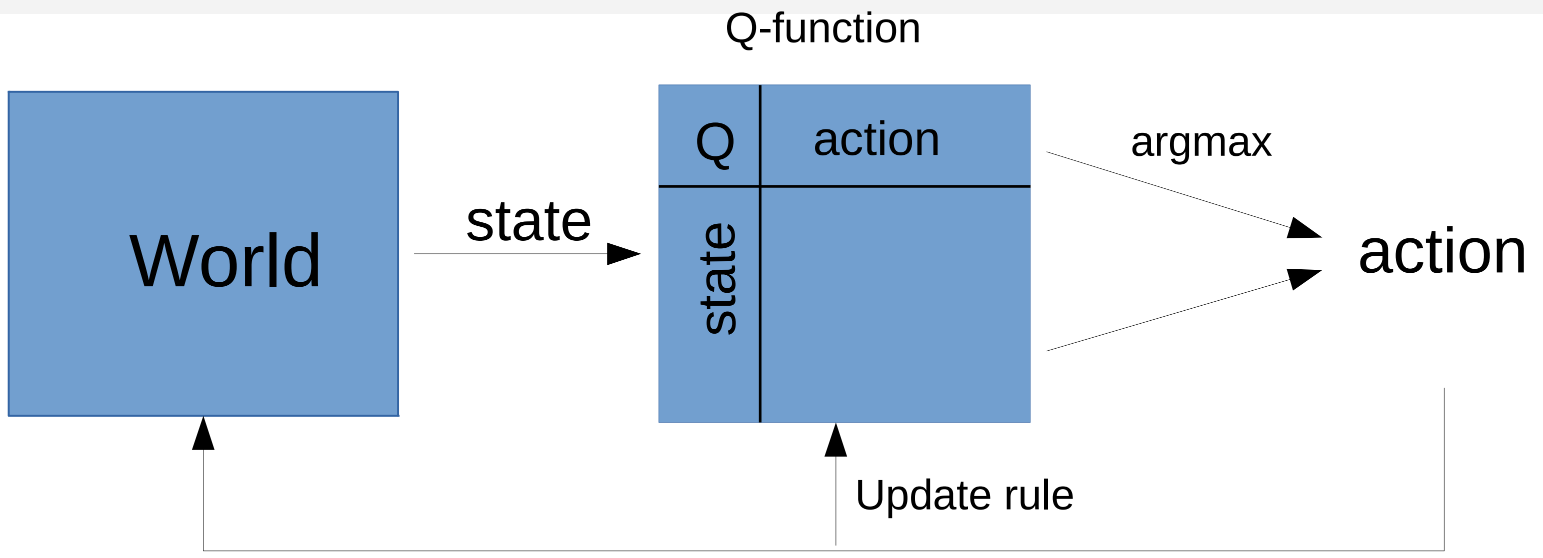
Deep RL

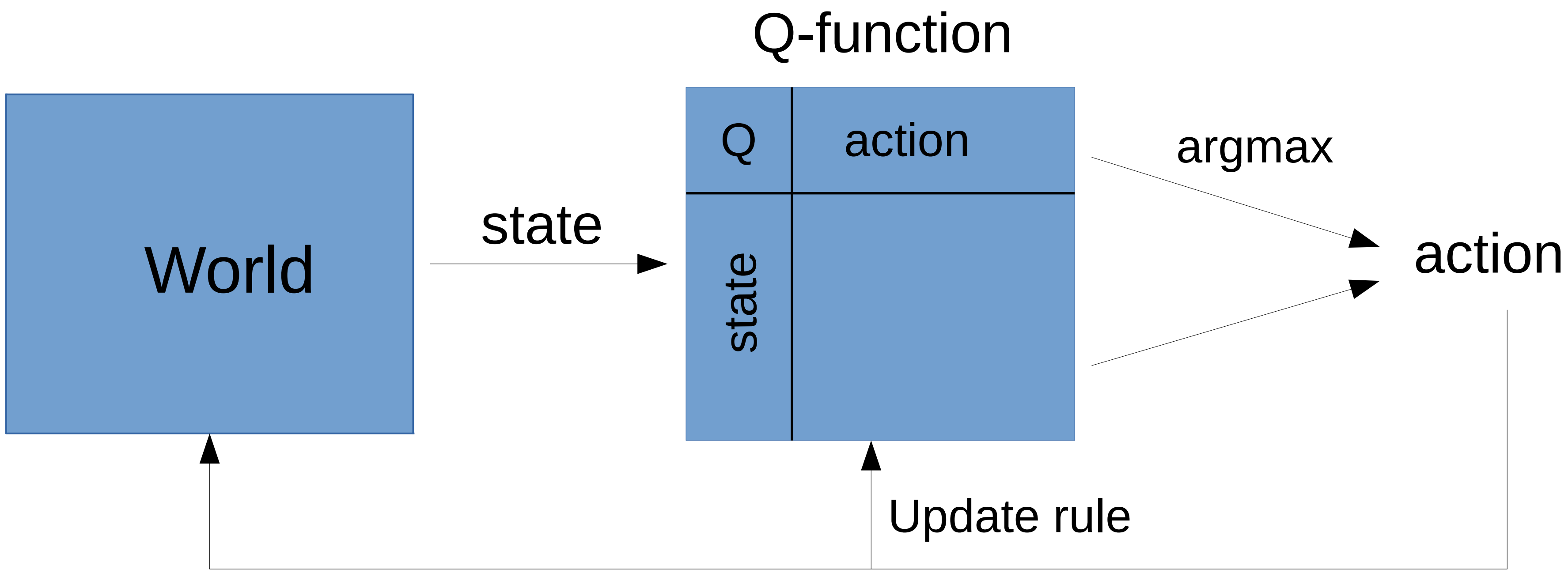
Robert Platt
Northeastern University

Q-learning

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

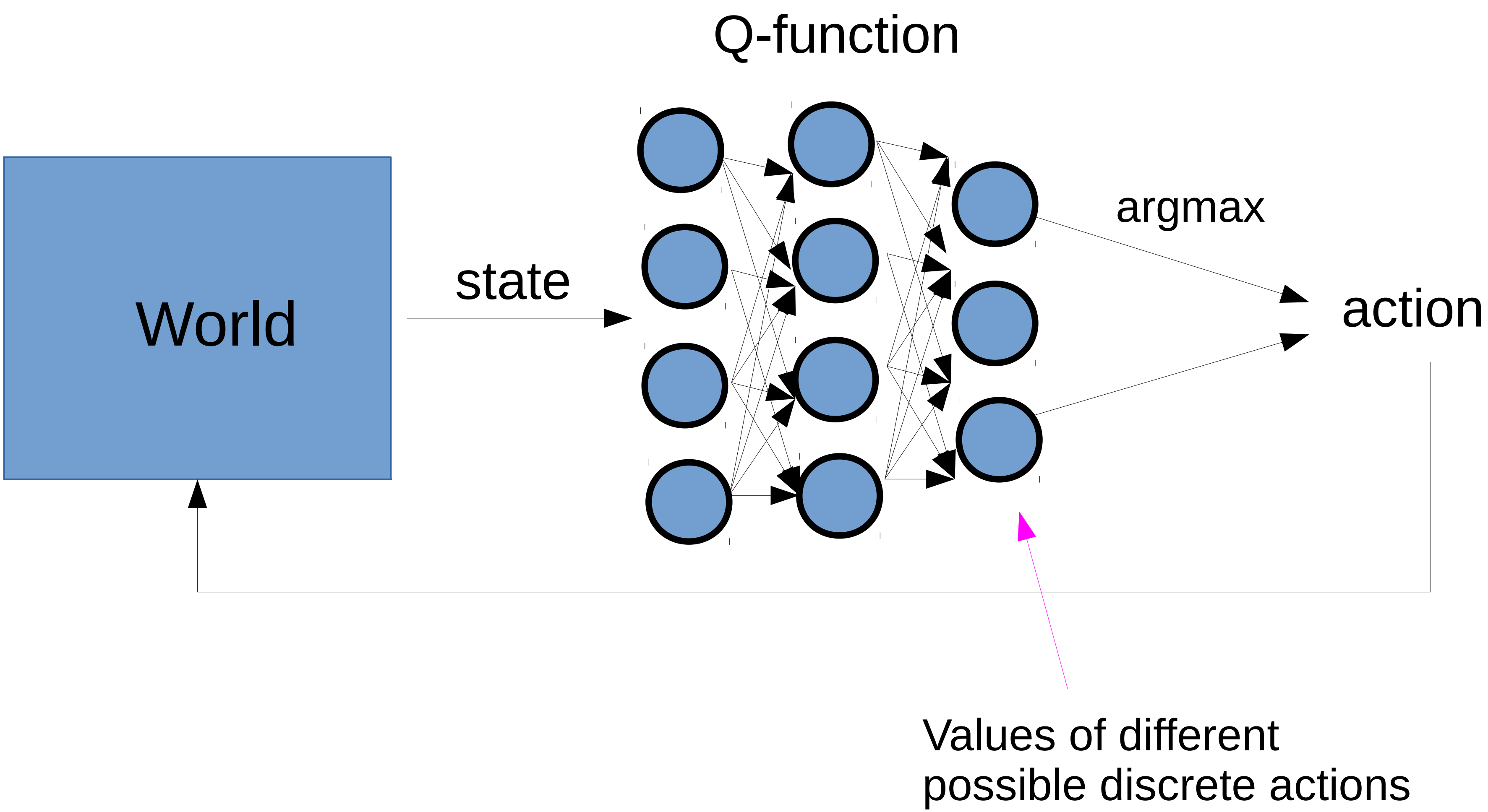


Q-learning

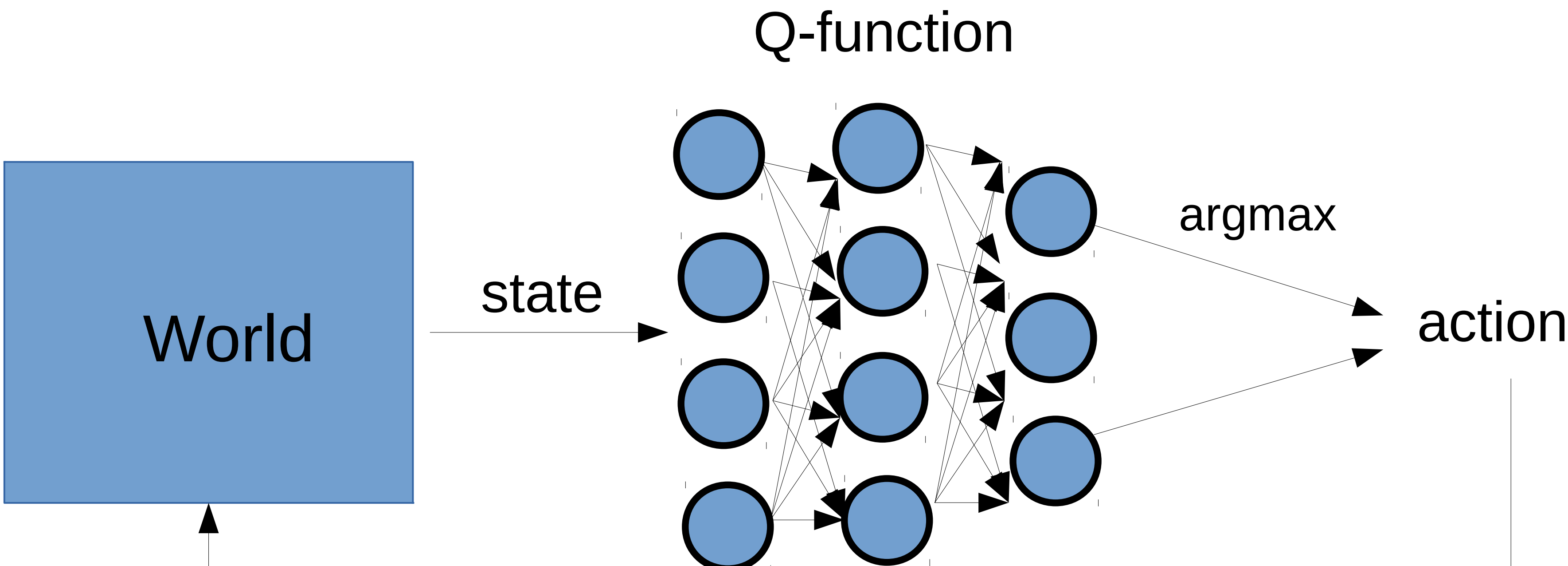


Q-function
World
state
Q
action
state
argmax
action
Update rule

Deep Q-learning (DQN)

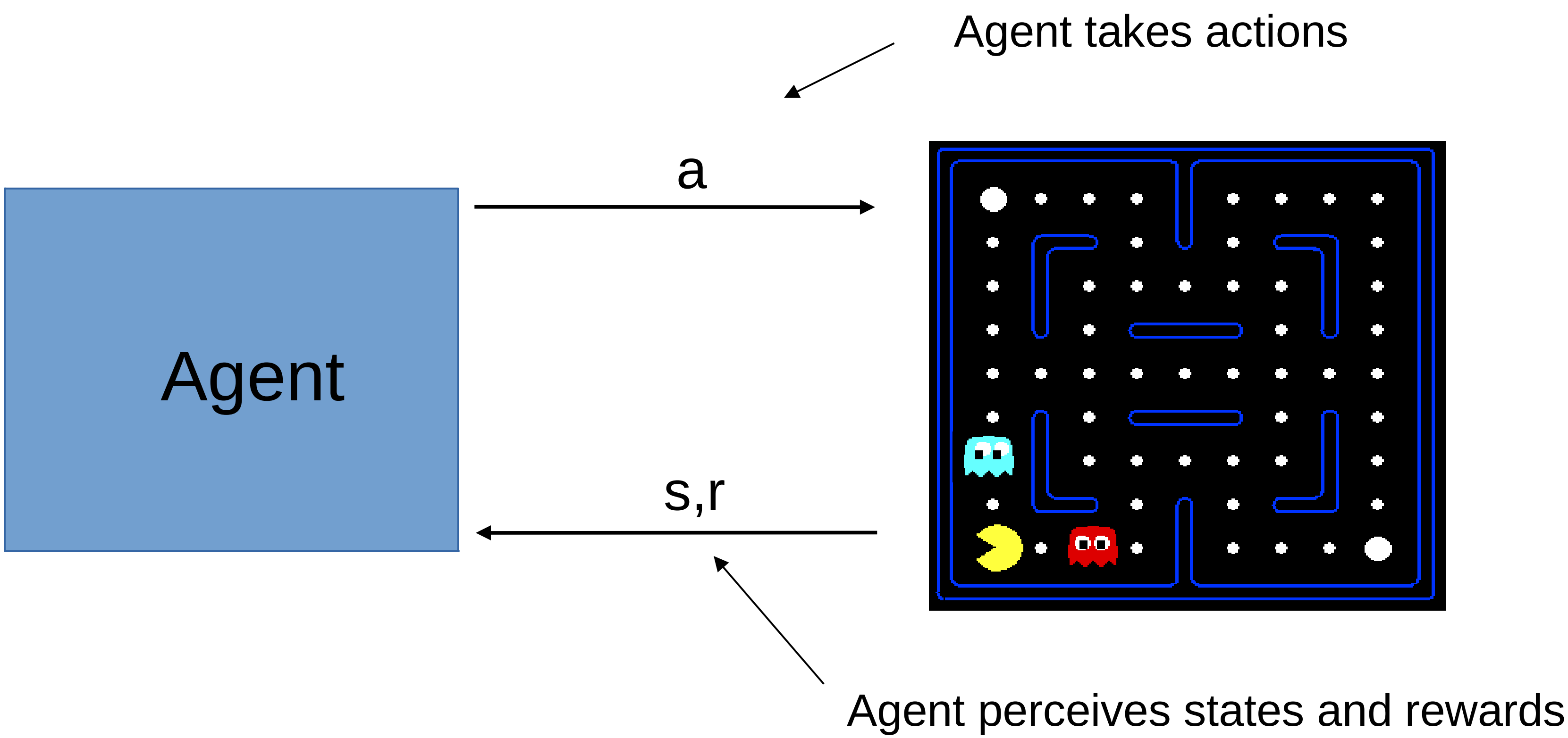


Deep Q-learning (DQN)



But, why would we want to do this?

