. \tag{2.8}$$

Since DQN only considers the deterministic case, we will drop the expectations. Furthermore, notice that the Q-function is quite useful: Given Q, we can chose the the optimal deterministic action as follows:

$$\pi(\boldsymbol{s}) := \operatorname*{argmax}_{\boldsymbol{a}} Q^{\pi}(\boldsymbol{s}, \boldsymbol{a}). \tag{2.9}$$

π can thus be implicitly be defined via Q. If the action space is small and discrete, we can further quite easily find the optimal action. However, the state space may still be extremely large, as is the case here since DQN learns directly from pixels. In DQN, the Q-function is thus realized using a deep neural network in the hope that it generalizes to previously unseen states during execution. Using the Bellman equation, the network is then trained to minimize the following loss:

$$\mathcal{L} = \left(r + \gamma \max_{\boldsymbol{a}'} Q_{\boldsymbol{\theta}}(\boldsymbol{s}', \boldsymbol{a}') - Q_{\boldsymbol{\theta}}(\boldsymbol{s}, \boldsymbol{a}) \right)^2. \tag{2.10}$$

In order to be able to compute this, DQN stores the transition tuple $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}', r)$ in a replay buffer. This also stabilizes the algorithm since samples are drawn uniformly from the replay buffer and the gradient is estimated in typical mini-batch fashion using these samples, thus de-correlating it. Furthermore, DQN uses the concept of a target network to compute that is only updated occasionally to make the learning target (mostly) stationary.

In order to explore, DQN typically uses ϵ-greedy exploration which simply selects a random action uniformly with probability ϵ and the greedy action otherwise. Exploration is covered in greater detail in Section 4.1 since this thesis is largely concerned with it.

Algorithm 1 describes the entire algorithm in pseudo-code. For more details about hyper-parameters and exact training procedures, please refer to Mnih et al. [2015]. DQN has also been extended to overcome problems with overly optimistic estimations for Q [van Hasselt, 2010, van Hasselt et al., 2016], improved experience sampling with prioritized experience replay [Schaul et al., 2015b], and improved architectures like dueling networks [Wang et al., 2016]. In all cases, the changes are mostly of incremental nature and do not significantly change the described mechanics of DQN.

Algorithm 2 Bootstrapped DQN [Osband et al., 2016]

Given: a masking distribution $\mathcal{M}$
Initialize replay memory $\mathcal{D}$
Initialize K action-value functions (or heads) $\{Q_k\}_{k=1}^K$ with random weights
for episode $= 1, \ldots, M$ **do**
  Receive initial state $\boldsymbol{s}_0$
  Pick a value function to act using $k \sim \mathcal{U}(\{1, \ldots, K\})$
  for $t = 0, \ldots, T-1$ **do**
    Select action $\boldsymbol{a}_t = \max_{\boldsymbol{a}} Q_k(\boldsymbol{s}_t, \boldsymbol{a})$ according to selected head k
    Execute action $\boldsymbol{a}_t$ and observe reward r_t and next state $\boldsymbol{s}_{t+1}$
    Sample bootstrap mask $\boldsymbol{m}_t \sim \mathcal{M}$
    Store transition $(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1}, r_t, \boldsymbol{m}_t)$ in $\mathcal{D}$
    Sample a mini-batch from $\mathcal{D}$ and perform DQN optimization on $\{Q_k \mid k \in \boldsymbol{m}_i\}$

2.3.2 Bootstrapped DQN

Bootstrapped DQN [Osband et al., 2016] was proposed to improve the exploration behavior of DQN. Instead of having a single estimate for Q like in regular DQN, Bootstrapped DQN uses K networks or heads:[4] $\{Q_k\}_{k=1}^K$. When generating a rollout, a specific k is selected according to $k \sim \mathcal{U}(\{1, \ldots, K\})$, where $\mathcal{U}(\cdot)$ denotes the uniform distribution over the specified set. This Q_k is then used for the entirety of the rollout to select actions, which works identical to regular DQN using Q_k instead of Q. When storing a transition, Bootstrapped DQN samples a *mask* from a masking distribution: $\boldsymbol{m} \sim \mathcal{M}$ with $\boldsymbol{m} \in \{0, 1\}^K$. During training, a transition is only used to train the subset $\{Q_k \mid k \in \boldsymbol{m}_i\}$ of all K network heads, where $\boldsymbol{m}_i$ denotes the mask of a sampled transition.

Since all K heads have different initializations and are trained on different subsets of the experienced data, different Q_k will exhibit different behavior. However, since one Q_k is selected at the beginning of the episode, exploration will be consistent within the episode. This allows Bootstrapped DQN to explore consistently within episodes but (hopefully) still exhibits diverse behaviors across episodes.

Algorithm 2 describes the entire algorithm in pseudo-code. For more details about hyper-parameters and exact training procedures, please refer to Osband et al. [2016].

2.3.3 Deep Deterministic Policy Gradient (DDPG)

In the previously describe DQN, finding the optimal action requires that we can efficiently find the action that corresponds to the optimal Q-value. While this is quite simple for discrete and relatively small action spaces (we can just compute all of them and pick the action that corresponds to the maximum), the problem becomes intractable if $\mathcal{A}$ is continuous. However, continuous action spaces are quite important in applications like robotics, where discretization often is not desirable. Lillicrap et al. [2015] propose DDPG for continuous control that utilizes Q-learning like DQN but relies on an actor-critic architecture.

More concretely, the critic still learns the Q-function as described in the previous section, which we now denote as $Q_{\boldsymbol{\phi}}$. Instead of maximizing over all possible actions (which is intractable for continuous $\mathcal{A}$), we utilize a different neural network, the actor, which we denote as $\pi_{\boldsymbol{\theta}}$: The loss for the critic is thus given as:

$$\mathcal{L}_{\text{critic}} = \left(r + \gamma Q_{\boldsymbol{\phi}}\left(\boldsymbol{s}', \pi_{\boldsymbol{\theta}}(\boldsymbol{s}')\right) - Q_{\boldsymbol{\phi}}(\boldsymbol{s}, \boldsymbol{a})\right)^2, \tag{2.11}$$

[4] A network with multiple heads shares early layers (e.g. the convolutional part), but has separate weights for later layers (the heads).

Algorithm 3 Deep Deterministic Policy Gradient with Experience Replay [Lillicrap et al., 2015]

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize critic network $Q_{\boldsymbol{\phi}}$ with random weights
Initialize actor network $\pi_{\boldsymbol{\theta}}$ with random weights
for episode $= 1, \ldots, M$ **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\boldsymbol{s}_0$
  for $t = 0, \ldots, T-1$ **do**
    Select action $\boldsymbol{a}_t = \pi_{\boldsymbol{\theta}}(\boldsymbol{s}_t) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $\boldsymbol{a}_t$ and observe reward r_t and next state $\boldsymbol{s}_{t+1}$
    Store transition $(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1}, r_t,)$ in $\mathcal{D}$
    Sample random mini-batch of transitions $(\boldsymbol{s}_i, \boldsymbol{a}_i, \boldsymbol{s}_{i+1}, r_i,)$ from $\mathcal{D}$
    Set $y_i = \begin{cases} r_i & \text{for terminal } \boldsymbol{s}_{i+1} \\ r_i + \gamma Q_{\boldsymbol{\phi}}\left(\boldsymbol{s}_{i+1}, \pi_{\boldsymbol{\theta}}(\boldsymbol{s}_{t+1})\right) & \text{for non-terminal } \boldsymbol{s}_{i+1} \end{cases}$
    Update critic by performing a gradient descent step on $\left(y_i - Q_{\boldsymbol{\phi}}(\boldsymbol{s}_i, \boldsymbol{a}_i)\right)^2$
    Update actor using the sampled policy gradient:

$$\frac{1}{N} \sum_i \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\phi}}(\boldsymbol{s}_i, \pi_{\boldsymbol{\theta}}(\boldsymbol{s}_i))$$

which we can optimize for w.r.t. $\boldsymbol{\phi}$ as previously described using a replay buffer that stores $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}', r)$. Notice that this loss is almost identical to Equation (2.10), which again highlights that both DQN and DDPG use Q-learning.

In contrast to DQN, DDPG uses an explicit policy, which is defined by the actor network $\pi_{\boldsymbol{\theta}}$. Since Q is a differentiable network, we can simply train π to maximize Q:

$$\mathcal{L}_{\text{actor}} = -Q_{\boldsymbol{\phi}}\left(\boldsymbol{s}, \pi_{\boldsymbol{\theta}}(\boldsymbol{s})\right), \tag{2.12}$$

which we minimize w.r.t. $\boldsymbol{\theta}$ utilizing states $\boldsymbol{s}$ from the replay buffer. By applying the chain rule, it becomes apparent that this is indeed the deterministic policy gradient described by Silver et al. [2014].

DDPG uses additive Gaussian noise that may be correlated for exploration. Exploration is covered in greater detail in Section 4.1 since this thesis is largely concerned with it.

Algorithm 3 describes the entire algorithm in pseudo-code. For more details about hyper-parameters and exact training procedures, please refer to Lillicrap et al. [2015].

2.3.4 Hindsight Experience Replay (HER)

Hindsight Experience Replay (HER, Andrychowicz et al. [2017]) has recently been introduced as a method that learns even in settings where rewards are extremely sparse. In contrast to the previously described algorithms, HER is more of a meta-algorithm that can be used in combination with any off-policy algorithm that utilizes experience replay.

The key insight that it builds on is the following: Assume you are learning to play ice hockey and you are practicing to hit a goal. At first, you will likely not be successful and the puck will end up in a relatively random place. If the reward were binary, i.e. 1 if the goal was achieved and 0 if the goal was missed, you would not have learned anything at all. You can, however, observe were the puck ended up and simply assume that you

wanted to achieve this goal in the first place. This is of course not what you really wanted to do, but at least you have now learned how to achieve this specific goal instead. HER does precisely this: It replays with goals that were achieved even though they were not initially desired, thus ensuring that there is always a reward signal even though the reward structure is sparse as described before.

This can be formalized by first extending the policy to also consider a goal $\pi : \mathcal{S} \times \mathcal{G} \times \mathcal{A} \mapsto [0, 1]$ or simply $\pi : \mathcal{S} \times \mathcal{G} \mapsto \mathcal{A}$ in the deterministic case. Here, $\boldsymbol{g} \in \mathcal{G}$ denotes a goal that we wish to achieve, for example the Cartesian position of the hockey puck from the previous example. This concept has recently been introduced by Schaul et al. [2015a], who have named it the *Universal Value Function Approximator* (UVFA).

In HER, a goal is sampled before an episode begins and kept constant. After the episode is completed, the rollout is stored in the replay buffer as previously described with the addition of the goal. However, HER also re-samples a set of different goals using a sampling strategy $\mathbb{S}$ and substitutes the original goal with a goal that was achieved during the execution of the current episode. Different sampling strategies are possible, but Andrychowicz et al. [2017] show that a strategy that samples a goal from the future of the trajectory performs well:

$$\mathbb{S}\left(\boldsymbol{s}_t, \boldsymbol{s}_{t+1}, \ldots, \boldsymbol{s}_{T-1}, \boldsymbol{s}_T\right) = m(\boldsymbol{s}_i), \qquad i \sim \mathcal{U}\left(\{t+1, \ldots, T\}\right), \tag{2.13}$$

where $\mathcal{U}(\cdot)$ denotes the uniform distribution over the specified set and $m : \mathcal{S} \mapsto \mathcal{G}$ is a function that converts an achieved state into the corresponding goal.

Since HER works with any off-policy algorithm, it can be used with DQN and DDPG. Algorithm 4 describes the entire algorithm in pseudo-code. For more details about sampling strategies and a more detailed description of the algorithm, please refer to Andrychowicz et al. [2017].

Algorithm 4 Hindsight Experience Replay [Andrychowicz et al., 2017]

Given:

- an off-policy RL algorithm $\mathbb{A}$, ▷ e.g. DQN or DDPG
- a strategy $\mathbb{S}$ for sampling goals for replay, ▷ e.g. Equation (2.13)
- a reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{G} \to \mathbb{R}$.

Initialize $\mathbb{A}$ ▷ e.g. initialize weights of neural networks
Initialize replay buffer $\mathcal{D}$
for episode=$1, \ldots, M$ **do**
  Sample a goal $\boldsymbol{g}$
  Receive initial state $\boldsymbol{s}_0$
  for $t = 0, \ldots, T-1$ **do**
    Select action $\boldsymbol{a}_t$ according to $\mathbb{A}$: $\boldsymbol{a}_t = \pi(\boldsymbol{s}_t || \boldsymbol{g})$ ▷ $||$ denotes concatenation
    Execute action $\boldsymbol{a}_t$ and observe next state $\boldsymbol{s}_{t+1}$
    Sample a mini-batch from $\mathcal{D}$ and perform optimization step of $\mathbb{A}$
  for $t = 0, \ldots, T-1$ **do**
    $r_t = r(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{g})$
    Store transition $(\boldsymbol{s}_t || \boldsymbol{g}, \boldsymbol{a}_t, \boldsymbol{s}_{t+1} || \boldsymbol{g}, r_t)$ in $\mathcal{D}$ ▷ standard experience replay
    Sample a set of additional goals for replay $G := \mathbb{S}(\textbf{current episode})$
    for $\boldsymbol{g}' \in G$ **do**
      $r' = r(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{g}')$
      Store transition $(\boldsymbol{s}_t || \boldsymbol{g}', \boldsymbol{a}_t, \boldsymbol{s}_{t+1} || \boldsymbol{g}', r')$ in $\mathcal{D}$ ▷ HER

3. Related Work

3.1 Overview

The problem of exploration in reinforcement has been studied extensively throughout the years. Its need arises since in typical reinforcement learning problems, no information about the MDP is available *a-priori*. Instead, the agent has to actively *explore* in order to discover behavior that leads to high returns. As learning progresses, the agent faces the *exploration vs. exploitation dilemma* [Kumar and Varaiya, 1986, Bertsekas, 1987, Thrun, 1992]. In essence, the agent has to decide whether it should exploit by using its current policy to select an action that it deems good to obtain a high return now or whether it should execute a different action and hope to discover new information that leads to higher returns in the future.

In general, one can differentiate between *undirected* and *directed* exploration [Thrun, 1992]. Examples for undirected exploration are ϵ-greedy exploration, Boltzmann exploration, and additive Gaussian noise, all of which will be defined and discussed in greater depth in Chapter 4. They all have in common that they utilize no exploration-specific knowledge and rely on adding some randomness into the action that the agent performs in order to achieve different outcomes, hence the name undirected. Directed exploration, on the other hand, utilizes such information in order to explore more efficiently. The name directed stems from the fact that these exploration schema typically actively seek out states that promise to maximize some gain. Such exploration can be count-based (e.g. how often has this action already been taken in this state), error-based (e.g. how big was the TD-error for this state-action tuple) or recency-based (e.g. when was the last time I took this action in this state) [Sutton and Barto, 1998]. This chapter provides just a brief summary and recent advances of some of these concepts. For a more in-depth discussion of the fundamentals of exploration, please refer to Thrun [1992] and Sutton and Barto [1998].

3.2 Near-Optimal Reinforcement Learning

A range of algorithms have been proposed that guarantee near-optimal reinforcement learning in polynomial time. Kearns and Singh [2002] prove that such an algorithm does indeed exist for general MDPs and show that it achieves near-optimal performance that is polynomial in the time horizon T and the number of states S. They obtain an algorithm, which they refer to as the *Explicit Explore or Exploit* (E^3) algorithm, that resolves the exploration vs. exploitation problem. Briefly speaking and slightly simplifying, E^3 divides

the state space into a set of *known* and *unknown* states. A state is considered known if model of the dynamics, which is learned concurrently, accurately predicts the transitions and rewards. During execution, the algorithm explores in unknown states by selecting the action that has been used least (referred to as *balanced wandering*). For known states, it exploits according to the current policy. While the resulting algorithm achieves near-optimal exploration, its practicality is still limited to very small state spaces, requires discrete action spaces, and adds additional complexity since a model of the dynamics has to be maintained.

An idea that is similar in spirit has been proposed by Brafman and Tennenholtz [2002], who refer to their algorithm as R_{max}. Similar to E^3, a model of the environment dynamics is maintained and exploration vs. exploitation is resolved according to it. They propose to initialize the model in optimistic fashion, i.e. for uncertain states assume a maximal reward (hence the name) and then optimize their policy according to this model. Due to this optimistic initialization, the agent is always driven towards states that are under-explored. As the agent covers more and more states, it improves its model with the true dynamics until it eventually is certain about a specific state and thus starts to exploit. This technique is commonly referred to as *optimism in the face of uncertainty* and Brafman and Tennenholtz [2002] prove that it achieves near-optimal return in polynomial time.

Kolter and Ng [2009] propose an algorithm that achieves near-Bayesian (and thus near-optimal) exploration in polynomial time. In Bayesian model-based reinforcement learning, one maintains a *believe state* about the model, which captures the uncertainty. In such a setting, exploration vs. exploitation can be achieved by considering the distribution over models and optimizing for the expected reward. While elegant, these methods are typically intractable since the dimension of the believe state grows polynomially in the number of states and actions. To overcome this, the authors propose to augment the reward function with an *Bayesian Exploration Bonus* (BOB) and always greedily select actions according to this objective. They show that their approach achieves near-Bayesian performance and results in lower sample-complexity than the aforementioned approaches.

3.3 Exploration in Deep Reinforcement Learning

As noted in the previous section, near-optimal reinforcement learning becomes increasingly difficult as the dimension of the state and action spaces increases. However, with the advent of deep learning and deep reinforcement learning [Mnih et al., 2015, Schulman et al., 2015a, Lillicrap et al., 2015], the state space is now typically extremely large or even infinite. The same is true for the action space, e.g. for continuous control problems, which are especially important in the field of robotics. This combinatorial explosion unfortunately often renders these algorithms impracticable for many modern applications, even if some form of discretization is applied to the state or action space.

To cope with this recent explosion in state and action space when learning directly from pixels, different approaches have been proposed. Tang et al. [2016] utilize hash functions to cleverly discretize the state space, which in turn allows them the use traditional count-based exploration. More concretely, they define a hash function $\phi : \mathcal{S} \mapsto \mathbb{Z}$ and use it to include a count-based exploration bonus in the reward function:

$$r^+(\boldsymbol{s}) = \frac{\beta}{\sqrt{n(\phi(\boldsymbol{s}))}}, \tag{3.1}$$

where β is a hyperparameter and $n(\cdot)$ is realized as a tabular counter that records the number of times the hashed version of state $\boldsymbol{s}$ was encountered (see also Strehl and Littman [2008]). The success of their method obviously hinges on the hash function ϕ, and the

authors investigate multiple candidates in their work. Bellemare et al. [2016] connect such count-based exploration and intrinsic motivation exploration. To do so, they propose the concept of a *pseudo-count*, which connects information-gain-as-learning-progress (some form of intrinsic motivation) and count-based exploration. More concretely, they use a density model, which, speaking in slightly simplifying terms, can be thought of as the conditional probability of encountering some state $\boldsymbol{s}$ after having observed the sequence of states $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_t$. The pseudo-count is then derived from this density model and used as an exploration bonus similarly to what was discussed before. This work was further extended by Ostrovski et al. [2017], who address some remaining open questions and use PixelCNN [van den Oord et al., 2016] as the density model of choice.

