UNSW AUSTRALIA

School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Adaptive Evolution Strategies

by

Xuefeng Li

Thesis submitted as a requirement for the degree of

Bachelor of Science (Honours) in Computer Science

Submitted:	June 2018	Student ID:	z5085453
Supervisor:	Alan Blair	Topic ID:	3834

Abstract

Evolution Strategies (ES) is a traditional black box optimization method which has recently gained renewed interest as a training method for neural networks, and as an alternative to Reinforcement Learning. For high dimensional problems, an isotropic Gaussian is normally used. Variants such as Natural ES or CMA-ES allow a more general covariance matrix but are computationally infeasible for large neural networks. In this thesis, we explore a novel variant of ES, comparable to Natural ES or CMA-ES, which is computationally efficient, allows parameters to be updated in an online manner, and can accommodate distributions with both diagonal and skewed covariance. We show that this algorithm can provide a statistically significant speedup for some MuJoCo and OpenAI Gym environments, and we discuss the effect of the various hyperparameter choices and algorithm components on the overall performance.

Acknowledgements

My supervisor Alan Blair for his supervision, guidance and patient explanations.

Joel Mason for his valuable suggestions.

Contents

1 Introduction **1**

2 Background **3**

2.1 Black-box Optimization 3

2.2 Evolution Strategies 4

2.2.1 Hill Climbing 5

2.2.2 Search Gradient 6

2.2.3 Limitation of Plain Search Gradient 7

2.2.4 Natural Evolution Strategy 9

2.2.5 OpenAI's Evolution Strategies 12

2.2.6 Adaptive Covariance 13

3 Method **15**

3.1 Proposed Algorithms 15

3.2 Techniques 19

3.2.1 Fitness Normalization 19

3.2.2 Variance Regularization 19

3.2.3 Virtual Batch Normalization 20

4 Implementation 21

4.1 Hardware 21

4.1.1 NCI Raijin 21

4.2 Software 22

4.2.1 Modules 22

4.2.2 Software Architecture 23

4.2.3 Parrallelization 24

5 Exerimental Results 26

5.1 Experimental Setup and Hyperparameters 27

5.1.1 Envrionments 27

5.1.2 Hyperparameters(need to update) 28

5.2 Results 31

6 Conclusion and Future Work 33

6.1 Conclusion 33

6.2 Future Work 34

Bibliography 35

List of Algorithms

1 General Procedure of Evolution Strategies 5
2 Evolution Strategies using Search Gradient 7
3 Evolution Strategies using Search Gradient 8
4 Natural Evolution Strategies 10
5 OpenAI Evolution Strategies 13

List of Figures

2.1 ES optimization process 4
2.2 Limitation of plain search gradient 9
2.3 Illustration of the effect of multiplying with the inverse of FIM 11

4.1 Software Architecture 23

5.1 Distributions updated with variant algorithms 26
5.2 Selected Environments 27
5.3 Distribution of stopping times for Algorithm 1 28
5.4 Distribution of stopping times for Algorithm 1 29
5.5 Distribution of stopping times for Algorithm 2 30
5.6 Distribution of stopping times for Algorithm 3 30
5.7 Results for Half Cheetah 31
5.8 Results for Pendulum 32
5.9 Results for Inverted Double Pendulum 32

Chapter 1

Introduction

There has recently been an increasing interest in the use of evolution inspired optimization methods for tasks traditionally solved using Reinforcement Learning (RL). Evolution Strategies (ES) have been shown in (Salimans et al., 2017) to produce comparable results to standard policy gradient-based methods. Furthermore, despite the higher sample complexity, its amenability to efficient parallelization means that ES can be quick to train in terms of wall-clock time, given sufficient computational resources. Additionally, ES has a number of advantages over traditional RL algorithms, namely insensitivity to sparse and delayed rewards, and the ability to accommodate non-differentiable elements in a policy network.

We explore a novel and efficient variant of ES, which can adjust the parameters of a diagonal covariance matrix and can also accommodate skewed distributions over the parameter space, i.e. those with non-diagonal covariance. This technique uses two matrices of lower rank than the number of parameters, which stretch and compress the distribution in specified directions.

In our experiments, we conduct an ablation study of the technique, training on the MuJoCo (Todorov et al., 2012) environments using OpenAI Gym (Brockman et al., 2016). We examine results when the variance of each dimension of the parameter space is fixed, allowed to vary, both with and without the low-rank covariance approximation. We also analyze the effect of various hyperparameter choices on the effectiveness of a number of ES variants.

Our results show the adaptive covariance of the proposed algorithm is more effective than ES with non-adaptive covariance.

Chapter 2

Background

In this chapter, we give an introduction to the background knowledge for this thesis. Starting with an overview of ES which belongs to a general class of black box optimization. Then we introduce several variants of ES.

2.1 Black-box Optimization

Black-box optimization is a family of numerical optimization methods that aims to find the best solution to some optimization problems without gradient information. Formally, black-box optimization is defined as finding a solution x in a search space $\boldsymbol{X}$:

$$x = \arg\min_{x \in \boldsymbol{X}} f(x), \tag{2.1}$$

where f(x) is called the objective function.

Black-box optimization algorithms were originally created for solving optimization problems where objective functions are unknown, and we can observe the inputs and the outcomes. The objective function is like a black box, hence it is named black-box optimization. As it does not rely on the gradient information, it can be used on func-

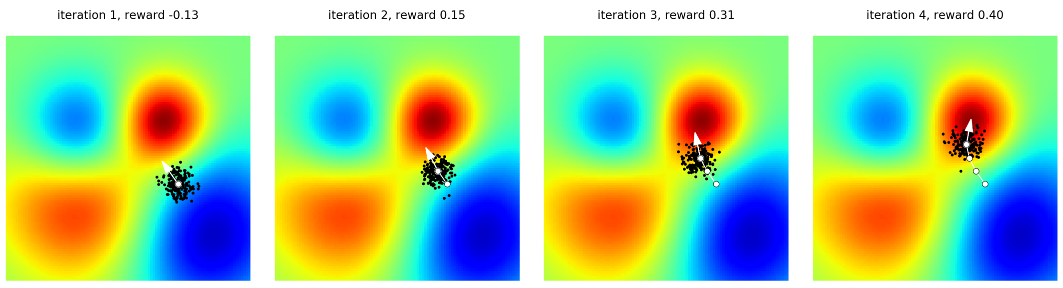


Figure 2.1: ES optimization process: At each iteration, a population of perturbed parameters (in black) sampled from the current parameters(in white), and we move the parameters to the direction(white arrow) where the perturbed parameters achieved higher fitnesses.

tions that are not continuous or differentiable.

In black-box optimization, the function evaluation and parameter updates are normally computationally expensive. Hence, we need to take these two elements into account when designing a black-box optimization algorithm. In addition, the algorithm needs to have the ability to find a solution as good as possible. To evaluate different black-box optimization algorithms, there are mainly three criteria: 1) Number of function evaluations or parameter updates for searching the optimum. 2) Computation time for searching the optimum. 3) The difference between the fitness obtained with the final solution and the optimum if it is known in advance.

The experimental framework of this thesis is based on these three criteria.

2.2 Evolution Strategies

ES is an optimization algorithm inspired by natural evolution. It was first proposed in (Rechenberg, 1973) and then developed in (Schwefel, 1993). In addition, it has many variants such as CMA-ES Hansen et al. (1995), NES Wierstra et al. (2014) which are successful in solving optimization problems. The general procedure of ES can be described as follows: At each iteration, a population of sample parameters is generated by adding noise to the current parameters and all the sample parameters are evaluated. Then the current parameters are updated based on these sampled evaluations to achieve a higher expectation of fitnesses. This procedure is iterated until meeting the stopping criteria. The algorithm is summarized in Algorithm 1.

