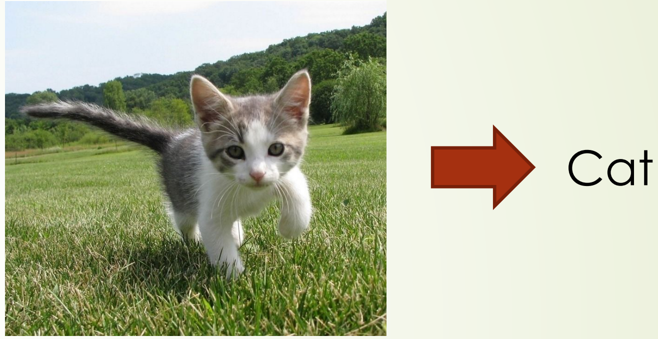
An Introduction to Reinforcement Learning

Yuan YAO

HKUST

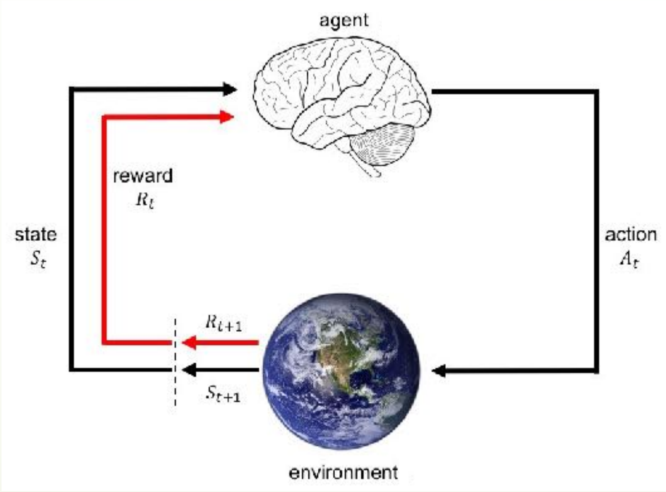
Supervised Learning

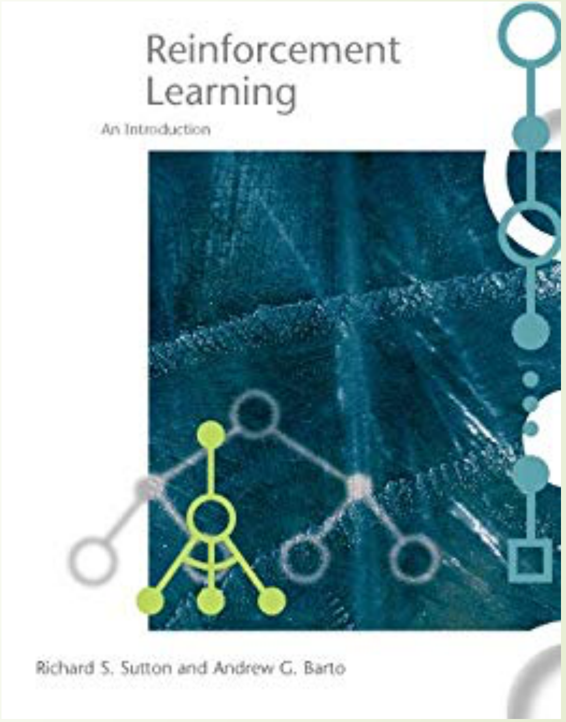
- **Data**: (x, y)
  x is input, y is output/response (label)
- **Goal**: Learn a *function* to map x -> y
- **Examples**:
  - Classification,
  - regression,
  - object detection,
  - semantic segmentation,
  - image captioning, etc.



Today: Reinforcement Learning

- Problems involving an **agent**
- interacting with an **environment**,
- which provides numeric **reward** signals
- **Goal**:
  - Learn how to take actions in order to maximize reward in dynamic scenarios

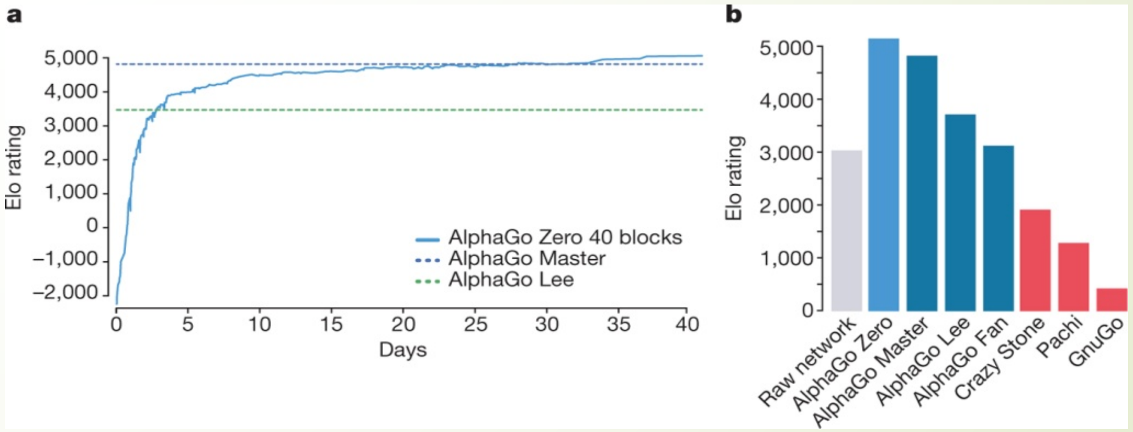




Playing games against human champions

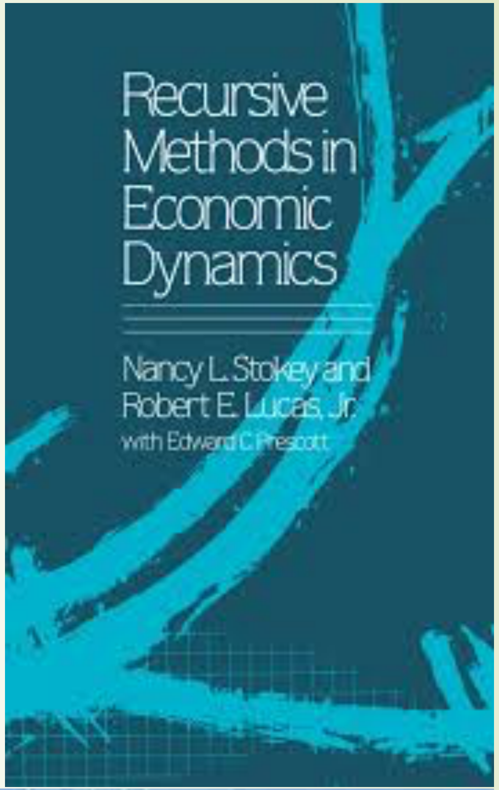
Deep Blue in 1997

AlphaGo "LEE" 2016

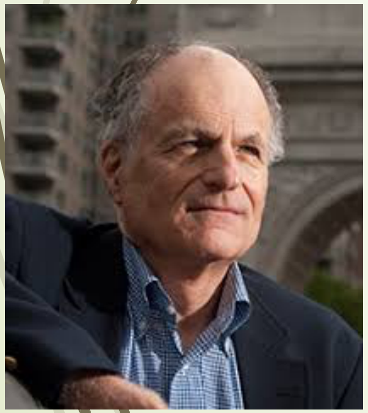


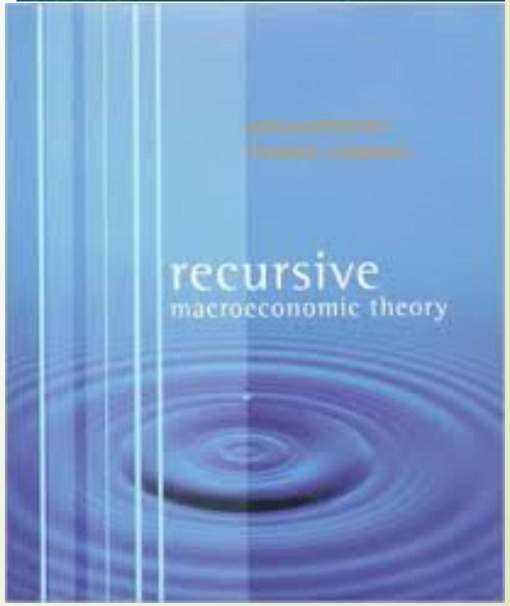
AlphaGo "ZERO" D Silver *et al.* *Nature* **550,** 354–359 (2017) doi:10.1038/nature24270

Markov Decision Process /Dynamic Programming in Economics



- The Sveriges Riksbank Prize in Economic Sciences in Memory of Alfred Nobel 1995 was awarded to **Robert E. Lucas Jr**. **"for having developed and applied the hypothesis of rational expectations, and thereby having transformed macroeconomic analysis and deepened our understanding of economic policy".**





- **Thomas John Sargent** was awarded the Nobel Memorial Prize in Economics in 2011 together with Christopher A. Sims for their **"empirical research on cause and effect in the macroeconomy"**

What supervision does an agent need to learn purposeful behaviors in dynamic environments?

- **Rewards**:
  - sparse feedback from the environment whether the desired goal is achieved e.g., game is won, car has not crashed, agent is out of the maze etc.
  - Rewards can be intrinsic, i.e., generated by the agent and guided by its curiosity as opposed to an external task
- Learning from **rewards**
  - Reward: jump as high as possible: It took years for athletes to find the right behavior to achieve this
- Learns from **demonstrations**
  - It was way easier for athletes to perfection the jump, once someone showed the right general trajectory
- Learns from specifications of optimal behavior
  - For novices, it is much easier to replicate this behavior if additional guidance is provided based on specifications: where to place the foot, how to time yourself etc.

How learning goal-seeking behaviors is different to supervised learning paradigms?

- The agent's **actions** affect the data she will receive in the future
- The **reward** (whether the goal of the behavior is achieved) is far in the future:
  - Temporal credit assignment: which actions were important and which were not, is hard to know
  - **Isn't it the same with loss of multi-layer deep networks?**
  - **No: here the horizon involves acting in the environment, rather than going from one neural layer to the next, we cannot apply chain rule to back propagate the gradient of rewards.**
  - But another way of "**Back Propagation**": **Bellman's Dynamic Programing** principle
- Actions take time to carry out in the real world, and thus this may limit the amount of experience
  - We can use simulated experience with multiple agents.

