# Comparing Different ES Methods

Ben Glotzer, Peter Leung, Yuxuan Liu, Ayush Madhogaria, and Praveen Sivashangaran

IEOR 4540 - Data Mining

Professor Choromanski

12/17/21

## Abstract

In our project, we sought to better understand the performance of several ES-optimization methods on RL tasks from the OpenAI Gym suite. The methods we used were the ES-style vanilla gradient estimator, gradient estimator with antithetic pairs, and the FD-style gradient estimator. From these different control variate term mechanisms, we used different gradient estimation procedures to compare the ES methods in different scenarios.

## Introduction

### Technical Background

Our project focuses on the optimization of blackbox functions for the purpose of reinforcement learning (RL). By blackbox function, we are referring to a function that we can query (i.e., provide an input and observe an output), but about which we know nothing else. In particular, we make no assumptions on the function's structure, convexity, differentiability or anything else.

A brute-force approach of using a grid search to test out a large set of different inputs and picking the one with the maximum query output would not be suitable for our optimization problem. Many RL tasks involve a large number of parameters, and a grid search would involve testing out several values for each one of these parameters. Since each query of the function may be computationally expensive, a grid search would be far too expensive overall. [1]

There are a number of more sophisticated, well-studied optimization algorithms, but many of these approaches cannot be used in this case, as they rely on information and assumptions that are unknown in the case of blackbox functions. Therefore, we need a method that relies only on observed query input/output pairs and not on any other knowledge about the function or any assumptions about it. We first present some mathematical background relevant to our approach.

**Gradient**

The gradient $\nabla f$ of a differentiable function $f$ is a vector containing the partial derivatives of the function with respect to each of its inputs.

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix}$$

The partial derivative of a function with respect to one of its inputs is the derivative of the function with respect to that input with the rest of the inputs held constant. Formally, the partial derivative has the following definition:

$$\frac{\partial f}{\partial x_i}(\boldsymbol{p}) = \lim_{h \to 0} \frac{f(\boldsymbol{p}+h\mathbf{e_i})-f(\boldsymbol{p})}{h}$$

Here, $x_i$ is one of the inputs of the function $f$, **e**$_i$ is the ith canonical vector (a vector in which the ith element is 1 and the rest of the elements are 0), and the partial derivative is being evaluated at **p**. The partial derivative can be interpreted as the instantaneous rate of change along the direction of the canonical vector. Therefore, the gradient is the direction of greatest ascent, and is fundamental to many optimization methods.

Gradient ascent/descent is one of the most widely used optimization algorithms to train machine learning models. It is extensively employed in machine learning as well as deep learning. It is used to estimate the values of function parameters so that the value of cost function is optimized. The algorithm is an iterative method in which, at each iteration, the input variable is updated by taking a step of a certain size the direction of the gradient of the function at the current point. Of course, it is only possible to take the gradient of a differentiable function whose form is known, so this technique can not be used directly in our application.

**Gradient Estimation**

To overcome this, we might consider adapting the aforementioned gradient ascent method by assuming that the function is differentiable and substituting the gradient with an empirical estimation of the gradient. An approach for estimating the gradient is based on the formal definition of the partial derivative shown above. Instead of evaluating the limit from the definition exactly, we can replace $h$ with a small scalar, $\epsilon > 0$, and use function queries to compute the following approximation of each partial derivative [1]:

$$\frac{\partial f}{\partial x_i}(\boldsymbol{p}) \approx \frac{f(\boldsymbol{p}+\epsilon\mathbf{e_i})-f(\boldsymbol{p})}{\epsilon}$$

This approach has two main issues: (1) we need to query the function twice to compute the estimated partial derivative with respect to each input, and the function may have a very large number of inputs, so this approach may be too computationally expensive, and (2) this approach makes the assumption that the function is differentiable, and in general we do not know if that is the case in blackbox optimization.

**Gaussian Smoothing**

Instead of attempting to estimate the gradient of the original function, we instead start by taking a gaussian smoothing of the blackbox function, which is defined below [2]:

$$F_\sigma(\theta) = \mathbb{E}_{\boldsymbol{g} \in \mathcal{N}(0,\boldsymbol{I})}[F(\theta + \sigma\boldsymbol{g})]$$

This can be computed using only function queries. The gaussian smoothing has the advantage of always being differentiable, even if the original blackbox function is not. Therefore, we can use the gradient of this function:

$$\nabla F_\sigma(\theta) = \frac{1}{\sigma}\mathbb{E}_{\boldsymbol{g}\in\mathcal{N}(0,\boldsymbol{I})}[F(\theta+\sigma\boldsymbol{g})\boldsymbol{g}]$$

Again, we can compute this using only function queries. Unlike the above method of estimating the gradient, which involves estimating the partial derivatives with respect to each input individually, this approach only requires one function query per sample. To show that this is the gradient of the gaussian smoothing of the blackbox function, it suffices to show that [1]:

$$\frac{\partial F_\sigma}{\partial \theta_i}(\theta) = \frac{1}{\sigma}\mathbb{E}_{\boldsymbol{g}\in\mathcal{N}(0,\boldsymbol{I})}[F(\theta+\sigma\boldsymbol{g})g_i]$$

*Proof.* We begin with the definition of the partial derivative. We assume $\theta$ is of length $d$.

