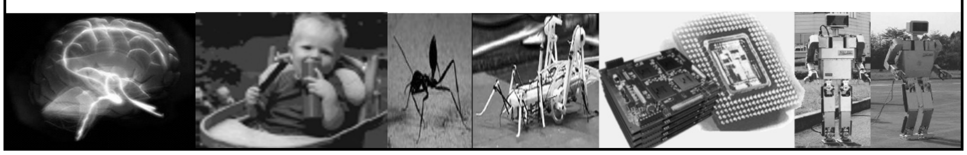
Neural Network and Reinforcement Learning

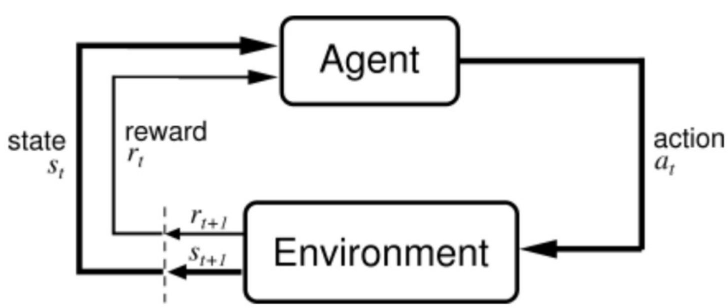
Prof. Xiangnan Zhong
Department of Electrical Engineering and Computer Science
Florida Atlantic University
Boca Raton, FL 33431



1

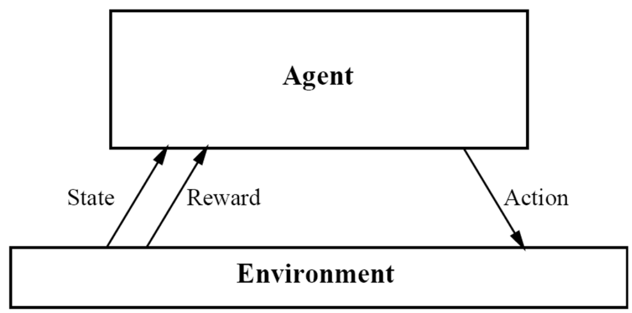
What is Reinforcement Learning

Learning how to make good decisions by interaction.



2

Reinforcement Learning problem



$$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \ldots$$

Goal: Learn to choose actions that maximize

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots \text{ , where } 0 \leq \gamma < 1$$

3

Q function

Define new function very similar to V^*

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$

If agent learns Q, it can choose optimal action even without knowing δ!

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}[r(s,a) + \gamma V^*(\delta(s,a))]$$

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a)$$

Q is the evaluation function the agent will learn

4

An algorithm for Learning *Q*

Note Q and V^* closely related:

$$V^*(s) = \max_{a'} Q(s, a')$$

Which allows us to write Q recursively as

$$\begin{aligned} Q(s_t, a_t) &= r(s_t, a_t) + \gamma V^*(\delta(s_t, a_t))) \\ &= r(s_t, a_t) + \gamma \max_{a'} Q(s_{t+1}, a') \end{aligned}$$

5

An algorithm for Learning *Q*

$$\begin{aligned} Q(s_t, a_t) &= r(s_t, a_t) + \gamma V^*(\delta(s_t, a_t))) \\ &= r(s_t, a_t) + \gamma \max_{a'} Q(s_{t+1}, a') \end{aligned}$$

Nice! Let $\hat{Q}$ denote learner's current approximation to Q. Consider training rule

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

where s' is the state resulting from applying action a in state s

6

Q Learning algorithm

For each s, a initialize table entry $\hat{Q}(s,a) \leftarrow 0$

Observe current state s

Do forever:

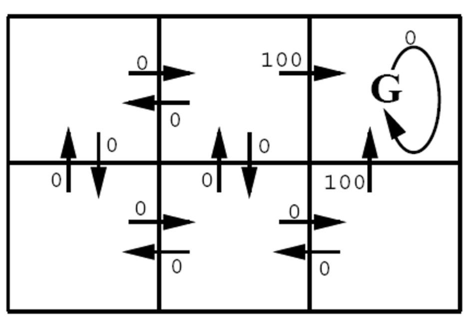
- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s,a)$ as follows:

