Natural Evolution Strategies (NES) toy example that optimizes a quadratic function

nes.py

```python
"""
A bare bones examples of optimizing a black-box function (f) using
Natural Evolution Strategies (NES), where the parameter distribution is a
gaussian of fixed standard deviation.
"""

import numpy as np
np.random.seed(0)

# the function we want to optimize
def f(w):
  # here we would normally:
  # ... 1) create a neural network with weights w
  # ... 2) run the neural network on the environment for some time
  # ... 3) sum up and return the total reward

  # but for the purposes of an example, lets try to minimize
  # the L2 distance to a specific solution vector. So the highest reward
  # we can achieve is 0, when the vector w is exactly equal to solution
  reward = -np.sum(np.square(solution - w))
  return reward

# hyperparameters
npop = 50 # population size
sigma = 0.1 # noise standard deviation
alpha = 0.001 # learning rate

# start the optimization
solution = np.array([0.5, 0.1, -0.3])
w = np.random.randn(3) # our initial guess is random
for i in range(300):

  # print current fitness of the most likely parameter setting
  if i % 20 == 0:
    print('iter %d. w: %s, solution: %s, reward: %f' %
          (i, str(w), str(solution), f(w)))

  # initialize memory for a population of w's, and their rewards
  N = np.random.randn(npop, 3) # samples from a normal distribution N(0,1)
  R = np.zeros(npop)
  for j in range(npop):
    w_try = w + sigma*N[j] # jitter w using gaussian of sigma 0.1
    R[j] = f(w_try) # evaluate the jittered version

  # standardize the rewards to have a gaussian distribution
  A = (R - np.mean(R)) / np.std(R)
  # perform the parameter update. The matrix multiply below
  # is just an efficient way to sum up all the rows of the noise matrix N,
  # where each row N[j] is weighted by A[j]
  w = w + alpha/(npop*sigma) * np.dot(N.T, A)

# when run, prints:
# iter 0. w: [ 1.76405235  0.40015721  0.97873798], solution: [ 0.5  0.1 -0.3], reward: -3.3230
# iter 20. w: [ 1.63796944  0.36987244  0.84497941], solution: [ 0.5  0.1 -0.3], reward: -2.678
# iter 40. w: [ 1.50042904  0.33577052  0.70329169], solution: [ 0.5  0.1 -0.3], reward: -2.063
# iter 60. w: [ 1.36438269  0.29247833  0.56990397], solution: [ 0.5  0.1 -0.3], reward: -1.540
# iter 80. w: [ 1.2257328   0.25622233  0.43607161], solution: [ 0.5  0.1 -0.3], reward: -1.092
# iter 100. w: [ 1.08819889  0.22827364  0.30415088], solution: [ 0.5  0.1 -0.3], reward: -0.72
# iter 120. w: [ 0.95675286  0.19282042  0.16682465], solution: [ 0.5  0.1 -0.3], reward: -0.43
# iter 140. w: [ 0.82214521  0.16161165  0.03600742], solution: [ 0.5  0.1 -0.3], reward: -0.22
# iter 160. w: [ 0.70282088  0.12935569 -0.09779598], solution: [ 0.5  0.1 -0.3], reward: -0.08
# iter 180. w: [ 0.58380424  0.11579811 -0.21083135], solution: [ 0.5  0.1 -0.3], reward: -0.01
# iter 200. w: [ 0.52089064  0.09897718 -0.2761225 ], solution: [ 0.5  0.1 -0.3], reward: -0.00
# iter 220. w: [ 0.50861791  0.10220363 -0.29023563], solution: [ 0.5  0.1 -0.3], reward: -0.00
# iter 240. w: [ 0.50428202  0.10834192 -0.29828744], solution: [ 0.5  0.1 -0.3], reward: -0.00
# iter 260. w: [ 0.50147991  0.1044559  -0.30255291], solution: [ 0.5  0.1 -0.3], reward: -0.00
```

refactor commented on Apr 1, 2017

I just wondering, if I know solution, why I need to calculate w? Just assign like this w = solution, DONE?

Thinginitself commented on Apr 1, 2017

@refactor I think it's just a example to introduce Natural Evolution Strategies. This program uses solution in the f function, but f is always calculated without knowing solution in real problem.

refactor commented on Apr 1, 2017

@Thinginitself Got it. So I guess the 'solution vector' is desired outputs from an ideal neural network, solution is NOT weights. the example f function use it to measure the distance from imperfect neural network, just for demo.

marcsto commented on Apr 2, 2017

Thanks! This is really useful as it's a simple and intuitive way of doing RL. One small issue is that A = (R - np.mean(R)) / np.std(R) will result in a division by 0 when all R's are equal which can be somewhat common for simpler problems.

siddharthanpr commented on Apr 3, 2017 • edited

Why do we standardize the reward? The algorithm in the paper does not standardize the reward and looks right. I understand that by standardizing we move in the opposite direction to those weights that yielded rewards less than the mean reward. But this is not included in the paper where the derivation of the gradient is transparent with rigor and no standardization?