A different approach to exploration learns a dynamics model of the environment. If such a model exists, the error in such a model can again be used as a bonus in the reward function, i.e. the agent is incentivized to visit states that it cannot currently model accurately. Stadie et al. [2015] learn such a dynamics model using a deep neural network and assign exploration bonuses based on the squared prediction error of their learned model, which hinges on the assumption that the model produces accurate predictions for frequently visited state-action tuples and inaccurate ones otherwise. Houthooft et al. [2016] propose an exploration method that is based on the same idea but instead incentivize the agent to maximize the information gain about its beliefs of learned environment dynamics. Achiam and Sastry [2017] define a measure of surprise, again using a learned model of the environment's dynamics. Finally, Pathak et al. [2017b] formalize the important insight that a dynamics model should not capture everything that is observable by the agent. Instead, the agent should only be concerned with modeling the parts of the environment that it can actively influence or that have an influence on itself. To do so, the authors learn a model of both the forward (i.e. given state $\boldsymbol{s}_t$ and action $\boldsymbol{a}_t$, what is the next state $\boldsymbol{s}_{t+1}$) and inverse dynamics (i.e. given two subsequent states $\boldsymbol{s}_t$ and $\boldsymbol{s}_{t+1}$, what action $\boldsymbol{a}_t$ was performed) and cleverly connect them in an embedded representation of the state space. They show that an embedding trained in this fashion only captures information that is relevant to the agent. Like before, they use the error in prediction as measured in the embedded space to incentivize exploration by including it as a bonus in the reward function.

A common problem with the aforementioned approaches is that they are typically not universally applicable to all problems. They also often require additional models that need to be trained in parallel with the policy, adding computation and complexity overhead. As a result, we find that they are simply not used and simpler methods like ϵ-greedy and additive Gaussian noise are still omnipresent in most reinforcement learning literature. Another interesting observation is that these approaches usually augment the optimization objective (i.e. the reward function) by including some form of exploration bonus. This, in turn, means that they still rely on some form of undirected action space noise to discover novel states that yield high exploration bonuses.

A very different approach to exploration was proposed by Gal and Ghahramani [2016]. Instead of augmenting the optimization objective like previous approaches, they utilize dropout [Srivastava et al., 2014] and its connections to uncertainty estimation to drive exploration. More concretely, they train the Q-network of the DQN algorithm [Mnih et al., 2015] using dropout. When generating rollout data, they simply keep dropout enabled and use Thompson sampling [Thompson, 1933] to explore. In effect, this means that the agent samples a wide variety of actions if it has high uncertainty and converges towards just one action as the uncertainty decreases. Osband et al. [2016] propose a similar idea, *Bootstrapped DQN*, which has ties to the previously described uncertainty-based exploration. They use multiple Q-networks that are trained on different subsets of the observed experience data. When exploring, they pick one specific network for an entire episode and follow its predictions, thus achieving consistent exploration within episodes

but different behavior across episodes. Since Bootstrapped DQN is highly relevant for this work, we include an in-depth discussion of it in Section 2.3.2.

3.4 Parameter Perturbations

The idea of perturbing parameters of a neural network is related to *Evolutionary Strategies* (ES, Rechenberg and Eigen [1973], Schwefel [1977]) and especially *Neural Evolutionary Strategies* (NES, Sun et al. [2009a,b], Glasmachers et al. [2010a,b], Schaul et al. [2011], Wierstra et al. [2014]). In such approaches, noise in the parameters is typically used as a derivative-free approximation of the gradient. Until recently, it was believed that such methods are only applicable if the number of parameters is small and thus seemed intractable for deep neural network-based policies. However, Salimans et al. [2017] showed that this assumption actually turned out to be incorrect and that policies using deep neural networks could indeed be learned in such a fashion. They further noted that parameter perturbations seemed to lead to improved exploration behavior. Unfortunately, their proposed approach is vastly more sample-inefficient than the already quite sample inefficient state-of-the-art deep reinforcement learning algorithms. Still, their work is closely related to this thesis since we investigate parameter space noise not for the sake of approximating gradients but instead for its exploration properties.

In the context of policy gradient reinforcement learning, the idea of parameter perturbations has previously been explored by Rückstieß et al. [2008]. The authors show that perturbations often lead to improved exploration behavior if combined with the *REINFORCE* [Williams, 1992] and *Natural Actor-Critic* [Peters and Schaal, 2008] algorithms. However, the discussion of parameter noise is strictly limited to the on-policy policy gradient case and is only applied to extremely shallow policies with very small state spaces. This work was further extended by Kober and Peters [2008] and Sehnke et al. [2010].

Concurrently to the work on this thesis, Fortunato et al. [2017] have proposed an approach for exploration, which uses parameter noise and is conceptually similar to what we propose. In contrast to our approach, they directly learn the magnitude of the noise for each parameter. We believe that this may be problematic since it increases the number of parameters significantly. Additionally, by learning the scale of the parameter there is a real risk that the noise is simply disabled very quickly since, from the perspective of the agent, this noise hinds the agent at executing its optimal strategy. This is a common problem and is typically addressed by introducing some from of entropy bonus in the reward function [Mnih et al., 2016]. Unfortunately, this potential problem is not discussed in their work. They also do not include an analyses that specifically tests for the exploration properties of their algorithm. It is as such unclear how much their approach improves the actual exploration capabilities of the evaluated algorithms. Furthermore, their discussion is strictly limited to Atari and does not explore the effect of parameter noise on tasks with continuous action spaces.

4. Parameter Space Noise for Exploration

The exploration vs. exploitation dilemma is a long standing issue in reinforcement learning. Since the agent has no knowledge about its environment initially, it has to explore. Once the agent finds states of high reward, it should learn to exploit behaviors that lead towards these high rewards. However, it still remains important to explore since there may still be strategies that are more optimal that have not yet been discovered.

While a variety of sophisticated exploration mechanisms exist (as discussed in Chapter 3), in practice most reinforcement learning agents still rely on simple *action space noise* to realize exploration due to its conceptual simplicity as well as trivial implementation. In this thesis, we therefore propose a similarly simple yet more powerful exploration schema that we show is capable of exploration in settings where traditional action space noise fails. We call this exploration method *parameter space noise*.

During the work on this thesis, some parts have already been published by this author in heavily condensed form as a conference paper pre-print [Plappert et al., 2017].

4.1 Background and Formulation

Before describing parameter space noise, let us review typical action space noise. In the discrete case, ϵ-greedy exploration is typically used, which is defined as follows:

$$\widehat{\pi}(a_i \mid \boldsymbol{s}) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & \text{if } a_i = \pi(\boldsymbol{s}), \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise.} \end{cases} \tag{4.1}$$

In general, we denote by $\widehat{\pi}$ the *behavioral policy*, i.e. the policy that is used to generate rollouts during training and includes exploration. In other words, the optimal action according to π is selected with probability $1-\epsilon$, and a random action is uniformly selected with probability ϵ.

A slight variation of this formulation is Boltzmann or softmax exploration, which is defined as follows:

$$\widehat{\pi}(a_i \mid \boldsymbol{s}) = \frac{\exp Q(\boldsymbol{s}, a_i)}{\sum_j \exp Q(\boldsymbol{s}, a_j)}, \tag{4.2}$$

where $Q(\cdot, \cdot)$ is the action-value function. In contrast to ϵ-greedy exploration, actions are now selected with probability proportional to their values.

For the continuous case, noise is typically introduced in additive manner:

$$\widehat{\pi}(\boldsymbol{a} \mid \boldsymbol{s}) = \pi_\theta(\boldsymbol{a} \mid \boldsymbol{s}) + \mathcal{N}(\mathbf{0}, \sigma^2 \boldsymbol{I}). \tag{4.3}$$

Instead of additive Gaussian noise, other noise processes are sometimes used, e.g. the Ornstein–Uhlenbeck process [Lillicrap et al., 2015] to obtain noise that is temporarily correlated; however, the additive formulation remains the same.

While these approaches work, they all suffer from *inconsistent exploration*. To see why, consider a fixed state $\boldsymbol{s}$ in a trajectory. Whenever this fixed state $\boldsymbol{s}$ occurs, the actual action that the policy selects will be vastly different, due to the fact that the noise is *not* conditioned on the current state. This is especially apparent in Equation (4.3), where $\mathcal{N}(\mathbf{0}, \sigma^2 \boldsymbol{I})$ clearly has no dependence on $\boldsymbol{s}$ whatsoever. If the noise process is temporarily correlated, this problem becomes even more apparent. This thesis argues that this inconsistency in exploration behavior is problematic since it can cause oscillation. However, one cannot simply use a deterministic policy that always outputs the same action given the same state since the agent would then not explore at all.

The central idea of parameter space noise is to move the noise process from the actions to the parameters of a policy $\pi_{\boldsymbol{\theta}}$, where $\boldsymbol{\theta}$ is the parameter vector.[1] Instead of introducing noise in the actions, parameter space noise perturbs the parameters:

$$\widetilde{\boldsymbol{\theta}} = \boldsymbol{\theta} + \mathcal{N}(\mathbf{0}, \sigma^2 \boldsymbol{I}). \tag{4.4}$$

We sometimes denote this perturbed version of the policy as $\widetilde{\pi} := \pi_{\widetilde{\boldsymbol{\theta}}}$ and use it directly as the behavior policy $\widehat{\pi}(\boldsymbol{a} \mid \boldsymbol{s}) = \widetilde{\pi}(\boldsymbol{a} \mid \boldsymbol{s})$. Notice that the produced action is now fully conditioned on the state $\boldsymbol{s}$ through the perturbed behavior policy.

If this perturbation is not performed at every step, it should be clear that parameter space noise does indeed explore consistently, that is given a fixed state $\boldsymbol{s}$, the produced action will be the same.[2] For example, if the parameters are perturbed at the beginning of each episode, we achieve consistent exploration *within* that episode but still exhibit vastly different behavior (due to the re-perturbation at the beginning) *across* episodes. Similar observations have been pointed out by Rückstieß et al. [2008] and Osband et al. [2016].

Figure 4.1 summarizes the difference between traditional action space noise exploration and our newly proposed parameter space noise exploration.

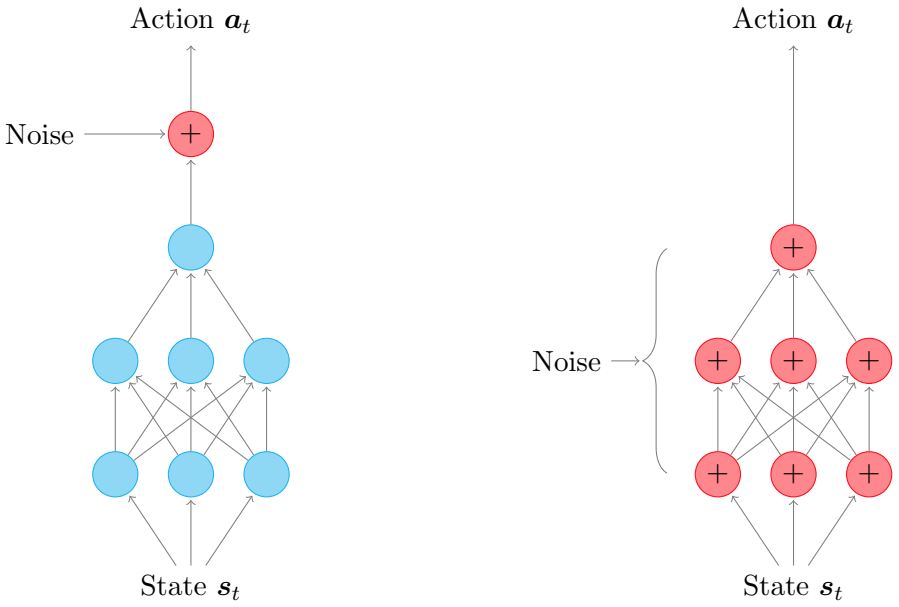
While the idea of parameter space noise is very simple (compare Equation (4.4)), applying it to deep neural networks is not straightforward for the following two reasons.

- Different layers within the policy network may exhibit very different sensitivity to perturbations.
- In contrast to action space noise, the effect of a perturbation in parameter space cannot intuitively be understood and it is thus hard to reason about it. In particular, selecting an appropriate scale σ is hard and may even have to vary with time.

Section 4.2 addresses the first problem and Section 4.3 the second one.

[1] In this work, all policies are always represented by (deep) neural networks, but this should only be considered an implementation detail since the approach can be generalized to all functions that are somehow parameterized.

[2] This is true for the deterministic case. For the probabilistic case, the distribution will be the same *and* is fully conditioned on the current state.



(a) *Action space noise* (b) *Parameter space noise*

Figure 4.1: *Comparison between action space noise (left) and parameter space noise (right). Depicted in red are the parts that are subject to noise. The action is produced by a parameterized policy, in this case a neural network.*

4.2 Perturbing Deep Neural Networks

It is not immediately obvious that deep neural networks, with potentially millions of parameters and complicated non-linear interactions, can be perturbed in meaningful ways by applying spherical Gaussian noise. A similar problem has recently been addressed by Salimans et al. [2017], who also perturb the weights of a deep neural network in order to approximate the gradient for evolution strategies-based optimization (i.e. a derivative-free approach). They utilize *(virtual) batch normalization* [Ioffe and Szegedy, 2015, Salimans et al., 2016] to re-parameterize the network such that different layers within the network have similar sensitivities to perturbations.

In this work, we follow a similar approach but replace batch normalization with the conceptually much simpler *layer normalization* [Ba et al., 2016]. More concretely, we apply layer normalization to each fully-connected layer:

$$\boldsymbol{h} = f\left[\frac{\boldsymbol{g}}{\sigma} \odot (\boldsymbol{a} - \mu) + \boldsymbol{b}\right], \tag{4.5}$$

where $f[\cdot]$ denotes the non-linearity (e.g. a ReLU), which is applied element-wise, $\odot$ denotes the Hadamard product, and $\boldsymbol{a} = \boldsymbol{W}\boldsymbol{x}$ is the projection of the input $\boldsymbol{x}$ using the weights $\boldsymbol{W}$ of the fully-connected layer. $\boldsymbol{g}$ and $\boldsymbol{b}$ are the gain and bias and are learnable parameters that we do not perturb. σ and μ are statistics that are based on the activations $\boldsymbol{a}$ (a vector of dimension H) before applying the non-linearity:

$$\mu = \frac{1}{H}\sum_{i=1}^{H} a_i \qquad \sigma = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i - \mu)^2}, \tag{4.6}$$

i.e. the empirical mean and standard deviation of all activations within that layer.

In contrast to (virtual) batch normalization, layer normalization has the advantage of being independent of the current batch, since statistics are computed not across batches

but within layers. This makes it especially attractive for applications in reinforcement learning where the distribution of the samples is expected to shift as more and more of the environment is discovered.[3]

Usually, normalization schemes like batch normalization and layer normalization are used to stabilize training since the output of a layer has approximately zero mean and unit variance. While this is of course useful, the role of normalization has a second, more important angle in the context of perturbation. Assume that we perturb the weights of a layer, i.e. $\widetilde{\boldsymbol{W}} = \boldsymbol{W} + \mathcal{N}(\boldsymbol{0}, \sigma^2 \boldsymbol{I})$.[4] Since the statistics are now computed based on $\widetilde{\boldsymbol{a}} = \widetilde{\boldsymbol{W}}\boldsymbol{x}$, we can guarantee that arbitrary perturbations still result in an output of that layer that has approximately zero mean and unit variance. This is important to ensure that upstream (i.e. close to the input of the network) perturbations do not cause problems in downstream layers. Just like Salimans et al. [2017], we find that such a re-parameterization is crucial for our method to work since it allows use to use the same perturbation scale across all layers.

4.3 Adaptive Scaling

While Section 4.2 was concerned with allowing us to have spherical Gaussian noise that is applied equally to all layers, this section is concerned with determining an appropriate value for the scalar σ^2. This problem is actually quite non-trivial since it is hard to intuitively understand the parameter. Consider additive, spherical Gaussian noise in the *action space*. In this case, we can understand the effect that such a perturbation has quite easily since we typically understand the action space of an environment. The same is true for the ϵ parameter of ϵ-greedy exploration. On the other hand, a deep neural network routinely has hundreds of thousands or millions of parameters and we cannot possibly hope to understand what effects a perturbation with Gaussian noise of, say, $\mathcal{N}(\boldsymbol{0}, 0.02 \cdot \boldsymbol{I})$ causes. Even worse, the sensitivity to perturbations is likely going to change as we train the network since we move away from a random initialization around zero towards a quite complicated and non-linear mapping from states to (hopefully) optimal actions.

Our proposed solution to this problem is to move the problem from the *parameter space* to the *action space*. This trick allows us to measure the effect of a parameter perturbation in a space that we typically understand. Conversely, since we now understand the effect of the perturbation, we can close the loop and adapt the scale of the parameter perturbation accordingly.

This idea can be formalized as follows:

$$\sigma_{k+1} = \begin{cases} \alpha\sigma_k & \text{if } d\left(\pi(\boldsymbol{a} \mid \boldsymbol{s}), \widetilde{\pi}(\boldsymbol{a} \mid \boldsymbol{s})\right) \leq \delta, \\ \frac{1}{\alpha}\sigma_k & \text{otherwise.} \end{cases} \tag{4.7}$$

Here, $d(\cdot, \cdot)$ defines some distance measure between the perturbed policy $\widetilde{\pi}$ and the non-perturbed policy π, $\delta \in \mathbb{R}_{>0}$ is a positive threshold value, and $\alpha \in \mathbb{R}_{>0}$ is a positive parameters that is used to adapt the current value σ_k.[5] In other words, we increase the scale of parameter perturbations if the effect of the current scale as measured in action space is below a desired threshold and decrease it otherwise. For example, if $\alpha = 1.01$,

[3] Notice that this observation is true even if the the distribution of inputs (i.e. the states) is fixed. However, from the perspective of the agent, it constantly changes as it explores and discovers previously unseen states.

[4] This slight abuse of notation should be read as flattening W into a vector first, adding Gaussian noise to each element, and then re-shaping the vector into a matrix again.

[5] Instead of this very simple linear scaling, more complicated adaption mechanisms could of course be used instead. However, we leave this part to future work.

