# Deep Reinforcement Learning

CEng 783 – Deep Learning
Fall 2017

Emre Akbaş

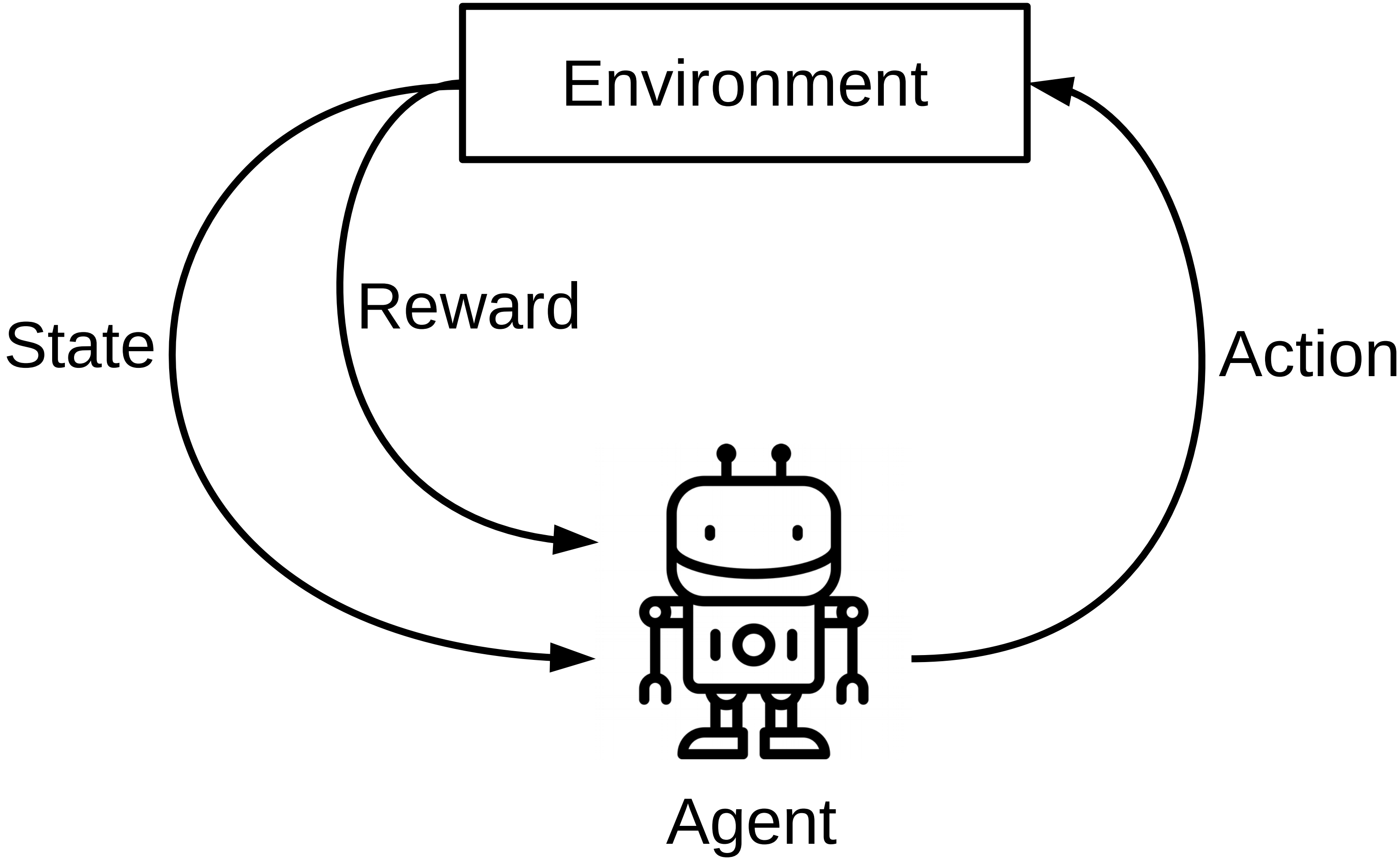
Timeline for projects:

- Presentations (selected groups)
- Final report due Jan 14 (details announced at ODTUClass)