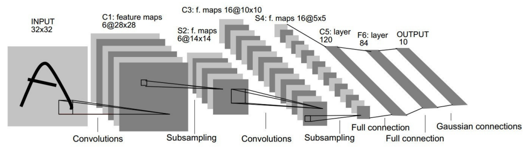
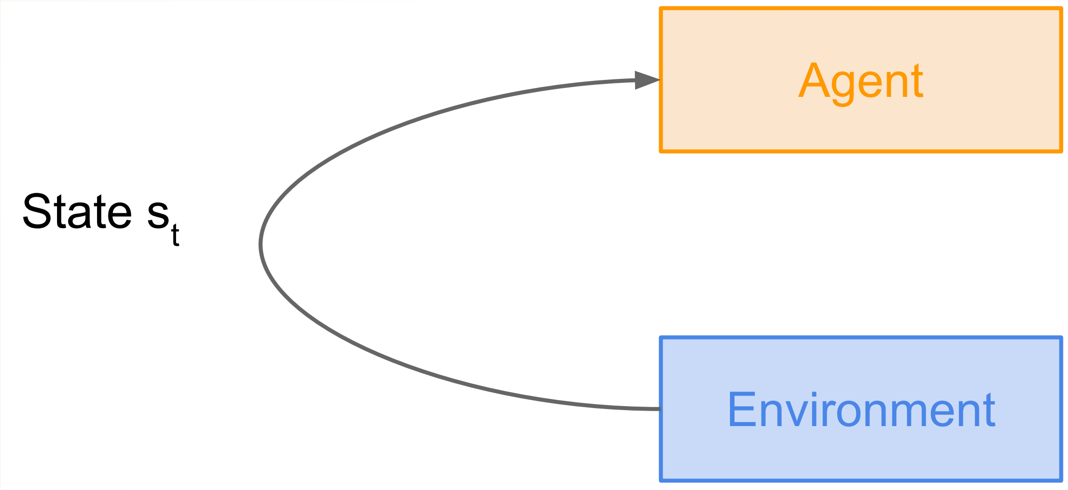


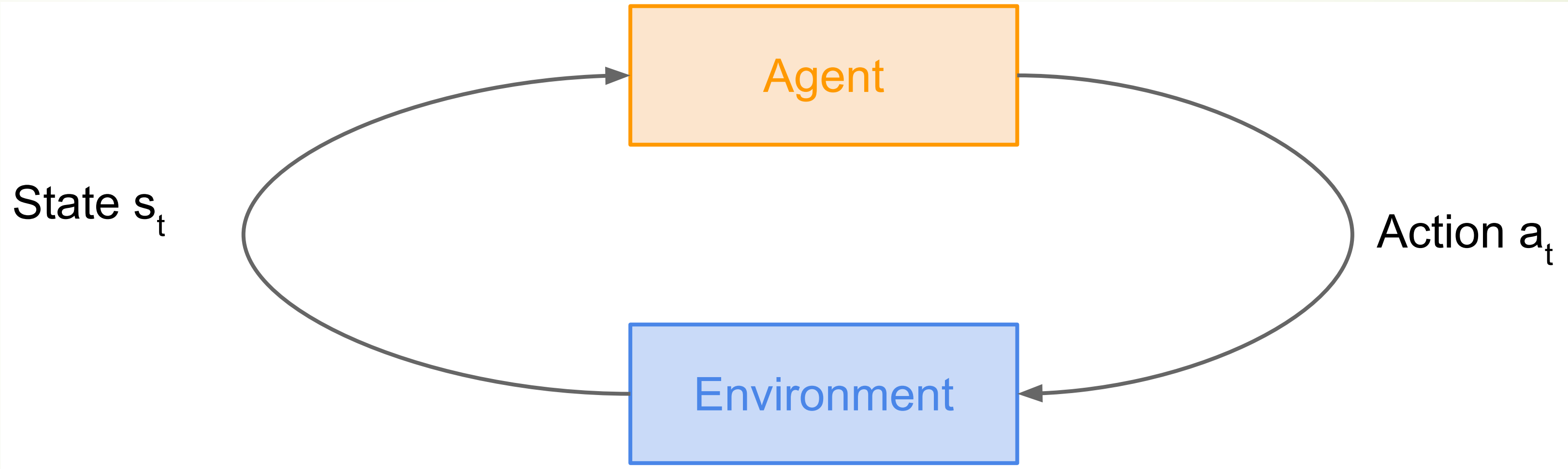
Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

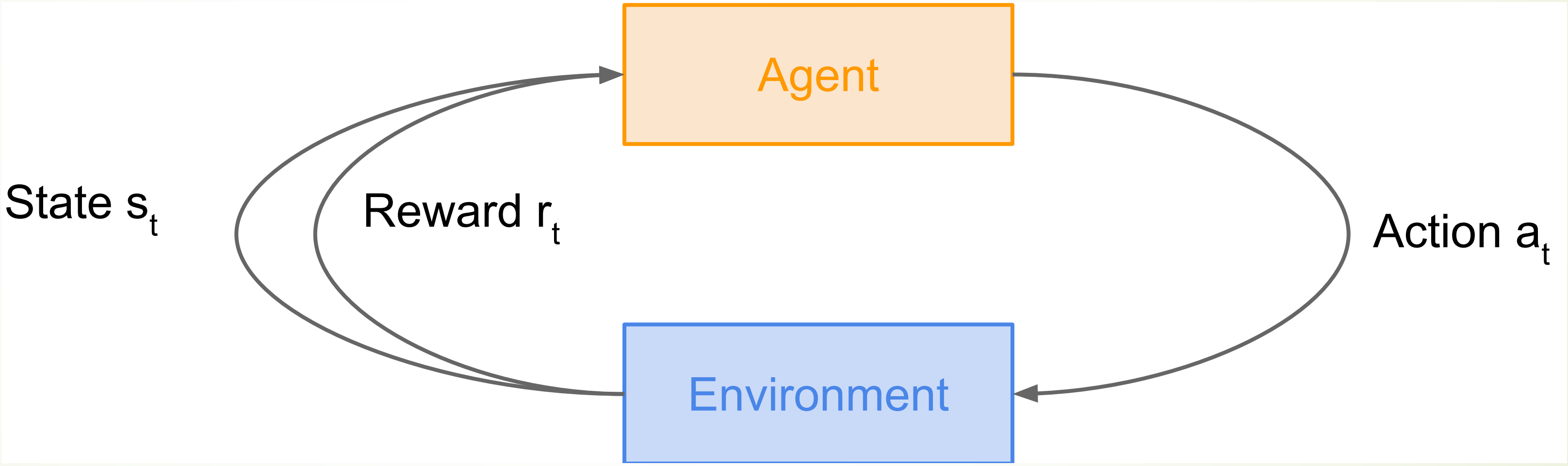
Outline

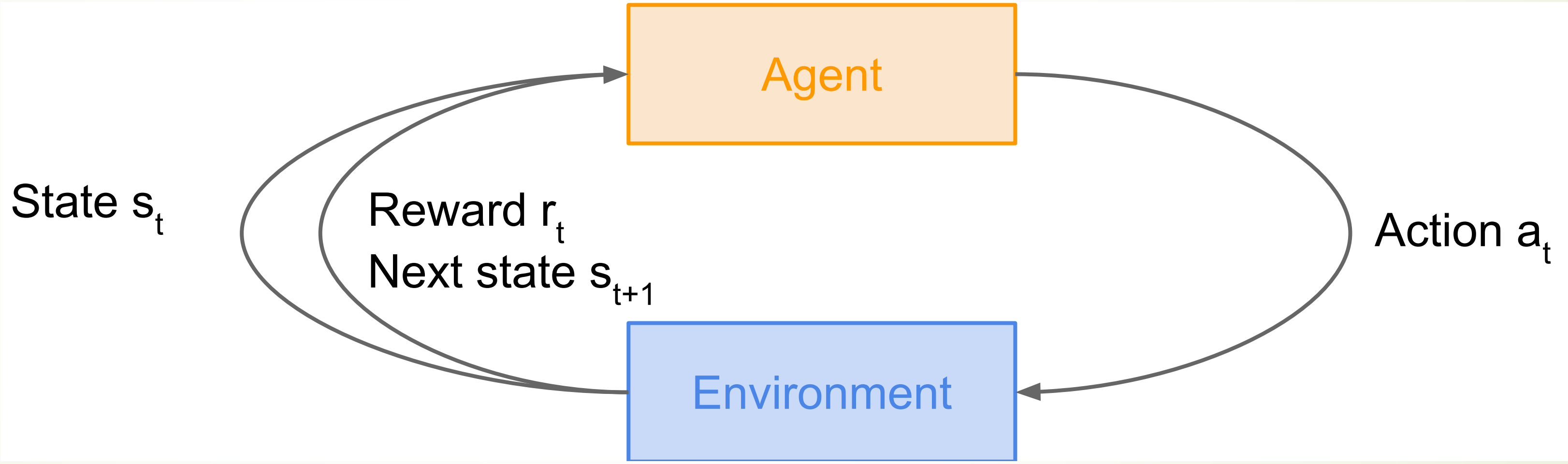
- What is Reinforcement Learning?
- Markov Decision Processes
- Bellman Equation as Linear Programming
- Q-Learning
- Policy Gradients
- Actor-Critics (Q-learning+Policy gradient)
- Examples:
  - Deep RL for quantitative trading
  - Order Book Optimization via Discrete Q-Learning by Prof. Michael Kearns

Reinforcement Learning

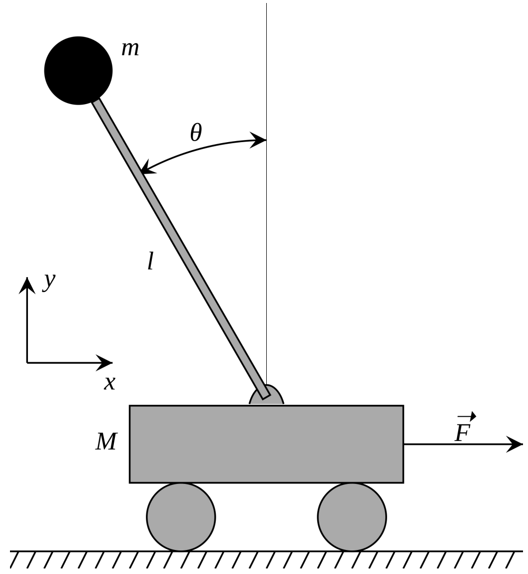

Agent
State s_t
Environment


Agent
State s_t
Action a_t
Environment


Agent
State s_t
Reward r_t
Action a_t
Environment


Agent
State s_t
Reward r_t
Next state s_{t+1}
Action a_t
Environment

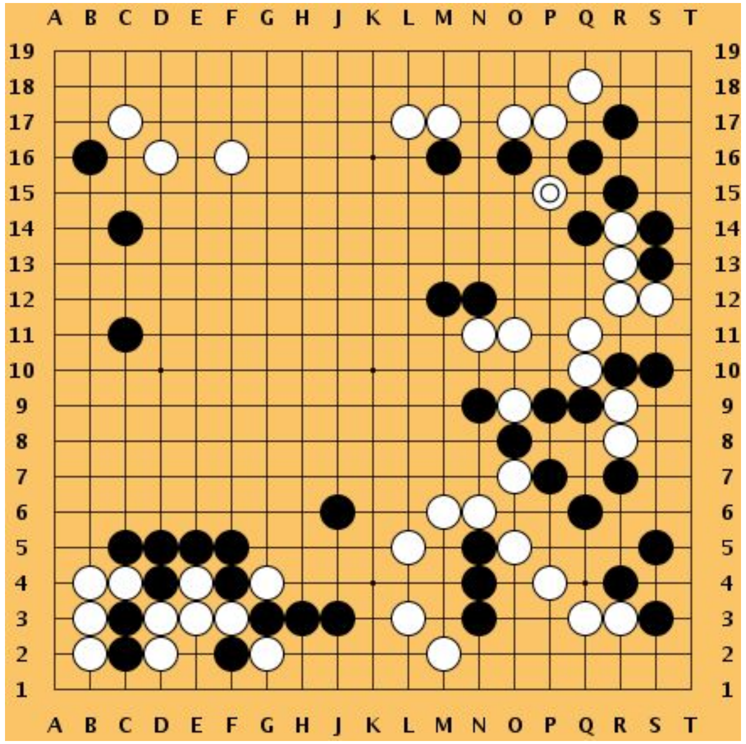
Car-Pole Control Problem



Objective: Balance a pole on top of a movable cart

State: angle, angular speed, position, horizontal velocity
Action: horizontal force applied on the cart
Reward: 1 at each time step if the pole is upright

Go Game



Objective: Win the game!