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

- $s \leftarrow s'$

7

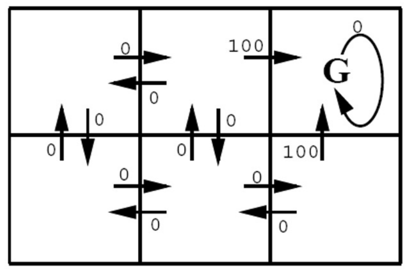
Let's look an example

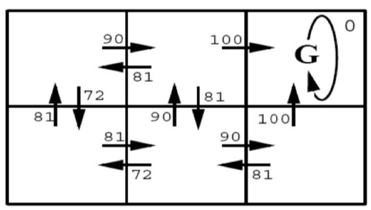


$r(s,a)$ (immediate reward) values

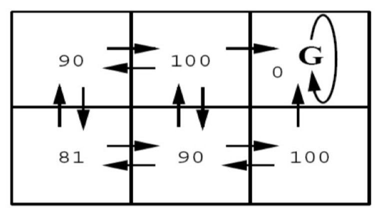
8

Now, let's revisit the problem again





$Q(s, a)$ values



$V^{*}(s)$ values

Learning from Experience

How to use trajectory data?

- Model based approach: estimate $T(x'|x, u)$, then use model to plan
- Model free:
  - Value based approach: estimate optimal value (or Q) function from data
  - Policy based approach: use data to determine how to improve policy
  - Actor Critic approach: learn both a policy and a value/Q function

Model-free, value based: Q Learning

Optimal Q function satisfies

$$Q^*(x,u) = R(x,u) + \gamma \sum_{x' \in \mathcal{X}} T(x'|x,u) \max_{u'} Q^*(x',u')$$

So, in expectation,

$$\mathrm{E}\left[\underbrace{Q^*(x_t,u_t) - \left(r_t + \gamma \max_{u'} Q^*(x_{t+1},u')\right)}_{\text{Temporal Difference (TD) error}}\right] = 0$$

11

Deep Q Learning

- Many possible function approximators for Q
  - Linear, nearest neighbors, aggregation
- Recent success: neural networks with loss function

$$\left(r_t + \gamma \max_{u} Q_{\theta'}(x_{t+1},u) - Q_\theta(x_t,u_t)\right)^2$$

- Deep Q Network (DQN; Mnih et al. 2013)
  - Experience replay

12

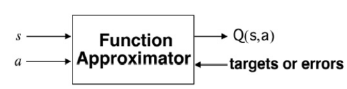
Think about the **Breakout** game

- State: screen pixels
  - Image size: $\mathbf{84} \times \mathbf{84}$ (resized)
  - Consecutive **4** images
  - Grayscale with **256** gray levels

$\Big\}$ $\mathbf{256^{84 \times 84 \times 4}}$ rows in the Q-table!

13

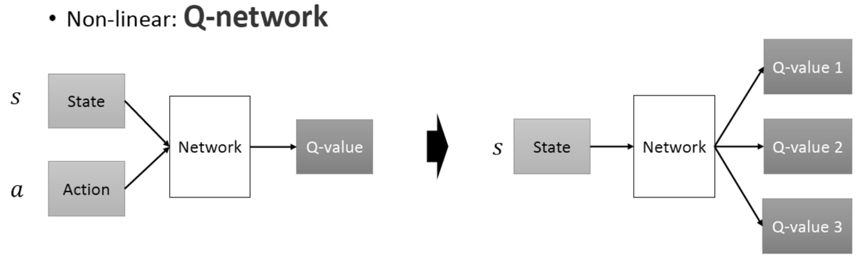
Function Approximator



- Use a function (with parameters) to approximate the Q-function

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

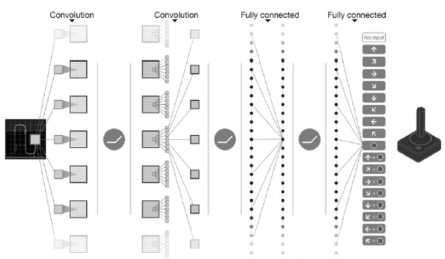
  - Linear
  - Non-linear: **Q-network**



14

Deep Q-Network

Deep Q-Network used in the DeepMind paper:



Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

Note: No Pooling Layer!