# Today

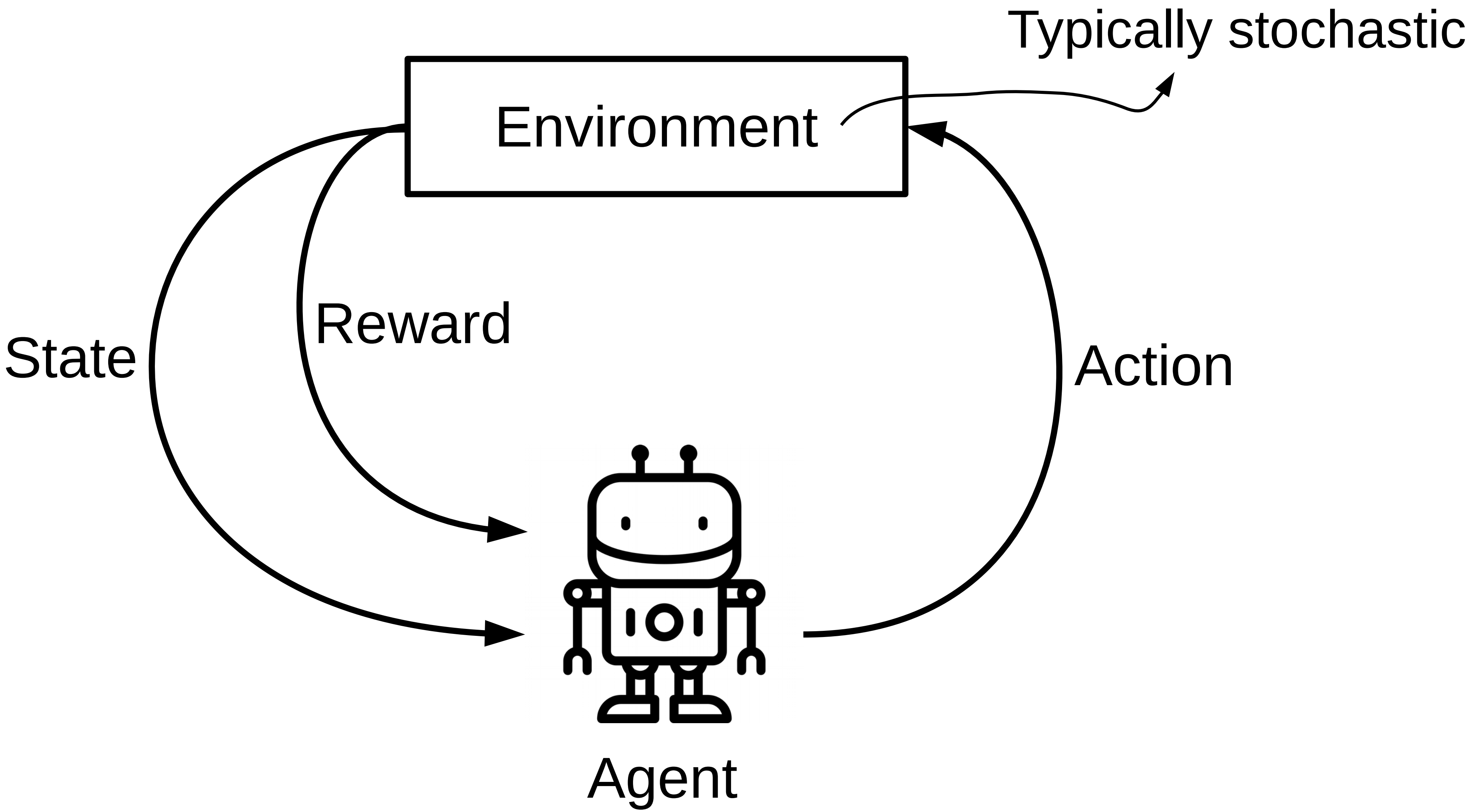
- Hands-on RNN tutorial (Hw #3)
- Brief introduction to Reinforcement Learning
  - Concepts
  - Markov Decision Processes
  - Bellman Equation
  - Q-learning
- Deep Q-learning
- Deep policy gradients