$$\begin{aligned}
\frac{\partial F_\sigma}{\partial \theta_i}(\theta) &= \lim_{\epsilon\to 0}\frac{F_\sigma(\theta+\epsilon\, e_i)-F_\sigma(\theta)}{\epsilon}\\
&= \lim_{\epsilon\to 0}\frac{\mathbb{E}_{g\in\mathcal{N}(0,I)}\left[F(\theta+\sigma g+\epsilon\, e_i)\right]-\mathbb{E}_{g\in\mathcal{N}(0,I)}\left[F(\theta+\sigma g)\right]}{\epsilon}\\
&= \mathbb{E}_{g\in\mathcal{N}(0,I)}\left[\lim_{\epsilon\to 0}\frac{F(\theta+\sigma g+\epsilon\, e_i)-F(\theta+\sigma g)}{\epsilon}\right]\\
&= \mathbb{E}_{g\in\mathcal{N}(0,I)}\left[\lim_{\epsilon\to 0}\frac{F(\theta+\sigma(g+\frac{\epsilon}{\sigma}e_i))-F(\theta+\sigma g)}{\epsilon}\right]
\end{aligned}$$

let $g' = g + \frac{\epsilon}{\sigma}e_i$

$$\begin{aligned}
&= \mathbb{E}_{g\in\mathcal{N}(0,I)}\left[\lim_{\epsilon\to 0}\frac{F(\theta+\sigma g')-F(\theta+\sigma g)}{\epsilon}\right]\\
&= \int\limits_{\mathbb{R}^d}\lim_{\epsilon\to 0}\frac{F(\theta+\sigma g')-F(\theta+\sigma g)}{\epsilon}e^{\frac{-\|g\|^2}{2}}\,dg
\end{aligned}$$

since $g' = g + \frac{\epsilon}{\sigma}e_i$, where $e_i$ is a canonical vector, we have that $g'_j = g_j\ \forall j \neq i \Rightarrow dg'_j = dg_j\ \forall j \neq i$

$$= \int\limits_{\mathbb{R}^d}\lim_{\epsilon\to 0}\frac{F(\theta+\sigma g')}{\epsilon}e^{\frac{-\|g\|^2}{2}}\,dg' - \int\limits_{\mathbb{R}^d}\lim_{\epsilon\to 0}\frac{F(\theta+\sigma g)}{\epsilon}e^{\frac{-\|g\|^2}{2}}\,dg$$

notice that:

$$\begin{aligned}
e^{\frac{-\|g\|^2}{2}} = e^{\frac{-\sum g_i^2}{2}} = e^{\frac{-\sum\limits_{j\neq i} g_i'^2 - g_i^2}{2}} &= e^{\frac{-\sum\limits_{j\neq i} g_i'^2-(g_i'-\frac{\epsilon}{\sigma})^2}{2}}\\
= e^{\frac{-\sum g_i'^2+2\frac{\epsilon}{\sigma}g_i'-\frac{\epsilon^2}{\sigma^2}}{2}} &= e^{\frac{-\|g'\|^2}{2}}e^{g_i'\frac{\epsilon}{\sigma}}e^{\frac{-\epsilon^2}{2\sigma^2}}
\end{aligned}$$

applying this to the integral from above:

$$= \int\limits_{\mathbb{R}^d} \lim_{\epsilon \to 0} \frac{F(\theta + \sigma g')}{\epsilon} e^{\frac{-\|g'\|^2}{2}} e^{g'_i \frac{\epsilon}{\sigma}} e^{\frac{-\epsilon^2}{2\sigma^2}} \, dg' - \int\limits_{\mathbb{R}^d} \lim_{\epsilon \to 0} \frac{F(\theta + \sigma g)}{\epsilon} e^{\frac{-\|g\|^2}{2}} \, dg$$

since we have $g'$ within the integrand and we are integrating with respect to $g'$, we can substitute back to $g$

$$= \int\limits_{\mathbb{R}^d} \lim_{\epsilon \to 0} \frac{F(\theta + \sigma g)}{\epsilon} e^{\frac{-\|g\|^2}{2}} (e^{g_i \frac{\epsilon}{\sigma} - \frac{\epsilon^2}{2\sigma^2}} - 1) dg$$

$$= \int\limits_{\mathbb{R}^d} F(\theta + \sigma g) e^{\frac{-\|g\|^2}{2}} \left( \lim_{\epsilon \to 0} \frac{e^{g_i \frac{\epsilon}{\sigma} - \frac{\epsilon^2}{2\sigma^2}} - 1}{\epsilon} \right) dg$$

we can use L'Hôpital's rule to evaluate the limit:

$$\lim_{\epsilon \to 0} \frac{e^{g_i \frac{\epsilon}{\sigma} - \frac{\epsilon^2}{2\sigma^2}} - 1}{\epsilon} = \lim_{\epsilon \to 0} \frac{e^{g_i \frac{\epsilon}{\sigma} - \frac{\epsilon^2}{2\sigma^2}} \left( \frac{g_i}{\sigma} - \frac{\epsilon}{\sigma^2} \right)}{1} = \frac{g_i}{\sigma}$$

plugging back into the integral:

$$= \int\limits_{\mathbb{R}^d} F(\theta + \sigma g) e^{\frac{-\|g\|^2}{2}} \frac{g_i}{\sigma} dg$$

$$= \frac{1}{\sigma} \mathbb{E}_{g \in \mathcal{N}(0,I)} \left[ F(\theta + \sigma g) g_i \right]$$

□

In our project, we will be investigating different approaches of computing this expectation using Monte Carlo methods, in which we evaluate the inside of the expectation using *k* samples of **g** drawn from a normal distribution and take the mean, which yields an unbiased estimate. Furthermore, it can be shown that the gradient can be estimated not only with gaussian smoothing, but also other probability distributions, so we will also explore Monte Carlo methods using sampling from other distributions. Finally, we can use a finite difference style approach for estimating the gradient [2]:

$$\frac{1}{k\sigma} \sum_{j=1}^{k} (F(\theta + \sigma \boldsymbol{g_j}) - F(\theta)) \boldsymbol{g_j}$$

Here, $\boldsymbol{g_j}$ is a vector of length *d* sampled from a normal distribution. We can use a Taylor expansion to show that

$$\frac{F(\theta + \sigma \boldsymbol{g_j}) - F(\theta)}{\sigma}$$

is an estimation of the dot product

$$\nabla F(\theta)^T \boldsymbol{g_j}$$

Therefore, we can also use regression techniques to estimate the gradient:

$$y_j = \frac{F(\theta + \sigma \boldsymbol{g_j}) - F(\theta)}{\sigma}$$

$$G \nabla F(\theta) = \boldsymbol{y}$$

Here, each row of the matrix G is $\boldsymbol{g_j}$, a vector sampled from a probability distribution. With regression, we estimate the gradient by approximately solving this system of equations.

