COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

IEOR E4540 Data Mining

Final Project

Comparing Different Evolution Strategy Methods

Zhaoyang Liu *Ding Yuan*
Kaixuan Ding *Tianqi Li*
Yecheng Ma *Chunxue Chen*

Supervised by
Professor Krzysztof Choromanski

Fall 2021

Abstract

This paper performed Evolution Strategies (ES) methods to solve BlackBox optimization problems. We first experimented with a generic ES method on a preset function, including control variate terms to reduce sampling variance, and regression-based methods with regularization to estimate the gradient of our objective function. Once we are confident with our framework, we tested it using several reinforcement learning tasks from the OpenAI Gym Suite. After careful implementation, we proved that ES methods worked well to solve a BlackBox optimization problem, without explicitly computing gradients of the objective function. Furthermore, adding control variate techniques and regressions-based methods indeed improved our gradient estimation results and efficiency.

I. OpenAI Gym Environment

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. It allows us to simulate an environment to teach agents actions and receive rewards. Initializing an environment with a chosen game is our first step. We will be able to obtain a random initial state and an action space, to begin with. Our goal is to predict actions based on initial states to generate high rewards. As Objective F is unknowable, we need to perturb the actions and estimate the gradient based on the generated data of the perturbed actions and results. The estimated gradient would allow us to improve the parameters with general gradient descent methods. The following games are the ones that choose to perform our ES optimization, we will introduce the settings, states, and actions in each of the following scenarios.

CartPole

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum starts upright, and the goal is to prevent it from falling over by moving the cart left and right. (Please refer to the video in the drive.)

States: Cart position: the horizontal position of the cart (positive means to the right) Cart velocity: the horizontal velocity of the cart (positive means moving to the right) Pole angle: the angle between the pole and the vertical position (positive means clockwise) Pole angular velocity: angular velocity of the pole (positive means rotating clockwise)	
Reward: The reward is 1 for every step taken. Termination: The Pole angle is more than $\pm$15 degrees. Cart position is more than $\pm$2.4.	Actions: 0: Push the cart to the left 1: Push the cart to the right

Mountain Car

There is a car on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum. (Please refer to the video in the drive regarding ES optimization performance.)

States: Car position Car velocity

Reward: 0: if the car reaches the flag (position = 0.5) on top of the mountain. -1: if the position of the car is less than 0.5. Termination: The car position is more than 0.5. The episode length is greater than 200.	Actions: Accelerate to the left Don't accelerate Accelerate to the left

Pendulum

The inverted pendulum swing-up problem is a classic problem in the control literature. In this problem, the pendulum starts in a random position, and the goal is to swing it up so it stays upright. (Please refer to the video in the drive regarding ES optimization performance.)

States: cos(theta): the angle of the pendulum from -1 to 1 sin(theta): the angle of the pendulum from -1 to 1 Theta dot: the angular velocity from -8 to 8	
Reward: -(theta^2 + 0.1*theta_dt^2 + 0.001*action^2) Termination: There is no specified termination.	Actions: The joint effort is a value between -2.0 and 2.0, representing the amount of left or right force on the pendulum.

II. Mathematical Principles and Algorithm

To show our understanding of the Evolution Strategy topics, we wrote a separate Jupyter Notebook to perform ES methods on a preset function without explicitly computing for its gradient. Please refer to *example_ES.ipynb*[1] for more details.

To solve an optimization problem, most of the time, we will have a differentiable objective function and we can reach an explicit form of its gradient. Once we have its gradient, we can proceed to shift our parameters in the gradient's direction for a maximization problem or the reverse for minimization. However, sometimes our objective function's gradient is too complicated to compute, or we don't even know what the objective function is. For this case, we can apply the ES method with gaussian smoothing.

[1] Please refer to *example_ES.ipynb* in the drive. We also have this in the Appendix section below.

Suppose we have a BlackBox function $F(\theta)$, and we want to find an optimal solution θ^* that maximizes this BlackBox function. Since we don't know the gradient of F, we can first generate N gaussian noises $g_{i\in\{1,2,\dots,N\}}$ and move the initial θ_0 to those noisy directions. Next, we need to plug those updated θ_1's into the BlackBox function F to compare the results, and we will move θ_0 in the direction of g_i that has the highest $F(\theta_1)$. Finally, we can repeat this process until a stopping condition is met or an optimal solution is found.

We can also use a regression-based method to estimate the gradient of our BlackBox function. For instance, after generating a set of gaussian noises $g_{i\in\{1,2,\dots,N\}}$, we estimate the gradient component on the direction of noise g_i as $y_i = [F(\theta + \epsilon g_i) - F(\theta)]/\epsilon$, and we can apply a regression-based model with regularization to estimate the gradient in the direction of those noises. Then we update θ by $\theta + \epsilon\hat{\nabla}F(\theta)$. By doing so iteratively, we can reach the optimal value. Below is the Pseudocode to explain this method

---

Pseudocode:

```
1. Initialize a random $\theta_0$
2. Around $\theta_0$, generate $N$ random gaussian noises $g_{i\in\{1,2,3,\dots,N\}}$
3. Use finite difference method, set $y_i = \frac{F(\theta+\epsilon g_i)-F(\theta)}{\epsilon}$, $y \sim g_i^T \cdot grad$, which is the gradient in the direction of those $g_i$
4. Use regression method, set $\hat{\nabla} = argmin_{v\in R^d} \frac{1}{2N}||y - Zv||_p^p + \alpha||v||_q$, where $p, q$ is the norm input, $\alpha$ is the regularization constant, and $Z = [g_i^T]_{i\in\{1,2,3,\dots,N\}}$ is a matrix with each row to be gaussian noise $g_i^T$. Depending on the regularization model we picked, $q$ can be set correspondingly.
5. Update $\theta_0$ with $\theta_0 + \epsilon\hat{\nabla}F(\theta)$. ("minus" for minimization problem)
6. Repeat steps 2~5 until an optimal solution is found or the stopping condition is met.
```

