Reinforcement Learning

internal state

reward

evironment

learning rate $\alpha$
inverse temperature $\beta$
discount rate $\gamma$

observation

Some slides from:
David Silver, Radford Neal

CSC411: Machine Learning and Data Mining, Winter 2017
Michael Guerzhoy
Reinforcement Learning

• **Supervised learning:**
  • The training set consists of inputs and outputs. We try to build a function that predicts the outputs from the inputs. The cost function is a *supervision signal* that tells us how well we are doing.

• **Unsupervised Learning**
  • The training set consists of data (just the inputs). We try to build a function that models the inputs. There is no supervision signal.

• **Reinforcement Learning**
  • The *agent* performs *actions* that change the *state* and receives *rewards* that depend on the state.
  • Trade-off between exploitation (go to states you already discovered give you high reward) and exploration (try going to states that give even higher rewards).
Reinforcement Learning

• The world is going through a sequence of states $s_1, s_2, s_3, \ldots, s_n$ and times $t_1, t_2, \ldots, t_n$

• At each time $t_i$, the agent performs action $a_i$, moves to state $s_{i+1}$ (depending on the action taken) and receives reward $r_i$ (the reward could be 0)

• Goal: maximize the total reward over time
  • Total reward: $r_1 + r_2 + \cdots + r_n$
  • Total reward with discounting, so that rewards for away in the future count for less: $r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots + \gamma^{n-1} r_n$
    • Getting a reward now is better than getting the same reward later on
Reinforcement Learning: Go

AlphaGo defeats Lee Sedol (2016)
Reinforcement Learning: Go

- State: the position on the board
- Reward: 0 if the game hasn’t ended, 1 if the agent wins, -1 if the opponent wins
- Action: make a legal Go move (place a stone on a free point)
- Goal: make a function that, given the state (position on the board), finds an optimal move
  - Note: we could have intermediate goals as well, like learning a function that evaluates every state

- Exploitation vs. Exploration
  - Make moves the function already thinks will lead to a good outcome vs
  - Try making novel moves and see if you don’t discover a way to adjust the function to get even better outcomes
Reinforcement Learning: Walking

https://gym.openai.com/envs/Walker2d-v1
Reinforcement Learning: Walking

• State: the positions of all the joints
• Reward: if we haven’t walked to the destination yet, 0. If we reached the destination, 1
• Action: move a joint in a particular direction
• Goal: learn a function that applies a particular force to a particular joint at every time-step $t$ so that the walker reaches the destination
A policy function \( \pi \) takes in the current state \( s \), and outputs the move the agent should take.

- Deterministic policy: \( a = \pi(s) \)
- Stochastic policy: \( \pi(a|s) = P(A_t = a|S_t = s) \)
  - Must have for things like playing poker
  - But also good for exploration in general!

Just like for other functions we approximate, we can parametrize \( \pi \) using a parameter vector \( \theta \)

- Initialize \( \theta \) randomly
- Follow the policy \( \pi_\theta \), and adjust \( \theta \) based on the rewards we receive
Softmax Policy (discrete actions)

• Compute features $\phi(a, s)$ for each action-state tuple
  • Some kind of representation that makes sense
  • Could be something very complicated
    • E.g. something computed using a deep neural network
      (similar in spirit to what we did in Project 2 or word2vec)
  • In general, we can think of the features as the last layer of the neural network, before it’s passed into the softmax
  
• $\pi_\theta(s, a) \propto \exp(\phi(s, a)^T \theta)$
Gaussian Policy (continuous actions)

• For continuous actions, it makes sense to use a Gaussian distribution for the actions, centred around $\phi(s)^T \theta$
• $a \sim N(\phi(s)^T \theta, \sigma^2)$
How good is policy $\pi_\theta$?

- $d^{\pi_\theta}(s)$ is the probability of the agent being in state $s$ at time-step $t$ if we follow policy $\pi_\theta$
  - Not easily computed at all!
  - But we can simply follow policy $\pi_\theta$ for a long time and record how often we find ourselves in each state
  - For continuous states, do some approximation of that

- $J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s)V^{\pi_\theta}(s)$
  - $V^{\pi_\theta}(s)$ is the (expected) total reward if we start from state $s$
    - Start from state $s$ at time 0
    - Follow policy $\pi_\theta$, and compute $r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots$
  - We want states that lead to high rewards to be high probability
  - We want to take actions that lead to high rewards