## Our Application

Reinforcement learning is a computational approach to learning how to maximize the total sum of rewards when interacting with an environment. In reinforcement learning, one or more agents interact within an environment which may in a simulation environment. At each step, the agent receives an observation i.e.,the state of the environment, takes an action, and usually receives a reward. Agents learn from repeated trials, and a sequence of those is called an episode. The learning portion of an RL framework trains a policy about which actions cause agents to maximize their long-term, cumulative rewards. In our project we are using Open AI gym environments to simulate tasks like Cartpole, Mountain Car, and Pendulum.

Evolution strategy (ES) has shown great promise in many challenging reinforcement learning (RL) tasks, rivaling other state-of-the-art deep RL methods. Yet, there are two limitations in the current ES practice that may hinder its otherwise further capabilities. First, most current methods rely on Monte Carlo type gradient estimators to suggest search direction, where the policy parameter is, in general, randomly sampled. Due to the low accuracy of such estimators, the RL training may suffer from slow convergence and require more iterations to reach optimal solutions. Secondly, the landscape of reward functions can be deceptive and contains many local maxima, causing ES algorithms to prematurely converge and be unable to explore other parts of the parameter space with potentially greater rewards. Employing a Gaussian Smoothing Evolutionary Strategy can accelerate RL training, which is well-suited to address these two challenges with its ability to i) provide gradient estimates with high accuracy, and ii) find nonlocal search direction which lays stress on large-scale variation of the reward function and disregards local fluctuation. Through several benchmark RL tasks demonstrated herein, we show that Gaussian Smoothing ES is highly scalable, possesses superior wall-clock time, and achieves competitive reward scores to other popular policy gradient and ES approaches.

## Methodologies

### Different Control Variate Term Mechanisms

There are three different unbiased gradient estimators used, namely the vanilla gradient estimator, the antithetic pairs gradient estimator, and the FD-style gradient estimator. Note that the last two estimators are variants of the vanilla gradient estimator with additional control variate terms as a means of achieving variation reduction.

$$\widehat{\nabla}_N J(\theta) = \frac{1}{N\sigma} \sum_{i=1}^{N} F(\theta + \sigma\epsilon_i)\epsilon_i$$

Equation 1 - ES-style Vanilla Gradient Estimator [1]

The Vanilla ES gradient estimator is often characterized by large variance so it is not necessarily the best one out of the three. The control variate terms are added as a means of variance reduction and are expected to outperform the Vanilla ES gradient estimator. The estimator with these extra terms is still unbiased i.e. the expectation of the estimator is the true value of the gradient of the gaussian smoothing(quantity being estimated). Hence, the estimator remains unbiased but there is a noticeable reduction in variance. If N is the number of samples within the neighborhood, the vanilla gradient estimator requires N calls to the blackbox function, the FD-style gradient estimator requires N+1 calls to the blackbox function while the antithetic pairs gradient estimator requires 2N calls to the blackbox function.

$$\widehat{\nabla}_N J(\theta) = \frac{1}{2N\sigma} \sum_{i=1}^{N} (F(\theta + \sigma\epsilon_i)\epsilon_i - F(\theta - \sigma\epsilon_i)\epsilon_i)$$

Equation 2 - ES-style Antithetic Pairs Gradient Estimator [1]

$$\widehat{\nabla}_N J(\theta) = \frac{1}{N\sigma} \sum_{i=1}^{N} (F(\theta + \sigma\epsilon_i)\epsilon_i - F(\theta)\epsilon_i)$$

Equation 3 - ES-style Finite Difference Gradient Estimator [1]

In particular, the antithetic estimator is especially efficient in practice. The drawback is that two times as many queries to the blackbox function are required in order to estimate the gradient as compared to the other two gradient estimators. However, better convergence can be expected due to the improvement in accuracy over the antithetic pairs gradient estimator.

## Gradient estimation procedures

### Monte Carlo Simulation Algorithm

In practice, the multivariate gaussian distribution with mean 0 and some standard deviation is used to sample the points from the neighborhood of the point, and the value of F at these sampled points is computed, and averaged to obtain the smoothened function, J. The variance of the distribution tells how aggressively the space around the point in question is being explored, where a small variance corresponds to averaging over a small neighborhood whereas a large variance corresponds to greater exploration around the point. In addition, the number of samples to take should also be specified. While a large sample size will lead to a better approximation of the gradient of the gaussian smoothed function, J, the additional computational cost is especially significant if the cost of querying the blackbox function is high as each sample requires one blackbox function query.

One important property of gaussian smoothing is that the smoothened function, J, will always be differentiable even if the original function, F is not. However, there is no guarantee that the optimal of the smoothing will correspond to the optimal of the original function, but it is safe to speculate that it will not be far off given that the standard deviation is not sufficiently large. Once the gradient estimation is complete, gradient ascent is used to optimize the reward function.

### Regression-based Algorithms

Another procedure for estimating the gradient of the Gaussian-smoothed function is to use regression-based algorithms. Instead of approximating the gradient by taking the expectation, we applied Ridge and Lasso regression to find the coefficients of the best fitting function in the action-reward neighborhood.