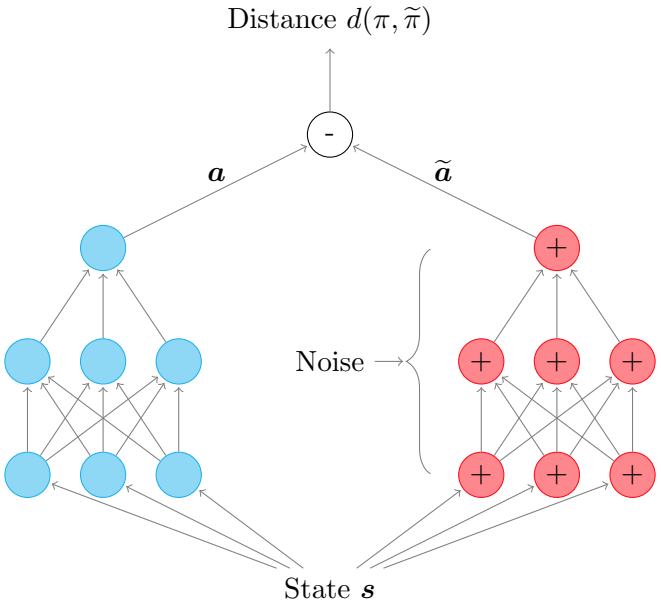


Figure 4.2: *Distance estimation between unperturbed policy π (left) and perturbed policy $\widetilde{\pi}$ (right), which produce $\boldsymbol{a}$ and $\widetilde{\boldsymbol{a}}$, respectively, using the same input $\boldsymbol{s}$. The distance $d(\pi, \widetilde{\pi})$ is then computed using these actions.*

the scale is increased by 1% whenever the desired threshold has not been achieved and is decreased by 1% otherwise. Also note that this is an instance of the Levenberg-Marquardt heuristic [Ranganathan, 2004].

In practice, the distance between $\widetilde{\pi}$ and π cannot be computed. Instead, we estimate the expected distance using a randomly sampled mini-batch $\{\boldsymbol{s}^{(1)}, \ldots, \boldsymbol{s}^{(M)}\}$ of states:

$$\mathbb{E}_{\boldsymbol{s}}\left[d(\widetilde{\pi}, \pi)\right] \approx \frac{1}{M} \sum_{i=1}^{M} d\left(\pi\left(\boldsymbol{a}^{(i)} \mid \boldsymbol{s}^{(i)}\right), \widetilde{\pi}\left(\boldsymbol{a}^{(i)} \mid \boldsymbol{s}^{(i)}\right)\right). \tag{4.8}$$

This is clearly tractable and computationally cheap since it requires only two forward passes for each sample. Figure 4.2 illustrates the distance computation $d(\pi, \widetilde{\pi})$ graphically. The entire adaptive scaling approach is also summarized in Algorithm 5.

However, there is one last missing piece: the distance measure $d(\cdot, \cdot)$. In principle, there is no single choice for $d(\cdot, \cdot)$ and it depends on the algorithm at hand as well as the desired properties that one wishes to achieve with it. The proposed adaption approach should thus be considered a framework in which the distance function is a concrete design choice. In the following two subsections, we describe a distance measure for DQN and DDPG, respectively, thus covering both discrete and continuous action spaces with deterministic policies. Furthermore, Plappert et al. [2017] describe a distance measure for TRPO [Schulman et al., 2015b], which uses the Kullback-Leibler divergence and is thus an instance of the probabilistic case, which we do not re-iterate here in detail.

4.3.1 A Distance Measure for DQN

For DQN, the policy is defined implicitly by the Q-function. Unfortunately, this means that a naïve distance measure between Q and $\widetilde{Q}$ has pitfalls. To see why, consider a policy where the perturbation is a constant offset to the bias of the final layer. In this case, the Q-values for the perturbed policy can be expressed as $\widetilde{Q}(\boldsymbol{s}, a_i) = Q(\boldsymbol{s}, a_i) + (\widetilde{b}_i - b_i) = Q(\boldsymbol{s}, a_i) + c$, where $\boldsymbol{b}$ denotes the bias of the unperturbed Q-function and $\widetilde{\boldsymbol{b}} = \boldsymbol{b} + c$ denotes the perturbed

Algorithm 5 Parameter Space Noise Exploration with Adaptive Scaling

Given:

- an RL algorithm $\mathbb{A}$, ▷ e.g. DQN or DDPG
- an initial parameter space noise scale $\sigma_0 \in \mathbb{R}_{>0}$,
- a distance measure $d(\cdot, \cdot)$,
- an adaption scalar and threshold $\alpha, \delta \in \mathbb{R}_{>0}$,
- intervals for training and adaptive scaling $T_{\text{train}}, T_{\text{adapt}} \in \mathbb{R}_{>0}$

Initialize $\mathbb{A}$ and in particular $\pi = \pi_\theta$
Initialize $\widetilde{\pi} = \pi_{\widetilde{\theta}}$ for exploration
Initialize $\bar{\pi} = \pi_{\bar{\theta}}$ for noise adaption
for episode $= 0, \ldots, M-1$ **do**
  Perturb $\widetilde{\theta} \leftarrow \theta + \mathcal{N}(0, \sigma_k^2 \boldsymbol{I})$ and obtain $\widetilde{\pi} = \pi_{\widetilde{\theta}}$
  for $t = 0, \ldots, T-1$ **do**
    Sample an action $\boldsymbol{a}_t$ using the perturbed policy $\widetilde{\pi}$
    Execute action $\boldsymbol{a}_t$ and observe a new state $\boldsymbol{s}_{t+1}$
    if $t \bmod T_{\text{train}} = 0$ **then**
      Execute training step of $\mathbb{A}$
    if $t \bmod T_{\text{adapt}} = 0$ **then**
      Perturb $\bar{\theta} \leftarrow \theta + \mathcal{N}(0, \sigma_k^2 \boldsymbol{I})$ and obtain $\bar{\pi} = \pi_{\bar{\theta}}$
      Estimate $d_k \leftarrow \mathbb{E}_{\boldsymbol{s}}\left[d\left(\pi(\boldsymbol{a} \mid \boldsymbol{s}), \bar{\pi}(\boldsymbol{a} \mid \boldsymbol{s})\right)\right]$ ▷ see Section 4.3.1 and Section 4.3.2
      Adapt $\sigma_{k+1} \leftarrow \begin{cases} \alpha\sigma_k & \text{if } d_k \leq \delta, \\ \frac{1}{\alpha}\sigma_k & \text{otherwise} \end{cases}$
      $k \leftarrow k + 1$

bias. Thus, if we would use a natural but naïve distance measure like the Euclidean norm $\|Q - \widetilde{Q}\|$, the distance between Q and $\widetilde{Q}$ would clearly be non-zero.[6] However, since the deterministic policy is defined as $\pi(\boldsymbol{s}) := \operatorname{argmax}_a Q(\boldsymbol{s}, a)$, both π and $\widetilde{\pi}$ are exactly the same (since the argmax operation is invariant to the constant offset). Thus a distance measure between π and $\widetilde{\pi}$ should be zero, which is not the case for $\|Q - \widetilde{Q}\|$.

Our proposed distance measure is thus based on the probabilistic Boltzmann policy (compare Equation (4.2)) for both π and its perturbed version $\widetilde{\pi}$. Since both are probability distributions, we can measure the distance using the the Kullback-Leibler (KL) divergence:[7]

$$d(\pi, \widetilde{\pi}) := D_{\text{KL}}(\pi \parallel \widetilde{\pi}) = \sum_{i=1}^{|\mathcal{A}|} \pi(a_i \mid \boldsymbol{s}) \log \frac{\pi(a_i \mid \boldsymbol{s})}{\widetilde{\pi}(a_i \mid \boldsymbol{s})}. \tag{4.9}$$

We note that this formulation is only used to define a well-behaved distance measure, and the deterministic policy is still used for rollouts.

There are two advantages to this formulation. First, the distance measure is now invariant to constant offsets to the Q-function as described previously due to the normalization term of the Boltzmann distribution. Second, we can relate this distance measure to the commonly used ϵ-greedy action space noise since both are probability distributions. This is convenient since we avoid the need for another hyperparameter and can instead simply scale the parameter space noise such that the effect in action space as measured by the

[6] In fact it would be $\|Q - \widetilde{Q}\| = \|[c, \ldots, c]^T\| = \sqrt{\sum_{i=1}^{|\mathcal{A}|} c^2} = \sqrt{|\mathcal{A}|c^2} = \sqrt{|\mathcal{A}|}c$ and thus greater than zero if $c > 0$.

[7] Strictly speaking, the KL divergence is not a distance measure since $D_{\text{KL}}(\pi \parallel \widetilde{\pi}) \neq D_{\text{KL}}(\widetilde{\pi} \parallel \pi)$.

KL divergence is similar to that of ϵ-greedy exploration.[8] More concretely, the KL divergence between the deterministic policy π and $\widehat{\pi}$, where $\widehat{\pi}$ denotes ϵ-greedy exploration (see Equation (4.1)), is defined as follows:

$$D_{\mathrm{KL}}(\pi \parallel \widehat{\pi}) = \sum_{i=1}^{|\mathcal{A}|} \pi(a_i \mid \boldsymbol{s}) \log \frac{\pi(a_i \mid \boldsymbol{s})}{\widehat{\pi}(a_i \mid \boldsymbol{s})} \tag{4.10}$$

$$= -\log\left(1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|}\right). \tag{4.11}$$

Thus, by setting $\delta := -\log\left(1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|}\right)$ and using the KL divergence as defined in Equation (4.9) in our adaptive scaling approach, we achieve parameter space noise that is comparable in magnitude to ϵ-greedy action space noise.

4.3.2 A Distance Measure for DDPG

For DDPG, we can measure the effect of parameter space noise in action space noise directly since DDPG defines an explicit policy network π. More concretely, we use the following distance measure:

$$d(\pi, \widetilde{\pi}) = \sqrt{\frac{1}{|\mathcal{A}|} \sum_{i=1}^{|\mathcal{A}|} \mathbb{E}_{\boldsymbol{s}} \left[\left(\pi(\boldsymbol{s})_i - \widetilde{\pi}(\boldsymbol{s})_i \right)^2 \right]}, \tag{4.12}$$

where $\mathbb{E}_{\boldsymbol{s}}[\cdot]$ is estimated using a mini-batch of states from the replay buffer, $|\mathcal{A}|$ denotes the number of continuous actions, and $\pi(\boldsymbol{s})_i$ denotes the i-th action selected by the deterministic policy π. In other words, we estimate the standard deviation between perturbed and non-perturbed policy.

The interpretation as the standard deviation is again convenient since action space noise in the continuous case is usually realized as additive Gaussian noise (see Equation (4.3)). This type of noise is quantified by its scale, the standard deviation σ. Similar to the DQN case, we can thus relate parameter space noise and action space noise through this measure. Thus, by setting $\delta := \sigma$ and using the distance measure from Equation (4.12), we can ensure that the magnitude of the perturbation induced by parameter space noise is comparable to additive Gaussian action space noise.

[8] Although the magnitude of noise in action space is similar, the structure and thus behavior of the perturbed policy is still significantly different since the behavior is now fully conditioned on the state.

5. Experiments

In this chapter, we evaluate parameter space noise in a large variety of settings. We start by considering a simple toy experiment (Section 5.1) that specifically tests for the exploration capabilities of a reinforcement learning algorithm. In Section 5.2, we evaluate the performance of DQN with parameter space noise on a set of 21 challenging Atari games with discrete action spaces. We then move to domains that require continuous control in Section 5.3. In this section, we evaluate the performance of parameter space noise on a set of 7 continuous control problems that are part of OpenAI Gym Brockman et al. [2016] and are considered a standard benchmark. To specifically test for exploration, we also consider modified environments that exhibit a sparse reward structure. We conclude the experiments chapter by testing the capabilities of parameter space noise on a set of four real-world robotics problems in Section 5.4. In this section, we also compare parameter space noise against Hindsight Experience Replay (HER, Andrychowicz et al. [2017]) and investigate if parameter space noise and HER can be combined to achieve start-of-the-art exploration on these challenging robotics tasks.

5.1 A First Toy Problem

To explore the properties of parameter space noise, we start our experimental evaluation using a well-known toy problem. The environment at hand consists of N states, where state s_n is connected to state s_{n-1} to its left and state s_{n+1} to its right, thus building a chain of length N, with $N > 3$. The leftmost state s_1 and rightmost state s_N are connected to itself, respectively. In each state, the agent can select between two discrete actions: left and right. State s_1 yields a reward of $r(s_1) = 0.001$ and state s_N a reward of $r(s_N) = 1$. The agent always starts in state s_2 and can perform $N + 9$ interactions with the environment before it resets. Figure 5.1 depicts this environment.

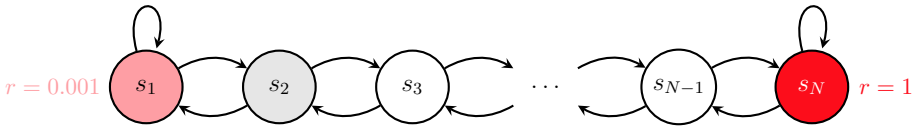


Figure 5.1: *Simple and scalable environment to test for exploratory behavior [Osband et al., 2016]. State s_2 is the initial state and states s_1 and s_N yield the indicated rewards. Arrows represent possible actions ("left" and "right").*

The optimal strategy is to always select the "right" action since the high reward in state s_N yields a higher return.[1] However, finding this reward becomes increasingly more difficult as the length of the chain increases since, from the perspective of the agent, going to the right yields zero rewards until it eventually finds state s_N. In contrast, it is extremely likely that it will find the reward of state s_1 immediately since it requires only a single "left" action, which can easily lead to converging onto a sub-optimal strategy. This environment thus specifically tests for exploration, where the agent has to select seemingly sub-optimal actions for many steps to eventually discover the high reward state s_N.

5.1.1 Experimental Setup

We follow the state encoding proposed by Osband et al. [2016] and use $\phi(s_n) = (\mathbb{1}\{x \leq s_n\})$ as the observation, where $\mathbb{1}$ denotes the indicator function. DQN is used with a very simple network to approximate the Q-value function that consists of 2 hidden layers with 16 ReLU units. Layer normalization [Ba et al., 2016] is used for all hidden layers before applying the non-linearity. Each agent is trained and evaluated for up to 2 000 episodes: After each training episode, the current performance is evaluated by disabling any exploration noise and performing a single evaluation rollout.

We compare our proposed adaptive parameter space noise DQN against Bootstrapped DQN [Osband et al., 2016] with $K = 20$ heads and Bernoulli masking with $p = 0.5$ as well as ϵ-greedy DQN where we anneal ϵ linearly from 1 to 0.1 over the first 100 episodes. For parameter space noise exploration, we use the previously describe distance measure (see Section 4.3.1) and scale σ such that $\delta \approx 0.05$. For all evaluated DQN variants, we set $\gamma = 0.999$, use a replay buffer of size 100 000 and update the target network every 100 timesteps. We use the Adam optimizer [Kingma and Ba, 2014] with a learning rate of 10^{-3} and a batch size of 32. Training starts after the first five episodes to ensure that the replay buffer holds a sufficient amount of transitions. As mentioned before, we display exploration during the evaluation rollouts, which is straightforward for parameter space noise and ϵ-greedy exploration. For Bootstrapped DQN, we perform majority voting over all heads during evaluation.

To better understand how each exploration mechanism behaves, we vary the length of the chain N. We also repeat this experiment using three different random seeds and report the median number of episodes after which the environment is considered solved. We consider the problem solved if a policy achieves the optimal return in 100 subsequent evaluation rollouts.[2] If an agent fails to solve the environment for a a specific value of N after 2 000 episodes, we considered it unsolved. We evaluate the performance for chain lengths $N = 4, 5, \ldots, 100$.

5.1.2 Results

Figure 5.2 shows the results for DQN with parameter space noise exploration (ours), Bootstrapped DQN and DQN with ϵ-greedy action space exploration. Depicted are the number of episodes before the problem is solved for varying chain lengths. ϵ-greedy exploration fails even for relatively short chain lengths and does never discover the optimal strategy for $N > 20$. This is because ϵ-greedy explores inconsistently, that is given the same state s_n, it will likely select different actions. Bootstrapped DQN performs much better and finds

[1] This holds for moderately large values of N until eventually the chain becomes so long that the return to the left becomes larger due to the fact that we can stay in this low reward state for many steps. However, we only consider chains up to length 100, so this is not a concern in our experiments.

[2] This means that even if the policy would immediately be optimal after the first episode, we would run the algorithm for at least 99 more episodes. We can therefore not expect to do better than solving the environment after 100 episodes under this evaluation schema.

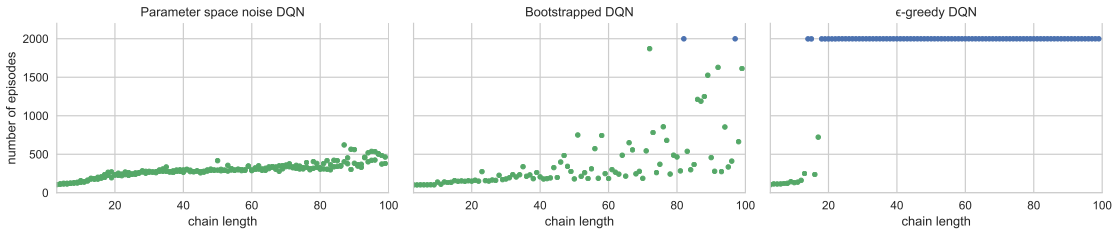


Figure 5.2: *Median number of episodes before considered solved for DQN with different exploration strategies. Green indicates that the problem was solved whereas blue indicates that no solution was found within the* 2 000 *episodes limit. Less episodes before solved is better.*

solutions for almost all chain lengths. In contrast to ϵ-greedy exploration, Bootstrapped DQN commits to a single head for each episode, which means that it will consistently explore within this episode. However, as the chain length increases, Bootstrapped DQN also starts to struggle and the number of episodes it requires to solve the problem increases significantly. Finally, our proposed parameter space noise exploration strategy consistently finds the optimal strategy even for large N and clearly outperforms Bootstrapped DQN while being conceptually simpler and less computationally expensive. Like Bootstrapped DQN, we also explore consistently since we perturb the network only at the beginning of each episode. However, our network does not have multiple heads that need to be trained, which we believe is the main factor that makes parameter space noise learn the problem more quickly.