For our project, we consider this BlackBox function F as a hidden reward function for each game in the Gym environment. We came up with two different ES methods as shown below:

ES Method I

For the first method, we consider our parameter θ as a sequence of action a. We perturb this sequence of action a by adding N different gaussian noises $g_{i\in\{1,2,3,\dots,N\}}$. In order to keep the underlying BlackBox function stable and make sure we can get back to the right starting point after perturbation, we used env.seed() to reset the environment. Below are our function specifications:

initialize()	Given the max step of a game and the name of the game, we will use the OpenAI Gym package to generate a series of initial actions in a 1-D array format.
generate_z()	This function will generate noises using three variance reduction sampling methods – Vanilla, Antithetic, and Forward-FD, where the input is the number of samplings, control variate method, and max step of the game, and the output is a matrix with each row being gaussian noise directions.
convertToAction()	The perturbed action is likely to be not legitimate. For instance, in the Cart-Pole example, actions are either 0 or 1 integer. However, after gaussian perturbation, actions can be floating-point numbers. Therefore, we used convertToAction() function to convert perturbed actions back to legitimate actions.
get_reward()	This function inputs an action and current game in the gym environment, and the result will be the corresponding rewards.
regression()	This function uses a regression-based method to estimate the gradient in the direction of generated noises.
get_new_action()	This function inputs gradient, current action, learning rate, and the game, and it outputs perturbed legitimate action.

ES Method II

For the second method, we reference from MorvanZhou's Evolutionary-Algorithm. The overall method would be similar to the ES Method I. We created a neural network that inputs a current state in the game, and the network output will be the optimal policy or action that maximizes the final reward. Our job for this method is to perturb and add noises on the neural network parameters W and b to find the optimal network parameters.

Our neural network is constructed with 3 hidden layers. To make it easy with adding noises, we convert the 2-D dimensional matrix W and b into the 1-D array. Otherwise, the skeleton is pretty similar to ES Method I as discussed above.

III. Variance Reduction & Sampling Techniques

Vanilla Sampling

Using the Vanilla sampling, we generate N gaussian noises $g_{i\in\{1,2,3,\ldots,N\}}$. The gradient can be estimated using the formula below. Once we have the gradient, we can update our current parameter θ in the way specified above. Hence, the Vanilla gradient estimator has a standard Monte Carlo estimator of the expectation.

$$\hat{\nabla}F(\theta) = \frac{1}{N\sigma}\sum_{i=1}^{N} F(\theta + \sigma g_i)g_i$$

Antithetic Variate

The antithetic sampling, also known as mirrored sampling, is a variance reduction technique used in Monte Carlo methods. In addition to the vanilla sampling, it adds the antithetic variables in opposite directions, so the direction of the second half of the sampling is the opposite of the first half, and as a result, it reduces the variance of simulations/sampling.

$$\hat{\nabla}F(\theta) = \frac{1}{2N\sigma}\sum_{i=1}^{N} (F(\theta + \sigma g_i)g_i - F(\theta - \sigma g_i)g_i)$$

Forward-FD (Forward finite difference)

The Forward FD sampling is a variance reduction technique used in Monte Carlo methods based on the Vanilla sampling. It controls variance through adding generated samples with unperturbed actions, which can be regarded as a shift on F of the Vanilla sampling.

$$\hat{\nabla}F(\theta) = \frac{1}{N\sigma}\sum_{i=1}^{N} (F(\theta + \sigma g_i)g_i - F(\theta)g_i)$$

Gaussian Orthogonal Matrices

One problem with our vanilla ES method is that if we generate N noises completely random, it is possible that noises are biased towards some directions. To make sure that we span the whole space, we need to make our noise matrix Z to have orthogonal rows.

Random Hadamard Matrices

Hadamard Matrix is a special case of Gaussian Orthogonal Matrix. The key is to make sure that every perturbation we applied to the original sequence is orthogonal to each other. Instead of using the Hadamard matrix directly, which is composed of +1 and 0, we multiply each element of each row with a number generated from the standard normal distribution. Therefore, the perturbation remains orthogonal, and the matrix can be directly used as a gaussian noise. Then, we randomly choose different rows from the matrix as our perturbation matrix.

Givens Random Rotations

We tried implementing Givens random rotations to the random Hadamard matrix to further improve its performance. It is a low-dimensional random rotation inside the Hadamard matrix. We have space and time complexity gains from it. The algorithm looks like the following:

$$R[i,j] = \{1,\ if\ i = j\ and\ i \notin \{I,J\};\quad 0,\ if\ i \neq j\ and\ \{i,j\} \neq \{I,J\};\quad cos\theta\ if\ i = j\ and\ i \in \{I,J\};$$
$$sin\theta\ if\ i = J,\ j = I;\quad -\ sin\theta\ if\ i = I,\ j = J\}$$

IV. Regression-based gradient estimator

With generated actions space and the corresponding reward scores, we can set up the following optimization task to solve for the estimated gradient that minimizes the Empirical Risk.

$$\hat{\nabla} = argmin_{v \in R^d} \frac{1}{2N} ||y - Zv||_p^p + \alpha||v||_q$$

LP-decoding

Lp-decoding applies a linear regression model to minimize the residual sum of squares. In this case, the alpha parameter above is zero. With the estimated v, we could predict the reward scores based on its actions. However, this method may have an overfitting issue that weakens our results. We may need to explore alternative methods that apply regularization to reduce the scale and/or numbers of gradients to reduce variance.

