# REINFORCE and Policy Gradient Algorithms for Reinforcement Learning

Gerald Pho

Mahmood Mohammadi Shad

December 13, 2018

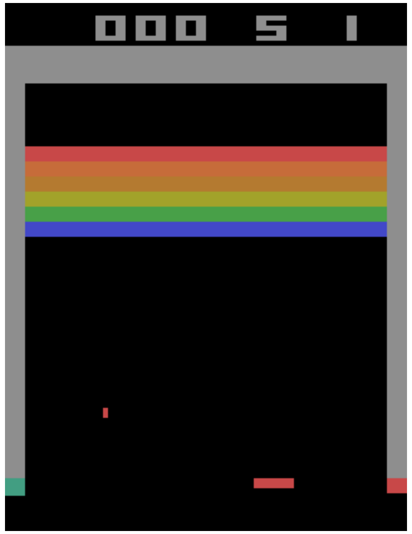
# **Reinforcement learning**: A framework for learning intelligent **behaviors**

- A completely distinct framework from supervised and unsupervised learning
  - Trained from rewards – no supervisor
  - Making decisions to maximize reward – not just finding hidden structure
  - Applies to sequential problems that evolve over time

## **Reinforcement learning**:
## A framework for learning intelligent **behaviors**

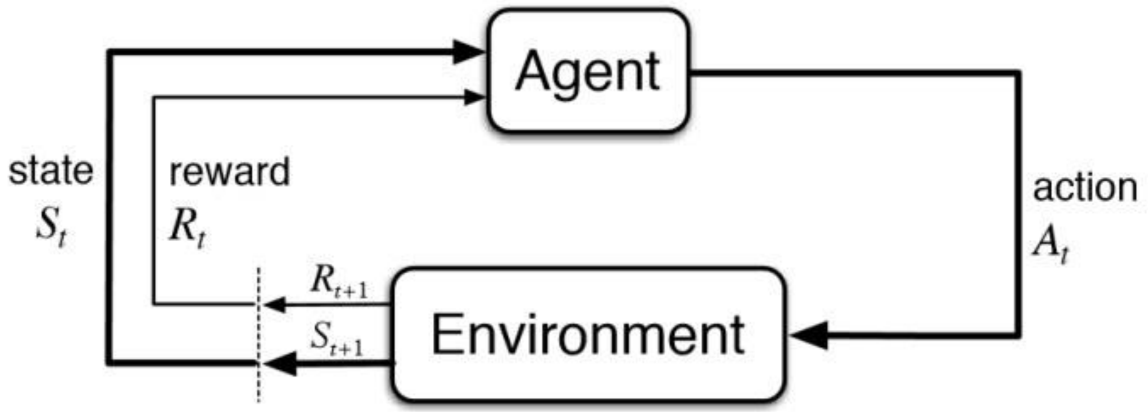
- A completely distinct framework from supervised and unsupervised learning
  - Trained from rewards – no supervisor
  - Making decisions to maximize reward – not just finding hidden structure
  - Applies to sequential problems that evolve over time

- Example problems
  - Games
  - Robots (real or simulated)
  - Advertising
  - Managing investments
  - Optimizing factory processes
  - Self-driving cars?