15

Estimate the Q-Network

- Objective Function
  - Recall the Bellman Equation: $Q(s,a) = r + \gamma max_{a'} Q(s',a')$
  - Here, we use simple squared error:

$$L = \mathbb{E}[(\underbrace{\boldsymbol{r + \gamma max_{a'} Q(s',a')}}_{\textbf{target}} - Q(s,a))^2]$$

  - Leading to the following Q-learning gradient

$$\frac{\partial L(w)}{\partial w} = \mathbb{E}[(\boldsymbol{r + \gamma max_{a'} Q(s',a')} - Q(s,a))\frac{\partial Q(s,a,w)}{\partial w}]$$

  - Optimize objective end-to-end by **SGD**

16

Learning Stability

- Non-linear function approximator (Q-Network) is not very stable

Deep Learning		**Reinforcement Learning**
Data samples are I.I.D.	*vs.*	States are highly correlated
Underlying data distribution is fixed		Data distribution changes

⇒

1. Exploration-Exploitation

2. Experience Replay

17

Exploration-Exploitation Dilemma

- During training, how do we choose an action at time t?
  - Exploration: random guessing
  - Exploitation: choose the best one according to the Q-value
- ϵ-greedy policy
  - With probability ϵ select a random action (Exploration)
  - Otherwise select $a = argmax_{a'} Q(s, a')$ (Exploitation)

18

Experience Replay

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

1. Take action a_t according to **ϵ-greedy policy**
2. During gameplay, store transition $< s_t, a_t, r_{t+1}, s_{t+1} >$ in replay memory D
3. Sample random mini-batch of transitions $< s, a, r, s' >$ from D
4. Optimize MSE between Q-network and Q-learning targets

$$L = \mathbb{E}_{s,a,r,s' \sim D} \frac{1}{2} [\boldsymbol{r + \gamma max_{a'} Q(s', a')} - Q(s, a)]^2$$

19

ϵ-greedy policy → With probability ε select a random action a_t

Experience memory → Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

Target network → Every C steps reset $\hat{Q} = Q$

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    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 $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \boxed{\hat{Q}}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - \boxed{Q}\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

20

Q Learning Recap

Pros:

- Can learn Q function from any interaction data, not just trajectories gathered using the current policy (“**off-policy” algorithm**)
- Relatively data-efficient (can reuse old interaction data)

Cons:

- Need to optimize over actions: hard to apply to continuous action spaces
- Optimal Q function can be complicated, hard to learn
- Optimal policy might be much simpler!

21

Model-free, policy based: Policy Gradient

Instead of learning the Q function, learn the policy directly!

Define a class of policies π_θ where θ are the parameters of the policy.

Can we learn the optimal θ from interaction?

Goal: use trajectories to estimate a gradient of policy performance w.r.t parameters θ

22

Policy Gradient

A particular value of θ induces a distribution of possible trajectories.

Objective function:

$$J(\theta) = E_{\tau \sim p(\tau;\theta)}[r(\tau)]$$

$$J(\theta) = \int_{\tau} r(\tau) p(\tau;\theta) d\tau$$

where $r(\tau)$ is the total discounted cumulative reward of a trajectory.

23

Policy Gradient

Gradient of objective w.r.t. parameters:

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

Trick: $\nabla_\theta p(\tau;\theta) = p(\tau;\theta)\frac{\nabla_\theta p(\tau;\theta)}{p(\tau;\theta)} = p(\tau;\theta)\nabla_\theta \log p(\tau;\theta)$

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

$$\nabla_\theta J(\theta) = E_{\tau \sim p(\tau;\theta)}[r(\tau)\nabla_\theta \log p(\tau;\theta)]$$

24

Policy Gradient

$$\nabla_\theta J(\theta) = E_{\tau \sim p(\tau;\theta)}[r(\tau)\nabla_\theta \log p(\tau;\theta)]$$

$$\log p(\tau;\theta) = \log\left(\prod_{t\geq 0} T(x_{t+1}|x_t, u_t)\pi_\theta(u_t|x_t)\right)$$

$$= \sum_{t\geq 0} \log T(x_{t+1}|x_t, u_t) + \log \pi_\theta(u_t|x_t)$$

$$\nabla_\theta \log p(\tau;\theta) = \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(u_t|x_t)$$

We don't need to know the transition model to compute this gradient!

25

Policy Gradient

If we use π_θ to sample a trajectory, we can approximate the gradient:

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

Intuition: adjust theta to:

- Boost probability of actions taken if reward is high
- Lower probability of actions taken if reward is low

Learning by trial and error

26

Policy Gradient Recap

Pros:

- Learns policy directly – often more stable
- Works for continuous action spaces
- Converges to local maximum of $J(\theta)$

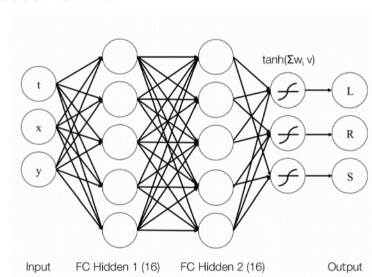
Cons:

- Needs data from current policy to compute gradient – data inefficient
- Gradient estimates can be very noisy

27

Deep policy gradient

- Parametrize policy as deep neural network
- In practice, very unstable
  - Need to combine with value estimate: *actor-critic*



28

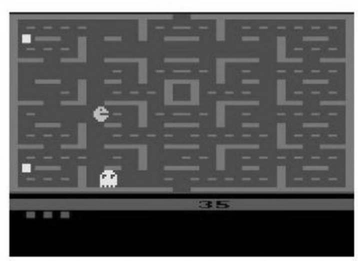
RL

- Model-based RL
  - Learn model from interacting with environment
- Model-free RL
  - Value-based methods: learn via minimizing bootstrapped TD error
  - Policy-based methods: directly optimize policy

29

Atari Games: Human-level Control through Deep Reinforcement Learning

- Volodymyr Mnih et al. (Google DeepMind) 2013/2015
- Idea: Let a neural network play Atari games!
- Input: Current and three subsequent video frames from game
- Processed by network trained with reinforcement learning
- Goal: learn best controller movements
- Convolutional layers for frame processing, fully-connected for final decision making

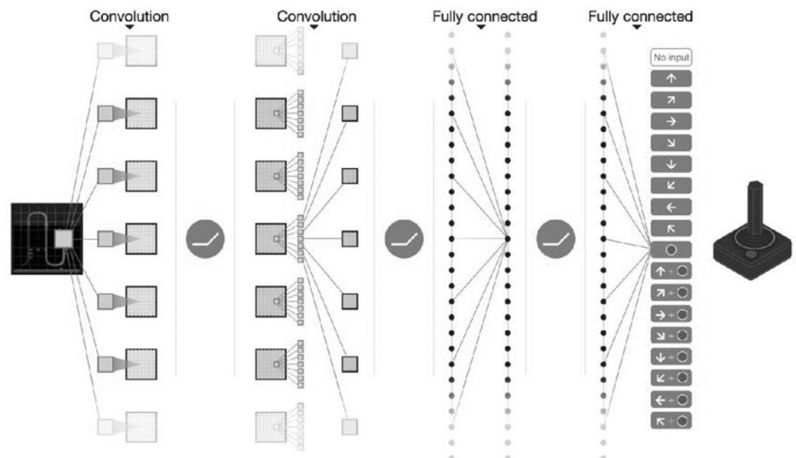


Atari Pac-Man

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. “Human-level control through deep reinforcement learning”. In: Nature 518.7540 (2015), pp. 529–533.

30

Learning Atari Games



Learning Atari Games

- **Deep Q-network**: Deep network that applies Q-learning
- State s_t of the game: current + 3 previous frames (image stack)
- 18 outputs associated with an action

→ Each output estimates optimal action value for "its" action given the input

- Instead of label & cost function, update to maximize reward
- Reward: +1/-1 when game score increased/decreased, 0 otherwise
- ϵ-greedy policy with ϵ decreasing to a low value during training
- Semi-gradient form of Q-learning to update network weights **w**
- Uses mini-batches to accumulate weight updates

Resource: Andreas Maier, Reinforcement Learning — Part 5, Deep Q-Learning

Target Network

- Weight update:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[r_{t+1} + \gamma \max_a \hat{q}(s_{t+1}, a, \mathbf{w}_t) - \hat{q}(s_t, a_t, \mathbf{w}_t) \right] \cdot \nabla \mathbf{w}_t \hat{q}(s_t, a_t, \mathbf{w}_t)$$

- Problem: The target $\gamma \max_a \hat{q}(s_{t+1}, a, \mathbf{w}_t)$ is a function of $\mathbf{w}_t$.

➔ Target changes simultaneously with the weights we want to learn!

➔ Training can oscillate or diverge

- Idea: Use a second **target network**:
- After each C steps, copy weights of action-value network to a duplicate network and keep them fixed
- Use output $\bar{q}$ of "target network" as a target to stabilize:

$$\gamma \max_a \bar{q}(s_{t+1}, a, \mathbf{w}_t)$$

33

Experience Replay

Goal: Reduce correlation between updates

- After performing action a_t for image stack s_t (state) and receiving reward r_t, add (s_t, a_t, r_t, s_{t+1}) to **replay memory**

➔ Memory accumulates experiences

- To update the network, draw random samples from memory, instead of taking the most recent ones

➔ Removes dependence on current weights

➔ Increases stability

34