# Reinforcement learning




The agent is situated & operating in an environment

# Reinforcement learning

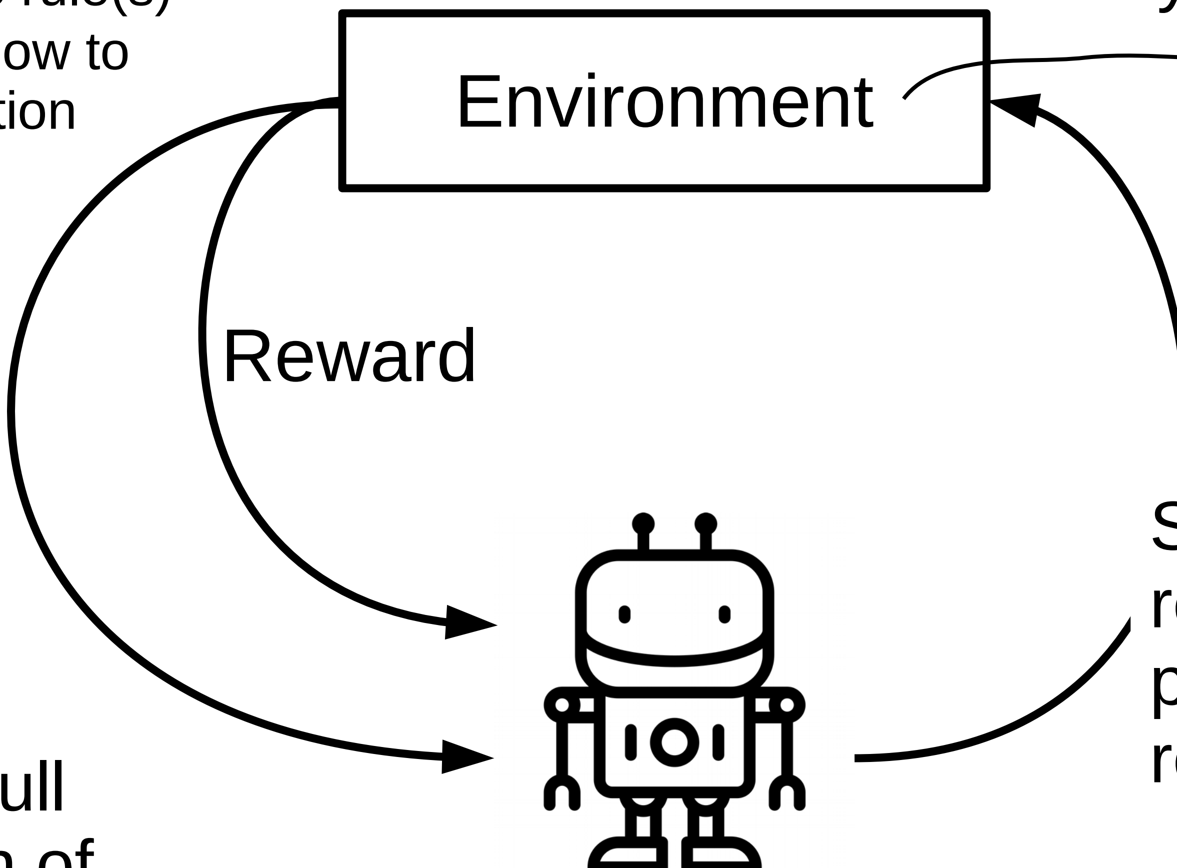



The agent is situated & operating in an environment

# Reinforcement learning

Policy: The rule(s) that specify how to choose an action

Typically stochastic




State

Action

Partial or full description of the environment at a certain time.

Some actions may result in positive/negative rewards.

Agent

The agent is situated & operating in an environment

# Reinforcement learning

Example: the backgammon game


- State
- Agent
- Action
- Reward
- Policy
- Environment (is stochastic in general)

How could we train an agent that would perform well in such a setting?

Well = maximize reward

Looking at only immediate rewards would not work well.

We need to take into account “future” rewards.

At time $t$, the total future reward is

$$R_t = r_t + r_{t+1} + \cdots + r_n$$

We want to take the action that maximizes $R_t$.

BUT we have to consider the fact that ...

At time $t$, the total future reward is

$$R_t = r_t + r_{t+1} + \cdots + r_n$$

We want to take the action that maximizes $R_t$.

BUT we have to consider the fact that **the environment is stochastic.**

So, discounting the future rewards is a good idea.

Discounted future rewards:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-t} r_n$$

where gamma is the discount factor between 0 and 1.

The more a reward is into the future, the less we care about it.

Discounted future rewards:

$$\begin{aligned} R_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-t} r_n \\ &= r_t + \gamma R_{t+1} \end{aligned}$$

Consider the cases

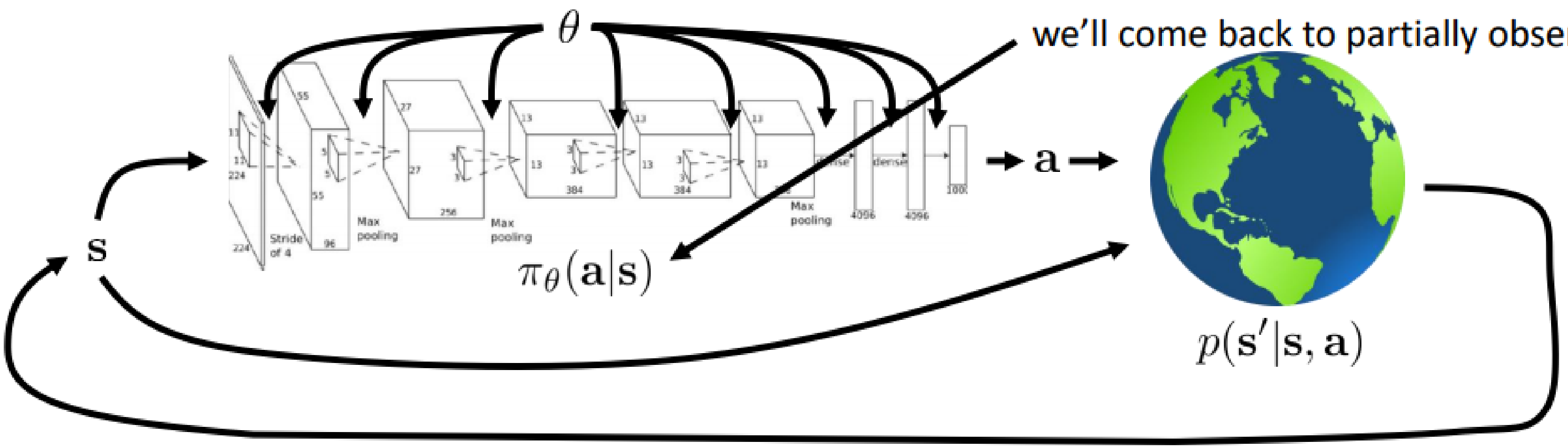
$$\gamma = 0 \quad \text{and} \quad \gamma = 1$$

# Discounted future rewards:

$$\begin{aligned} R_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-t} r_n \\ &= r_t + \gamma R_{t+1} \end{aligned}$$