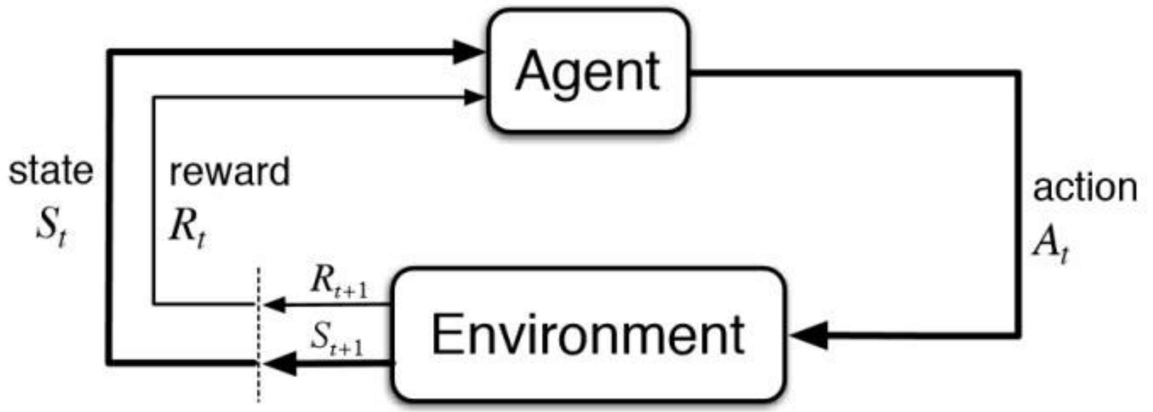



https://ai.googleblog.com/2018/06/scalable-deep-reinforcement-learning.html

# The reinforcement learning problem



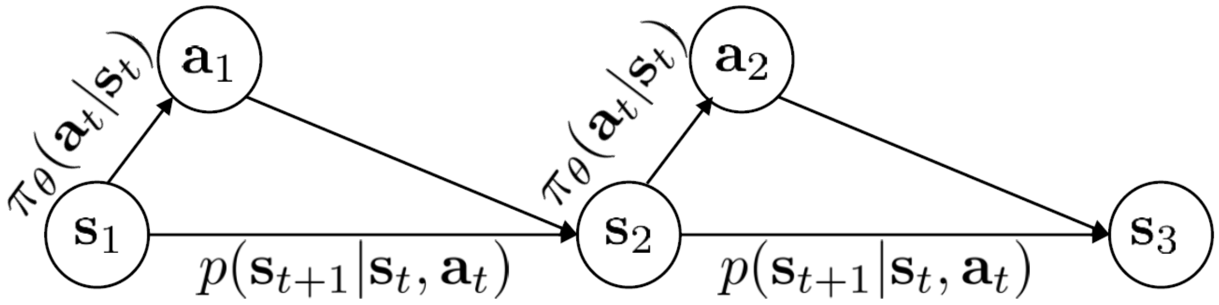
# The reinforcement learning problem



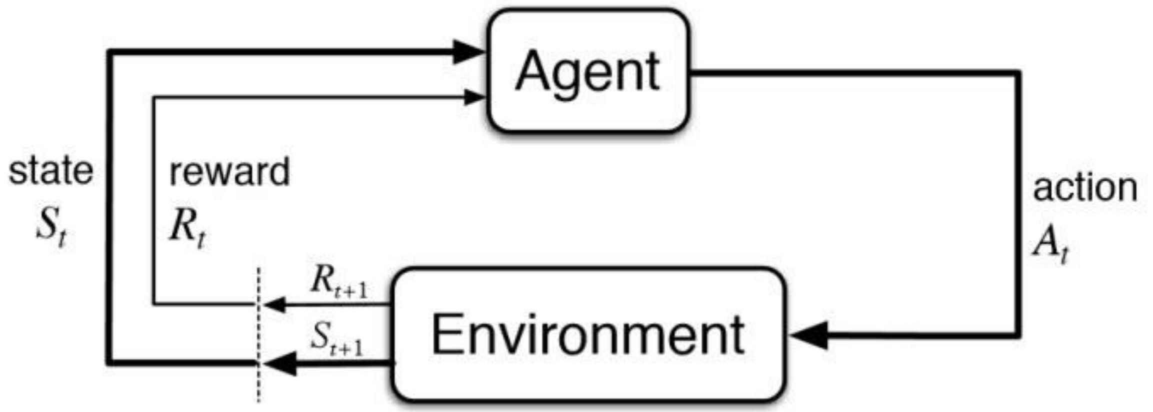

**Markov Decision Process** – key components

$p(s_{t+1}|s_t, a_t)$: transition model of world

$\pi_\theta(a_t|s_t)$: policy (probability) of choosing actions given states



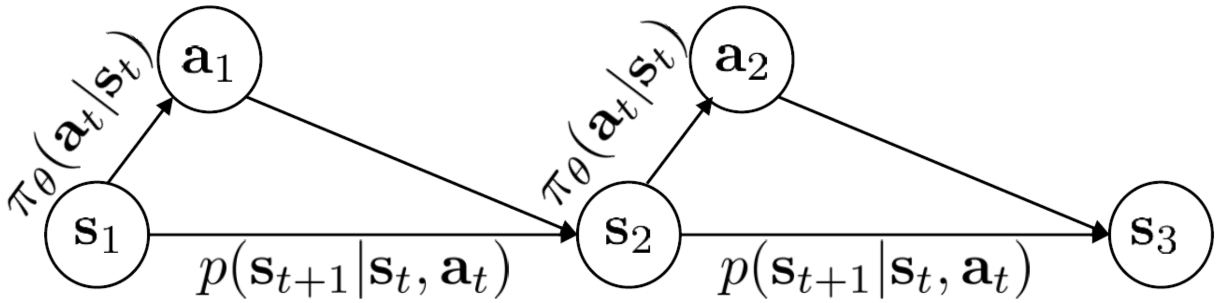
# The reinforcement learning problem




**Markov Decision Process** – key components

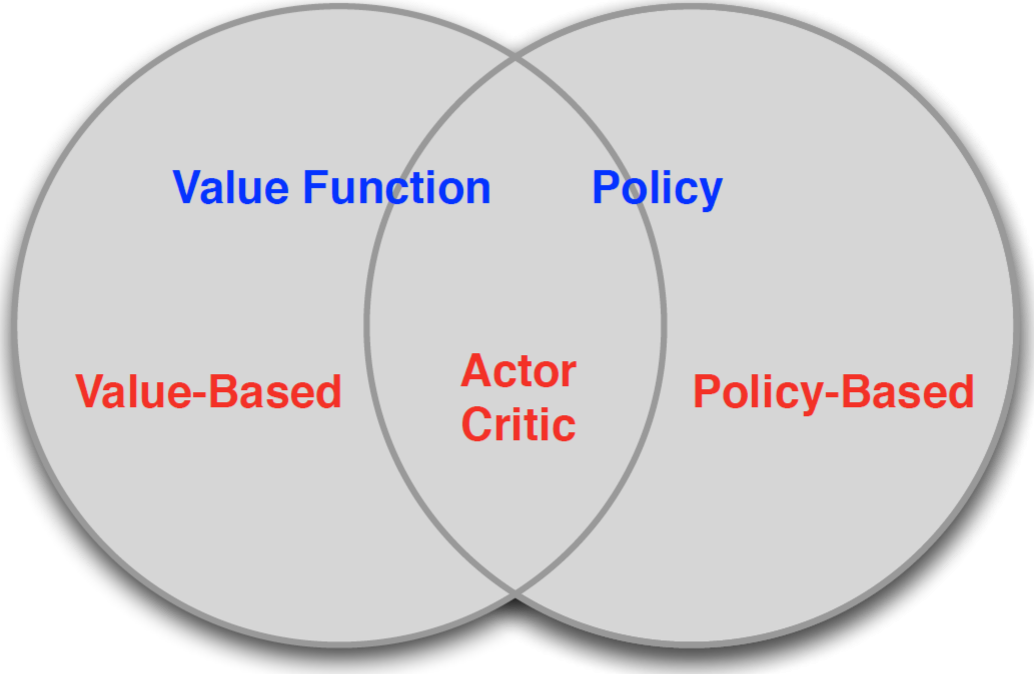
$p(s_{t+1}|s_t, a_t)$: transition model of world

$\pi_\theta(a_t|s_t)$: policy (probability) of choosing actions given states




**Goal:** Learn a policy $\pi_\theta$ that will pick actions to maximize reward

# Types of RL algorithms

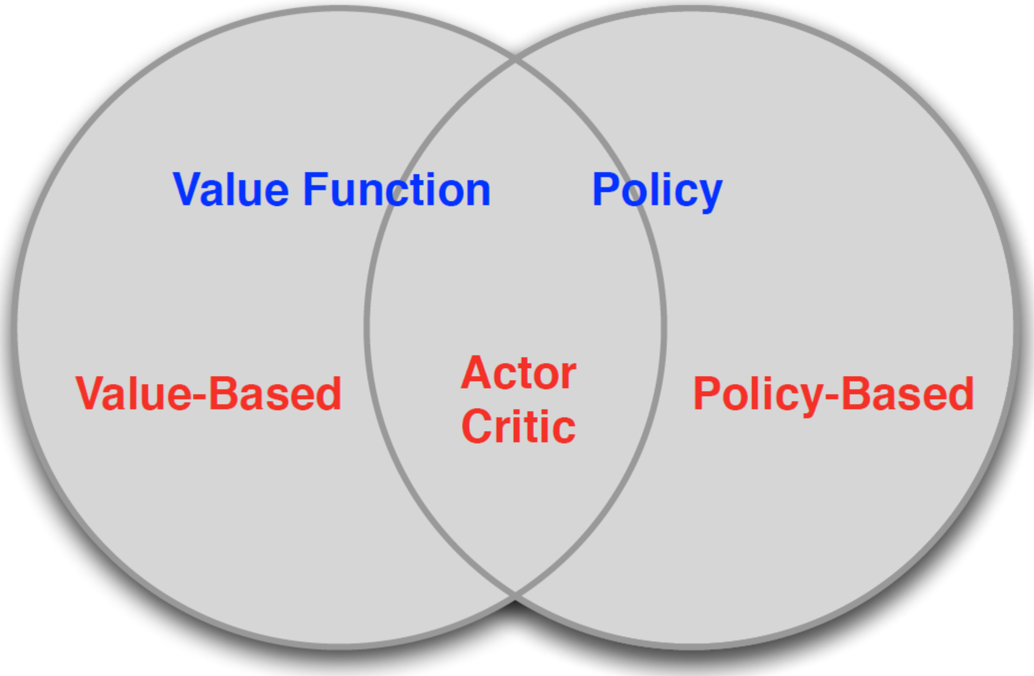



http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf

# Types of RL algorithms

**Value-based methods**

- Learn a value function that estimates expected reward of states $V(s)$ and/or actions $Q(s, a)$
- No explicit policy = instead choose actions that have the highest value as predicted by the value function
- Examples: TD learning, Q-learning




http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf

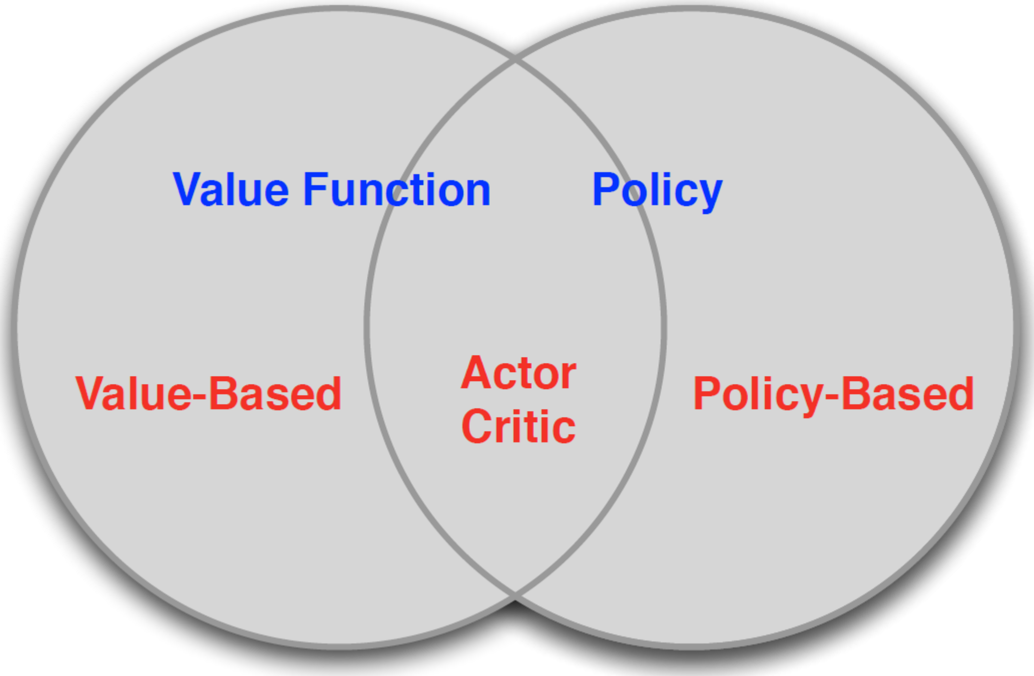
# Types of RL algorithms

**Value-based methods**

- Learn a value function that estimates expected reward of states $V(s)$ and/or actions $Q(s,a)$
- No explicit policy = instead choose actions that have the highest value as predicted by the value function
- Examples: TD learning, Q-learning

