An Introduction to Reinforcement Learning

Yuan YAO
HKUST
Based on Feifei Li, Mengdi Wang, et al.
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.
Unsupervised Learning

- **Data**: \( x \)
  Just input data, no output labels!

- **Goal**: Learn some underlying hidden structure of the data

- **Examples**:
  - Clustering,
  - dimensionality reduction (manifold learning),
  - feature learning,
  - density estimation,
  - Generating samples, etc.

**Generative Models**
Given training data, generate new samples from same distribution

Training data \( \sim p_{\text{data}}(x) \)

Generated samples \( \sim p_{\text{model}}(x) \)

Want to learn \( p_{\text{model}}(x) \) similar to \( p_{\text{data}}(x) \)
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
Playing games against human champions

Deep Blue in 1997

AlphaGo “LEE” 2016

Deep Blue in 1997

Outline

- What is Reinforcement Learning?
- Markov Decision Processes
- Bellman Equation as Linear Programming
- Q-Learning
- Policy Gradients
Agent

Environment
State $s_i$ → Agent → Environment
Reinforcement Learning

Agent

State $s_t$

Environment

Action $a_t$
Agent

Environment

State $s_t$

Reward $r_t$

Action $a_t$
Reinforcement Learning

Agent

State \( s_t \)

Reward \( r_t \)

Next state \( s_{t+1} \)

Environment

Action \( a_t \)
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
**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

- **Markov property**: Current state completely characterizes the state of the world

Defined by: \((S, A, R, P, \gamma)\)

- \(S\) : set of possible states
- \(A\) : set of possible actions
- \(R\) : distribution of reward given (state, action) pair
- \(P\) : transition probability i.e. distribution over next state given (state, action) pair
- \(\gamma\) : discount factor
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 $r_t \sim R(\cdot | s_t, a_t)$
- Environment samples next state $s_{t+1} \sim P(\cdot | s_t; a_t)$
- Agent receives reward $r_t$ and next state $s_{t+1}$

A policy $\pi$ is a function from $S$ to $A$ that specifies what action to take in each state

**Objective**: find policy that maximizes the cumulated discounted reward
A simple MDP: Grid World

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

actions = {
  1. right
  2. left
  3. up
  4. down
}

Set a negative “reward” for each transition (e.g. $r = -1$)
A simple MDP: Grid World

Random Policy

Optimal Policy
The optimal policy $\pi^*$

We want to find optimal policy $\pi^*$ 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 $s_0 \sim p(s_0), a_t \sim \pi(\cdot | s_t), s_{t+1} \sim p(\cdot | s_t, a_t)$
Definitions: Value function and Q-value function

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

**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 $V^* : S \rightarrow R = x^*$ satisfied the following nonlinear fixed point equation

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

where a policy $\pi^*$ 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**
The Bellman equation is equivalent to

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

- 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 = $|\mathcal{S}| \times |\mathcal{A}|$. 

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

$$\max_{\lambda_a} \lambda_a^T r_a, \quad a \in A$$

subject to

$$\sum_{a \in A} (I - \gamma P_a^T) \lambda_a = e, \quad \lambda_a \geq 0, \quad a \in A$$

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

**Theorem**

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

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

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

*Finding the optimal policy $\mu^* = Finding the basis of the dual solution $\lambda^*$*
Online Value-Policy Iteration

Stochastic primal-dual (value-policy) algorithm

- **Input:** Simulation Oracle $\mathcal{M}$, $n = |S|$, $m = |A|$, $\alpha \in (0, 1)$.
- Initialize $x^{(0)}$ and $\lambda = (\lambda_u^{(0)} : u \in A)$ arbitrarily.
- For $k = 1, 2, \ldots, T$
  - Sample $i_k$ uniformly from $S$ and sample $u_k$ uniformly from $A$.
  - 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
    
    $$
    x^{(k-\frac{1}{2})} = x^{(k-1)} - \gamma_k \left( -\epsilon + m\lambda_u^{(k-1)} - \alpha mn \left( \lambda_{u_k}^{(k-1)} \cdot e_{i_k} \right) e_{j_k} \right),
    $$
    $$
    \lambda_{u_k}^{(k-\frac{1}{2})} = \lambda_{u_k}^{(k-1)} + m\gamma_k \left( x^{(k-1)} - \alpha n \left( x^{(k-1)} \cdot e_{j_k} \right) e_{i_k} - n g_{i_k,j_k} u_k e_{i_k} \right),
    $$
    $$
    \lambda_u^{(k-\frac{1}{2})} = \lambda_u^{(k-1)}, \quad \forall \ u \neq u_k,
    $$
  - Project the iterates orthogonally to some regularization constraints
    $$
    x^{(k)} = \Pi_x x^{(k-\frac{1}{2})}, \quad \lambda^{(k)} = \Pi_{\lambda} \lambda^{(k-\frac{1}{2})}.
    $$
- **Output:** Averaged dual iterate $\hat{\lambda} = \frac{1}{T} \sum_{k=1}^T \lambda^{(k)}$
Near Optimal Primal-Dual Algorithms