Lasso

Lasso method allows us to improve estimated gradients through bias-variance tradeoff. With a chosen alpha (0.05), we add an additional l1-norm to the objective, which will punish large estimators to reduce variance. Lasso tends to shrink estimators to zero, which will reduce the change in actions per iteration.

Ridge

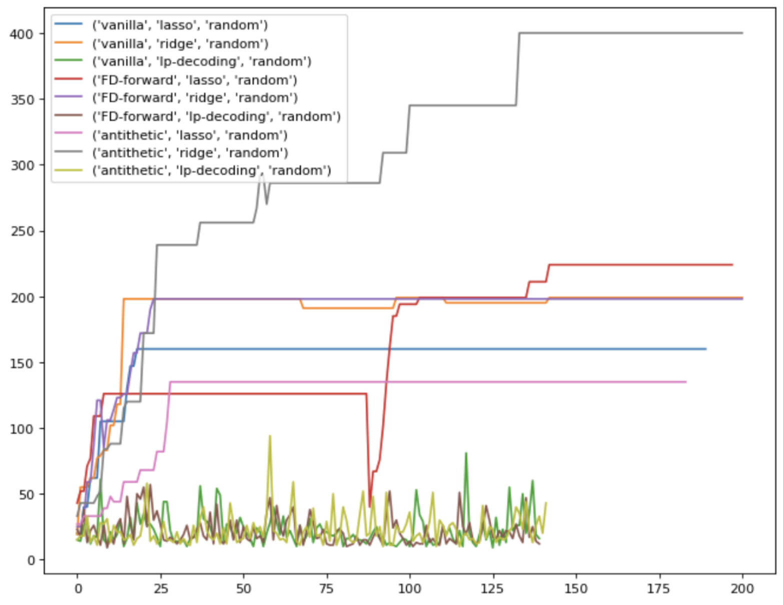
The Ridge method allows us to improve estimated gradients through a bias-variance tradeoff. With a chosen alpha (0.05), we add l2-norm to the objective, which will punish large estimators to reduce variance. Ridge tends to shrink estimators to small values, but not zeros, which will change in actions per iteration.

V. Observation and Conclusion

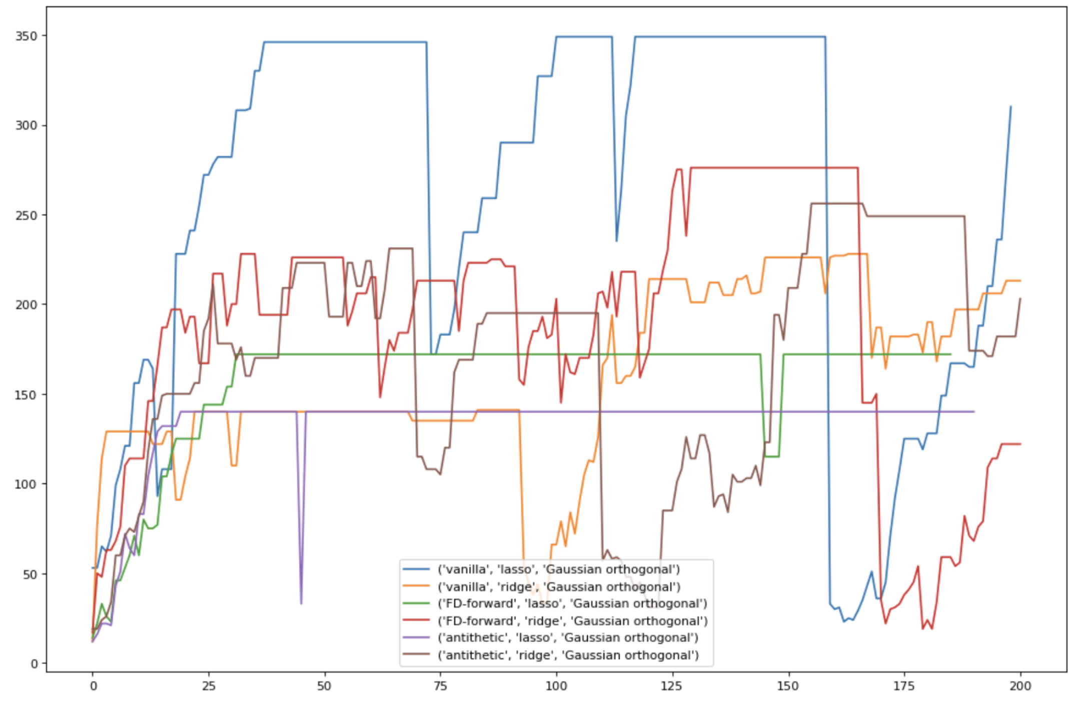
ES Method I

In this section, we will cross-compare three different types of variance reduction sampling methods with three regression-based gradient estimator methods, along with the random sampling matrix, Gaussian orthogonal matrix, rotations, and Random Hadamard matrices. From the graph, the x-axis is the number of iterations, and the y-axis is the reward scores. Due to the limitations in the computing power, we chose not to set the number of iterations too large. We will take both the converging speed, average score levels, and converged score levels into consideration.