Algorithm 1 General Procedure of Evolution Strategies

```
1: Initilization
2: repeat
3:     Parameter sampling
4:     Evaluation
5:     Parameter updating
6: until Stopping criteria is met
```

2.2.1 Hill Climbing

Hill climbing is an optimization technique similar with ES except lacking the concept of population. The basic idea of hill climbing is that starting with a random set of parameters, generate perturbed parameters at each iteration to find a better solution. If the perturbation generates a better solution, the perturbation will be kept as the new parameters. Then repeat this procedure until no better solution is found with further perturbing. The simplicity of this idea makes it popular in optimization research and has been used widely for solving game problems for decades.

(Pollack and Blair, 1998) applied the simple hill climbing in learning the game of backgammon. In their work, a standard feed-forward neural network was used as a function approximator with the board information as input and all legal moves as output. The learning procedure started with all weights of the neural network set to zero. Then at each iteration, Gaussian noise was added to the weights and let the network play against the mutant for a number of games. If the mutant wins more than half the games, select it for the next generation.

Their work showed the simple hill climbing can achieve reasonable performance at the early stage. However, there was a limitation of hill climbing which they called Buster Douglas Effect. Due to the randomness of the game, the selected perturbation may just be some lucky novice. This may make the perturbing generation move too far from the previous generation, which means forgetting the learned weights and causes the catastrophic effect on performance.

For avoiding the Buster Douglas Effect, rather than replacing the champion by the challenger, they instead make only a small adjustment in the direction of (challenger - champion):

$$champion = champion + \alpha(challenger - champion), \tag{2.2}$$

where α is the learning rate.

At each iteration, by moving a small step to the challenger, most of the current decisions are preserved thus avoiding the catastrophic replacement by a lucky novice challenger.

2.2.2 Search Gradient

The idea of Search Gradient was introduced in (Berny, 2000) and (Berny, 2001). Assume we have a fitness function $F(x)$ we want to maximize, θ is the parameters vector which defines the search distribution $\pi(x|\theta)$. At each iteration, a population of mutants is generated from this distribution and evaluated. Instead of moving a small step to the mutants with higher fitness, search gradient is computed as the sampled gradient of expected fitness. The expected fitness under the search distribution can be written as

$$J(\boldsymbol{\theta}) = E_p[f(\boldsymbol{x})] = \int f(\boldsymbol{x})p(\boldsymbol{x}|\boldsymbol{\theta})d\boldsymbol{x} \tag{2.3}$$

Taking the derivative gives

$$\begin{aligned}
\nabla_\theta J(\boldsymbol{\theta}) = \nabla_\theta E_p[f(\boldsymbol{x})] &= \nabla_\theta \int f(\boldsymbol{x}) p(\boldsymbol{x}|\boldsymbol{\theta}) d\boldsymbol{x} \\
&= \int f(\boldsymbol{x}) \nabla_\theta p(\boldsymbol{x}|\boldsymbol{\theta}) d\boldsymbol{x} \\
&= \int f(\boldsymbol{x}) \nabla_\theta p(\boldsymbol{x}|\boldsymbol{\theta}) \frac{p(\boldsymbol{x}|\boldsymbol{\theta})}{p(\boldsymbol{x}|\boldsymbol{\theta})} d\boldsymbol{x} \\
&= \int [f(\boldsymbol{x}) \nabla_\theta \log p(\boldsymbol{x}|\boldsymbol{\theta})] p(\boldsymbol{x}|\boldsymbol{\theta}) d\boldsymbol{x} \\
&= E_\theta[f(\boldsymbol{x}) \nabla_\theta \log p(\boldsymbol{x}|\boldsymbol{\theta})] \qquad (2.4)
\end{aligned}$$

This enables us to use Monte Carlo methods to estimate the gradient and update the parameters using gradient ascent:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \nabla_\theta J(\boldsymbol{\theta}), \qquad (2.5)$$

where η it a learning rate parameter. Updating the parameters with the search gradient is the core idea of ES. Algorithm 2 shows how ES uses the search gradient to do the optimization.

Algorithm 2 Evolution Strategies using Search Gradient

```
1: Input: Learning rate $\eta$, distribution $p(x|\theta)$, initial policy parameters $\theta_0$
2: for $t = 0, 1, 2, \ldots, n$ do
3:     Sample $x_1, \ldots, x_n \sim p(x|\theta)$
4:     Evaluate the fitnesses $f(x_i)$ for $i = 1, \ldots, n$
5:     Calculate log-derivatives $\nabla_\theta \log p(x_i|\theta)$ for $i = 1, \ldots, n$
6:     $\theta_{t+1} \leftarrow \theta_t + \eta \cdot \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \log p(x_i|\theta) \cdot f(x_i)$
7: end for
```

In practice, the most popular search distribution is the multinormal distribution. Then the algorithm can be written as in Algorithm 3.

2.2.3 Limitation of Plain Search Gradient

By updating with plain search gradient, we are actually updating the parameters in the direction of the gradient widely believed it is also the steepest direction. However,

Algorithm 3 Evolution Strategies using Search Gradient

```
1: Input: Learning rate $\eta$, multinormal distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$
2: for $t = 0, 1, 2, \ldots, n$ do
3:     Sample $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$
4:     Evaluate the fitnesses $f(\boldsymbol{x_i})$ for $i = 1, \ldots, n$
5:     Calculate log-derivatives :
6:       $\nabla_{\boldsymbol{\mu}} \log \mathcal{N}(\boldsymbol{x_i}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}_i - \boldsymbol{\mu})$ for $i = 1, \ldots, n$
7:       $\nabla_{\boldsymbol{\Sigma}} \log \mathcal{N}(x_i|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{1}{2}\boldsymbol{\Sigma}^{-1} + \frac{1}{2}\boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i} - \boldsymbol{\mu})(\boldsymbol{x_i} - \boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}$ for $i = 1, \ldots, n$
8:     $\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} + \eta \cdot \frac{1}{n}\sum_{i=1}^{n} \nabla_{\boldsymbol{\mu}} \log \mathcal{N}(\boldsymbol{x_i}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \cdot f(\boldsymbol{x}_i)$
9:     $\boldsymbol{\Sigma} \leftarrow \boldsymbol{\Sigma} + \eta \cdot \frac{1}{n}\sum_{i=1}^{n} \nabla_{\boldsymbol{\Sigma}} \log \mathcal{N}(\boldsymbol{x_i}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \cdot f(\boldsymbol{x}_i)$
10: end for
```

this is true only when an orthonormal coordinate system is used in an Euclidean space. Obviously, the parameter space here is not Euclidean but Riemannian. Hence, applying plain search gradient may cause catastrophic effects in learning.

To see why, suppose we have an one-dimensional normal distribution $x \sim \mathcal{N}(\mu, \sigma)$, the search gradients then become

$$\nabla_\mu F = \frac{x - \mu}{\sigma^2}, \tag{2.6}$$

$$\nabla_\sigma F = \frac{(x - \mu)^2 - \sigma^2}{\sigma^3} \tag{2.7}$$

To let the distribution converge to a minimum, σ must decrease as the parameters updating. However, updates of μ and σ both depend on σ. This correlation increases the variance of the updates. We can see this by treating σ as part of the learning rate, if σ is very small, the learning rate will become very large and cause overshooting updates. But if σ is very large, the update step will become quite small and slow the convergence procedure. This makes ES with plain search gradient unstable and hard to converge.