Where does "state" come from?



Earlier, we dodged this question: "it's part of the MDP problem statement"

But, that's a cop out. How do we get state?

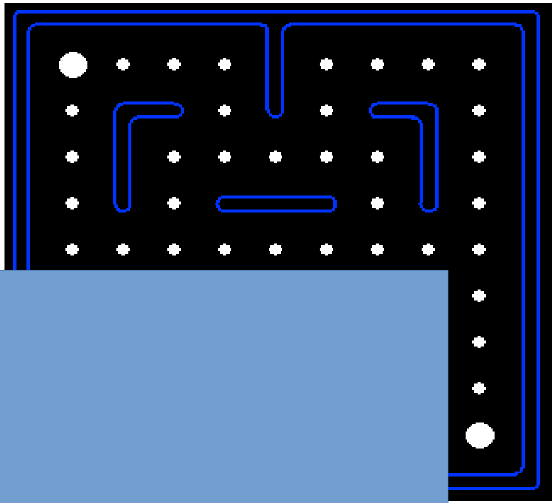
Typically can't use "raw" sensor data as state w/ a tabular Q-function
– it's too big (e.g. pacman has something like 2^(num pellets) + … states)

Where does “state” come from?

Agent takes actions

a

Agent



nd rewards

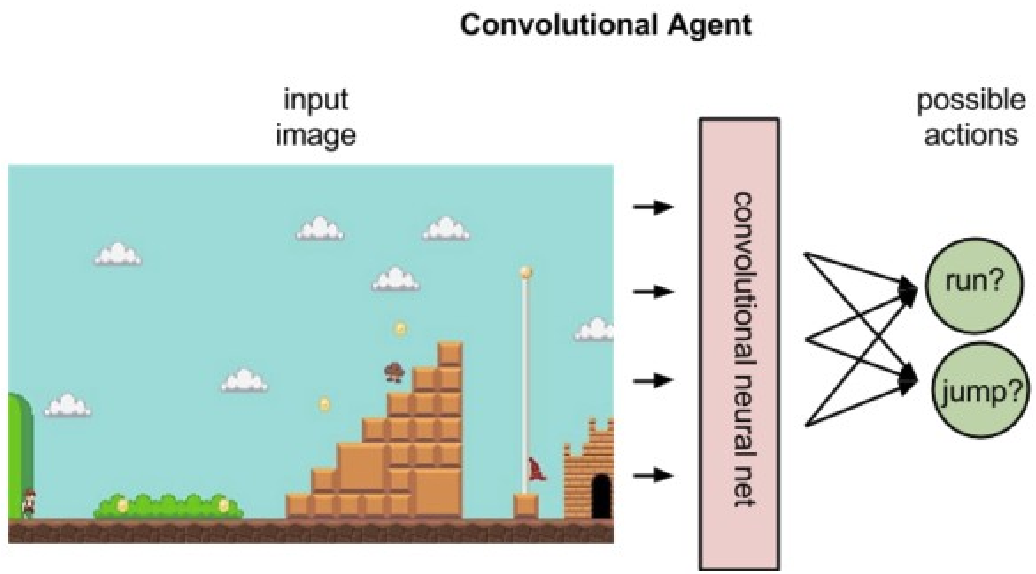
Is it possible to do RL WITHOUT
hand-coding states?

