Policy Optimization

Fabrício Barth

Insper Instituto de Ensino e Pesquisa

Abril de 2024

Objetivos desta aula

Ao final desta aula, você será capaz de:

- entender a diferença entre os algoritmos da família **policy-based** e os algoritmos da família **value-based** (Q-Learning, Sarsa, DQN), e;
- compreender como funciona e como implementar um algoritmo da família **policy-based**.

Policy Gradient ou Reinforce

O algoritmo REINFORCE é um algoritmo que ao invés de definir uma **policy** em termos de

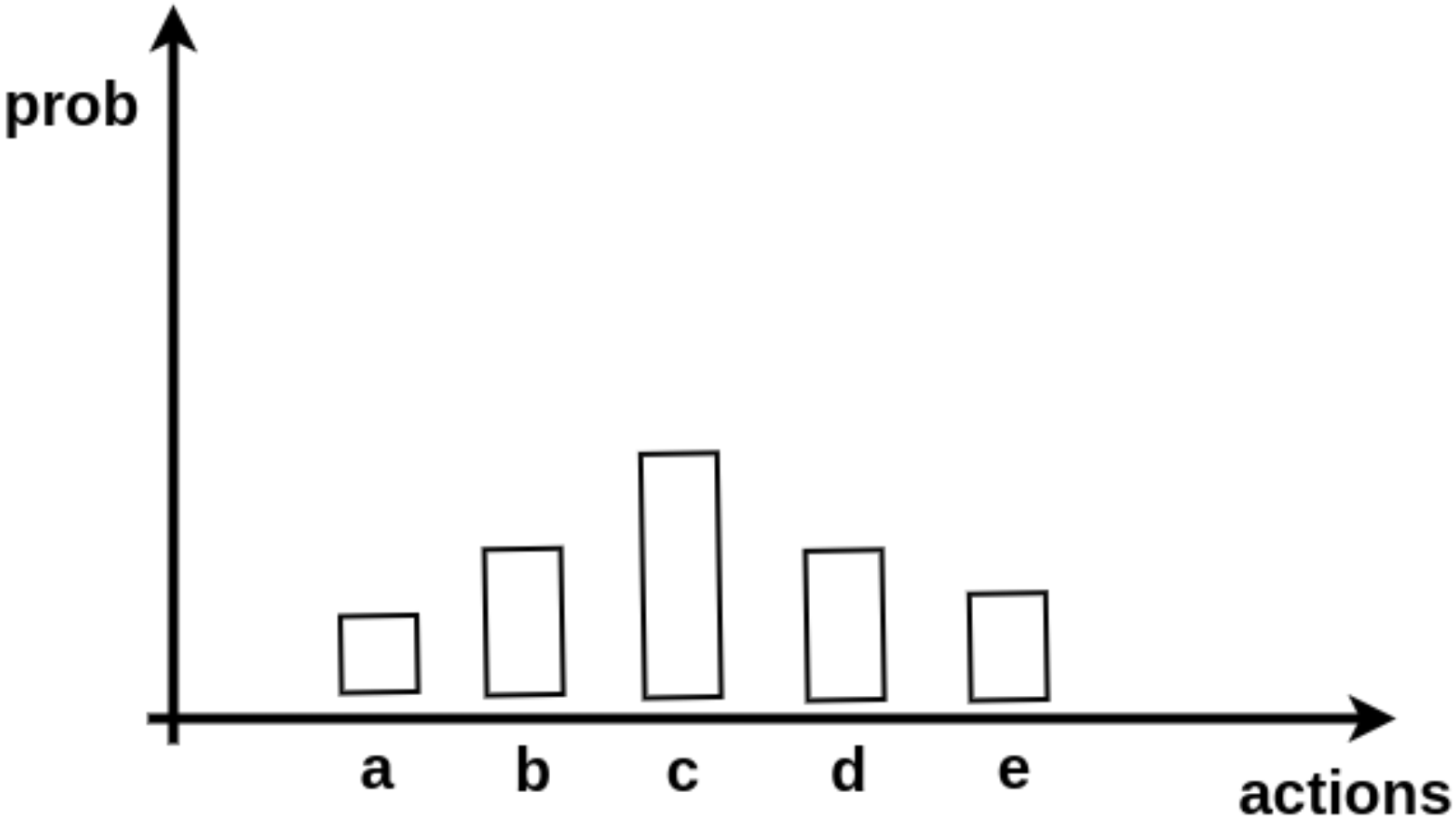
$$\pi(s) = \arg\max_a Q(s, a) \tag{1}$$

como é feito com o Q-Learning e DQN, ele define a **policy** no formato de uma distribuição:

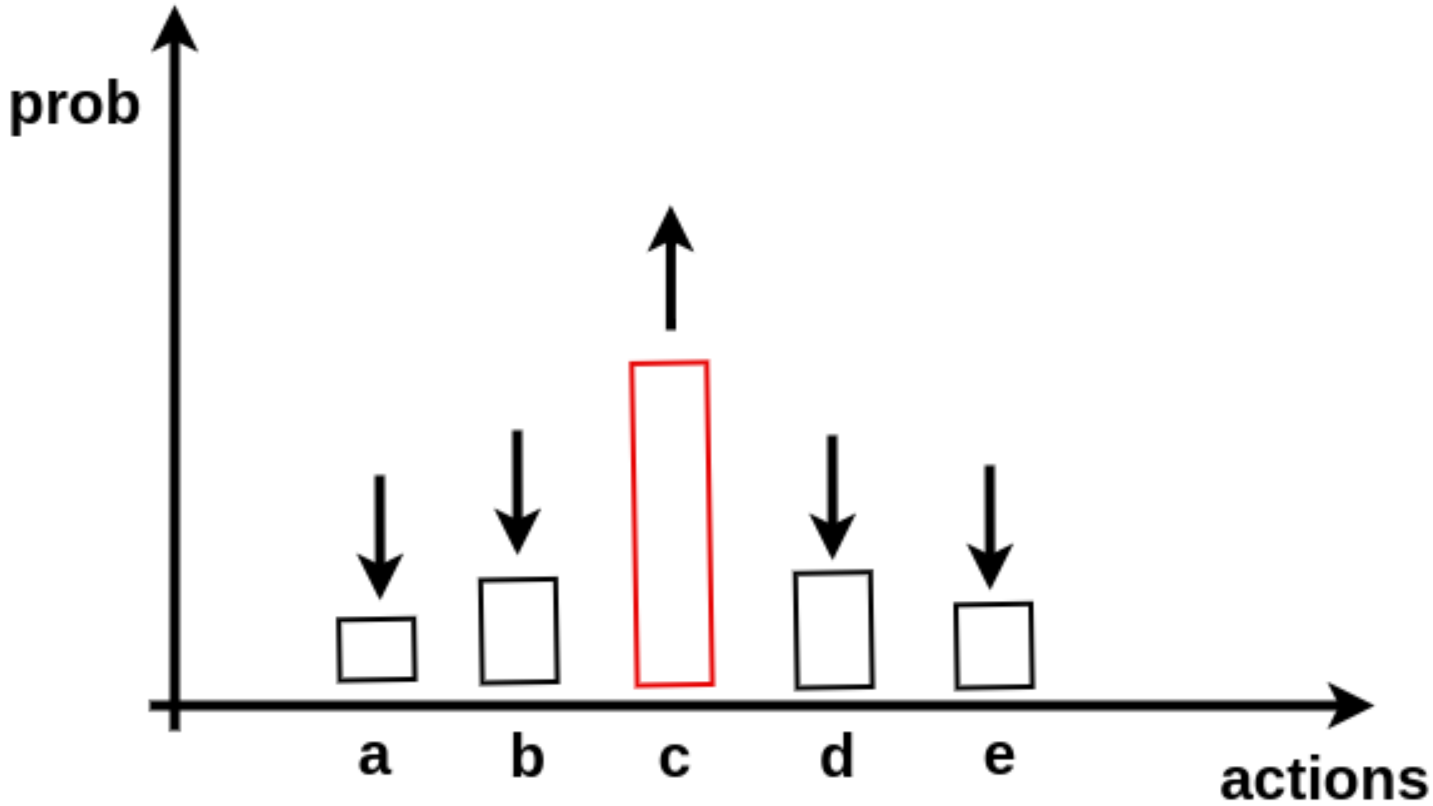
$$a_t \sim \pi_\theta(a_t|s_t) \tag{2}$$

onde θ representa os parâmetros da **policy** e a ideia é atualizar estes parâmetros usando um gradiente ascendente para **maximizar** a expectativa de **reward** futuro.