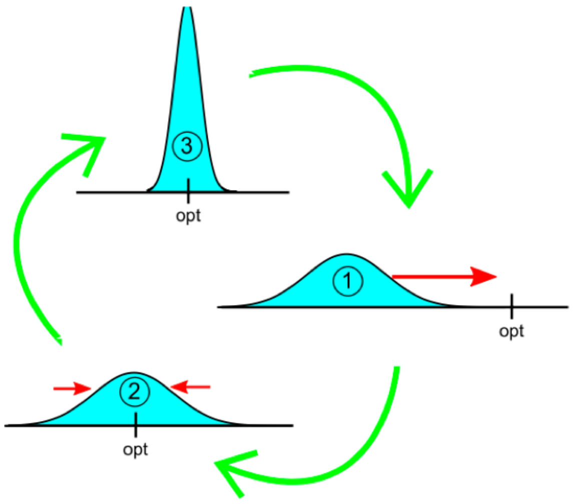


Figure 2.2: Illustration of the limitation: from (1) to (2), μ is adjusted to make the distribution cover the optimum. From (2) to (3), σ is reduced to allow for a precise localization of the optimum. But then from (3), the small σ causes a largely overshooting update, leading to (1) again. Figure from (Wierstra et al., 2014)

2.2.4 Natural Evolution Strategy

To avoid the shortcomings of plain search gradient, Natural Evolution Strategies(NES) (Wierstra et al., 2014) uses the natural gradient(Amari, 1998) which is a part of the information geometry (Amari and Nagaoka, 2007) to do the updates. In information geometry, the Fisher information matrix (FIM) is regarded as the Riemannian metric on the statistical manifold, and then the direction of the steepest descent on the manifold is used as the search direction, which is the natural gradient. In ES, the parameter space is a Riemannian manifold and each point in the parameter space denotes a search distribution, hence the natural gradient is the steepest search direction.

Recall that the aim of ES is to maximize the expected fitness $J(\boldsymbol{\theta})$, the natural gradient can also be seen as the solution to the constrained optimization problem:

$$\max_{\delta\theta} J(\theta + \delta\theta) \approx J(\theta) + \delta\theta^T \nabla_{J_\theta}, \tag{2.8}$$

$$s.t. D(\theta + \delta\theta || \theta) = \epsilon, \tag{2.9}$$

where $D(\theta + \delta\theta||\theta)$ is the Kullback-Leibler divergence between distribution $p(x|\theta)$ and

$p(x|\theta + \delta\theta)$ and ϵ is a small constant. Solving this constrained optimization problem with Lagrangian multiplier gives

$$\widetilde{\nabla}_\theta J = \boldsymbol{F}^{-1}\nabla_\theta J(\theta) \tag{2.10}$$

where

$$F = \int p(x|\theta)\nabla_\theta \log p(x|\theta)\nabla_\theta \log p(x|\theta)^T dx \tag{2.11}$$

$$= E[\nabla_\theta \log p(x|\theta)\nabla_\theta \log p(x|\theta)^T] \tag{2.12}$$

is the FIM of the search distribution. From the equation we can see that the FIM can be estimated by sampling, using the derivatives $\nabla_\theta \log p(x|\theta)$ which are already computed in $\nabla_\theta J(\theta)$. The NES algorithm is summarized in algorithm 4.

Algorithm 4 Natural Evolution Strategies

```
1: Input: Learning rate $\eta$, distribution $p(x|\theta)$, initial policy parameters $\theta_0$
2: for $t = 0, 1, 2, ...n$ do
3:     Sample $x_1, \ldots, x_n \sim p(x|\theta)$
4:     Evaluate the fitnesses $f(x_i)$ for $i = 1, \ldots, n$
5:     Calculate log-derivatives $\nabla_\theta \log p(x_i|\theta)$ for $i = 1, \ldots, n$
6:     $\nabla_\theta J \leftarrow \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta \log p(x_i|\theta) \cdot f(x_i)$
7:     $\boldsymbol{F} \leftarrow \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta \log p(x_i|\theta)\nabla_\theta \log p(x_i|\theta)^T$
8:     $\theta_{t+1} \leftarrow \theta_t + \eta \cdot \boldsymbol{F}^{-1}\nabla_\theta J$
9: end for
```

While NES solves the limitations of plain search gradient, since the update at each iteration involves calculating the inverse of FIM, the complexity greatly increases to $\mathcal{O}(n^3)$. This leads to the infeasibility of applying it for solving high-dimensional problems like training neural networks.

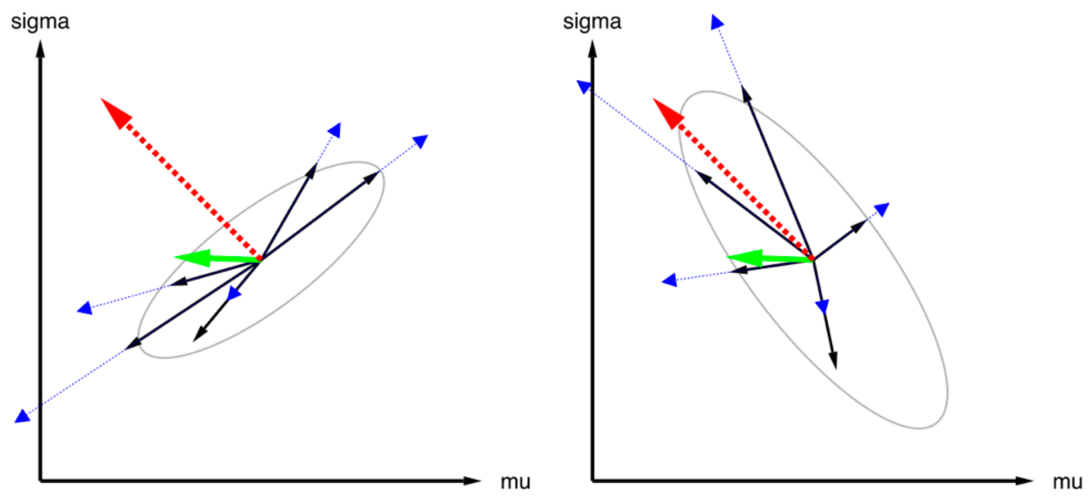


Figure 2.3: Illustration of the effect of multiplying with the inverse of FIM. We have two parameters for one dimensional Gaussian, $\theta = (\mu, \sigma)$. On the left, the solid black arrows indicate the gradient samples, while the blue dotted arrows are the gradient estimates scaled with fitness. The green bold arrow indicates the estimated fitness gradient, while the bold dashed (red) arrow indicates the corresponding natural gradient.
The gray ellipse indicates the covariance of gradient samples, multiply the inverse of FIM to get natural gradient here is equivalent to multiply with the inverse of the covariance of the gradient samples. This means gradient samples in the direction with high variance will be compressed while those in the direction with low variance will be enlarged. The right figure shows the effect of multiplying the inverse of the covariance of sampled gradients. Figure from (Wierstra et al., 2014)

2.2.5 OpenAI's Evolution Strategies

ES as a black box optimization algorithm have several attractive advantages: It is easy for implementation, highly parallelizable and can be applied to various problems. However, these black box optimization algorithms are criticized they are inefficient compared with gradient-based optimization algorithm like stochastic gradient descent. Besides, for solving reinforcement problems, it is widely believed that reinforcement learning is a more efficient method in terms of timing and data than ES, especially for complicated problems. These disadvantages make ES neglected for a long time.

A recent work of OpenAI (Salimans et al., 2017) renewed people's interest in ES by successfully training neural networks for solving a variety of reinforcement learning problems with ES.

To overcome the limitations of plain search gradient while not greatly increases the complexity, they make a compromise by using non-adaptive isotropic multi-normal distribution as the search distribution. This means the covariance matrix of the search distribution is a non-adaptive diagonal matrix. Hence, at each iteration, only the mean of the search distribution needs to be updated. Since the variance is fixed, the gradient will not explode during training and thus stabilizing the process of convergence.

Another important factor for the success of OpenAI's work is the high parallelizability of ES. As pointed out in their work, there are three main properties which make ES highly parallelizable: 1) It only updates after complete episodes, thus less communication between workers. 2) Each worker only needs to send/receive scalar fitnesses to/from other workers provided the workers can reconstruct the parameters of other workers. 3) ES does not require value function approximations. The algorithm is showed in Alogirithm 5.