5.2 Arcade Learning Environment Experiments

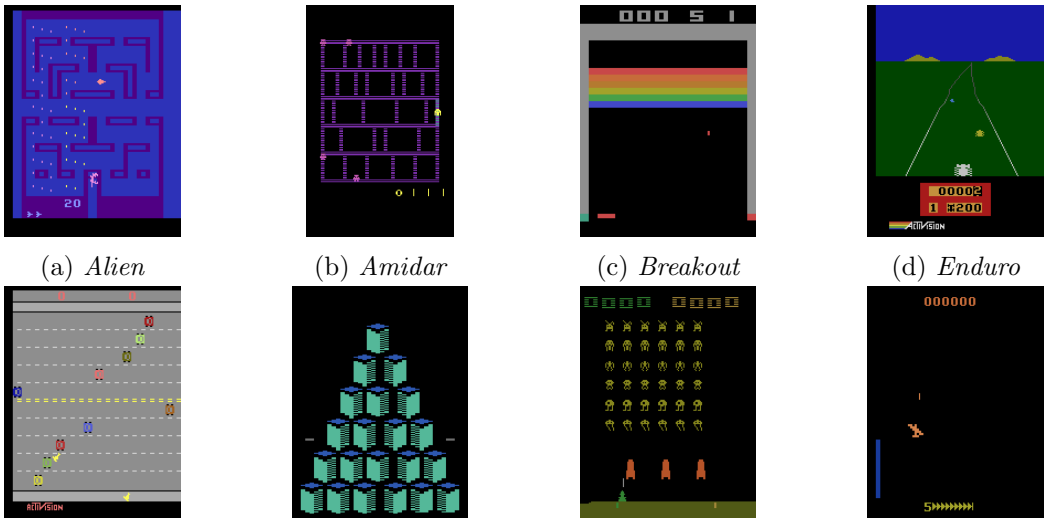
While Section 5.1 shows that parameter space noise exhibits exploration behavior that is significantly different from typical action space noise exploration, the chain problem is relatively simple due to its low state and action dimensionality as well as trivial optimal policy. In this section, we therefore explore how parameter space noise behaves in much more complicated environments if we combine it with state of the art reinforcement learning algorithms like DQN.

5.2.1 Environment

The Arcade Learning Environment (ALE, Bellemare et al. [2013]) is an Atari 2600 simulator that is widely used for the evaluation of reinforcement learning algorithms. These games offer a challenging set of problems since the agent has to learn directly from pixels, making the state space extremely large. Furthermore, a great variety of games exist that have vastly different game mechanics and often require non-trivial exploration. In all cases, the action space is discrete and its dimension may vary from game to game. Figure 5.3 depicts some exemplary games.

Concretely, we use the following 21 ALE environments. This selection is based on the taxonomy presented by Bellemare et al. [2016], and contains a mix of games with easy exploration and hard exploration with both dense and sparse rewards.

- **Alien** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,18\}$)
- **Amidar** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,10\}$)
- **BankHeist** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,18\}$)
- **BeamRider** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,9\}$)



(a) *Alien* (b) *Amidar* (c) *Breakout* (d) *Enduro*

(e) *Freeway* (f) *Qbert* (g) *SpaceInvaders* (h) *Zaxxon*

Figure 5.3: *Eight exemplary Atari games simulated using ALE and OpenAI Gym. In all cases, the agent learns directly from pixels.*

- **Breakout** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,4\}$)
- **Enduro** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,9\}$)
- **Freeway** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,3\}$)
- **Frostbite** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,18\}$)
- **Gravitar** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,18\}$)
- **MontezumaRevenge** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,18\}$)
- **Pitfall** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,18\}$)
- **Pong** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,6\}$)
- **PrivateEye** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,18\}$)
- **Qbert** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,6\}$)
- **Seaquest** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,18\}$)
- **Solaris** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,18\}$)
- **SpaceInvaders** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,6\}$)
- **Tutankham** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,8\}$)
- **Venture** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,18\}$)
- **WizardOfWor** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,10\}$)
- **Zaxxon** ($\mathcal{S} \subset [0,1]^{4\times84\times84}$, $\mathcal{A} = \{1,\dots,18\}$)

We utilize OpenAI Gym [Brockman et al., 2016] to interface with the ALE simulator since it provides a common and easy to use Python interface. OpenAI Gym is open-source and publicly available.[3] We further apply the preprocessing that DQN implementations typically use. We down-scale each frame to 84×84 pixels to make the computation of the convolutions less expensive. Furthermore, we convert all frames into grayscale and

[3] `https://github.com/openai/gym`

concatenate 4 subsequent frames into a single observation. This is necessary since the games are typically not fully observable from a single frame since, for example, the velocity of an object could not be inferred. Finally, we normalize the pixel values to be in in $[0, 1]$. The observation is thus $\boldsymbol{s} \in [0, 1]^{4\times84\times84}$, where each frame is considered a channel of the resulting "image". We further clip rewards such that $r \in [-1, 1]$. This is common practice and ensures that rewards are on comparable scales across different games. We also perform up to 30 noop actions at the beginning of each game to effectively randomize the initial state across different runs. During training, we further consider a state as terminal as soon as the agent loses a single life. This effectively reduces the time horizon and thus makes the estimation of Q a bit simpler. Lastly, we employ frame skip of length 4 that repeats the current action 4 times before querying the policy for a new action. All of the aforementioned changes to the environment are what Mnih et al. [2015] describe and we follow their setup as close as possible to obtain comparable results. The preprocessing code is available online.[4]

5.2.2 Experimental Setup

We use DQN for ALE and follow the experimental setup of Mnih et al. [2015] as closely as possible. More concretely, we use a Q-network that consists of 3 convolutional layers (32 filters of size 8×8 and stride 4, 64 filters of size 4×4 and stride 2, 64 filters of size 3×3 and stride 1) followed by 1 hidden layer with 512 units followed by a linear output layer with one unit for each action. ReLUs are used in each layer, while layer normalization [Ba et al., 2016] is used in the fully connected part of the network. Layer normalization is not used in the original paper but we require it since we wish to perturb the Q-network. To allow for a fair comparison, we include it in cases since we have found that it does not hurt performance for the baseline case. The target networks are updated every 10 000 steps. The Q-network is trained using the Adam optimizer [Kingma and Ba, 2014] with a learning rate of 10^{-4} and a batch size of 32. The replay buffer can hold 1 000 000 state transitions. For the ϵ-greedy baseline, we linearly anneal ϵ from 1 to 0.1 over the first 1 000 000 steps. For parameter space noise, we adaptively scale the noise to have a similar effect in action space (see Section 4.3.1 for details), effectively ensuring that the maximum KL divergence between perturbed and non-perturbed π is softly enforced. This also ensures a fair comparison between ϵ-greedy exploration and our proposed parameter space exploration since the magnitude in resulting noise as measured in action space is comparable. The policy is perturbed at the beginning of each episode and the standard deviation is adapted as described in Section 4.3 every 50 steps. We only perturb the fully connected part of the network. This is because the convolutional part of the network is concerned with extracting features from the input images. For that reason, it would not make sense to perturb this part since the decision making for choosing an action is done in the fully connected part of the network. To avoid getting stuck (which can potentially happen for a perturbed policy), we also use ϵ-greedy action selection with $\epsilon = 0.01$. In all cases, we perform 50 000 random actions to collect initial data for the replay buffer before training starts. We set $\gamma = 0.99$ and clip gradients for the output layer of Q to be within $[-1, 1]$. Table 5.1 summarizes all hyperparameters.

We further found that for parameter space noise, it is beneficial to include a second, separate policy head in the standard DQN network architecture. More concretely, we share the convolutional part between both heads, followed by the fully connected layers for the Q estimation (as per usual) plus another head with the same architecture for the policy. The final output layer of the policy head uses a softmax activation to produce a probability distribution over actions (in contrast to the Q-head which produces a scalar per action). We train the Q-head as per standard DQN practices. For the policy head, we

[4] `https://github.com/openai/baselines/blob/master/baselines/common/atari_wrappers.py`

Table 5.1: *Hyperparameters for DQN experiments.*

Hyperparameter	Value
Target network update interval	10 000
Optimizer	Adam [Kingma and Ba, 2014]
Batch size	32
Learning rate	10^{-4}
Replay buffer size	40^6
Warm-up steps	50 000
Training steps	10^6
Discount factor γ	0.99
Reward normalization	$r \in [-1, 1]$
Gradient clipping	Gradient in $[-1, 1]$ for output layer
Network architecture	Mnih et al. [2015]
ϵ-greedy annealing	Linear over 10^6 steps
ϵ-greedy minimum value	0.1
Parameter noise adaption interval (T_{adapt})	50
Parameter noise distance measure	Section 4.3.1

simply freeze the convolutional layers and train the head to maximize the probability of the best action as defined by the Q-head (i.e. with label $y = \text{argmax}_a Q(\boldsymbol{s}, a)$) by minimizing the cross entropy between the label and the prediction. In effect, this causes the policy network head to simply track the behavior of the Q-network head, but we found that perturbations to the policy head result in richer exploration behavior than perturbing Q directly. During rollouts, we thus use this separate policy head (or its perturbed version). To ensure that this architecture change does not meaningfully alter the behavior of ϵ-greedy exploration, we include a version with separate policy network head but regular ϵ-greedy exploration as a control in our experiments.

An open-source implementation of adaptive parameter space noise with DQN is available online.[5]

5.2.3 Results

The learning curves for all aforementioned 21 ALE environments are depicted in Figure 5.4. To better estimate the overall performance of each strategy, we run each experiment with 3 random seeds and present the median return (line) as well as the interquartile range (shaded area) for each configuration. Also notice that we depict the performance of the behavior policy, i.e. including noise, since we are especially interested in its behavior.

On Enduro, which requires a player to continuously press a button in order to accelerate a racing car, parameter space noise finds this strategy significantly earlier than ϵ-greedy exploration and thus achieves higher scores. The same holds true for Freeway, which requires an agent to cross a street crowded with cars, which again requires some consistency to successfully obtain high scores. However, parameter space noise also achieves higher scores on games with dense rewards like Breakout during training. In Breakout, the agent has to move a paddle at the bottom of the screen such that a ball is hit against a row of bricks. We believe that action space noise causes the paddle to jitter, resulting the agent to drop balls more easily compared to parameter space noise exploration. We also note that there are some games like Montezuma's Revenge that neither action space noise nor

[5] `https://github.com/openai/baselines/tree/master/baselines/deepq`

parameter space noise are able to learn. These games are considered to be extremely hard since an agent needs to complete a few high-level steps like collecting a key, finding the relevant door, and finally opening it before receiving any reward. While we believe that these games will require more sophisticated exploration methods, parameter space noise will likely be useful to learn the low-level steps to achieve each of the high-level goals.

Overall, our results show that parameter space noise exploration typically achieves much higher scores during training due to improved exploration over ϵ-greedy action space noise exploration. We find that DQN with parameter space noise clearly outperforms ϵ-greedy on 8 out of the 21 games used in our experiments. On the remaining games, parameter space noise exhibits comparable performance, such that we can conclude that parameter space noise either improves performance or, in the worst case, performs at least similarly to action space noise.

Table 5.2: *Performance comparison between Evolution Strategies (ES) as reported in Salimans et al. [2017], DQN with ϵ-greedy, and DQN with parameter space noise (proposed method). ES was trained on* 1 000 M*, while DQN was trained on only* 40 M *frames.*

Game	ES	DQN with ϵ-greedy	DQN with parameter noise
Alien	994.0	1535.0	**2070.0**
Amidar	112.0	281.0	**403.5**
BankHeist	225.0	510.0	**805.0**
BeamRider	744.0	**8184.0**	7884.0
Breakout	9.5	**406.0**	390.5
Enduro	95.0	1094	**1672.5**
Freeway	31.0	**32.0**	31.5
Frostbite	370.0	250.0	**1310.0**
Gravitar	**805.0**	300.0	250.0
MontezumaRevenge	**0.0**	**0.0**	**0.0**
Pitfall	**0.0**	-73.0	-100.0
Pong	**21.0**	**21.0**	20.0
PrivateEye	100.0	**133.0**	100.0
Qbert	147.5	**7625.0**	7525.0
Seaquest	1390.0	8335.0	**8920.0**
Solaris	**2090.0**	720.0	400.0
SpaceInvaders	678.5	1000.0	**1205.0**
Tutankham	130.3	109.5	**181.0**
Venture	**760.0**	0	0
WizardOfWor	**3480.0**	2350.0	1850.0
Zaxxon	6380.0	**8100.0**	8050.0

Since parameter space noise is related to evolution strategies due to the way it uses parameter perturbations to explore, we also include a comparison with the results of Salimans et al. [2017] on ALE environments in Table 5.2. Notice that we here only compare the final performance of the policy *without* exploration behavior, which explains some discrepancies between Figure 5.4 and Table 5.2. For this experiment, we run the final policy obtained by ϵ-greedy exploration and parameter space noise exploration for 10 subsequent episodes and compute the mean performance. The same evaluation procedure was used by Salimans et al. [2017] such that we can directly compare our results with theirs. However, our agent was only trained on 40M frames, which is in stark contrast to the 1 000 M frames that were required to train the ES agent proposed by Salimans et al. [2017], while still outperforming it on the vast majority of the 21 included environments. This highlights that a feasible

middle ground between ES and its superior exploration behavior as reported by Salimans et al. [2017] and traditional and more sample efficient reinforcement learning with action space noise exists.

On a final note, other proposed improvements to DQN like double DQN van Hasselt [2010], prioritized experience replay Schaul et al. [2015b], and dueling networks Wang et al. [2016] are orthogonal to our improvements and would therefore likely improve results further. We leave the experimental validation of this theory to future work since it would require significant additional computational resources.

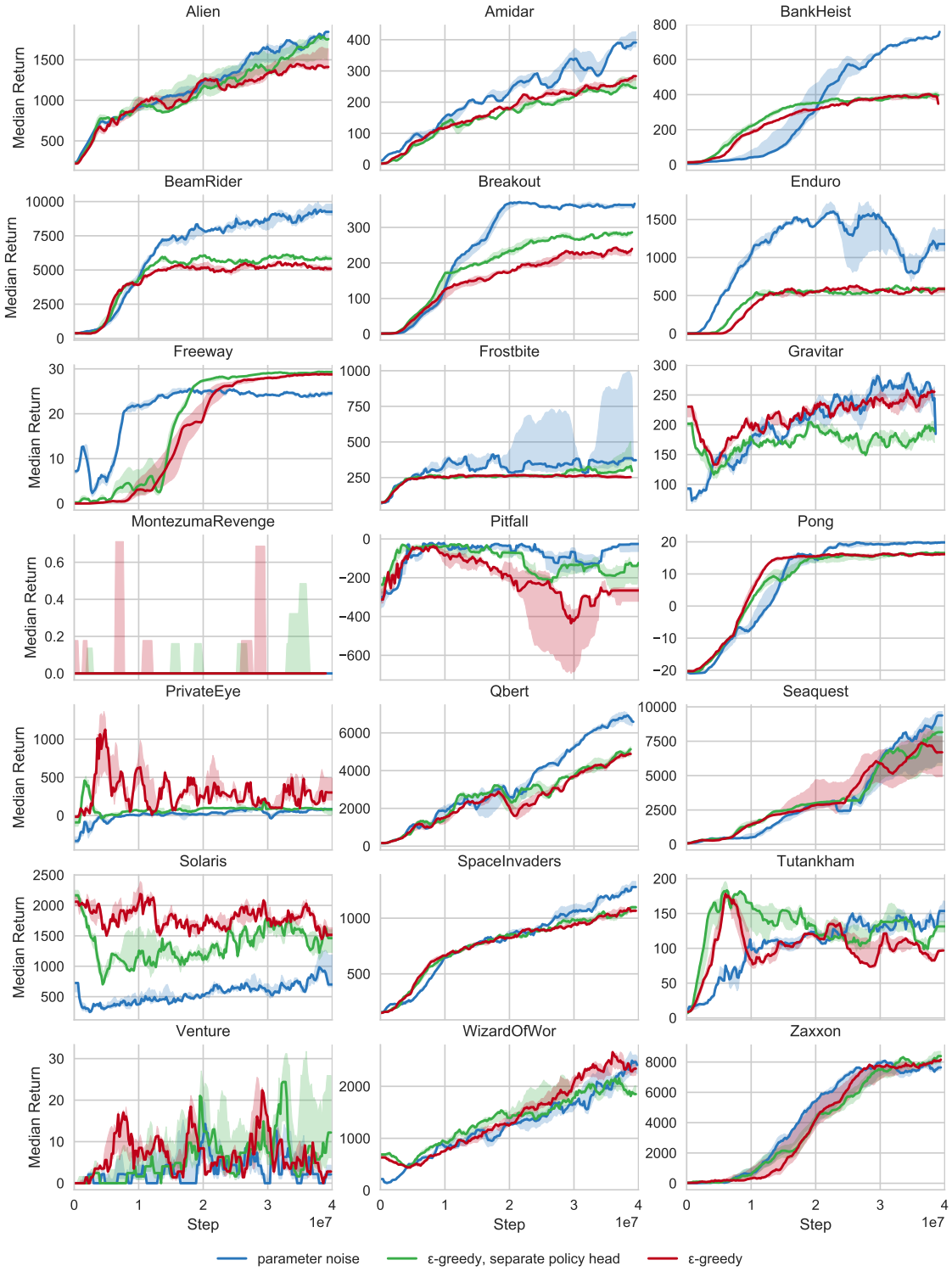


Figure 5.4: *Median DQN returns for 21 ALE environments for parameter space noise exploration (in blue), ϵ-greedy exploration (in red), and ϵ-greedy exploration with separate policy head as a control (in green) plotted over training steps. Returns are smoothed slightly to remove noise.*

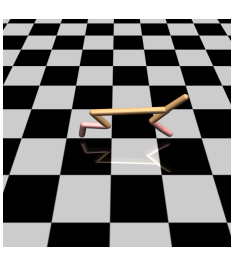
5.3 Continuous Control Experiments

Section 5.1 and Section 5.2 have shown promising results but so far we have only evaluated parameter space noise on domains with discrete action spaces and with DQN as the deep reinforcement learning algorithm of choice. In this section we thus extend our experimental results to the continuous domain with DDPG, which is especially relevant for the field of robotics.

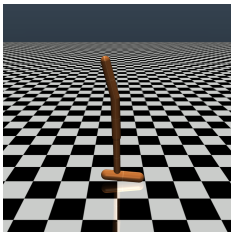
5.3.1 Environments

We use two different but similar sets of environments in this section. We first describe the standard benchmark for continuous control problems that is part of OpenAI Gym. While these environments are standard benchmarks for deep reinforcement learning, they all exhibit very dense rewards, which makes them not ideal for testing for exploration behavior. For that reason, we also introduce similar problems that have been modified such that the reward function is extremely sparse, often only yielding a single reward once a goal has been achieved and zero otherwise.