Ridge and Lasso regression are the extensions of basic linear regressions. They create parsimonious models to filter for / put more weights on more relevant variables, and are mostly used to reduce multicollinearity or for feature selections. The key distinction between ridge and lasso regression is that, while they both try to minimize the influence of irrelevant predictors, lasso penalizes the sum of absolute values of the weights (Equation 5) and ridge penalizes the sum of the squared values of the weights (Equation 4). In other words, while lasso can set variable coefficients as zero, ridge only reduces the coefficients of the variables as close to zero as possible and never eliminates a variable.

$$\sum_{i=1}^{M} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{M} \left( y_i - \sum_{j=0}^{p} w_j \times x_{ij} \right)^2 + \lambda \sum_{j=0}^{p} w_j^2$$

Equation 4 - Cost Function of Ridge Regression

$$\sum_{i=1}^{M}(y_i-\hat{y}_i)^2=\sum_{i=1}^{M}\left(y_i-\sum_{j=0}^{p}w_j\times x_{ij}\right)^2+\lambda\sum_{j=0}^{p}|w_j|$$

Equation 5 - Cost Function of Lasso Regression

In each training iteration, we collected the list of our policies in the neighborhood as the independent variables and the finite difference approximation as the dependent variable. The coefficients of the fitted regression function will be our estimated gradient. We then again use the gradient ascent method: walking in the direction of the approximated gradient at a specified step size, and repeat the action until we reach the optimal solution.

**LP Decoding Algorithm**

Finally, we tried fitting this same system of equations by minimizing the sum of the absolute value of the differences rather than the sum of squares. In this special case, we can use LP decoding, in which we solve the following linear program:

$$\begin{array}{ll}\underset{\nabla F(\theta)\in\mathbb{R}^d}{\text{minimize}} & \boldsymbol{y}-G\,\nabla F(\theta)\\ \text{subject to} & \boldsymbol{y}-G\,\nabla F(\theta)\geq\boldsymbol{y}-G\,\nabla F(\theta),\\ & \boldsymbol{y}-G\,\nabla F(\theta)\geq-\boldsymbol{y}+G\,\nabla F(\theta)\end{array}$$

## Methods for sampling

The epsilon terms are sampled independently from a gaussian distribution with mean 0 and the identity matrix for the covariance. From the equations above, it should be noted that $F(\theta)$ is deterministic and the randomization is over the $\varepsilon$ terms. Regardless of the estimator chosen, different methods of sampling these gaussian vector terms can be explored. One way is to independently sample from a gaussian distribution as previously mentioned while another way is to sample them in a structured way so that the directions of the different samples are correlated with each other. If the number of samples you will need is at most the number of parameters that are being optimized, an orthogonal ensemble of gaussian vectors can be created. The margin of distribution should still be gaussian and any single vector should also be gaussian, but as a whole, the vectors should establish an orthogonal ensemble i.e. their angles are exactly $\pi/2$.

Intuitively speaking, in practice, without assuming anything about the blackbox function, the mean squared error of the estimator of the gradient of the smoother is strictly smaller when orthogonal ensembles are used. Gram-Schmidt Orthogonalization can be used to make the rows of a square matrix orthogonal. Since the time complexity of using Gram-Schmidt orthogonalization is significant, it should be done periodically instead of at every step, especially at the beginning of training.

There are two alternative methods of  producing a gaussian orthogonal ensemble, while also being more computational efficient than Gram-Schmidt Orthogonalization. These two methods produce an approximate gaussian orthogonal ensemble instead of an actual gaussian orthogonal ensemble, but for practical purposes, the approximation is sufficient. The primary idea here is to construct a combinatorial matrix where the entries are not from a continuous distribution but from a discrete distribution such that the orthogonalisation is embedded in the structure of the matrix. One method involves the usage of Hadamard Matrices while the other method involves Random Givens Rotations.

Hadamard Matrices can be visualized as square matrices with black and white pixels, where black corresponds to +1 and white corresponds to -1. The first order Hadamard matrix is simply a black pixel and increasing orders of Hadamard matrices are square matrices with dimensionality $2^{n-1}\, x\, 2^{n-1}$. The nth order Hadamard matrix is constructed by copying the n-1th order Hadamard matrix four times, and placing three copies in the first three quadrants while the 4th quadrant is inverted.

$$H_N = \begin{pmatrix} H_{N-1} & H_{N-1} \\ H_{N-1} & -H_{N-1} \end{pmatrix}$$

The key properties of Hadamard matrices that make them an interesting choice for proxying a gaussian orthogonal matrix is as follows. The rows of a Hadamard matrix are exactly orthogonal to each other, and the Hadamard conjecture proposes that a Hadamard matrix of order 4k exists for every positive integer k. Therefore, the idea is to construct the proxy to the gaussian orthogonal matrix as a product of Hadamard matrices. The independent Hadamard matrices will be orthogonal so the resulting proxy matrix will also be orthogonal. However, since the creation of Hadamard matrices is completely deterministic as illustrated in the figure above, randomness needs to be introduced to mimic gaussian properties. The simplest method to introduce randomness is to right multiply the Hadamard matrices by random diagonal matrices where the diagonal entries are taken randomly from a uniform distribution ranging from -1 to +1. There is no need for any further orthogonalization and any row from the resulting proxy matrix will be roughly uniformly sampled and orthogonal to each other. In practice, only around 3 or so “blocks” are required to create this approximate gaussian orthogonal ensemble.

The other method of producing an approximate gaussian orthogonal ensemble utilizes low dimensional rotations referred to as Random Givens Rotations. This method of construction is even more convenient and simpler to implement than the hadamard construction previously discussed. The idea here is to take two particular indices (i, j) that define where the rotation actually occurs. The subsequent 2-dimensional rotation will occur in the two-dimensional space given by the canonical vectors $e_i \, and \, e_j$. Additionally, the product of the Givens rotations is by definition normalized as it is a rotation matrix meaning that there is no l2-normalization required. The more Givens rotations used, the closer to the gaussian orthogonalization matrix this construction looks like.

Therefore, when dealing with high dimensionality, Hadamard Construction or Random Givens Rotations Construction should be used to produce a proxy of the gaussian orthogonal ensemble so that the perturbations used to approximate the gradient of the gaussian smoothing are orthogonal.

## Parameters Tuning

In terms of our model architecture, besides which methodology to use, we also have to decide the optimal value for the hyperparameters based on research, experience, and trials and errors. For our ES methods, the key hyperparameters to tune include: initial policy parameters θ**,** step size or learning rate**,** noise standard deviation σ**,** number of iterations N, and the number of samples (S) used to create the function value neighborhood.

