Compreendendo os métodos críticos do ator e A2C

Conceitos importantes na aprendizagem por reforço profundo

Chris Yoon 6 de fevereiro de 2019 · 6 min de leitura

Preliminares

Em meu post anterior , derivamos gradientes de política e implementamos o algoritmo REINFORCE (também conhecido como gradientes de política Monte Carlo). Existem, no entanto, alguns problemas gritantes com os gradientes de política convencionais: gradientes barulhentos e alta variância.

Mas por que isso acontece?

Lembre-se do gradiente de política:

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t]$$

Como no algoritmo REINFORCE, atualizamos o parâmetro de política por meio de atualizações de Monte Carlo (ou seja, tomando amostras aleatórias). Isso introduz uma

alta variabilidade inerente nas probabilidades de log (log da distribuição da política) e valores de recompensa cumulativos, porque cada trajetória durante o treinamento pode se desviar umas das outras em grandes graus.

Consequ um e ntly, a alta variabilidade em probabilidades de registro e valores de recompensa cumulativos fará gradientes ruidosos, e causar aprendizagem instável e / ou a distribuição de políticas inclinando para uma direção não-ideal.

Além da alta variância dos gradientes, outro problema com os gradientes das políticas ocorre: as trajetórias têm uma recompensa cumulativa de 0. A essência do gradiente das políticas é aumentar as probabilidades de ações "boas" e diminuir as de ações "ruins" na distribuição das políticas; tanto as ações "boas" quanto as "más" com não serão aprendidas se a recompensa cumulativa for 0.

No geral, essas questões contribuem para a instabilidade e a lenta convergência dos métodos de gradiente de política vanilla.

Melhorando os gradientes da política: reduzindo a variação com uma linha de base

Uma maneira de reduzir a variância e aumentar a estabilidade é subtrair a recompensa cumulativa por uma linha de base:

$$\text{Introducing baseline } b(s):$$

$$\nabla_\theta J(\theta) = \mathbb{E}[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)(G_t - b(s_t))]$$

Intuitivamente, tornar a recompensa cumulativa menor subtraindo-a com uma linha de base criará gradientes menores e, portanto, atualizações menores e mais estáveis.

Aqui está uma explicação muito ilustrativa, tirada da postagem de Jerry Liu em "Por que o método de gradiente de política tem alta variação" :

For the sake of hypothetical example, lets say that $\nabla_\theta log \pi_\theta(\tau)$ is $[0.5, 0.2, 0.3]$ respectively for three trajectories, and $r(\tau)$ is $[1000, 1001, 1002]$.

Then the variance of the product of the two terms for these three samples is $Var(0.5 \times 1000, 0.2 \times 1001, 0.3 \times 1002)$ which according to WolframAlpha is around 23286.8.

Now what if we reduce all values of $r(\tau)$ by a constant, say, 1001? Then the variance of the product becomes $Var(0.5 \times 1, 0.2 \times 0, 0.3 \times 1)$, which is 0.1633, a much smaller value.

[Retirado da postagem de Jerry Liu "Por que o método de gradiente de política tem alta variância"]

Resumo das funções de linha de base comuns

A linha de base pode assumir vários valores. O conjunto de equações abaixo ilustra as variantes clássicas dos métodos do ator crítico (com relação ao REINFORCE). Neste artigo, daremos uma olhada em Q Actor Critic e Advantage Actor Critic.

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\, G_t\right] && \text{REINFORCE} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; Q^w(s,a)\right] && \text{Q Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; A^w(s,a)\right] && \text{Advantage Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; \delta\right] && \text{TD Actor-Critic}
\end{aligned}
$$

Imagem retirada de slides de palestra CMU CS10703

Métodos de crítica do ator

Antes de retornarmos às linhas de base, vamos primeiro dar uma olhada no gradiente de política vanilla novamente para ver como a arquitetura do Ator Crítico entra (e o que realmente é). Lembre-se de que:

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t\right]$$

Podemos então decompor a expectativa em:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s_0, a_0, \ldots, s_t, a_t}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\right] \mathbb{E}_{r_{t+1}, s_{t+1}, \ldots, r_T, s_T}[G_t]$$

$$\sum_{t=0}$$

O segundo termo de expectativa deve ser familiar; é o valor Q! (Se você ainda não sabia disso, sugiro que leia sobre iteração de valor e aprendizado Q).

$$\mathbb{E}_{r_{t+1},s_{t+1},\ldots,r_T,s_T}[G_t] = Q(s_t, a_t)$$

Conectando isso, podemos reescrever a equação de atualização como tal:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s_0,a_0,\ldots,s_t,a_t}[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)]Q_w(s_t, a_t)$$

$$= \mathbb{E}_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)Q_w(s_t, a_t)]$$

Como sabemos, o valor Q pode ser aprendido parametrizando a função Q com uma rede neural (denotada pelo subscrito `w` acima).