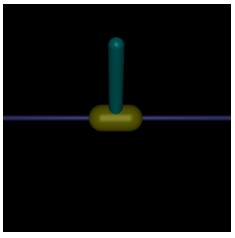
5.3.1.1 OpenAI Gym Continuous Control



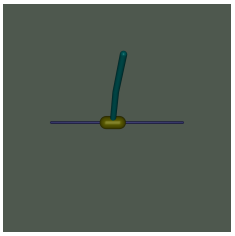
(a) *HalfCheetah*



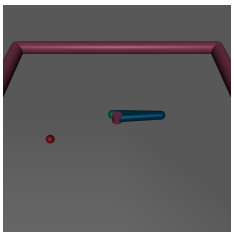
(b) *Hopper*



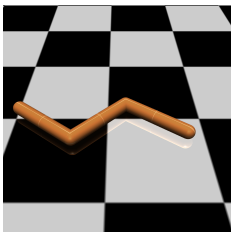
(c) *InvertedPendulum*



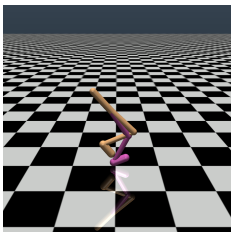
(d) *InvertedDoublePendulum*



(e) *Reacher*



(f) *Swimmer*



(g) *Walker2d*

Figure 5.5: *Continuous control environments simulated using MuJoCo and OpenAI Gym. The agent learns from the state of the physics simulation and pixel renderings are only provided here for visualization purposes.*

For the evaluation of parameter space noise with continuous action spaces, we use the set of continuous control problems that are part of OpenAI Gym [Brockman et al., 2016]. These problems have become the de facto standard for benchmarking algorithms for continuous control and the corresponding code is available online.[6] Under the hood, OpenAI Gym uses the rigid body simulator MuJoCo [Todorov et al., 2012], which allows for fast and numerically stable simulation. We depict all continuous control environments that we use for our experiments in Figure 5.5, which range from manipulation to challenging locomotion problems.

Concretely, we use the following 7 environments in all our continuous control experiments:

[6] `https://github.com/openai/gym/blob/master/gym/envs/mujoco`

- **HalfCheetah** ($\mathcal{S} \subset \mathbb{R}^{17}$, $\mathcal{A} \subset \mathbb{R}^6$) The agent controls a 6 DoF model of a 2-dimensional (hence the name) cheetah. The goal is to run forward as quickly as possible. The reward contains the delta in distance covered as well as a small penalty proportional to the magnitude of the action. The state contains the angular position and velocities of the joints of the model.
- **Hopper** ($\mathcal{S} \subset \mathbb{R}^{11}$, $\mathcal{A} \subset \mathbb{R}^3$) The agent controls a 3 DoF model for single-legged locomotion. The goal is to run move forward, which is usually achieved by hopping (hence the name). The reward contains the delta in distance covered as well as a small penalty proportional to the magnitude of the action. The state contains the angular position and velocities of the joints of the model. If the hopper model falls to the ground, the environment resets.
- **InvertedDoublePendulum** ($\mathcal{S} \subset \mathbb{R}^{11}$, $\mathcal{A} \subset \mathbb{R}$) The agent controls a sledge that has a pendulum mounted to it. The goal is to balance the pendulum by moving the sledge left and right. Crucially, the agent cannot actuate the joint of the pendulum directly. The agent receives a constant reward of 1 in each step. The state contains the angular position and velocities of the joints of the model. If the pendulum falls below a certain threshold, the environment resets.
- **InvertedPendulum** ($\mathcal{S} \subset \mathbb{R}^4$, $\mathcal{A} \subset \mathbb{R}$) The agent controls a sledge that has a double pendulum mounted to it. The goal is to balance the double pendulum by moving the sledge left and right. Crucially, the agent cannot actuate any of the joints of the double pendulum directly. The agent receives a constant reward of 1 in each step. The state contains the angular position and velocities of the joints of the model. If the double pendulum falls below a certain threshold, the environment resets.
- **Reacher** ($\mathcal{S} \subset \mathbb{R}^{11}$, $\mathcal{A} \subset \mathbb{R}^2$) The agent controls a 2 DoF model of a robotic arm. The goal is to move the end effector towards a defined goal position. The agent receives a penalty proportional to the distance between the position of its current end effector and the goal position as well as a small penalty proportional to the magnitude of the action. The state contains the angular position and velocities of the current distance between end effector and goal position in Cartesian space.
- **Swimmer** ($\mathcal{S} \subset \mathbb{R}^8$, $\mathcal{A} \subset \mathbb{R}^2$) The agent controls a 2 DoF model of a worm-like creature. The goal is to move forward as fast as quickly, usually by a swimming motion. The reward contains the delta in distance covered as well as a small penalty proportional to the magnitude of the action. The state contains the angular position and velocities of the joints of the model.
- **Walker2D** ($\mathcal{S} \subset \mathbb{R}^{17}$, $\mathcal{A} \subset \mathbb{R}^6$) The agent control a 6 DoF model of a two-legged creature without an upper body. The goal is to move forward as fast as quickly, usually by a walking motion. The reward contains the delta in distance covered as well as a small penalty proportional to the magnitude of the action. The state contains the angular position and velocities of the joints of the model. If the walker model falls to the ground, the environment resets.

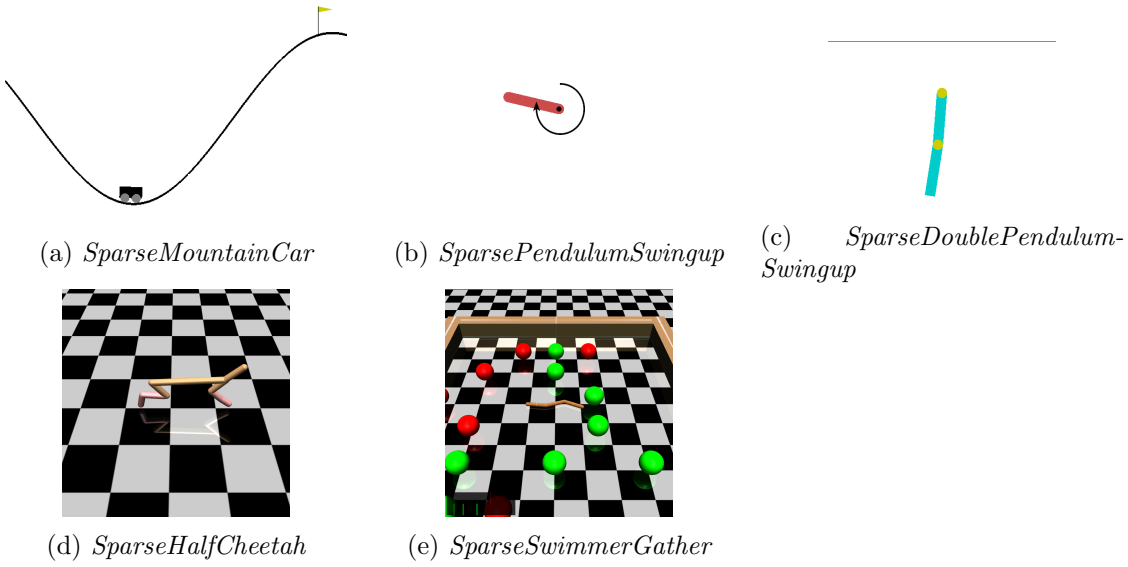
A problem with the aforementioned problems is that observations can be on vastly different scales (e.g. velocities are typically much larger than the corresponding positions of the respective joints). To overcome this problem, we employ a simple normalization schema that has also been popularized by Schulman et al. [2015b]. More concretely, we normalize each state $\boldsymbol{s}$ before passing it into a neural network:

$$\bar{\boldsymbol{s}} = \frac{\boldsymbol{s} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}, \tag{5.1}$$

where all operations are element wise and $\boldsymbol{\mu}, \boldsymbol{\sigma}$ have the same dimension as $\boldsymbol{s}$. As should be obvious, this ensures that each dimension of $\boldsymbol{s}$ has approximately zero variance and

unit variance. Unfortunately, we have no information about $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$, which is why we continuously re-estimate them.[7] This re-estimation causes some non-stationarity but we found that the change occurs slow enough that it does not really matter in practice. On the other hand, if we do not employ this normalization schema, performance degrades severely.

5.3.1.2 Continuous Control with Sparse Rewards



(a) *SparseMountainCar* (b) *SparsePendulumSwingup* (c) *SparseDoublePendulum-Swingup*

(d) *SparseHalfCheetah* (e) *SparseSwimmerGather*

Figure 5.6: *Continuous control environments with sparse rewards. Renderings are from OpenAI Gym and rllab [Duan et al., 2016].*

While the dense continuous control problems described in the previous section are frequently used as a benchmark, they have an extremely dense reward structure that yields a reward in every step. Since we are especially interested in the exploration behavior of our proposed method, we use continuous control problems with sparse rewards instead. Houthooft et al. [2016] have observed similar problems when evaluating their exploration method, and we use the same environments that they describe, which are based on rllab [Duan et al., 2016].[8]

Concretely, we use the following 7 environments in all our continuous control experiments, which are also depicted in Figure 5.6:

- **SparseMountainCar** ($\mathcal{S} \subset \mathbb{R}^2$, $\mathcal{A} \subset \mathbb{R}$) The agent controls a mountain car with a single DoF. The goal is to drive the car up the hill towards the flag. However, the car is underpowered such that the agent needs to collect momentum first before being able to escape the valley. The agent only receives a reward if it successfully reaches the flag and zero otherwise.
- **SparsePendulumSwingup** ($\mathcal{S} \subset \mathbb{R}^4$, $\mathcal{A} \subset \mathbb{R}$) The agent controls a single DoF pendulum. The goal is to swing up the pendulum and then balance it. The agent only receives a reward if the pendulum is moved above a certain threshold value and zero otherwise.
- **SparseDoublePendulumSwingup** ($\mathcal{S} \subset \mathbb{R}^6$, $\mathcal{A} \subset \mathbb{R}$) The agent controls a 2 DoF pendulum. The goal is to swing up the double pendulum above a certain threshold.

[7] Collecting an estimate early on would be an alternative but the distribution may shift significantly as training progresses.

[8] `https://github.com/rll/rllab`

However, the agent can only actuate the base joint of the double pendulum, further increasing the difficulty.

- **SparseHalfCheetah** ($\mathcal{S} \subset \mathbb{R}^{17}$, $\mathcal{A} \subset \mathbb{R}^6$) The agent controls a half cheetah model like described in the previous section. However, in contrast to the previous environment, the agent only receives a reward if the cheetah runs for multiple meters and zero otherwise.
- **SparseSwimmerGather** ($\mathcal{S} \subset \mathbb{R}^{33}$, $\mathcal{A} \subset \mathbb{R}^2$) The agent controls a swimmer model like described in the previous section. However, the goal is now to collect apples (the blue spheres) and to avoid bombs (the red spheres). For all other states, the agent receives a reward of zero. This is an extremely challenging environment since it requires the swimmer to learn a locomotion skill and move towards desirable goals with almost no reward signal.

For all tasks, we use a time horizon of $T = 500$ steps before resetting the environment.

We further normalize states as described in the previous section for the dense continuous control case to avoid the aforementioned difference in scale across dimensions.

5.3.2 Experimental Setup

For all our DDPG, we use a network architecture that is similar to the one described by Lillicrap et al. [2015]. However, we use a slightly smaller network with 2 hidden layers with 64 units each followed by a ReLu non-linearity for both the actor and the critic network. Similarly to the architecture described by Lillicrap et al. [2015], we do not include the actions until the second layer in the critic network to force the network to learn a representations from states alone first. Since we wish to perturb the network, we include layer normalization [Ba et al., 2016] between all layers (compare Section 4.2). In order to ensure that actions are bound, we use a tanh activation in the last layer of the actor network, thus ensuring that actions are in $[-1, 1]$. If an environment requires actions within different bounds, we rescale them appropriately before execution in the environment. We also include layer normalization in all networks and for all exploration methods since we found that it generally improves performance of all methods and significantly stabilizes training. For the target networks, we use soft updates with $\tau = 0.001$. We use the Adam optimizer [Kingma and Ba, 2014] with batch sizes of 128 to train both the actor and the critic network. However, for the actor network we use a learning rate of 10^{-4} whereas the critic is updated using a learning rate of 10^{-3}. Furthermore, we include $L2$ regularization with a coefficient of 10^{-2} when training the critic network to avoid overfitting. In all experiments, we use a standard replay buffer that holds up to 100 000 transitions. We use $\gamma = 0.99$ to discount returns and apply the normalization of states as previously described.

We consider four different exploration methods:

- **No exploration** In this configuration, we disable all exploration and directly execute the actions that our policy predicts. This configuration functions as a control since an environment that requires no exploration is not an appropriate benchmark for our proposed exploration method.
- **Exploration with uncorrelated additive Gaussian noise** In this configuration, we include uncorrelated additive Gaussian noise before executing an action: $\boldsymbol{a}_t = \pi(\boldsymbol{s}) + \mathcal{N}(\boldsymbol{0}, \sigma^2 \boldsymbol{I})$. For environments with dense rewards we use $\sigma = 0.2$ and for environments with sparse rewards we use $\sigma = 0.6$.
- **Exploration with correlated additive Gaussian noise** In this configuration, we include correlated additive Gaussian noise using the Ornstein-Uhlenbeck process [Uhlenbeck and Ornstein, 1930] before executing an action: $\boldsymbol{a}_t = \pi(\boldsymbol{s}) + \mathrm{OU}(\sigma, \theta)$. For

Table 5.3: *Hyperparameters for DDPG experiments.*

Hyperparameter	Value
Soft target network update parameter τ	$\tau = 10^{-3}$
Optimizer	Adam [Kingma and Ba, 2014]
Batch size	128
Actor learning rate	10^{-4}
Critic learning rate	10^{-3}
Critic $L2$ regularization	10^{-2}
Training steps	10^6
Replay buffer size	10^5
Discount factor γ	0.99
Network architecture	Lillicrap et al. [2015] with 64 units each
Uncorrelated noise σ	$\sigma = 0.2$ or $\sigma = 0.6$
Correlated noise σ	$\sigma = 0.2$ or $\sigma = 0.6$
Correlated noise θ	$\theta = 0.15$
Parameter noise adaption interval (T_{adapt})	50
Parameter noise distance measure	Section 4.3.2

environments with dense rewards we use $\sigma = 0.2$ and for environments with sparse rewards we use $\sigma = 0.6$. We keep $\theta = 0.15$ fixed for both cases.

- **Exploration with parameter space noise** In this configuration, we use parameter space noise as described in Chapter 4: $\boldsymbol{a}_t = \widetilde{\pi}(\boldsymbol{s})$. In order to ensure a fair comparison, we use adaptive parameter space noise as described in Section 4.3.2 such that we achieve a perturbation as measured in action space with $\sigma = 0.2$ and $\sigma = 0.6$ for the dense and sparse environments, respectively.

All hyperparameters are conveniently summarized in Table 5.3. An open-source implementation of adaptive parameter space noise with DDPG is available online.[9]

5.3.3 Results

5.3.3.1 OpenAI Gym Continuous Control

We start by evaluating the performance on the standard MuJoCo continuous control tasks as implemented by OpenAI Gym. Similar to the DQN case, we plot the achieved return over training steps with exploration noise enabled since we are, in particular, interested in this behavior. Since we aggregate over multiple runs for each configuration to avoid fluctuations due to the random seed, we depict the median (line) and the interquartile range (shaded area) for each of the four exploration configurations: adaptive parameter noise, correlated action space noise (Ornstein-Uhlenbeck process), uncorrelated action space noise (Gaussian), and no exploration altogether.

On *HalfCheetah* (Figure 5.7), parameter space noise achieves significantly higher returns than all other configurations. By visualizing the policy during training, we find that this is due to the fact that other exploration methods prematurely converge to sub-optimal locomotion strategies. For example, policies that were trained without parameter space noise often learn to flip the cheetah on its back and then use rapid movements of the legs to "wiggle" forward. Another sub-optimal strategy that we see is one where the agent

[9] `https://github.com/openai/baselines/tree/master/baselines/ddpg`

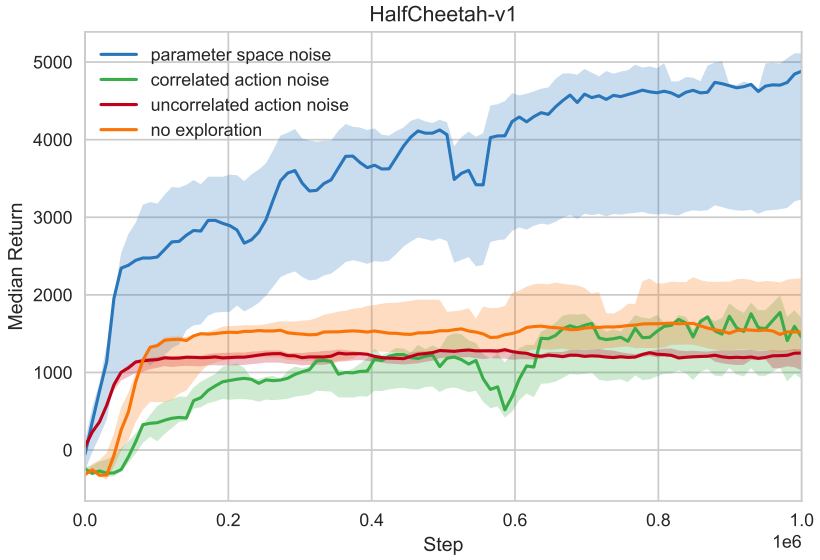


Figure 5.7: *Median DDPG returns for* HalfCheetah *with four different exploration methods. Returns are smoothed slightly to remove noise.*

balances the cheetah model on a single leg and then hops forward. Importantly, parameter space noise also occasionally exhibits this behavior during our experiments but, in contrast to all other exploration strategies, will escape these sub-optimal strategies and converges to a "normal" locomotion strategy. This is also apparent in the plot for HalfCheetah: All other exploration methods converge to a strategy that achieves a return around $2,000$, whereas parameter space noise finds much better strategies.

