Reinforcement Learning Report:
Proximal Policy Optimization in discrete action space

Coding Framework: Python3, PyTorch
Testing environment: MountainCar (https://drive.google.com/open?id=1qmAjzxgDGyQuYP1jQlQ5p-NHqlD2c5-H)
Experimental environment: Pong (https://drive.google.com/open?id=1G34hFiMEb4PNS0Xv9NBfyeWQUBxrPwau)
Simone Rossetti

PPO algorithm is a new kind of policy gradient method for reinforcement learning, in which this kind of methods are an appealing approach because they directly optimize the cumulative reward and can straightforwardly be used with nonlinear function approximators such as neural networks.
PPO alternates between sampling data through interaction with the environment, and optimizing a surrogate objective function using stochastic gradient ascendent.
The algorithm shown in the picture below is a generic representation of a proximal policy optimization method.
Whereas standard policy gradient methods perform one gradient update per data sample, it proposes a different objective function that enables multiple epochs of mini-batch updates.

Algorithm

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

PPO trains a policy in multiple environments exploiting interaction data of multiple agents.
This algorithm is divided in two main cycles.
First step of the algorithm is to collect, for T time steps, online data about actual policy approaching to the environment. The agent simply ‘acts’ in the surround.

The actions are picked with some initial probability from the policy by the agent, so that, at the beginning, actions are chosen randomly in a complete unknown of the result.
For each action picked, the policy computes the value function for that action, which means that the policy expresses to the agent how good is the state in which now the agent is in (the expected total reward for an agent starting from state).

The agent execute the action chosen in the environment, the action modifies the state in the environment and new percepts are collected.
So we collect causes and effects for each action in their relative states (data of value function, rewards and probabilities for each actions).

At the end of this first empirical step, the algorithm computes the generalized advantage estimation GAE, to estimate for each action if it improves the value function for the next state (if the action has taken the agent in a better state).

For each result collected by the algorithm with respect to each action, it computes delta, which is the sum of current reward and the expected success of the next state (next state value minus current state value), zero if 'done' is reached.
So, it computes the sum over all collected states, until last is reached.

GAE is computed summing all GAE of previous actions. It is higher if near actions succeed, lower otherwise. At the end it sums again the value of current state, so that actions in different states obtain bonus or malus if a better states are reached thanks to those actions.

$$\begin{aligned}\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} &:= (1-\lambda)\left(\hat{A}_t^{(1)} + \lambda\hat{A}_t^{(2)} + \lambda^2\hat{A}_t^{(3)} + \dots\right)\\ &= (1-\lambda)(\delta_t^V + \lambda(\delta_t^V + \gamma\delta_{t+1}^V) + \lambda^2(\delta_t^V + \gamma\delta_{t+1}^V + \gamma^2\delta_{t+2}^V) + \dots)\\ &= (1-\lambda)(\delta_t^V(1+\lambda+\lambda^2+\dots) + \gamma\delta_{t+1}^V(\lambda+\lambda^2+\lambda^3+\dots)\\ &\quad + \gamma^2\delta_{t+2}^V(\lambda^2+\lambda^3+\lambda^4+\dots) + \dots)\\ &= (1-\lambda)\left(\delta_t^V\left(\frac{1}{1-\lambda}\right) + \gamma\delta_{t+1}^V\left(\frac{\lambda}{1-\lambda}\right) + \gamma^2\delta_{t+2}^V\left(\frac{\lambda^2}{1-\lambda}\right) + \dots\right)\\ &= \sum_{l=0}^{\infty}(\gamma\lambda)^l\delta_{t+l}^V\end{aligned}$$

Now that all necessary data are collected, the second step of the algorithm analyzes and updates the weights of the model thanks to the Adam algorithm. At the end of the second step the epoch ends.

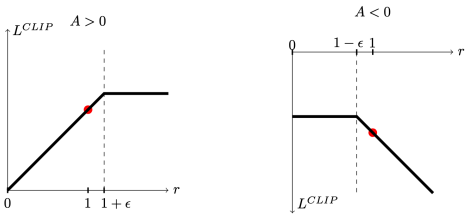
$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

In the second step, the policy is continuously trained for K times to understand which action is best suited for each possible percept that comes from the environment, and so to reduce the uncertainty related to the best action to pick in each possible state of the world in which the agent is; policy is updated iteratively by computing gradient updates according to a specific loss function.

At first, the algorithm grabs random mini-batches several times until it covers all data.
It takes a state in the batch, from the policy it computes how good is to stay in it, then it takes the action picked by the agent that generated that state and computes the new log probability for the action with respect to the current policy (which is updated during K iterations, so, first iteration for non visited new actions will have new log probability to 1).

