QueensGambit added policy training and eval logs for NES `3ffe7b0` an hour ago

3 contributors

123 lines (78 sloc) 6.57 KB

Challenge 3

Authors: Johannes Czech, Jannis Weil, Fabian Otto

Vanilla Policy Gradient

In this section, we describe our experience with implementing vanilla policy gradients.

Our main focus for the implementation is the environment `Levitation-v1` from the Quanser platform.

Implementation and observations

The implementation can be found in the python module `Challenge_3.REINFORCE`.

We experimented a lot with with different policy types (discrete and continuous policies) and with different ways to weight the policy gradients (episode return, immediate rewards, discounting, baselines, normalization). However, we still had trouble learning a good policy for `Levitation-v1` (see issues).

Issues

The levitation environment appeared very trivial to us at first sight, but this appear not to be the case. The agent has a tendency to get stuck between -810 and -850 total reward. This holds true for many different learning algorithms such as PG, NPG, PPO with baseline parameter settings. Random actions give a similar reward with very little variance. That's why expect that there is a strong local optima. Moreover, since there's no rendering option it was hard to tell what the agent is learning or if he's learning something at all.

Results

By using a discrete action space with only two bins and small neural network with 8 hidden units, it was able to overcome this optimum. However, this came at the cost of prematurely ending the episode.

```
Eval (198 episodes): -38.8739 +/- 2.7076 (253.5606 +/- 16.5262 steps)
```

Natural Policy Gradient

In this section, we describe our experience with implementing NPG.

NPG was tested out on the `BallBalancerSim-v0` environment form the Quanser platform.

Implementation and observations

The implementation can be found in the python module `Challenge_3.NPG`.

We first tried to implement NPG as a simple extension of REINFORCE by sampling the fisher information matrix from the policy gradient,

$$\hat{F}_{\theta_k} = \frac{1}{T}\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \nabla_\theta \log \pi_\theta(a_t|s_t)^T$$

but we soon realized that this is too computationally inefficient.

We then build upon the solution by Woongwon Lee et al., where the natural gradient step is performed by calculating the fisher information matrix using the second derivative of the KL-divergence (see this blog post from Boris) and then using conjugate gradient instead of explicitly creating the inverse fisher information matrix.

This way, we are able to achieve good results.

Issues

As the plain npg implementation uses a fixed alpha for the parameter update, we tried to improve it by using the "normalized" step size as menioned in the paper from Rajeswaran et al.:

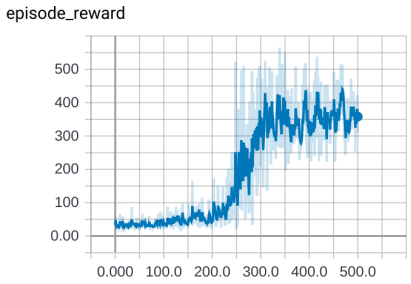
$$\theta_{k+1} = \theta_k + \sqrt{\frac{\delta}{g^T \hat{F}_{\theta_k}^{-1} g}} \hat{F}_{\theta_k}^{-1} g$$

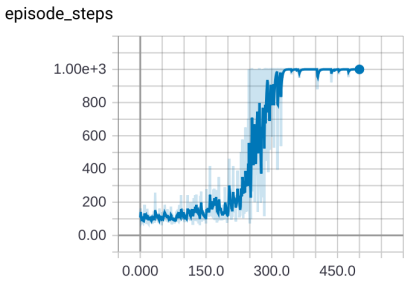
but unfortunately, we were not able to improve our policy in comparison to the plain update step.

Additionally, we had some problems when predicting the mean and the std of the policies normal distribution together with our policy network. Using an independent network parameter instead helped a lot and caused improved stability during training.

Results

We see stable training behaviour like the following on `BallBalancerSim-v0` very frequently:

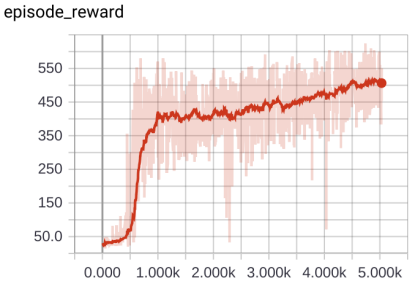




One can see that that the number of steps converge to the maximum episode steps (1000) and the reward increases until the maximum step size is reached. The illustrated policy receives an average reward of 365.0112 +/- 117.8526 (22 evaluation episodes).

Lowering the step size also lowers the variance in the policy, we get 403.4890 +/- 63.5127 (40 evaluation episodes) for `step_size=0.2` after training 1000 episodes.

Using better gradient estimates by evaluating more samples for each update step yields even better results, but also requires longer training times (especially when combined with the lower step size). Our final submission receives a cumulative reward of about 500.1526 +/- 60.1290 (100 episodes) but already takes approximately one hour to train. However, the training behaviour seems to be very stable (take a look at the following episode return plot) and one could most likely improve the policy even further with longer training.



Natural Evolution Strategies

In this section, we describe our experience with implementing NES.

NES was tested out on the `BallBalancerSim-v0` environment form the Quanser platform.

Implementation and observations

The implementation can be found in the python module `Challenge_3.NES`.

We experimented with different learning rate settings and exploration parameters. Training with no decay and choosing `sigma=1` helped the most for our experiments. We found that including reward normalization significantly improved the progress of learning and almost immediately resulted in better rewards. In general, NES does not require larger networks, two hidden layers with 10 nodes where sufficient. Increasing nodes or layers did not improve the results. Smaller population also helped to maintain computational efficiency.

Issues

We found that NES does not change a lot in performance when adjusting hyperparameters for `BallBalancerSim-v0`. Almost all settings end up at a policy, which tries to keep the ball in place without moving it. Even longer training times, larger networks and smaller learning rate did not change this. This could be result of a local optima, because moving to strongly in the wrong direction almost immediately results in a worse reward.

The Monitor env does not support thread pool's mapping function, therefore the training time might be longer, because we can only use one worker in the submission.

Results

For evaluation, we achieve with almost all hyperparameter setting a cumulative reward about 371.7286 +/- 29.39473 (1000.0000 +/- 0.0000 steps).