

# CSC321 Lecture 21: Policy Gradient

Roger Grosse

# Overview

- Most of this course was about supervised learning, plus a little unsupervised learning.
- Final 3 lectures: reinforcement learning
  - Middle ground between supervised and unsupervised learning
  - An agent acts in an environment and receives a reward signal.
- Today: policy gradient (directly do SGD over a stochastic policy using trial-and-error)
- Next lecture: Q-learning (learn a value function predicting returns from a state)
- Final lecture: policies and value functions are way more powerful in combination

# Reinforcement learning



- An **agent** interacts with an **environment** (e.g. game of Breakout)
- In each time step  $t$ ,
  - the agent receives **observations** (e.g. pixels) which give it information about the **state**  $s_t$  (e.g. positions of the ball and paddle)
  - the agent picks an **action**  $a_t$  (e.g. keystrokes) which affects the state
- The agent periodically receives a **reward**  $r(s_t, a_t)$ , which depends on the state and action (e.g. points)
- The agent wants to learn a **policy**  $\pi_{\theta}(a_t | s_t)$ 
  - **Distribution over actions depending on the current state and parameters  $\theta$**

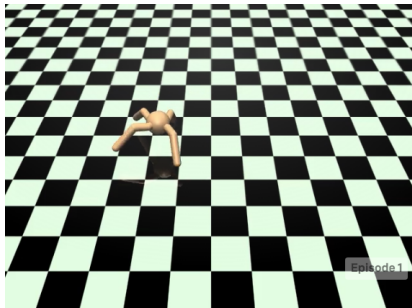
# Markov Decision Processes

- The environment is represented as a **Markov decision process**  $\mathcal{M}$ .
- Markov assumption: all relevant information is encapsulated in the current state; i.e. the policy, reward, and transitions are all independent of past states given the current state
- Components of an MDP:
  - initial state distribution  $p(\mathbf{s}_0)$
  - **policy**  $\pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t)$
  - transition distribution  $p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)$
  - reward function  $r(\mathbf{s}_t, \mathbf{a}_t)$
- Assume a **fully observable** environment, i.e.  $\mathbf{s}_t$  can be observed directly
- **Rollout, or trajectory**  $\tau = (\mathbf{s}_0, \mathbf{a}_0, \mathbf{s}_1, \mathbf{a}_1, \dots, \mathbf{s}_T, \mathbf{a}_T)$
- Probability of a rollout

$$p(\tau) = p(\mathbf{s}_0) \pi_{\theta}(\mathbf{a}_0 | \mathbf{s}_0) p(\mathbf{s}_1 | \mathbf{s}_0, \mathbf{a}_0) \cdots p(\mathbf{s}_T | \mathbf{s}_{T-1}, \mathbf{a}_{T-1}) \pi_{\theta}(\mathbf{a}_T | \mathbf{s}_T)$$

# Markov Decision Processes

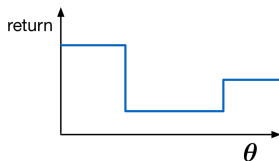
Continuous control in simulation, e.g. teaching an ant to walk



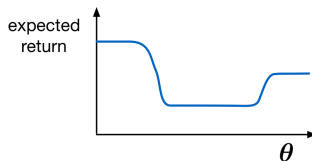
- State: positions, angles, and velocities of the joints
- Actions: apply forces to the joints
- Reward: distance from starting point
- Policy: output of an ordinary MLP, using the state as input
- More environments: <https://gym.openai.com/envs/#mujoco>

# Markov Decision Processes

- **Return** for a rollout:  $r(\tau) = \sum_{t=0}^T r(\mathbf{s}_t, \mathbf{a}_t)$ 
  - Note: we're considering a finite **horizon**  $T$ , or number of time steps; we'll consider the infinite horizon case later.
- Goal: maximize the expected return,  $R = \mathbb{E}_{p(\tau)}[r(\tau)]$
- The expectation is over both the environment's dynamics and the policy, but we only have control over the policy.
- The stochastic policy is important, since it makes  $R$  a continuous function of the policy parameters.
  - Reward functions are often discontinuous, as are the dynamics (e.g. collisions)



deterministic policies



stochastic policies

# REINFORCE

- **REINFORCE** is an elegant algorithm for maximizing the expected return  $R = \mathbb{E}_{p(\tau)} [r(\tau)]$ .
- Intuition: trial and error
  - Sample a rollout  $\tau$ . If you get a high reward, try to make it more likely. If you get a low reward, try to make it less likely.
- Interestingly, this can be seen as stochastic gradient ascent on  $R$ .

