

# Policy Optimization

Fabrício Barth

Inspere Instituto de Ensino e Pesquisa

Maio de 2023

# Objetivos desta aula

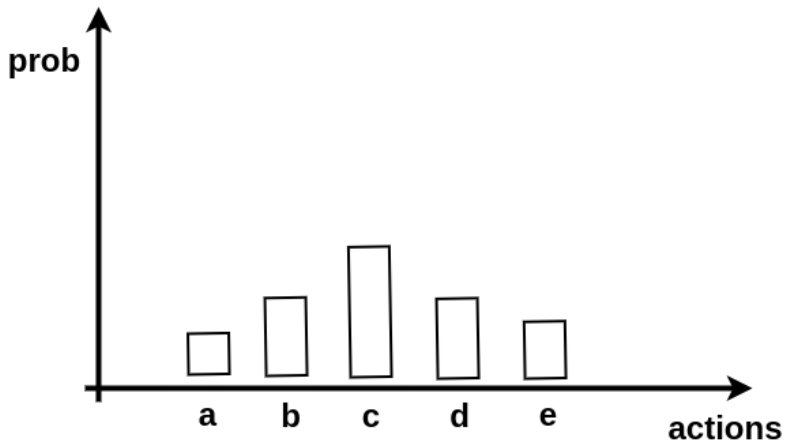
Ao final desta aula, você será capaz de:

- entender a diferença entre os algoritmos da família **policy optimization** e os algoritmos da família **Q-learning**, e;
- compreender como funciona e como implementar os algoritmos **policy gradient** e **proximal policy optimization**.

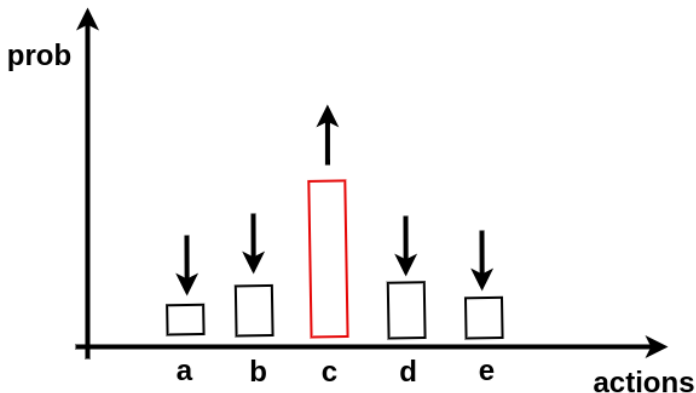
O algoritmo REINFORCE é um algoritmo que ao invés de definir uma **policy** em termos de  $\pi(s) = \arg \max_a Q(s, a)$ , como é feito com o Q-Learning e Deep Q-Learning, ele define a **policy** no formato de uma distribuição:

$$a_t \sim \pi_{\theta}(a_t | s_t) \quad (1)$$

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



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



**function** *REINFORCE*( $\alpha, \gamma, \text{episódios}$ ):

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

**for** todos os episódios **do**

gerar um episódio  $s_0, a_0, r_0, \dots, s_{T-1}, a_{T-1}, r_T$  seguindo  $\pi(.|., \theta)$

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

$$G \leftarrow \sum_{t=0}^T \gamma^t \times r_t$$

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

**end for**

**end for**

**function** *REINFORCE*( $\alpha, \gamma, \text{episódios}$ ):

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

**for** todos os episódios **do**

gerar um episódio  $s_0, a_0, r_0, \dots, s_{T-1}, a_{T-1}, r_T$  seguindo  $\pi(.|., \theta)$

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

$$G \leftarrow \sum_{t=0}^T \gamma^t \times r_t$$

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

**end for**

**end for**

- Inicializando uma rede neural

**function** *REINFORCE*( $\alpha, \gamma, \text{episódios}$ ):

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

**for** todos os episódios **do**

gerar um episódio  $s_0, a_0, r_0, \dots, s_{T-1}, a_{T-1}, r_T$  seguindo  $\pi(.|., \theta)$

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

$$G \leftarrow \sum_{t=0}^T \gamma^t \times r_t$$

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

**end for**

**end for**

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



**function** *REINFORCE*( $\alpha, \gamma, \text{episódios}$ ):

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

**for** todos os episódios **do**

gerar um episódio  $s_0, a_0, r_0, \dots, s_{T-1}, a_{T-1}, r_T$  seguindo  $\pi(\cdot|., \theta)$

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

$$G \leftarrow \sum_{t=0}^T \gamma^t \times r_t$$

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

**end for**

**end for**

- Se  $T = 3$  então  $G = \gamma^0 r_0 + \gamma^1 r_1 + \gamma^2 r_2 + \gamma^3 r_3$
- Muitos dos artigos sobre PO descrevem  $G$  como *discounted reward*.
- Também conhecido como *reforço acumulado* em Q-Learning.

**function** *REINFORCE*( $\alpha, \gamma, \text{episódios}$ ):

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

**for** todos os episódios **do**

gerar um episódio  $s_0, a_0, r_0, \dots, s_{T-1}, a_{T-1}, r_T$  seguindo  $\pi(\cdot|., \theta)$

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

$$G \leftarrow \sum_{t=0}^T \gamma^t \times r_t$$

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

**end for**

**end for**

- $\pi(a_t|s_t, \theta)$  é a probabilidade de  $a_t$  ser escolhida em  $s_t$  seguindo  $\pi(\cdot|., \theta)$
- a ideia principal desta equação é ajustar os pesos de  $\theta$  usando um gradiente ascendente para aumentar a probabilidade de escolher trajetórias onde  $G$  é alto.
- $\alpha$  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() #  $\Rightarrow 1$ 

dist.log_prob(action) #  $\Rightarrow -0.105$ 
np.log(0.9)           #  $\Rightarrow -0.105$ 
```

# Definição da rede neural

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

nn = torch.nn.Sequential(
    torch.nn.Linear(8,128),
    torch.nn.ReLU(),
    torch.nn.Linear(128, env.action_space.n),
    torch.nn.Softmax(dim=-1)
)

# usa o Adam algorithm para otimizacao
optim = torch.optim.Adam(nn.parameters(), lr=lr)
```

# 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 += (gamma**k)*r  
    DiscountedReturns.append(G)
```

# Atualizando $\theta$

```
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()
```



# 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)
```

# Treinando e salvando o modelo

```
env = gym.make('CartPole-v1')  
lr = 0.0001  
gamma = 0.999  
nn, statistics = train(env, gamma, 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')
```

# 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.

- Williams, R.J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8, 229–256 (1992). <https://doi.org/10.1007/BF00992696>