State: Position of all pieces
Action: Where to put the next piece down
Reward: 1 if win at the end of the game, 0 otherwise

Mathematical Formulation of Reinforcement Learning

A Markov Decision Process is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$

- $\mathcal{S}$ is a set of states
- $\mathcal{A}$ is a set of actions
- $\mathcal{R}$ is a distribution of reward given (state, action) pair

$$R_{t+1} \sim \mathcal{R}\left[\cdot \mid S_t = s, A_t = a\right]$$

- $\mathbb{P}$ is a state transition probability function, satisfying the **Markov Property**:

$$\begin{aligned} & \mathbb{P}\left[R_{t+1} = r, S_{t+1} = s' \mid S_t, A_t\right] \\ = \; & \mathbb{P}\left[R_{t+1} = r, S_{t+1} = s' \mid S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\right] \end{aligned}$$

- γ is a discount factor $\gamma \in [0, 1]$

- At time step $t=0$, environment samples initial state $s_0 \sim p(s_0)$
- Then, for $t=0$ until done:
  - Agent selects **action** a_t
  - Environment samples **reward** $r_t \sim R(. \mid s_t, a_t)$
  - Environment samples next **state** $s_{t+1} \sim P(. \mid s_t; a_t)$
  - Agent receives reward r_t and next state s_{t+1}
- A **policy** π:S->A is a map from S to A that specifies what action to take in each state
- **Objective**: find policy that maximizes the cumulated discounted reward

Rewards

- They are scalar values (not vector rewards) provided by the environment to the agent that indicate whether goals have been achieved, e.g., ***1 if goal is achieved, 0 otherwise, or -1 for overtime step the goal is not achieved***
- **Episodic tasks**: A sequence of interactions based on which the reward will be judged at the end is called episode. Interaction breaks naturally into episodes, e.g., plays of a game, trips through a maze.
- Goal-seeking behavior of an agent can be formalized as the behavior that seeks maximization of the expected value of the ***cumulative sum of (potentially time discounted) rewards, we call it return. We want to maximize returns.***
  - Return in Finite horizon:
  - Return (discounted) in infinite horizon:

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T$$

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

$$\gamma \in [0, 1]$$

$$r(s, a) = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$$

Dynamics of Environment or Model

- How the states and rewards change given the actions of the agent

$$p(s', r \mid s, a) = \mathbb{P}\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}$$

- Transition function or next step function:

$$T(s' \mid s, a) = p(s' \mid s, a) = \mathbb{P}\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in \mathbb{R}} p(s', r \mid s, a)$$

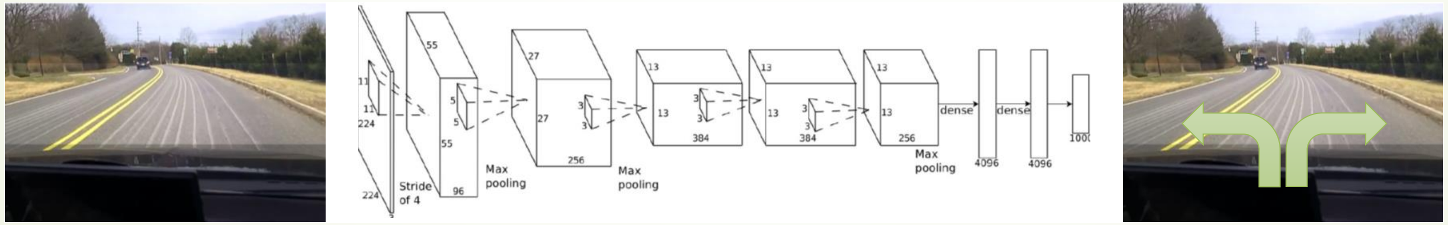
- Model-based RL: dynamics are known or are estimated, and are used for learning the policy
- Model-free RL: we do not know the dynamics, and we do not attempt to estimate them

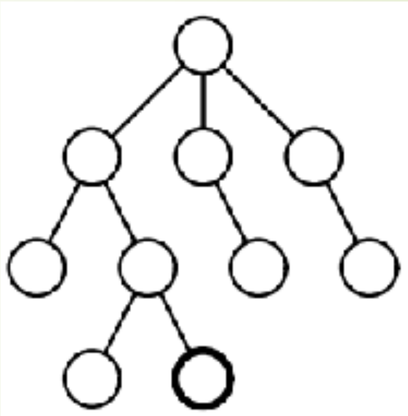
Policy

- A mapping function from states to actions of the end effectors, e.g. stochastic actions:

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$$

- It can be a shallow or deep **network**, or involving a **tree** look-ahead search





The optimal policy **π***

We want to find optimal policy **π*** that maximizes the sum of rewards.

How do we handle the randomness (initial state, transition probability…)?
Maximize the **expected sum of rewards!**

Formally: $\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi\right]$ with $s_0 \sim p(s_0), a_t \sim \pi(\cdot|s_t), s_{t+1} \sim p(\cdot|s_t, a_t)$

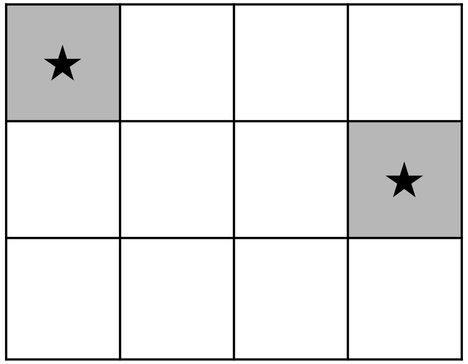
A simple MDP: Grid World

actions = {

1. right ⟶
2. left ⟵
3. up ↑
4. down ↓

}

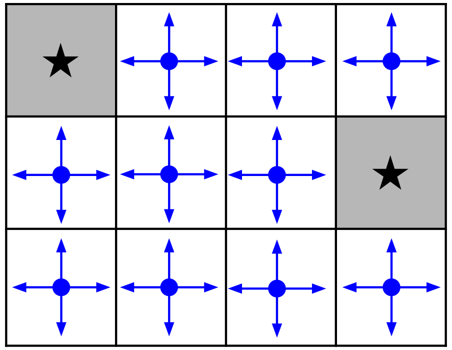
states



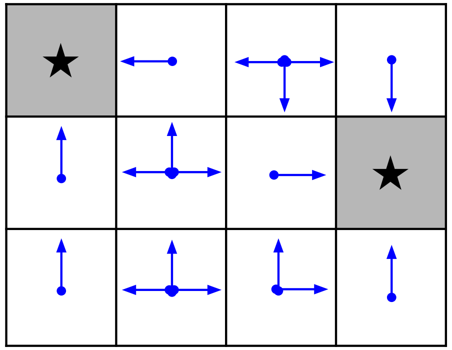
Set a negative "reward" for each transition (e.g. r = -1)

Objective: reach one of terminal states (greyed out) in least number of actions

A simple MDP: Grid World



Random Policy



Optimal Policy

- Finding the optimal policy: ***Bellman's Principle of Dynamic Programming***
  - Begin with the terminal states, find the nearest neighbors (depth-1) states with their optimal move (policy);
  - From depth-1 neighbor cells, find the optimal move (policy) of depth-2 neighbor cells;
  - And so on recursively…

Definitions: Value function and Q-value function

Following a policy produces sample trajectories (or paths) s_0, a_0, r_0, s_1, a_1, r_1, …

How good is a state?
The **value function** at state s, is the expected cumulative reward from following the policy from state s:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, \pi\right]$$

How good is a state-action pair?
The **Q-value function** at state s and action a, is the expected cumulative reward from taking action a in state s and then following the policy:

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Bellman Equation of Optimal Value

Optimal Value Function $V^* : \mathcal{S} \to R = x^*$ satisfied the following nonlinear fixed point equation

$$x^*(i) = \max_{a \in \mathcal{A}} \left\{ r_a(i) + \gamma \sum_{j \in \mathcal{S}} P_a(i,j) x^*(j) \right\}$$

where a policy π^* is an optimal policy if and only if it attains the optimality of the Bellman equation.

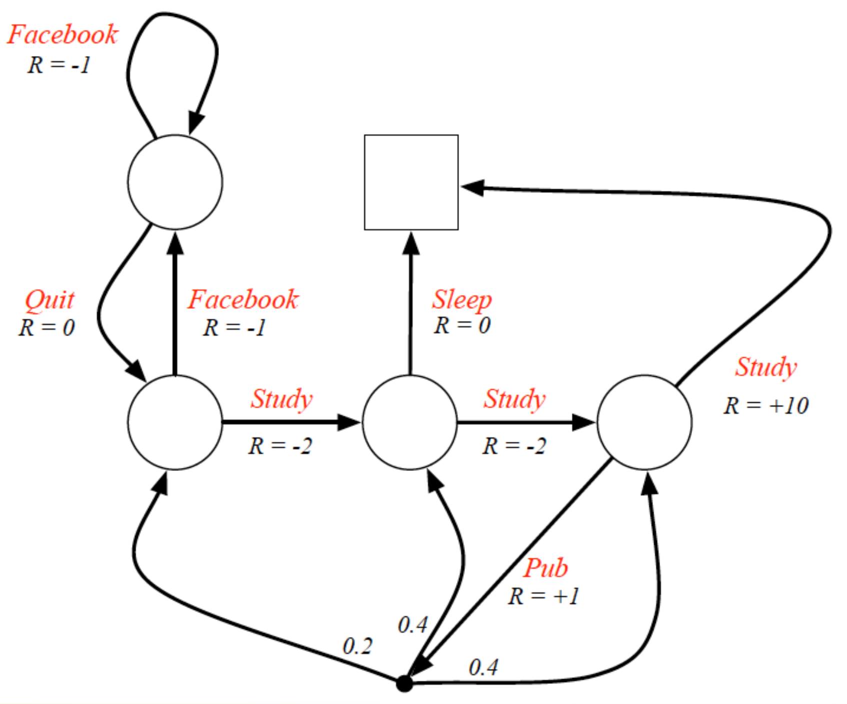
Remarks

- In the continuous-time analog of MDP, i.e., stochastic optimal control, the Bellman equation is the HJB
- Exact solution methods: value iteration, policy iteration, variational analysis
- What makes things hard:

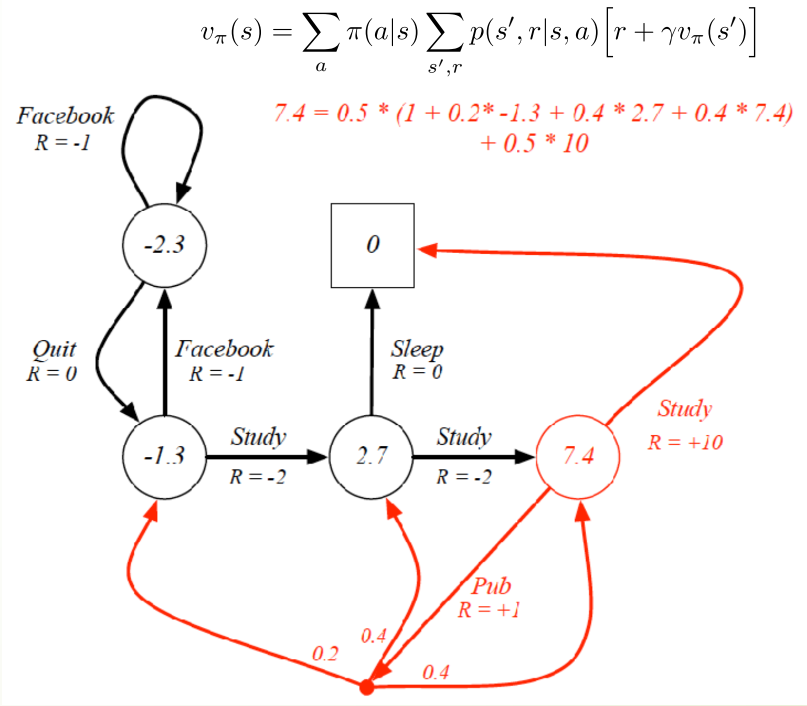
Curse of dimensionality + Modeling Uncertainty

Example: the student MDP

The Student MDP



Value function



Bellman Equation as LP (Farias and Van Roy, 2003)

The Bellman equation is equivalent to

$$\begin{array}{ll} \text{minimize} & e^T x \\ \text{subject to} & (I - \gamma P_a)x - r_a \geq 0, \quad a \in \mathcal{A}, \sum_{i \in \mathcal{S}} e(i) = 1, e > 0. \end{array}$$

- Exact policy iteration is a form of simplex method and exhibits strongly polynomial performance (Ye 2011)
- Again, curse of dimensionality:
- Variable dimension $= |\mathcal{S}|$.
- Number of constraints $= |\mathcal{S}| \times |\mathcal{A}|$.

Duality between Value Function and Policy

Let $\lambda_{i,a} \geq 0$ be the multiplier associated with the i-th row of the primal constraint $\gamma P_a x + r_a \leq x$. The dual problem is

$$\begin{array}{ll} \text{maximize} & \lambda_a^T r_a, \quad a \in \mathcal{A} \\ \text{subject to} & \sum_{a \in \mathcal{A}} (I - \gamma P_a^T)\lambda_a = e, \quad \lambda_a \geq 0, \quad a \in \mathcal{A} \end{array}$$

where the dual variable is high-dimensional $\lambda = (\lambda_a)_{a \in \mathcal{A}} \in \mathbb{R}^{|\mathcal{A}||\mathcal{S}|}$.

Theorem

The optimal dual solution $\lambda^* = (\lambda^*_{i,a})_{i \in \mathcal{S}, a \in \mathcal{A}}$ is sparse and has exact $|S|$ nonzeros. It satisfies

$$\left(\lambda^*_{i,\mu^*(i)}\right)_{i \in \mathcal{S}} = (I - \alpha P_{\mu^*}^T)^{-1} e,$$

and $\lambda^*_{i,a} = 0$ if $a \neq \mu^*(i)$.

Finding the optimal policy μ^* = Finding the basis of the dual solution λ^*

Stochastic Primal-Dual Value-Policy Iteration (Mengdi Wang 2017, arXiv:1704.01869)

Stochastic primal-dual (value-policy) algorithm

- **Input:** Simulation Oracle $\mathcal{M}$, $n = |\mathcal{S}|$, $m = |\mathcal{A}|$, $\alpha \in (0, 1)$.
- Initialize $x^{(0)}$ and $\lambda = (\lambda_u^{(0)} : u \in \mathcal{A})$ arbitrarily.
- For$k = 1, 2, \ldots, T$
  - Sample i_k uniformly from $\mathcal{S}$ and sample u_k uniformly from $\mathcal{A}$.
  - Sample next state j_k and immediate reward $g_{i_k j_k u_k}$ conditioned on (i_k, u_k) from $\mathcal{M}$.
  - Update the iterates by

$$x^{(k-\frac{1}{2})} = x^{(k-1)} - \gamma_k \Big(-e + m\lambda_{u_k}^{(k-1)} - \alpha mn \left(\lambda_{u_k}^{(k-1)} \cdot e_{i_k}\right) e_{j_k} \Big),$$

$$\lambda_{u_k}^{(k-\frac{1}{2})} = \lambda_{u_k}^{(k-1)} + m\gamma_k \Big(x^{(k-1)} - \alpha n \left(x^{(k-1)} \cdot e_{j_k} \right) e_{i_k} - n g_{i_k j_k u_k} e_{i_k} \Big),$$

$$\lambda_u^{(k-\frac{1}{2})} = \lambda_u^{(k-1)}, \qquad \forall\ u \neq u_k,$$

  - Project the iterates orthogonally to some regularization constraints

$$x^{(k)} = \Pi_X x^{(k-\frac{1}{2})}, \qquad \lambda^{(k)} = \Pi_\Lambda \lambda^{(k-\frac{1}{2})}.$$

- **Ouput:** Averaged dual iterate $\hat{\lambda} = \frac{1}{T} \sum_{k=1}^{T} \lambda^{(k)}$

Near Optimal Primal-Dual Algorithms

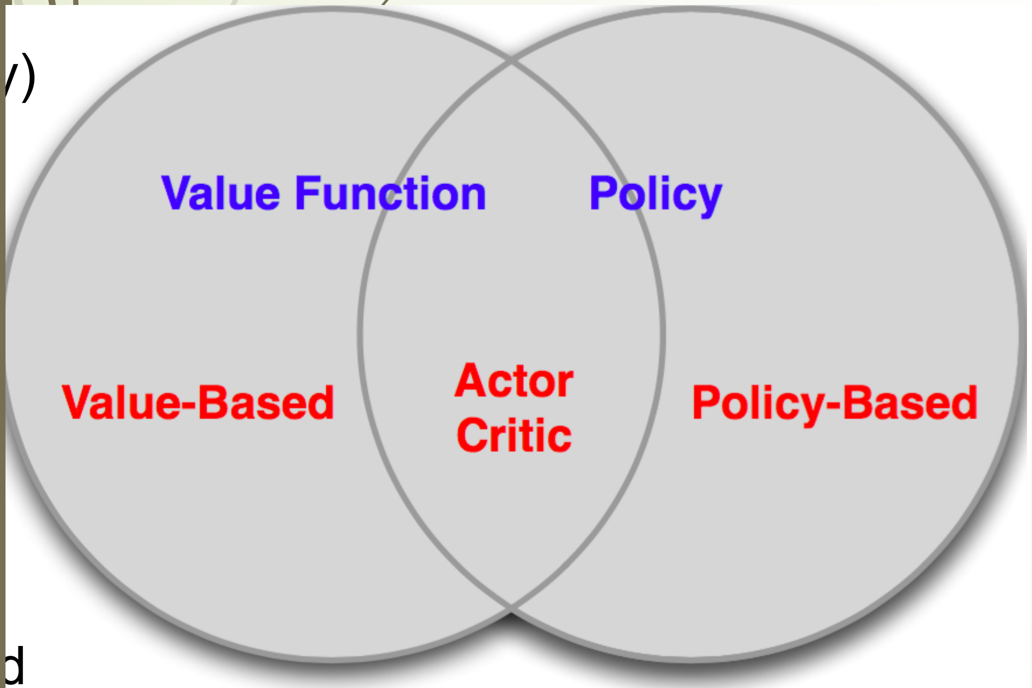
Method	Setting	Sample Complexity	Run-Time Complexity	Space Complexity	Reference
Phased Q-Learning	γ discount factor, ϵ-optimal value	$\frac{\|\mathcal{S}\|\|\mathcal{A}\|}{(1-\gamma)^3\epsilon^2}\ln\frac{1}{\delta}$	$\frac{\|\mathcal{S}\|\|\mathcal{A}\|}{(1-\gamma)^3\epsilon^2}\ln\frac{1}{\delta}$	$\|\mathcal{S}\|\|\mathcal{A}\|$	[17]
Model-Based Q-Learning	γ discount factor, ϵ-optimal value	$\frac{\|\mathcal{S}\|\|\mathcal{A}\|}{(1-\gamma)^3\epsilon^2}\ln\frac{\|\mathcal{S}\|\|\mathcal{A}\|}{\delta}$	NA	$\|\mathcal{S}\|^2\|\mathcal{A}\|$	[1]
Randomized P-D	γ discount factor, ϵ-optimal policy	$\frac{\|\mathcal{S}\|^3\|\mathcal{A}\|}{(1-\gamma)^6\epsilon^2}$	$\frac{\|\mathcal{S}\|^3\|\mathcal{A}\|}{(1-\gamma)^6\epsilon^2}$	$\|\mathcal{S}\|\|\mathcal{A}\|$	[25]
Randomized P-D	γ discount factor, τ-stationary, ϵ-optimal policy	$\tau^4\frac{\|\mathcal{S}\|\|\mathcal{A}\|}{(1-\gamma)^4\epsilon^2}$	$\tau^4\frac{\|\mathcal{S}\|\|\mathcal{A}\|}{(1-\gamma)^4\epsilon^2}$	$\|\mathcal{S}\|\|\mathcal{A}\|$	[25]
Randomized VI	γ discount factor, ϵ-optimal policy	$\frac{\|\mathcal{S}\|\|\mathcal{A}\|\cdot}{(1-\gamma)^4\epsilon^2}$	$\frac{\|\mathcal{S}\|\|\mathcal{A}\|\cdot}{(1-\gamma)^4\epsilon^2}$	$\|\mathcal{S}\|\|\mathcal{A}\|$	[23]
Primal-Dual π Learning	τ-stationary, t^*_{mix}-mixing, ϵ-optimal policy	$\frac{(\tau\cdot t^*_{mix})^2\|\mathcal{S}\|\|\mathcal{A}\|}{\epsilon^2}$	$\frac{(\tau\cdot t^*_{mix})^2\|\mathcal{S}\|\|\mathcal{A}\|}{\epsilon^2}$	$\|\mathcal{S}\|\|\mathcal{A}\|$	This Paper

Table 1: Complexity Results for Sampling-Based Methods for MDP. The sample complexity is measured by the number of queries to the $\mathcal{SO}$. The run-time complexity is measured by the total run-time complexity under the assumption that each query takes $\tilde{\mathcal{O}}(1)$ time. The space complexity is the additional space needed by the algorithm in addition to the input.