Imagine um agente que tem **5** ações possíveis:



Se durante a experiência do agente, o mesmo percebe que uma ação tem *reward* positivo (por exemplo, a ação c) então...



Algoritmo

function *REINFORCE(α, γ, episódios):*
inicializar os valores de θ para a policy $\pi(A|S,\theta)$ arbitrariamente
for todos os episódios **do**
  gerar um episódio $s_0, a_0, r_1, \cdots, s_{T-1}, a_{T-1}, r_T$ seguindo $\pi(.|.,\theta)$
  for cada passo dentro do episódio $t = 0, 1, 2, \cdots, T-1$ **do**
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} \times r_k$
    $\theta \leftarrow \theta + \alpha \times \bigtriangledown \log \pi(a_t|s_t,\theta) \times G$
  end for
end for

function *REINFORCE(α, γ, episódios):*

inicializar os valores de θ para a policy $\pi(A|S,\theta)$ arbitrariamente

for todos os episódios **do**

gerar um episódio $s_0, a_0, r_1, \cdots, s_{T-1}, a_{T-1}, r_T$ seguindo $\pi(.|.,\theta)$

for cada passo dentro do episódio $t = 0, 1, 2, \cdots, T-1$ **do**

$G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} \times r_k$

$\theta \leftarrow \theta + \alpha \times \bigtriangledown \log \pi(a_t|s_t,\theta) \times G$

end for

end for

- Inicializando uma rede neural

function *REINFORCE(α, γ, episódios):*
inicializar os valores de θ para a policy $\pi(A|S,\theta)$ arbitrariamente
for todos os episódios **do**
  gerar um episódio $s_0, a_0, r_1, \cdots, s_{T-1}, a_{T-1}, r_T$ seguindo $\pi(.|.,\theta)$
  for cada passo dentro do episódio $t = 0, 1, 2, \cdots, T-1$ **do**
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} \times r_k$
    $\theta \leftarrow \theta + \alpha \times \bigtriangledown \log \pi(a_t|s_t,\theta) \times G$
  end for
end for

- Executando um episódio por completo até encontrar um estado terminal (T).
- Armazena toda a sequência de $s_0, a_0, r_1, \cdots, s_{T-1}, a_{T-1}, r_T$

function *REINFORCE(α, γ, episódios):*
inicializar os valores de θ para a policy $\pi(A|S,\theta)$ arbitrariamente
for todos os episódios **do**
  gerar um episódio $s_0, a_0, r_1, \cdots, s_{T-1}, a_{T-1}, r_T$ seguindo $\pi(.|.,\theta)$
  for cada passo dentro do episódio $t = 0, 1, 2, \cdots, T-1$ **do**
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} \times r_k$
    $\theta \leftarrow \theta + \alpha \times \bigtriangledown \log \pi(a_t|s_t,\theta) \times G$
  end for
end for

- Considerando $T = 5$, temos:
  - Para a_0 em s_0, $G = \gamma^0 r_1 + \gamma^1 r_2 + \gamma^2 r_3 + \gamma^3 r_4$
  - Para a_1 em s_1, $G = \gamma^0 r_2 + \gamma^1 r_3 + \gamma^2 r_4$
- G é chamado de *discounted reward*.

function *REINFORCE(α, γ, episódios):*
inicializar os valores de θ para a policy $\pi(A|S,\theta)$ arbitrariamente
for todos os episódios **do**
  gerar um episódio $s_0, a_0, r_1, \cdots, s_{T-1}, a_{T-1}, r_T$ seguindo $\pi(.|.,\theta)$
  for cada passo dentro do episódio $t = 0, 1, 2, \cdots, T-1$ **do**
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} \times r_k$
    $\theta \leftarrow \theta + \alpha \times \bigtriangledown \log \pi(a_t|s_t,\theta) \times G$
  end for
end for

- $\pi(a_t|s_t,\theta)$ é a probabilidade de a_t ser escolhida em s_t seguindo $\pi(.|.,\theta)$
- a ideia principal desta equação é ajustar os pesos de θ usando um gradiente ascendente para **aumentar a probabilidade de escolher trajetórias onde G é alto**.
- α é a taxa de aprendizado.

