CS 4803 / 7643: Deep Learning

Topics:
  – Continue Deep Reinforcement Learning
    – Q-Learning
    – Policy gradient methods

Zsolt Kira
Georgia Tech
Administrativia

• Projects!

• Project Check-in due **April 11\textsuperscript{th}**
  – Will be graded pass/fail, if fail then you can address the issues
  – Counts for 5 points of project score

• Poster due date moved to **April 23\textsuperscript{rd}** (last day of class)
  – No presentations

• Final submission due date **April 30\textsuperscript{th}**
Administrativia

- Piazza me requests for topics after reinforcement learning
Types of Learning

• Supervised learning
  – Learning from a “teacher”
  – Training data includes desired outputs

• Unsupervised learning
  – Discover structure in data
  – Training data does not include desired outputs

• Reinforcement learning
  – Learning to act under evaluative feedback (rewards)
**RL API**

- At each step $t$ the agent:
  - Executes action $a_t$
  - Receives observation $o_t$
  - Receives scalar reward $r_t$

- The environment:
  - Receives action $a_t$
  - Emits observation $o_{t+1}$
  - Emits scalar reward $r_{t+1}$
Summary of Last time - MDPs

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

- Life is trajectory: \(\ldots S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, R_{t+2}, S_{t+2}, \ldots\)

**Policy:**

\[
\pi^* = \arg \max_{\pi} \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t | \pi \right]
\]

Deterministic policy: \(a = \pi(s)\)

Stochastic policy: \(\pi(a|s) = P[A_t = a|S_t = s]\)

**Value:**

\[
V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right]
\]

Bellman Equation

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Summary of Last time – Solving MDPs

- Known transition and reward functions (planning)
  - (Q-)Value iteration – just iteratively update based on Bellman Equation
    - Q-values easier to extract policy from (argmax)

- Unknown transition and reward functions
  - Model-based – Learn $T$ and $R$ and then do planning
  - Model-free
    - Take sample trajectories (policy not controlled by us)
    - Estimate expected values (the world samples from true distribution)
  - Direct Evaluation: Just average over all samples
  - Temporal Difference (TD): Incremental update per example
    \[
    \text{sample} = R(s, \pi(s), s') + \gamma V^\pi(s')
    \]
    \[
    V^\pi(s) \leftarrow V^\pi(s) + \alpha(\text{sample} - V^\pi(s))
    \]

- Function approximation: Estimate Q or V using a function, e.g. neural network

- Today:
  - Policy iteration – Learn a policy directly without values
  - Actor-Critic – Estimate both value and policy function!

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Q-Networks

\[ Q(s, a, w) \approx Q^*(s, a) \]
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

Figures copyright Volodymyr Mnih et al., 2013. Reproduced with permission.
Q-network Architecture

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

- FC-4 (Q-values)
- FC-256
- 32 4x4 conv, stride 2
- 16 8x8 conv, stride 4

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]

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
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)

Input: state \( s_t \)

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

[<i>Mnih et al. NIPS Workshop 2013; Nature 2015</i>]
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)

Familiar conv layers, FC layer

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

[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)

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

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

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

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
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 => efficient!

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

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

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

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Deep Q-learning

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

\[ Q^*(s, a) = \mathbb{E}[r + \gamma \max_{a'} Q^*(s', a') \mid s, a] \]
Deep Q-learning

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

\[ Q^*(s, a) = \mathbb{E}[r + \gamma \max_{a'} Q^*(s', a') \mid s, a] \]

Forward Pass
Loss function: \( L_i(\theta_i) = \mathbb{E} \left[ (y_i - Q(s, a; \theta_i))^2 \right] \)
Deep Q-learning

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

\[ Q^*(s, a) = \mathbb{E}[r + \gamma \max_{a'} Q^*(s', a') | s, a] \]

**Forward Pass**

Loss function:

\[ L_i(\theta_i) = \mathbb{E} [(y_i - Q(s, a; \theta_i))^2] \]

where

\[ y_i = \mathbb{E}[r + \gamma \max_{a'} Q^*(s', a') | s, a] \]
Deep Q-learning

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

\[ Q^*(s, a) = \mathbb{E}[r + \gamma \max_{a'} Q^*(s', a') | s, a] \]

Forward Pass
Loss function:

\[ L_i(\theta_i) = \mathbb{E} \left[ (y_i - Q(s, a; \theta_i))^2 \right] \]

where

\[ y_i = \mathbb{E}[r + \gamma \max_{a'} Q^*(s', a') | s, a] \]

Backward Pass
Gradient update (with respect to Q-function parameters \( \theta \)):

\[ \nabla_{\theta_i} L_i(\theta_i) = \mathbb{E} \left[ r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right] \nabla_{\theta_i} Q(s, a; \theta_i) \]
Deep Q-learning