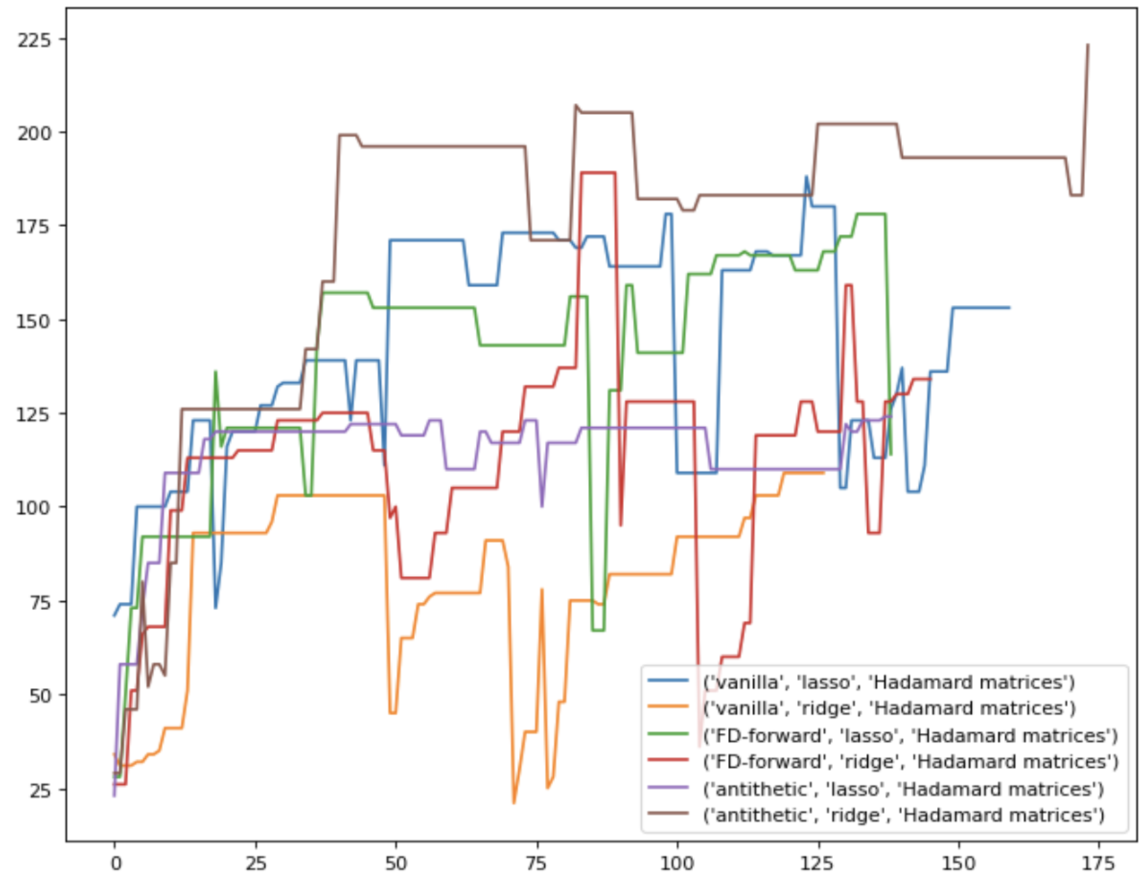
The graph below generates results from the normal sampling matrix. The pair antithetic and ridge performs significantly better than the rest. Also for the following analysis, > means outperform. Given Lasso regression method, Forward-FD > Vanilla > Antithetic; given Ridge regression method, Antithetic > Vanilla > Forward-FD. LP-decoding performs the worst because it has an overfitting issue, which cannot be mitigated by adjusting parameters.