Earlier, we d atement”

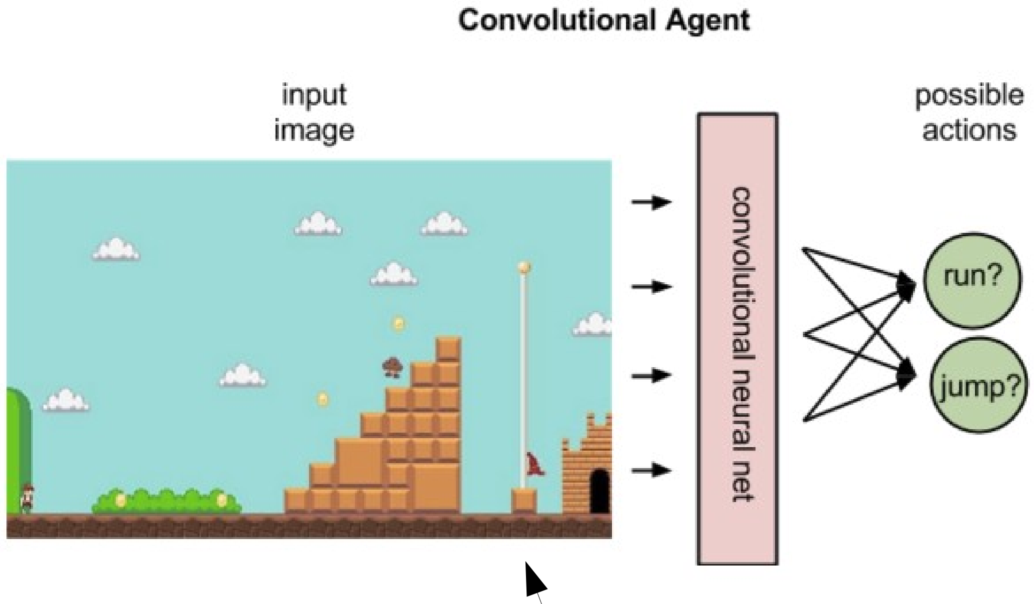
But, that’s a

Typically can’t use “raw” sensor data as state w/ a tabular Q-function
– it’s too big (e.g. pacman has something like 2^(num pellets) + … states)

DQN


Convolutional Agent
input
image
convolutional neural net
possible
actions
run?
jump?

DQN



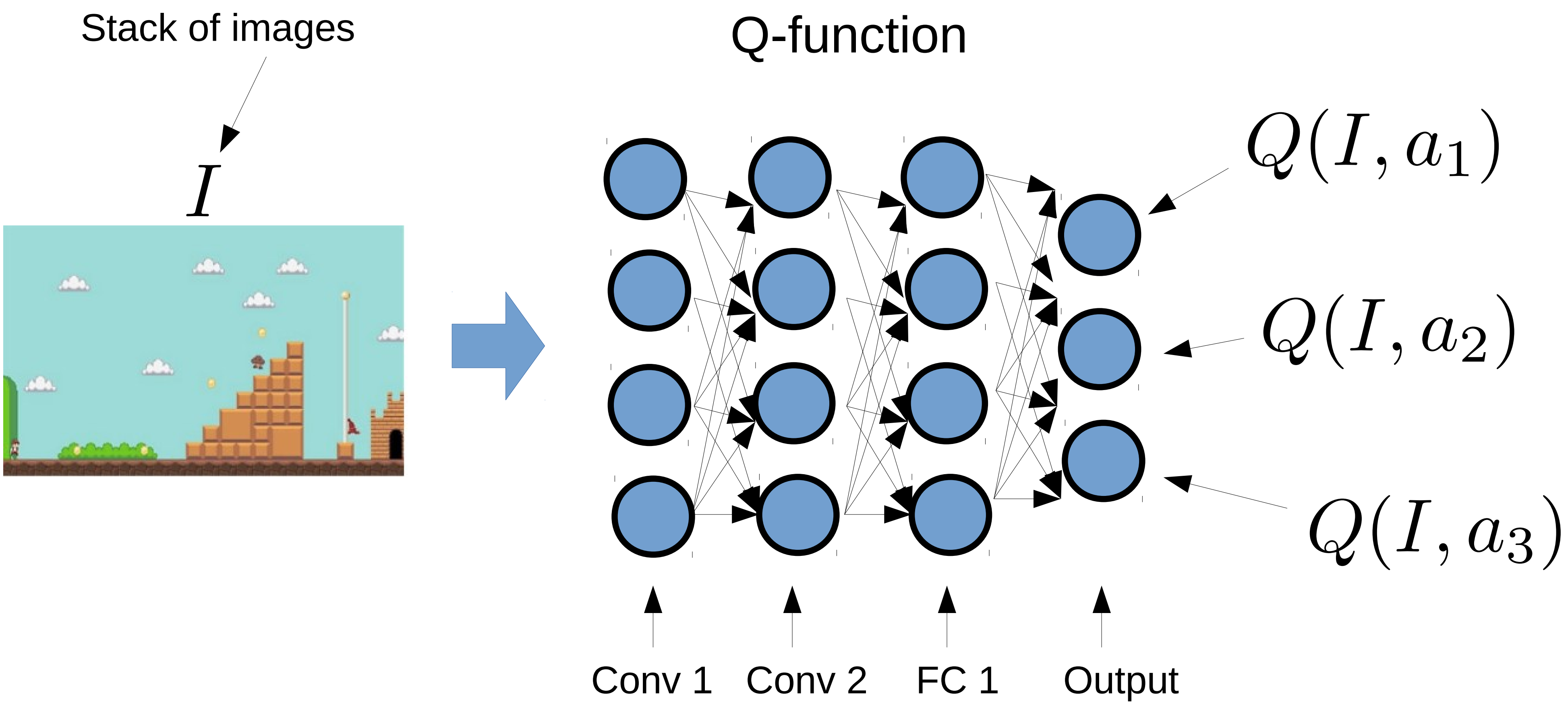
Instead of state, we have an image

– in practice, it could be a history of the *k* most recent images stacked as a single *k*-channel image

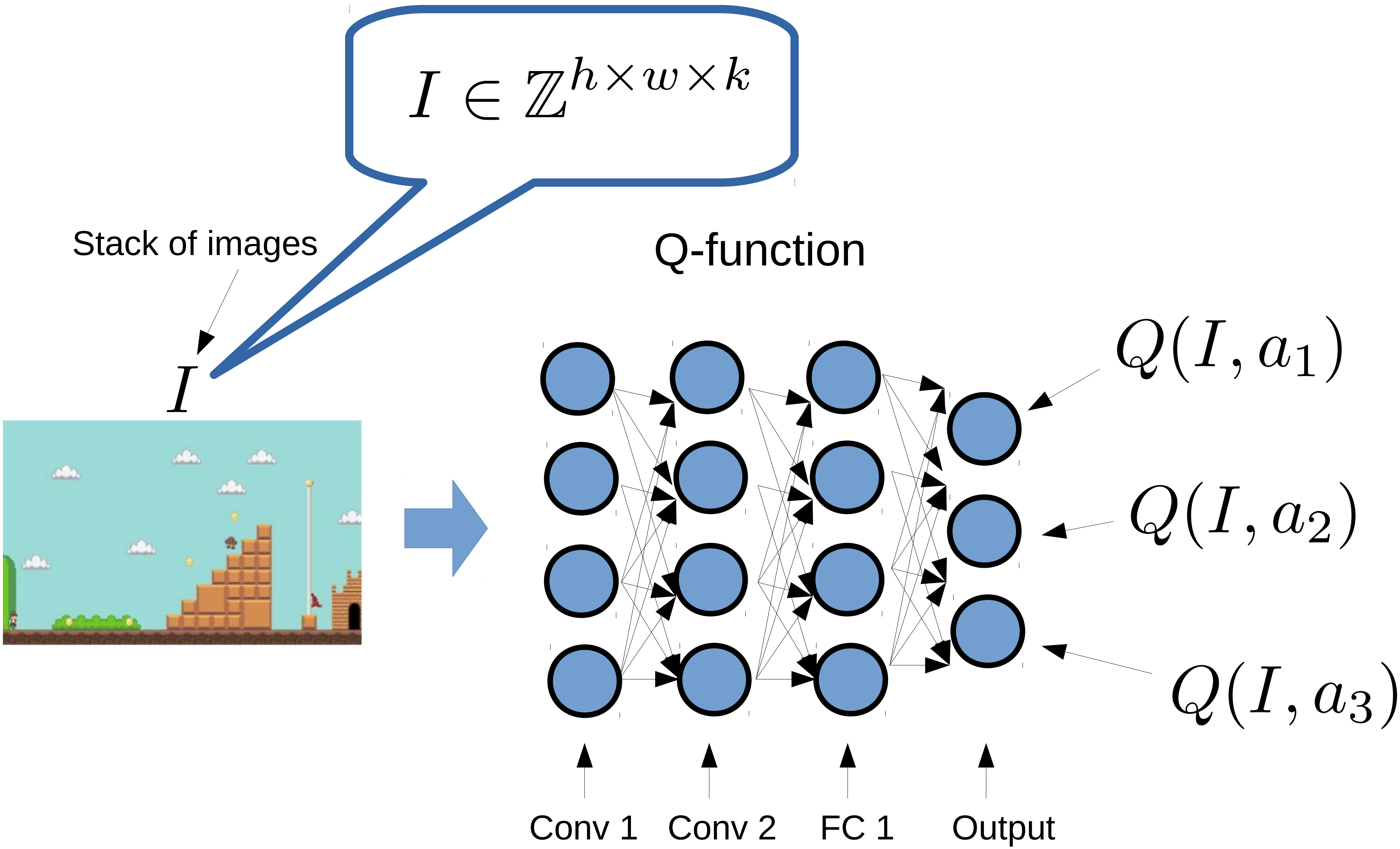
Hopefully this new image representation is Markov…

– in some domains, it might not be!