**Policy-based methods**

- Represent policy explicitly using a parametrized function of states $\pi_\theta(s)$
- Optimize the policy parameters using gradient descent
- Examples: REINFORCE, Natural Policy Gradient, PPO




http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf

# Types of RL algorithms

**Value-based methods**

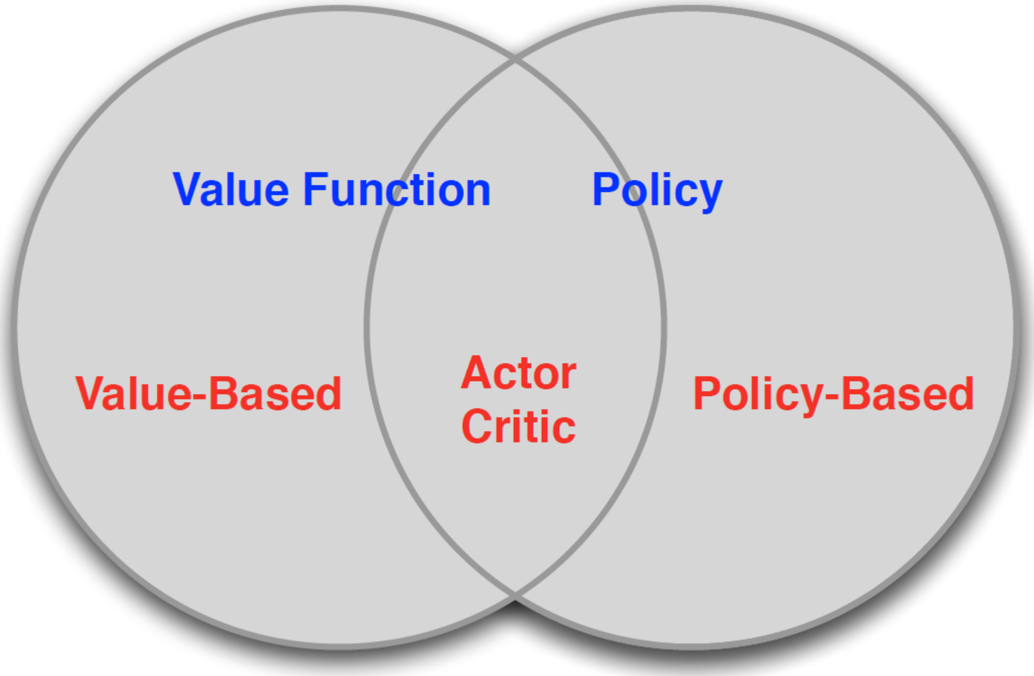
- Learn a value function that estimates expected reward of states $V(s)$ and/or actions $Q(s,a)$
- No explicit policy = instead choose actions that have the highest value as predicted by the value function
- Examples: TD learning, Q-learning

**Policy-based methods**

- Represent policy explicitly using a parametrized function of states $\pi_\theta(s)$
- Optimize the policy parameters using gradient descent
- Examples: REINFORCE, Natural Policy Gradient, PPO

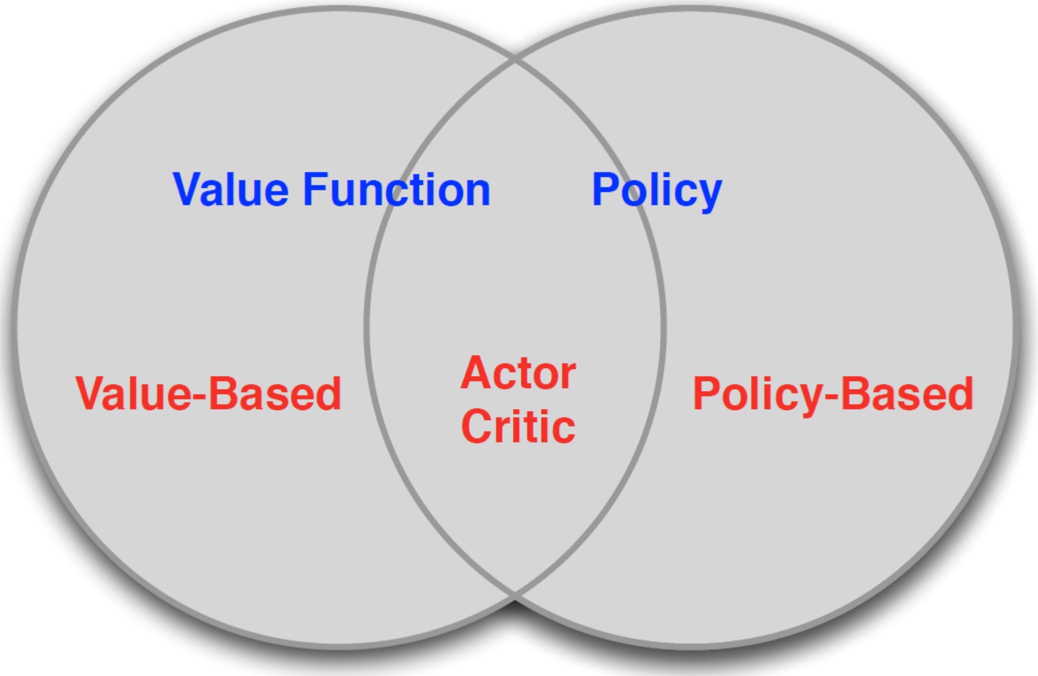
**Actor-Critic**

- Combine both: learn a value function (critic) that is used to guide improvements of the policy (actor)




http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf

# Types of RL algorithms



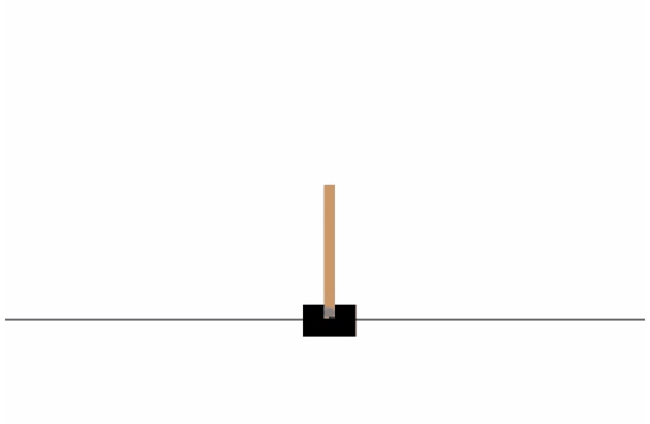

**Policy-based methods**

- Represent policy explicitly using a parametrized function of states $\pi_\theta(s)$
- Optimize the policy parameters using gradient descent
- Examples: REINFORCE, Natural Policy Gradient, PPO

http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf

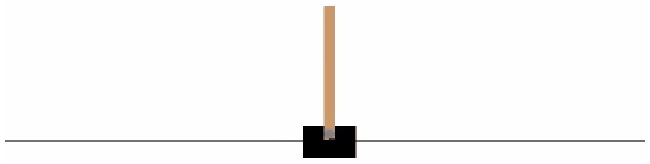
# Gaining intuition: two toy problems

- Cartpole task
  - Goal: keep the pole straight (within 12 degrees) and the cart within the screen
  - States are 4-dimensional vector: $[x, \dot{x}, \theta, \dot{\theta}]$
  - Actions are discrete: [move left, move right]

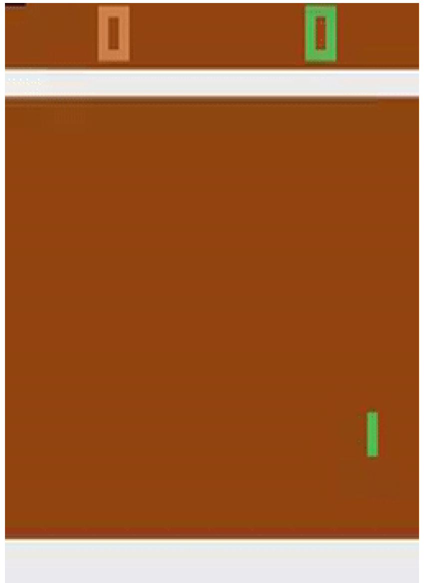


# Gaining intuition: two toy problems

- Cartpole task
  - Goal: keep the pole straight (within 12 degrees) and the cart within the screen
  - States are 4-dimensional vector: $[x, \dot{x}, \theta, \dot{\theta}]$
  - Actions are discrete: [move left, move right]