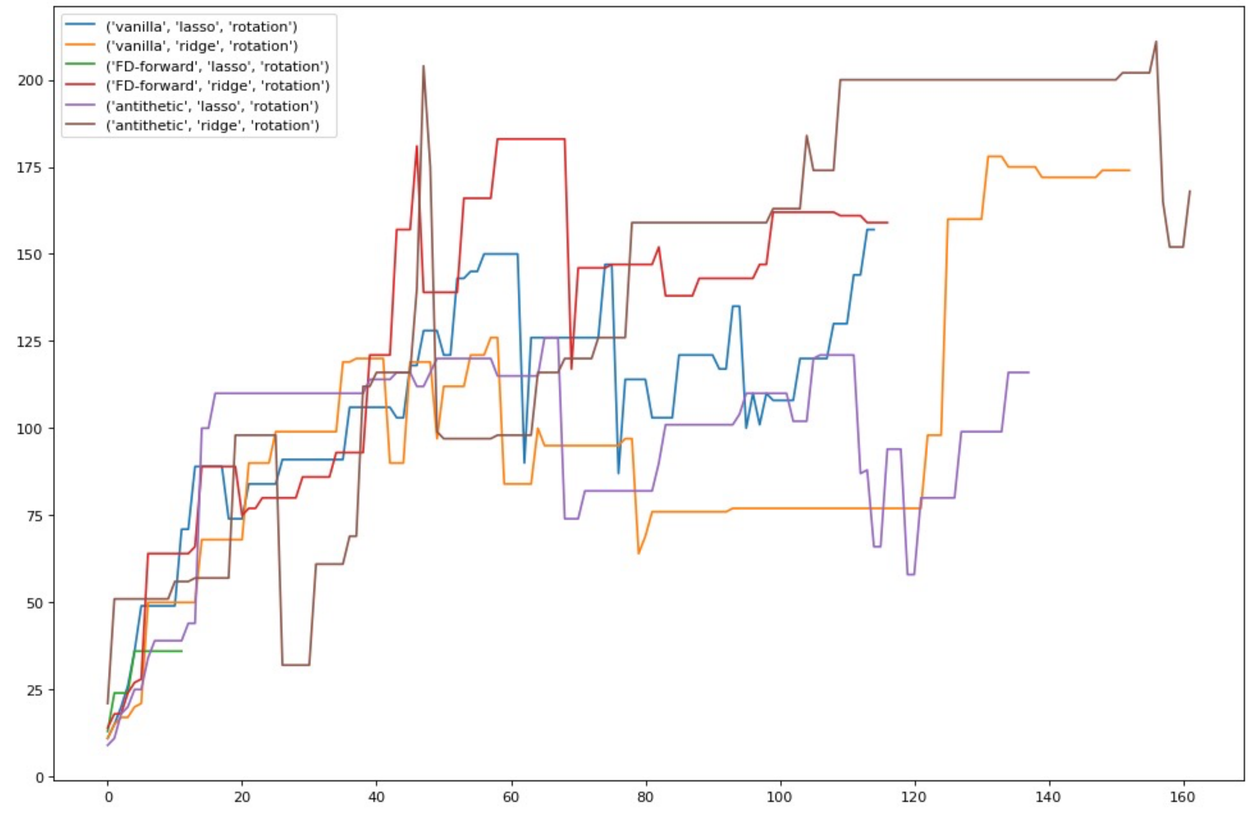
The graph below generates results from the Gaussian orthogonal sampling matrix. The pair of Vanilla and Lasso performs the best compared to others. We didn't use LP-decoding in this graph again because as shown from the graph above, the result is significantly worse than the others. Given Lasso method, Vanilla > Forward-FD > Antithetic; given Ridge method, Forward-FD > Antithetic > Vanilla.



The graph below generates results from the random Hadamard matrix. The pair of Antithetic and Ridge performs the best compared to others. We didn't use LP-decoding in this graph again because from the graph above, the result is significantly worse than the others. Compared to other Monte Carlo simulation methods, the best result of Hadamard is significantly lower. The reason is that Hadamard is composed of a large number of 0 in it, so the regression based on that sampling will take in fewer variables. Therefore, it may need more iterations to achieve better results. Given Lasso method, Vanilla > Forward-FD > Antithetic; given Ridge method, Antithetic > Forward-FD> Vanilla.



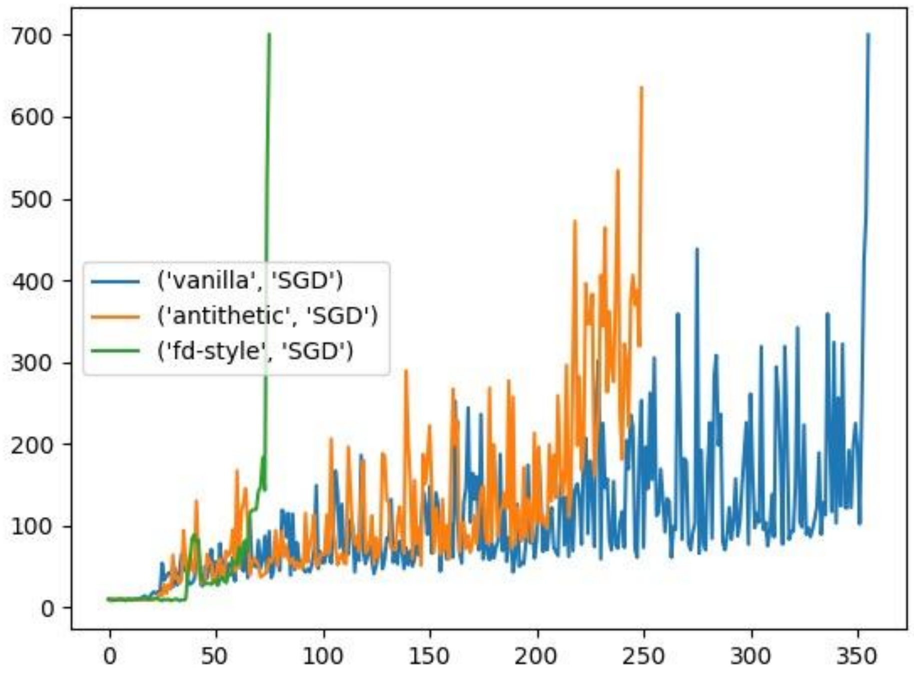
The graph below generates results from a Givens rotation matrix. The pair of Antithetic and Ridge performs the best compared to others. We didn't use LP-decoding in this graph again because from the graph above, the result is significantly worse than the others. During implementation, we noticed that a significant number of iterations result in no change in the score for each perturbation. That led to some lines being shorter than others. Given Lasso method, Vanilla > Antithetic > Forward-FD; given Ridge method, Antithetic > Vanilla > Forward-FD.



There remain some considerations from our graphic results. Ideally, there shouldn't be a significant drop after hitting a relatively great score, but this situation still reflects from our graphs. This is because 1) step size issue and 2) there are random factors when choosing samplings. An overestimated step size will make the new sequence of action away from the optimal even though it's moving in the direction with a better gradient. While sampling, if the samples are chosen from someplace that deviates from the original position too much, the estimated gradient will be inaccurate and lead to a sudden drop in the plot.