### Θ, σ, and N

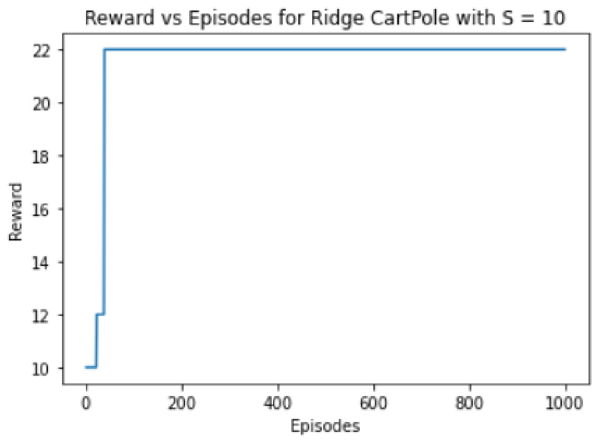
Based on our research on relevant papers and discussion with Professor Choromanski, we decided to initialize the policy parameter by using zero weights W and random normal bias b, and to use 0.1 as the noise standard deviation σ based on empirical results.

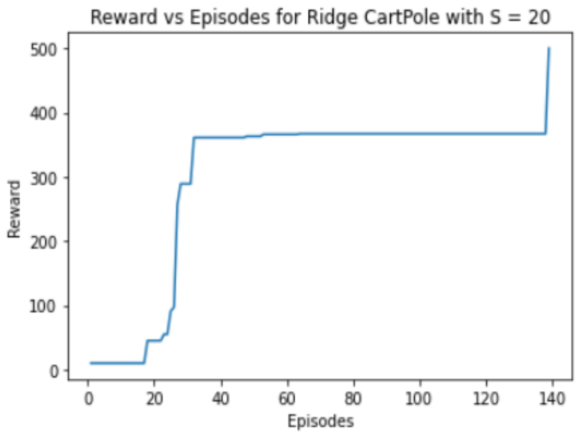
Due to limitations on computing power, we generally perform 1000 iterations to train the algorithm. In certain environments (e.g. Cartpole), by combining the optimal methodology and parameters,  we are able to find an optimized solution within fewer runs and could introduce an early stopping condition to make the algorithm more efficient.

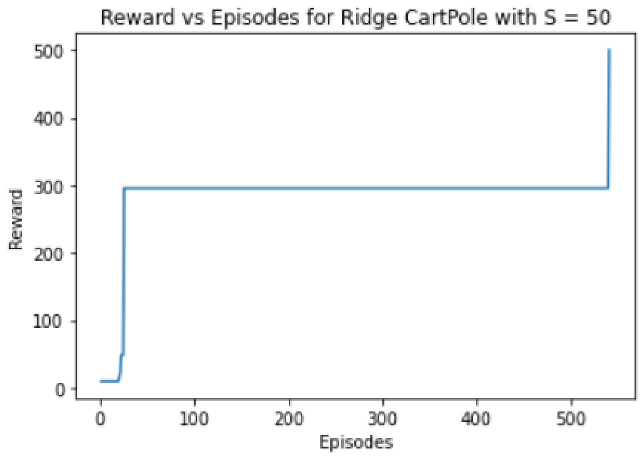
### Number of Samples (S)

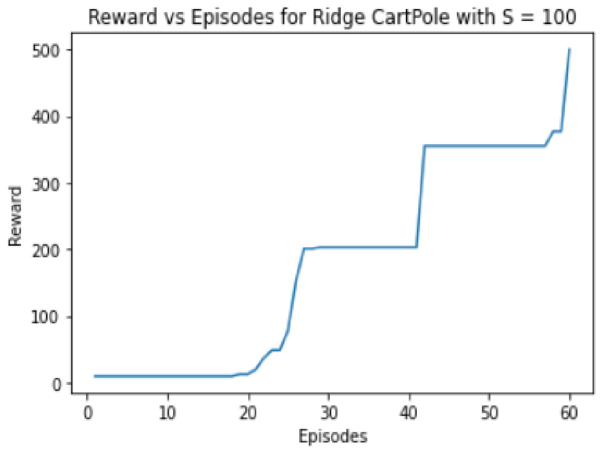
For the numbers of samples S, we tested a range of values from 10 to 100 to observe how the training curve changes with the different values and which value gives the highest rewards within the least number of episodes. Based on our experiments with different environments, smaller values such as 10 failed to find the optimal solution and larger values such as 100 overtrained the algorithm and also did not give the best results. In-between we find the optimal S around 20-50, achieving the optimal solution within a reasonable amount of time.

To break this down into further detail on how we decided on the appropriate value for S, we found the optimal value through trial and error by running the ridge regression algorithm with an S of 10, 20, 50 and 100 for the Vanilla ES method on cartpole.









Looking at our results, we quickly eliminated S = 10 as a viable solution, because there is just not enough data for our model to make an accurate judgment on which direction to turn and it is unable to solve the function within 1000 episodes. This is because noisy data can be easily included in our model, so we decided that we needed more data.

Using S = 100 was also eliminated from consideration, because it was more expensive to run, while lower values of S were able to reach the optimal solution.

Between S = 20 and S = 50, they were both able to reach an optimal solution. However, using S = 20 was more efficient in terms of time to run and was able to solve the problem in fewer episodes, so we decided to use S = 20 for the number of samples for all of our models.

## Step Size

When training the step size, we manually tested values for the step size from 0.01 to 10. However, none of these values led to particularly good results, and in many cases, the method failed to find a policy that “solved” the environment. Instead, we had the idea of implementing a rule by which, in each iteration of the gradient ascent method, the policy and reward only updates if the new policy yields a better reward. The problem with this approach is that it could lead to being trapped in a local maxima. To remedy this, we decided to implement an adaptive step size, which is a common approach in many iterative optimization methods.