Mengdi Wang, Primal-Dual π Learning, arXiv:1710.0610

Approaches of Deep RL: approximate dynamic programming

- Value-based RL
  - Learn an optimal value function $Q_*(s,a)$ or $V_*(s)$
  - Implicit derivation of policy
  - Deep Q-Learning (DQN), Double DQN, Dueling DQN
- Policy-based RL
  - Learn directly an optimal policy π_*
  - This is the policy achieving maximum future reward
  - Policy Gradient (PG)
- Actor-Critic RL
  - Learn a value function and a policy
  - A2C, SAC
- Model-based RL (not here)
  - Build a model of the environment
  - Plan (e.g. by look-ahead) using model



Q-Learning

Bellman equation

The optimal Q-value function Q* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Q* satisfies the following **Bellman equation**:

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') | s, a\right]$$

Intuition: if the optimal state-action values for the next time-step Q*(s',a') are known, then the optimal strategy is to take the action that maximizes the expected value of $r + \gamma Q^*(s',a')$

The optimal policy **π*** corresponds to taking the best action in any state as specified by Q*

Solving for the optimal policy

Value iteration algorithm: Use Bellman equation as an iterative update

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a') | s,a\right]$$

Q_i will converge to Q^* as i -> infinity

What's the problem with this?
Not scalable. Must compute Q(s,a) for every state-action pair. If state is e.g. current game state pixels, computationally infeasible to compute for entire state space!

Solution: use a function approximator to estimate Q(s,a). E.g. a neural network!

Solving for the optimal policy: Q-learning

Q-learning: Use a function approximator to estimate the action-value function

$$Q(s, a; \theta) \approx Q^*(s, a)$$

If the function approximator is a deep neural network => **deep q-learning**!

Solving for the optimal policy: Q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a')|s,a\right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a\right]$

Backward Pass

Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot);s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

Yet, such a training might be unstable …

- Learning from batches of consecutive samples is problematic:
  - Samples are correlated => inefficient learning
  - Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand size) => can lead to bad feedback loops
- Experience replay will help!

DQN: Experience Replay

- To remove correlations, build a replay memory data-set D from agent's own experience

s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
s_3, a_3, r_4, s_4
$\ldots$
$s_t, a_t, r_{t+1}, s_{t+1}$

$\rightarrow \quad s, a, r, s'$

- Sample random mini-batch of transitions (s,a,r,s') from D, instead of consecutive samples
- Compute Q-learning targets w.r.t. old, fixed parameters w−
- Optimize MSE between Q-network and Q-learning target by SGD, where each transition can also contribute to multiple weight updates => greater data efficiency

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s', a'; w_i^-)}_{\text{Q-learning target}} - \underbrace{Q(s, a; w_i)}_{\text{Q-network}}\right)^2\right]$$

[Mnih et al. NIPS Workshop 2013; Nature 2015]

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for

[Mnih et al. NIPS Workshop 2013; Nature 2015]

Case Study: Playing Atari Games

Objective: Complete the game with the highest score

State: Raw pixel inputs of the game state
Action: Game controls e.g. Left, Right, Up, Down
Reward: Score increase/decrease at each time step

[Mnih et al. NIPS Workshop 2013; Nature 2015]

Q-network Architecture

$Q(s, a; \theta)$: neural network with weights θ

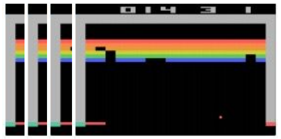
A single feedforward pass to compute Q-values for all actions from the current state => efficient!

FC-4 (Q-values)

FC-256

32 4x4 conv, stride 2

16 8x8 conv, stride 4



Last FC layer has 4-d output (if 4 actions), corresponding to $Q(s_t, a_1)$, $Q(s_t, a_2)$, $Q(s_t, a_3)$, $Q(s_t,a_4)$

Number of actions between 4-18 depending on Atari game

Current state s_t: 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

Example

- Google DeepMind's Deep Q-learning playing Atari Breakout:
  - https://www.youtube.com/watch?v=V1eYniJ0Rnk
  - Google DeepMind created an artificial intelligence program using deep reinforcement learning that plays Atari games and improves itself to a superhuman level. It is capable of playing many Atari games and uses a combination of deep artificial neural networks and reinforcement learning. After presenting their initial results with the algorithm, Google almost immediately acquired the company for several hundred million dollars, hence the name Google DeepMind. Please enjoy the footage and let me know if you have any questions regarding deep learning!

Prioritized Replay: importance sampling

[Schaul, Quan, Antonoglou, Silver, ICLR 2016]

- Current Q-network w is used to select actions
- Older Q-network w− is used to evaluate actions

Action evaluation: w−

$$I = \Big(r + \gamma Q(s', \operatorname*{argmax}_{a'} Q(s', a', \mathbf{w}), \mathbf{w}^-) - Q(s, a, \mathbf{w})\Big)^2$$

Action selection: w

$$\left| r + \gamma \max_{a'} Q(s', a', \mathbf{w}^-) - Q(s, a, w) \right|$$

- Importance Weight experience according to ``surprise'' (or error):

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

- Store experience in priority according to DQN error:
- α determines how much prioritization is used, with $\alpha = 0$ corresponding to the uniform case.

Maximization Bias

- We often need to maximize over our value estimates. The estimated maxima suffer from maximization bias
- Consider a state for which all ground-truth $Q_*(s,a)=0$. Our estimates $Q(s,a)$ are uncertain, some are positive and some negative. Q(s,argmax_a(Q(s,a)) is positive while Q_*(s,argmax_a(Q_*(s,a))=0.

Double Q-Learning (DDQN)

- Train 2 **action-value** functions, Q_1 and Q_2
- Do Q-learning on both, but
  - never on the same time steps (Q_1 and Q_2 are independent)
  - pick Q_1 or Q_2 at random to be updated on each step
- If updating Q_1, use Q_2 for the value of the next state:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \\ + \alpha\Big(R_{t+1} + Q_2\big(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)\big) - Q_1(S_t, A_t)\Big)$$

- Action selections are with respect to the sum of Q_1 and Q_2

Double DQN:

Initialize $Q_1(s,a)$ and $Q_2(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily
Initialize $Q_1(\textit{terminal-state}, \cdot) = Q_2(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        Choose A from S using policy derived from Q_1 and Q_2 (e.g., ε-greedy in $Q_1 + Q_2$)
        Take action A, observe R, S'
        With 0.5 probabilility:

$$Q_1(S,A) \leftarrow Q_1(S,A) + \alpha\Big(R + \gamma Q_2\big(S', \arg\max_a Q_1(S',a)\big) - Q_1(S,A)\Big)$$

        else:

$$Q_2(S,A) \leftarrow Q_2(S,A) + \alpha\Big(R + \gamma Q_1\big(S', \arg\max_a Q_2(S',a)\big) - Q_2(S,A)\Big)$$

        $S \leftarrow S'$;
    until S is terminal

Summary of Q-Learning

- We have introduced Q-learning with several variants:
  - DQN, Double DQN, Dueling DQN
  - Experience replay, prioritization
- What is a problem with Q-learning?
  - The Q-function can be very complicated!
  - Example: a robot grasping an object has a very high-dimensional state => hard to learn exact value of every (state, action) pair
- But the policy can be much simpler: just close your hand
- Can we learn a policy directly, e.g. finding the best policy from a collection of policies?

Policy Gradients

Policy Gradients

Formally, let's define a class of parametrized policies: $\Pi = \{\pi_\theta, \theta \in \mathbb{R}^m\}$

For each policy, define its value:

$$J(\theta) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi_\theta\right]$$

We want to find the optimal policy $\theta^* = \arg\max_\theta J(\theta)$

How can we do this?

Gradient ascent on policy parameters!

REINFORCE algorithm

Mathematically, we can write:

$$J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)}\left[r(\tau)\right]$$
$$= \int_{\tau} r(\tau)p(\tau;\theta)\mathrm{d}\tau$$

Where r(τ) is the reward of a trajectory $\tau = (s_0, a_0, r_0, s_1, \ldots)$

Expected reward:
$$J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)}[r(\tau)]$$
$$= \int_\tau r(\tau)p(\tau;\theta)\mathrm{d}\tau$$

Now let's differentiate this:
$$\nabla_\theta J(\theta) = \int_\tau r(\tau)\nabla_\theta p(\tau;\theta)\mathrm{d}\tau$$

Intractable! Gradient of an expectation is problematic when p depends on θ

However, we can use a nice trick:
$$\nabla_\theta p(\tau;\theta) = p(\tau;\theta)\frac{\nabla_\theta p(\tau;\theta)}{p(\tau;\theta)} = p(\tau;\theta)\nabla_\theta \log p(\tau;\theta)$$

If we inject this back:
$$\nabla_\theta J(\theta) = \int_\tau \left(r(\tau)\nabla_\theta \log p(\tau;\theta)\right) p(\tau;\theta)\mathrm{d}\tau$$
$$= \mathbb{E}_{\tau \sim p(\tau;\theta)}[r(\tau)\nabla_\theta \log p(\tau;\theta)]$$

Can estimate with Monte Carlo sampling

REINFORCE algorithm

$$\nabla_\theta J(\theta) = \int_\tau (r(\tau)\nabla_\theta \log p(\tau;\theta))\, p(\tau;\theta)\mathrm{d}\tau$$
$$= \mathbb{E}_{\tau\sim p(\tau;\theta)}\left[r(\tau)\nabla_\theta \log p(\tau;\theta)\right]$$

Can we compute those quantities without knowing the transition probabilities?

We have: $p(\tau;\theta) = \prod_{t\geq 0} p(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$

Thus: $\log p(\tau;\theta) = \sum_{t\geq 0} \log p(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t)$

And when differentiating: $\nabla_\theta \log p(\tau;\theta) = \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(a_t|s_t)$

Doesn't depend on transition probabilities!

Therefore when sampling a trajectory τ, we can estimate J(θ) with

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0} r(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)$$

Intuition

Gradient estimator:

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Interpretation:

- If r(τ) is high, push up the probabilities of the actions seen
- If r(τ) is low, push down the probabilities of the actions seen

Might seem simplistic to say that if a trajectory is good then all its actions were good. But in expectation, it averages out!

However, this also suffers from high variance because credit assignment is really hard. Can we help the estimator?