ES Method II

In ES Method II, because we get a new action from a neural network, the variance is significantly higher than ES Method I. Hence, we see a more fluctuating graph below. Forward-FD hits the score limits the fastest and then is the Antithetic method. Vanilla performs the worst because the variant control is the most limited.



Conclusion and Future Guidance

Based on the observations above, we can conclude that the ES optimization algorithm can help us estimate the gradient of an unknown function. Reward scores consistently improve across multiple scenarios. With variance reduction techniques, the overall performance on regularized regression methods efficiently improves, especially for antithetic variates. According to our trials, LP-Decoding is comparably weak compared with other methods due to its over-fitting issue. Regularization methods are necessary to ensure good performance. Also, we need to make sure to reset the environment to a fixed state during perturbation. This is important because we once encountered an issue that final rewards were not going up after iterations. If we forgot to reset the environment, we are essentially training on the different BlackBox functions.

Admittedly, several improvements can be made to our algorithm. First of all, the size of each perturbation, epi in the code, can be more accurately tuned. A suitable perturbation will sample near the original position so that the gradient estimate can be accurate. Secondly, the step size needs more review. Good step size is expected to move toward the gradient without surpassing the optimal point.

Works Cited

Fan Mo. “Evolution Strategy 强化学习” *mofanpython*, https://mofanpy.com/tutorials/machine-learning/evolutionary-algorithm/evolution-strategy-reinforcement-learning/. Accessed 2 Dec. 2021.

Appendix

Evolutional Strategies (ES)

This notebook is to show our understanding of Evolutional Strategies (ES). ES method is a blackbox optimization method. Suppose we don't know the blackbox function F, and we want to find an optimal parameter θ that minimizes this F. We are unable to explicitly compute its gradient because it is a blackbox function. However, at current θ, we can generate a set of gaussian noises as the direction/perturbation on our current parameter, and move our current parameter to the noise which minimizes F, or which gives the best proxy for the gradient on the direction of those noises. By doing so iteratively, we can find our optimal parameter θ. This is the general idea of ES method.

For example, define a blackbox function as:

$$F(\theta = \{\theta_1, \theta_2, \theta_3\}) = e^{\theta_1+\theta_2-5\theta_3+0.1} + e^{\theta_1-\theta_2+\theta_3-0.1} + e^{-0.1\theta_1-0.1}$$

Let's proceed with an ES method to find an optimal θ^* that minimize this blackbox function F.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model
```

In [2]:
```
def evalfun(x1,x2,x3):
    e1 = np.exp(x1 + x2 - 5*x3 + 0.1)
    e2 = np.exp(x1 - x2 + x3 - 0.1)
    e3 = np.exp(-0.1*x1 - 0.1)
    return (e1 + e2 + e3)
```

First Method

- Initialize a random θ_0;
- Around θ_0, generate N number of gaussian noises $g_{i\in\{1,2,3,\ldots,N\}}$ as the direction we want to move our θ_0;
- Update $\theta' = \theta_0 + \epsilon g_i$;
- Put θ' into our blackbox function F, compare those N results and choose the one with the best value;
- Repeat until an optimal value θ^* is found.

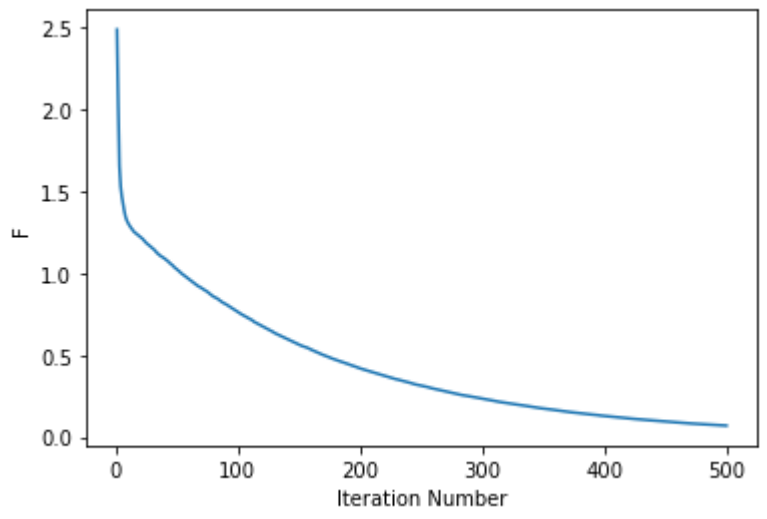
In [3]:
```
### Initialization
numParams = 3
theta_0   = np.random.randn(numParams)         # Initialize a random theta_0
current_F = evalfun(theta_0[0], theta_0[1], theta_0[2])
numIter   = 500
numNoises = 100
mu, sigma = 0, 0.5
epsilon   = 0.1
vectorFun = np.vectorize(evalfun)
threshold = 1e-20
```

In [4]:
```
F_vec = []       # List to store current function value
for step in range(numIter):
    Z = np.random.normal(mu, sigma, (numNoises, numParams))    # Create gaussian noises
    new_theta  = theta_0 + Z*epsilon                           # Update theta
    last_opt_F = current_F
    for noise in range(Z.shape[0]):
        x1, x2, x3 = new_theta[noise][0], new_theta[noise][1], new_theta[noise][2]
        F = evalfun(x1, x2, x3)
        if F < current_F:
            current_F = F
            theta_0 = new_theta[noise]             # Choose the best direction g
    if abs(last_opt_F - current_F) < threshold:    # Stop if we didn't make any significant improvement
        break
    F_vec.append(current_F)