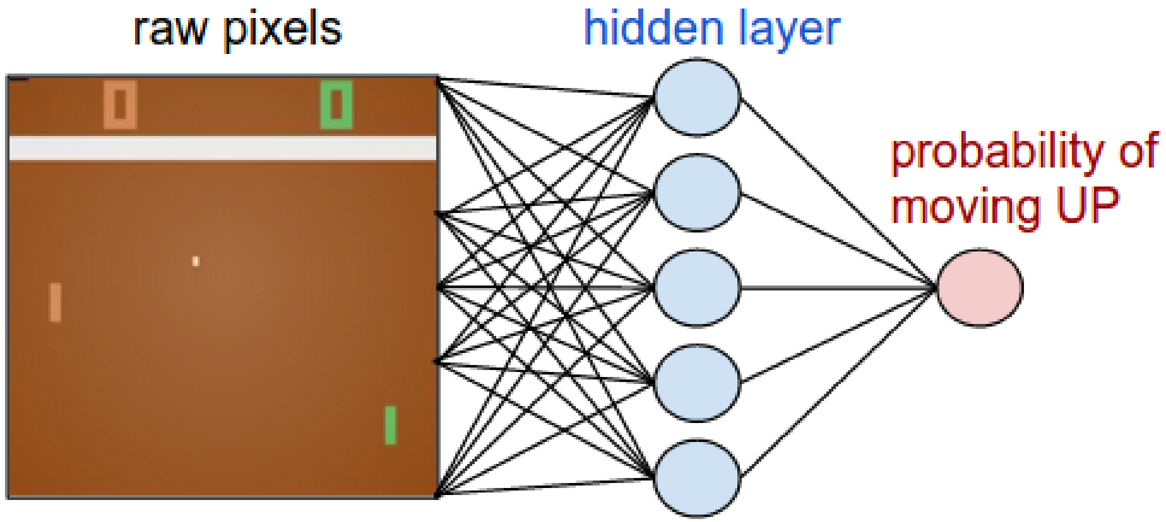
- Pong from Atari
  - Goal: maximize score relative to opponent
  - States are images [210 x 160 x 3]
  - Actions are discrete: [move up, move down]



# Policy parametrization

For discrete actions, often a softmax policy is used: $\pi_\theta(a|s) \propto e^{f_\theta(s)}$

- $f_\theta(s)$ is some function approximator, e.g. a neural network

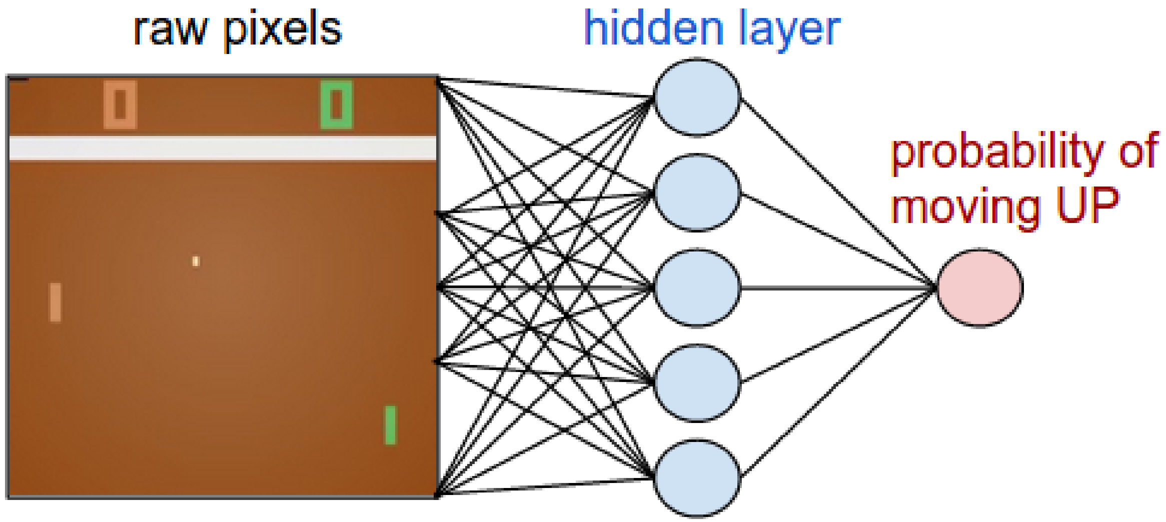



http://karpathy.github.io/2016/05/31/rl/
https://deepmind.com/blog/producing-flexible-behaviours-simulated-environments/

# Policy parametrization

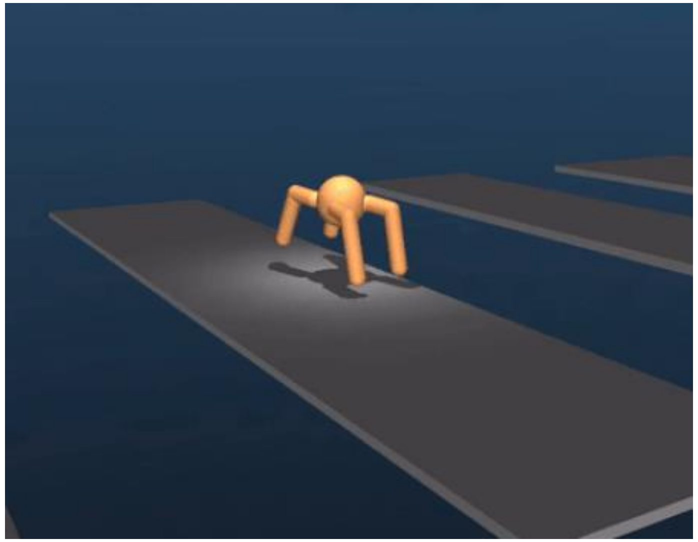
For discrete actions, often a softmax policy is used: $\pi_\theta(a|s) \propto e^{f_\theta(s)}$

- $f_\theta(s)$ is some function approximator, e.g. a neural network




For continuous actions, usually a Gaussian policy is used:

$$\pi_\theta(a|s) \sim Normal(f_\theta(s), \sigma)$$



used in robotics

http://karpathy.github.io/2016/05/31/rl/
https://deepmind.com/blog/producing-flexible-behaviours-simulated-environments/

# Supervised Learning approach

<u>Supervised learning for classification</u>

Maximum likelihood objective:

$$J(\theta) \propto \sum_{i=1}^{N} \log p(y_i | x_i; \theta)$$

# Supervised Learning approach

Supervised learning for classification

Maximum likelihood objective:

$$J(\theta) \propto \sum_{i=1}^{N} \log p(y_i | x_i; \theta)$$

Gradient of ML objective:

$$\nabla_\theta J(\theta) \propto \sum_{i=1}^{N} \nabla_\theta \log p(y_i | x_i; \theta)$$

Algorithm: move in direction of gradient, weighted by some learning rate $\alpha$

# Supervised Learning approach

Supervised learning for classification

Maximum likelihood objective:

$$J(\theta) \propto \sum_{i=1}^{N} \log p(y_i | x_i; \theta)$$

Gradient of ML objective:

$$\nabla_\theta J(\theta) \propto \sum_{i=1}^{N} \nabla_\theta \log p(y_i | x_i; \theta)$$

Algorithm: move in direction of gradient, weighted by some learning rate $\alpha$

Supervised learning of a policy

If given the correct actions $a^*$, our objective becomes:

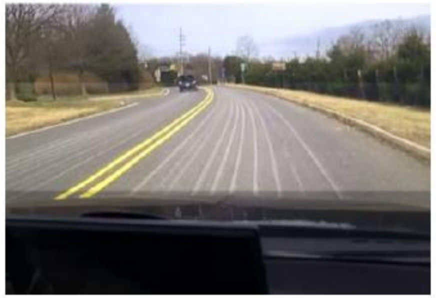
$$J(\theta) \propto \sum_{i=1}^{N} \log \pi_\theta(a_i^* | s_i)$$

Gradient of ML objective:

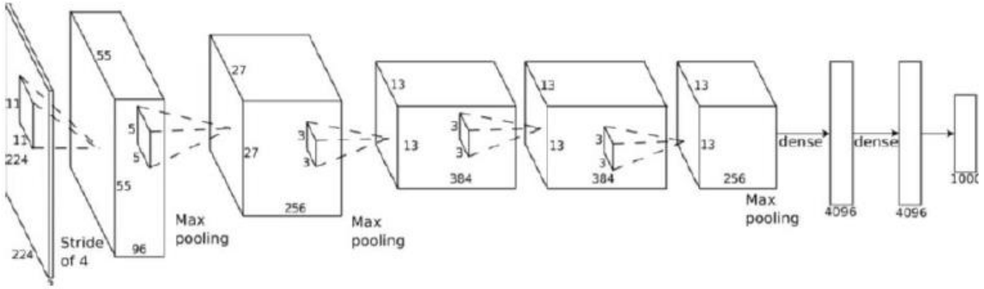
$$\nabla_\theta J(\theta) \propto \sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(a_i^* | s_i)$$

$\pi_\theta$ = policy that outputs a probability distribution over actions, given states

# Supervised Learning approach



$\mathbf{s}_t$




$\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$

$\mathbf{a}_t$

$\mathbf{s}_t$

$\mathbf{a}_t$

→ training data → supervised learning → $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$

http://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-5.pdf

# Policy Gradient vs Supervised Learning

But we don't know the correct actions → we only know that the sequence of actions we tried:

$$s_1, a_1, r_1, s_2, a_2, r_2, s_3, a_3, r_3, \ldots$$

# Policy Gradient vs Supervised Learning

But we don't know the correct actions → we only know that the sequence of actions we tried:

$$s_1, a_1, r_1, s_2, a_2, r_2, s_3, a_3, r_3, \ldots$$

The basic idea of policy gradient is to use the same supervised learning gradient – but instead use the *actual* actions we actually took, **weighted by the reward** (summed over the future)

$$\nabla_\theta J(\theta) \propto \sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(a_i^* | s_i) * R$$

# Policy Gradient vs Supervised Learning

But we don’t know the correct actions → we only know that the sequence of actions we tried:

$$s_1, a_1, r_1, s_2, a_2, r_2, s_3, a_3, r_3, \dots$$

The basic idea of policy gradient is to use the same supervised learning gradient – but instead use the *actual* actions we actually took, **weighted by the reward** (summed over the future)

$$\nabla_\theta J(\theta) \propto \sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(a_i^* | s_i) * R$$

**Intuition:** Increase the probability of “good” actions, decrease the probability of “bad” actions.

A formalized algorithm for trial-and-error!

