Reinforcement Learning Implementation Project Report

Alessio Ragno

Master's Degree in Artificial Intelligence and Robotics

Department of Computer, Control and Management Engineering

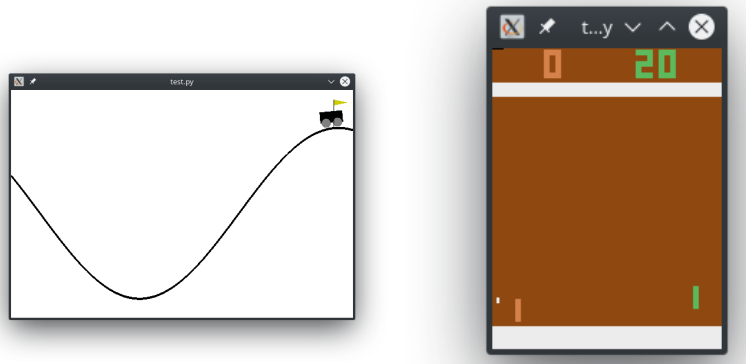
Sapienza University of Rome

December 2019

Contents

1 Introduction

2 Additional Tests and Conclusions



(a) MountainCar-v0

(b) Pong-v4

Figure 1: Environments

What follows is the report for the Reinforcement Learning Implementation Project of Machine Learning Class at Artificial Intelligence and Robotics Course.

The project consists in the implementation of an assigned Reinfocement Learning algorithm and its evaluation in two different environments.

The assigned algorithm is Trust Region Policy Optimization (TRPO) and it has to be evaluated on the following OpenAI Gym environments:

- **MountainCar-v0** is an environment in which a car has to climb a hill and reach a specific point indicated by a flag. MountainCar-v0 is a continuous state space environment that can be controlled with discrete actions. At each timestep the agent can choose between three actions: pushing the car left, right or not applying any force to the car.

- **Pong-v4** is an environment in which the agent has to play Pong game against a virtual player. The state is the image of the game and the agent acts using the Atari 2600 commands: NOOP, FIRE, RIGHT, LEFT, RIGHTFIRE, LEFTFIRE. For this project Pong-v4 has been used instead of Pong-v0 since the former is deterministic.

3 Trust Region Policy Optimization

Trust Region Policy Optimization was published by John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan and Pieter Abbeel on April 2017. TRPO is an improvement of the vanilla Policy Gradient algorithm implemented by REINFORCE with baseline. TRPO aims to make gradient step which are not "too big" to improve the algorithm stability. The length of the step is defined in terms of a hyperparameter δ which represents the radius of the trust region.

Algorithm 1 TRPO

for iteration = 1,2,... **do**
  Run policy for T timesteps or N trajectories
  Estimate advantage function at all timesteps
  for $t = 1$, T **do**
    maximize $\sum_{n=1}^{N} \frac{\pi_\theta(a_n|s_n)}{\pi_{\theta_{old}}(a_n|s_n)} A_n$
    s.t. $\overline{KL}_{\pi_{\theta_{old}}}(\pi_\theta) < \delta$
  end for
end for

Algorithm 2 TRPO

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
Hyperparameters: KL-divergence limit δ, backtracking coefficient α, maximum number of backtracking steps K
for $k = 0,1,2, ...$ **do**
    Collect set of trakectories $D_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment
    Compute rewards-to-go R_t
    Compute advantage estimates, A_t (using any method of advantage estimation) based on the current value function V_{ϕ_k}
    Estimate policy gradient as

$$g_k = \frac{1}{D_k} \sum_{\tau \in D_k} \sum_{t=0}^{T} \nabla_\theta log \pi_\theta(a_t|s_t)|_{\theta_k} A_t$$

    Use the conjugate gradient algorithm to compute

$$x_k = H_k^{-1} g_k$$

    where H_k is the Hessian of the sample average KL-divergence
    Update the policy by backtracking line search with

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{x_k^T H_k x_k}} x_k$$

    where $j \in \{0, 1, 2, ..., K\}$ is the smallest value which improves the sample oss and satisfies the sample KL-divergence constraint
    Fit the value function by regression on mean-squared-error:

$$\phi_{k+1} = argmin_\phi \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0} (V_\phi(s_t) - R_t)^2$$

    tipically via some gradient descent algorithm
end for

Algorithm 1 shows the basic TRPO idea in pseudocode: It actually consists in a Vanilla Policy Gradient with the constraint of taking only "improving" steps.

To implement TRPO, the authors suggest to maximize the vanilla policy gradient (which they call surrogate loss) and at the same time minimize the KL divergence calculating the hessian vector product with the surrogate loss

gradient.

After generating the step direction using conjugate gradients, the algorithm uses linesearch to find the right stepsize.

Applying the suggested calculation we obtain Algorithm 2.

In the paper, the authors showed the hyperparameters used in each environment and it turned out that using $K = 10$ maximum backtracking steps for the line search is sufficient.

Increasing δ the algorithm becomes more "aggressive" and at the same time unstable, while decreasing it, the algorithm learns slower but it is more cautious. The authors showed that using $\delta = 0.01$ is a good tradeoff between velocity and stability.

Since the calculation of the gradients, the conjugate gradient and the linesearch make this algorithm computationally expensive, a more efficient and improved version was later published as Proximal Policy Optimization (PPO) by Schulman et al..