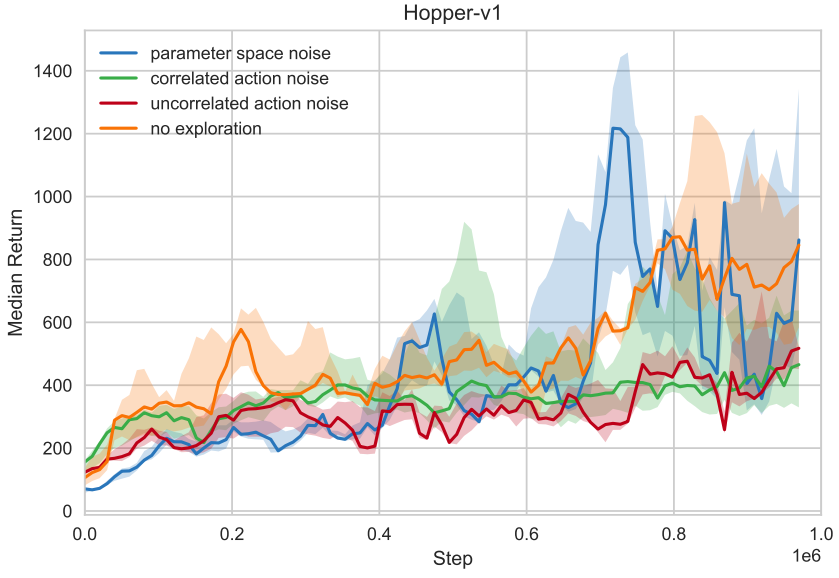
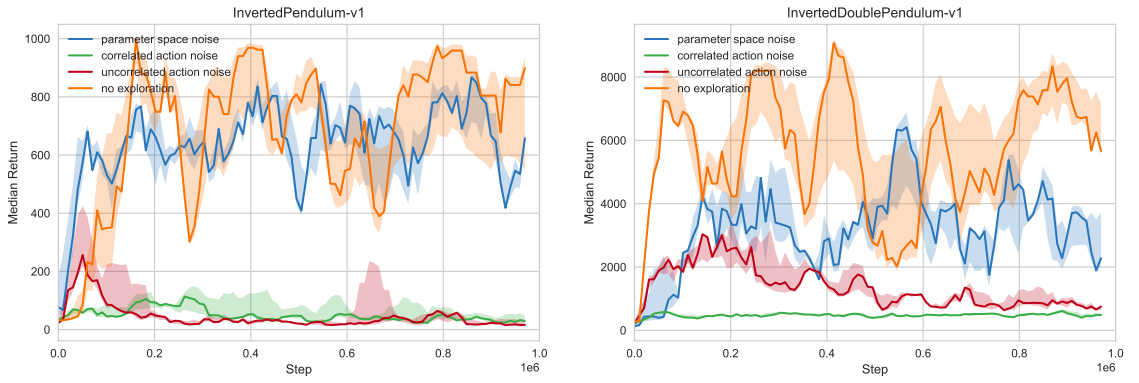


Figure 5.8: *Median DDPG returns for* Hopper *with four different exploration methods. Returns are smoothed slightly to remove noise.*

In the case of *Hopper* (Figure 5.8), parameter space noise seems to perform reasonably well and does seem to find good strategies. However, we find that performance is quite noisy and performance seems to occasionally collapse. We believe that this is due to the inherent instability of training DDPG (also experienced by Duan et al. [2016]). Still, parameter

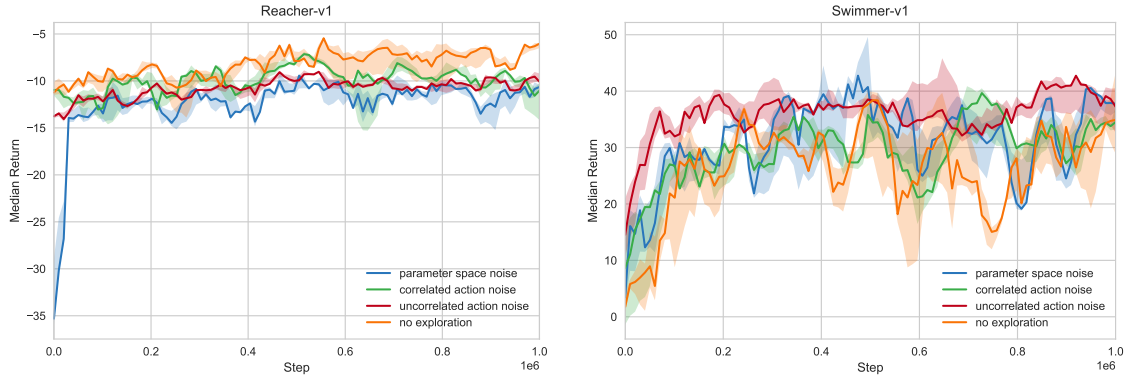
space noise seems to outperform correlated and uncorrelated action space noise on this environment.



(a) (b)

Figure 5.9: *Median DDPG returns for* InvertedPendulum *(left) and* InvertedDoublePendulum *(right) with four different exploration methods. Returns are smoothed slightly to remove noise.*

We jointly summarize the results for *InvertedPendulum* (Figure 5.9a) and *InvertedDoublePendulum* (Figure 5.9b) since they are quite similar: We find that no exploration noise seems to perform best on these environments. This makes sense since we here opted to depict the performance of the agent during training since we are especially interested in its exploration behavior. However, since this task does not require significant exploration due to its simplicity, the agent without any exploration is better off in this comparison. Note, however, that parameter space noise outperforms both forms of action space noise, presumably due to the fact that its actions are much less subject to random jitter, which seems to make it quite hard to maintain control over the pendulum.



(a) (b)

Figure 5.10: *Median DDPG returns for* Reacher *(left) and* Swimmer *(right) with four different exploration methods. Returns are smoothed slightly to remove noise.*

On *Reacher* (Figure 5.10a), parameter space noise seems to fare worst when compared against all other exploration methods. We are not sure why this is the case, but notice that no exploration outperforms all other methods on this task. This seems to indicate that the agent explores sufficiently without any exploration mechanism and, for some reason, parameter space noise seems to hinder performance significantly. Regarding our result on *Swimmer* (Figure 5.10b), we conclude that none of the exploration methods seem to achieve good policies here. We believe that this is again a problem with the inherent

stability of DDPG where the agent is simply incapable of learning from the experience it generates. This claim is supported by the fact that no exploration again performs comparably, indicating that exploration is not a limiting factor but rather algorithm stability is. In both cases, parameter space noise does not seem to make a significant difference.

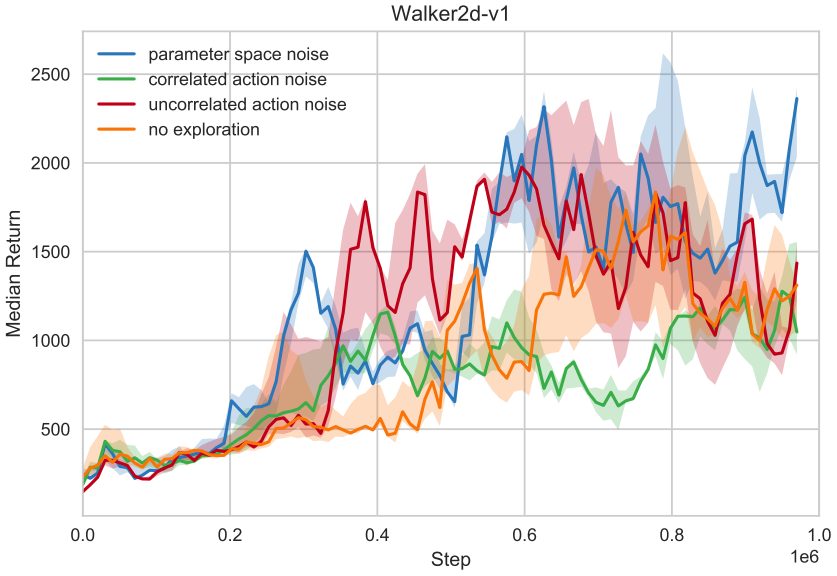


Figure 5.11: *Median DDPG returns for* Walker2d *with four different exploration methods. Returns are smoothed slightly to remove noise.*

Finally, on *Walker2d* (Figure 5.11), parameter space noise seems to again lead performance, followed by uncorrelated action noise. As in previous cases, the learning progress seems to be again quite unstable, which is yet another instance of the aforementioned stability problems of DDPG. Interestingly, even on this relatively complicated task, DDPG with no exploration still seems to perform somewhat competitively.

Overall, we find that parameter space noise vastly helps on *HalfCheetah*, where it avoids converging towards sub-optimal policies and always seems to find high performing ones. On other environments, parameter space noise does not seem to provide a significant benefit. However, as we have already described, we find that no exploration typically achieves very similar results. This is worrying since it indicates that exploration seems unnecessary to begin with. We believe that this is due to two factors. First, the rewards in all continuous control tasks are inherently dense, where a non-zero reward is emitted in every step. Second, the start conditions of the environment are significantly randomized, which, combined with the random initialization of the policy, presumably already covers a significant fraction of the state space. These two factors combined seem to make the OpenAI Gym continuous control problems unsuitable for testing for exploration, which has similarly been noted by Houthooft et al. [2016].

5.3.3.2 Continuous Control with Sparse Rewards

In the previous section, we have shown that exploration is not a real concern on the standard OpenAI Gym continuous control since DDPG without any exploration still achieves comparable performance on most problems. We therefore now turn to the previously described environments that exhibit a sparse reward structure to allow us to directly test for the exploration capabilities of our proposed method in continuous action spaces.

Similar to before, we plot the achieved return over training steps with exploration noise enabled since we are, in particular, interested in this behavior. Since we aggregate over mul-

tiple runs for each configuration to avoid fluctuations due to the random seed, we depict the median (line) and the interquartile range (shaded area) for each of the four exploration configurations: adaptive parameter noise, correlated action space noise (Ornstein-Uhlenbeck process), uncorrelated action space noise (Gaussian), and no exploration altogether.

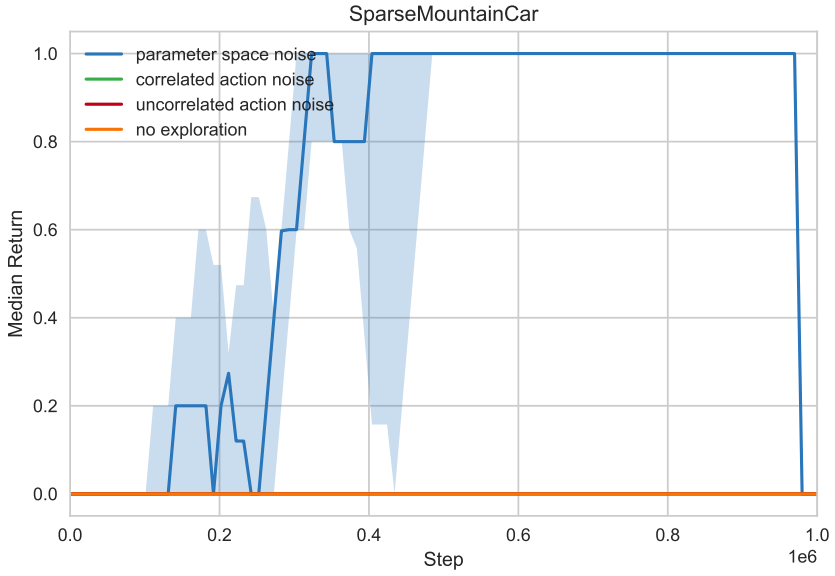
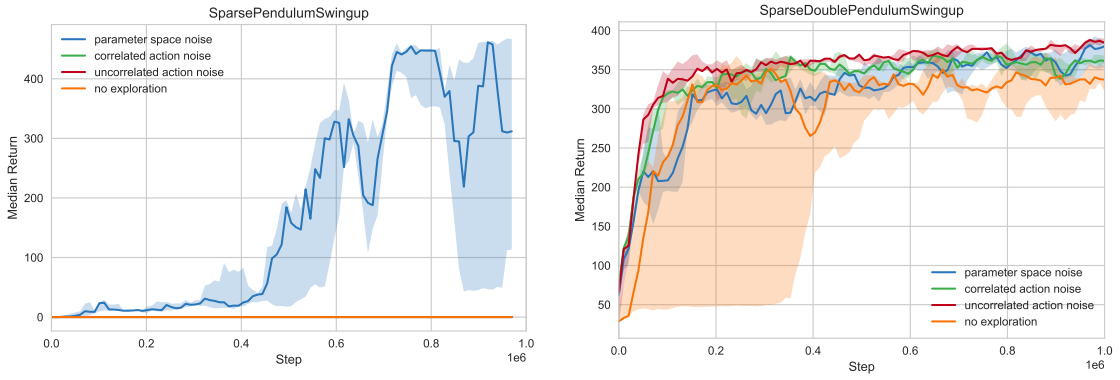


Figure 5.12: *Median DDPG returns for* SparseMountainCar *with four different exploration methods. Returns are smoothed slightly to remove noise.*

MountainCar (Figure 5.12) is a first naturally sparse environment that shows very encouraging result: Only parameter space noise learns a successful strategy whereas all other methods of exploration fail. We believe that the property of consistent exploration is what makes parameter space so much more efficient in this case. Since the behavior policy is fully conditioned on the state, the mountain car can consistently collect momentum until it reaches the goal position. Correlated and uncorrelated action space noise instead cause oscillation in the exploration behavior, which causes the mountain car to never obtain sufficient momentum to drive up the hill. It thus comes as no surprise that our control, which disables all exploration, does not find a solution since it will simply get stuck in the valley.

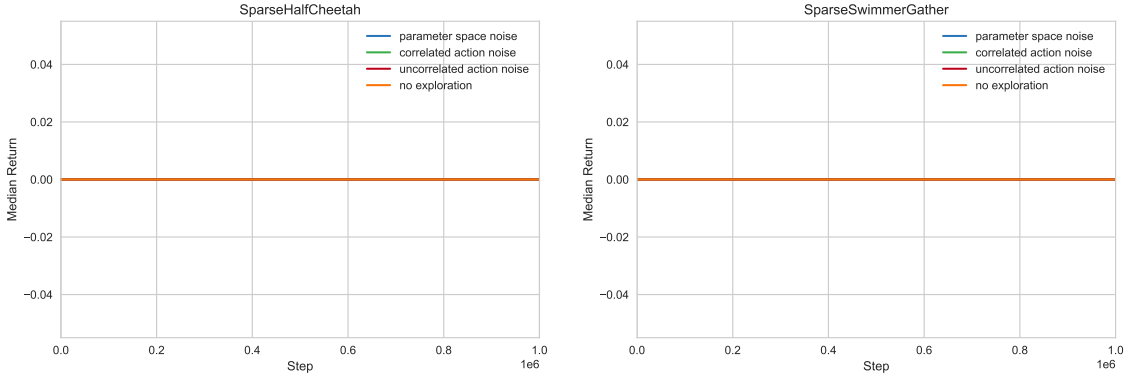


(a) (b)

Figure 5.13: *Median DDPG returns for* SparsePendulumSwingup *(left) and* SparseDoublePendulumSwingup *(right) with four different exploration methods. Returns are smoothed slightly to remove noise.*

Similarly, on *SparsePendulumSwingup* (Figure 5.13a), only parameter space noise learns a successful strategy to solve the environment. All other configurations never even obtain a single reward during training. Clearly, exploration is important in this environment, and parameter space noise clearly outperforms traditional action space noise. In contrast and quite interestingly, all exploration strategies succeed on the conceptually similar *SparseDoublePendulumSwingup* (Figure 5.13b). However, the results still make sense: First, our control experiment that enables all exploration clearly takes significantly longer to find a successful strategy and also converges to a worse solution even though we depict the training performance. This also indicates that exploration is surprisingly unnecessary in this specific environment. We believe that this is due to the chaotic nature of the problem: It is somewhat likely that a random action combined with a fortunate initial state leads to a state where the pendulum is swung such that it achieves the threshold. It therefore seems consistent that parameter space noise, correlated action space noise, and uncorrelated action space noise perform very similarly on this environment since exploration helps on this environment but is far from crucial.

SparseHalfCheetah (Figure 5.14a) is a quite challenging environment in which all exploration methods fail. To better understand this failure case, we visualized the behavior during training. Even though parameter space noise results in locomotion behavior occasionally, it is typically undirected and thus does not cover the necessary minimum distance of 5 meters before a reward is obtained. Finally, we consider the *SparseSwimmerGather* (Figure 5.14b) environment. This is an extremely challenging environment which requires the agent to first learn a locomotion skill ("swimming") and then requires it to follow a goal-driven high policy ("collect the apples") by using this low-level skill to achieve it. Unfortunately, this environment proved to be too hard for parameter space noise exploration, which never manages to develop the necessary locomotion and thus never receives a learning signal. The same holds true for all other exploration and for the control.



(a) (b)

Figure 5.14: *Median DDPG returns for* SparseHalfCheetah *(left) and* SparseSwimmerGather *(right) with four different exploration methods. Returns are smoothed slightly to remove noise.*

In summary, we believe that this set of experiments in sparse domains highlights the superior exploration behavior of parameter space noise over action space noise. However, it also highlights that parameter space noise alone cannot drive sufficient exploration in domains that require the development of non-trivial basic skills like locomotion in order to achieve higher-level goals. This is apparent in both the *SparseHalfCheetah* and *SparseSwimmerGather* environments. That being said, parameter space noise was designed to be a conceptually simple yet effective drop-in replacement for action space noise. Our results on these sparse environments highlight again that parameter space noise indeed

results in richer exploration behavior.

5.4 Robot Manipulation Experiments

In the previous experimental sections, we evaluated the performance of parameter space noise for discrete and continuous environments. However, so far all environments that we considered were more of theoretical importance and had no immediate real-world applications. We therefore now evaluate parameter space noise on challenging robot manipulation tasks that are based on real-world models and have real-world applications.

5.4.1 Environments

Figure 5.15: *The research robotics platform from Fetch Robotics used for all our experiments. Product image obtained from* `http://fetchrobotics.com`.

We use the environments that were recently proposed by Andrychowicz et al. [2017] since they use a real-world robot and exhibit sparse reward structures. All environments use the Fetch Robotics robot platform, which is depicted in Figure 5.15 and that features a 7 DoF arm with a 2-fingered gripper. We do not use the platform's capability to move around since all tasks only require stationary manipulation skills.

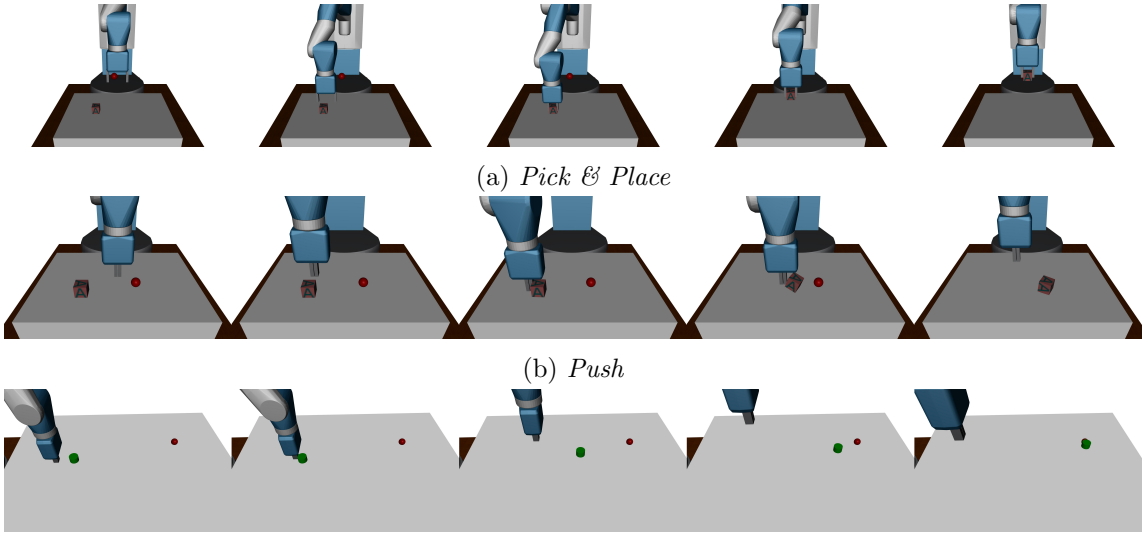
To simplify the control problem, we use position control in Cartesian space to control the movement of the robot's arm. Instead of producing absolute coordinates, we use relative control, i.e. the policy produces an offset relative to the current position of the robot's end effector. Since all tasks further interact with objects that lie flat on the surface of a table, we ignore the rotation of the robot's end effector fixed and thus only consider the position instead of the entire pose. We use inverse kinematics to find the joint angles of the robot arm given the desired end effector pose after applying the delta as predicted by the policy and with fixed rotation. The gripper is modeled as having 2 distinct states: open and close. Again, we use position control to move the 2 fingers to the respective position depending on this state. The action space for all tasks is thus $\mathcal{A} \subset \mathbb{R}^4$, where the first 3 dimensions define the relative offset in Cartesian space and the last dimension models the gripper, which gets discretized into a binary value by thresholding.