Variance reduction

Gradient estimator: $$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

First idea: Push up probabilities of an action seen, only by the cumulative future reward from that state

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} r_{t'} \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Second idea: Use discount factor γ to ignore delayed effects

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} r_{t'} \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Variance reduction: Baseline

Problem: The raw value of a trajectory isn't necessarily meaningful. For example, if rewards are all positive, you keep pushing up probabilities of actions.

What is important then? Whether a reward is better or worse than what you expect to get

Idea: Introduce a baseline function dependent on the state.
Concretely, estimator is now:

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} r_{t'} - b(s_t) \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

How to choose the baseline?

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0} \left(\sum_{t'\geq t} \gamma^{t'-t} r_{t'} - b(s_t) \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

A simple baseline: constant moving average of rewards experienced so far from all trajectories

Variance reduction techniques seen so far are typically used in “Vanilla REINFORCE”

How to choose the baseline?

A better baseline: Want to push up the probability of an action from a state, if this action was better than the **expected value of what we should get from that state**.

Q: What does this remind you of?

A: Q-function and value function!

Intuitively, we are happy with an action a_t in a state s_t if $Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)$ is large. On the contrary, we are unhappy with an action if it's small.

Using this, we get the estimator: $\nabla_\theta J(\theta) \approx \sum_{t \geq 0} (Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)) \nabla_\theta \log \pi_\theta(a_t | s_t)$

Actor-Critic Algorithm

Problem: we don't know Q and V. Can we learn them?

Yes, using Q-learning! We can combine Policy Gradients and Q-learning by training both an **actor** (the policy) and a **critic** (the Q-function).

- The actor decides which action to take, and the critic tells the actor how good its action was and how it should adjust
- Also alleviates the task of the critic as it only has to learn the values of (state, action) pairs generated by the policy
- Can also incorporate Q-learning tricks e.g. experience replay
- **Remark:** we can define by the **advantage function** how much an action was better than expected

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

Actor-Critic Model

- Learn both **actor** (policy **π**) and **critic** (value Q and V)
  - Actor decides which action to take $\pi_\theta(a|s)$
  - **Advantage** function in critic tells how much an action might be better than expected:

$$A^{\pi_\theta}(s,a;w) = Q^{\pi_\theta}(s,a;w) - V^{\pi_\theta}(s;w)$$

- Policy gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) A^{\pi_\theta}(s,a)\right]$$

  - Stochastic Advantage can be approximated by TD-error (Temporal-Difference error)

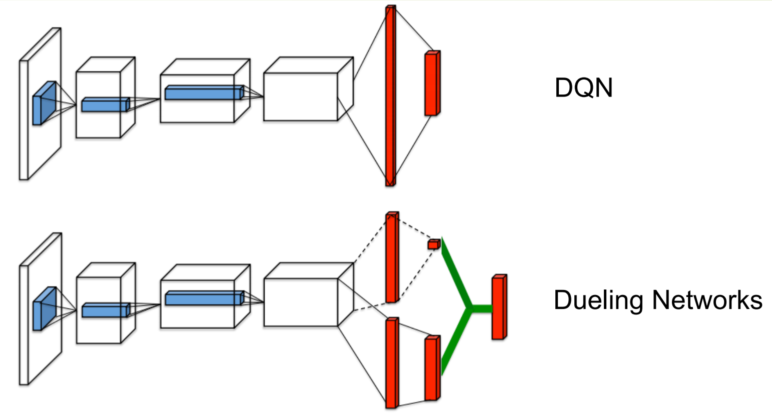
$$\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s)$$

One-step Actor–Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S, \boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S, \boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Dueling DQN

[Wang et.al., ICML, 2016]



- Split Q-network into two channels:
  - Action-independent value function $V(s; \mathbf{w})$
  - Action-dependent advantage function $A(s, a; \mathbf{w})$

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

- Dueling DQN learns Q-function using

$$Q(s,a;\mathbf{w}) = V(s;\mathbf{w}) + \left(A(s,a;\mathbf{w}) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a';\mathbf{w}) \right)$$

PG Summary

- Policy Gradient:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)}$$

- Policy Gradient with Baseline:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \Big(G_t - b(S_t) \Big) \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)}$$

- Actor-Critic Policy Gradient:

$$\theta_{t+1} = \theta_t + \alpha(R_t + \gamma \hat{v}(S_{t+1}) - \hat{v}(S_t)) \frac{\nabla \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)}$$

Maximal Entropy RL

- Promoting the stochastic policies

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi} \left[\sum_{t=1}^{T} \underbrace{R(s_t, a_t)}_{\text{reward}} + \alpha \underbrace{\mathrm{H}(\pi(\cdot \mid s_t))}_{\text{entropy}} \right]$$

- Why?
  - Better exploration
  - Learning alternative ways of accomplishing the task
  - Better generalization, e.g., in the presence of obstacles a stochastic policy may still succeed.

- "Soft" Bellman Equation:

$$Q^{\pi}(s,a) = r(s,a) + \mathbb{E}_{s',a'}\left[Q^{\pi}\left(s',a'\right) - \log\left(\pi\left(a' \mid s'\right)\right)\right]$$

- "Soft" Value function:

$$V\left(s\right) = \mathbb{E}_{a \sim \pi}\left[Q\left(s,a\right) - \log \pi\left(a \mid s\right)\right]$$

Soft version of actor-critic model

- Learn the following value and policy functions: $V_\psi(s_t)$ $Q_\theta(s_t, a_t)$ $\pi_\phi(a_t | s_t)$
  - Gradient for the **state**-value function V:

$$J_V(\psi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}} \left[\frac{1}{2} \left(V_\psi(\mathbf{s}_t) - \mathbb{E}_{\mathbf{a}_t \sim \pi_\phi} \left[Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - \log \pi_\phi(\mathbf{a}_t | \mathbf{s}_t) \right] \right)^2 \right]$$

$$\hat{\nabla}_\psi J_V(\psi) = \nabla_\psi V_\psi(\mathbf{s}_t) \left(V_\psi(\mathbf{s}_t) - Q_\theta(\mathbf{s}_t, \mathbf{a}_t) + \log \pi_\phi(\mathbf{a}_t | \mathbf{s}_t) \right)$$

  - Gradient for the **state-action** value Q-function:

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \mathcal{D}} \left[\frac{1}{2} \left(Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - \hat{Q}(\mathbf{s}_t, \mathbf{a}_t) \right)^2 \right]$$

$$\hat{Q}(\mathbf{s}_t, \mathbf{a}_t) = r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \mathbb{E}_{\mathbf{s}_{t+1} \sim p} \left[V_{\bar{\psi}}(\mathbf{s}_{t+1}) \right]$$

$$\hat{\nabla}_\theta J_Q(\theta) = \nabla_\theta Q_\theta(\mathbf{a}_t, \mathbf{s}_t) \left(Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - r(\mathbf{s}_t, \mathbf{a}_t) - \gamma V_{\bar{\psi}}(\mathbf{s}_{t+1}) \right)$$

- “Soft” Policy gradient:

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}} \left[\mathrm{D_{KL}} \left(\pi_\phi(\cdot | \mathbf{s}_t) \middle\| \frac{\exp(Q_\theta(\mathbf{s}_t, \cdot))}{Z_\theta(\mathbf{s}_t)} \right) \right]$$

$$\nabla_\phi J_\pi(\phi) = \nabla_{\phi} \mathbb{E}_{s_t \in D} \mathbb{E}_{a_t \sim \pi_{\phi}(a|s_t)} \log \frac{\pi_\phi(a_t | s_t)}{\exp(Q_\theta(s_t, a_t))}$$

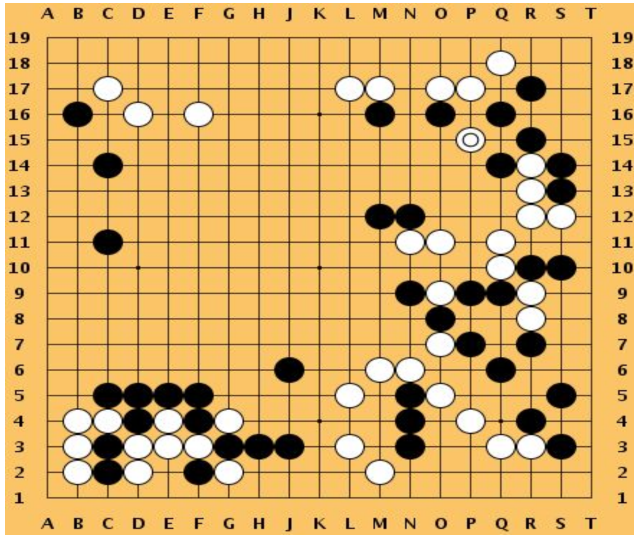
Soft Actor-Critic

- Different to openAI implementation which is essentially SoftDDQN:
  - https://spinningup.openai.com/en/latest/algorithms/sac.html

Algorithm 1 Soft Actor-Critic

Initialize parameter vectors ψ, $\bar{\psi}$, θ, ϕ.
for each iteration **do**
  for each environment step **do**
    $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$
    $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
  end for
  for each gradient step **do**
    $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    $\bar{\psi} \leftarrow \tau\psi + (1 - \tau)\bar{\psi}$
  end for
end for

More policy gradients: AlphaGo



Overview:

- Mix of supervised learning and reinforcement learning
- Mix of old methods (Monte Carlo Tree Search) and recent ones (deep RL)

How to beat the Go world champion:

- Featurize the board (stone color, move legality, bias, …)
- Initialize policy network with supervised training from professional go games, then continue training using policy gradient (play against itself from random previous iterations, +1 / -1 reward for winning / losing)
- Also learn value network (critic)
- Finally, combine combine policy and value networks in a Monte Carlo Tree Search algorithm to select actions by lookahead search

[Silver et al., Nature 2016]

Summary

- **Q-learning**: does not always work but when it works, usually more sample-efficient. **Challenge**: exploration
- **Policy gradients**: very general but suffer from high variance so requires a lot of samples. **Challenge**: sample-efficiency
- Guarantees:
  - **Policy Gradients**: Converges to a local minima, often good enough!
  - **Q-learning**: Zero guarantees since you are approximating Bellman equation with a complicated function approximater

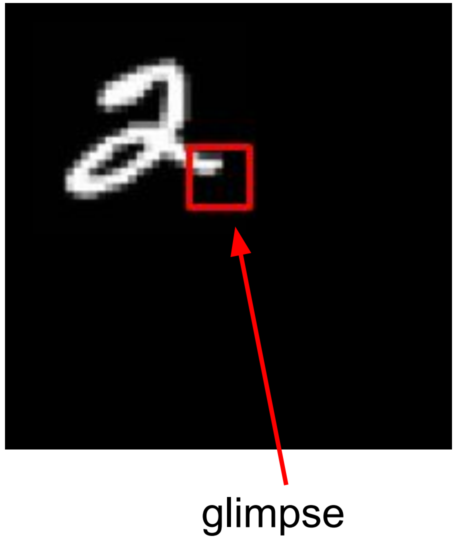
REINFORCE in action: Recurrent Attention Model (RAM)

