Deep Reinforcement Learning – Project 3 : Collaboration and Competition
Jovan Jovanovic

Goal of the project is to learn two agents to collaborate in Tennis game. Without any prior knowledge both agents are trying to explore and exploit the environment to learn the rules and master the environment. Solving the environment consist of collecting average score above +0.5. 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.
This is very good chance for Evolution strategies to be used instead of Reinforcement learning.

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").

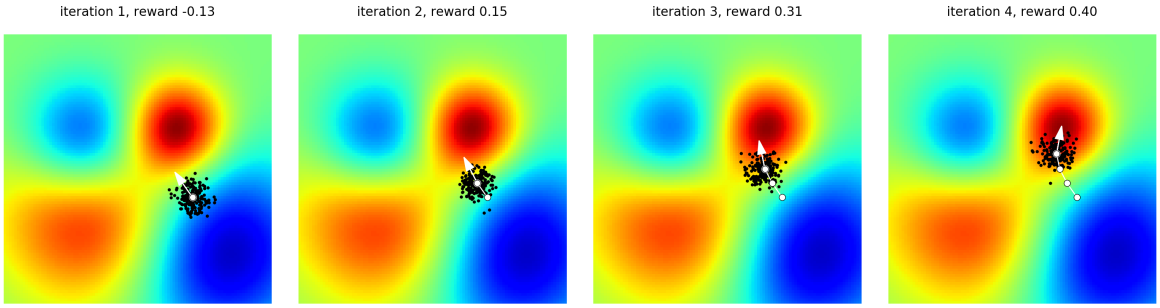
In simple words, evolution strategies are type of black box optimization where we are trying to directly change the parameters based on rewards that we are getting back from the environment. So comparing to Reinforcement learning we still use reward as a training signal but instead of doing 'guess and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

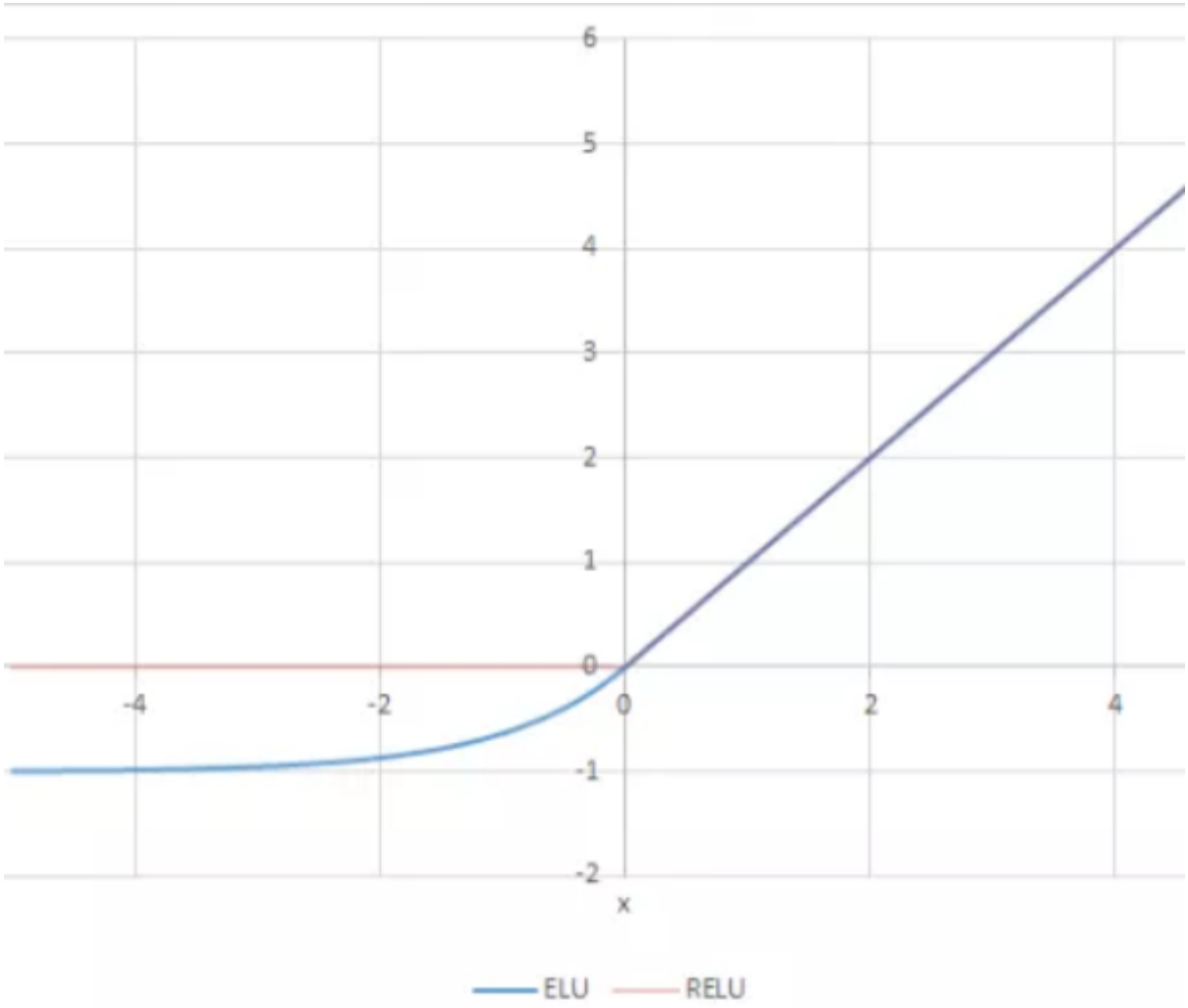
At the end one of the advantages over Reinforcement learning is that exploration is done out of the box. I would say that type of learning is much more creative because you cover much broader set of solutions comparing to gradient descent which is always aiming towards one specific solution. Also we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

Solution

So to make this work here is the list of steps I took:

1) First thing I realized is that with less neurons I can make NN generalizing better. Only two linear layers with **100 neurons** each was enough.