Algorithm 5 OpenAI Evolution Strategies

```
1: Input: Learning rate $\eta$, noise standard deviation $\sigma$, initial policy parameters $\theta_0$
2: Initialize: $n$ workers with known random seeds, and initial parameters $\theta_0$
3: for $t = 0, 1, 2, ...n$ do
4:     for each worker $i = 1, \dots, n$ do
5:         Sample $\epsilon_i \sim \mathcal{N}(0, I)$
6:         Get fitnesses $F_i = F(\theta_t + \sigma\epsilon_i)$
7:     end for
8:     Send all scalar returns $F_i$ from each worker to every other worker
9:     for each worker $i = 1, ..., n$ do
10:        Reconstruct all perturbations $\epsilon_j$ for $j = 1, \dots, n$
11:        Set $\theta_{t+1} \leftarrow \theta_t + \alpha\frac{1}{n\sigma}\sum_{j=1}^{n} F_j\epsilon_j$
12:    end for
13: end for
```

According to OpenAI's experiments, ES can achieve competitive results on most Atari games after one hour of training time and solve dynamic control problem in 10 minutes with hundreds to thousands of parallel workers.

2.2.6 Adaptive Covariance

Even if OpenAI's work has achieved good results with diagonal non-adaptive covariance, there are still some drawbacks of non-adaptive covariance need to notice: 1) ES with non-adaptive covariance tends to converge to a local minima with smaller curvature compared to the global minima but with bigger curvature. 2) Adaptive covariance will accelerate the speed of convergence by stretching the distribution.

To illustrate why non-adaptive covariance tends to converge to local minimum, recall in ES we are optimizing the expected fitness $E_p[f(\boldsymbol{x})] = \int f(\boldsymbol{x})p(\boldsymbol{x}|\boldsymbol{\theta})d\boldsymbol{x}$. If we are doing the optimization in a fitness landscape with two regions, A with the global minimum but has a bigger curvature while B has a local minimum but has a smaller curvature.

Even the smallest $f(\boldsymbol{x})$ appears in A, the expected fitness in A may smaller than the expected fitness in B, thus leading to the convergence to the local minimum.

Adaptive covariance can accelerate the process of convergence by stretching the sample distribution in the direction with high fitnesses and squash in the direction with low fitnesses. This is similar with adjusting the learning rate in different directions. This also implies ES with non-diagonal covariance will achieve faster convergence than with diagonal covariance.

Chapter 3

Method

As discussed in the previous chapter, non-diagonal adaptive covariance can accelerate the process of convergence. However, with plain search gradient, the convergence is unstable while using NES and other variants of ES is computationally infeasible for high-dimensional problems. Here, we introduce the algorithm recently developed by Alan Blair. It is designed to be more computationally efficient compared with NES or CMA-ES and allow non-diagonal covariance for high dimensional problems.

3.1 Proposed Algorithms

Sampling

For sampling at each generation, we use a Gaussian distribution constructed by taking a Gaussian distribution with n-dimensional covariance matrix and then stretching or compressing the sample in a smaller number of dimensions $s+r$, where s is the number of stretching vectors and r is the number of compressing vectors.

In practice, we first choose a random vector x from $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma})$ by generating random variables $\{\varepsilon_i\}_{1\leq i\leq n}$ from a standard normal distribution $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$ and setting $x = \mathcal{S}(\epsilon)$

given by

$$x_i = \mu_i + \sum_{i=1}^{n} \varepsilon_i \sigma_i \hat{x}^i$$

where ε_i are treated as *latent* (or scale-free) parameters, and $\hat{x}^i$ denotes the vector whose i^{th} component is 1 and all other components are 0.

Then we rescale the sample with the rescaling function $\mathcal{R} : \varepsilon \mapsto z$. It is determined by an orthonormal basis $\{\hat{u}^{(j)}\}_{1 \leq j \leq r+s}$ and a rescaling factor $\rho_{(j)}$ in each direction.

$$z = \mathcal{R}(\varepsilon) = \varepsilon + \sum_{j=1}^{r+s} (\rho_{(j)} - 1) < \varepsilon \,|\, \hat{u}^{(j)} > \hat{u}^{(j)}$$

Finally, sample points x are obtained from the latent variables z by

$$x = \mu + \sigma\, z$$

Update the Mean of the Distribution

The generated sample x is evaluated by the objective function f(x). Then we use the fitness to update the parameters of the search distribution. Our algorithm is similar with the plain search gradient descent, but with an implicit reparameterization of μ and σ. If we were to apply gradient descent directly on μ the updates would take the form

$$\mu_i \leftarrow \mu_i + \alpha_\mu \, \tilde{f}(x) \frac{(x_i - \mu_i)}{\sigma_i^2} \tag{3.1}$$

However, as pointed out in NES by Wierstra et al. (2014), this approach is numerically unstable and generally leads to poor results — essentially because the magnitude of the updates to μ_i scale inversely with the standard deviation σ_i in the corresponding direction.

Instead, we adopt a simpler and arguably more natural approach of updating μ explicitly in the direction of $(x-\mu)$:

$$\mu_i \leftarrow \mu_i + \alpha_\mu \tilde{f}(x)(x_i - \mu_i) \tag{3.2}$$

One way to view this is as an implicit rescaling of μ. If we denote by $z_i^{(\mu)}$ the number of standard deviations between 0 and μ in the i^{th} dimension (i.e. $\mu_i = \sigma_i z_i^{(\mu)}$) then it is essentially the same as performing gradient descent with respect to $z_i^{(\mu)}$ rather than μ_i (except that μ_i does not change when σ_i is updated, as it would if σ_i and $z_i^{(\mu)}$ were treated as truly independent variables).

Update the Standard Deviation

Instead of updating $\boldsymbol{\sigma}$ with plain search gradient

$$\sigma_i \leftarrow \sigma_i + \alpha_\sigma f(x)\frac{(\varepsilon_i^2 - 1)}{\sigma_i}, \tag{3.3}$$

we need to do explicit regularization to make the updates numerically stable. We parameterize σ as:

$$\sigma_i = \log(e_i^\kappa + 1), \tag{3.4}$$

then apply gradient descent to κ_i instead of σ_i to do the updates. Updates are computed by the midpoint method to keep positive and negative σ updates balanced. When σ is small, we have

$$\frac{d\sigma_i}{d\kappa_i} = \exp(\kappa_i - \sigma_i) \simeq \sigma_i$$

According to the chain rule, equation 3.6 is equivalent to:

$$\kappa_i \leftarrow \kappa_i + \alpha_\kappa f(x)(\varepsilon_i^2 - 1), \tag{3.5}$$

In other words, like the μ updates, the magnitude of the σ updates in each dimension will be roughly proportional to σ_i in that dimension. In addition, when σ_i is big, $\frac{d\sigma_i}{d\kappa_i}$ is approximately constant, thus preventing any exponential blow-up in the size of σ_i.

Update rescaling factor and orthonormal basis

The rescaling factors $\rho_{(j)}$ can be updated in the same manner as the σ We turn now to the updating of the scaling vectors $\{\hat{u}_{(j)}\}$.

For convenience, we divide the vectors $\{\hat{u}_{(j)}\}_{1\leq j\leq r+s}$ into two groups $\{\hat{u}_{(j)}\}_{1\leq j\leq r}$ and $\{\hat{v}_{(k)}\}_{1\leq k\leq s}$. Our intention is the algorithm should try to identify certain directions $\{\hat{u}_{(j)}\}$ in which the distribution needs to be stretched, and other directions $\{\hat{v}_{(k)}\}$ in which the distribution needs to be compressed. In other words, $\{\hat{u}_{(j)}\}$ should span a subspace near which $f(x)$ is high while $\{\hat{v}_{(k)}\}$ spans an orthogonal subspace near which $f(x)$ is low. Every sample z from $\mathcal{N}_{\hat{u},\rho}$ can be uniquely decomposed as

$$z = z^{\overset{\parallel}{u}} + z^{\overset{\perp}{u}}$$

where $z^{\overset{\parallel}{u}}$ lies in the subspace spanned by $\{\hat{u}^{(j)}\}$. Because $\int_z f(\mathcal{S}_0(z))\,\mathcal{N}_{\hat{u},\rho}(z)\,dz = 0$, maximizing f near the subspace is equivalent to *minimizing* it away from the subspace. With this in mind, we seek to adjust the vectors $\{\hat{u}^{(j)}\}$ in such a way as to minimize

$$\int_z f(\mathcal{S}_0(z))\|z^{\overset{\perp}{u}}\|\,\mathcal{N}_{\hat{u},\rho}(z)\,dz$$

Note that $\|z^{\overset{\perp}{u}}\| = \|z\|\sin(\theta_{z,u})$ where $\theta_{z,u}$ is the angle between $z^{\overset{\perp}{u}}$ and $z^{\overset{\parallel}{u}}$. Applying gradient descent to this angle (with learning rate α_u) we rotate the orthonormal frame $\{\hat{u}^{(j)}\}$ by an amount proportional to

$$\alpha_u f(\mathcal{S}_0(z))\|z^{\overset{\parallel}{u}}\|\cos(\theta_{z,u}) = \alpha_u f(\mathcal{S}_0(z))\|z^{\overset{\parallel}{u}}\|$$

in the direction which most efficiently reduces $\|z^{\overset{\perp}{u}}\|$.