Objective: Image Classification

Take a sequence of “glimpses” selectively focusing on regions of the image, to predict class

- Inspiration from human perception and eye movements
- Saves computational resources => scalability
- Able to ignore clutter / irrelevant parts of image

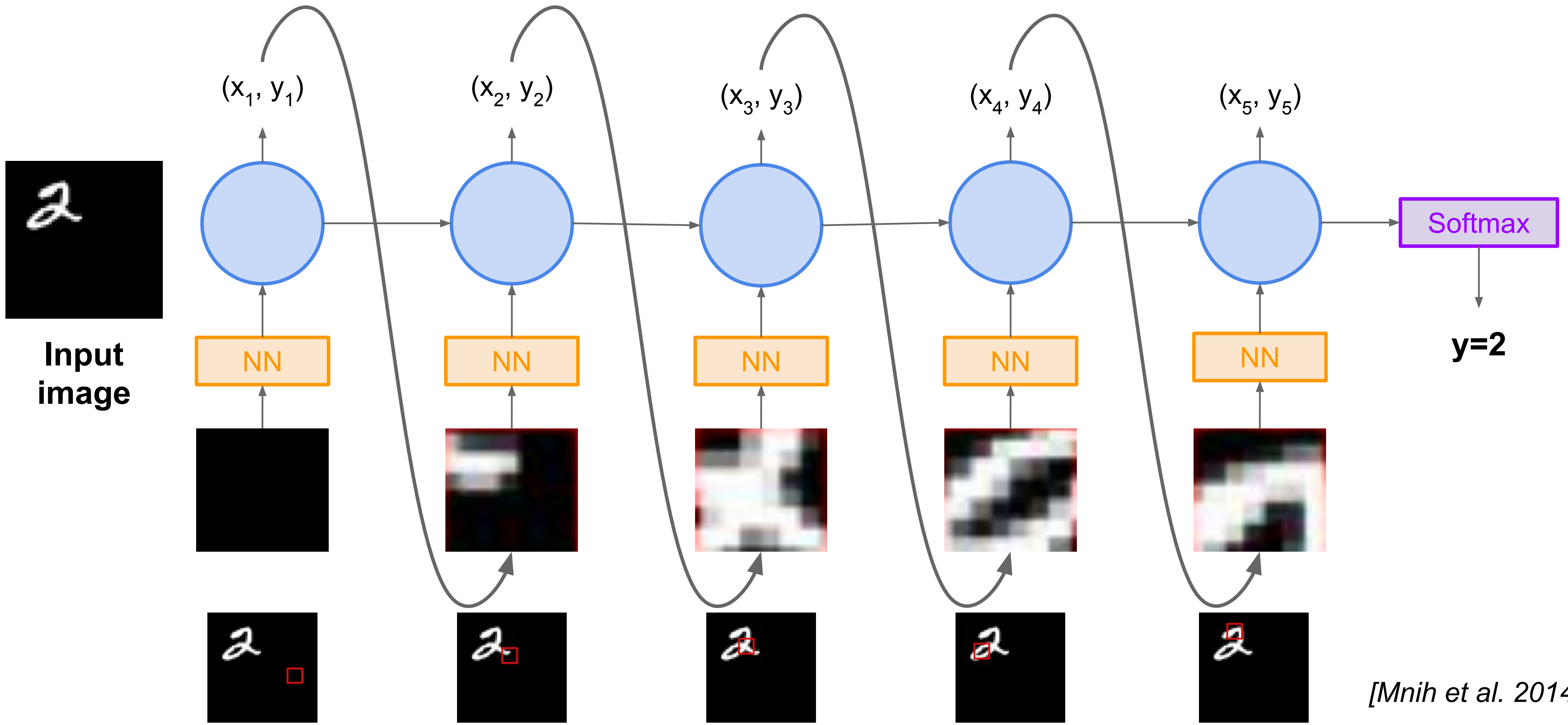
State: Glimpses seen so far
Action: (x,y) coordinates (center of glimpse) of where to look next in image
Reward: 1 at the final timestep if image correctly classified, 0 otherwise



Glimpsing is a non-differentiable operation => learn policy for how to take glimpse actions using REINFORCE
Given state of glimpses seen so far, use RNN to model the state and output next action

[Mnih et al. 2014]

REINFORCE in action: Recurrent Attention Model (RAM)



[Mnih et al. 2014]

Pytorch Implementation

- https://github.com/kevinzakka/recurrent-visual-attention
- A Pytorch implementation for the paper, Recurrent Models of Visual Attention by *Volodymyr Mnih, Nicolas Heess, Alex Graves and Koray Kavukcuoglu*, NIPS 2014.

Reinforcement Learning for Quantitative Trading

FinRL: A deep reinforcement learning library for automated stock trading in quantitative finance, Liu et al. Deep RL Workshop, NeurIPS 2020.

https://github.com/AI4Finance-Foundation/FinRL

FinRL: A Deep Reinforcement Learning Library for Automated Trading in Quantitative Finance

Xiao-Yang Liu[*+], Bruce Yang[*+], Zihan Ding[*$], Christina Dan Wang[*#], Anwar Walid[*+]
[*]AI4Finance LLC., [+]Columbia University, [$]Princeton University, [#]New York University

https://github.com/AI4Finance-LLC/FinRL-Library

Why RL for Trading?

1. Modern Portfolio Theory (MPT) performs not well in out-of-sample data, sensitive to outliers and only based on stock returns.
2. DRL doesn't need large labeled training datasets. It uses a reward function to optimize future return.
3. Goal of stock trading: maximize returns.
   DRL solves optimization problems by maximizing the expected total reward

Trading Markov Decision Process

- Trading agent is modeled as a Markov Decision Process (MDP)
- Note that this Markov process might not be stationary or static
- Components:
  - **State**
    - $\boldsymbol{s}$ = [$\boldsymbol{p}$, $\boldsymbol{h}$, b], $\boldsymbol{p}$: stock prices, $\boldsymbol{h}$: stock shares, b: remaining balance
  - **Action**
    - Three actions: **a** ∈ {−1, 0, 1}, where −1, 0, 1 represent selling, holding, and buying one stock.
    - Multiple action space **a** ∈ {−k, ..., −1, 0, 1, ..., k}, where k denotes the number of shares.
    - An action can be carried upon multiple shares. For example, "Buy 10 shares of AAPL" or "Sell 10 shares of AAPL" are 10 or −10, respectively. Resulting in $(2k+1)^d$ actions for d stocks.
  - **Reward**
    - $r(s,a,s')$: the direct reward of acting a at state s and arriving at the new state s', e.g. the change of the portfolio value when action a is taken at state s and arriving at new state s', i.e., r(s, a, s') = v' − v, where v' and v represent the portfolio values at state s' and s, respectively'.
  - Q-value function
    - Q_π (s, a): the expected reward of acting a at state s following policy π

State Space

- State Space
  - **Balance**: available amount of money left in the account currently
  - **Price**: current adjusted close price of each stock
  - **Shares**: shares owned of each stock
  - **ADX**: *Average Directional Index*, is a trend strength indicator.
  - **MACD**: *Moving Average Convergence Divergence,* is a trend-following momentum indicator that shows the relationship between two moving averages of a security's price. The MACD is calculated by subtracting the 26-period exponential moving average (EMA) from the 12-period EMA.
  - **RSI**: *Relative Strength Index*, is classified as a momentum oscillator, measuring the velocity and magnitude of directional price movements
  - **CCI**: *Commodity Channel Index,* is a momentum-based oscillator used to help determine when an investment vehicle is reaching a condition of being overbought or oversold.
  - One could use language models such as LSTM to extract more features.

Action space

- **Action**
  - Three actions: $a \in \{-1, 0, 1\}$, where −1, 0, 1 represent selling, holding, and buying one stock.
  - Multiple action space $a \in \{-k, ..., -1, 0, 1, ..., k\}$, where k denotes the number of shares one can buy or sell.
  - An action can be carried upon multiple stocks. Therefore the size of the enire action space is $(2k+1)^d$ where d is the number of stocks.
  - For example, "Buy 10 shares of AAPL" or "Sell 10 shares of AAPL" are a=10 or a=−10, respectively.

Reward function

- **Reward**
  - $r(s,a,s')$: the direct reward of acting a at state s and arriving at the new state s'
  - For example, the change of the portfolio value when action a is taken at state s and arriving at new state s', i.e., r(s, a, s') = v′ − v, where v′ and v represent the portfolio values at state s′ and s, respectively′
  - Transaction cost is usually involved
  - One can also use *Sharpe ratio* as reward,

The Formula for Sharpe Ratio Is

$$\text{Sharpe Ratio} = \frac{R_p - R_f}{\sigma_p}$$

where:

R_p = return of portfolio

R_f = risk-free rate

σ_p = standard deviation of the portfolio's excess return

Constraints

- **Market liquidity:**
  - Assume that stock market will not be affected by our reinforcement trading agent
- **Nonnegative balance:**
  - the allowed actions should not result in a negative balance.
- **Transaction cost:**
  - transaction costs are incurred for each trade.
- **Risk-aversion for market crash:**
  - employ the financial **turbulence index** that measures extreme asset price movements.

Learning Algorithms

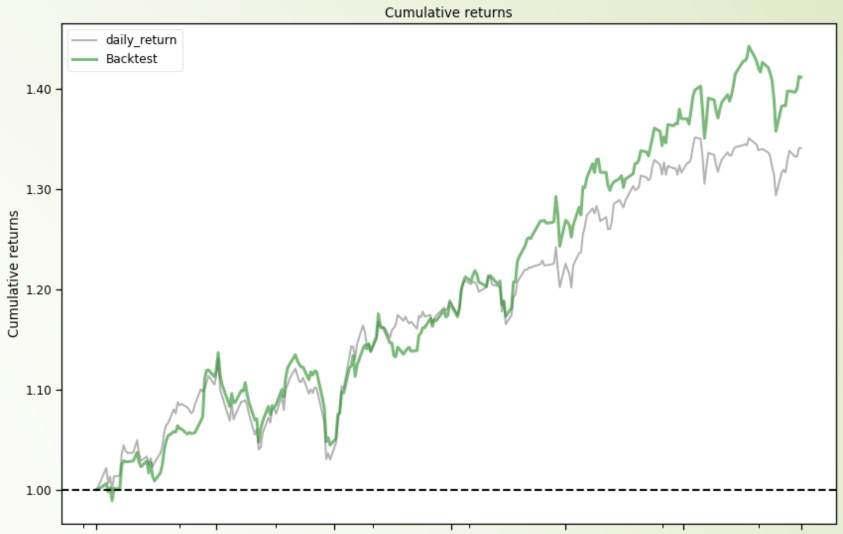
- Critic-only approach
  - Q-learning, DQN, etc
- Actor-only approach
  - Policy Gradient
- Actor-critic approach
  - A2C
  - PPO
  - DDPG
  - **SAC**

