Continuous Deep Q-Learning with Model-based Acceleration

Shixiang Gu[1 2 3] SG717@CAM.AC.UK
Timothy Lillicrap[4] COUNTZERO@GOOGLE.COM
Ilya Sutskever[3] ILYASU@GOOGLE.COM
Sergey Levine[3] SLEVINE@GOOGLE.COM
[1]University of Cambridge [2]Max Planck Institute for Intelligent Systems [3]Google Brain [4]Google DeepMind

【强化学习 52】NAF

张楚珩
清华大学 交叉信息院博士在读

NAF是normalized advantage function的简称，可以看做是连续控制版本的Q-learning。

原文传送门

Gu, Shixiang, et al. "Continuous deep q-learning with model-based acceleration." International Conference on Machine Learning. 2016.

特色

本文提出了NAF，它可以被看做是连续控制版本的Q-learning；同时本文还提出了Imaginary rollout技术，使得学到的model可以用来加速model-free算法。

过程

1. NAF

回顾Q-learning，就是通过最小化bootstrap的目标和Q函数之间的距离，来训练得到optimal Q function。

$$L(\theta^Q) = \mathbb{E}_{\boldsymbol{x}_t \sim \rho^\beta, \boldsymbol{u}_t \sim \beta, r_t \sim E}[(Q(\boldsymbol{x}_t, \boldsymbol{u}_t|\theta^Q) - y_t)^2]$$
$$y_t = r(\boldsymbol{x}_t, \boldsymbol{u}_t) + \gamma Q(\boldsymbol{x}_{t+1}, \boldsymbol{\mu}(\boldsymbol{x}_{t+1}))$$

要想在连续控制问题上使用Q-learning，最大的一个问题是Q函数对应的greedy policy是很难直接得到的。

$$\boldsymbol{\mu}(\boldsymbol{x}_t) = \arg\max_{\boldsymbol{u}} Q(\boldsymbol{x}_t, \boldsymbol{u}_t)$$
$$\pi(\boldsymbol{u}_t|\boldsymbol{x}_t) = \delta(\boldsymbol{u}_t = \boldsymbol{\mu}(\boldsymbol{x}_t))$$

文章选择了一个greedy policy能够直接被表示的Q函数表示方法

$$
\begin{aligned}
Q(\boldsymbol{x}, \boldsymbol{u}|\theta^Q) &= A(\boldsymbol{x}, \boldsymbol{u}|\theta^A) + V(\boldsymbol{x}|\theta^V) \\
A(\boldsymbol{x}, \boldsymbol{u}|\theta^A) &= -\frac{1}{2}(\boldsymbol{u} - \boldsymbol{\mu}(\boldsymbol{x}|\theta^\mu))^T \boldsymbol{P}(\boldsymbol{x}|\theta^P)(\boldsymbol{u} - \boldsymbol{\mu}(\boldsymbol{x}|\theta^\mu))
\end{aligned}
$$

其中 $P(x|\theta^P) = L(x|\theta^P)L(x|\theta^P)^T$ ，而 $L(x|\theta^P)$ 是一个下三角矩阵，即它有 $n(n+1)/2$ 个自由度。设定一个神经网络（dueling network structrue），输入一个状态，输出 $V(x), \mu(x), L(x)$ 并得到Q函数，通过Q-learning的形式来学习这些参数。

Algorithm 1 Continuous Q-Learning with NAF

Randomly initialize normalized Q network $Q(\boldsymbol{x}, \boldsymbol{u}|\theta^Q)$.
Initialize target network Q' with weight $\theta^{Q'} \leftarrow \theta^Q$.
Initialize replay buffer $R \leftarrow \emptyset$.
for episode=1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $\boldsymbol{x}_1 \sim p(\boldsymbol{x}_1)$
  for t=1, T **do**
    Select action $\boldsymbol{u}_t = \mu(\boldsymbol{x}_t|\theta^\mu) + \mathcal{N}_t$
    Execute $\boldsymbol{u}_t$ and observe r_t and $\boldsymbol{x}_{t+1}$
    Store transition $(\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1})$ in R
    for iteration=1, I **do**
      Sample a random minibatch of m transitions from R
      Set $y_i = r_i + \gamma V'(\boldsymbol{x}_{i+1}|\theta^{Q'})$
      Update θ^Q by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(\boldsymbol{x}_i, \boldsymbol{u}_i|\theta^Q))^2$
      Update the target network: $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
    end for
  end for
end for