The update for $\{\hat{v}^{(k)}\}$ is similar, except that, to keep $\{\hat{v}^{(k)}\}$ orthogonal to $\{\hat{u}^{(j)}\}$, we decompose $(z - z^{\overset{\parallel}{u}})$ as

$$z - z^{\overset{\parallel}{u}} = z^{\overset{\parallel}{v}} + z^{\overset{\perp}{v}}$$

We then adjust the angle between $(z - z^{\overset{\parallel}{u}})$ and $z^{\overset{\parallel}{v}}$ so as to maximize

$$\int_z f(\mathcal{S}_0(x))\,\|z^{\overset{\perp}{v}}\|\,\mathcal{N}_{\hat{u},\rho}(z)dz$$

Although this algorithm keeps the basis vectors orthonormal to first order, an explicit Gram-Schmidt orthonormalization is required periodically (roughly, every 100 iterations) at a cost of $\mathcal{O}\big(((r+s)^2 n\big)$. Apart from this, the computational cost at each iteration is $\mathcal{O}\big((r+s)n\big)$.

3.2 Techniques

In this section we will introduce several techniques we used to improve our algorithms' performance and robustness. Fitness normalization is used to achieve an appropriate scaling of the fitnesses. Variance regularization is used to avoid the over shrinking of the search distribution which leads to insufficient exploration thus converging to local minimums. Virtual batch normalization is used for having better exploration at the early stages of training in some of our testing environments.

3.2.1 Fitness Normalization

To achieve an appropriate scaling of fitnesses f, as well as variance reduction, we make use of running averages to maintain estimates F_{mean} and F_{var} of the mean and variance of f over our current distribution $q(x)$, and scale the current fitness $f(x)$ to

$$\tilde{f}(x) = (f(x) - F_{\text{mean}})/\sqrt{F_{\text{var}}} \tag{3.6}$$

This is related to fitness shaping used in NES (Wierstra et al., 2014). However, our method can achieve a more accurate measure of fitnesses while keep the algorithm invariant to fitness transformation.

3.2.2 Variance Regularization

ES tends to shrink the distribution in some directions to have a higher expected fitness. If the variance gets too small during the process, it will lead to poor explorations and converging to local minimums. To avoid this, we initialize the standard deviation σ_0

and set $\mathcal{N}_{\mu,\sigma_0}$ as our "prior" distribution. Then our aim is to find a distribution $q(x)$ for which $\int_x f(x)\, q(x)dx$ is as large as possible, but does not differ too greatly from the "prior" distribution $\mathcal{N}_{\mu,\sigma_0}$ with the same mean μ and initial standard deviation σ_0 in each direction. Then the function we seek to maximize becomes: We therefore we seek to maximize

$$\int_x \tilde{f}(x)\, q(x)dx + \lambda_\sigma \mathrm{D}_{\mathrm{KL}}(q, \mathcal{N}_{\mu,\sigma_0}), \tag{3.7}$$

where $\mathrm{D}_{\mathrm{KL}}(q, \mathcal{N}_{\mu,\sigma_0})$ is the Külback-Leibler divergence between $q = \mathcal{N}_{\mu,\sigma}$ and $\mathcal{N}_{\mu,\sigma_0}$.

We also apply it on the scaling factor ρ, then the equation becomes:

$$\int_x \tilde{f}(x)\, q(x)dx + \lambda_\sigma \mathrm{D}_{\mathrm{KL}}(q, \mathcal{N}_{\mu,\sigma_0}) + \mathrm{D}_{\mathrm{KL}}(\mathcal{N}_{\hat{u},\rho}, \mathcal{N}_{\hat{u},1}) \tag{3.8}$$

The last two terms can be seen as the regularizer for σ and ρ with Külback-Leibler divergence, this is similar with weight decay but using different metric.

3.2.3 Virtual Batch Normalization

As pointed out by (Salimans et al., 2017), for some environments, Gaussian parameter perturbations did not always lead to adequate exploration. For some environments, the state is dominated by some elements which are not important like the background colour. In this situation randomly perturbed parameters tends to encode policies that always took one specific action. This can be solved by using virtual batch normalization (Salimans et al., 2016).

Virtual batch normalization is similar with batch normalization (Ioffe and Szegedy, 2015) except that the normalization is based on the statistics collected on a reference batch of examples that are chosen once and fixed at the start of training. Where the mini-batch used for calculating normalizing statistics is chosen at the start of training and is fixed. This makes the policy more sensitive to small changes in the input at the early stages of training when the weights of the policy are random, ensuring that different policies will have different fitnesses.

Chapter 4

Implementation

Our algorithm is highly parallelized thus need large amounts of computational resources. We will introduce the support hardware we use from National Computational Infrastructure (NCI). And explain our software architecture and parallelization plan.

4.1 Hardware

We use our local computing device for neural network architectures design and testing while use supercomputers raijin from NCI for more computationally intensive problems.

4.1.1 NCI Raijin

Raijin is a hybrid Fujitsu Primergy and Lenovo NeXtScale high-performance, distributed-memory cluster procured with funding from the Australian Government. It compromises: 1) 84,656 cores (Intel Xeon Sandy Bridge 2.6 GHz, Broadwell 2.6 GHz) in 4416 compute nodes. 2) 120 NVIDIA Tesla K80 GPUs in 30 nodes and 8 NVIDIA Tesla P100 GPUs in 2 nodes 3) 32 Intel Xeon Phi (64 core Knights Landing, 1.3 GHz) in 32 compute nodes. 4) 300 Terabytes of main memory. 5) 8 Petabytes of high-performance operational storage capacity.

For our experiments, we use only CPUs in order to scale our algorithm. Depends on the environments, we use 4/8/16 nodes, each node has 16 CPU cores, to do function evaluation and parameters updating.

4.2 Software

4.2.1 Modules

OpenAI Gym is a toolkit providing benchmarks for comparing artificial intelligence algorithms. It has a range of game environments including classic control problems, Atari games and board games. It is compatible with any numerical computation library including TensorFlow. For this thesis, we use various environments including Atari games like pong, control problems such as pendulum etc.

Tensorflow is an open source numerical computation software library using data flow graphs. The nodes in the graph represents the math operation which the edge in the graph represent the multi-dimensional tensor between nodes. TensorFlow can run on multiple CPUs and GPUs and it is widely used for building and training neural networks. For this thesis, we use Tensorflow to build our neural network models and do the inference.

Ctypes module is a built-in function module used by Python to call dynamic link library functions. It can be used for mixed programming of Python and other languages. For this thesis, we use ctypes to do the communication between C and Python by sharing library functions and global variables.