- Larger $J_{avV}(\theta)$ means better $\theta$
Policy Gradient

• $J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s)V^{\pi_\theta}(s)$
  
  $$= \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(a|s)q^{\pi_\theta}(a|s)$$

• $\nabla J = \begin{pmatrix} \partial J/\partial \theta_1 \\ \vdots \\ \partial J/\partial \theta_n \end{pmatrix}$

• Idea: $\theta \leftarrow \theta + \alpha \nabla J(\theta)$
Policy Gradient: Finite Differences

- For each $k$ in $1..n$
  \[
  \frac{\partial J(\theta)}{\partial \theta_k} \approx \frac{J(\theta+u_k) - J(\theta)}{\epsilon} \quad (u_k \text{ is all 0's except the } k\text{-th coordinate is } \epsilon)
  \]

- Approximate $J(\theta)$ by following policy $\pi_\theta$ for a while and keeping track of the rewards you get!

- Has actually been used to make physical robots that walk
  - The policy function had about 12 parameters
  - Vary each parameter in turn, have the robot run, measure how fast it walked, and compute the gradient based on that
Policy Gradient Theorem

• \( J_{\text{av}}(\theta) = \sum_s d^{\pi_\theta}(s)V^{\pi_\theta}(s) \), so

• \( J_{\text{av}}(\theta) = \sum_s d^{\pi_\theta}(s)\sum_a \pi_\theta(a|s)q^{\pi_\theta}(a|s) \)
  • \( \pi_\theta(a|s) \) is the probability of taking action a starting from state s, following policy \( \pi_\theta(a|s) \)
  • \( q^{\pi_\theta}(a|s) \) is the total expected reward for performing action a in state s, and then following \( \pi_\theta \)

• \( \nabla_\theta J_{\text{av}}(\theta) = \sum_s d^{\pi_\theta}(s)\sum_a q^{\pi_\theta}(a|s)\nabla_\theta \pi_\theta(a|s) \)
  • \( q^{\pi_\theta}(a|s) \) is the total expected reward for performing action a in state s, and then following \( \pi_\theta \)
  • Not obvious! We are differentiating an expression involving both \( d^{\pi_\theta} \) and \( V^{\pi_\theta} \)
Policy Gradient Theorem

\[ \nabla_\theta J_{\text{avV}}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a q^{\pi_\theta}(a|s) \nabla_\theta \pi_\theta(a|s) \]

- Weighted sum over \( \sum_a q^{\pi_\theta}(s, a) \nabla_\theta \pi_\theta(a|s) \)
- If it looks like we should take action \( a \) in state \( s \) (since \( q^{\pi_\theta}(s, a) \) is high), care more about \( \nabla_\theta \pi_\theta(a|s) \), which tells us how to change \( \theta \) to make it more likely that we take action \( a \) in state \( s \)
- Take the weighted average over the gradients for all states, weighing the states that we are more likely to visit more
Policy Gradient: Gaussian Policy

• $a \sim N(\phi(s)^T \theta, \sigma^2)$

• $\nabla_\theta \log \pi_\theta(a|s) = \nabla_\theta \log \exp \left(-\frac{(a-\phi(s)^T \theta)^2}{2\sigma^2}\right) = \nabla_\theta \left(-\frac{(a-\phi(s)^T \theta)^2}{2\sigma^2}\right) = \frac{(a - \phi(s)^T \theta)\phi(s)}{\sigma^2}$

• (How to make it more like that we take action $a$ in state $s$?)

• (Aside: $\nabla \exp(f) = \exp(f) \nabla f$, $\nabla \log(f) = (\nabla f)/f$)
Expectation trick

• At time t, starting from state $S_t$:

• $\nabla_{\theta} J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a q^{\pi_\theta}(a|s) \nabla_\theta \pi_\theta(a|s) = E_{\pi_\theta} [\gamma^t \sum_a q^{\pi_\theta}(a|S_t) \nabla_\theta \pi_\theta(a|S_t)]$

• (Just follow policy $\pi_\theta$, and in the long term, will encounter states in proportions $d^{\pi_\theta}$)
Expectation trick, again

\[ \nabla_{\theta} J_{\text{avg}}(\theta) = E_{\pi_\theta} \left[ \gamma^t \sum_a q^{\pi_\theta}(a|S_t) \nabla_{\theta} \pi_\theta (a|S_t) \right] 
\]

\[ = E_{\pi_\theta} \left[ \gamma^t \sum_a \pi_\theta (a|S_t) q^{\pi_\theta}(a|S_t) \frac{\nabla_{\theta} \pi_\theta (a|S_t)}{\pi_\theta (a|S_t)} \right] \]