Isso nos leva aos ***Métodos de Críticos do Ator,*** onde:

1. O "crítico" estima a função de valor. Este poderia ser o valor da ação (o *valor Q*) ou o *valor do* estado (o *valor V*).
2. O "Ator" atualiza a distribuição da política na direção sugerida pelo Crítico (por exemplo, com gradientes da política).

e ambas as funções Crítico e Ator são parametrizadas com redes neurais. Na derivação acima, a rede neural Crítica parametriza o valor Q - então, é chamada de ***Q Ator Crítico.***

Abaixo está o pseudocódigo para Q-Actor-Critic:

Algorithm 1 Q Actor Critic

Initialize parameters s, θ, w and learning rates α_θ, α_w; sample $a \sim \pi_\theta(a|s)$.
for $t = 1 \ldots T$: **do**
  Sample reward $r_t \sim R(s, a)$ and next state $s' \sim P(s'|s, a)$
  Then sample the next action $a' \sim \pi_\theta(a'|s')$

Then sample the next action $a' \sim \pi_\theta(a'|s')$
Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s,a)\nabla_\theta \log \pi_\theta(a|s)$; Compute the correction (TD error) for action-value at time t:
$\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s, a)$
and use it to update the parameters of Q function:
$w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$
Move to a $\leftarrow a'$ and s $\leftarrow s'$
end for

Adaptado da postagem “Algoritmos de gradiente de política” de Lilian Weng

Conforme ilustrado, atualizamos a rede crítica e a rede de valor em cada etapa de atualização.

Voltar para Baselines

Para fazer um argumento de autoridade (já que não fui capaz de descobrir o motivo), a função de valor de estado faz uma função de linha de base ótima. Isso é afirmado nos slides da aula Carnegie Mellon CS10703 e Berekely CS294, mas sem nenhum motivo fornecido.

Portanto, usando a função V como a função de linha de base, subtraímos o termo de valor Q com o valor V. Intuitivamente, isso significa*quão melhor é tomar uma ação específica em comparação com a ação geral média em um determinado estado*.Chamaremos esse valor de ***valor de vantagem*** *:*

$$A(s_t, a_t) = Q_w(s_t, a_t) - V_v(s_t)$$

Isso significa que temos que construir duas redes neurais para o valor Q e o valor V (além da rede de políticas)? Não. Isso seria muito ineficiente. Em vez disso, podemos usar a relação entre o Q e o V da equação de otimalidade de Bellman:

$$Q(s_t, a_t) = \mathbb{E}[r_{t+1} + \gamma V(s_{t+1})]$$

Portanto, podemos reescrever a vantagem como:

$$A(s_t, a_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)$$

Então, só temos que usar uma rede neural para a função V (parametrizada v acima). Portanto, podemos reescrever a equação de atualização como:

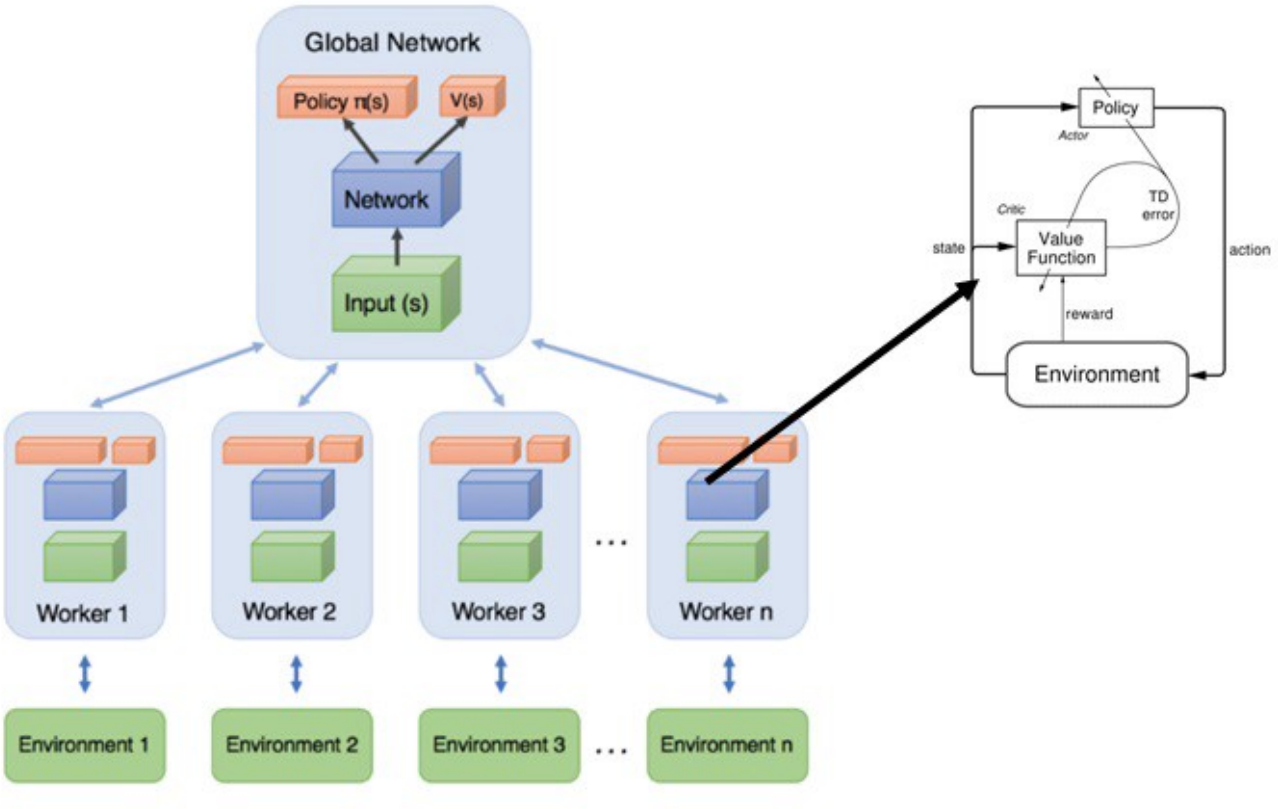
$$\nabla_\theta J(\theta) \sim \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)(r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t))$$
$$= \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t)$$