Now, it computes the ratio (new_prob/old_prob) of variation of the probability of the action according to the new policy.

The algorithm computes the first surrogate function, which promotes the action that increased the advantage, due to its probability variation.
After that, it repeats the same computation as before but cutting the ratio values out of a certain range to avoid hysteresis, this is called CLIP, and is made to penalize changes to the policy that move the ratio away from 1. It only ignores the change in probability ratio when it would make the objective improve, and it includes the ratio when ratio makes the objective worse. That's why this variant of algorithm is called PPO Clip.



$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \mathrm{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right]$$

At this point, the algorithm takes the minimum of the two surrogate and calculates the mean, it computes the 'surrogate loss' in which includes only actions that decrease the performances of the actor.
It also computes a squared loss for the actor, that is the squared loss of the reward of the action in the state with respect to the policy estimation of the action in the state.

Now, the algorithm can compute the loss function, adding the clipped actor loss and the squared critic loss, summing also some entropy bonus, to guarantee the model to promote exploration beyond exploitation.

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{CLIP}(\theta) - c_1L_t^{VF}(\theta) + c_2S[\pi_\theta](s_t)\right],$$

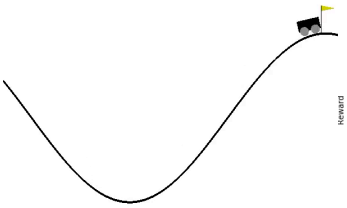
At the end of each step, the algorithm updates the weights of the model thanks to the gradient, generated thanks to Adam algorithm.

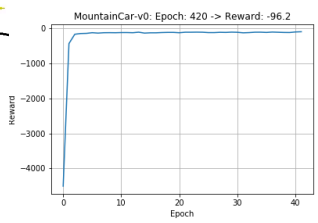
$$\hat{g} = \hat{\mathbb{E}}_t\left[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\hat{A}_t\right]$$

The policy simply 'critics' actions of the agent. That's why this PPO algorithm is called, 'Actor-Critic Style'.

MountainCar

Code, video and plot at: https://drive.google.com/open?id=1qmAjzxgDGyQuYP1jQlQ5p-NHqlD2c5-H






```
Epoch: 10 -> Reward: -4763.6
Epoch: 20 -> Reward: -624.1
Epoch: 30 -> Reward: -179.1
Epoch: 40 -> Reward: -132.3
Epoch: 60 -> Reward: -123.9
Epoch: 70 -> Reward: -118.1
Epoch: 160 -> Reward: -114.3
Epoch: 280 -> Reward: -106.3
Epoch: 330 -> Reward: -104.8
Epoch: 370 -> Reward: -101.2
Epoch: 390 -> Reward: -97.4
Epoch: 420 -> Reward: -96.2
```

```
LinearModel(
  (critic): Sequential(
    (0): Linear(in_features=2, out_features=256, bias=True)
    (1): ReLU()
    (2): Linear(in_features=256, out_features=1, bias=True)
  )
  (actor): Sequential(
    (0): Linear(in_features=2, out_features=256, bias=True)
    (1): ReLU()
    (2): Linear(in_features=256, out_features=3, bias=True)
    (3): Softmax(dim=1)
  )
)
Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    eps: 1e-08
    lr: 0.01
    weight_decay: 0
)
```

MountainCar environment is very simple, for each step where the car does not reach the goal, located at position 0.5, the environment returns a reward of -1. So the aim is to reach the goal in less time, as to maximize the reward. Possible actions are ‘right’ +1, ‘stay’ 0, ‘left’ -1. Performing one of each action separately does not allow the agent to reach the goal. Instead, the model should learn such policy that permits the agent to exploit the required inertia to reach the top of the mountain.

Hyper-parameters have been chosen according to PPO research paper (E_CLIP, G_GAE, L_GAE, E_CLIP, C_2), others parameters according to some empirical tests.

Hyperparameter	Value
Learning Rate L	1e-02
GAE gamma	0.99
GAE lambda	0.95
Epsilon CLIP	0.2
C1	0.5
C2	0.01
PPO steps T	256
Mini-Batch size M	64
PPO epochs K	10
Actors N	8

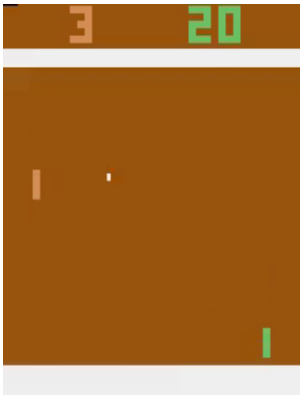
For instance, L_RATE = 1e-3 with ReLU gives very bad performances, too slow and inconsistent training; L_RATE = 1e-1 with ReLU gets stuck in single classification.
L_RATE = 1e-2 gives very good results, the model converges to such acceptable policy in only 40 epoch (very fast solution), reward > -150, after 400 epoch the model totalize a reward > -100.