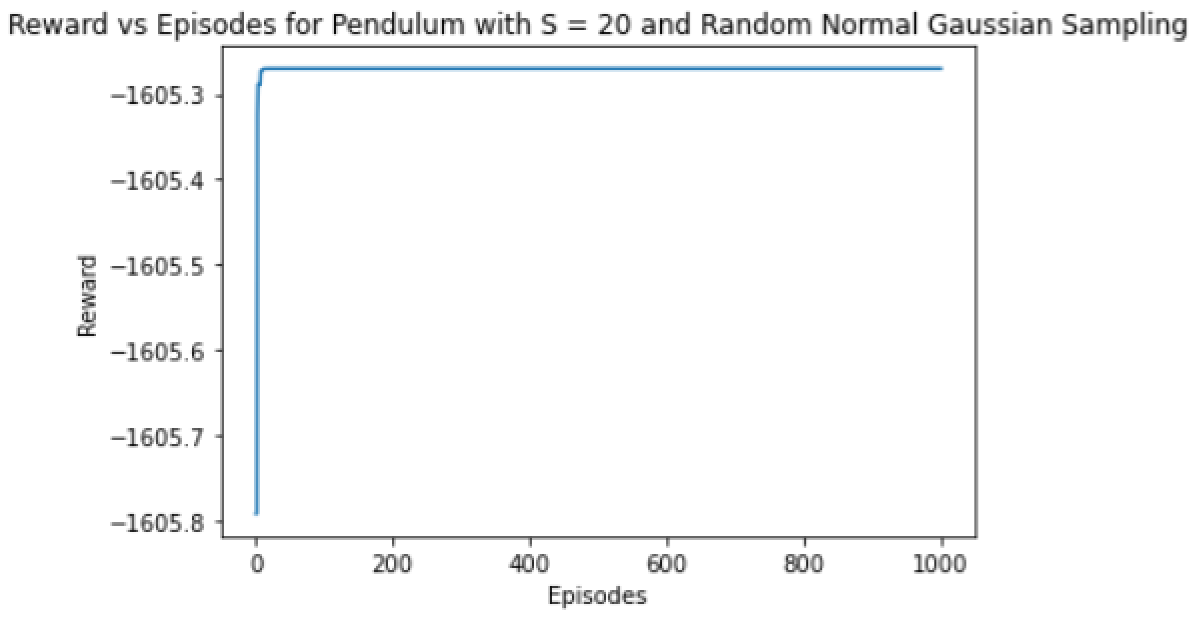
We start with a step size = 0.01. At each iteration, if a step in the direction of the gradient yields an improved reward, we update our policy and current reward and we divide the learning rate by some factor. The motivation for this is that as we approach a maxima, we want to ensure that we do not overstep and miss it. If, in a given iteration, a step in the direction of the gradient does not yield a higher reward, we also divide the learning rate by some factor and initiate a counter, but we do not update the policy or reward. The counter increases by 1 for each consecutive iteration in which the reward does not improve. If the counter is small, we might be approaching an optimal point, so continuing to decrease the learning rate will help to avoid overstepping past the optimal point. However, if we continue to decrease the learning rate without achieving any improvement on the reward, we might be stuck in a local maximum. Therefore, after we pass a certain threshold for the counter, we start increasing the learning rate by some factor. This way, we will eventually arrive at a high enough learning rate such that a step of that size in the direction of the gradient will escape the local maximum and start exploring elsewhere in the domain of the function. With trial and error, we found counter = 5 as a good threshold and a factor = 2 to adjust the learning rate up or down gave the best results.

We find that this heuristic works well for solving the AI gym environments, but it is based on intuition rather than on theory. It is entirely possible that this approach would work poorly in real-world blackbox optimization applications, so future research should be done to explore such a rule outside the context of AI gym environments.

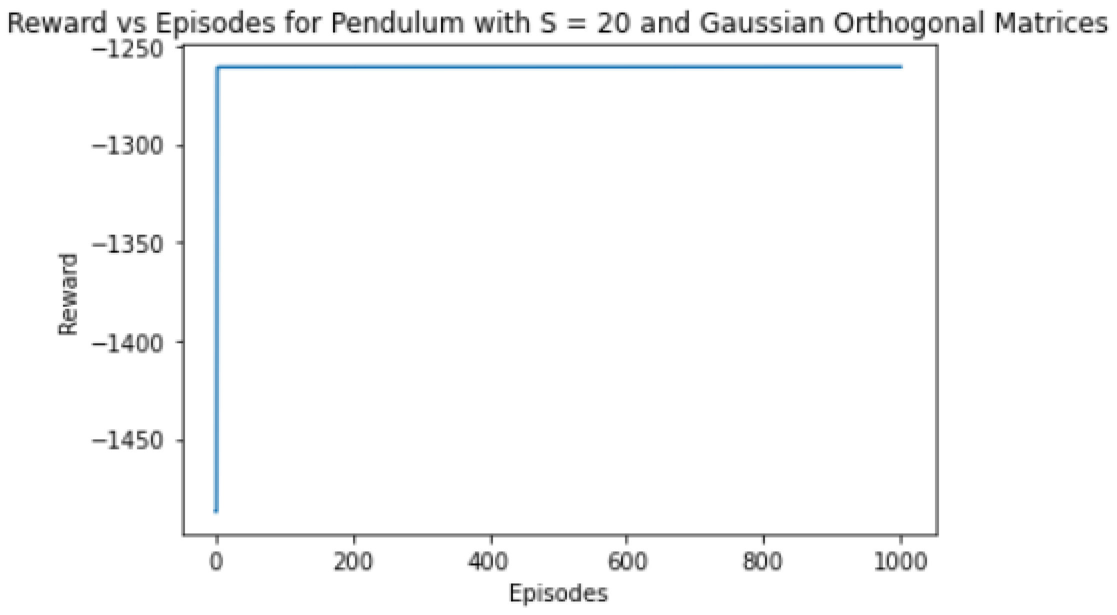
## Results

Running our code in different environments, we were able to get the following results for Pendulum, Cartpole, and Continuous Mountaincar.