# REINFORCE algorithm (Williams 1992) aka "vanilla policy gradient"

**function REINFORCE**
  Initialise $\theta$ arbitrarily
  **for** each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
    **for** $t = 1$ to $T - 1$ **do**
      $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) \boldsymbol{g_t}$
    **end for**
  **end for**
  **return** $\theta$
**end function**

http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf

# REINFORCE algorithm (Williams 1992) aka "vanilla policy gradient"

1) Run the policy for 1+ episodes

**function REINFORCE**
  Initialise $\theta$ arbitrarily
  **for** each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
    **for** $t = 1$ to $T - 1$ **do**
      $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) \boldsymbol{g_t}$
    **end for**
  **end for**
  **return** $\theta$
**end function**

http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf

# REINFORCE algorithm (Williams 1992) aka "vanilla policy gradient"

1) Run the policy for 1+ episodes

2) Estimate the return $g_t = \sum_{\tau=t}^{T} \gamma^{t-\tau} r_\tau$

**function REINFORCE**
  Initialise $\theta$ arbitrarily
  **for** each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
    **for** $t = 1$ to $T - 1$ **do**
      $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) \, \boldsymbol{g_t}$
    **end for**
  **end for**
  **return** $\theta$
**end function**

http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf

# REINFORCE algorithm (Williams 1992) aka "vanilla policy gradient"

1) Run the policy for 1+ episodes

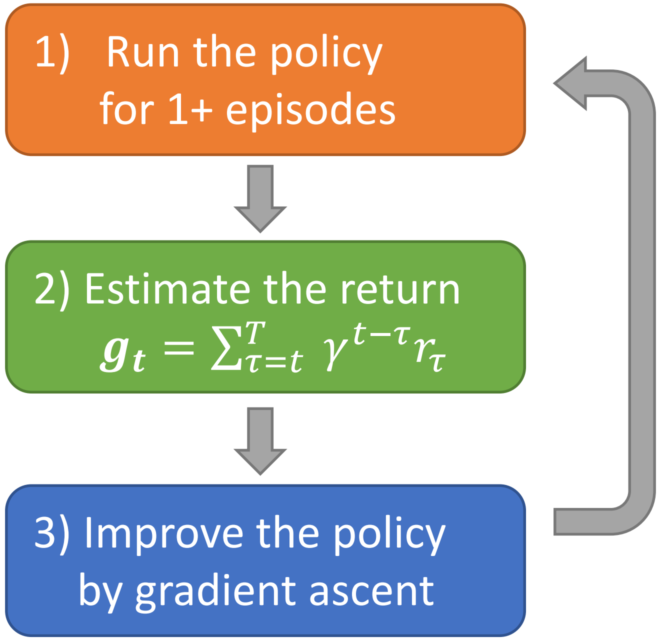
2) Estimate the return $g_t = \sum_{\tau=t}^{T} \gamma^{t-\tau} r_\tau$

3) Improve the policy by gradient ascent

**function REINFORCE**
  Initialise $\theta$ arbitrarily
  **for** each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
    **for** $t = 1$ to $T - 1$ **do**
      $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) \boldsymbol{g_t}$
    **end for**
  **end for**
  **return** $\theta$
**end function**

http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf

# REINFORCE algorithm (Williams 1992) aka "vanilla policy gradient"

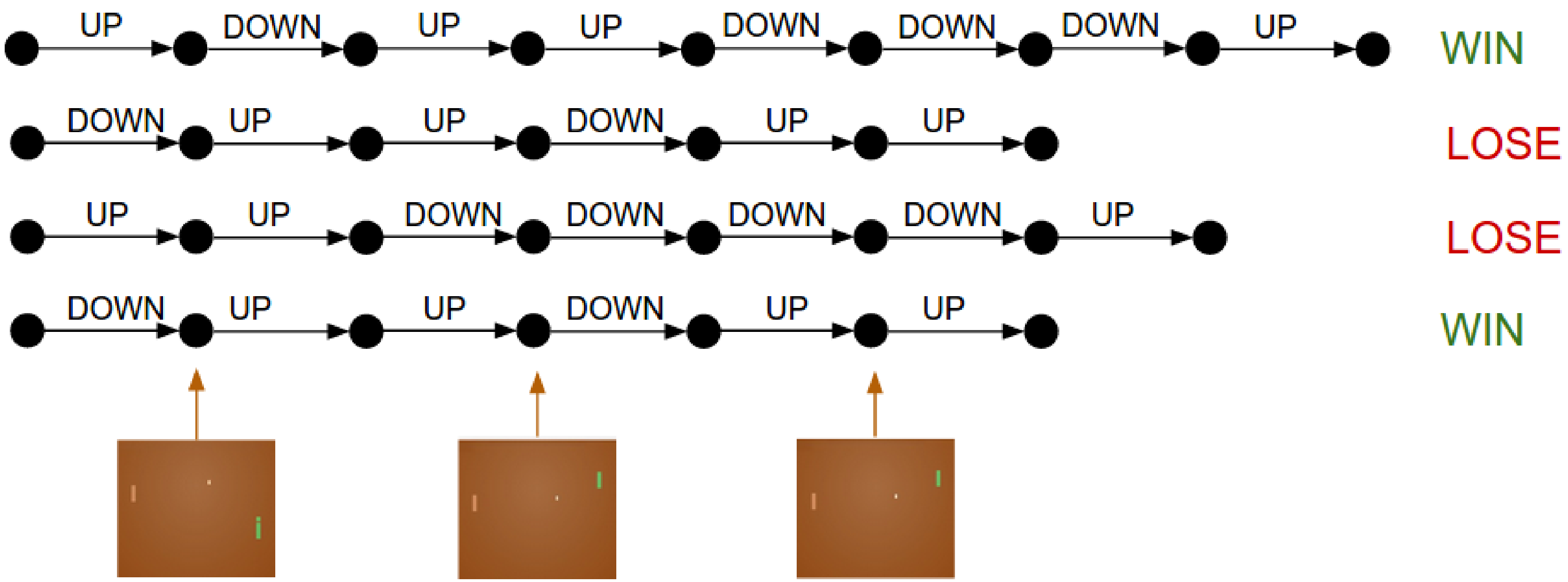



**function REINFORCE**
    Initialise $\theta$ arbitrarily
    **for** each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
        **for** $t = 1$ to $T - 1$ **do**
            $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) \boldsymbol{g_t}$
        **end for**
    **end for**
    **return** $\theta$
**end function**

Guaranteed by the Policy Gradient Theorem to eventually converge on the optimal policy!
(Marbach and Tsitskilis 1998, Sutton 200)

http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf

# Intuition for Pong



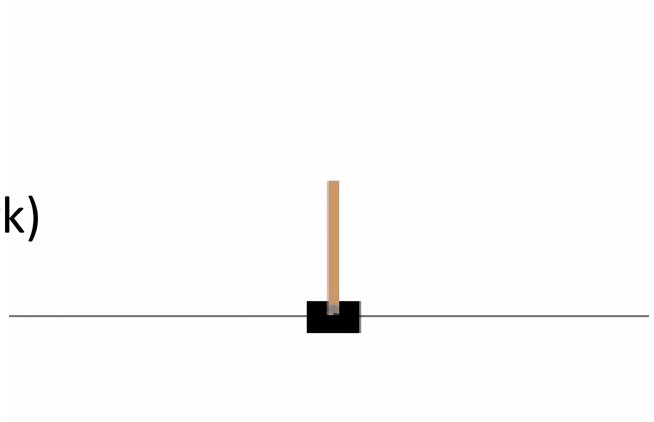

For each episode we **WIN**: increase probability of the actions a little bit in those states
For each episode we **LOSE**: decrease probability of the actions a little bit in those states