# REINFORCE

- Recall the derivative formula for log:

$$\frac{\partial}{\partial \theta} \log p(\tau) = \frac{\frac{\partial}{\partial \theta} p(\tau)}{p(\tau)} \implies \frac{\partial}{\partial \theta} p(\tau) = p(\tau) \frac{\partial}{\partial \theta} \log p(\tau)$$

- Gradient of the expected return:

$$\begin{aligned} \frac{\partial}{\partial \theta} \mathbb{E}_{p(\tau)} [r(\tau)] &= \frac{\partial}{\partial \theta} \sum_{\tau} r(\tau) p(\tau) \\ &= \sum_{\tau} r(\tau) \frac{\partial}{\partial \theta} p(\tau) \\ &= \sum_{\tau} r(\tau) p(\tau) \frac{\partial}{\partial \theta} \log p(\tau) \\ &= \mathbb{E}_{p(\tau)} \left[ r(\tau) \frac{\partial}{\partial \theta} \log p(\tau) \right] \end{aligned}$$

- Compute stochastic estimates of this expectation by sampling rollouts.



# REINFORCE

- For reference:

$$\frac{\partial}{\partial \theta} \mathbb{E}_{p(\tau)} [r(\tau)] = \mathbb{E}_{p(\tau)} \left[ r(\tau) \frac{\partial}{\partial \theta} \log p(\tau) \right]$$

- If you get a large reward, make the rollout more likely. If you get a small reward, make it less likely.
- Unpacking the REINFORCE gradient:

$$\begin{aligned} \frac{\partial}{\partial \theta} \log p(\tau) &= \frac{\partial}{\partial \theta} \log \left[ p(\mathbf{s}_0) \prod_{t=0}^T \pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t) \prod_{t=1}^T p(\mathbf{s}_t | \mathbf{s}_{t-1}, \mathbf{a}_{t-1}) \right] \\ &= \frac{\partial}{\partial \theta} \log \prod_{t=0}^T \pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t) \\ &= \sum_{t=0}^T \frac{\partial}{\partial \theta} \log \pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t) \end{aligned}$$

- Hence, it tries to make *all* the actions more likely or less likely, depending on the reward. I.e., it doesn't do **credit assignment**.
  - This is a topic for next lecture.

# REINFORCE

Repeat forever:

Sample a rollout  $\tau = (\mathbf{s}_0, \mathbf{a}_0, \mathbf{s}_1, \mathbf{a}_1, \dots, \mathbf{s}_T, \mathbf{a}_T)$

$$r(\tau) \leftarrow \sum_{k=0}^T r(\mathbf{s}_k, \mathbf{a}_k)$$

For  $t = 0, \dots, T$ :

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha r(\tau) \frac{\partial}{\partial \boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\mathbf{a}_k | \mathbf{s}_k)$$

- Observation: actions should only be reinforced based on future rewards, since they can't possibly influence past rewards.
- You can show that this still gives unbiased gradient estimates.

Repeat forever:

Sample a rollout  $\tau = (\mathbf{s}_0, \mathbf{a}_0, \mathbf{s}_1, \mathbf{a}_1, \dots, \mathbf{s}_T, \mathbf{a}_T)$

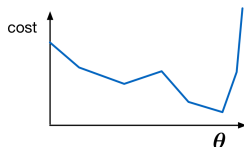
For  $t = 0, \dots, T$ :

$$r_t(\tau) \leftarrow \sum_{k=t}^T r(\mathbf{s}_k, \mathbf{a}_k)$$

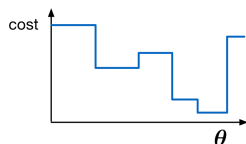
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha r_t(\tau) \frac{\partial}{\partial \boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\mathbf{a}_k | \mathbf{s}_k)$$