For pendulum, we first used Vanilla ES with random normal gaussian sampling and Monte Carlo estimation. We were unable to achieve a particularly high reward with this method, but we were able to initially increase the reward before it stagnated. We did not perform more iterations because it would have been computationally more expensive.



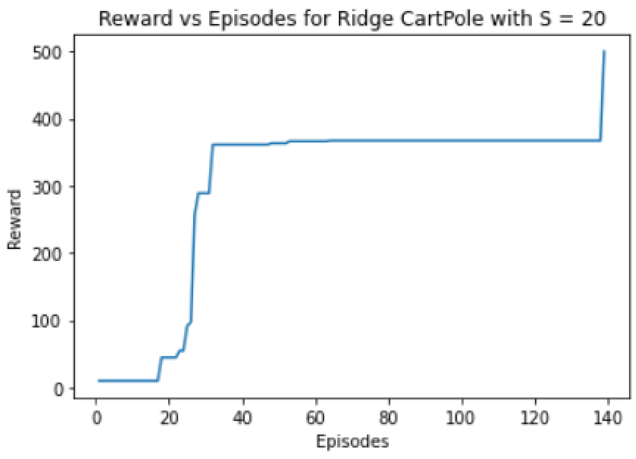
Since we were not able to improve our solution, we sampled our values for epsilon from gaussian orthogonal matrices instead of just random normal values. As discussed in lecture [1], the mean squared error of the estimator of the gradient of the smoothing is strictly smaller when orthogonal ensembles are used. Therefore, we thought we would see an improvement in the rewards. As the graph below illustrates, there is only a slight improvement in the reward for pendulum within the first 1000 episodes.

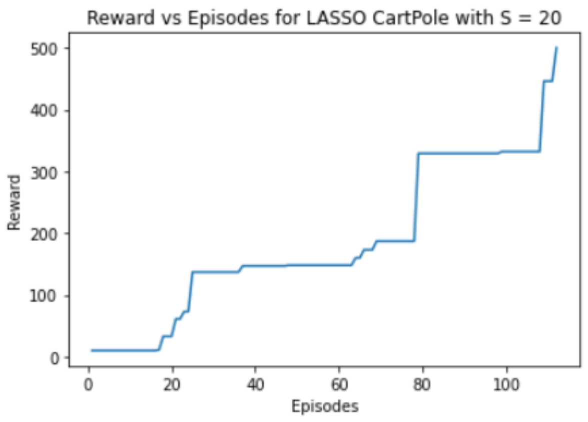


Upon further research, we realized that the pendulum environment cannot be “solved.” Each episode lasts for 200 iterations, and in each iteration, there is a penalty associated with the angle from the upright position, which decreases the reward. Instead, we decided to shift our focus to cartpole, which does have a solution.

For cartpole, we used regression-based algorithms for estimating the gradient of the gaussian smoothing. We looked at ridge and lasso and compared the results. The cartpole problem is considered solved when the reward returned is 500, which is the point where we break the algorithm. If the problem is never solved, the function will iterate for a 1000 iterations (N) and then break after that.

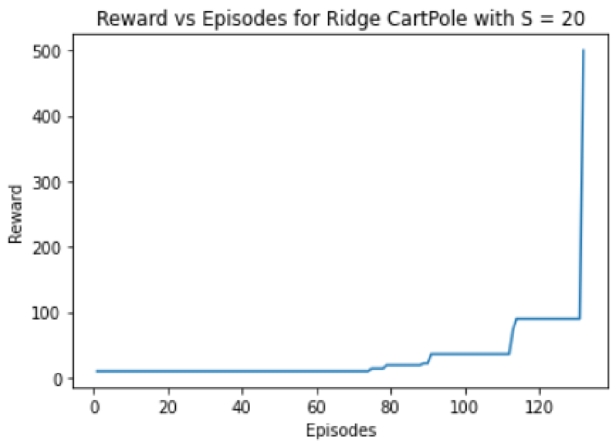
We tuned the alpha parameter for ridge and lasso regression by choosing the value that produced the best mean absolute error (MAE). We set the range between 0.01 and 1 and did a grid search with increments of 0.01. Therefore, for each iteration of the model, there would be different values for alpha between 0.01 and 1. A higher alpha means that there is a greater shrinkage of the coefficients, and the coefficients will be closer to 0, while a smaller alpha means there is less shrinkage on the coefficients. The results below are what we got for cartpole with Vanilla ES.

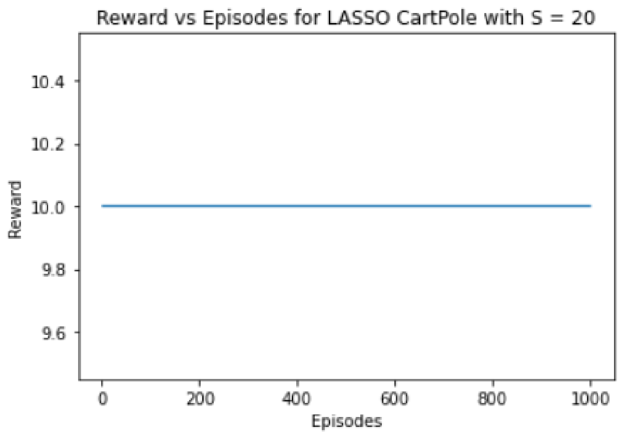




We can see that both regressions were able to reach an optimal solution of 500. The only difference is that LASSO took more incremental steps, which meant that it was able to find a policy with a higher reward more frequently, while ridge took longer to find the policy that produced a higher reward, but the increase in reward was higher for each shift in policy.

For the antithetic method, our rewards for ridge regression consistently improved over each subsequent episode. However, for the case with LASSO, our coefficients were repeatedly 0 despite the parameter tuning for alpha. This could be due to the fact that in LASSO, the coefficients can be set to 0, while in ridge, the coefficients can only get close to 0 but never equal 0. Since we do not have that many features in our model, LASSO may be setting the coefficients to zero, when each variable is crucial towards predicting the results.