- $\gamma = 0$
  - Considers the immediate rewards only.
  - Short-sighted. Won't work well
- $\gamma = 1$
  - Future rewards are not discounted
  - Should work in deterministic environments

# The goal of reinforcement learning



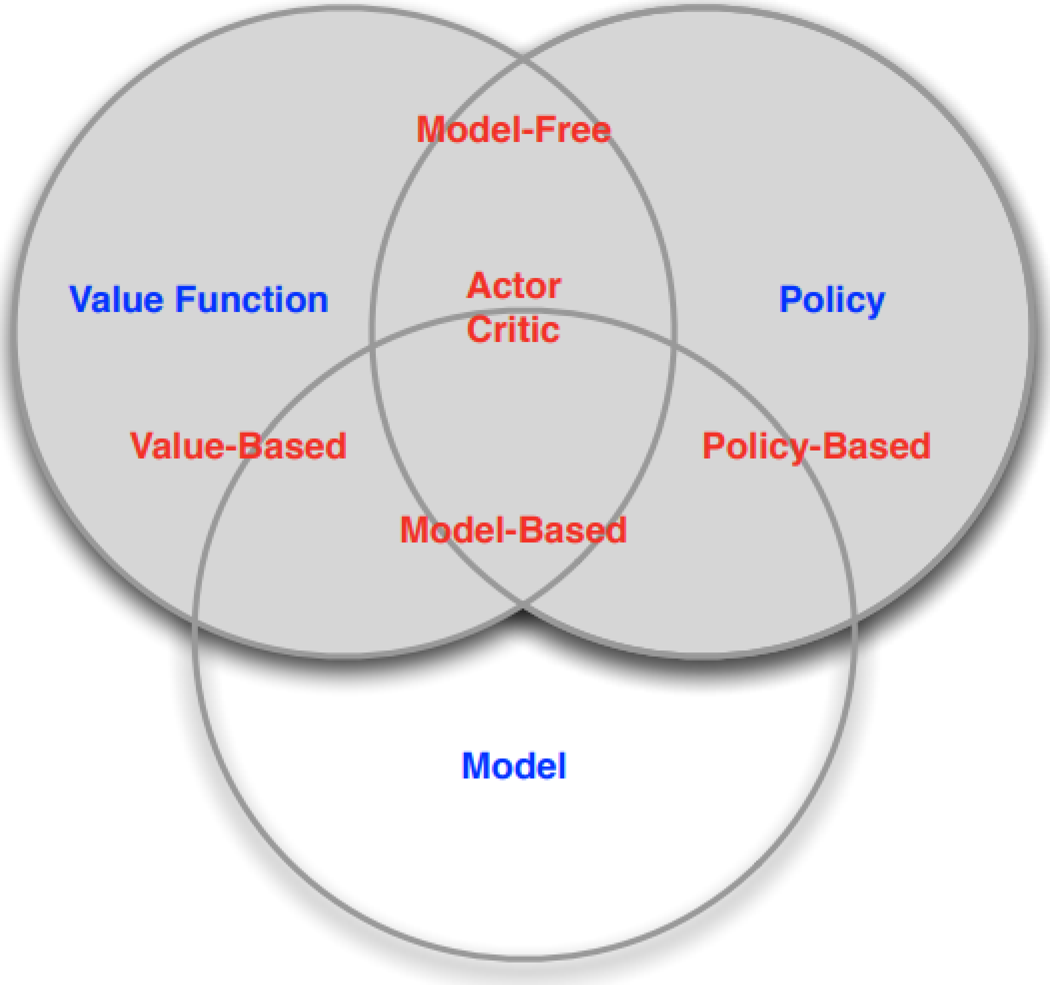

[Figure from Sergey Levine's slide.]

$$\theta^\star = \arg\max_\theta E_{\tau\sim p_\theta(\tau)}\left[\sum_t r(\mathbf{s}_t,\mathbf{a}_t)\right]$$

The goal of RL

$$p_\theta(\underbrace{\mathbf{s}_1,\mathbf{a}_1,\ldots,\mathbf{s}_T,\mathbf{a}_T}_{\tau})$$

# RL Agent Taxonomy




[Slide by David Silver]

# A policy-gradient method

- Based on direct differentiation of the cost function.

The goal of RL is to find model params $\theta$ to maximize the expected reward:

$$J(\theta) = E_{\tau \sim p_\theta(\tau)} \left[ \sum_t r(s_t, a_t) \right]$$

$p_\theta(\cdot)$ here is the joint distribution of all states and actions in an episode:

$$p_\theta(\underbrace{s_1, a_1, s_2, a_2, \ldots, s_T, a_T}_{\tau})$$

Maximize $J(\theta)$ $\Rightarrow$ use gradient ascent

what is $\nabla_\theta J(\theta)$ ?