# Optimizing Discontinuous Objectives

- Edge case of RL: handwritten digit classification, but maximizing accuracy (or minimizing 0–1 loss)
- Gradient descent completely fails if the cost function is discontinuous:



Non-differentiable: OK



Discontinuous: not OK

- Original solution: use a surrogate loss function, e.g. logistic-cross-entropy
- RL formulation: in each episode, the agent is shown an image, guesses a digit class, and receives a reward of 1 if it's right or 0 if it's wrong
- We'd never actually do it this way, but it will give us an interesting comparison with backprop

# Optimizing Discontinuous Objectives

- RL formulation
  - one time step
  - state  $\mathbf{x}$ : an image
  - action  $\mathbf{a}$ : a digit class
  - reward  $r(\mathbf{x}, \mathbf{a})$ : 1 if correct, 0 if wrong
  - policy  $\pi(\mathbf{a} | \mathbf{x})$ : a distribution over categories
    - Compute using an MLP with softmax outputs – this is a **policy network**

# Optimizing Discontinuous Objectives

- Let  $z_k$  denote the logits,  $y_k$  denote the softmax output,  $t$  the integer target, and  $t_k$  the target one-hot representation.
- To apply REINFORCE, we sample  $\mathbf{a} \sim \pi_{\theta}(\cdot | \mathbf{x})$  and apply:

$$\begin{aligned}\theta &\leftarrow \theta + \alpha r(\mathbf{a}, \mathbf{t}) \frac{\partial}{\partial \theta} \log \pi_{\theta}(\mathbf{a} | \mathbf{x}) \\ &= \theta + \alpha r(\mathbf{a}, \mathbf{t}) \frac{\partial}{\partial \theta} \log y_a \\ &= \theta + \alpha r(\mathbf{a}, \mathbf{t}) \sum_k (a_k - y_k) \frac{\partial}{\partial \theta} z_k\end{aligned}$$

- Compare with the logistic regression SGD update:

$$\begin{aligned}\theta &\leftarrow \theta + \alpha \frac{\partial}{\partial \theta} \log y_t \\ &\leftarrow \theta + \alpha \sum_k (t_k - y_k) \frac{\partial}{\partial \theta} z_k\end{aligned}$$

# Reward Baselines

- For reference:

$$\theta \leftarrow \theta + \alpha r(\mathbf{a}, \mathbf{t}) \frac{\partial}{\partial \theta} \log \pi_{\theta}(\mathbf{a} | \mathbf{x})$$

- Clearly, we can add a constant offset to the reward, and we get an equivalent optimization problem.
- Behavior if  $r = 0$  for wrong answers and  $r = 1$  for correct answers
  - wrong: do nothing
  - correct: make the action more likely
- If  $r = 10$  for wrong answers and  $r = 11$  for correct answers
  - wrong: make the action more likely
  - correct: make the action more likely (slightly stronger)
- If  $r = -10$  for wrong answers and  $r = -9$  for correct answers
  - wrong: make the action less likely
  - correct: make the action less likely (slightly weaker)

# Reward Baselines

- Problem: the REINFORCE update depends on arbitrary constant factors added to the reward.
- Observation: we can subtract a **baseline**  $b$  from the reward without biasing the gradient.

$$\begin{aligned}\mathbb{E}_{p(\tau)} \left[ (r(\tau) - b) \frac{\partial}{\partial \theta} \log p(\tau) \right] &= \mathbb{E}_{p(\tau)} \left[ r(\tau) \frac{\partial}{\partial \theta} \log p(\tau) \right] - b \mathbb{E}_{p(\tau)} \left[ \frac{\partial}{\partial \theta} \log p(\tau) \right] \\&= \mathbb{E}_{p(\tau)} \left[ r(\tau) \frac{\partial}{\partial \theta} \log p(\tau) \right] - b \sum_{\tau} p(\tau) \frac{\partial}{\partial \theta} \log p(\tau) \\&= \mathbb{E}_{p(\tau)} \left[ r(\tau) \frac{\partial}{\partial \theta} \log p(\tau) \right] - b \sum_{\tau} \frac{\partial}{\partial \theta} p(\tau) \\&= \mathbb{E}_{p(\tau)} \left[ r(\tau) \frac{\partial}{\partial \theta} \log p(\tau) \right] - 0\end{aligned}$$

- We'd like to pick a baseline such that good rewards are positive and bad ones are negative.
- $\mathbb{E}[r(\tau)]$  is a good choice of baseline, but we can't always compute it easily. There's lots of research on trying to approximate it.