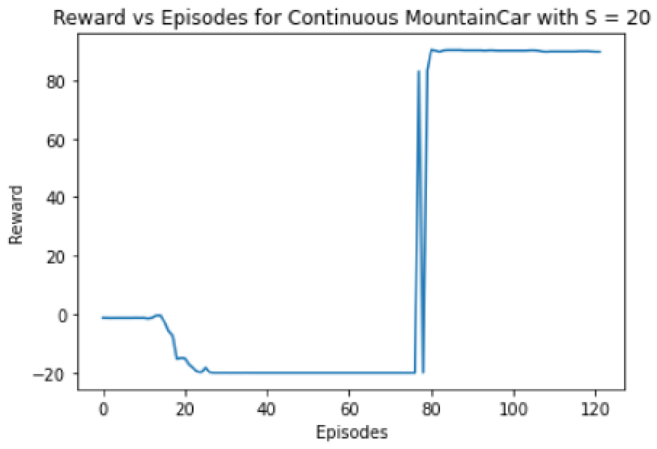




For the forward definite method, our rewards were very similar to the antithetic method results above.

After seeing successful results for cartpole with the regression methods for the vanilla gradient estimator, antithetic pairs gradient estimator, and the FD-style gradient estimator, we wanted to see if we could find similar success in a continuous environment, which was continuous mountaincar.

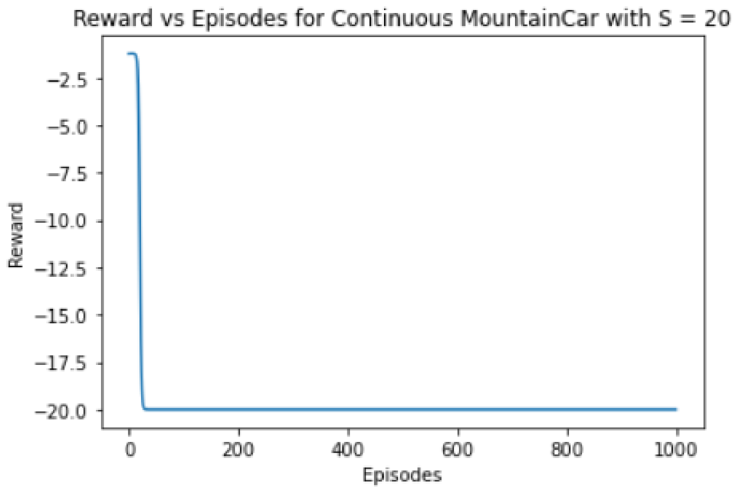
The continuous mountaincar problem is solved when the position is 0.5 [5]. Using this as our criteria on when to break the function, we first tried solving the problem by using Vanilla ES by sampling for epsilon with a random gaussian vector and Monte Carlo. By doing this, we were able to solve the problem in roughly 120 episodes.



We then looked at sampling from a Hadamard matrix over Monte Carlo with a vanilla gradient estimator to see if we can improve on our solution of just sampling from a random gaussian vector. By doing this, we were able to reach the solution on the first step, which is very impressive. The car reaches the top of the mountain and the position of the car is 0.502 and the reward is 89.7.

When sampling for epsilon from a Random Givens Rotation and using Monte Carlo, the algorithm is able to solve for the continuous mountaincar problem within the first step. At the first iteration, the position of the car is 0.502 and the reward is 89.7, identical to the Hadamard matrix case, and tells us that sampling from the Random Givens Rotation and Hadamard matrix both perform very well.

Since continuous mountaincar has only 2 features, we applied LP decoding as a regression method to see if we could improve the results over Monte Carlo with a vanilla gradient estimator. Unfortunately, our algorithm is never able to improve the reward after the first several episodes, as the reward plummets and remains flat at approximately -20 for most of the 1000 iterations.



In general, we realized that after a 1000 iterations, our model would not be able to solve an environment. This is because of our step size. When the rewards reach a local maximum, we increase the step size by a factor of 2 to try to break out of the local maximum. However, this means that if the rewards never improve and flatline, the coefficients of our policy will either be very large or very small, and if our solution does not improve at this point, then there is no reason for us to continue, because the absolute value of our coefficients will only continue to increase. If this does not help us break out of a local maximum, we would not be able to solve the solution. We found that our step size rule worked very effectively in most cases, so we did not alter it, and instead concluded that LP decoding is not the best approach in this case.

## Conclusion

Overall, we found that adjusting the step size was beneficial in helping us to improve our rewards in each environment. Pendulum was difficult for us to improve on our solution, cartpole was easier to solve consistently, and continuous mountaincar was solvable because it only had 2 action states and was easier to control.

For future study, we would be interested in seeing our algorithm implemented in different environments. In our current situation, we were unable to use more complicated environments because we could not get the MuJoCo environments to properly work on our local computers. Although our affine transformation policy is able to solve some of the environments, using more complicated neural networks may be able to produce better results.

## References

[1] Krzysztof Choromanski. Blackbox: ES optimization, 2021. Class lecture.

[2] Krzysztof Choromanski, Aldo Pacchiano, Jack Parker-Holder, Jasmine Hsu, Atil Iscen, Deepali Jain, and Vikas Sindhwani. When random search is not enough: Sample-efficient and noise-robust blackbox optimization of RL policies. *CoRR*, abs/1903.02993, 2019.

[3] Krzysztof Choromanski, Mark Rowland, Vikas Sindhwani, Richard E. Turner, and Adrian Weller. Structured evolution with compact architectures for scalable policy optimization. 2018.

[4] Adrian S. Lewis and Michael L. Overton. Nonsmooth optimization via bfgs. *SIAM J. Optimization*, 2009.

[5] Shiva Verma. Solving reinforcement learning classic control problems — OpenAIGym, Mar 2019.