Section 3.2 in the paper shows how not standardizing is actually equivalent to standardizing

DanielTakeshi commented on Apr 7, 2017

@siddpr It might just be a convenience here to make the code work faster. I tried with and without standardizing, and the standardized version runs faster (though both work). Intuitively, this is because we get more controlled rewards about zero which help to quickly drive weights to appropriate directions. If a weight is negative, but has to be positive, and our rewards are all negative, the best we can do is to multiply the change by zero to keep it unchanged. Beyond that, standardizing is just generally useful in many cases.

kokorzyc commented on Apr 22, 2017 • edited

tried to understand NES paper, but not fully got it, as i understand it may be good for stable solution.
just played with little change to update solution each iteration, seems its very sensitive to too big moves of solution (animal is running away too quickly), but with small move is able still to catch solution

with too quickly moving solution, the normalization didnt helped, and the target was quickly lost
with A=R, was still able to catch somehow moving target

--# lets move the solution, as imitation of moving target
moving_target = np.random.randint(0,3); # values from 0 till 2
solution_jitter = 1+moving_target * alpha/2.5
solution = solution*solution_jitter

iter 0. w: [1.76405235 0.40015721 0.97873798], solution: [10 3 -2], reward: -83.462896
iter 260. w: [4.07337088 1.17450879 0.15430971], solution: [11.12672453 3.33801736 -2.22534491], reward: -60.093323
iter 1000. w: [10.78848973 3.22619184 -1.88405165], solution: [15.08420061 4.52526018 -3.01684012], reward: -21.423920
iter 2980. w: [29.06941533 8.73460539 -5.81538955], solution: [32.50510382 9.75153115 -6.50102076], reward: -13.308184

holdenlee commented on Apr 24, 2017

Why is sigma multiplied for the perturbations and divided for the update?

```
w_try = w + sigma*N[j] # jitter w using gaussian of sigma 0.1
w = w + alpha/(npop*sigma) * np.dot(N.T, A)
```

taey16 commented on Jun 8, 2017

I appreciate for your effort. I have a question.
In my thought(as you said), Evolution Strategy (ES) could be useful in case we do not know exact gradient. Therefore, we alternatively compute gradients from randomly re-generated samples. I think such idea is analogous to the random-mutation behavior in genetic algorithm. Then, Is there any way of implementing crossover operation which is a basic behavior in the genetic algorithm?

Thanks.

alirezamika commented on Jun 17, 2017

I was looking for a general module for this and I couldn't find it so I developed one based on the mentioned algorithm. the code is available here if anyone's looking for it too. And I got some pretty interesting results here. I hope this helps.

bercikr commented on Jul 31, 2017

How would you go about persisting and executing a trained model for NES. Would just just save the list of weights? Would it be practical to use a trained NES model rather than using a trained DQN for instance?

mynameisvinn commented on Nov 17, 2017

@bercikr aside from the most recent parameter vector, why would you need previous parameter vectors? as long as youve correctly defined the fitness function `f`, then your NES model will never revisit (ie *regress*) previous states.

GoingMyWay commented on Mar 29, 2018 • edited

@alirezamika, hi, may I ask you a question, in line 50 why this update rule works in NES?

```
w = w + alpha/(npop*sigma) * np.dot(N.T, A)
```

since in every iteration, `N` is randomly sampled from Gaussian distribution.

sandorvasas commented on Jun 27, 2018

After reading the article, I feel each couple of years we can bring back old algorithms from the past, which in their time were too computationally intensive, and discover they work excellent, even [almost] better than current cutting-edge algorithms.

How can this be? Taking it to the extremes: if we have infinite computational capacity, even a model guessing with rand() will converge to a perfect policy.

The point I'm trying to make is that besides all the fancy names and "innovative" algorithms in deep learning, and ES, I'm starting to think that most of the achievements can be credited not to the algorithms themselves, but the constantly improving computational performance.

EliasHasle commented on Jul 31, 2018

@holdenlee I am wondering about that too. I guess if it were a mistake, it could still go unnoticed for this simple example.

heidekrueger commented on Aug 29, 2019

@holdenlee, @EliasHasle, the sigma in the denominator is indeed correct and required to make the ES-gradient the same length as the true gradient in expectation. (To see why, you could replace $F(\theta + \sigma\varepsilon)$ in the ES-definition by its first-order taylor expansion and then solve the expectation.)

onerachel commented on Oct 1, 2022

Wrt this: w = w + alpha/(npop*sigma) * np.dot(N.T, A), my understanding is that the author optimizes over w directly using stochastic gradient ascent with the score function estimator: (N*A)/(npop*sigma). Matrix N is transposed by N.T to multiply with A easier. The return value is the estimate average reward value over npop, then this value is multiplied with the step size alpha, the w is updated.