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

$$Q^*(s, a) = \mathbb{E}[r + \gamma \max_{a'} Q^*(s', a') \mid s, a]$$

**Forward Pass**

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

where  
$$y_i = \mathbb{E}[r + \gamma \max_{a'} Q^*(s', a') \mid s, a]$$

**Backward Pass**

Gradient update (with respect to Q-function parameters $\theta$):

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

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Deep Q-Networks

DQN provides a stable solution to deep value-based RL

1. Use experience replay
   ▶ Break correlations in data, bring us back to iid setting
   ▶ Learn from all past policies

2. Freeze target Q-network
   ▶ Avoid oscillations
   ▶ Break correlations between Q-network and target

3. Clip rewards or normalize network adaptively to sensible range
   ▶ Robust gradients
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
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 side) => 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
Experience Replay

To remove correlations, build data-set from agent’s own experience

\[
\begin{array}{cccc}
    s_1, a_1, r_2, s_2 \\
    s_2, a_2, r_3, s_3 \\
    s_3, a_3, r_4, s_4 \\
    \vdots \\
    s_t, a_t, r_{t+1}, s_{t+1}
\end{array}
\rightarrow
\begin{array}{cccc}
    s, a, r, s' \\
    s_t, a_t, r_{t+1}, s_{t+1}
\end{array}
\]
Stable Deep RL (2): Fixed Target Q-Network

To avoid oscillations, fix parameters used in Q-learning target

- Compute Q-learning targets w.r.t. old, fixed parameters $w^-$
  
  $$r + \gamma \max_{a'} Q(s', a', w^-)$$

- Optimise MSE between Q-network and Q-learning targets

  $$\mathcal{L}(w) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}} \left[ \left( r + \gamma \max_{a'} Q(s', a', w^-) - Q(s, a, w) \right)^2 \right]$$

- Periodically update fixed parameters $w^- \leftarrow w$
Stable Deep RL (3): Reward/Value Range

- DQN clips the rewards to $[-1, +1]$
- This prevents Q-values from becoming too large
- Ensures gradients are well-conditioned
- Can’t tell difference between small and large rewards
Putting it together: Deep Q-Learning with Experience Replay

**Algorithm 1 Deep Q-learning with Experience Replay**

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

Algorithm 1 Deep Q-learning with Experience Replay

1. Initialize replay memory $\mathcal{D}$ to capacity $N$
2. Initialize action-value function $Q$ with random weights

for episode = 1, $M$ do

1. Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ do

1. With probability $\epsilon$ select a random action $a_t$
2. otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
3. Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
4. Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
5. Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
6. Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$

1. Set $y_j = r_j$ for terminal $\phi_{j+1}$
2. $y_j = r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta)$ for non-terminal $\phi_{j+1}$
3. Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

end for

end for

[O investigation]