print(f'Optimal F = {current_F:.5f}')
```

```
Optimal F = 0.06941
```

In [5]:
```
plt.plot(range(1,(len(F_vec)+1)), F_vec)
plt.xlabel("Iteration Number")
plt.ylabel("F")
```

Out[5]:
```
Text(0, 0.5, 'F')
```



ES Method with Different Regression Based Gradient Estimating and Gaussian Smoothing

In [6]:
```
### Initialization
numParams  = 3
theta_0    = np.random.randn(numParams)
numIter    = 500
numNoises  = 100
mu, sigma  = 0, 0.001
epsilon    = 0.9
alpha = 0.01
```

Build up on what we discussed above, here we add more color into our vanilla ES method. Use finite difference, we can set the gradient in the direction of each gaussian noise g_i as y_i:

$$y_i = \frac{F(\theta + \epsilon g_i) - F(\theta)}{\epsilon}$$

To find an estimated gradient on the direction of each g_i, we can use regression based method with regularization to solve an objective function as below:

$$\hat{\nabla} = argmin_{v \in R^3} \frac{1}{2N} ||y - Zv||_p^p + \alpha ||v||_q^q$$

- If we use LASSO regularizer, we can set q = 1 to compute l-1 norm. (not always work b/c it forces some elements in v to zero after certain iterations, need to tune learning rate and gaussian parameters)
- If we use Ridge, we can set q = 2 to compute l-2 norm.
- If we use LP-decoding, we can set α = 0

Once we have estimated gradient $\hat{\nabla} F(\theta)$, we can update θ with

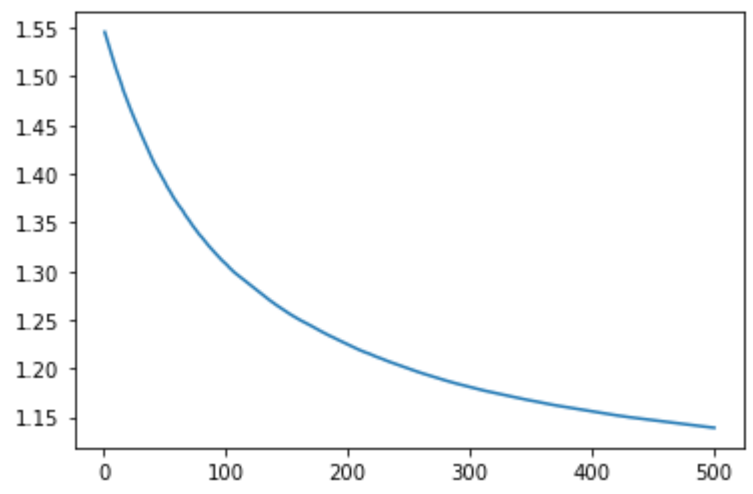
$$\theta' = \theta - \epsilon \hat{\nabla}$$

By doing so iteratively, we can find our optimal θ^*

```
In [7]: def ES_with_regularization(method, numIter, numParams, numNoises, mu, sigma, epsilon, alpha=0.1):
            F_vec   = []
            theta_0 = np.random.randn(numParams)

            if method == 'LASSO':
                model = linear_model.Lasso(alpha)
            elif method == 'Ridge':
                model = linear_model.Ridge(alpha)
            else:
                model = linear_model.LinearRegression()

            for step in range(numIter):
                current_F = evalfun(theta_0[0], theta_0[1], theta_0[2])
                F_vec.append(current_F)
                Z = np.random.normal(mu, sigma, (numNoises, numParams))      # Change Z for Control Variate
                new_theta  = theta_0 + Z*epsilon
                y = np.zeros(numNoises)                                       # Gradient on the direction of noise

                for noise in range(Z.shape[0]):
                    x1, x2, x3 = new_theta[noise][0], new_theta[noise][1], new_theta[noise][2]
                    F = evalfun(x1, x2, x3)
                    y[noise] = (F-current_F)/epsilon

                model.fit(Z, y)            # Solve the objective function above
                v = model.coef_            # Estimated gradient
                theta_0 = theta_0 - epsilon*v

            print(f'Optimal F = {current_F:.5f}')
            plt.plot(range(1,(len(F_vec)+1)), F_vec)
```

```
In [8]: ES_with_regularization("Ridge", numIter, numParams, numNoises, mu, sigma, epsilon, alpha=0.01)
```

```
Optimal F = 1.13900
```



ES Method with Control Variate

- Antithetic Variate

There are several ways for the Antithetic Variate method to reduce the variance here. One way is to create a Z matrix with the second half columns to be the additive inverse to the first half. However, this method requires the number of columns to be an even number, which doesn't work out here as we have 3 parameters. Another way is to create first $N/2$ noises, and the remaining just put -1 in front of these $N/2$ noises. There are also other ways, for instance,

$$\hat{\nabla} = \frac{1}{2N\epsilon}\sum_{i=1}^{N}(F(\theta + \epsilon g_i)g_i - F(\theta - \epsilon g_i)g_i)$$

For an illustration, we used method 2 as below:

