Deep Reinforcement Learning – Project 2 : Continuous control
Jovan Jovanovic

Goal of the project is to learn agent to control the ball. Without any prior knowledge he is trying to explore and exploit the environment to learn the rules and master the environment. Solving the environment consist of collecting average score above +30. Since the environment is designed to be MDP problem suits for Reinforcement Learning. Now the problem with Reinforcement Learning in this case is that rewards are very sparse. Especially if we want to train quickly and with less steps. For example if we put number of steps to 100, rewards in the beginning are around 0 and that makes things pretty hard for learning.
That is where Evolution strategies shines.

Evolutionary strategies

Most of the theory is taken from here https://openai.com/blog/evolution-strategies/ and original paper posted by OpenAI https://arxiv.org/abs/1703.03864.
I will try to both give my opinion and describe how I fall in love with this approach and also go to some technical details later.

In ES, we forget entirely that there is an agent, an environment, that there are neural networks involved, or that interactions take place over time, etc. The whole setup is that 1,000,000 numbers (which happen to describe the parameters of the policy network) go in, 1 number comes out (the total reward), and we want to find the best setting of the 1,000,000 numbers. Mathematically, we would say that we are optimizing a function `f(w)` with respect to the input vector `w` (the parameters / weights of the network), but we make no assumptions about the structure of `f`, except that we can evaluate it (hence "black box").

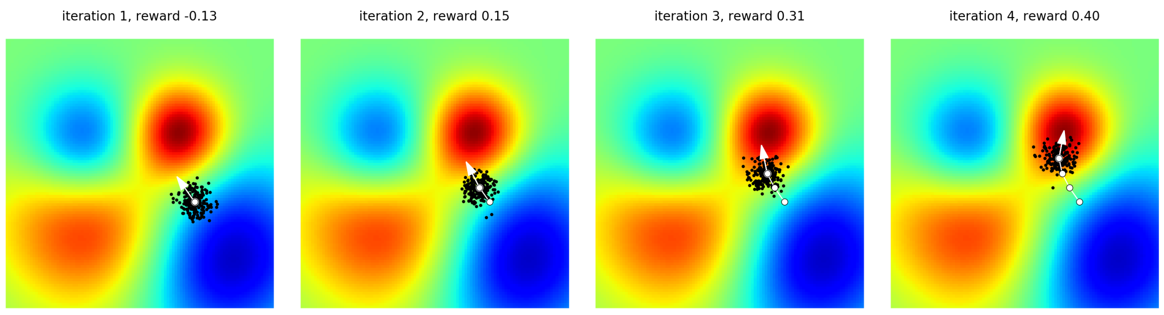
In simple words, evolution strategies are type of black box optimization where we are trying to directly change the parameters based on rewards that we are getting back from the environment. So comparing to Reinforcement learning we still use reward as a training signal but instead of doing 'guess and check' on actions we do 'guess and check' on parameters.

Whole algorithm looks like this:
1 – Create a population of N neural networks (with same architecture but different weights)
2 - Initialize weights randomly (using some normal distribution)
3 – Act on environment to collect the rewards
4 – Create new generation of N neural networks with formula

$$W = W + R*Noise$$

Where W represents weights of neural network, while R is vector of rewards from particular generation and Noise is random noise that we got from normal distribution.

The intuitive idea is to move our neural net parameters (W) in direction of those who earned biggest rewards.



On this picture you can see whole generation (in black) moving towards global minimum (read area).
One of main strentghts is that it is very hard to get stuck in local minimum because agent is trying to do constant exploration.
There are few hyper params that need to be tweaked. Sigma represents how wide you want to go when generating new weights. So wider you go, more variance is introduced in system. Therefore it is good practice to start with sigma big enough and then decay it once your training is close to finish. Also there is learning rate that represents the same thing like gradient descent training.

Now just imagine that you can run the training with the same speed as you would do on powerfull GPU. That is something unique to this approach because it does only forward pass and all passes inside generation are easily parallelizable because they just need to communicate final summed reward at the end. In my case I was not able to reproduce this because version of Unity environment cannot run in parallel. Therefore I simulated same thing with 20 agents environment that can be used later to solve 1 agent environment. OpenAI claims that they manage to train Mujoco on AWS with more than 100 CPUs in 10 minutes. That is really promising for the future of Evolution Strategies.

At the end one of the strentgths over Reinforcement learning is that exploration is done out of the box. I would say that type of learning is much more creative because you cover much broader set of solutions comparing to gradient descent which is always aiming towards one specific solution. Also we don’t need to tune ‘epsilon’ parameter or OUNoise in case of continuous spaces.