DQN

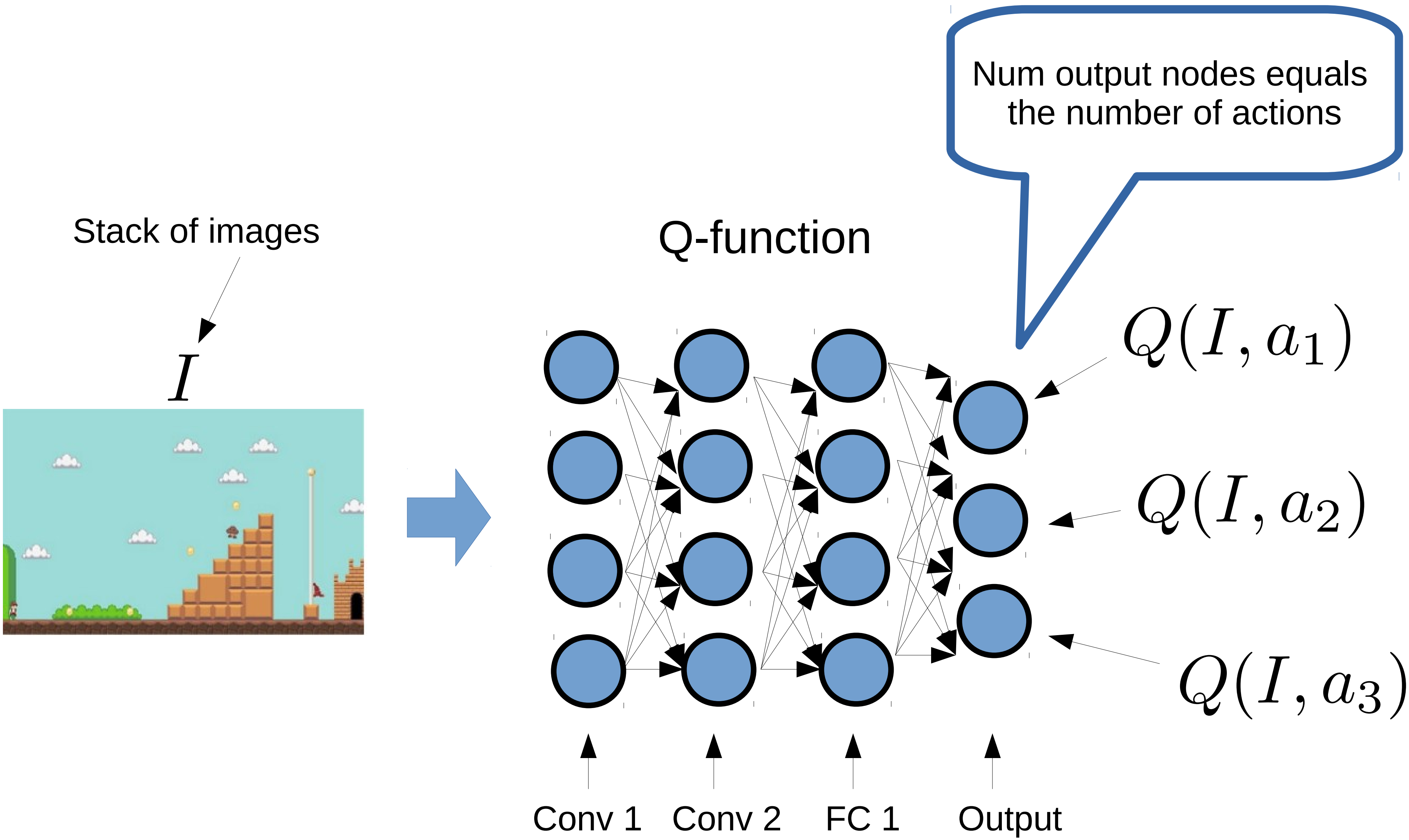

Stack of images
I
Q-function
$Q(I, a_1)$
$Q(I, a_2)$
$Q(I, a_3)$
Conv 1
Conv 2
FC 1
Output

DQN



$I \in \mathbb{Z}^{h \times w \times k}$
Stack of images
I
Q-function
$Q(I, a_1)$
$Q(I, a_2)$
$Q(I, a_3)$
Conv 1
Conv 2
FC 1
Output

DQN



Num output nodes equals the number of actions
Stack of images
I
Q-function
$Q(I, a_1)$
$Q(I, a_2)$
$Q(I, a_3)$
Conv 1
Conv 2
FC 1
Output

Q-function updates in DQN

Here's the standard Q-learning update equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

Q-function updates in DQN

Here's the standard Q-learning update equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        Choose A from S using policy derived from Q (e.g., ϵ-greedy)
        Take action A, observe R, S'
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
        $S \leftarrow S'$
    until S is terminal

Q-function updates in DQN

Here's the standard Q-learning update equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

Rewriting:

$$Q(s,a) \leftarrow (1-\alpha) Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') \right]$$

Q-function updates in DQN

Here's the standard Q-learning update equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

Rewriting:

$$Q(s,a) \leftarrow (1-\alpha) Q(s,a) + \alpha \underbrace{\left[r + \gamma \max_{a'} Q(s',a') \right]}_{\text{let's call this the "target"}}$$

This equation adjusts Q(s,a) in the direction of the target

Q-function updates in DQN

Here's the standard Q-learning update equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

Rewriting:

$$Q(s,a) \leftarrow (1-\alpha) Q(s,a) + \alpha \underbrace{\left[r + \gamma \max_{a'} Q(s',a') \right]}_{\text{let's call this the "target"}}$$

This equation adjusts Q(s,a) in the direction of the target

We're going to accomplish this same thing in
a different way using neural networks...

Q-function updates in DQN

Use this loss function:

$$L(s, a, s'; w) = \frac{1}{2}\left(r + \gamma \max_{a'} Q_w(s', a') - Q_w(s, a)\right)^2$$

Q-function updates in DQN

Use this loss function:

$$L(s, a, s'; w) = \frac{1}{2}\left(r + \gamma \max_{a'} Q_w(s', a') - Q_w(s, a)\right)^2$$

Notice that Q is now parameterized by the weights, *w*

Q-function updates in DQN

Use this loss function:

I'm including the bias in the weights

$$L(s, a, s'; w) = \frac{1}{2}\left(r + \gamma \max_{a'} Q_w(s', a') - Q_w(s, a)\right)^2$$

Q-function updates in DQN

Use this loss function:

$$L(s, a, s'; w) = \frac{1}{2}\Big(\underbrace{r + \gamma \max_{a'} Q_w(s', a')}_{\text{target}} - Q_w(s, a)\Big)^2$$

Question

Use this loss function:

$$L(s,a,s';w) = \frac{1}{2}\Big(\underbrace{\overbrace{\boxed{r + \gamma \max_{a'} Q_w(s',a')}}^{\text{target}} - Q_w(s,a)}_{\text{What's this called?}}\Big)^2$$

Q-function updates in DQN

Use this loss function:

$$L(s,a,s';w) = \frac{1}{2}\left(\overbrace{r + \gamma \max_{a'} Q_w(s',a')}^{\text{target}} - Q_w(s,a)\right)^2$$

We're going to optimize this loss function using the following gradient:

$$\nabla_w L(s,a,s';w) \approx -\left(r + \gamma \max_{a'} Q_w(s',a') - Q_w(s,a)\right) \nabla_w Q_w(s,a)$$

Think-pair-share

Use this loss function:

$$L(s,a,s';w) = \frac{1}{2}\Big(\overbrace{r + \gamma \max_{a'} Q_w(s',a')}^{\text{target}} - Q_w(s,a)\Big)^2$$

We’re going to optimize this loss function using the following gradient:

$$\nabla_w L(s,a,s';w) \approx -\Big(r + \gamma \max_{a'} Q_w(s',a') - Q_w(s,a)\Big)\nabla_w Q_w(s,a)$$

What’s wrong with this?

Q-function updates in DQN

Use this loss function:

$$L(s,a,s';w) = \frac{1}{2}\Big(\underbrace{r + \gamma \max_{a'} Q_w(s',a')}_{\text{target}} - Q_w(s,a)\Big)^2$$

We're going to optimize this loss function using the following gradient:

$$\nabla_w L(s,a,s';w) \approx -\Big(r + \gamma \max_{a'} Q_w(s',a') - Q_w(s,a)\Big)\nabla_w Q_w(s,a)$$

What's wrong with this?