# More Tricks

- We left out some more tricks that can make policy gradients work a lot better.
  - Evaluate each action using only future rewards, since it has no influence on past rewards. It can be shown this gives unbiased gradients.
  - Natural policy gradient corrects for the geometry of the space of policies, preventing the policy from changing too quickly.
  - Rather than use the actual return, evaluate actions based on estimates of future returns. This is a class of methods known as actor-critic, which we'll touch upon next lecture.
- Trust region policy optimization (TRPO) and proximal policy optimization (PPO) are modern policy gradient algorithms which are very effective for continuous control problems.



# Discussion

- What's so great about backprop and gradient descent?
  - Backprop does credit assignment – it tells you exactly which activations and parameters should be adjusted upwards or downwards to decrease the loss on some training example.
  - REINFORCE doesn't do credit assignment. If a rollout happens to be good, all the actions get reinforced, even if some of them were bad.
  - Reinforcing all the actions as a group leads to random walk behavior.

# Discussion

- Why policy gradient?
  - Can handle discontinuous cost functions
  - Don't need an explicit model of the environment, i.e. rewards and dynamics are treated as black boxes
    - Policy gradient is an example of **model-free reinforcement learning**, since the agent doesn't try to fit a model of the environment
    - Almost everyone thinks model-based approaches are needed for AI, but nobody has a clue how to get it to work

# Evolution Strategies (optional)

- REINFORCE can handle discontinuous dynamics and reward functions, but it requires a differentiable network since it computes  $\frac{\partial}{\partial \theta} \log \pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t)$
- Evolution strategies (ES) take the policy gradient idea a step further, and avoid backprop entirely.
- ES can use deterministic policies. It randomizes over the choice of policy rather than over the choice of actions.
  - I.e., sample a random policy from a distribution  $p_{\eta}(\theta)$  parameterized by  $\eta$  and apply the policy gradient trick

$$\frac{\partial}{\partial \eta} \mathbb{E}_{\theta \sim p_{\eta}} [r(\tau(\theta))] = \mathbb{E}_{\theta \sim p_{\eta}} \left[ r(\tau(\theta)) \frac{\partial}{\partial \eta} \log p_{\eta}(\theta) \right]$$

- The neural net architecture itself can be discontinuous.

# Evolution Strategies (optional)

---

## Algorithm 1 Evolution Strategies

---

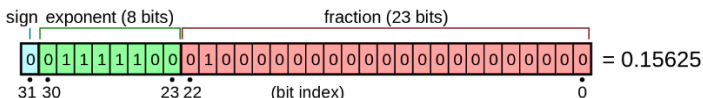
```
1: Input: Learning rate  $\alpha$ , noise standard deviation  $\sigma$ , initial policy parameters  $\theta_0$ 
2: for  $t = 0, 1, 2, \dots$  do
3:   Sample  $\epsilon_1, \dots, \epsilon_n \sim \mathcal{N}(0, I)$ 
4:   Compute returns  $F_i = F(\theta_t + \sigma \epsilon_i)$  for  $i = 1, \dots, n$ 
5:   Set  $\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^n F_i \epsilon_i$ 
6: end for
```

---

<https://arxiv.org/pdf/1703.03864.pdf>

# Evolution Strategies (optional)

- The IEEE floating point standard is nonlinear, since small enough numbers get truncated to zero.



- This acts as a discontinuous activation function, which ES is able to handle.
- ES was able to train a good MNIST classifier using a “linear” activation function.
- <https://blog.openai.com/nonlinear-computation-in-linear->