Solution

As said previously we have chance to train our agent for lot less steps and get some good results. In my case I decided to go with only 50 steps and population of 200 different neural nets. Sigma is 0.3 at start and it decays with coefficient 0.999, while learning rate is 0.03. With this set of parameters environment is solved after only 30 minutes:

```
313 Best candidate 1.5400999655783176 Max in generation
1.0900999756366014 Total Average 0.7637999829299749 Best average
1.4868999667674305
314 Best candidate 1.5400999655783176 Max in generation
0.6200999861419201 Total Average 0.7599999830149115 Best average
1.488299966736138
315 Best candidate 1.5400999655783176 Max in generation
1.1200999749660492 Total Average 0.7660999828785658 Best average
1.4896999667048456
316 Best candidate 1.5400999655783176 Max in generation
1.030099976977706 Total Average 0.7689999828137458 Best average
1.4910999666735532
317 Best candidate 1.5400999655783176 Max in generation
1.2600999718368053 Total Average 0.7726999827310443 Best average
1.4924999666422607
318 Best candidate 1.5400999655783176 Max in generation
0.7100999841302633 Total Average 0.7720999827444553 Best average
1.4938999666109682
319 Best candidate 1.5400999655783176 Max in generation
0.8000999821186066 Total Average 0.7737999827064574 Best average
1.4952999665796758
320 Best candidate 1.5400999655783176 Max in generation
0.6000999865889549 Total Average 0.7707999827735127 Best average
1.4966999665483833
321 Best candidate 1.5400999655783176 Max in generation
0.6600999852478504 Total Average 0.7697999827958644 Best average
1.498099966517091
322 Best candidate 1.5400999655783176 Max in generation
0.6500999854713678 Total Average 0.767199982853979 Best average
1.4994999664857986
323 Best candidate 1.5400999655783176 Max in generation
0.6400999856948852 Total Average 0.7629999829478562 Best average
1.5008999664545062

Best average 1.500899966454506 represents the average amongst best
representatives.
```

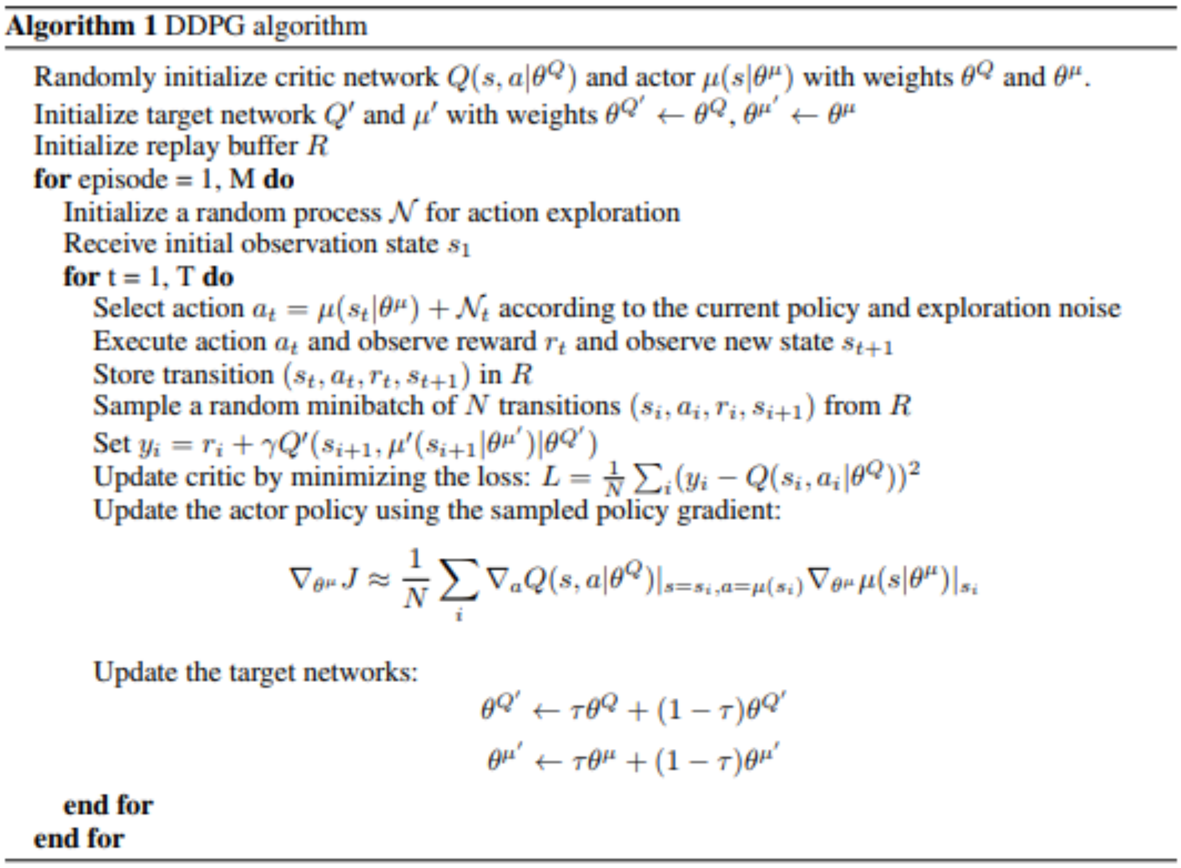
This knowledge can be transferred to full episode training (with 1000 steps).
Path to right hyperparameters were not that easy and therefore I needed to implement realtime param changing using 'keyboard events' in python. So basically after episode by pressing corresponding key you are able to set new value for hyperparameter (sigma, population size or learning rate)