值得一提的是，在策略的探索方面，这里并没有像很多其他文章一样使用每步独立的高斯噪声，因为多步的高斯噪声会在连续的几步中相互抵消；为了解决这个问题，有些其他的文章使用Ornstein-Uhlenbech process。本文在这里则直接利用了已经估计出来的二次型系数矩阵 $P(x|\theta^P)$ ，让策略更加有方向性地探索。

$$\pi(\boldsymbol{u}|\boldsymbol{x}) = \exp^{Q(\boldsymbol{x},\boldsymbol{u}|\theta^Q)} / \int \exp^{Q(\boldsymbol{x},\boldsymbol{u}|\theta^Q)} \, d\boldsymbol{u}$$
$$= \mathcal{N}(\boldsymbol{\mu}(\boldsymbol{x}|\theta^\mu), c\boldsymbol{P}(\boldsymbol{x}|\theta^P)^{-1}).$$

Q. 为什么这里估计的A函数不需要做类似ACER中Stochastic Dueling Network那样的操作以保证学到的A函数就是advantage呢？注意advantage需要满足 $\sum_a \pi(a|s)A(s,a)=0$ 。

A. 这里的A函数定义上有 $A(s,\mu(s|\theta^\mu))=0$ ，等于也是对它有了一个“锚定”，虽然不是锚定到我们通常定义的A函数的位置，但是学习的过程中不至于“飘走”；再说，最后只是学习Q函数，A函数少的那一截会被V函数学到。

2. Imaginary Rollout

NAF使得在连续控制问题上能够使用Replay Experience（RE），这样就自然产生了一种使用学习到的model来提高sample efficiency的方法，即使用model来产生experience，并加入RE中供算法去学习。使用model产生的轨迹称作imaginary rollout（IR）。本文的实验说明了一下几个有意思的点。

- 即使使用真实model，IR如果只使用好的policy去产生，效果会比不加IR更差；IR如果去产生on-policy的rollout能提高sample efficiency；
- 使用估计的model，如果使用NN直接估计，效果并不好，如果使用iLQG估计，效果还不错；
- 在使用iLQG估计产生IR的同时，如果真实rollout也混入一定比例（或者全部的）iLQG控制器产生的optimal control，效果也不错；
- 如果真实的rollout全部由iLQG产生，还有额外的好处，就是能够保证产生的动作都比较safe，在实际中不会损害机器人硬件。

算法如下

Algorithm 2 Imagination Rollouts with Fitted Dynamics and Optional iLQG Exploration

Randomly initialize normalized Q network $Q(\boldsymbol{x}, \boldsymbol{u}|\theta^Q)$.
Initialize target network Q' with weight $\theta^{Q'} \leftarrow \theta^Q$.
Initialize replay buffer $R \leftarrow \emptyset$ and fictional buffer $R_f \leftarrow \emptyset$.
Initialize additional buffers $B \leftarrow \emptyset, B_{old} \leftarrow \emptyset$ with size nT.
Initialize fitted dynamics model $\mathcal{M} \leftarrow \emptyset$.
for $episode = 1, M$ **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $\boldsymbol{x}_1$
  Select $\mu'(\boldsymbol{x}, t)$ from $\{\mu(\boldsymbol{x}|\theta^\mu), \pi_t^{iLQG}(\boldsymbol{u}_t|\boldsymbol{x}_t)\}$ with probabilities $\{p, 1-p\}$
  for $t = 1, T$ **do**
    Select action $\boldsymbol{u}_t = \mu'(\boldsymbol{x}_t, t) + \mathcal{N}_t$
    Execute $\boldsymbol{u}_t$ and observe r_t and $\boldsymbol{x}_{t+1}$
    Store transition $(\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1}, t)$ in R and B
    if $\mod(episode \cdot T + t, m) = 0$ and $\mathcal{M} \neq \emptyset$ **then**
      Sample m $(\boldsymbol{x}_i, \boldsymbol{u}_i, r_i, \boldsymbol{x}_{i+1}, i)$ from B_{old}
      Use $\mathcal{M}$ to simulate l steps from each sample
      Store all fictional transitions in R_f
    end if
    Sample a random minibatch of m transitions $I \cdot l$ times from R_f and I times from R, and update $\theta^Q, \theta^{Q'}$ as in Algorithm 1 per minibatch.
  end for
  if B_f is full **then**
    $\mathcal{M} \leftarrow$ FitLocalLinearDynamics(B_f) (see Section 5.3)
    $\pi^{iLQG} \leftarrow$ iLQG_OneStep($B_f, \mathcal{M}$) (see appendix)
    $B_{old} \leftarrow B_f, B_f \leftarrow \emptyset$
  end if
end for