We call this the *semigradient* rather than the gradient

– semi-gradient descent still converges

– this is often more convenient

“Barebones” DQN

Initialize Q(*s,a;w*) with random weights
Repeat (for each episode):
    Initialize *s*
    Repeat (for each step of the episode):
        Choose *a* from *s* using policy derived from Q (e.g. e-greedy)
        Take action *a*, observe *r, s'*
        $w \leftarrow w - \alpha \nabla_w L(s, a, s'; w)$
        $s \leftarrow s'$
    Until *s* is terminal

Where:

$$\nabla_w L(s, a, s'; w) \approx -\left(r + \gamma \max_{a'} Q_w(s', a') - Q_w(s, a)\right) \nabla_w Q_w(s, a)$$

“Barebones” DQN

Initialize Q(*s,a;w*) with random weights
Repeat (for each episode):
    Initialize *s*
    Repeat (for each step of the episode):
        Choose *a* from *s* using policy derived from Q (e.g. e-greedy)
        Take action *a*, observe *r, s'*

$$\boxed{\begin{aligned} & w \leftarrow w - \alpha \nabla_w L(s, a, s'; w) \\ & s \leftarrow s' \end{aligned}}$$

    Until *s* is terminal

This is all that changed relative to standard q-learning

Where:

$$\nabla_w L(s, a, s'; w) \approx -\left(r + \gamma \max_{a'} Q_w(s', a') - Q_w(s, a)\right) \nabla_w Q_w(s, a)$$

Example: 4x4 frozen lake env

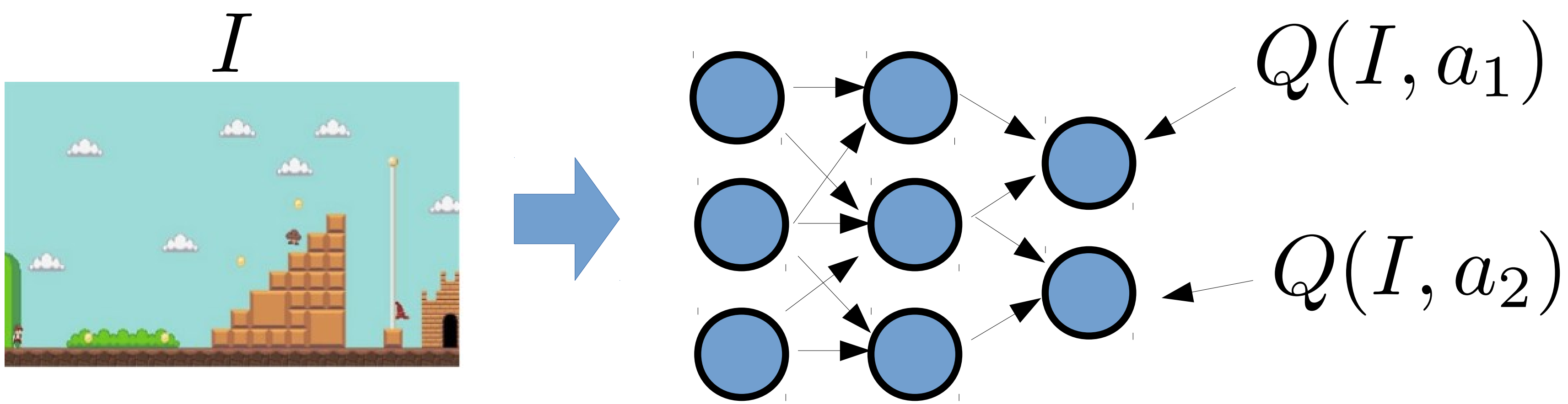
Get to the goal (G)
Don’t fall in a hole (H)

```
SFFF
FHFH
FFFH
HFFG
  (Left)
```

```
steps: 11582, episodes: 770, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.0873818397522
steps: 11609, episodes: 771, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.0872020721436
steps: 11652, episodes: 772, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.138998985291
steps: 11672, episodes: 773, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.0649240016937
steps: 11689, episodes: 774, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.0546970367432
steps: 11697, episodes: 775, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.0260739326477
steps: 11731, episodes: 776, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.110991954803
steps: 11773, episodes: 777, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.135339975357
steps: 11798, episodes: 778, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.0810689926147
steps: 11818, episodes: 779, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.0643260478973
steps: 11870, episodes: 780, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.169064044952
steps: 11906, episodes: 781, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.117113113403
steps: 11992, episodes: 782, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.279519796371
steps: 12064, episodes: 783, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.234206199646
steps: 12090, episodes: 784, mean 100 episode reward: 0.7, % time spent exploring: 2, time elapsed: 0.0835938453674
steps: 12137, episodes: 785, mean 100 episode reward: 0.7, % time spent exploring: 2, time elapsed: 0.150979042053
steps: 12185, episodes: 786, mean 100 episode reward: 0.7, % time spent exploring: 2, time elapsed: 0.155304908752
steps: 12245, episodes: 787, mean 100 episode reward: 0.7, % time spent exploring: 2, time elapsed: 0.194122076035
steps: 12277, episodes: 788, mean 100 episode reward: 0.7, % time spent exploring: 2, time elapsed: 0.102608919144
steps: 12293, episodes: 789, mean 100 episode reward: 0.7, % time spent exploring: 2, time elapsed: 0.0520431995392
```

Demo!

Think-pair-share



Suppose the "barebones" DQN algorithm w/ this DQN network experiences the following transition: s, a_1, s', r

Which weights in the network *could* be updated on this iteration?

$$\nabla_w L(s,a,s';w) \approx -\left(r + \gamma \max_{a'} Q_w(s',a') - Q_w(s,a)\right) \nabla_w Q_w(s,a)$$

Experience replay

Deep learning typically assumes independent, identically distributed (IID) training data

Experience replay

Deep learning typically assumes independent, identically distributed (IID) training data

But is this true in the deep RL scenario?

Initialize Q(*s,a;w*) with random weights
Repeat (for each episode):
    Initialize *s*
    Repeat (for each step of the episode):
        Choose *a* from *s* using policy derived from Q (e.g. e-greedy)
        Take action *a*, observe *r, s'*
        $w \leftarrow w - \alpha \nabla_w L(s, a, s'; w)$
        $s \leftarrow s'$
    Until *s* is terminal

Experience replay

Deep learning typically assumes independent, identically distributed (IID) training data

But is this true in the deep RL scenario?

Initialize Q(*s,a;w*) with random weights
Repeat (for each episode):
    Initialize *s*
    Repeat (for each step of the episode):
        Choose *a* from *s* using policy derived from Q (e.g. e-greedy)
        Take action *a*, observe *r*, *s'*

<u>Our solution:</u> buffer experiences and then "replay" them during training

Experience replay

Initialize Q_w with random weights
$D \leftarrow \emptyset$

Replay buffer

Repeat (for each episode):
    Initialize *s*
    Repeat (for each step of the episode):
        Choose *a* from *s* using policy derived from Q (e.g. e-greedy)
        Take action *a*, observe *r, s'*
        $D \leftarrow D \cup (s, a, s', r)$

Add this exp to buffer

        $s \leftarrow s'$
        If mod(*step*,*trainfreq*) == 0:

Train every
trainfreq steps

            sample batch *B* from *D*
            $w \leftarrow w - \alpha \nabla_w L(B; w)$

One step grad
descent WRT buffer

Experience replay

Initialize Q_w with random weights

$D \leftarrow \emptyset$

Repeat (for each episode):

  Initialize *s*

  Repeat (for each step of the episode):

    Choose *a* from *s* using policy derived from Q (e.g. e-greedy)

    Take action *a*, observe *r, s'*

    $D \leftarrow D \cup (s, a, s', r)$

    $s \leftarrow s'$

    If mod(*step*,*trainfreq*) == 0:

      sample batch *B* from *D*

      $w \leftarrow w - \alpha \nabla_w L(B; w)$

Replay buffer

Add this exp to buffer

Train every *trainfreq* steps

One step grad descent WRT buffer

Where: $\nabla_w L(B; w) \approx -\frac{1}{|B|} \sum_{(s,a,s',r) \in B} (target(s'; w) - Q_w(s, a)) \nabla_w Q_w(s, a)$

$$target(s'; w) = r + \gamma \max_{a'} Q_w(s', a')$$

Experience replay

Initialize Q_w with random weights

Buffers like this are pretty common in DL

Take ac… r, s'

$D \leftarrow D \cup ($…

$s \leftarrow s'$

If mod(*step*,*trainfreq*) == 0:

    sample batch *B* from *D*

    $w \leftarrow w - \alpha \nabla_w L(B; w)$

Add this exp to buffer

Train every *trainfreq* steps

One step grad descent WRT buffer

Where: $\nabla_w L(B; w) \approx -\frac{1}{|B|} \sum_{(s,a,s',r) \in B} \left(target(s'; w) - Q_w(s,a)\right) \nabla_w Q_w(s,a)$

$$target(s'; w) = r + \gamma \max_{a'} Q_w(s', a')$$

Think-pair-share

Initialize Q_w with random weights
$D \leftarrow \emptyset$
Repeat (for each episode):
    Initialize *s*
    Repeat (for each step of the episode):
        Choose *a* from *s* using policy derived from Q (e.g. e-greedy)
        Take action *a*, observe *r, s'*
        $D \leftarrow D \cup (s, a, s', r)$
        $s \leftarrow s'$
        If mod(*step,trainfreq*) == 0:
            sample batch *B* from *D*
            $w \leftarrow w - \alpha \nabla_w L(B; w)$

<u>What do you think are the tradeoffs between:</u>

– large replay buffer vs small replay buffer?
– large batch size vs small batch size?