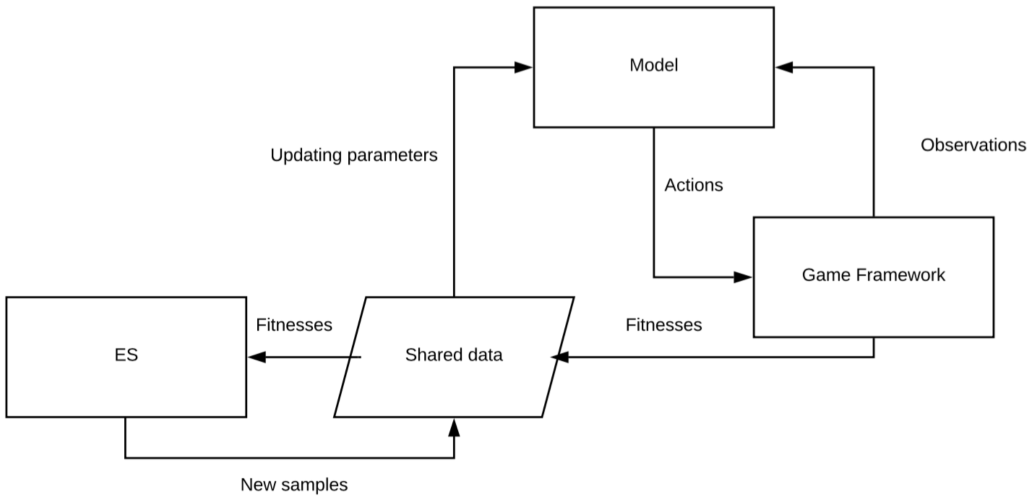


Figure 4.1: Software Architecture

Mpi4py is a Python library built on top of MPI. It allows Python's data structure to be easily passed in multiple processes. More specifically, Mpi4py is a powerful library that implements many of the MPI standard's interfaces, including peer-to-peer communication, group-wide communication, inter-group communication, etc. In addition, it also provides SWIG and F2PY interfaces that allow us to use our mpi4py objects and interfaces for parallel processing after wrapping our own Fortran or C/C++ programs into Python. These advantages make it the first choice for parallelizing the program for this thesis.

4.2.2 Software Architecture

The implementation consists of three main parts, ES program, game framework and neural network model. The ES program is written in C while the game framework and neural network model are implemented in Python. We use ctypes to do the communication between the C program and the Python program.

The parameters and sample parameters are saved in shared memory using ctypes. At each iteration, ES program generates new sample parameters in shared memory and network model will be updated with the sample parameters. Then the game framework will evaluate the new model and save the fitness value in shared memory and the fitness will be used to do the update by ES program.

4.2.3 Parrallelization

Before describing our parrallelization method, we first illustrate some important concepts about Parallel Computing:1)A **Cluster** is a group of loosely connected computers that work together, so that they can be regarded as a single computer. Clusters are composed of multiple nodes connected by a network. 2) A **Node** is a standalone "computer in a box" compromised of multiple CPUs/processors/cores, memory, network interfaces, etc. They are networked together to comprise a supercomputer.3) **Shared Memory** describes a computer architecture where all processors have direct access to common physical memory. 4) **Communications** means the exchange of data in Parallel Computing. This can be accomplished through a shared memory bus or over a network. 5) **Embarrassingly Parallel** means solving many similar, but independent tasks simultaneously with little to no need for coordination between the tasks.5) **Parallel Overhead** is the amount of time required to coordinate parallel tasks, as opposed to doing useful work.

We adopted a slightly different communication strategy used in OpenAI ES (Salimans et al., 2017). We use a master-worker architecture: at each iteration, masters broadcasts parameters to the workers, and the workers send returns to the master. The communication is based on using shared random seeds and shared memories, which drastically reduces parallel overhead.

In practice, we have n nodes and p processes per node, so $k = n * p$ processes in total in the cluster. Each node has a master process which stores a sample for all workers processes and creates a shared memory block for each node-local worker's policy network parameters. Each worker in a node has a global_rank (MPI global_comm_rank), global_sample_index, node_param_index (index into shared memory block for each local worker). The pseudo random number generator (prng) has two seeding parameters: seed, and sequence.

At each iteration, each node's master creates a prng for each worker in the cluster using the same initial seed, and use the sequence as the worker's global sample index. After the master generated new parameters for the worker, it sends a message to the worker. And then the worker does a rollout and sends the fitness to the relay. The purpose of the relay is to ensure all masters get the fitnesses in the same order.

By using hundreds to thousands of parallel CPU workers, our implementation can solve Bipedal Walker/Lunar Lander/Half Cheetah in minutes and obtain competitive results on Pong after hours of training time.

Chapter 5

Exerimental Results

To test the effectiveness of different parts of our algorithm, we test four variants of our algorithm:

1. Non-adaptive covariance (Fixed sigma)
2. Adaptive diagonal covariance (Adaptive sigma)
3. Adaptive orthonormal basis and non-adaptive diagonal covariance (Adaptive frames)
4. Adaptive orthonormal basis and adaptive diagonal covariance (Adaptive frames and sigma)

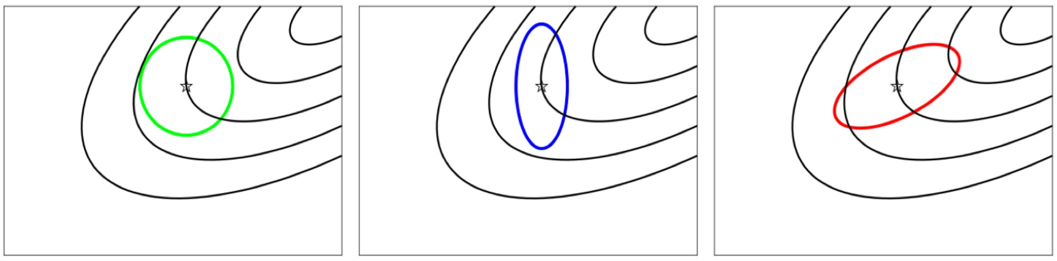
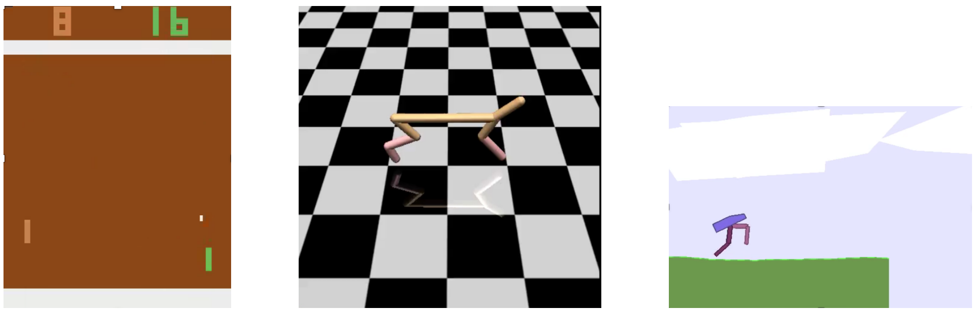


Figure 5.1: Distributions updated with: Algorithm 1 (Left), Alorithm 2 (Middle), Algorithm 3 (Right), Algorithm 4 (Right)



(a) Pong (b) Half-cheetah (c) Bipedal Walker

Figure 5.2: Selected environments from OpenAI Gym

5.1 Experimental Setup and Hyperparameters

5.1.1 Envrionments

Mujoco. We use a benchmark of continuous robotic control problems in the OpenAI Gym [Brockman et al., 2016] for analyzing our algorithms. We have tested on various problems including cart-pole balancing, inverted pendulum, and more difficult ones like half-cheetah. The environments were simulated by MuJoCo (Todorov et al., 2012) For these environments, we use multilayer perceptrons with 2-3 10-unit hidden layers separated by tanh nonlinearities. The architecture is smaller than the one used in OpenAI's work (Salimans et al., 2017), but we found our algorithms can still learn effectively and have faster convergence speed.

Atari. We also ran our algorithms on some Atari games available in OpenAI Gym (Todorov et al., 2012). We used the same preprocessing methods used by (Mnih et al., 2016). The preprocessing processes include downsampling and converting the RGB observations to gray scale.