Este é o ***Critico do Ator de Vantagens.***

Critico do ator Advantage (A2C) vs Crítico do ator Advantage Assíncrono (A3C)

O ***Advantage Actor Critic*** tem duas variantes principais: o **Asynchronous Advantage Actor Critic (A3C)** e o **Advantage Actor Critic (A2C).**

A3C foi apresentado no artigo da Deepmind "Métodos Assíncronos para Aprendizagem por Reforço Profundo" (Mnih et al, 2016) . Em essência, o A3C implementa *treinamento paralelo* onde vários trabalhadores em *ambientes paralelos* atualizam *independentemente* uma função de valor *global* - portanto, "assíncrono". Um dos principais benefícios de ter atores assíncronos é a exploração eficaz e eficiente do espaço de estados.



Arquitetura de alto nível de A3C (imagem retirada da postagem do blog GroundAI)

A2C é como A3C, mas sem a parte assíncrona; isso significa uma variante do A3C para um único trabalhador. Foi descoberto empiricamente que A2C produz desempenho comparável ao A3C, embora seja mais eficiente. De acordo com esta postagem do blog OpenAI, os pesquisadores não têm certeza se ou como a assincronia beneficia o aprendizado:

> *Depois de ler o artigo, os pesquisadores de IA se perguntaram se a assincronia levou a um melhor desempenho (por exemplo, "talvez o ruído adicionado proporcionaria alguma regularização ou exploração?"), Ou se era apenas um detalhe de implementação que permitiu um treinamento mais rápido com um baseado em CPU implementação ...*
>
> *Nossa implementação A2C síncrona tem um desempenho melhor do que nossas implementações assíncronas - não vimos nenhuma evidência de que o ruído introduzido pela assincronia forneça qualquer benefício de desempenho. Esta implementação A2C é mais econômica do que A3C ao usar máquinas de GPU única e é mais rápida do que uma implementação A3C somente CPU ao usar políticas maiores.*

De qualquer forma, implementaremos o **A2C** neste post, pois é mais simples de implementação. (Isso pode ser facilmente estendido para A3C)

Implementando A2C

Portanto, lembre-se da nova equação de atualização, substituindo o prêmio cumulativo com desconto de gradientes de política vanilla pela função Advantage:

$$\nabla_\theta J(\theta) \sim \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t)$$

Em cada etapa de aprendizagem, atualizamos o parâmetro Ator (com gradientes de política e valor de vantagem) e o parâmetro Crítico (com a minimização do erro quadrático médio com a equação de atualização de Bellman). Vamos ver como isso fica no código:

Abaixo estão os includes e hiperparâmetros:

```
import sys
import torch
import gym
import numpy as np
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.autograd import Variable
import matplotlib.pyplot as plt
import pandas as pd

# hyperparameters
hidden_size = 256
learning_rate = 3e-4

# Constants
GAMMA = 0.99
num_steps = 300
max_episodes = 3000
```