$$J(\theta) = E_{\tau \sim p_\theta(\tau)} \Big[ \underbrace{\sum_t r(s_t, a_t)}_{\text{let's call this } R\text{, cumulative reward}} \Big]$$

$$J(\theta) = \sum_{\tau} p_\theta(\tau) R(\tau)$$

$$\nabla_\theta J(\theta) = \sum_{\tau} R(\tau) \nabla_\theta p_\theta(\tau)$$

$$p_\theta(\tau) \nabla_\theta \log p_\theta(\tau)$$

$$= \sum_{\tau} p_\theta(\tau) R(\tau) \nabla_\theta \log p_\theta(\tau)$$

$$= E_{\tau \sim p_\theta(\tau)} \left[ R(\tau) \nabla_\theta \underbrace{\log p_\theta(\tau)}_{\text{This is a problem. Why?}} \right]$$

We simplify $p_\theta(\tau)$ by assuming it has the Markov property:

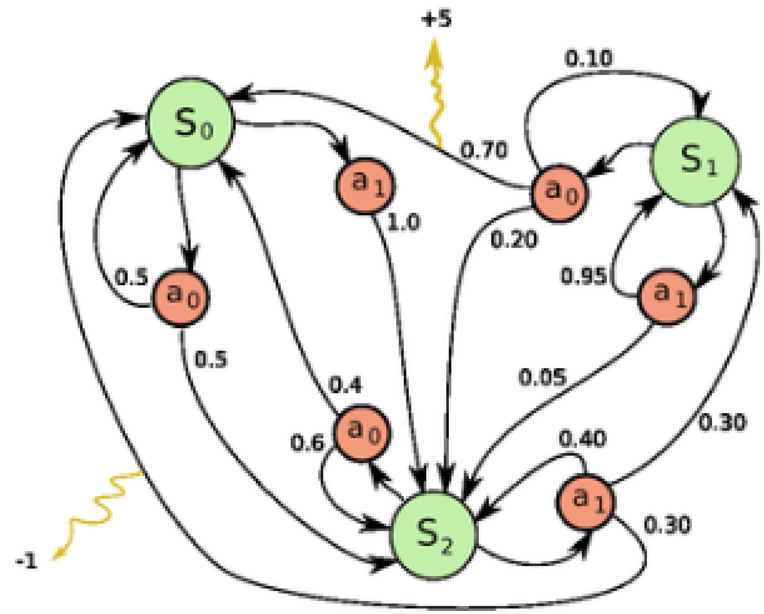
$$p_\theta(s_1, a_1, s_2, a_2, \dots, s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t | s_t)\, p(s_{t+1} | s_t, a_t)$$

Then, $\log p_\theta(\tau) = \log p(s_1) + \sum_{t=1}^{T} \log \pi_\theta(a_t | s_t) + \log p(s_{t+1} | s_t, a_t)$

$$\Rightarrow \nabla_\theta \log p_\theta(\tau) = \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t | s_t)$$

# Markov Decision Process

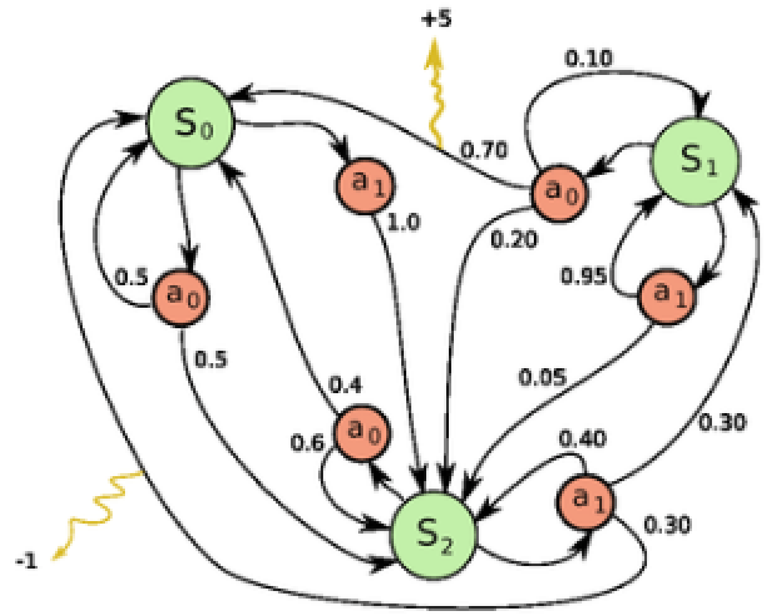
- Relies on the Markov assumption
- The probability of the next state depends only on the current state and the action but not on preceding states or actions.




[Image from nervanasys.com]

# Markov Decision Process

- Relies on the Markov assumption
- The probability of the next state depends only on the current state and the action but not on preceding states or actions.




[Image from nervanasys.com]

One episode (e.g. a game from start to finish) of this process forms a sequence:

$$< s_0, a_0, r_1, s_1 >, < s_1, a_1, r_2, s_2 >, \ldots, < s_{n-1}, a_{n-1}, r_n, s_n >$$

# The REINFORCE algorithm

The previous derivations result in the following algorithm.

REINFORCE algorithm:

1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run the policy)
2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right)\left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$
3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

[From Sergey Levine's slide.]

An example application is in the next slides.