4 Model Training and Evaluation

This section shows the results obtained in the two environment mentioned above. For MountainCar-v0 using a fully connected was sufficient while for Pong-v0, which gives an image as observation, some tricks were necessary.

The code was fully written in Tensorflow 2.0 using Keras APIs for the construction of the Neural Networks. In Tensorflow 2.0, contrary to Tensorflow 1.x, eager execution is enabled by default and the graph is not kept for the whole execution but only built when needed. This led to multiple nested functions for the calculation of the gradients.

4.1 MountainCar-v0

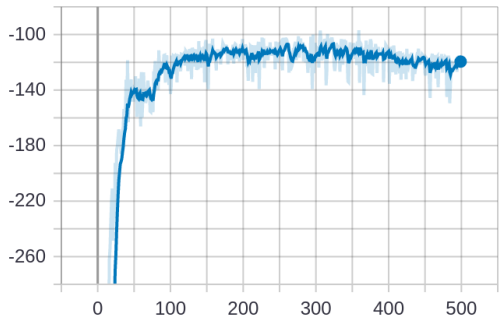


Figure 2: MountainCar-v0

In MountainCar, a car has to climb a hill and reach a goal point taking advantage of the cumulated momentum. At each non-goal step, the agent is given a reward equal to -1 while when the goal is reached the episode is ended. The observation consists in the current position and velocity of the car.

The gym environment by default sets the maximum number of timesteps to 200. This turned out to be too low since the agent was not capable to reach the goal "randomly" within 200 steps. To ensure proper exploration the limit has been then increased to 1600.

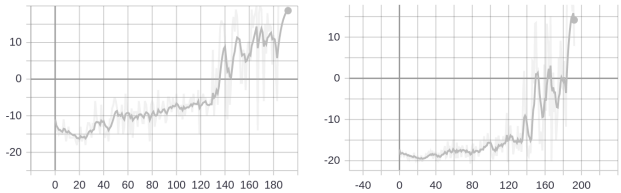
Although this trick helped a little, the agent wasn't still capable of reaching the goal state enough times to learn a good policy. To solve this problem a "correlated ϵ-greedy" algorithm for exploration was added to the policy: With probability ϵ the agent picks a random action, which is, with probability 0.8 the same of the last action. This increased a lot exploration, since, mostly in this environment, it is very likely that the best action to take at the next step is the same of the just taken one.

The agent was trained for 340 episodes and it converged to a mean total reward of -110 (it takes about 110 steps to reach the goal state).

Figure 2 is the plot of the mean total reward over the different training epochs.

It is possible to test this environment using the saved model *"saved_models/MountainCar-v0/340.ckpt"* with the script "test.py". Also intermediate models have been saved.

4.2 Pong-v4



(a) Pong-v4 (b) Pong-v4

Figure 3: Pong-v4

In Pong, the agent has to play Pong game using a simulated Atari 2600 console. The observation consists in the image perceived by the agent while the reward is the difference between the agent score and the opponent one.

The paper reports that for training Atari environments it took about 30hrs for them. Since a gpu wasn't available, the observation needed to be

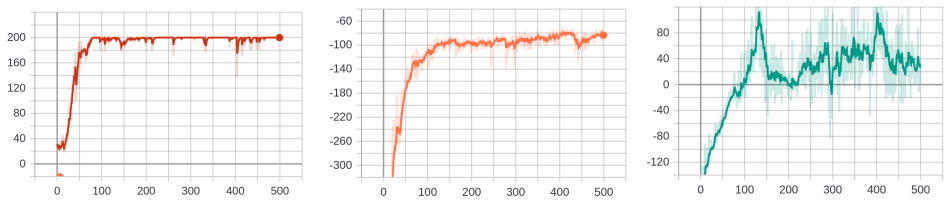
preprocessed. To make the learning process not too long, the agent was given as observation 3 integers which represent the position of the ball in the image and the height of the agent's stick. This preprocessing speeded up not only the training phase but also the sampling one. Not only the state space was reduced, but also the action one: not all the actions are necessary, so I decided to limit the action space to RIGHT and LEFT.

Moreover, to train Atari environments, the authors suggest to use vine sampling which consists in saving a state and performing different actions starting from the same state. This is unfortunally impossible to apply on all environments since it requires restoring a previous state and it is also very computationally expensive.

Altough reducing the input space to 3 integers helped a lot, the exploration phase was still too long. I decided to give it a hand by applying the optimal policy instead of a random one with ϵ-greedy. This helped a lot, the algorithm was not only able to see a lot of space but it also learnt very quickly.

Figure 3 is the plot of the mean total reward and the max reward over the different training epochs. It is not possible to say that this environment has been solved since it doesn't actually reach a mean reward of 21, but given the complexity of the game and that the calculator architecture was probably not good as the authors one, these are quite good results. It is possible to test this environment using the saved model *"saved_models/Pong-v4v0/340.ckpt"* with the script "test.py". Also intermediate models have been saved.

5 Additional Tests and Conclusions



(a) CartPole-v0 (b) Acrobot-v1 (c) LunarLander-v2

Figure 4: Additional Environments

Some additional tests have been run over different environments. In particular, three more environments have been tried out. Figure 4 shows the reward plots of the training phases of CartPole-v0, Acrobot-v1 and LunarLander-v2. Also for these three environments pretrained models are given.