<table>
<thead>
<tr>
<th>Method</th>
<th>Setting</th>
<th>Sample Complexity</th>
<th>Run-Time Complexity</th>
<th>Space Complexity</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phased Q-Learning</td>
<td>$\gamma$ discount factor, $\epsilon$-optimal value</td>
<td>$\frac{</td>
<td>S</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>Model-Based Q-Learning</td>
<td>$\gamma$ discount factor, $\epsilon$-optimal value</td>
<td>$\frac{</td>
<td>S</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>Randomized P-D</td>
<td>$\gamma$ discount factor, $\epsilon$-optimal policy</td>
<td>$\frac{</td>
<td>S</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>Randomized P-D</td>
<td>$\gamma$ discount factor, $\tau$-stationary, $\epsilon$-optimal policy</td>
<td>$\tau^4 \frac{</td>
<td>S</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>Randomized VI</td>
<td>$\gamma$ discount factor, $\epsilon$-optimal policy</td>
<td>$\frac{</td>
<td>S</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>Primal-Dual $\pi$ Learning</td>
<td>$\tau$-stationary, $t^{\text{mix}}$-mixing, $\epsilon$-optimal policy</td>
<td>$\frac{(\tau \cdot t^{* \text{mix}})^2</td>
<td>S</td>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>

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

Mengdi Wang, Primal-Dual $\pi$ Learning, arXiv:1710.0610
Q-Learning

Bellman equation

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

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

$Q^*$ satisfies the following Bellman equation:

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

**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 $\pi^*$ 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 \( i \to \infty \)
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 =&gt; 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 \( \theta \)):

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

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

[Mnih et al. NIPS Workshop 2013; Nature 2015]
Q-network Architecture

\[ Q(s, a; \theta) : \]
neural network with weights \( \theta \)

A single feedforward pass to compute Q-values for all actions from the current state \( \Rightarrow \) efficient!

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

[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

[Mnih et al. NIPS Workshop 2013; Nature 2015]
Training the Q-network: Loss function (from before)

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 \( \theta \)):

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

- 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

- Address these problems using experience replay
  - Continually update a replay memory table of transitions \( (s_t, a_t, r_t, s_{t+1}) \) as game (experience) episodes are played
  - Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples

Each transition can also contribute to multiple weight updates => greater data efficiency
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 $\epsilon$ 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
Example

- https://www.youtube.com/watch?v=V1eYniJ0Rnk
Policy Gradients

- 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

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:

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

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

\[ J(\theta) = \mathbb{E}_{\tau \sim p(\tau; \theta)} [r(\tau)] \]

\[ = \int_{\tau} r(\tau)p(\tau; \theta) d\tau \]

Now let’s differentiate this:  
\[ \nabla_{\theta} J(\theta) = \int_{\tau} r(\tau) \nabla_{\theta} p(\tau; \theta) d\tau \]

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:

\[ \nabla_{\theta} J(\theta) = \int_{\tau} (r(\tau) \nabla_{\theta} \log p(\tau; \theta)) p(\tau; \theta) d\tau \]

\[ = \mathbb{E}_{\tau \sim p(\tau; \theta)} [r(\tau) \nabla_{\theta} \log p(\tau; \theta)] \]

Intractable! Gradient of an expectation is problematic when \( p \) depends on \( \theta \)

Can estimate with
Monte Carlo sampling
REINFORCE algorithm

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

Can we compute those quantities without knowing the transition probabilities?

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

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: 
\[ \nabla_\theta \log p(\tau; \theta) = \sum_{t \geq 0} \nabla_\theta \log \pi_\theta(a_t|s_t) \]

Therefore when sampling a trajectory \( \tau \), we can estimate \( J(\theta) \) with

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

Doesn't depend on transition probabilities!
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 \( r(\tau) \) is high, push up the probabilities of the actions seen
- If \( r(\tau) \) 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 $\gamma$ 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: $\nabla_\theta J(\theta) \approx \sum_{t \geq 0} (Q^\pi(s_t, a_t) - V^\pi(s_t)) \nabla_\theta \log \pi_\theta(a_t|s_t)$
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 Algorithm

Initialize policy parameters $\theta$, critic parameters $\phi$

For iteration=1, 2 … do

Sample m trajectories under the current policy

$\Delta \theta \leftarrow 0$

For i=1, …, m do

For t=1, … , T do

$$A_t = \sum_{t' > t} \gamma^{t' - t} r_t^i - V_\phi(s_t^i)$$

$\Delta \theta \leftarrow \Delta \theta + A_t \nabla_\theta \log(p_t^i|s_t^i)$

$\Delta \phi \leftarrow \sum_i \sum_t \nabla_\phi ||A_t^i||^2$

$\theta \leftarrow \alpha \Delta \theta$

$\phi \leftarrow \beta \Delta \phi$

End for
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)

\[
\begin{align*}
(x_1, y_1) & \quad \text{NN} & \quad (x_2, y_2) & \quad \text{NN} & \quad (x_3, y_3) & \quad \text{NN} & \quad (x_4, y_4) & \quad \text{NN} & \quad (x_5, y_5) & \quad \text{Softmax} \\
\end{align*}
\]

Input image

\[\text{y}=2\]

[Mnih et al. 2014]
Pytorch Implementation

- [https://github.com/kevinzakka/recurrent-visual-attention](https://github.com/kevinzakka/recurrent-visual-attention)
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 policy and value networks in a Monte Carlo Tree Search algorithm to select actions by lookahead search

[Silver et al., Nature 2016]
Summary

- **Policy gradients**: very general but suffer from high variance so requires a lot of samples. **Challenge**: sample-efficiency

- **Q-learning**: does not always work but when it works, usually more sample-efficient. **Challenge**: exploration

- **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 approximator
Thank you!