Escolha de uma ação

```
import torch, numpy as np
probs = torch.tensor([0.1,0.9])
dist = torch.distributions.Categorical(probs)
action = dist.sample()
```

Avaliação de uma ação

```
import torch, numpy as np
probs = torch.tensor([0.1,0.9])
dist = torch.distributions.Categorical(probs)
action = dist.sample() # => 1

dist.log_prob(action)  # => -0.105
np.log(0.9)            # => -0.105
```

O log da probabilidade da ação 1 é o mesmo que o log da probabilidade de 0.9.

Definição da rede neural

```
env = gym.make('CartPole-v1')

nn = torch.nn.Sequential(
    torch.nn.Linear(4,64),
    torch.nn.ReLU(),
    torch.nn.Linear(64, env.action_space.n),
    torch.nn.Softmax(dim=-1)
)
optim = torch.optim.Adam(nn.parameters(), lr=lr)
```

- O número de nodos da primeira camada é 4 porque o **estado é representado por 4 valores**.
- Na última camada é utilizada uma função **Softmax** porque queremos a probabilidade de cada uma das ações acontecer. Ou seja, o valor de todos os nodos de saída precisa ser igual a 1.

Gerando os episódios

```
(state, _) = env.reset()
obs = torch.tensor(state, dtype=torch.float)
done = False
Actions, States, Rewards = [], [], []

while not done:
    probs = nn(obs)
    dist = torch.distributions.Categorical(probs)
    action = dist.sample().item()
    obs_, rew, done, truncated, _ = env.step(action)

    Actions.append(torch.tensor(action, dtype=torch.int))
    States.append(obs)
    Rewards.append(rew)

    obs = torch.tensor(obs_, dtype=torch.float)
```

Calculando G

```
DiscountedReturns = []
for t in range(len(Rewards)):
    G = 0.0
    for k,r in enumerate(Rewards[t:]):
        G += (y**k)*r
    DiscountedReturns.append(G)
```

Ao final deste loop, **DiscountedReturns** vai ter um valor de *reward acumulado* para cada par (a_i, s_j) da trajetória do agente.

Atualizando θ

```
for State, Action, G in zip(States, Actions, DiscountedReturns):
    probs = nn(State)
    dist = torch.distributions.Categorical(probs=probs)
    log_prob = dist.log_prob(Action)

    # importante: aqui deve ser negativo pq eh um gradient ascent
    loss = -log_prob*G

    optim.zero_grad()
    loss.backward()
    optim.step()
```

Treinando e salvando o modelo

```
env = gym.make('CartPole-v1')
lr = 0.0001
y = 0.999
nn, statistics = train(env, y, lr, 1200)
torch.save(nn, 'data/nn.pt')
cl = ['episode','actions','rewards']
df = pd.DataFrame(statistics, columns = cl)
df.to_csv('results/statistics_cartpole.csv')
```

Usando o modelo

```
env = gym.make('CartPole-v1', render_mode='human')
(state, _) = env.reset()
obs = torch.tensor(state, dtype=torch.float)
done = False

while not done:
    probs = nn(obs)
    dist = torch.distributions.Categorical(probs)
    action = dist.sample().item()
    obs_, rew, done, _, _ = env.step(action)
    obs = torch.tensor(obs_, dtype=torch.float)
```

Lendo e usando o modelo

```
torch.load('data/nn.pt')
env = gym.make('CartPole-v1', render_mode='human')
(state, _) = env.reset()
obs = torch.tensor(state, dtype=torch.float)
done = False

while not done:
    probs = nn(obs)
    dist = torch.distributions.Categorical(probs)
    ac = dist.sample().item()
    obs_, rew, done, truncated, _info = env.step(ac)
    obs = torch.tensor(obs_, dtype=torch.float)
```

Sugestão de implementação

Implemente uma versão do algoritmo Reinforce com base no pseudo-código deste material e com base nos trechos de códigos disponibilizados.

Ambientes

Teste nos ambientes CartPole-v1 e LunarLander-v2.

Referências

- Williams, R.J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine Learning 8, 229–256 (1992). https://doi.org/10.1007/BF00992696
- REINFORCE: Reinforcement Learning Most Fundamental Algorithm. Vídeo produzido por Andriy Drozdyuk. Disponível em https://www.youtube.com/watch?v=5eSh5F8gjWU