In all experiments, we assume direct knowledge of the entire state. When deploying on the physical robot, the state can then either be inferred by systems that predict it given an image or other sensor readings (e.g. Tobin et al. [2017]) or the network can be trained directly from images end-to-end by utilizing the available information about state in simulation (e.g. Pinto et al. [2017]). However, we only consider the simulated case since we focus on the problem of exploration, which is orthogonal to the problem of transferability. The concrete state depends on the task at hand and we describe it in greater detail when introducing the concrete tasks.

Finally, all tasks utilize fully sparse reward functions that are based on the achievement of a *goal state*. In practice, we follow the setup described by Andrychowicz et al. [2017] and use the following reward function for all environments:

$$r(\boldsymbol{s}, \boldsymbol{g}, \boldsymbol{a}) = \begin{cases} 0 & \text{if } \boldsymbol{g} = m(\boldsymbol{s}), \\ -1 & \text{otherwise,} \end{cases} \tag{5.2}$$

where $m : \mathcal{S} \mapsto \mathcal{G}$ transforms a state into the corresponding goal (compare Section 2.3.4 for details on this notation). In other words, the agent receives a reward of -1 if the goal state has not yet been achieved and a reward of 0 otherwise. Notice that this formulation is equivalent to $r(\cdot, \cdot, \cdot) + 1$ and we simply use this formulation since it follows what Andrychowicz et al. [2017] describe.



(a) *Pick & Place*

(b) *Push*

(c) *Slide*

Figure 5.16: *Three robot manipulation problems with sparse rewards. All problems were proposed by Andrychowicz et al. [2017]. We also include a fourth, simple task,* Reach*, which is not depicted here.*

We again use MuJoCo [Todorov et al., 2012] to efficiently simulate all environments, including an open-sourced model of Fetch.[10] More concretely, we consider the following 3 tasks that were originally proposed by Andrychowicz et al. [2017] and which are illustrated in Figure 5.16.

- **Pick & Place** ($\mathcal{S} \subset \mathbb{R}^{22}, \mathcal{G} \subset \mathbb{R}^3, \mathcal{A} \subset \mathbb{R}^4$) The agent controls the arm and gripper of a Fetch robot. The goal is to move a wooden block that lies on the surface of a table to a goal position, which is above the surface of the table. In order to do so, the robot has to first grasp the object and then lift it up and move it towards

[10] https://github.com/openai/mujoco-py/tree/master/xmls/fetch

the goal position. We include a single demonstration frame that shows a successful grasp to aid with learning this multi-step problem. The agent observes the state of the robot arm (i.e. joint positions and velocities), the gripper position, as well as the position of the wooden block and the goal position.

- **Push** ($\mathcal{S} \subset \mathbb{R}^{22}, \mathcal{G} \subset \mathbb{R}^3, \mathcal{A} \subset \mathbb{R}^3$) The agent controls the arm of a Fetch robot while the gripper is kept closed. The goal is to move a wooden block that lies on the surface of a table towards a goal position on the table surface. Since the gripper is closed, the robot has to push the block towards that goal position, likely requiring multiple such pushes. The agent observes the state of the robot arm (i.e. joint positions and velocities) as well as the position of the wooden block and the goal position. For this task, we include an additional penetration penalty to discourage the agent to exploit the soft constrained realization of the physics simulation.
- **Slide** ($\mathcal{S} \subset \mathbb{R}^{22}, \mathcal{G} \subset \mathbb{R}^3, \mathcal{A} \subset \mathbb{R}^3$) The agent controls the arm of a Fetch robot while the gripper is kept closed. The goal is to slide a puck across a long table with little surface friction (think air hockey) towards a goal position. The goal position is selected such that the robot cannot reach it and has to indeed slide the puck towards it. The agent observes the state of the robot arm (i.e. joint positions and velocities) as well as the position of the puck and the goal position.
- **Reach** ($\mathcal{S} \subset \mathbb{R}^{10}, \mathcal{G} \subset \mathbb{R}^3, \mathcal{A} \subset \mathbb{R}^3$) The agent controls the arm of a Fetch robot while the gripper is kept closed. The goal is to simply move the end effector towards a goal position. The agent observes the state of the robot arm (i.e. joint positions and velocities) as well as the position of the goal. This is a very simple task and is mostly included as a control, where we expect all agents to be able to learn.

5.4.1.1 Experimental Setup

For the following experiments, we use DDPG with an actor and critic network consisting of 3 layers with 64 ReLu units each. In contrast to the previous experiments, we include actions immediately to follow the setup described by Andrychowicz et al. [2017] as closely as possible. Since we wish to perturb the network, we include layer normalization [Ba et al., 2016] between all layers (compare Section 4.2). In order to ensure that actions are bound, we use a tanh activation in the last layer of the actor network, thus ensuring that actions are in $[-1, 1]$. If an environment requires actions within different bounds, we rescale them appropriately before execution in the environment. We also include layer normalization in all networks and for all exploration methods since we found that it generally improves performance of all methods and significantly stabilizes training. For the target networks, we use soft updates with $\tau = 0.001$. We use the Adam optimizer [Kingma and Ba, 2014] with batch sizes of 128 to train both the actor and the critic network. We use a use a learning rate of 10^{-3} to train both networks. To generate rollout data, we use an implementation that uses 16 simulations in parallel. In all experiments, we use a replay buffer that holds up to 100 000 transitions. In all cases, we use $\gamma = 0.98$ to discount returns and apply the normalization of states as previously described. All hyperparameters are conveniently summarized in Table 5.4.

We consider four different configurations:

- **Exploration with parameter space noise** In this configuration, we use parameter space noise as described in Chapter 4: $\boldsymbol{a}_t = \widetilde{\pi}(\boldsymbol{s})$. In order to ensure a fair comparison, we use adaptive parameter space noise as described in Section 4.3.2 such that we achieve a perturbation as measured in action space with $\sigma = 0.1$.
- **Exploration with action space noise** In this configuration, we include uncorrelated additive Gaussian noise before executing an action: $\boldsymbol{a}_t = \pi(\boldsymbol{s}) + \mathcal{N}(\boldsymbol{0}, \sigma^2 \boldsymbol{I})$

Table 5.4: *Hyperparameters for HER and DDPG experiments.*

Hyperparameter	Value
Soft target network update parameter τ	$\tau = 10^{-3}$
Optimizer	Adam [Kingma and Ba, 2014]
Batch size	128
Actor learning rate	10^{-3}
Critic learning rate	10^{-3}
Training steps	500 000
Epoch length	2 500 steps
Time horizon T	50
Replay buffer size	10^5
Discount factor γ	0.98
Network architecture	3 layers with 64 ReLu units each
HER replay strategy	`future`
HER replay ration	4:1
Random action probability ϵ	0.2
Additive Gaussian noise scale σ	0.1
Parameter noise adaption interval (T_{adapt})	50
Parameter noise distance measure	Section 4.3.2

with $\sigma = 0.1$. To follow Andrychowicz et al. [2017], we also include a version of ϵ-greedy exploration adapted for the continuous case: With probability ϵ, we take a completely random action by uniformly sampling from the action space, and with probability $1 - \epsilon$, we execute the action predicted by the policy plus the additive Gaussian noise.

- **Exploration with HER and action space noise** In this configuration, we use the same action space exploration as described earlier and combine it with HER. We replay experience with a ratio of $4 : 1$, meaning that we sample four times as much HER data compared to standard experience replay data (compare Section 2.3.4).
- **Exploration with parameter space noise** In this configuration, we use the same parameter space exploration as described earlier and combine it with HER. We replay experience with a ratio of $4 : 1$, meaning that we sample four times as much HER data compared to standard experience replay data (compare Section 2.3.4).

The goal of these experiments is to compare vanilla DDPG against DDPG with HER (usually abbreviated as only HER). Furthermore, we wish to investigate if parameter space noise aids exploration if combined with HER.

5.4.2 Results

We start by considering the simplest environment: *Reach*. As mentioned before, this environment is mostly included as a control since we expect all algorithms to be able to learn on this task. In contrast to previous experiments, we now use the *success rate* and plot it over training epochs (1 epoch corresponds to 2 500 steps) since it is a more meaningful measure for these tasks, which is simply defined as:

$$\text{success rate} = \frac{\text{number of successful attempts}}{\text{number of total attempts}}. \tag{5.3}$$

Like before, we repeat the experiment multiple times using different random seeds and plot the median (line) and the interquartile range (shaded area) of the success rate. Results for Reach are depicted in Figure 5.17.

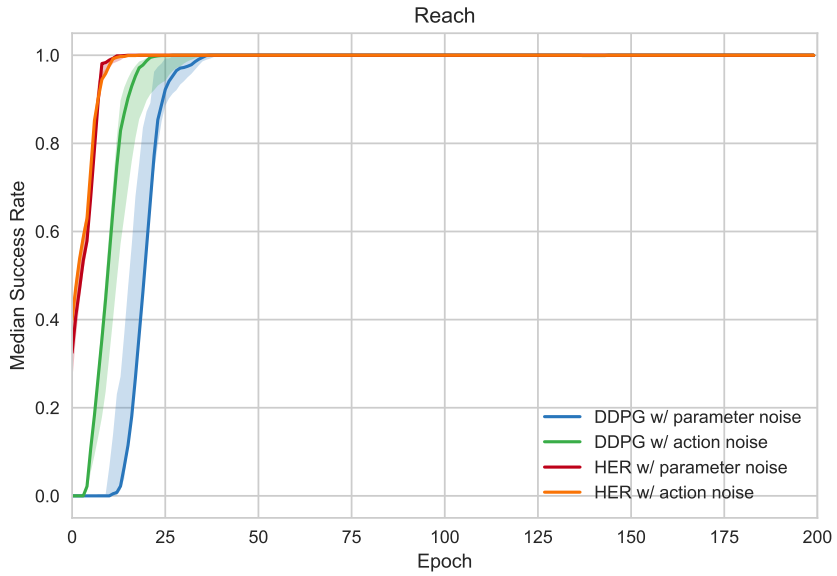


Figure 5.17: *Median success rate for* Reach *with four different exploration methods. Returns are smoothed slightly to remove noise.*

As expected, all configurations are able to learn on this simple task. Interestingly, parameter space noise gets outperformed by action space noise if used with vanilla DDPG. We believe that this is due to the fact that completely random exploration is actually beneficial here since it is much more likely to "hit" the target position by moving around randomly. Parameter space noise instead drives consistent exploration, but if it consistently moves in the wrong direction it does not help. This theory is backed up by the fact that HER with action space noise and parameter space performs virtually the same.

We now consider the *Pick & Place* task. This is a much more challenging environment and we expect that exploration is an important property here. We further note that this environment is extremely hard to learn for vanilla DDPG since the agent needs to pick up the block and move it to a goal position. Results are depicted in Figure 5.18.

Surprisingly, vanilla DDPG with parameter space noise is able to successfully learn this environment whereas vanilla DDPG with action space noise never even obtains a reward. This again highlights that parameter space noise can drive consistent and deep exploration, even on tasks that are very challenging (in this case it requires two high-level actions, i.e. picking up the block and then moving it towards the goal position) and extremely sparse. Furthermore, we can see that HER is able to successfully learn with both exploration strategies. However, parameter space noise seems to learn slightly faster and, more importantly, seems to result in more stable and consistent progress.

Next, we analyze the results for the *Push* task, which are depicted in Figure 5.19. On this environment, we find that vanilla DDPG is unfortunately unable to learn in both cases, i.e. parameter space noise alone does not seem to drive sufficient exploration here. We believe that the problem here are the complicated contact dynamics between gripper and block: In both cases, DDPG occasionally achieves a non-zero success rate but never seems to successfully learn from this experience, presumably because it never "understands" how different interactions with the block change its position. HER, on the other hand, excels at this since it always learns something from each interaction with the block. Both

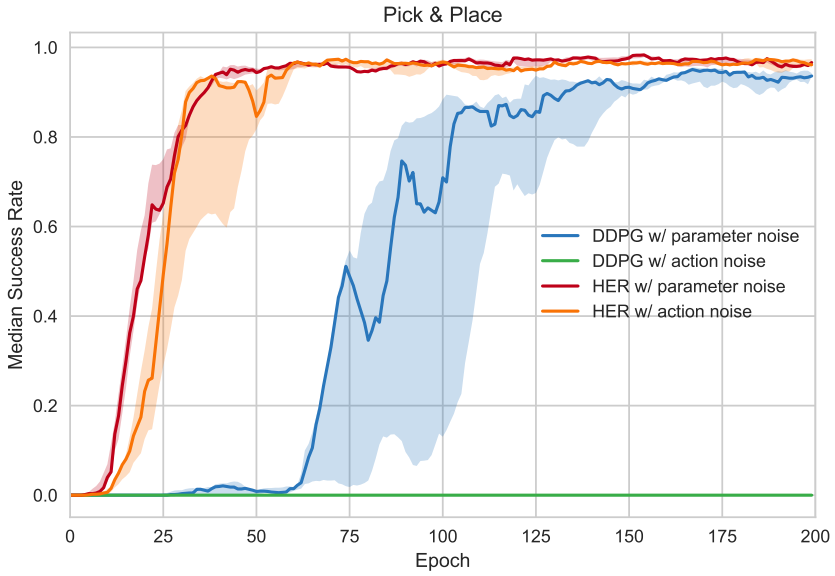


Figure 5.18: *Median success rate for* Pick & Place *with four different exploration methods. Returns are smoothed slightly to remove noise.*

configurations of HER with action noise and parameter noise for exploration are capable of learning the task at hand. Like before, parameter space noise seems to achieve slightly more consistent results compared to action space noise, but on this task the advantage is less obvious.

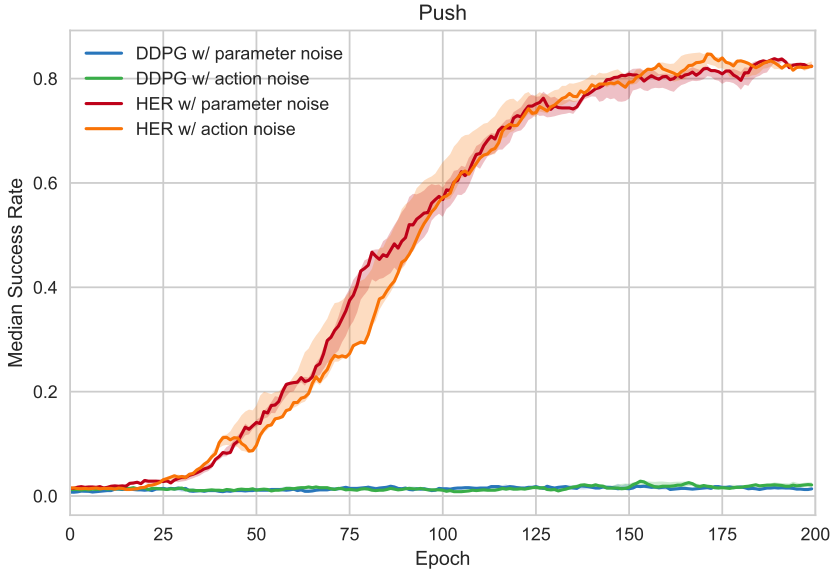


Figure 5.19: *Median success rate for* Push *with four different exploration methods. Returns are smoothed slightly to remove noise.*

Finally, we consider the last task, which is *Slide*. Results are depicted in Figure 5.20. On this task, all environments are again able to learn strategies, although they never achieve very high success rates. HER with parameter space noise again outperforms HER with traditional action space noise, again indicating that it seems to result in richer exploration behavior. However, due to the fact that all four configurations learn on this environment, not too much can be read into the results on this specific environment.

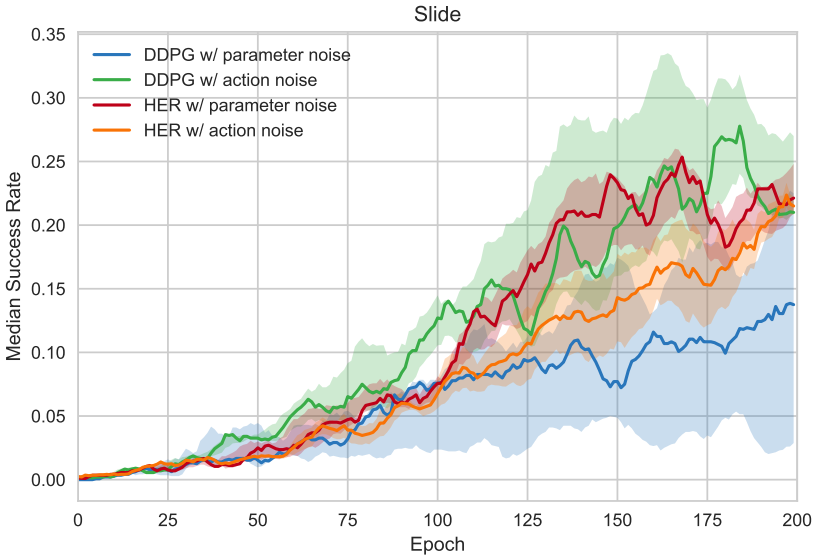


Figure 5.20: *Median success rate for* Slide *with four different exploration methods. Returns are smoothed slightly to remove noise.*

Overall, these results show that parameter space noise is capable of exploring on extremely challenging results with sparse rewards even with vanilla DDPG (compare results for *Pick & Place*). Furthermore, our results confirm that parameter space noise can successfully be combined with more sophisticated exploration mechanisms like Hindsight Experience Replay. This is an important results since we believe that exploration can only be solved by a system that learns high-level objectives and then relies on simpler exploration like action space noise and parameter space noise to discover and fine-tune the behavior that can achieve these high-level skills. As shown in this section, parameter space noise can indeed be combined in such a fashion and typically improves performance in these cases. We therefore believe that parameter space noise should be preferred over action space noise in such cases and have empirically confirmed this claim.

6. Conclusion

6.1 Summary

Exploration in today's reinforcement learning is still mostly realized using action space noise (e.g. ϵ-greedy or additive Gaussian noise), which has been around for centuries. While more sophisticated exploration schema exist, they often are quite complicated to implement. Furthermore, these schema often also require some basic exploration capabilities, which are typically still realized through action space noise exploration. The goal of this thesis was to investigate whether a conceptually simple idea we call *parameter space noise* can replace this traditional action space noise.