a3c_params.py hosted with ♡ by GitHub view raw

Primeiro, vamos começar com a implementação da Rede Crítica do Ator com as seguintes configurações:

```python
class ActorCritic(nn.Module):
    def __init__(self, num_inputs, num_actions, hidden_size, learning_rate=3e-4):
        super(ActorCritic, self).__init__()

        self.num_actions = num_actions
        self.critic_linear1 = nn.Linear(num_inputs, hidden_size)
        self.critic_linear2 = nn.Linear(hidden_size, 1)

        self.actor_linear1 = nn.Linear(num_inputs, hidden_size)
        self.actor_linear2 = nn.Linear(hidden_size, num_actions)

    def forward(self, state):
        state = Variable(torch.from_numpy(state).float().unsqueeze(0))
        value = F.relu(self.critic_linear1(state))
        value = self.critic_linear2(value)

        policy_dist = F.relu(self.actor_linear1(state))
        policy_dist = F.softmax(self.actor_linear2(policy_dist), dim=1)

        return value, policy_dist
```

a2c_model.py hosted with ♡ by GitHub view raw

O loop principal e o loop de atualização, conforme descrito acima:

```python
def a2c(env):
    num_inputs = env.observation_space.shape[0]
    num_outputs = env.action_space.n

    actor_critic = ActorCritic(num_inputs, num_outputs, hidden_size)
    ac_optimizer = optim.Adam(actor_critic.parameters(), lr=learning_rate)

    all_lengths = []
    average_lengths = []
    all_rewards = []
    entropy_term = 0

    for episode in range(max_episodes):
        log_probs = []
        values = []
        rewards = []

        state = env.reset()
        for steps in range(num_steps):
            value, policy_dist = actor_critic.forward(state)
            value = value.detach().numpy()[0,0]
            dist = policy_dist.detach().numpy()
```

```
23
24              action = np.random.choice(num_outputs, p=np.squeeze(dist))
25              log_prob = torch.log(policy_dist.squeeze(0)[action])
26              entropy = -np.sum(np.mean(dist) * np.log(dist))
27              new_state, reward, done, _ = env.step(action)
28
29              rewards.append(reward)
30              values.append(value)
31              log_probs.append(log_prob)
32              entropy_term += entropy
33              state = new_state
34
35              if done or steps == num_steps-1:
36                  Qval, _ = actor_critic.forward(new_state)
37                  Qval = Qval.detach().numpy()[0,0]
38                  all_rewards.append(np.sum(rewards))
39                  all_lengths.append(steps)
40                  average_lengths.append(np.mean(all_lengths[-10:]))
41                  if episode % 10 == 0:
42                      sys.stdout.write("episode: {}, reward: {}, total length: {}, average length
43                  break
44
45          # compute Q values
46          Qvals = np.zeros_like(values)
47          for t in reversed(range(len(rewards))):
48              Qval = rewards[t] + GAMMA * Qval
49              Qvals[t] = Qval
50
51          #update actor critic
52          values = torch.FloatTensor(values)
53          Qvals = torch.FloatTensor(Qvals)
54          log_probs = torch.stack(log_probs)
55
56          advantage = Qvals - values
57          actor_loss = (-log_probs * advantage).mean()
58          critic_loss = 0.5 * advantage.pow(2).mean()
59          ac_loss = actor_loss + critic_loss + 0.001 * entropy_term
60
61          ac_optimizer.zero_grad()
62          ac_loss.backward()
63          ac_optimizer.step()
64
65
66
67      # Plot results
68      smoothed_rewards = pd.Series.rolling(pd.Series(all_rewards), 10).mean()
69      smoothed_rewards = [elem for elem in smoothed_rewards]
70      plt.plot(all_rewards)
```

```
71      plt.plot(smoothend_rewards)
72      plt.plot()
73      plt.xlabel('Episode')
74      plt.ylabel('Reward')
75      plt.show()
76
77      plt.plot(all_lengths)
78      plt.plot(average_lengths)
79      plt.xlabel('Episode')
80      plt.ylabel('Episode length')
81      plt.show()
```

Executando o código, podemos ver como o desempenho melhora no ambiente OpenAI Gym CartPole-v0:

Azul: recompensas brutas; Laranja: recompensas suavizadas

Encontre a implementação completa aqui:

https://github.com/thechrisyoon08/Reinforcement-Learning /

Referências:

1. Slides de palestras da UC Berkeley CS294
2. Slides de aula da Carnegie Mellon University CS10703
3. Postagem de Lilian Weng sobre algoritmos de gradiente de política
4. A resposta de Jerry Liu no Quora Post “Por que o método de gradiente de política tem uma grande variação”
5. Vídeo da palestra Naver D2 RLCode
6. Postagem do blog OpenAI em A2C e ACKTR

7. Diagrama da postagem do blog da GroundAI "Figura 4: Representação esquemática do algoritmo do algoritmo Asynchronous Advantage Actor Critic (A3C)."

Outras postagens:

Confira minhas outras postagens sobre Aprendizado por Reforço:

- Derivando Gradientes de Política e Implementando REINFORCE
- Compreendendo os métodos críticos do ator
- Explicação de gradientes de política determinísticos profundos

Obrigado por ler!