With target network

Initialize Q_w, Q_{w^-} with random weights

$D \leftarrow \emptyset$

Repeat (for each episode):

  Initialize *s*

  Repeat (for each step of the episode):

    Choose *a* from *s* using policy derived from Q (e.g. e-greedy)

    Take action *a*, observe *r, s'*

    $D \leftarrow D \cup (s, a, s', r)$

    $s \leftarrow s'$

    If mod(*step,trainfreq*) == 0:

      sample batch *B* from *D*

      $w \leftarrow w - \alpha \nabla_w L(B; w, w^-)$

      if mod(*step,copyfreq*) == 0:

        $w^- \leftarrow w$

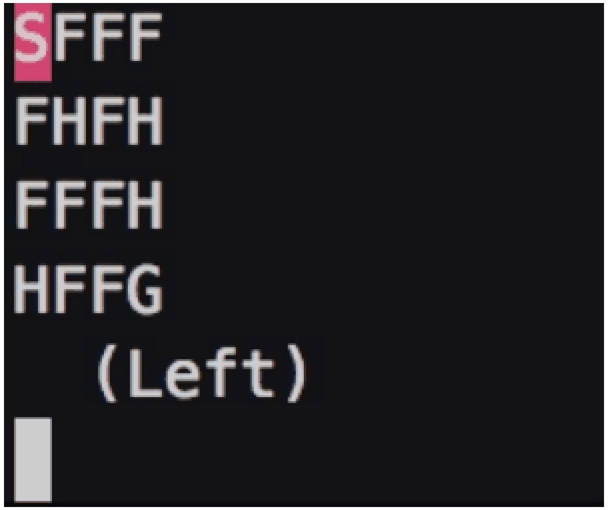
Target network helps stabilize DL – **why**?

<u>Where:</u> $\nabla_w L(B; w, w^-) \approx -\frac{1}{|B|} \sum_{(s,a,s',r) \in B} \left(target(s'; w^-) - Q_w(s,a) \right) \nabla_w Q_w(s,a)$

$$target(s'; w^-) = r + \gamma \max_{a'} Q_{w^-}(s', a')$$

Example: 4x4 frozen lake env

Get to the goal (G)
Don’t fall in a hole (H)




```
steps: 11582, episodes: 770, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.0873818397522
steps: 11609, episodes: 771, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.0872020721436
steps: 11652, episodes: 772, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.138998985291
steps: 11672, episodes: 773, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.0649240016937
steps: 11689, episodes: 774, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.0546970367432
steps: 11697, episodes: 775, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.0260739326477
steps: 11731, episodes: 776, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.110991954803
steps: 11773, episodes: 777, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.135339975357
steps: 11798, episodes: 778, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.0810689926147
steps: 11818, episodes: 779, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.0643260478973
steps: 11870, episodes: 780, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.169064044952
steps: 11906, episodes: 781, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.117113113403
steps: 11992, episodes: 782, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.279519796371
steps: 12064, episodes: 783, mean 100 episode reward: 0.6, % time spent exploring: 2, time elapsed: 0.234206199646
steps: 12090, episodes: 784, mean 100 episode reward: 0.7, % time spent exploring: 2, time elapsed: 0.0835938453674
steps: 12137, episodes: 785, mean 100 episode reward: 0.7, % time spent exploring: 2, time elapsed: 0.150979042053
steps: 12185, episodes: 786, mean 100 episode reward: 0.7, % time spent exploring: 2, time elapsed: 0.155304908752
steps: 12245, episodes: 787, mean 100 episode reward: 0.7, % time spent exploring: 2, time elapsed: 0.194122076035
steps: 12277, episodes: 788, mean 100 episode reward: 0.7, % time spent exploring: 2, time elapsed: 0.102608919144
steps: 12293, episodes: 789, mean 100 episode reward: 0.7, % time spent exploring: 2, time elapsed: 0.0520431995392
```

Demo!

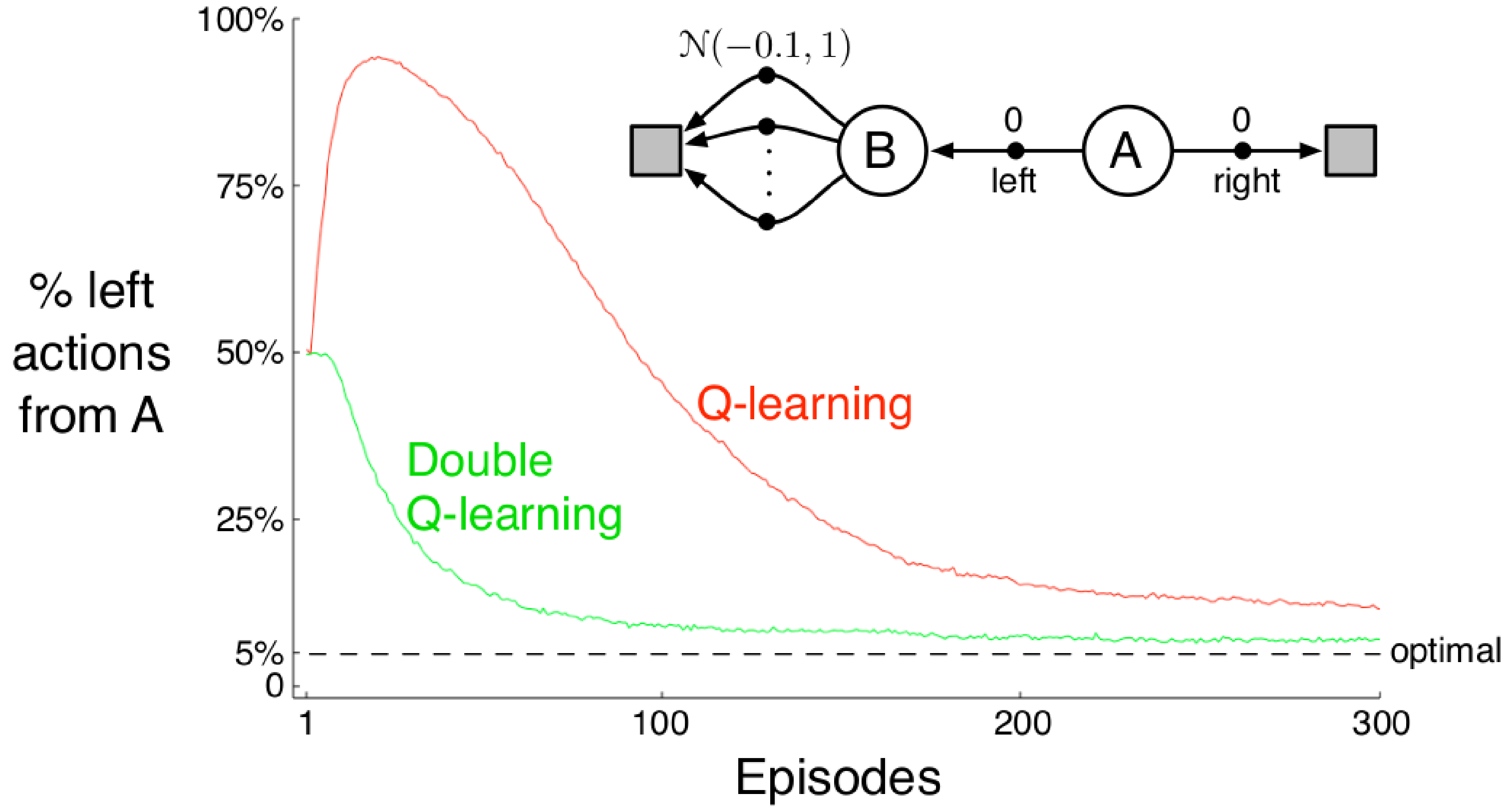
Comparison: replay vs no replay

	Replay Fixed-Q	Replay Q-learning	No replay Fixed-Q	No replay Q-learning
Breakout	316.81	240.73	10.16	3.17
Enduro	1006.3	831.25	141.89	29.1
River Raid	7446.62	4102.81	2867.66	1453.02
Seaquest	2894.4	822.55	1003	275.81
Space Invaders	1088.94	826.33	373.22	301.99

(Avg final score achieved)

Double DQN

Recall the problem of maximization bias:



Double DQN

Recall the problem of maximization bias:

Our solution from the TD lecture:

Initialize $Q_1(s,a)$ and $Q_2(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily
Initialize $Q_1(\textit{terminal-state}, \cdot) = Q_2(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q_1 and Q_2 (e.g., ε-greedy in $Q_1 + Q_2$)
    Take action A, observe R, S'
    With 0.5 probabilility:
      $$Q_1(S,A) \leftarrow Q_1(S,A) + \alpha\Big(R + \gamma Q_2\big(S', \arg\max_a Q_1(S',a)\big) - Q_1(S,A)\Big)$$
    else:
      $$Q_2(S,A) \leftarrow Q_2(S,A) + \alpha\Big(R + \gamma Q_1\big(S', \arg\max_a Q_2(S',a)\big) - Q_2(S,A)\Big)$$
    $S \leftarrow S'$
  until S is terminal

Can we adapt this to the DQN setting?