In this thesis, we first derive the necessary theory behind our approach and analyze why the resulting parameter space noise should drive more consistent exploration. We then analyze two potential problems that we face when applying parameter perturbations to policies that are implemented using deep neural networks. To quickly summarize: The first problem is due to the fact that we perturb a network with thousands of parameters that exhibit vastly different sensitivities with a noise process that is parameterized by just a single scalar scale parameter. We mitigate this first problem by re-parameterizing the neural network using layer normalization [Ba et al., 2016]. The second problem is concerned with picking a suitable scale for the perturbation noise. This is notoriously difficult since we cannot intuitively understand the effect of a perturbation in parameter space due to its high dimensionality and complex non-linear behavior. Instead, we propose to address the problem by relating the perturbation in parameter space into action space, where we can more easily reason about it. We can then use this information to adapt the scale of the perturbation such that a threshold value w.r.t. to the desired perturbation in action space is achieved. Using this insight, we devise an adaption scheme that rescales the noise magnitude over time, which we call *adaptive parameter space noise*.

To evaluate the performance of our proposed method, we first consider a simple toy example that specifically tests for the exploration capabilities of a reinforcement learning algorithm. On this task, we show that parameter space noise vastly outperforms ϵ-greedy action space noise exploration. Furthermore, we compare our approach against Bootstrapped DQN [Osband et al., 2016] and demonstrate that parameter space noise again compares favorably, although it is conceptually much simpler.

To demonstrate the effectiveness of our approach in more complex domains, we evaluate parameter space noise on 21 Atari games of varying complexity with dense and sparse

rewards. In our experiments, we clearly outperform the ϵ-greedy baseline in 8 out of these 21 games and perform comparably on the remaining ones. This demonstrates that parameter space noise is a suitable drop-in replacement for action space noise that works with state-of-the-art deep reinforcement learning algorithms like Deep Q-Networks (DQN, Mnih et al. [2015]). We further show that parameter space noise effectively utilizes the improved exploration behavior of Evolutionary Strategies (ES, Salimans et al. [2017]) while being multiple orders of magnitude more sample efficient.

Since we are in particular interested in robotics applications, we perform an exhaustive evaluation on 7 continuous control problems that are part of OpenAI Gym [Brockman et al., 2016] as well as 5 modified environments that exhibit a sparse reward structure and specifically test for exploration capabilities. We show that Deep Deterministic Policy Gradient (DDPG, Lillicrap et al. [2015]) can be combined with parameter space noise to solve the majority of these challenging domains. This is especially apparent on the sparse environments that were specifically selected to test for exploration, on which parameter space noise successfully learns on 3 out of 5 of these challenging tasks, whereas action space noise can only solve a single one.

Lastly, we consider some real-world robotics manipulation problems that are learned using only sparse rewards. We show that, again, parameter space noise alone is capable of learning in a complex pick & place environment where regular action space noise fails completely. Furthermore, we show that parameter space noise can be combined with Hindsight Experience Replay (HER, Andrychowicz et al. [2017]). This suggests that more sophisticated exploration methods can successfully combined with our approach, which typically improves convergence speed and stability.

In summary, we have shown that parameter space noise can be combined with a range of state-of-the-art deep reinforcement learning algorithms like DQN and DDPG. Parameter space noise can also be used in environments with discrete and continuous action spaces and effectively explores even in settings with extremely sparse rewards. Finally, parameter space noise can be combined with more sophisticated exploration methods like HER, which we believe makes it an important building block towards "solving" exploration. Implementation for DQN and DDPG have also been open-sourced.[1]

6.2 Future Work

There are multiple interesting extension of the currently proposed version of parameter space noise. First, it would be interesting to explore other adaption schema. In particular, it would be beneficial to learn a perturbation scale for each parameter. Caution must be exercised when doing so since the agent may simply "learn" to disable the perturbation noise such that it can always execute its current policy, which it believes to be optimal. If, however, the optimal scale of the perturbation could be learned, it could potentially result in further improved exploration behavior and would also avoid the devised adaption scheme proposed in this work.

More experiments that combine parameter space noise with more state-of-the-art reinforcement learning algorithms would also be interesting. For example, Proximal Policy Optimization (PPO, Schulman et al. [2017]) has recently achieved impressive results and was shown to be very stable. Additionally, there are a wide range of improvements over the original version of DQN (van Hasselt [2010], Schaul et al. [2015b], Wang et al. [2016]) and it would be interesting to evaluate how much parameter space noise benefits exploration on these architectures.

[1] `https://github.com/openai/baselines`

Finally, we have obtained promising results when combining parameter space noise with more sophisticated exploration methods like HER. It would be interesting to extend this analysis towards other exploration schema that still rely on some form of action space noise exploration. Examples for this are recent publications by Tang et al. [2016], Bellemare et al. [2016], and Pathak et al. [2017a], all of which introduce some novel idea but still require some form of action space exploration that could be replaced with parameter space noise.

Bibliography

J. Achiam and S. Sastry. Surprise-based intrinsic motivation for deep reinforcement learning. *CoRR*, abs/1703.01732, 2017. URL http://arxiv.org/abs/1703.01732.

M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba. Hindsight experience replay. *CoRR*, abs/1707.01495, 2017. URL http://arxiv.org/abs/1707.01495.

L. J. Ba, R. Kiros, and G. E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016. URL http://arxiv.org/abs/1607.06450.

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014. URL http://arxiv.org/abs/1409.0473.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013. doi: 10.1613/jair.3912. URL http://dx.doi.org/10.1613/jair.3912.

M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos. Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 1471–1479, 2016. URL http://papers.nips.cc/paper/6383-unifying-count-based-exploration-and-intrinsic-motivation.

D. P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987. ISBN 0132215810.

R. I. Brafman and M. Tennenholtz. R-MAX - A general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3:213–231, 2002. URL http://www.jmlr.org/papers/v3/brafman02a.html.

G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI gym. *arXiv preprint arXiv:1606.01540*, 2016. URL http://arxiv.org/abs/1606.01540.

Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel. Benchmarking deep reinforcement learning for continuous control. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 1329–1338, 2016. URL http://jmlr.org/proceedings/papers/v48/duan16.html.

M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg. Noisy networks for exploration. *CoRR*, abs/1706.10295, 2017. URL http://arxiv.org/abs/1706.10295.

Y. Gal and Z. Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 1050–1059, 2016. URL http://jmlr.org/proceedings/papers/v48/gal16.html.

T. Glasmachers, T. Schaul, and J. Schmidhuber. A natural evolution strategy for multi-objective optimization. In *Parallel Problem Solving from Nature - PPSN XI, 11th International Conference, Kraków, Poland, September 11-15, 2010, Proceedings, Part I*, pages 627–636, 2010a. doi: 10.1007/978-3-642-15844-5_63. URL https://doi.org/10.1007/978-3-642-15844-5_63.

T. Glasmachers, T. Schaul, Y. Sun, D. Wierstra, and J. Schmidhuber. Exponential natural evolution strategies. In *Genetic and Evolutionary Computation Conference, GECCO 2010, Proceedings, Portland, Oregon, USA, July 7-11, 2010*, pages 393–400, 2010b. doi: 10.1145/1830483.1830557. URL http://doi.acm.org/10.1145/1830483.1830557.

I. J. Goodfellow, Y. Bengio, and A. C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016. ISBN 978-0-262-03561-3. URL http://www.deeplearningbook.org/.

A. Graves, A. Mohamed, and G. E. Hinton. Speech recognition with deep recurrent neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*, pages 6645–6649, 2013. doi: 10.1109/ICASSP.2013.6638947. URL https://doi.org/10.1109/ICASSP.2013.6638947.

S. Gu, T. P. Lillicrap, I. Sutskever, and S. Levine. Continuous deep q-learning with model-based acceleration. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 2829–2838, 2016. URL http://jmlr.org/proceedings/papers/v48/gu16.html.

N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. M. A. Eslami, M. A. Riedmiller, and D. Silver. Emergence of locomotion behaviours in rich environments. *CoRR*, abs/1707.02286, 2017. URL http://arxiv.org/abs/1707.02286.

G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Process. Mag.*, 29(6):82–97, 2012. doi: 10.1109/MSP.2012.2205597. URL https://doi.org/10.1109/MSP.2012.2205597.

S. Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

R. Houthooft, X. Chen, X. Chen, Y. Duan, J. Schulman, F. D. Turck, and P. Abbeel. VIME: Variational information maximizing exploration. In *Advances in Neural Information Processing Systems 29 (NIPS)*, pages 1109–1117, 2016. URL http://papers.nips.cc/paper/6591-vime-variational-information-maximizing-exploration.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 448–456, 2015. URL http://jmlr.org/proceedings/papers/v37/ioffe15.html.

M. J. Kearns and S. P. Singh. Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49(2-3):209–232, 2002. doi: 10.1023/A:1017984413808. URL http://dx.doi.org/10.1023/A:1017984413808.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980.

J. Kober and J. Peters. Policy search for motor primitives in robotics. In *Advances in Neural Information Processing Systems 21 (NIPS)*, pages 849–856, 2008. URL http://papers.nips.cc/paper/3545-policy-search-for-motor-primitives-in-robotics.

J. Z. Kolter and A. Y. Ng. Near-bayesian exploration in polynomial time. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, pages 513–520, 2009. doi: 10.1145/1553374.1553441. URL http://doi.acm.org/10.1145/1553374.1553441.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, pages 1106–1114, 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.

P. R. Kumar and P. Varaiya. *Stochastic systems: Estimation, identification, and adaptive control*. Prentice-Hall, Inc., Englewood Cliffs, NJ, USA, 1986. ISBN 1611974259.

Y. LeCun, Y. Bengio, and G. E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015. doi: 10.1038/nature14539. URL https://doi.org/10.1038/nature14539.

S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *CoRR*, abs/1504.00702, 2015. URL http://arxiv.org/abs/1504.00702.

S. Levine, P. Pastor, A. Krizhevsky, and D. Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *CoRR*, abs/1603.02199, 2016. URL http://arxiv.org/abs/1603.02199.

T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015. URL http://arxiv.org/abs/1509.02971.

S. Linnainmaa. The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. *Master's Thesis (in Finnish), Univ. Helsinki*, pages 6–7, 1970.

A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proceedings of the International Conference on Machine Learning (ICML)*, volume 30, 2013.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. doi: 10.1038/nature14236. URL http://dx.doi.org/10.1038/nature14236.

V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 1928–1937, 2016. URL http://jmlr.org/proceedings/papers/v48/mniha16.html.

I. Osband, C. Blundell, A. Pritzel, and B. V. Roy. Deep exploration via bootstrapped DQN. In *Advances in Neural Information Processing Systems 29 (NIPS)*, pages 4026–4034, 2016. URL http://papers.nips.cc/paper/6501-deep-exploration-via-bootstrapped-dqn.

G. Ostrovski, M. G. Bellemare, A. van den Oord, and R. Munos. Count-based exploration with neural density models. *arXiv preprint arXiv:1703.01310*, 2017. URL http://arxiv.org/abs/1703.01310.

D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 2778–2787, 2017a. URL http://proceedings.mlr.press/v70/pathak17a.html.

D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 2778–2787, 2017b. URL http://proceedings.mlr.press/v70/pathak17a.html.

J. Peters and S. Schaal. Natural actor-critic. *Neurocomputing*, 71(7-9):1180–1190, 2008. doi: 10.1016/j.neucom.2007.11.026. URL http://dx.doi.org/10.1016/j.neucom.2007.11.026.

L. Pinto and A. Gupta. Supersizing self-supervision: Learning to grasp from 50k tries and 700 robot hours. In *2016 IEEE International Conference on Robotics and Automation, ICRA 2016, Stockholm, Sweden, May 16-21, 2016*, pages 3406–3413, 2016. doi: 10.1109/ICRA.2016.7487517. URL https://doi.org/10.1109/ICRA.2016.7487517.

L. Pinto, M. Andrychowicz, P. Welinder, W. Zaremba, and P. Abbeel. Asymmetric actor critic for image-based robot learning. *CoRR*, abs/1710.06542, 2017. URL http://arxiv.org/abs/1710.06542.

M. Plappert, R. Houthooft, P. Dhariwal, S. Sidor, R. Y. Chen, X. Chen, T. Asfour, P. Abbeel, and M. Andrychowicz. Parameter space noise for exploration. *CoRR*, abs/1706.01905, 2017. URL http://arxiv.org/abs/1706.01905.

A. Ranganathan. The Levenberg-Marquardt algorithm. *Tutoral on LM algorithm*, pages 1–5, 2004.

I. Rechenberg and M. Eigen. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologishen Evolution.* Frommann-Holzboog Stuttgart, 1973.

F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

T. Rückstieß, M. Felder, and J. Schmidhuber. State-dependent exploration for policy gradient methods. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases ECML/PKDD*, pages 234–249, 2008. doi: 10.1007/978-3-540-87481-2_16. URL http://dx.doi.org/10.1007/978-3-540-87481-2_16.

D. E. Rumelhart, G. E. Hinton, R. J. Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

T. Salimans, I. J. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen. Improved techniques for training gans. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2226–2234, 2016. URL http://papers.nips.cc/paper/6125-improved-techniques-for-training-gans.

T. Salimans, J. Ho, X. Chen, and I. Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017. URL http://arxiv.org/abs/1703.03864.

T. Schaul, T. Glasmachers, and J. Schmidhuber. High dimensions and heavy tails for natural evolution strategies. In *13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Proceedings, Dublin, Ireland, July 12-16, 2011*, pages 845–852, 2011. doi: 10.1145/2001576.2001692. URL http://doi.acm.org/10.1145/2001576.2001692.

T. Schaul, D. Horgan, K. Gregor, and D. Silver. Universal value function approximators. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 1312–1320, 2015a. URL http://jmlr.org/proceedings/papers/v37/schaul15.html.

T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015b. URL http://arxiv.org/abs/1511.05952.

J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 1889–1897, 2015a. URL http://jmlr.org/proceedings/papers/v37/schulman15.html.

J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 1889–1897, 2015b. URL http://jmlr.org/proceedings/papers/v37/schulman15.html.

J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *CoRR*, abs/1506.02438, 2015c. URL http://arxiv.org/abs/1506.02438.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

H.-P. Schwefel. *Numerische Optimierung von Computermodellen mittels der Evolutionsstrategie*, volume 1. Birkhäuser, Basel Switzerland, 1977.

F. Sehnke, C. Osendorfer, T. Rückstieß, A. Graves, J. Peters, and J. Schmidhuber. Parameter-exploring policy gradients. *Neural Networks*, 23(4):551–559, 2010. doi: 10.1016/j.neunet.2009.12.004. URL http://dx.doi.org/10.1016/j.neunet.2009.12.004.

P. Sermanet, S. Chintala, and Y. LeCun. Convolutional neural networks applied to house numbers digit classification. In *Proceedings of the 21st International Conference on Pattern Recognition, ICPR 2012, Tsukuba, Japan, November 11-15, 2012*, pages 3288–3291, 2012. URL http://ieeexplore.ieee.org/document/6460867/.

D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. A. Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, pages 387–395, 2014. URL http://jmlr.org/proceedings/papers/v32/silver14.html.

D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014. URL http://dl.acm.org/citation.cfm?id=2670313.

B. C. Stadie, S. Levine, and P. Abbeel. Incentivizing exploration in reinforcement learning with deep predictive models. *arXiv preprint arXiv:1507.00814*, 2015. URL http://arxiv.org/abs/1507.00814.

A. L. Strehl and M. L. Littman. An analysis of model-based interval estimation for markov decision processes. *J. Comput. Syst. Sci.*, 74(8):1309–1331, 2008. doi: 10.1016/j.jcss.2007.08.009. URL https://doi.org/10.1016/j.jcss.2007.08.009.

Y. Sun, D. Wierstra, T. Schaul, and J. Schmidhuber. Stochastic search using the natural gradient. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, pages 1161–1168, 2009a. doi: 10.1145/1553374.1553522. URL http://doi.acm.org/10.1145/1553374.1553522.

Y. Sun, D. Wierstra, T. Schaul, and J. Schmidhuber. Efficient natural evolution strategies. In *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*, pages 539–546, 2009b. doi: 10.1145/1569901.1569976. URL http://doi.acm.org/10.1145/1569901.1569976.

I. Sutskever, J. Martens, G. E. Dahl, and G. E. Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 1139–1147, 2013. URL http://jmlr.org/proceedings/papers/v28/sutskever13.html.

I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3104–3112, 2014. URL http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.

R. S. Sutton and A. G. Barto. *Introduction to reinforcement learning*, volume 135. MIT Press Cambridge, Cambridge, MA, USA, 1998. ISBN 9780262193986.

C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 4278–4284, 2017. URL http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14806.

H. Tang, R. Houthooft, D. Foote, A. Stooke, X. Chen, Y. Duan, J. Schulman, F. D. Turck, and P. Abbeel. #exploration: A study of count-based exploration for deep reinforcement learning. *CoRR*, abs/1611.04717, 2016. URL http://arxiv.org/abs/1611.04717.

W. R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

S. B. Thrun. Efficient exploration in reinforcement learning. Technical report, Pittsburgh, PA, USA, 1992.

J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. *CoRR*, abs/1703.06907, 2017. URL http://arxiv.org/abs/1703.06907.

E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2012, Vilamoura, Algarve, Portugal, October 7-12, 2012*, pages 5026–5033, 2012. doi: 10.1109/IROS.2012.6386109. URL http://dx.doi.org/10.1109/IROS.2012.6386109.

G. E. Uhlenbeck and L. S. Ornstein. On the theory of the brownian motion. *Phys. Rev.*, 36:823–841, Sep 1930. doi: 10.1103/PhysRev.36.823. URL https://link.aps.org/doi/10.1103/PhysRev.36.823.

A. van den Oord, N. Kalchbrenner, L. Espeholt, K. Kavukcuoglu, O. Vinyals, and A. Graves. Conditional image generation with pixelcnn decoders. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 4790–4798, 2016. URL http://papers.nips.cc/paper/6527-conditional-image-generation-with-pixelcnn-decoders.

H. van Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010. Proceedings of a meeting held 6-9 December 2010, Vancouver, British Columbia, Canada.*, pages 2613–2621, 2010. URL http://papers.nips.cc/paper/3964-double-q-learning.

H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 2094–2100, 2016. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12389.

Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas. Dueling network architectures for deep reinforcement learning. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 1995–2003, 2016. URL http://jmlr.org/proceedings/papers/v48/wangf16.html.

D. Wierstra, T. Schaul, T. Glasmachers, Y. Sun, J. Peters, and J. Schmidhuber. Natural evolution strategies. *Journal of Machine Learning Research*, 15(1):949–980, 2014. URL http://dl.acm.org/citation.cfm?id=2638566.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992. doi: 10.1007/BF00992696. URL http://dx.doi.org/10.1007/BF00992696.