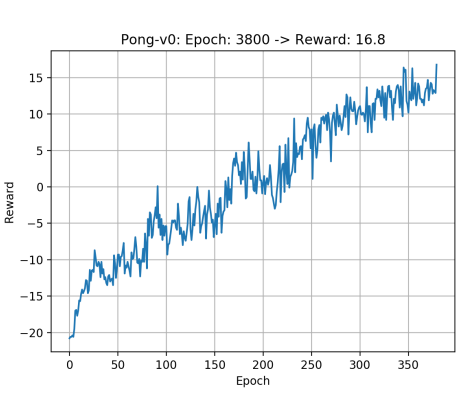
C_1 is lower from paper (C_1 = 1) because otherwise the classifier stucks in a single classification.
T, N, M have been chosen according to mini-batch size M <= N T.
K has been chosen to reach a good compromise between gradient updates and time requirements.

The policy model is a Neural Network with two 256 hidden size dense layers. It implements Adam algorithm. The layers are activated by non linear activation function ReLU. The first network, the critic, evaluate the state. The second network, the actor, estimate the most probable action to pick in different states.

Pong

Code, video and plot at: https://drive.google.com/open?id=1G34hFiMEb4PNS0Xv9NBfyeWQUBxrPwau

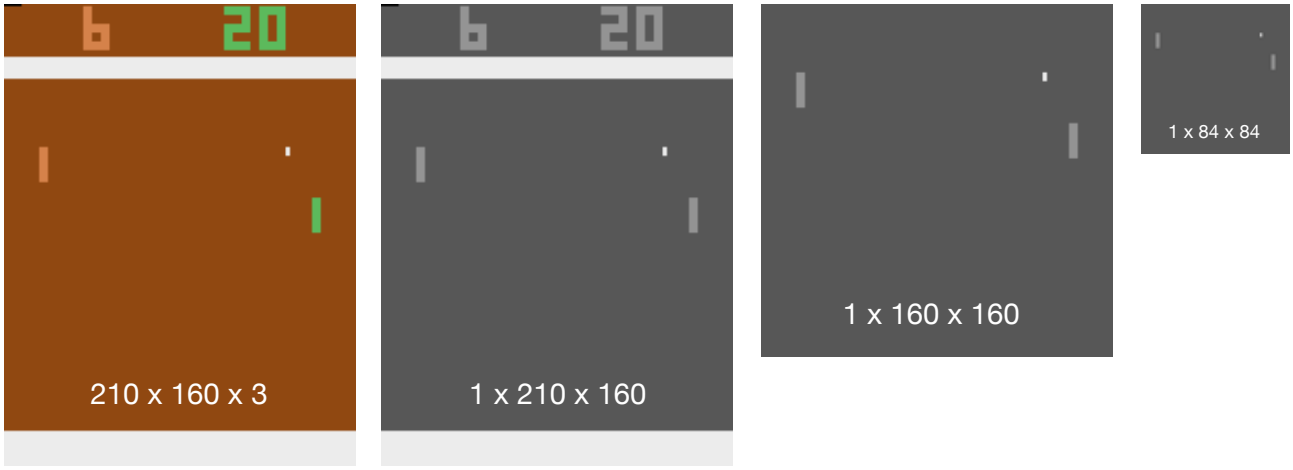





```
CNN(
  (critic): Sequential(
    (0): Conv2d(1, 16, kernel_size=(8, 8), stride=(4, 4))
    (1): ReLU()
    (2): Conv2d(16, 32, kernel_size=(4, 4), stride=(2, 2))
    (3): ReLU()
    (4): Flatten()
    (5): Linear(in_features=2592, out_features=256, bias=True)
    (6): ReLU()
    (7): Linear(in_features=256, out_features=1, bias=True)
  )
  (actor): Sequential(
    (0): Conv2d(1, 16, kernel_size=(8, 8), stride=(4, 4))
    (1): ReLU()
    (2): Conv2d(16, 32, kernel_size=(4, 4), stride=(2, 2))
    (3): ReLU()
    (4): Flatten()
    (5): Linear(in_features=2592, out_features=256, bias=True)
    (6): ReLU()
    (7): Linear(in_features=256, out_features=6, bias=True)
    (8): Softmax(dim=1)
  )
)
Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    eps: 1e-08
    lr: 1e-05
    weight_decay: 0
)
```