Double DQN

Initialize Q_w, Q_{w^-} with random weights
$D \leftarrow \emptyset$
Repeat (for each episode):
  Initialize *s*
  Repeat (for each step of the episode):
    Choose *a* from *s* using policy derived from Q (e.g. e-greedy)
    Take action *a*, observe *r, s'*
    $D \leftarrow D \cup (s, a, s', r)$
    $s \leftarrow s'$
    If mod(*step,trainfreq*) == 0:
      sample batch *B* from *D*
      $w \leftarrow w - \alpha \nabla_w L(B; w, w^-)$
      if mod(*step,copyfreq*) == 0:
        $w^- \leftarrow w$

<u>Where:</u> $$\nabla_w L(B; w, w^-) \approx -\frac{1}{|B|} \sum_{(s,a,s',r) \in B} \big(target(s', a'; w, w^-) - Q_w(s,a)\big) \nabla_w Q_w(s,a)$$

$$target(s', a'; w, w^-) = r + \gamma Q_{w^-}(s', \arg\max_{a'} Q_w(s', a'))$$

Think-pair-share

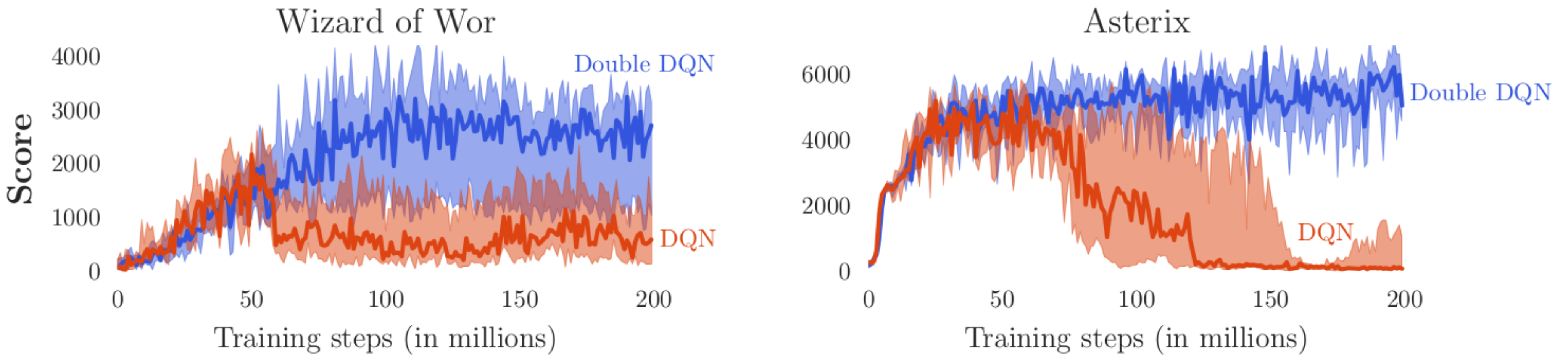
Initialize Q_w, Q_{w^-} with random weights
$D \leftarrow \emptyset$
Repeat (for each episode):
    Initialize *s*
    Repeat (for each step of the episode):
        Choose *a* from *s* using policy derived from Q (e.g. e-greedy)
        Take action *a*, observe *r, s'*
        $D \leftarrow D \cup (s, a, s', r)$
        $s \leftarrow s'$
        If mod(*step,trainfreq*) == 0:
            sample batch *B* from *D*
            $w \leftarrow w - \alpha \nabla_w L(B; w, w^-)$
            if mod(*step,copyfreq*) == 0:
                $w^- \leftarrow w$

1. In what sense is this double q-learning?
2. What are the pros/cons vs earlier version of double-Q?
3. Why not convert the original double-Q algorithm into a deep version?

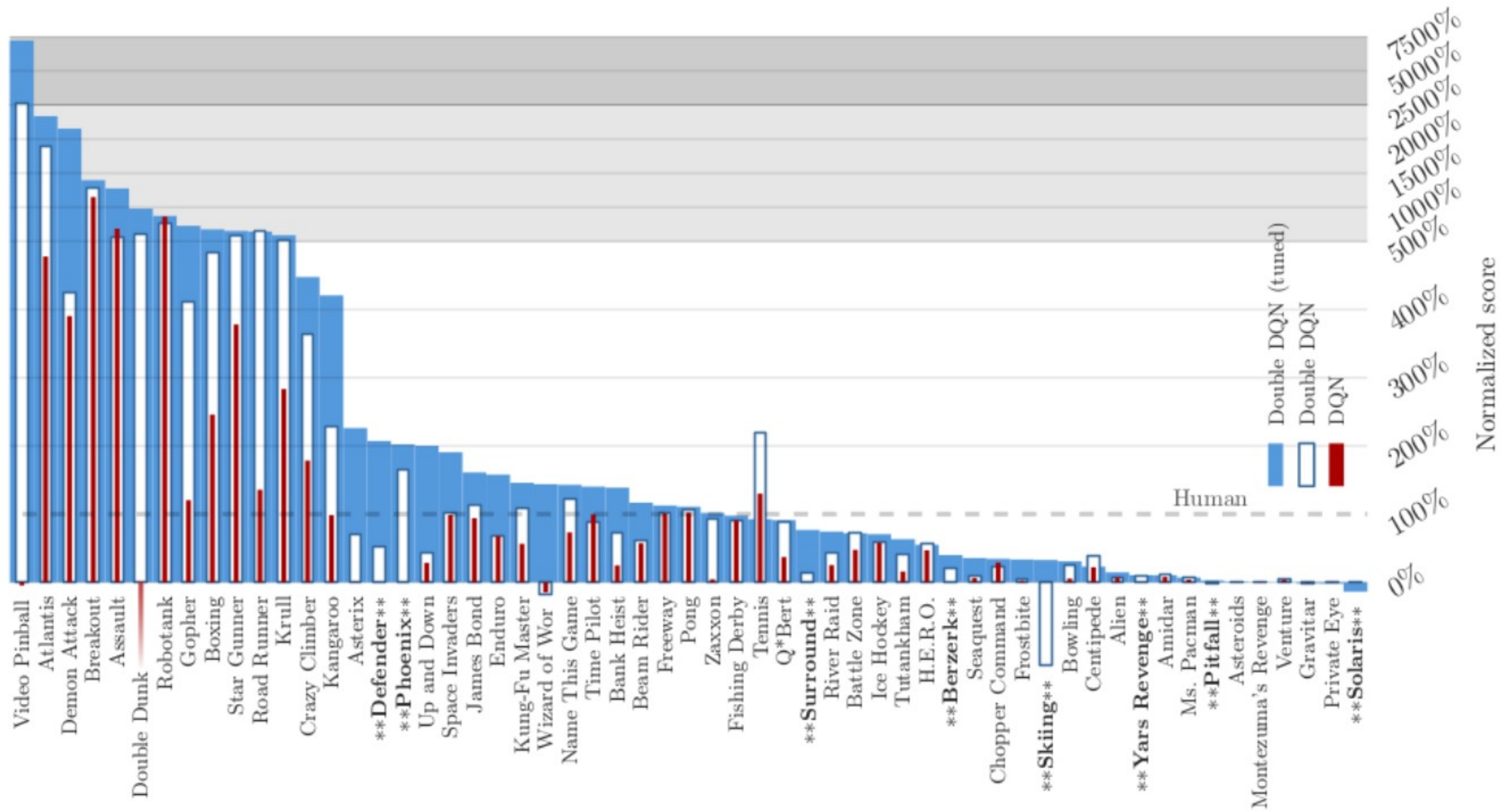
<u>Where:</u> $\nabla_w L(B; w, w^-) \approx -\frac{1}{|B|} \sum_{(s,a,s',r) \in B} \left(target(s', a'; w, w^-) - Q_w(s,a) \right) \nabla_w Q_w(s,a)$

$$target(s', a'; w, w^-) = r + \gamma Q_{w^-}(s', \arg\max_{a'} Q_w(s', a'))$$

Double DQN


Wizard of Wor
Asterix
Score
Double DQN
DQN
Double DQN
DQN
4000
3000
2000
1000
0
6000
4000
2000
0
0
50
100
150
200
0
50
100
150
200
Training steps (in millions)
Training steps (in millions)

Double DQN



	DQN	Double DQN	Double DQN (tuned)
Median	47.5%	88.4%	116.7%
Mean	122.0%	273.1%	475.2%

Table 2: Summary of normalized performance up to 30 minutes of play on 49 games with human starts. Results for DQN are from Nair et al. (2015).

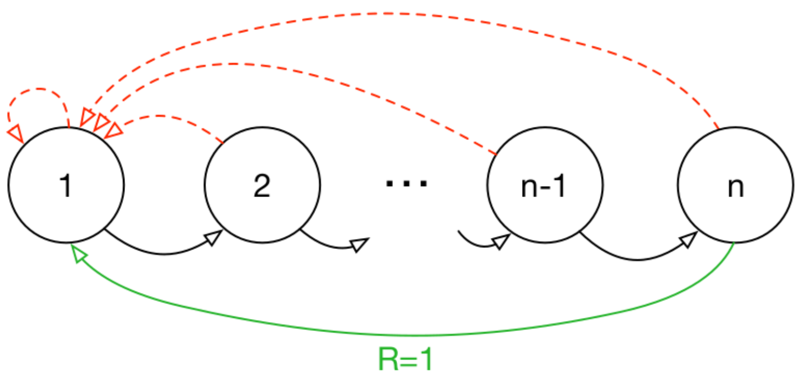
Prioritized Replay Buffer

Initialize Q_w with random weights
$D \leftarrow \emptyset$
Repeat (for each episode):
    Initialize *s*
    Repeat (for each step of the episode):
        Choose *a* from *s* using policy derived from Q (e.g. e-greedy)
        Take action *a*, observe *r, s'*
        $D \leftarrow D \cup (s, a, s', r)$
        $s \leftarrow s'$
        If mod(*step,trainfreq*) == 0:
            sample batch *B* from *D*
            $w \leftarrow w - \nabla_w L(B; w)$

Previously this sample was uniformly random

Can we do better by sampling the batch intelligently?