# Example: The “Pong” game

Image from http://karpathy.github.io/2016/05/31/rl/

State: current frame minus the last frame (to capture the motion)

Action: UP or DOWN

Reward: +1 if you win the game, otherwise -1

# Example: “Pong”

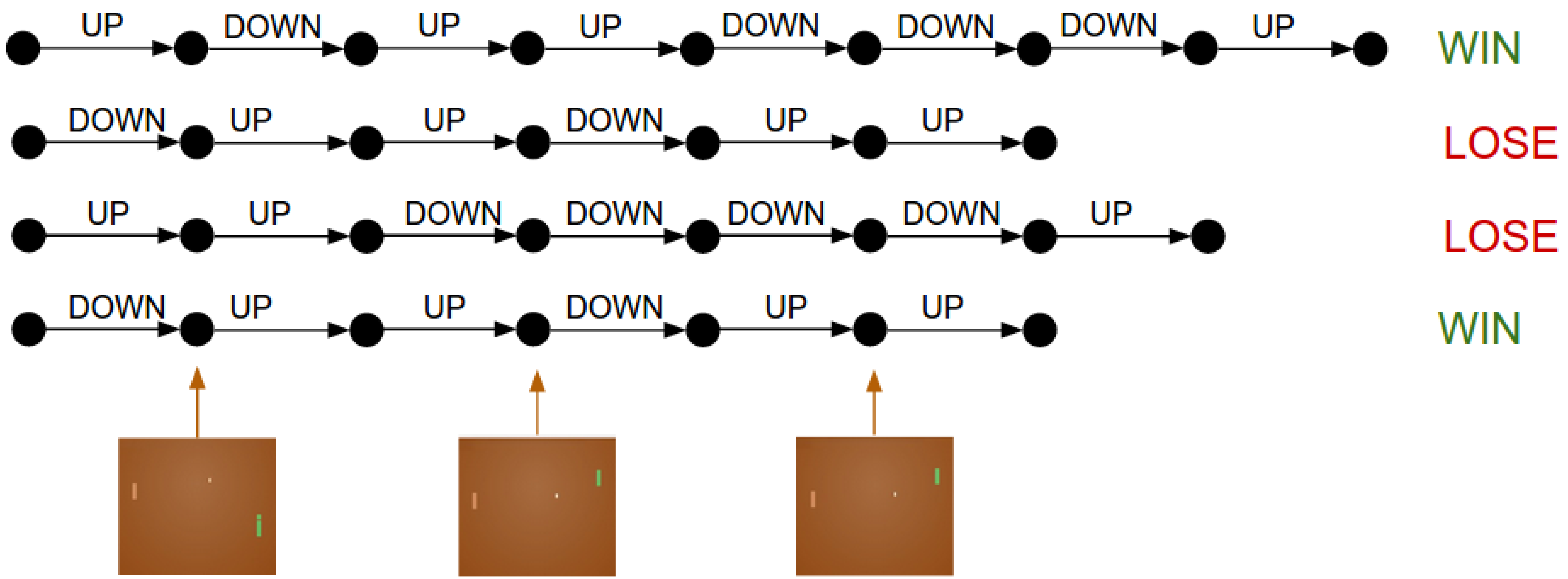



Image from http://karpathy.github.io/2016/05/31/rl/

# Learning the optimal policy

1) Initialize the policy network randomly
2) Play a batch of episodes
3) Label all the actions in a WIN game as CORRECT
4) Label all the actions in a LOST game as INCORRECT
5) Apply supervised learning, i.e. maximize

$$\sum_i r_i \log p(a_i|s_i)$$

where $r_i$ =1 for any action in a WON game, otherwise -1

6) Go to step 2, repeat until convergence.

# Q-learning: a value-based method

- The Q-function:
  - Expected total reward from taking action a_t in s_t

$$Q(s_t, a_t) = r_{t+1} + \gamma \max_x Q(s_{t+1}, x)$$

This is called the **Bellman equation**.

# The policy π

The rule that specifies which action to choose given the current state.

A typical and sensible policy is

$$\pi(s) = \arg\max_a \; Q(s, a)$$

# Q-learning

Suppose the current state is “*s*” and we take action “*a*”

Then, we observe the next state “*s'* “ and obtain reward “ *r*”.

It has been shown that the following iterative update rule will converge to an optimal Q function:

$$Q(s,a) = (1-\alpha)\, Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a'))$$