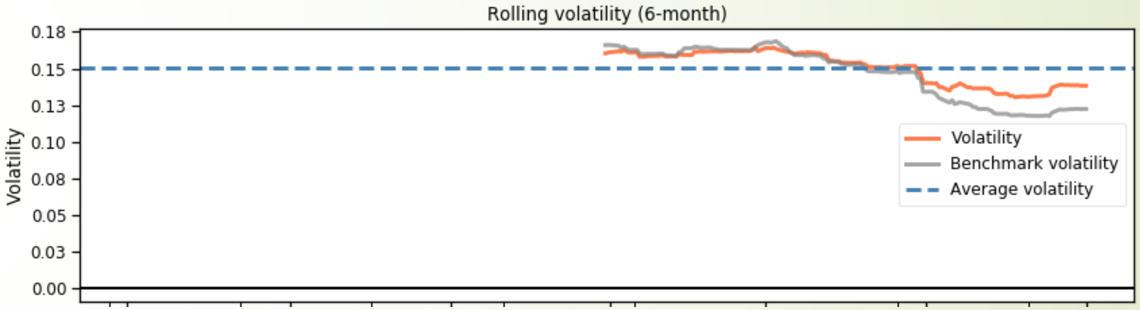
Data

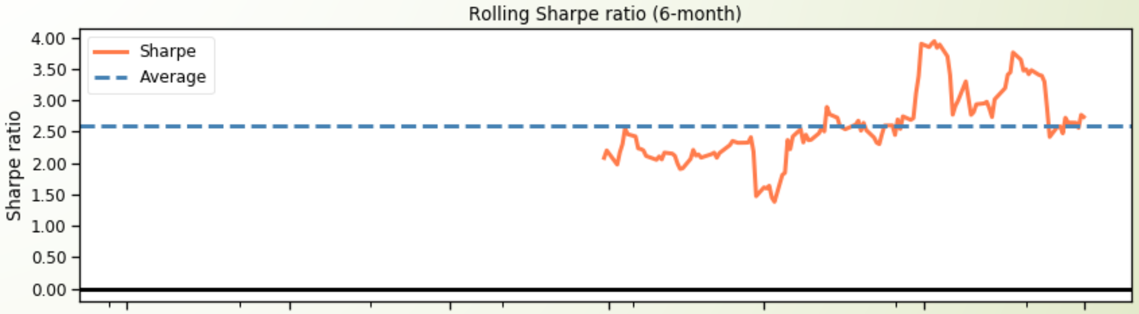
- Dow 30 constituents:
  - ['AXP', 'AMGN', 'AAPL', 'BA', 'CAT', 'CSCO', 'CVX', 'GS', 'HD', 'HON', 'IBM', 'INTC', 'JNJ', 'KO', 'JPM', 'MCD', 'MMM', 'MRK', 'MSFT', 'NKE', 'PG', 'TRV', 'UNH', 'CRM', 'VZ', 'V', 'WBA', 'WMT', 'DIS', 'DOW']
- Training
  - Daily OHLC prices and features from '2009-01-01' to '2020-07-01'
  - N = 83897
- BackTest trading
  - Daily OHLC prices and features from '2020-07-01' to '2021-07-06'
  - N = 7337
  - Baseline: Dow Jones Index (DJI)

A successful SAC agent

- SAC:
  - Annual return 0.409532
  - Cumulative returns 0.411453
  - Annual volatility 0.149417
  - Sharpe ratio 2.382402
- Baseline: DJI
  - Annual return 0.335107
  - Cumulative returns 0.336639
  - Annual volatility 0.145596
  - Sharpe ratio 2.066650







RL may be highly instable: two SAC runs

Good

- Results:
  - Annual return 0.409532
  - Cumulative returns 0.411453
  - Annual volatility 0.149417
  - Sharpe ratio 2.382402

Bad

- Results
  - Annual return 0.250596
  - Cumulative returns 0.251707
  - Annual volatility 0.148737
  - Sharpe ratio 1.584268

Summary

- Model-free reinforcement learning trading
- RL agent is unstable:
  - The reward is highly noisy
  - The environment in stock prices is not stationary
  - RL itself is not stable
  - Perhaps consider multiple agents

Optimized Execution, Market Microstructure and Reinforcement Learning

[Y. Nevmyvaka. Y. Feng, MK; ICML 2006]
[MK, Y. Nevmyvaka; In "High Frequency Trading", O'Hara et al. eds, Risk Books 2013]

Michael Kearns, University of Pennsylvania, ICML 2014, Beijing

A Brief Field Guide to Wall Street

- "Buy Side": Attempt to outperform market via proprietary research
  - Includes hedge funds, mutual funds, statistical arbitrage, HFT, prop trading groups
  - May or may not be quantitative and automated
  - Have investors but not clients
  - Take and hold positions → risk
  - Generation of "alpha" still more art than science
- "Sell Side": Provide brokerage and execution services
  - Includes bank and independent brokerages, exchanges
  - Almost entirely quantitative and automated
  - Clients are the buy side
  - Do not hold risk; paid via fees/commissions/etc.
- In reality, alpha and execution are blurred
  - Especially at shorter holding periods (e.g. HFT)

A Canonical Trading Problem

- Goal (buy side to sell side): Sell V shares in T time steps; maximize revenue
- Strategy Evaluation Metric Benchmarks:
  - Volume Weighted Average Price (VWAP)
  - Time Weighted Average Price (TWAP)
  - Implementation Shortfall (midpoint of bid-ask spread at beginning)
- Natural to view as a problem of *state-based control (RL)*
  - State variables: inventory V and time remaining T (discretized)
  - Features capturing market activity?

Market Microstructure


refresh | island home | disclaimer | help

GET STOCK: MSFT go — Symbol Search

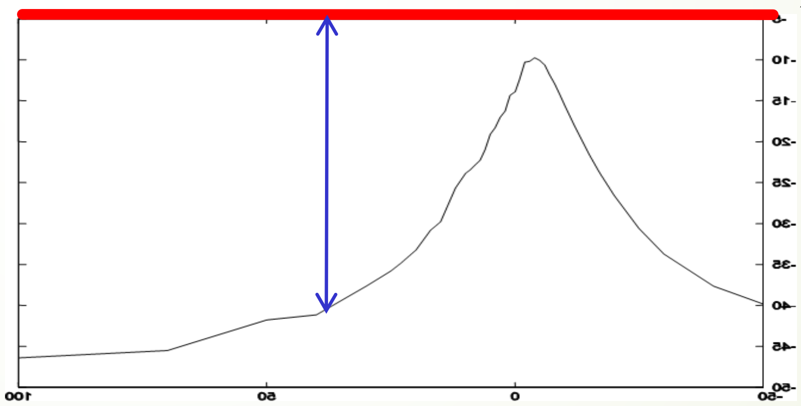
MSFT

LAST MATCH		TODAY'S ACTIVITY	
Price	23.7790	Orders	1,630
Time	9:01:55.614	Volume	44,839

BUY ORDERS		SELL ORDERS	
SHARES	PRICE	SHARES	PRICE
1,000	23.7600	100	23.7800
3,087	23.7500	800	23.7990
200	23.7500	500	23.8000
100	23.7400	1,720	23.8070
1,720	23.7280	900	23.8190
2,000	23.7200	200	23.8500
1,000	23.7000	1,000	23.8500
100	23.7000	1,000	23.8500
100	23.7000	1,000	23.8600
800	23.6970	200	24.0000
500	23.6500	500	24.0000
3,000	23.6500	1,000	24.0300
4,300	23.6500	200	24.0300
2,000	23.6500	1,100	24.0400
200	23.6200	500	24.0500
(195 more)		(219 more)	

- Continuous double auction with limit orders: buy orders decreasing; sell orders increasing
- Volatile and dynamic; sub-millisecond time scale
- Cancellations, revisions, partial executions
- How do individual orders (micro) influence aggregate market behavior (macro)?
- Tradeoff between *immediacy* and *price*
- Seen in "submit and leave" strategies:

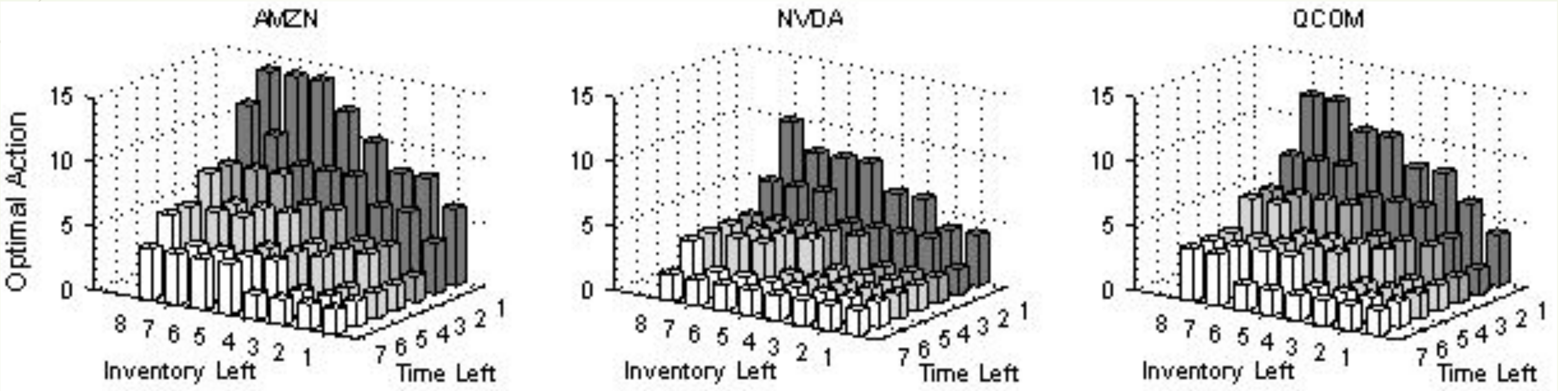
initial midpoint



Implementation Shortfall vs. Limit Price

deep in book market order

Policies Learned: Time and Volume Remaining



- Experimental framework
  - Full historical order book reconstruction and simulation
  - Learn optimal policy on 1 year training; test on following 6 months
  - Pitfalls: directional drift, “counterfactual” market impact
- Overall shape is consistent and sensible
  - Become more aggressive (spread crossing) as time runs out or inventory is too large
  - Learning optimizes this qualitative schedule

Additional Improvement From Order Book Features

Bid Volume	-0.06%	Ask Volume	-0.28%
Bid-Ask Volume Misbalance	0.13%	**Bid-Ask Spread**	**7.97%**
Price Level	0.26%	**Immediate Market Order Cost**	**4.26%**
Signed Transaction Volume	**2.81%**	Price Volatility	-0.55%
Spread Volatility	1.89%	Signed Incoming Volume	0.59%
Spread + Immediate Cost	**8.69%**	**Spread+ImmCost+Signed Vol**	**12.85%**

Thank you!