Others. We use simple envrionments like cart-pole balance for testing and debugging. And we also successfully trained on a variety of envrionments including Lunar-Lander, Bipedal Walker, Acrobot and etc.

5.1.2 Hyperparameters

We use the Bipedal Walker from OpenAI Gym as a test environment for hyperparameter tuning. We employ a multilayer perceptrons with 3 hidden layers and 10 nodes for each layer, separated by tanh nonlinearities. The total number of weights is 320. We terminate each run when the average over a set of 10 consecutive fitness evaluations reaches a threshold of +100. We fix the momentum at 0.9 for μ, σ, ρ and u (where applicable).

Algorithm 1: Fixed Sigma

We find it convenient to express σ_0 as a multiple of $1/\sqrt{n}$ where n is the number of free parameters. In this way, σ_0 becomes a measure of the Euclidean distance between the mean μ and a typical sample point x. We first fix the learning rate α_μ to the generally accepted value of 0.05 and test various values of σ_0 between 1 and 10. The results (shown in Fig. 5.3) indicate that $\sigma_0 \simeq 5$ is the best of these values.

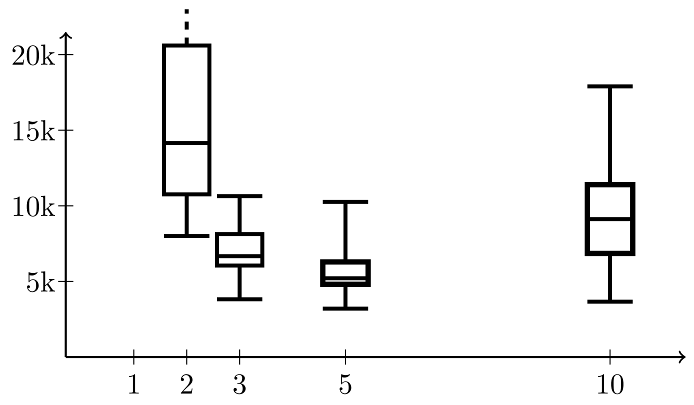


Figure 5.3: Distribution of stopping times for Algorithm 1 with $\alpha_\mu = 0.05$ and varying values of σ_0 (horizontal axis). For each set of 25 runs, the bounding box specifies the middle two quartiles, the horizontal line is the median, and the error bars indicate the lowest and highest stopping time.

An alternative approach is to vary σ_0 while keeping the product $\alpha_\mu\sigma_0$ fixed (since this measures the Euclidean distance of a typical μ-update). Fig. 5.4 shows the results of this approach. We see that $\sigma_0 = 3$, $\alpha_\mu = 0.083$ now emerges as a superior choice.

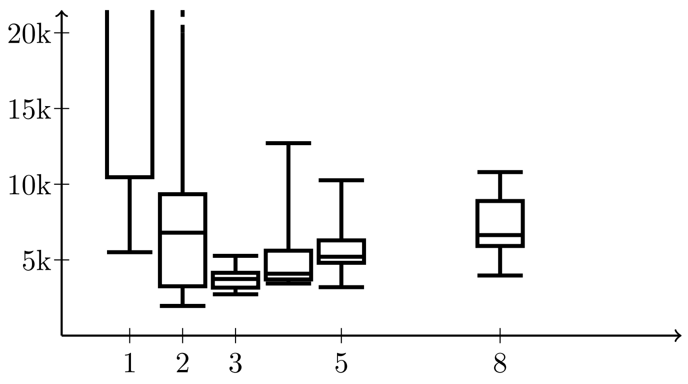


Figure 5.4: Distribution of stopping times for Algorithm 1 with varying values of σ_0 (horizontal axis) setting $\alpha_\mu = 0.25/\sigma_0$ in each case.

When σ_0 is too low, the algorithm may fail due to insufficient variation in behavior between different samples. When σ_0 is too large, the fitness at sample point x may not be representative of values between μ and x. Additionally, with the tanh activation function, small weights in the early stages of neural network evolution may restrict the network to essentially linear behavior, while large weights may saturate the activation function leading to more discretized computation. There should be a "sweet spot" in the middle where the weights are just large enough to produce the right amount of nonlinearity.

Algorithm 2: Adaptive Sigma

Algorithm 2 aims to adjust the values of the individual σ_i's based on the sequence of observed fitness evaluations. Considering the number of parameters compared to the number of observations, there will be a lot of noise in this signal, hence the need for the regularizing parameter λ_σ.

Fig. 5.5 shows the results from testing various values of λ_σ ranging from 0.05 to 1.0 . When λ_σ is large, each σ_i is constrained to remain very close to σ_0, so the distribution of stopping times becomes very similar to that observed in Fig. 5.3 with σ_0 fixed at 5. When λ_σ becomes small, the σ_i's are allowed to vary widely, with much of the variance will surely be attributable to noise in the signal, so the performance of the algorithm degrades. Theoretically, there should be an intermediate region in which

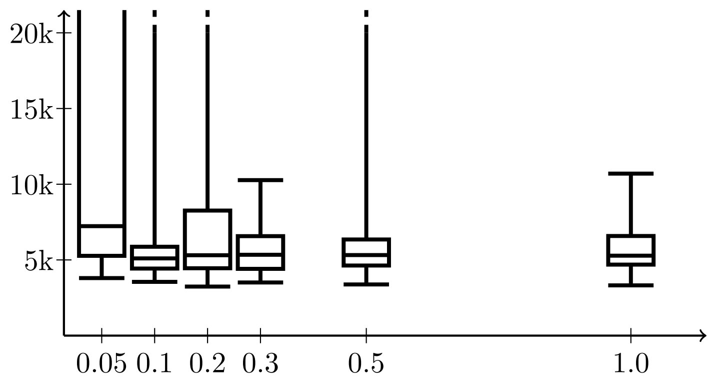


Figure 5.5: Distribution of stopping times for Algorithm 2 with $\alpha_\mu = 0.05$, $\sigma_0 = 5$, $\alpha_\sigma = 0.05$ and varying values of λ_σ (horizontal axis).

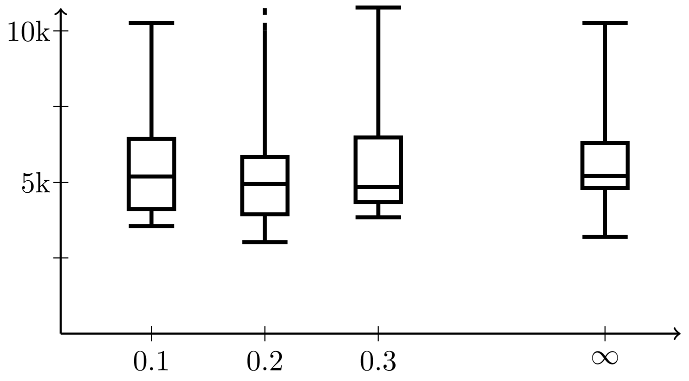


Figure 5.6: Distribution of stopping times for Algorithm 3 with $\alpha_\mu = 0.05$, $\sigma_0 = 5$, $\alpha_\rho = 0.1$ and $\lambda_\rho = 0.1, 0.2, 0.3$ (horizontal axis). For comparison, the result from Algorithm 1 with fixed $\sigma_0 = 5$ is labeled as "∞".

the performance is improved. In the present instance, the training times do appear to be slightly lower for values of λ_σ around 0.3, but the difference is not statistically significant. This may indicate that, for this particular domain, the sensitivity is roughly similar in magnitude for all parameters, in which case Algorithm 2 would not provide a significant improvement over Algorithm 1.