• Multiply and divide again by \( \pi_\theta (a|S_t) \)

• Now, replace the sum over actions \( a \) by a single action \( A_t \) that we actually take – can do that inside an expectation!

\[ = E_{\pi_\theta} \left[ \gamma^t q^{\pi_\theta}(A_t|S_t) \frac{\nabla_{\theta} \pi_\theta (A_t|S_t)}{\pi_\theta (A_t|S_t)} \right] \]
Expectation trick, again

\[ \nabla_{\theta} J_{\text{avV}}(\theta) = E_{\pi_{\theta}} \left[ \gamma^t q^{\pi_{\theta}}(A_t|S_t) \frac{\nabla_{\theta} \pi_{\theta}(A_t|S_t)}{\pi_{\theta}(A_t|S_t)} \right] \]

• Now, replace \( q^{\pi_{\theta}}(A_t|S_t) \) by the actual total reward we get by following policy \( \pi_{\theta} \), \( G_t \) -- again, can do that inside the expectation.

\[ \nabla_{\theta} J_{\text{avV}}(\theta) = E_{\pi_{\theta}} \left[ \gamma^t G_t \frac{\nabla_{\theta} \pi_{\theta}(A_t|S_t)}{\pi_{\theta}(A_t|S_t)} \right] = E_{\pi_{\theta}} \left[ \gamma^t G_t \nabla_{\theta} \log \pi_{\theta}(A_t|S_t) \right] \]

• Note: \( E[G_0] = V^{\pi_{\theta}}(S_0) \)
REINFORCE: Intro

\[ \nabla_{\theta} J_{avV}(\theta) = E_{\pi_{\theta}} \left[ \gamma^t G_t \frac{\nabla_{\theta} \pi_{\theta}(A_t | S_t)}{\pi_{\theta}(A_t | S_t)} \right] = E_{\pi_{\theta}} \left[ \gamma^t G_t \nabla_{\theta} \log \pi_{\theta}(A_t | S_t) \right] \]

- Intuition: a weighted sum of gradients, with more weight given in situations where we get larger total rewards. We upweight gradients for unlikely actions by dividing by \( \pi_{\theta}(A_t | S_t) \), so that we don’t just care about gradients of actions that are currently likely.
REINFORCE

• $\nabla_{\theta} J_{avV}(\theta) = \mathbb{E}_{\pi_{\theta}} \left[ \gamma^t G_t \frac{\nabla_{\theta} \pi_{\theta} (A_t|S_t)}{\pi_{\theta} (A_t|S_t)} \right]$

• Estimate the expectation by simply following policy $\pi_{\theta}$ and recording the rewards you get!

Input: a differentiable policy parameterization $\pi(a|s, \theta), \forall a \in A, s \in S, \theta \in \mathbb{R}^n$
Initialize policy weights $\theta$
Repeat forever:
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \theta)$
    For each step of the episode $t = 0, \ldots, T-1$:
    $G_t \leftarrow$ return from step $t$
    $\theta \leftarrow \theta + \alpha \gamma^t G_t \nabla_{\theta} \log \pi(A_t|S_t, \theta)$

• Note: $G_t$ is the total (discounted) reward starting from time $t$
REINFORCE

• $\nabla_{\theta} J_{\text{avg}}(\theta) = E_{\pi_\theta} \left[ \gamma^t G_t \frac{\nabla_{\theta} \pi_\theta (A_t | S_t)}{\pi_\theta (A_t | S_t)} \right]$

• Overall idea: follow the policy, if it seems that starting from time $t$ we’re getting a big reward, make state $A_t$ more likely
Case Study: AlphaGO

- Go is a remarkably difficult game
  - Lots of possible moves
  - At least $10^{10^{48}}$ possible games
  - Very hard to tell if a position is good or bad
Google Brain’s AlphaGo

- Defeated Lee Sedol, one of the world’s top Go professionals
- The first time a computer program managed to do that
- Highly engineered system with multiple moving parts
AlphaGo’s policy network

• Stage A: a deep convolutional network trained by trying using supervised learning to predict human moves in a game database
  • A ConvNet makes sense since Go “shapes” – configurations of stones – are local, and might be detectable with convolutional layers

• Stage B: use Reinforcement Learning to learn the policy network by making the policy network play against a previous iteration of the policy network
  • Reward: winning a game
  • Train using Policy Gradient

• Use a sophisticated game tree search algorithm together with the Policy Network to actually play the game