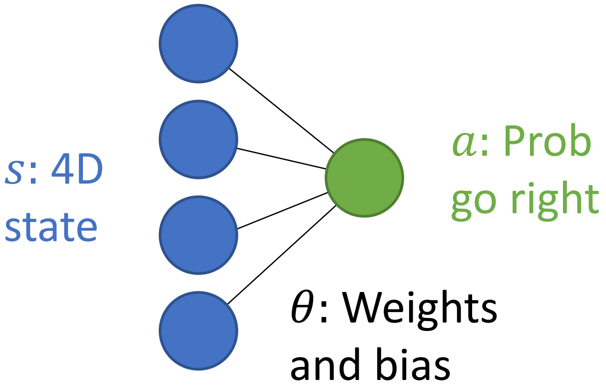
http://karpathy.github.io/2016/05/31/rl/

# Implementation (from scratch)

CartPole task

Can solve this with a logistic regression classifier (aka a single neuron network)

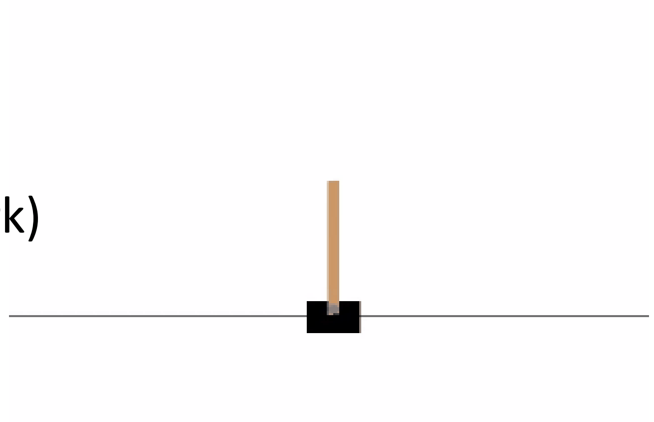




# Implementation (from scratch)

CartPole task

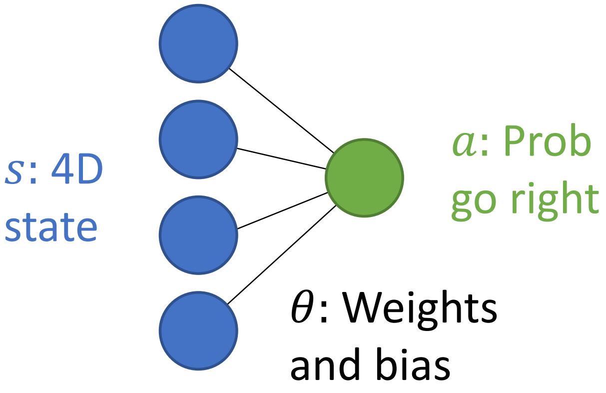
Can solve this with a logistic regression classifier (aka a single neuron network)



```
class ReinforceBinaryLR(RLAgent):

    def build_model(self):
        # initialize weights
        self.weights = np.random.randn(self.state_size[0])
        self.bias = np.random.randn(1)

    def act(self, state):
        # sample action using output probabilities of classifier
        logit = np.dot(self.weights, state) + self.bias
        p_action = 1/(1 + np.exp(-logit))
        action = 1 if p_action > np.random.rand() else 0
        return action
```



# Implementation (from scratch)

1) Run the policy for 1+ episodes

```
env = gym.make('CartPole-v0') # load environment
agent = ReinforceBinaryLR(env.observation_space.shape, env.action_space.n) # create agent

for episode in range(num_episodes):
    state = env.reset() # restart environment
    done = False

    while not done:
        action = agent.act(state)  # choose action
        next_state, reward, done, info = env.step(action) # apply action to env
        agent.remember(state, action, reward, done) # store experience in memory
        state = next_state

    agent.train() # train after each episode (or batch of episodes)
```

# Implementation (from scratch)

```
def train(self):
    # compute action probabilities
    predictions = 1/(1 + np.exp(-(np.dot(states, self.weights) + self.bias)))

    # compute gradient wrt weights using X * (Y-Yhat)
    states = np.hstack((np.ones((state_size, 1)), states)) # append ones for bias
    gradient = states * (actions-predictions)
    gradient *= returns # but weighted by the returns
    gradient = gradient.sum(axis=0)

    # move up the gradient
    self.bias    += self.learning_rate*gradient[0]
    self.weights += self.learning_rate*gradient[1:]
```

2) Estimate the return
$g_t = \sum_{\tau=t}^{T} \gamma^{t-\tau} r_\tau$

3) Improve the policy
by gradient ascent

# Implementation (from scratch)

For supervised learning, the update rule for logistic regression is: $\theta \coloneqq \theta + \alpha\, x_i(y_i - y_{pred})$

For policy gradient, the update rule is now weighted by the return: $\theta \coloneqq \theta + \alpha\, s_t(a_t - a_{prob})g_t$

```
def train(self):
    # compute action probabilities
    predictions = 1/(1 + np.exp(-(np.dot(states, self.weights) + self.bias)))

    # compute gradient wrt weights using X * (Y-Yhat)
    states = np.hstack((np.ones((state_size, 1)), states)) # append ones for bias
    gradient = states * (actions-predictions)
    gradient *= returns # but weighted by the returns
    gradient = gradient.sum(axis=0)

    # move up the gradient
    self.bias    += self.learning_rate*gradient[0]
    self.weights += self.learning_rate*gradient[1:]
```

2) Estimate the return
$g_t = \sum_{\tau=t}^{T} \gamma^{t-\tau} r_\tau$

3) Improve the policy
by gradient ascent

# Implementation (from scratch)

For supervised learning, the update rule for logistic regression is: $\theta \coloneqq \theta + \alpha \boxed{x_i (y_i - y_{pred})}$

For policy gradient, the update rule is now weighted by the return: $\theta \coloneqq \theta + \alpha\, s_t (a_t - a_{prob}) g_t$

```
def train(self):
    # compute action probabilities
    predictions = 1/(1 + np.exp(-(np.dot(states, self.weights) + self.bias)))

    # compute gradient wrt weights using X * (Y-Yhat)
    states = np.hstack((np.ones((state_size, 1)), states)) # append ones for bias
    gradient = states * (actions-predictions)
    gradient *= returns # but weighted by the returns
    gradient = gradient.sum(axis=0)

    # move up the gradient
    self.bias    += self.learning_rate*gradient[0]
    self.weights += self.learning_rate*gradient[1:]
```

2) Estimate the return
$g_t = \sum_{\tau=t}^{T} \gamma^{t-\tau} r_\tau$

3) Improve the policy
by gradient ascent

# Implementation (from scratch)

For supervised learning, the update rule for logistic regression is: $\theta \coloneqq \theta + \alpha\, x_i\left(y_i - y_{pred}\right)$

For policy gradient, the update rule is now weighted by the return: $\theta \coloneqq \theta + \alpha\, \boxed{s_t\left(a_t - a_{prob}\right)g_t}$

```
def train(self):
    # compute action probabilities
    predictions = 1/(1 + np.exp(-(np.dot(states, self.weights) + self.bias)))

    # compute gradient wrt weights using X * (Y-Yhat)
    states = np.hstack((np.ones((state_size, 1)), states)) # append ones for bias
    gradient = states * (actions-predictions)
    gradient *= returns # but weighted by the returns
    gradient = gradient.sum(axis=0)

    # move up the gradient
    self.bias    += self.learning_rate*gradient[0]
    self.weights += self.learning_rate*gradient[1:]
```

2) Estimate the return
$g_t = \sum_{\tau=t}^{T} \gamma^{t-\tau} r_\tau$

3) Improve the policy by gradient ascent

# Demo

## Problems with REINFORCE: it is inefficient!

- Main problem: **high variance**
  - Extremely, extremely noisy compared to supervised learning
  - Input data (samples from episodes) is highly correlated
- Ways to reduce variance
  - Use a **baseline** -- subtract rewards by baseline value
  - Use a **critic** – learn a value function → lower variance but high bias
  - Use **smarter step sizes** – e.g. natural gradient, TRPO, **PPO**
- Inefficient exploration

**What if one learning step makes your policy change dramatically (e.g., 500X)** ?

Too small step size: **very slow training process**
Too high step size, **too much variability in training.**

# Any Solution?

## Trust Region Policy Optimization (TRPO)

- A family of policy optimization methods
- Avoids large policy change by controlling parameter updates
- Enforcing parameter updates by a constraint on the KL divergence between $\pi_{\theta_{old}}(a_t|s_t)$ and $\pi_\theta(a_t|s_t)$
- Computationally intensive