3. iLQG

iLQG大致思想是使用均值为二次型的高斯分布对轨迹附近的dynamics建模，建模方式为对于得到的轨迹拟合得到线性模型的系数 f_{xt}, f_{ut}, F_t ，有了该模型就可以运行on-policy的策略并且得到相应的lR了。

如果需要使用iLQG的最优控制，那么需要运行以下程序，最优控制可以解析地表示

$$\text{dynamics } p(\boldsymbol{x}_{t+1}|\boldsymbol{x}_t, \boldsymbol{u}_t) = \mathcal{N}(\boldsymbol{f}_{xt}\boldsymbol{x}_t + \boldsymbol{f}_{ut}\boldsymbol{u}_t, \boldsymbol{F}_t)$$

$$\text{time-varying linear feedback controller } \boldsymbol{g}(\boldsymbol{x}_t) = \hat{\boldsymbol{u}}_t + \boldsymbol{k}_t + \boldsymbol{K}_t(\boldsymbol{x}_t - \hat{\boldsymbol{x}}_t)$$

$$\text{linear-Gaussian controller } \pi_t^{iLQG}(\boldsymbol{u}_t|\boldsymbol{x}_t) = \mathcal{N}(\hat{\boldsymbol{u}}_t + \boldsymbol{k}_t + \boldsymbol{K}_t(\boldsymbol{x}_t - \hat{\boldsymbol{x}}_t), -cQ_{\boldsymbol{u},\boldsymbol{u}t}^{-1})$$

$$\text{where } \boldsymbol{k}_t = -Q_{\boldsymbol{u},\boldsymbol{u}t}^{-1}Q_{\boldsymbol{u}t}, \boldsymbol{K}_t = -Q_{\boldsymbol{u},\boldsymbol{u}t}^{-1}Q_{\boldsymbol{u},\boldsymbol{x}t}$$

$$\begin{aligned}
Q_{\boldsymbol{xu},\boldsymbol{xu}t} &= r_{\boldsymbol{xu},\boldsymbol{xu}t} + \boldsymbol{f}_{\boldsymbol{xu}t}^T V_{\boldsymbol{x},\boldsymbol{x}t+1}\boldsymbol{f}_{\boldsymbol{xu}t} \\
Q_{\boldsymbol{xu}t} &= r_{\boldsymbol{xu}t} + \boldsymbol{f}_{\boldsymbol{xu}t}^T V_{\boldsymbol{x},\boldsymbol{x}t+1} \\
V_{\boldsymbol{x},\boldsymbol{x}t} &= Q_{\boldsymbol{x},\boldsymbol{x}t} - Q_{\boldsymbol{u},\boldsymbol{x}t}^T Q_{\boldsymbol{u},\boldsymbol{u}t}^{-1} Q_{\boldsymbol{u},\boldsymbol{x}t} \\
V_{\boldsymbol{x}t} &= Q_{\boldsymbol{x}t} - Q_{\boldsymbol{u},\boldsymbol{x}t}^T Q_{\boldsymbol{u},\boldsymbol{u}t}^{-1} Q_{\boldsymbol{u}t} \\
Q_{\boldsymbol{x},\boldsymbol{x}T} &= V_{\boldsymbol{x},\boldsymbol{x}T} = r_{\boldsymbol{x},\boldsymbol{x}T}
\end{aligned}$$

其中Q、V、r分布代表Q函数、V函数和reward，后面五个式子可以通过动态规划从后往前算出来，其中下标代表导数，并列的 xu 下标代表对于状态和控制组成向量的导数。

iLQG有以下一些劣势

- iLQG假设了dynamics和reward的单模形态，在某些任务上可能学得更慢或者学到的策略更差；
- iLQG是对于常访问的路径附近的动力学做线性化，因此假设了初始状态分布比较窄，对于不适应这一假设的问题不能适用；
- iLQG需要已知reward关于state的表达式，对于完全黑盒的reward不适用。

（更多关于iLQG可以参考本专栏之前的文章iLQG）

对比ACER

- NAF使用的是局部拟合的advantage函数，ACER用神经网络对advantage函数进行全局拟合，相应的代价是需要使用SDN（但它们都需要另外拟合V函数）；
- NAF主循环是遵循Bellman operator for control，ACER价值函数估计是用Bellman operator for policy evaluation，总体上是policy iteration的思路，因此后面还需要做policy gradient；

发布于 2019-04-07