Problem

Now it comes bad part of the story. The training is very sensitive to initial state. Since Reacher environment is not starting from the same state agent would need much much more training to start generalizing. Currently you would need to run whole generation (200 times) to find one which is solving the environment. Possible solution for this problem are additional exploration through adding some noise or running one net multiple times and then averaging the results.

I was not able to achieve this without parallelization of Unity env and using laptop CPU . Therefore I am posting additional solution using DDPG and priority replay.

DDPG



Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  for t = 1, T do
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

      $$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:
      $$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
      $$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$
  end for
end for
```

Prioritized Replay Buffer

Problem with using standard Replay Buffer is that we sample data from there absolutely randomly. When using Prioritized Replay Buffer we can overcome this problem and give priority to the data that is really useful for learning. In other words we are trying to measure the error of particular experience (state, action, reward, next_state) and based on that error give priority to that experience to be selected in learning process.

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

Here we control prioritization with parameter alpha where in alpha = 0 corresponds to the uniform case. With this way of sampling we introduce bias cause we now have correlation between observations. Therefore we need to correct this by using importance-sampling (IS) weights

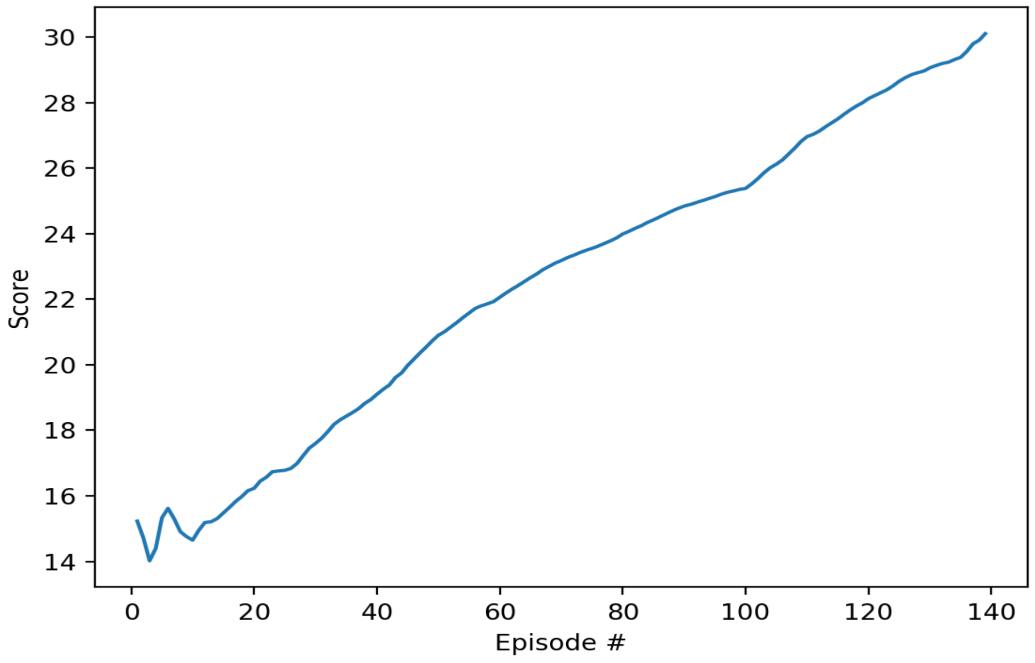
$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta$$

Introducing another hyperparameter beta which controls level of uniformly random sampling.

Final algorithm

Final solution combines DDPG with priority replay. This training uses pretrained weights (trained from two tries) and therefore it starts from reward = 14.
Here is provided list of hyperparameters:

```
BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 1024       # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR_ACTOR = 5e-4         # learning rate of the actor
LR_CRITIC = 1e-3        # learning rate of the critic
WEIGHT_DECAY = 0        # L2 weight decay
ALPHA = 0.6
PRIOR_EPS = 1e-3
BETA = 0.4
```



Ideas for improvement

Genetic algorithm – Already submitted cross_entropy method that was mentioned in lessons, but something more sophisticated is proven to work in complex environment. Therefore with some tweak simple GA can work as well

Additional exploration – currently OpenAI implementation of ES is very sensitive to initial state, therefore it needs some robustness

Hybrid approach – it would be interesting to combine ES with some gradient descent approach. One of the ideas would be to set the goal (in this case 30 points reward) and then instead of random noise use reward signal to follow the gradient of W to reach goal reward.