2) Next was that solution converged much faster with **'elu'** rather than 'relu'. I think that reason for that was that elu can activate negative inputs which in case of this game we have in form of -0.01 reward.



3) Then I added **Batch Normalization**. This environment is very stochastic and produces 'covariant shifts' (changes in the distribution of input values). For example game can start by throwing ball from different sides. Also once started it is hard to predict trajectory of the ball. Conclusion is that two games can differ a lot between each other. Batch Normalization is normalizing outputs of activations and therefore limits convariate shifts.
4) **Averaging scores over 100** tries for each member in population. I think this was a key since the game is very stochastic and sparse and we cannot really say that some member of population do good job if it scores well only after first try.
5) **Learning rate = 0.5**. Since we added Batch Normalization and did this averaging differences between reward signals were kinda small I needed to increase the learning rate.

Here is the whole list of hyperparams:

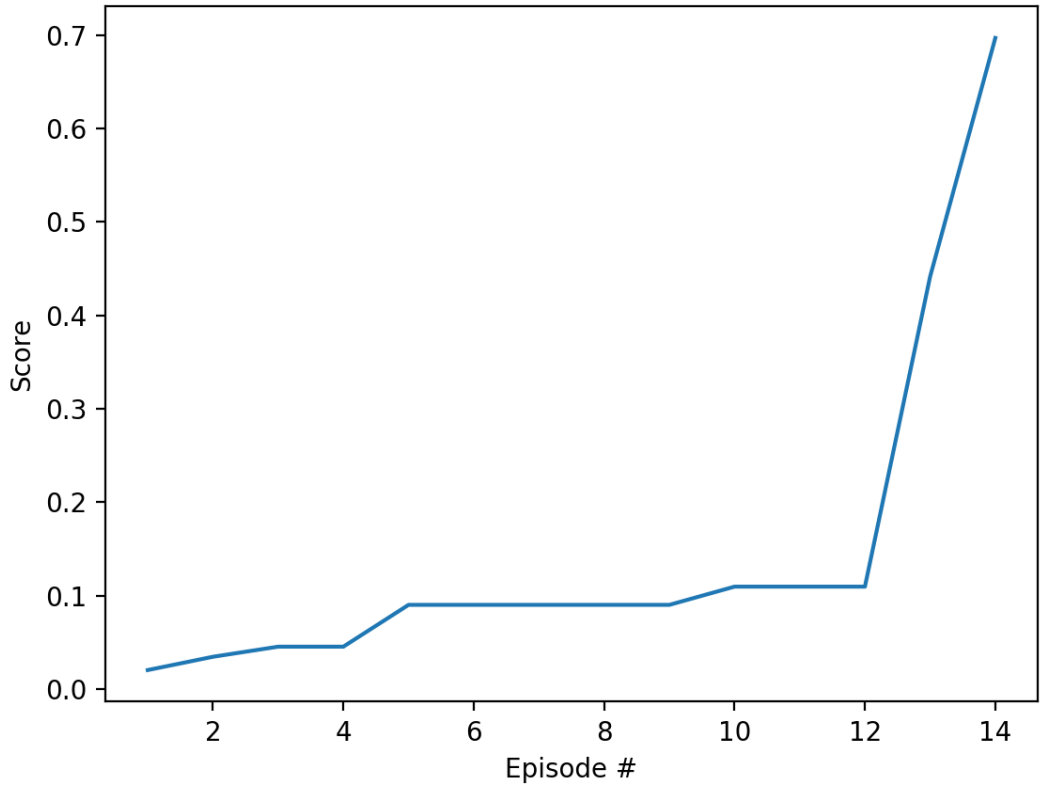
```
fc1_units = 100

fc2_units = 100

self.bn1 = nn.BatchNorm1d(fc1_units)

 self.linear1 = nn.Linear(inputs, fc1_units)

 self.linear2 = nn.Linear(fc1_units, outputs)

 self.linear3 = nn.Linear(fc2_units, outputs)

 self.population_size = 80

 self.sigma = 0.25

 self.learning_rate = 0.5
```

Final algorithm

Finally I was pretty happy to see this working. So the algorithm looks like this:

1) Form initial population by randomly initialize weights from normal distribution
2) For each member in population evaluate the game 100 times
3) Take the maximum of averages for both agents to form reward signal
4) Use reward signal as a weight for noise that is added for next generation
5) Repeat 2 to 4 until we don't get member who scores more than 0.5

After around 14 generations of training on CPU (2-3 hours) it was able to converge



Problems

Training was not that stable. It is obvious that it gets stuck for few generations in the same score. Sometimes it converges really fast (after 15 generations) but sometimes it needs more than 20 generations. Also the final solution looks a little bit noisy. The reason for that is that I was using the same model for both agents and introduced additional overhead by acting on both agents even if the ball is on specific side.

Ideas for improvement

Per agent model – I was not able to make two separate models working but I am sure that it can be more flexible and less noisy solution
Pretraining – This approach can be used as a pretraining to collect initial weights for some other algorithm.