Algorithm 3: Adaptive Frames

Algorithm 3 aims to rotate the frame vectors $\{\hat{u}^{(j)}\}$, $\{\hat{v}^{(k)}\}$ while simultaneously adjusting the corresponding scaling factors $\{\rho_{(j)}\}$. In general, α_ρ should be large enough that adjustments to $\rho_{(j)}$ will keep up with those of $u^{(j)}$ as it rotates, but not so large as to make the learning unstable. We choose $\alpha_{\text{vec}} = 0.01$ which means that, on average, each fitness evaluation will cause the frame vectors to rotate by 0.01 radians (about half a degree). We tested $\alpha_\rho = 0.05$ and 0.1 and found that 0.1 generally gave better results. Figure 5.6 shows the distribution in training times for $\lambda_\rho = 0.1, 0.2$ and 0.3. The distribution from Algorithm 1 with the same (fixed) value for σ_0 is included for comparison. Algorithm 3 with $\lambda_\rho = 0.2$ does seem to provide a slight advantage compared to Algorithm 1, but a two-tailed Mann-Whitney U-test gives a p-value of 0.3 which is not statistically significant.

5.2 Results

We tested our algorithms on three additional domains – the MuJoCo Half Cheetah, Inverted Pendulum and Inverted Double Pundulum. The results are shown in Figs. 5.7, 5.8 and 5.9.

For the Half Cheetah domain (Fig. 5.7), our training times seem quite fast compared to what has previously been reported, but this may be due in part to our smaller network size. The stopping times for Algorithm 2 do appear to be slightly lower than those for Algorithm 1, but the (two-tailed) significance value for this comparison is 0.134 which is not statistically significant.

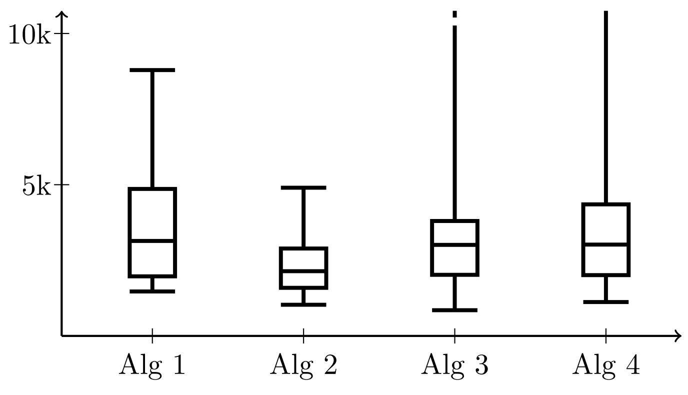


Figure 5.7: Comparison of Algorithms 1, 2, 3 and 4 for the Half Cheetah domain, with 10 hidden nodes in each layer, $\sigma_0 = 2$, $\alpha_\sigma = 0.01$, $\lambda_\sigma = 0.3$ and $\lambda_r ho = 0.1$

For the Inverted Pendulum (Fig. 5.8), Algorithm 2 appears to train faster than Algorithm 1, and a 2-tailed Mann Whitney U-test confirms this observation, with a

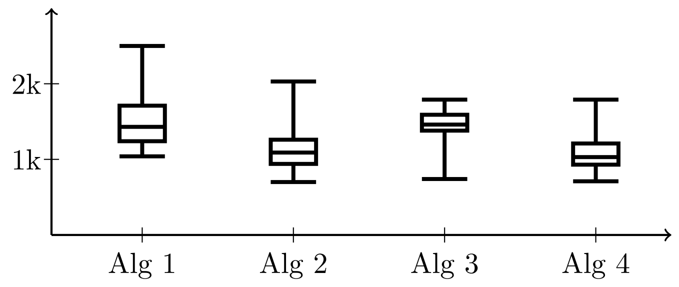


Figure 5.8: Comparison of Algorithms 1, 2, 3 and 4 for the Pendulum domain.

significance score of 0.00065 .

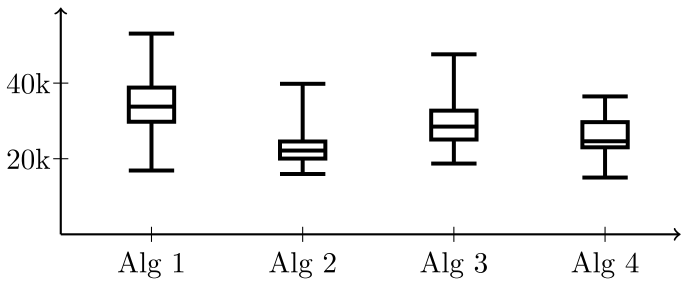


Figure 5.9: Comparison of Algorithms 1, 2, 3 and 4 for the Inverted Double Pendulum.

For the Double Inverted Pendulum (Fig. 5.9), Algorithm 2 clearly trains much faster than Algorithm 1, Algorithm 3 also trains faster than Algorithm 1, with a significance score of 0.025 .

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis focuses on the validation of a newly proposed efficient evolution strategies algorithm to alleviate the internal computational complexity. Our main contributions include:

1. We have analyzed the proposed algorithms and validate the algorithms' performances on a set of test problems. The experimental results show the proposed algorithm with adaptive covariance outperformed non-adaptive covariance which was used in OpenAI's recent work on ES (Salimans et al., 2017).

2. We have conducted several experiments to validate the performance of the scaling orthonormal basis, the proposed algorithm with the scaling orthonormal basis outperforms the algorithm with diagonal covariance on some test problems, but not others.

6.2 Future Work

Except what we introduce In this thesis, we have done and are willing to do many other experiments. We would like to point out some research issues for future works.

1. **Game of Go** We have done some experiments on the Game of Go along with this thesis. However, due to the complexity of the game and limited time we did not achieve a superior result. By using a neural network only to do the move selection, the trained agent is able to beat a random opponent stably while it is still far from beating a human player. We are looking to research on learning the game of go with ES in the future.

2. **Co-evolution**. Current experiments are all done in a fixed fitness environment which means the environment will not change while the agents keep evolving. This works fine in some environments. However, for two-player games like Go, unless we have a pre-tuned expert level opponent, it is difficult for the agents to learn with a fixed weak opponent. Thus, we need a method to let the opponent can evolve with the agent. Co-evolution has previously been applied in a variety of game-playing research and gained competitive result compared with only trained with a fixed opponent. Hence, it would be interesting to combine ES with co-evolution.

3. **Scaling Orthonormal Basis**. As shown in our experimental results, the variant with scaling orthonormal basis did not achieve a statistically significant better performance than simple ES. Explaining this performance involves knowing the loss landscape of the neural network with different environments, this is still an open question. We hope to do more research on that and improve the proposed algorithm.

References

Amari, S.-I. (1998). Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276.

Amari, S.-i. and Nagaoka, H. (2007). *Methods of information geometry*, volume 191. American Mathematical Soc.

Berny, A. (2000). Selection and reinforcement learning for combinatorial optimization. In *International Conference on Parallel Problem Solving from Nature*, pages 601–610. Springer.

Berny, A. (2001). Statistical machine learning and combinatorial optimization. In *Theoretical aspects of evolutionary computing*, pages 287–306. Springer.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym.

Hansen, N., Ostermeier, A., and Gawelczyk, A. (1995). On the adaptation of arbitrary normal mutation distributions in evolution strategies: The generating set adaptation. In *ICGA*, pages 57–64.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937.

Pollack, J. B. and Blair, A. D. (1998). Co-evolution in the successful learning of backgammon strategy. *Machine learning*, 32(3):225–240.

Rechenberg, I. (1973). Evolutionsstrategie: Optimierung technischer systeme nach prinzipien der biologischen evolution. frommann-holzbog, stuttgart, 1973. *Google Scholar*.

Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., and Chen, X. (2016). Improved techniques for training gans. In *Advances in Neural Information Processing Systems*, pages 2234–2242.

Salimans, T., Ho, J., Chen, X., and Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

Schwefel, H.-P. (1993). *Evolution and optimum seeking: the sixth generation*. John Wiley & Sons, Inc.

Todorov, E., Erez, T., and Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE.

Wierstra, D., Schaul, T., Glasmachers, T., Sun, Y., Peters, J., and Schmidhuber, J.

(2014). Natural evolution strategies. *Journal of Machine Learning Research*, 15(1):949–980.