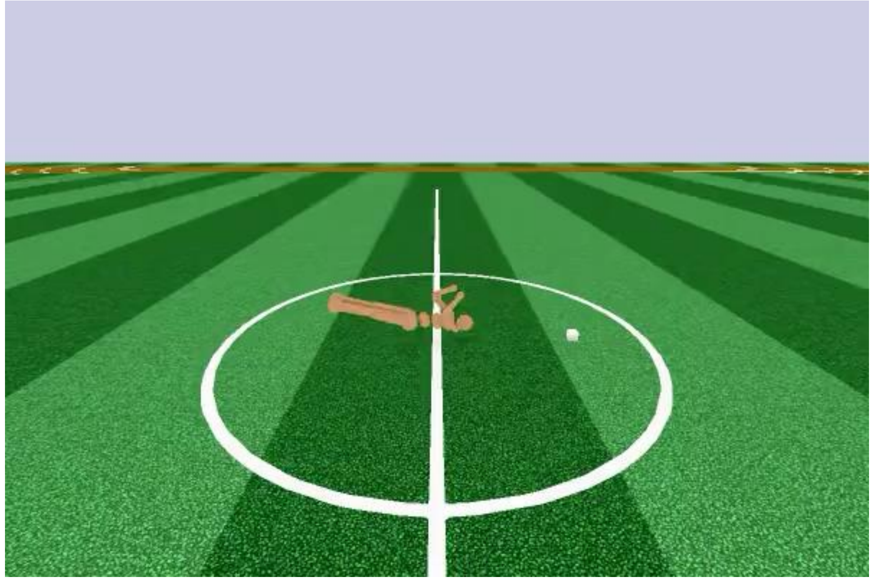
## Proximal Policy Optimization (PPO)

- A family of policy optimization methods
- Multiple epochs of stochastic gradient ascent for each policy update
- Same concept as well as stability and reliability of TRPO but simpler implementation
- A clipping term to the objective $[1-\epsilon, 1+\epsilon]$

**Clipped Surrogate Objective** → **Multiple epochs of stochastic gradient ascent for policy update**

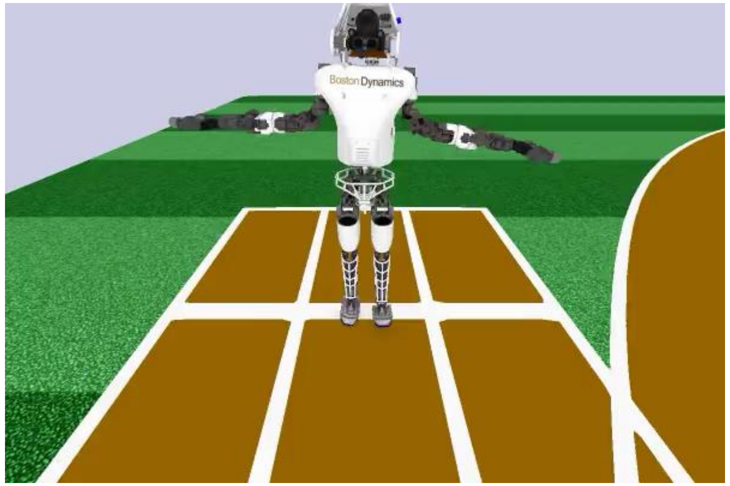
arXiv:1707.06347

# Any Solution?



*"learning to walk, run, turn, use its momentum to recover from minor hits, and how to stand up from the ground when it is knocked over."*

*"flexible movement policies that let them improvise turns and tilts as they head towards a target location"*

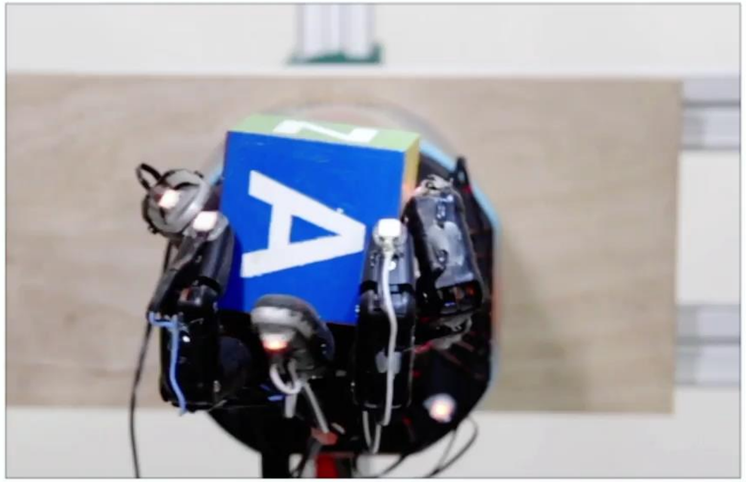



*"the 'Atlas' model from Boston Dynamics shown below; the model has 30 distinct joints"*

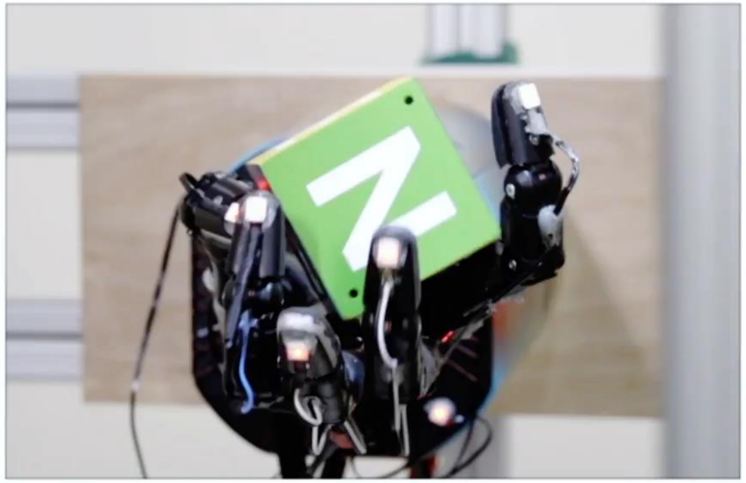
**Clipped Surrogate Objective** → **Multiple epochs of stochastic gradient ascent for policy update**

https://blog.openai.com/openai-baselines-ppo/

## Any Solution?



FINGER PIVOTING



SLIDING

FINGER GAITING

https://blog.openai.com/learning-dexterity/

# Clipped Surrogate Objective

## Objective Function

- Improve training stability by limiting the change you make to your policy at each step

$$\underbrace{L^{PG}(\theta)}_{\text{Policy Loss}} = \underbrace{E_t}_{\text{Expected}}[\underbrace{\log \pi_\theta(a_t|s_t)}_{\text{log probability of taking that action at that state}} * \underbrace{A_t}_{\text{Advantage}}]$$

Policy Loss

Expected

log probability of taking that action at that state

Advantage if A>0, this action is better than the other action possible at that state

**REINFORCE**

Gradient ascent step on $L^{PG}(\theta)$ $w.r.$ to network parameters will promote actions led to higher reward

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta\,old}(a_t|s_t)}$$

$r_t(\theta) > 1$ Action is more probable for current policy than old one

$0 < r_t(\theta) < 1$ Action is less probable for current policy than old one

arXiv:1707.06347

# Clipped Surrogate Objective

## Trust Region Policy Optimization (TRPO)

- Improve training stability by limiting the change you make to your policy at each step

$$L^{TRPO}(\theta) = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}\hat{A}_t\right] = \hat{\mathbb{E}}_t\left[r_t(\theta)\hat{A}_t\right]$$ **TRPO**

**What if the action is much more probable (e.g., 500X) for your current policy?**

Enforcing parameter updates by a constraint on the KL divergence between $\pi_{\theta_{old}}(a_t|s_t)$ and $\pi_\theta(a_t|s_t)$

**Can we instead build these properties into the objective function? Yes, PPO**

arXiv:1707.06347

# Clipped Surrogate Objective

## Proximal Policy Optimization (PPO)

- Clipped Surrogate Objective

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\underbrace{\min\left(\overbrace{r_t(\theta)\hat{A}_t}^{\text{same objective from before}}, \overbrace{\text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t}^{\text{same, but r}(\theta)\text{ is clipped between (1 - e, 1 + e)}}\right)}_{\text{min of the same objective from before and the clipped one}}\right]$$

$$\epsilon = 0.2\ , Clip\colon\ (0.8, 1.2)$$

```
def clipped_objective(new_log_probs, old_log_probs, advs, epsilon):
    """
    Compute the component-wise clipped PPO objective.
    """
    prob_ratio = tf.exp(new_log_probs - old_log_probs)
    clipped_ratio = tf.clip_by_value(prob_ratio, 1-epsilon, 1+epsilon)
    return tf.minimum(advs*clipped_ratio, advs*prob_ratio)
```

arXiv:1707.06347

# Clipped Surrogate Objective

## Proximal Policy Optimization (PPO)

- Clipped Surrogate Objective

same objective from before

same, but r(θ) is clipped between (1 - e, 1 + e)