Mnih et al. NIPS Workshop 2013; Nature 2015

(C) Dhruv Batra & Zsolt Kira
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
```

Play $M$ episodes (full games)
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

Initialize state (starting game screen pixels) at the beginning of each episode
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

For each timestep $t$ of the game
Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $D$ 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 $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        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

With small probability, select a random action (explore), otherwise select greedy action from current policy
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

Take the action $(a_t)$, and observe the reward $r_t$ and next state $s_{t+1}$
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**

Store transition in replay memory
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
Video by Károly Zsolnai-Fehér. Reproduced with permission.

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

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
DQN Results in Atari
Deep RL

- **Value-based RL**
  - Use neural nets to represent Q function \( Q(s, a; \theta) \)
  
  \[ Q(s, a; \theta^*) \approx Q^*(s, a) \]

- **Policy-based RL**
  - Use neural nets to represent policy \( \pi_\theta \)
  
  \[ \pi_\theta^* \approx \pi^* \]

- **Model**
  - Use neural nets to represent and learn the model
Formally, let’s define a class of parameterized policies: $\Pi = \{\pi_\theta, \theta \in \mathbb{R}^m\}$

For each policy, define its value:

$$J(\theta) = \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t |\pi_\theta \right]$$
Formally, let’s define a class of parameterized 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?
Formally, let’s define a class of parameterized 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 \( \tau = (s_0, a_0, r_0, s_1, \ldots) \)
REINFORCE algorithm

Expected reward:

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

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

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:

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

Intractable! Expectation of gradient is problematic when \( p \) depends on \( \theta \)
REINFORCE algorithm

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:
\[ \nabla_{\theta} p(\tau; \theta) = p(\tau; \theta) \frac{\nabla_{\theta} p(\tau; \theta)}{p(\tau; \theta)} = p(\tau; \theta) \nabla_{\theta} \log p(\tau; \theta) \]

Intractable! Expectation of gradient is problematic when \( p \) depends on \( \theta \)
REINFORCE algorithm

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

Intractable! Expectation of gradient is problematic when p depends on \( \theta \)

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

Can estimate with Monte Carlo sampling

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
REINFORCE algorithm

Can we compute those quantities without knowing the transition probabilities?

We have: 

\[ p(\tau; \theta) = \prod_{t \geq 0} p(s_{t+1}|s_t, a_t) \pi\theta(a_t|s_t) \]
REINFORCE algorithm

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: 

\[ \log p(\tau; \theta) = \sum_{t \geq 0} \log p(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t) \]
REINFORCE algorithm

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!
The REINFORCE algorithm

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

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

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

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

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

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
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
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!**
Intuition
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: \[ \nabla_\theta J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_\theta \log \pi_\theta(a_t | s_t) \]
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

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

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

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
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
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”

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
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

[Mnih et al. 2014]
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)

\[ (x_t, y_t) \]

Input image

\[ \text{NN} \]

\[ [\text{Mnih et al. 2014}] \]

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
REINFORCE in action: Recurrent Attention Model (RAM)

[Mnih et al. 2014]
REINFORCE in action: Recurrent Attention Model (RAM)

[Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n]
REINFORCE in action: Recurrent Attention Model (RAM)
REINFORCE in action: Recurrent Attention Model (RAM)

[Image of a diagram showing the process of REINFORCE in action with an input image and multiple (x, y) coordinates leading to a softmax output with y=2.]

[Mnih et al. 2014]
REINFORCE in action: Recurrent Attention Model (RAM)

Has also been used in many other tasks including fine-grained image recognition, image captioning, and visual question-answering!

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?
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!
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.
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
      $A_t = \sum_{t' \geq t} \gamma^{t'-t} r_t^i - V_\phi(s_t^i)$
      $\Delta \theta \leftarrow \Delta \theta + A_t \nabla \theta \log(a_t^i | s_t^i)$
      $\Delta \phi \leftarrow \sum_i \sum_t \nabla \phi ||A_t^i||^2$
      $\theta \leftarrow \alpha \Delta \theta$
      $\phi \leftarrow \beta \Delta \phi$
  End for
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 of $J(\theta)$, often good enough!
  - **Q-learning**: Zero guarantees since you are approximating Bellman equation with a complicated function approximator
Playing Go

Rules

- Each player puts a stone on the goban, black first
- Each stone remains on the goban, except:

  - group w/o degree freedom is killed
  - a group with two eyes can’t be killed

  - The goal is to control the max. territory
Go is a Difficult Game

Features

- Size of the state space $2 \times 10^{170}$
- Size of the action space 200
- No good evaluation function
- Local and global features (symmetries, freedom, ...)