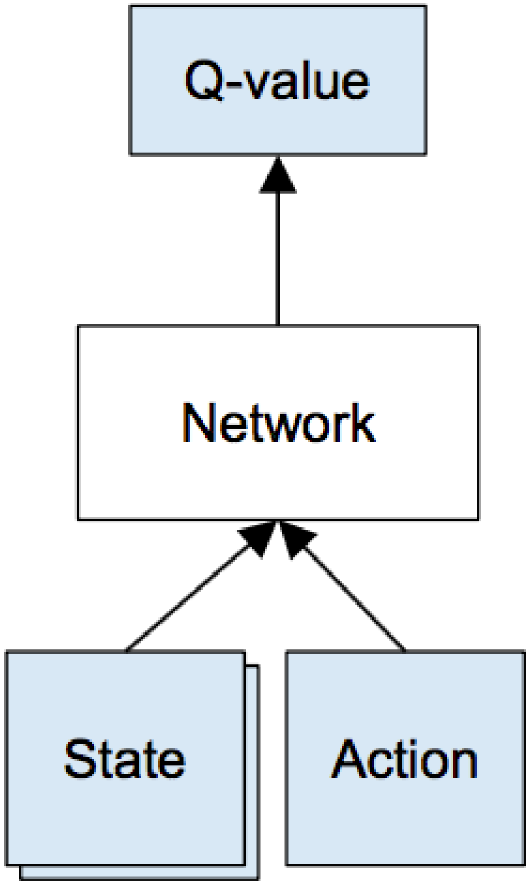
# Q-learning pseudo-code

1) Create a table (num_states x num_actions) for Q
2) Initialize the table randomly
3) Observe the initial state s
4) Repeat until the game is over (i.e. next state is terminal state)
   1) Choose an action, i.e. $a = \pi(s)$ /* s is the current state */
   2) Receive next state $s'$ and reward $r$
   3) $Q(s,a) = (1-\alpha)\, Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a'))$
   4) s = s' /* next state becomes the current state */

# How can we use deep-learning here?

The Q-function can be approximated using a neural network model.

Q(s,a) is implemented on the right.

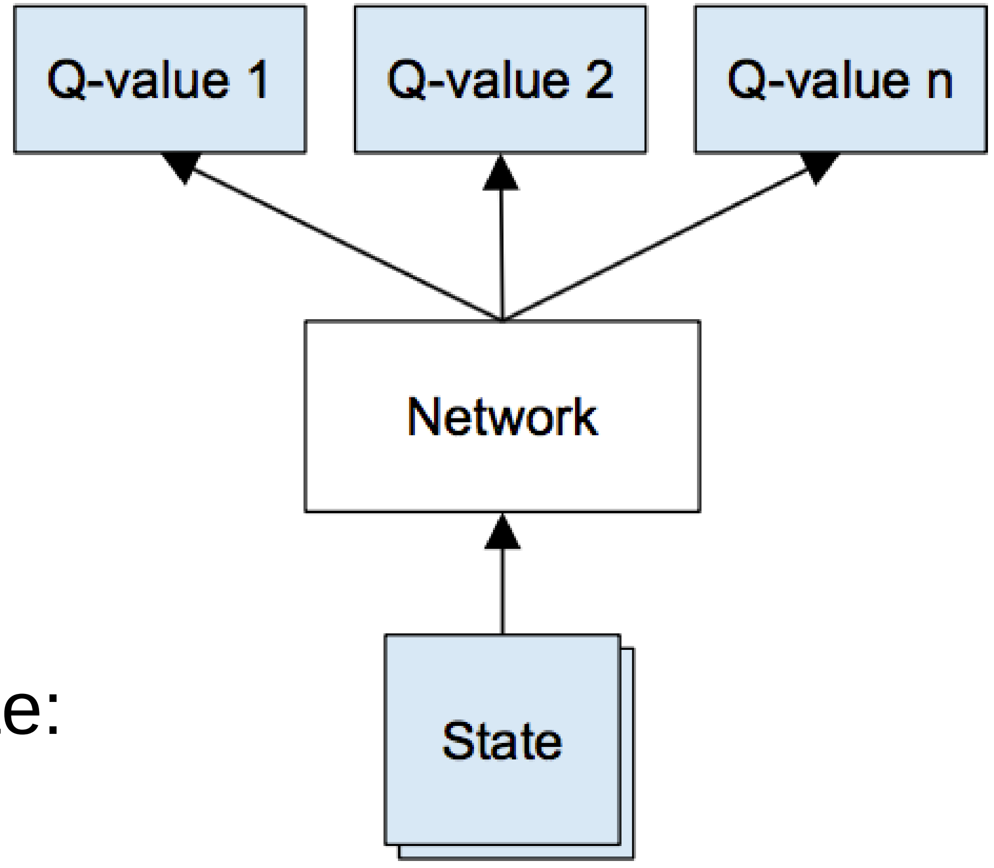



What about $\max_{a'} Q(s', a')$?

[Image from nervanasys.com]

# How can we use deep-learning here?

Alternative model for implementing Q(s,a)




Now, it's efficient to evaluate:

$$\max_{a'} Q(s', a')$$

[Image from nervanasys.com]

# Example

In DeepMind's 2013 paper [Mnih et al. (2013)] state is encoded by the images of last four frames of the game.

After pre-processing, a state is a 84x84x4 matrix.

# Example

The network used in DeepMind's 2013 paper

[Mnih et al. (2013)]

| 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 |

Qs: What type of a network is this? Why 18? Why no pooling layers?

# Deep Q-learning

Remember the Bellman equation?