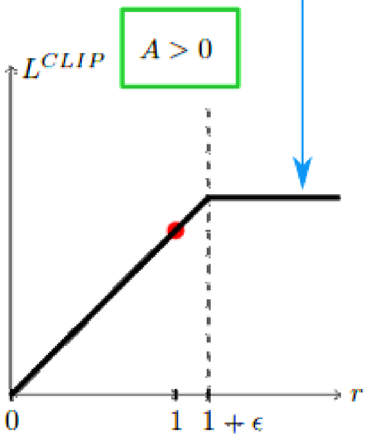
$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right]$$

min of the same objective from before and the clipped one

$$\epsilon = 0.2\ , Clip:\ (0.8, 1.2)$$

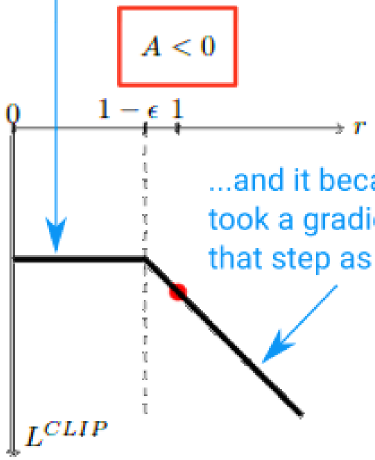
If the action was **good....**

...and it became **more probable** the last time you took a gradient step, don't keep updating it too far or else the policy might get worse




If the action was **bad....**

...and it became **less probable**, don't keep making it too much less probable or else the policy might get worse




...and it became **more probable** last time you took a gradient step, then you are free to undo that step as much as you want

**Surrogate Function vs. Probability Ratio $r$**

Why the minimum of clipped and unclipped terms?

arXiv:1707.06347

## Multiple Epochs for Policy Updating

### Proximal Policy Optimization (PPO)

- Multiple epochs of gradient ascent on samples avoiding large policy updates
- Extract more from your data and improving sample efficiency
- Parallel actors parts does not work with VPG properly

**Algorithm 1** PPO, Actor-Critic Style $K = 3 - 15, M = 64 - 4096, T = 128 - 2048$

**for** iteration=$1, 2, \ldots$ **do**
  **for** actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  **end for**
  Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
**end for**

Sample the environment with $\pi_{\theta_{old}}$

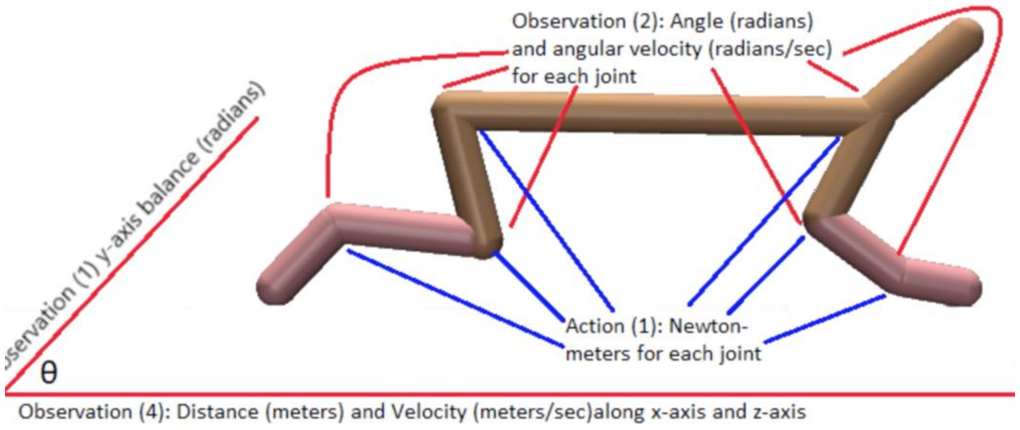
Start the optimization and objective will start hitting the clipping limits

arXiv:1707.06347

# VPG vs. TRPO vs. PPO

VPG with Learning Rate 3e-4

1. PPO → 2. TRPO → 3. VPG



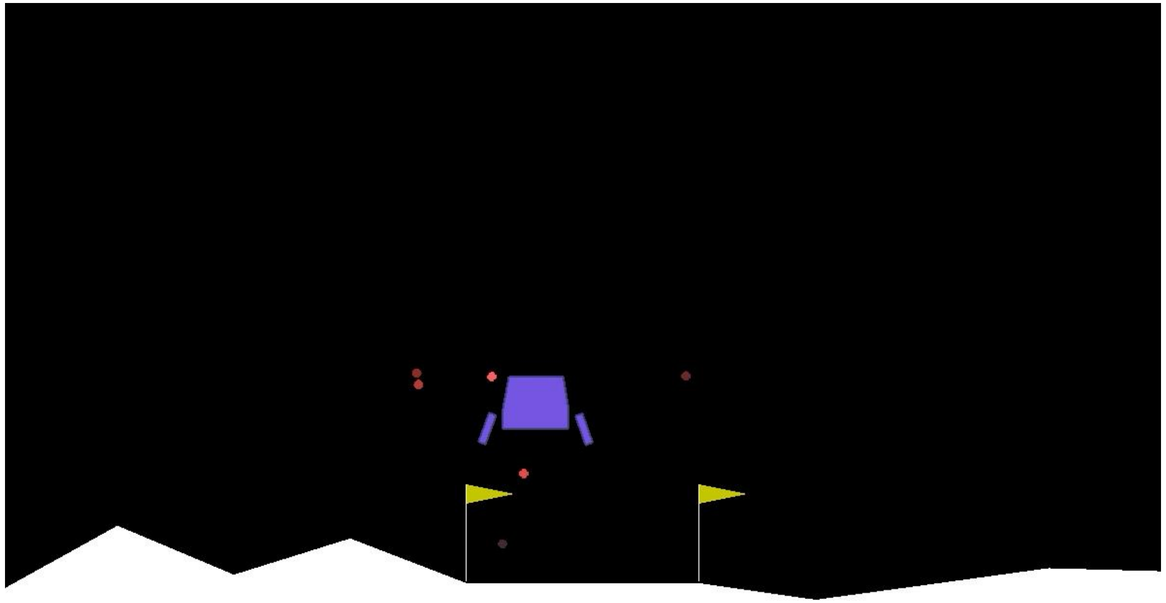

TRPO with Learning Rate 3e-4

PPO with Learning Rate 1e-4

https://youtu.be/VK2rXLoEtW8

## **Example with PPO -** LunarLander-v2

“Spinning Up is an educational resource produced by OpenAI that makes it easier to learn about deep reinforcement learning (deep RL).”



Spinning Up requires Python3, OpenAI Gym, and

```
python -m spinup.run ppo --hid "[32,32]" --env LunarLander-v2 --exp_name lunarTest --gamma 0.999
python -m spinup.run test_policy data/lundarData
```

- Landing pad is always at coordinates (0,0).
- **Coordinates** are the first two numbers in **state vector**.
- **Reward** for **moving from the top of the screen to landing pad** and **zero speed** is about 100-140 points.
- If lander moves away from landing pad it loses reward back.
- Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points.
- Each leg ground contact is +10.
- Firing main engine is -0.3 points each frame.
- Landing outside landing pad is possible.
- Fuel is infinite, so an agent can learn to fly and then land on its first attempt.
- Four discrete **actions** available:
  - do nothing
  - fire left orientation engine
  - fire main engine
  - fire right orientation engine.

https://github.com/openai/gym/wiki/Leaderboard

https://spinningup.openai.com

# References


**Learning material**

Great blog post intro by Andrej Karpathy: http://karpathy.github.io/2016/05/31/rl/

Great course material:

- Reinforcement Learning at UCL by David Silver: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html
- Deep RL at Berkeley by Sergey Levine: http://rail.eecs.berkeley.edu/deeprlcourse/

Best intro book: Reinforcement Learning by Sutton & Barto (free pdf): http://incompleteideas.net/book/the-book.html

Great YouTube explanations by Arxiv Insights: https://www.youtube.com/channel/UCNIkB2IeJ-6AmZv7bQ1oBYg

**More about PPO**

- John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov. Proximal Policy Optimization Algorithms, arXiv:1707.06347
- Youtube video on PPO: https://www.youtube.com/watch?v=5P7I-xPq8u8&t=48s
- RL — Proximal Policy Optimization (PPO) Explained [https://goo.gl/dH2Uwn]

**Code bases**

- OpenAI Spinning Up: https://spinningup.openai.com/en/latest/
  - See also Baselines https://github.com/openai/baselines)
- garage/rllab (Berkeley): https://github.com/rlworkgroup/garage
- Deep RL course with ipynb examples: https://github.com/simoninithomas/Deep_reinforcement_learning_Course
- Minimal and clean code examples: https://github.com/rlcode/reinforcement-learning