Pong environment is *210 x 160 x 3* pixel (*H x W x C*) in *RGB* (*3 channels*).
There are 6 possible actions in this environment, only 3 are useful: ‘stay’ 0, ‘up’ 2 and ‘down’ 5.
To keep track of our environment the model needs consistent representation of the environment.
Because of that, images have been reduced to a single channel (*greyscale*), then *cropped* along the playing area, *squared* and *reduced* by dimension. Resulting images of the states are *1 x 84 x 84* pixel (*C x H x W*).
Channels have been shifted with respect the axes because Convolutional Layers in PyTorch compute batch (*B*) of images in a different order (*B x C x H x W*).

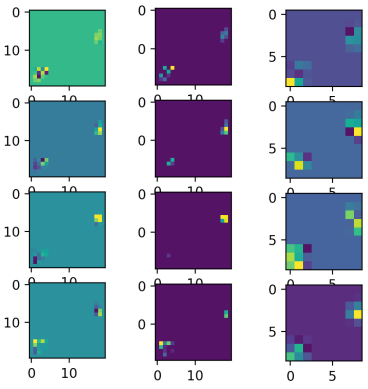
noop (0)	fire (1)	up (2)	right (3)	left (4)
down (5)	up-right (6)	up-left (7)	down-right (8)	down-left (9)
up-fire (10)	right-fire (11)	left-fire (12)	down-fire (13)	up-right-fire (14)
up-left-fire (15)	down-right-fire (16)	down-left-fire (17)	reset* (40)	



The input to the neural network consists in a batch (single or multiple) of *1 x 84 × 84* image produced by previous computations. The first hidden layer convolves *16* filters *8 × 8* with stride *4* with the input image and applies a rectifier nonlinearity. The second hidden layer convolves *32* filters *4 × 4* with stride *2*, again followed by a rectifier nonlinearity. The final hidden layer is fully-connected and consists of *256* rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action.
The first network, the critic, evaluate the state. The second network, the actor, estimate the most probable action to pick in different states.

Hyperparameter	Value
Learning Rate L	1e-05
GAE gamma	0.99
GAE lambda	0.95
Epsilon CLIP	0.2
C1	0.5
C2	0.01
PPO steps T	256
Mini-Batch size M	64
PPO epochs K	10
Actors N	8

In this environment, some hyper-parameters have been chosen according to PPO research paper (E_CLIP, G_GAE, L_GAE, E_CLIP, C_2), others according to some empirical tests.
For instance, L_RATE = 1e-4 with ReLU gives very bad performances, classifier immediately classify all the states with the same action with probability 1, all others 0 (L_RATE = 1e-4 with Tanh activation function for each layer gave very good results in very cheap time but in the middle of the training, the gradient exploded due to Tanh function); L_RATE = 1e-6 with ReLU returns too slow training.
C_1 is lower compared to the one on the official PPO paper (C_1 = 1) because otherwise the classifier stucks in a single classification.
T, N, M have been chosen according to mini-batch size M <= N T.
K has been chosen according to a good compromise between gradient updates and time requirements.



CNN ReLU feature maps in Pong

Conclusions

Proximal policy optimization algorithm performs very efficiently in discrete action spaces. This policy optimization method uses multiple epochs of stochastic gradient ascent to perform each policy update. This method is very stable and reliable, when using a joint architecture for the policy and value function, it gives better overall performances. This method has the faculty to learn very fast (fine tuning of the parameters). The algorithm could complete simple task in very minimal timing and could also succeed in a more complex dynamic task.

References

Proximal Policy Optimization - OpenAI - https://openai.com/blog/openai-baselines-ppo/

Proximal Policy Optimization Algorithm - John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov - https://arxiv.org/pdf/1707.06347.pdf

Proximal Policy Optimization documentation - OpenAI - https://spinningup.openai.com/en/latest/algorithms/ppo.html

High-Dimensional Continuous Control Using Generalized Advantage Estimation - John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan and Pieter Abbeel - https://arxiv.org/pdf/1506.02438.pdf

Playing Atari with Deep Reinforcement Learning - Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller - https://arxiv.org/pdf/1312.5602.pdf

Deep Reinforcement Learning: Pong from Pixels - Andrej Karpathy - http://karpathy.github.io/2016/05/31/rl/

Arcade Learning Environment Technical Manual (v.0.5.1) - Marlos C. Machado, Matthew Hausknecht, Marc G. Bellemare - https://github.com/openai/atari-py/blob/master/doc/manual/manual.pdf

Mountain Car & Pong Environments - OpenAI - https://gym.openai.com/envs/