```
In [14]: ### Initialization
         numParams = 3
         theta_0   = np.random.randn(numParams)          # Initialize a random theta_0
         current_F = evalfun(theta_0[0], theta_0[1], theta_0[2])
         numIter   = 500
         numNoises = 100
         mu, sigma = 0, 0.5
         epsilon   = 0.1
         vectorFun = np.vectorize(evalfun)
         threshold = 1e-20
```

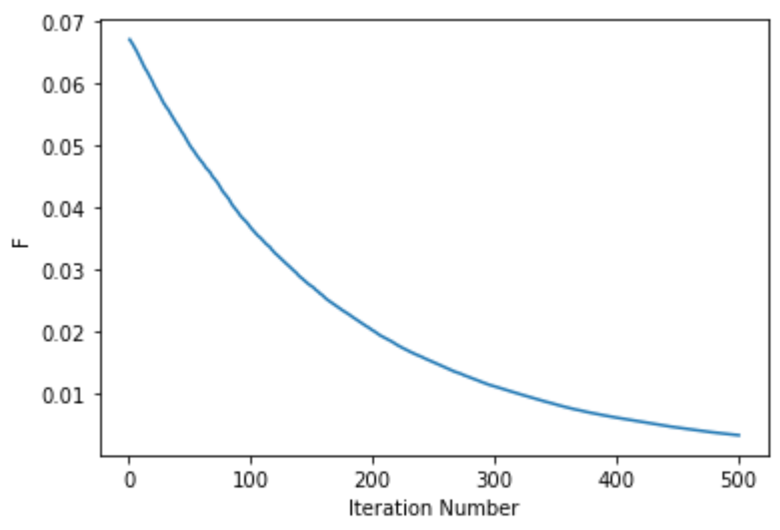
In [16]:
```python
F_vec = []
for step in range(numIter):
    Z_1 = np.random.normal(mu, sigma, (numNoises//2, numParams))
    Z_2 = -Z_1
    Z = np.concatenate((Z_1, Z_2),axis=0)
    new_theta  = theta_0 + Z*epsilon
    last_opt_F = current_F
    for noise in range(Z.shape[0]):
        x1, x2, x3 = new_theta[noise][0], new_theta[noise][1], new_theta[noise][2]
        F = evalfun(x1, x2, x3)
        if F < current_F:
            current_F = F
            theta_0 = new_theta[noise]           # Choose the best direction g
    if abs(last_opt_F - current_F) < threshold:
        break
    F_vec.append(current_F)

print(f'Optimal F = {current_F:.5f}')
```

```
Optimal F = 0.00339
```

Compared with the initial implementation, it is clear that the result after control variate looks better than the vanilla version.

In [17]:
```python
plt.plot(range(1,(len(F_vec)+1)), F_vec)
plt.xlabel("Iteration Number")
plt.ylabel("F")
```

Out[17]:
```
Text(0, 0.5, 'F')
```



- Hadamard Matrix:

One problem with our vanilla ES method is that if we generate N noises completely random, we are having chance to have dependencies among noises in sub-dimensions. To make sure that we span the whole space, we need to make our noise matrix Z to have orthoginal rows. This is where Hadamard Matrix comes into place.

For this function, Hadamard matrix is not a perfect example because we have odd number of parameters to be count for the gradient. However, we need to be aware of that even though the dimension is not a perfect fit, we can still apply Hadamard matrix here as well by adding 0 to the 4th parameter.

In [18]:
```python
def hadamard_mat(size, mu, sigma):
    '''
    Input: size must be an even number
    '''
    result    = np.zeros((size, size))
    half_size = size//2
    mat = np.random.normal(mu, sigma, (half_size, half_size))
    result[:half_size, :half_size] = mat
    result[:half_size, half_size:] = mat
    result[half_size:, :half_size] = mat
    result[half_size:, half_size:] = -mat

    return result
```

Compare the result with gradient descent method

In [19]:
```python
def evalgrad(x1,x2,x3):
    e1 = np.exp(x1 + x2 - 5*x3 + 0.1)
    e2 = np.exp(x1 - x2 + x3 - 0.1)
    e3 = np.exp(-0.1*x1 - 0.1)
    g1 = e1 + e2 - 0.1*e3
    g2 = e1 - e2
    g3 = -5*e1 + e2
    return g1, g2, g3
```

In [20]:
```python
def backtrack(x1,x2,x3, fval, g1,g2,g3, delta1,delta2,delta3, loud):
    alpha = 0.5
    beta = 0.75
    gradtimesdelta = g1*delta1 + g2*delta2 + g3*delta3
    t = 1
    goon = True
    while goon:
        fnew = evalfun(x1 + t*delta1, x2 + t*delta2, x3 + t*delta3)
        target = alpha*t*gradtimesdelta
        if fnew - fval <= target:
            goon = False
            if loud:
                print('done!')
        else:
            t = beta*t
        if t < 0.01:
            goon = False
    return t
```

- Gradient Descent

In [21]:
```python
x1 = .6
x2 = .5
x3 = .4
N = 250
x1sol = np.zeros(N)
x2sol = np.zeros(N)
x3sol = np.zeros(N)
fvalsol = np.zeros(N)
for iteration in range(N):
    x1sol[iteration] = x1
    x2sol[iteration] = x2
    x3sol[iteration] = x3
    fval = evalfun(x1,x2,x3)
    fvalsol[iteration] = fval
    g1, g2, g3 = evalgrad(x1,x2,x3)

    delta1 = -g1
    delta2 = -g2
    delta3 = -g3

    t = backtrack(x1,x2,x3,fval,g1,g2,g3,delta1,delta2,delta3,False)
    newx1 = x1 + t*delta1
    newx2 = x2 + t*delta2
    newx3 = x3 + t*delta3
    newfval = evalfun(newx1,newx2,newx3)
    x1 = newx1
    x2 = newx2
    x3 = newx3

print(f'Function value = {newfval:.4f}')
```

```
Function value = 0.6730
```