- A move might make a difference some dozen plies later
AlphaGo

• Go is a perfect information game
  – See entire board at all times
  – Has an optimal value function!

• Key idea: We cannot unroll search tree to learn a policy/value for a large number of states, instead:
  – Reduce depth of search via **position evaluation**: Replace subtrees with estimated value function $v(s)$
  – Reduce breadth of search via **action sampling**: Don’t perform unlikely actions
    • Start by predicting expert actions, gives you a probability distribution

• Use Monte Carlo rollouts, with a policy, selecting children with higher values
  – As policy improves this search improves too
Minimax Tree Search

From Wikipedia
Monte-Carlo Tree Search

Selection

Expansion

Simulation

Backpropagation

Rollout
(Random Search)

From Wikipedia
AlphaGo - Steps

1) Train supervised learning policy network from human moves

2) Train fast policy that can sample actions during rollouts

3) Train RL policy network that improves the SL policy network by optimizing the final outcome of games
   1) Use self-play

4) Train a value network that predicts the winner of games played by the RL policy network against itself
Supervised Learning Policy Network

- CNN to predict $p(a|s)$ trained on expert players
  - 13-layer deep
  - Accuracy 57% (55.7% using only board positions and move history)
  - Previous state of art was 44.4%

- Also train a fast ($2 \text{ us}!)$ **rollout policy network** with just linear softmax of pattern features

---

**Extended Data Table 4 | Input features for rollout and tree policy**

<table>
<thead>
<tr>
<th>Feature</th>
<th># of patterns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response</td>
<td>1</td>
<td>Whether move matches one or more response pattern features</td>
</tr>
<tr>
<td>Save atari</td>
<td>1</td>
<td>Move saves stone(s) from capture</td>
</tr>
<tr>
<td>Neighbour</td>
<td>8</td>
<td>Move is 8-connected to previous move</td>
</tr>
<tr>
<td>Nakade</td>
<td>8192</td>
<td>Move matches a <em>nakade</em> pattern at captured stone</td>
</tr>
<tr>
<td>Response pattern</td>
<td>32207</td>
<td>Move matches 12-point diamond pattern near previous move</td>
</tr>
<tr>
<td>Non-response pattern</td>
<td>69938</td>
<td>Move matches 3 x 3 pattern around move</td>
</tr>
<tr>
<td>Self-atari</td>
<td>1</td>
<td>Move allows stones to be captured</td>
</tr>
<tr>
<td>Last move distance</td>
<td>34</td>
<td>Manhattan distance to previous two moves</td>
</tr>
<tr>
<td>Non-response pattern</td>
<td>32207</td>
<td>Move matches 12-point diamond pattern centred around move</td>
</tr>
</tbody>
</table>

Features used by the rollout policy (first set) and tree policy (first and second set). Patterns are based on stone colour (black/white/empty) and liberties ($1, 2, \ldots, 3$) at each intersection of the pattern.

---

**Extended Data Table 2 | Input features for neural networks**

<table>
<thead>
<tr>
<th>Feature</th>
<th># of planes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stone colour</td>
<td>3</td>
<td>Player stone / opponent stone / empty</td>
</tr>
<tr>
<td>Ones</td>
<td>1</td>
<td>A constant plane filled with 1</td>
</tr>
<tr>
<td>Turns since</td>
<td>8</td>
<td>How many turns since a move was played</td>
</tr>
<tr>
<td>Liberties</td>
<td>8</td>
<td>Number of liberties (empty adjacent points)</td>
</tr>
<tr>
<td>Capture size</td>
<td>8</td>
<td>How many opponent stones would be captured</td>
</tr>
<tr>
<td>Self-atari size</td>
<td>8</td>
<td>How many of own stones would be captured</td>
</tr>
<tr>
<td>Liberties after move</td>
<td>8</td>
<td>Number of liberties after this move is played</td>
</tr>
<tr>
<td>Ladder capture</td>
<td>1</td>
<td>Whether a move at this point is a successful ladder capture</td>
</tr>
<tr>
<td>Ladder escape</td>
<td>1</td>
<td>Whether a move at this point is a successful ladder escape</td>
</tr>
<tr>
<td>Sensibleness</td>
<td>1</td>
<td>Whether a move is legal and does not fill its own eyes</td>
</tr>
<tr>
<td>Zeros</td>
<td>1</td>
<td>A constant plane filled with 0</td>
</tr>
<tr>
<td>Player color</td>
<td>1</td>
<td>Whether current player is black</td>
</tr>
</tbody>
</table>

Features used by the policy network (all but last feature) and value network (all features).
Reinforcement Learning Policy Network

• Initialized from supervised policy network
  – Play against policies sampled from previous iterations
  – Won 80% of games against supervised network
Value Network

- Use RL policy network self-play to train (30M positions)
Playing the Game

- Final play combines policy and value networks using MCTS
AlphaGo Zero

• MCTS with Self-Play
  – Don’t have to guess what opponent might do, so…
  – If no exploration, a big-branching game tree becomes one path
  – You get an automatically improving, evenly-matched opponent who is accurately learning your strategy
  – “We have met the enemy, and he is us” (famous variant of Pogo, 1954)
  – No need for human expert scoring rules for boards from unfinished games

• Treat board as an image: use residual convolutional neural network

• AlphaGo Zero: One deep neural network learns both the value function and policy in parallel

• Alpha Zero: Removed rollout altogether from MCTS and just used current neural net estimates instead
Domain Knowledge

1. The input features describing the position, and the output features describing the move, are structured as a set of planes; i.e. the neural network architecture is matched to the grid-structure of the board.

2. *AlphaZero* is provided with perfect knowledge of the game rules. These are used during MCTS, to simulate the positions resulting from a sequence of moves, to determine game termination, and to score any simulations that reach a terminal state.

3. Knowledge of the rules is also used to encode the input planes (i.e. castling, repetition, no-progress) and output planes (how pieces move, promotions, and piece drops in shogi).

4. The typical number of legal moves is used to scale the exploration noise (see below).

5. Chess and shogi games exceeding a maximum number of steps (determined by typical game length) were terminated and assigned a drawn outcome; Go games were terminated and scored with Tromp-Taylor rules, similarly to previous work (29).

*AlphaZero* did not use any form of domain knowledge beyond the points listed above.
AlphaGo Zero

It’s also more efficient than older engines!

D. Silver et al., Mastering the Game of Go without Human Knowledge, Nature 550, October 2017

https://deepmind.com/blog/alphago-zero-learning-scratch/