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

For a transition <s,a,r,s'>,

The update rule in ordinary Q-learning:

$$Q(s,a) = (1-\alpha)\, Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s', a'))$$

In deep Q-learning:

$$\text{minimize} \quad (r + \gamma \max_{a'} Q(s', a') - Q(s,a))^2$$

# Deep Q-learning

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
      select an action a
            with probability ε select a random action
            otherwise select a = argmax_a′ Q(s,a′)
      carry out action a
      observe reward r and new state s′
      store experience <s, a, r, s′> in replay memory D

      sample random transitions <ss, aa, rr, ss′> from replay memory D
      calculate target for each minibatch transition
            if ss′ is terminal state then tt = rr
            otherwise tt = rr + γmax_a′ Q(ss′, aa′)
      train the Q network using (tt - Q(ss, aa))^2 as loss

      s = s'
until terminated
```

[Pseudocode from nervanasys.com]

# Deep Q-learning

- “It turns out that approximation of Q-values using non-linear functions is not very stable.
- There is a whole bag of tricks (reward clipping, gradient clipping, batch normalization) that you have to use to actually make it converge.
- The most important trick is experience replay.”

[Quote from nervanasys.com]

Experience relay is nothing but minibatch training. i.e. collect <s,a,r,s'> transitions and use them in minibatches (instead of one by one) while training.

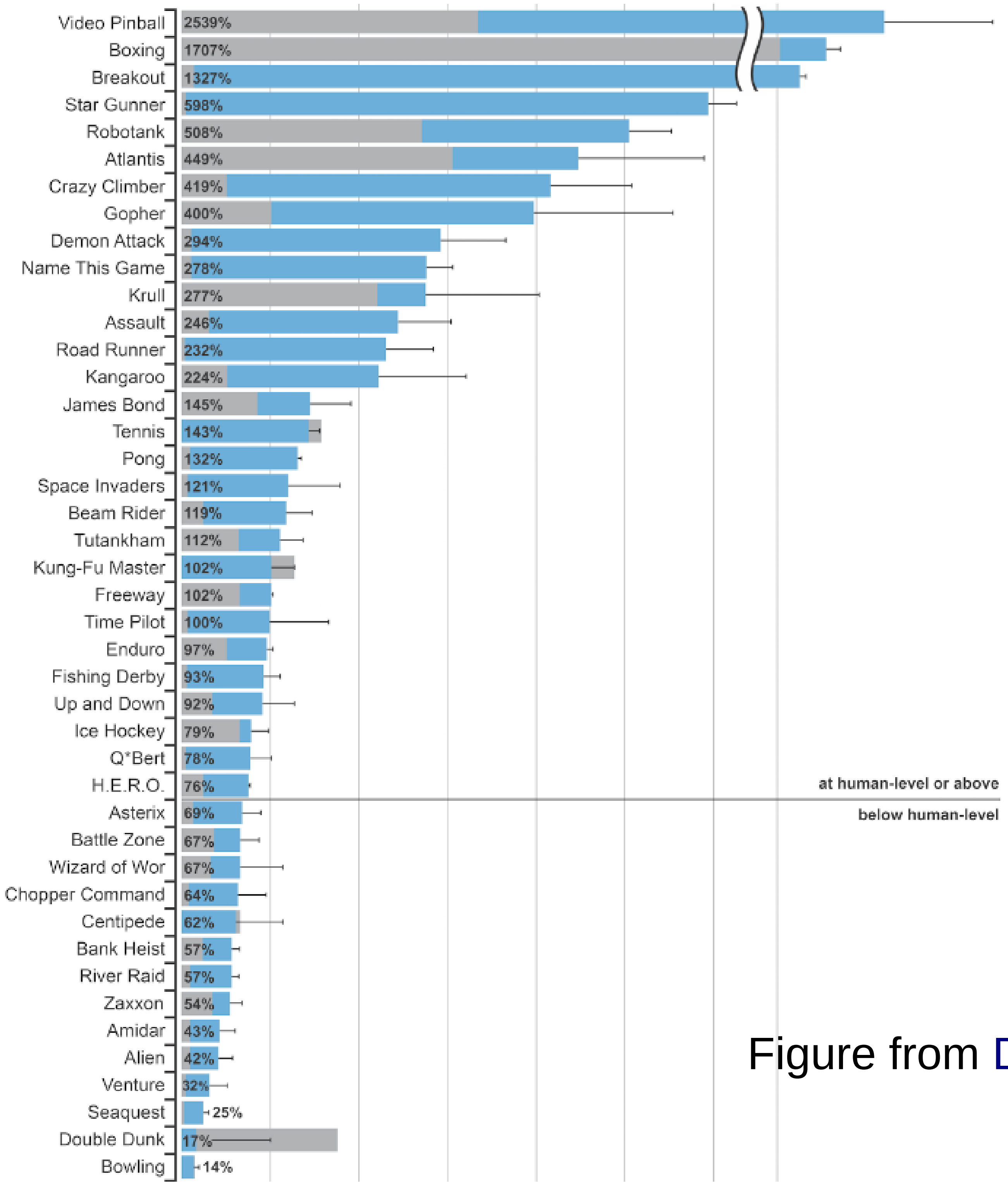



Figure from Deepmind’s post.]

# Fun fact

- Deep Q-learning has been patented by Google!
  - https://www.google.com/patents/US20150100530

# References


- Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.
- Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.
- Guest Post (Part I): Demystifying Deep Reinforcement Learning (Blog post), https://www.nervanasys.com/demystifying-deep-reinforcement-learning/
- Deep Reinforcement Learning: Pong from Pixels (Blog post) by A. Karpathy, http://karpathy.github.io/2016/05/31/rl/