Prioritized Replay Buffer

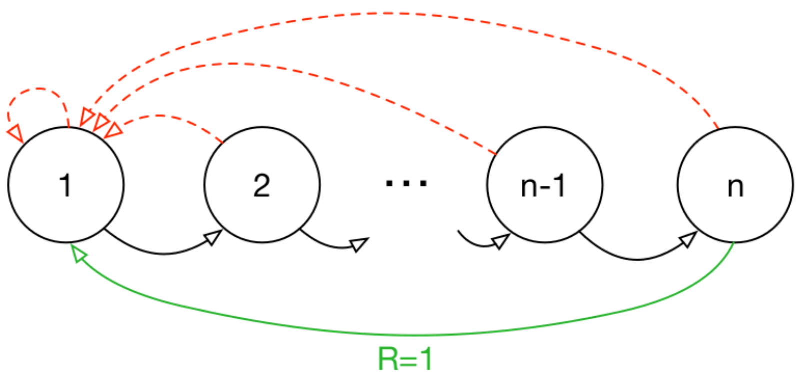


– Left action transitions to state 1 w/ zero reward

– Far right state gets reward of 1

Question

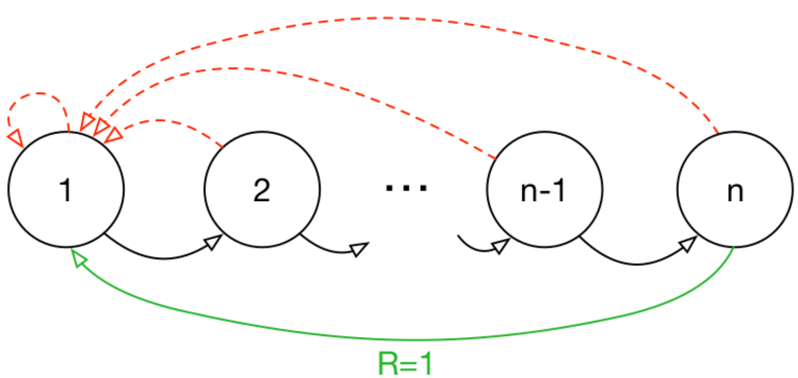
Why is the sampling method particularly important in this Domain?

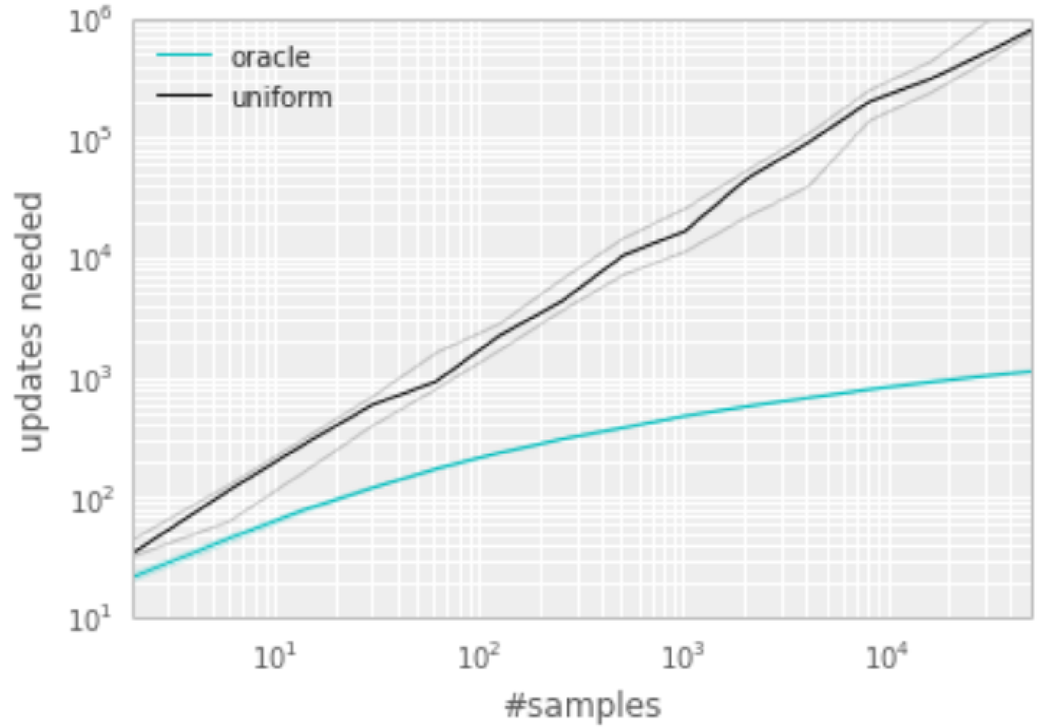


– Left action transitions to state 1 w/ zero reward

– Far right state gets reward of 1

Prioritized Replay Buffer





– Left action transitions to state 1 w/ zero reward

– Far right state gets reward of 1

Num of updates needed to learn true value fn as a function of replay buffer size

Larger replay buffer corresponds to larger values of *n* in cliffworld.

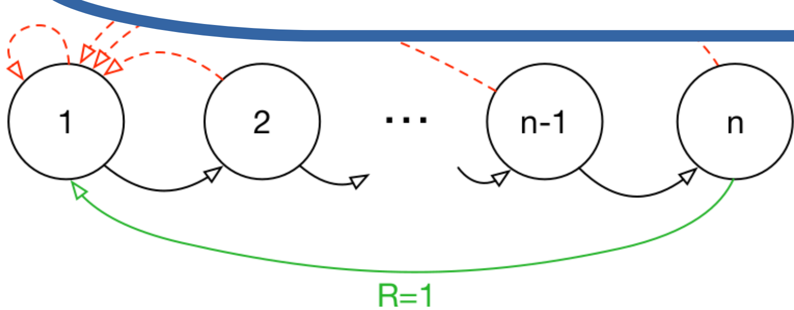
Black line selects minibatches randomly

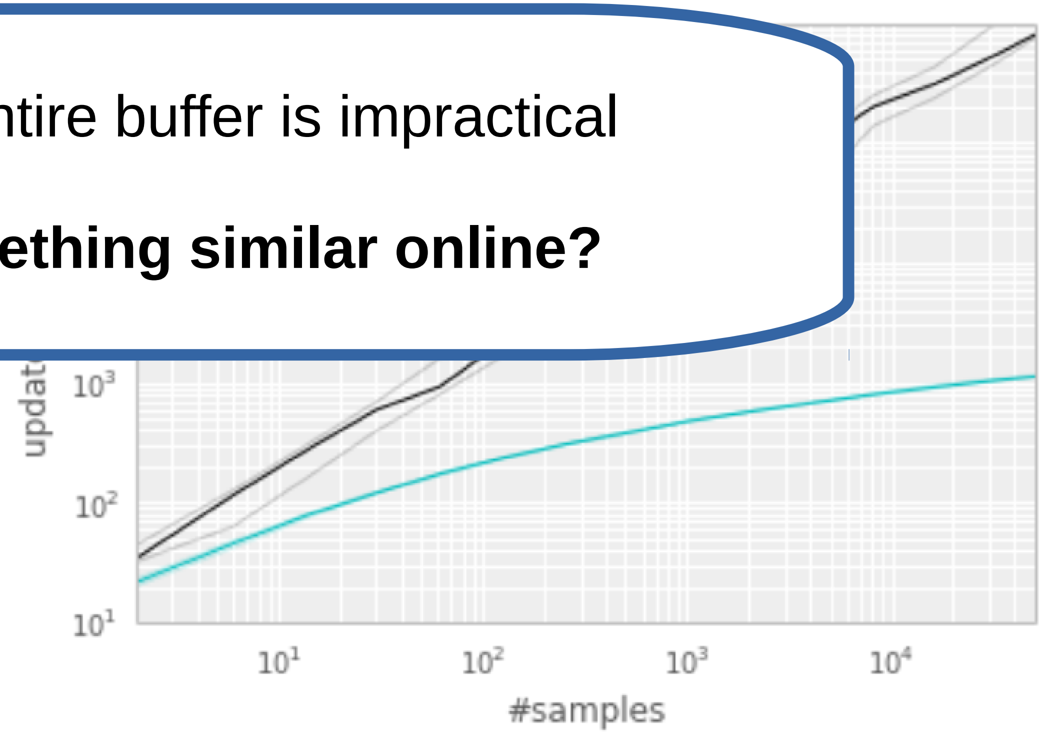
Blue line greedily selects transitions that minimize loss over entire buffer

Prioritized Replay Buffer

Minimizing loss over entire buffer is impractical

Can we achieve something similar online?





– Left action transitions to state 1 w/ zero reward

– Far right state gets reward of 1

Num of updates needed to learn true value fn as a function of replay buffer size

Larger replay buffer corresponds to larger values of *n* in cliffworld.

Black line selects minibatches randomly

Blue line greedily selects transitions that minimize loss over entire buffer

Question

Idea: sample elements of minibatch by drawing samples with probability:

$$P(i) = \frac{p_i}{\sum_k p_k}$$

where p_i denotes the priority of a sample
– simplest case: $p_i = \mathrm{TD\ error} + \epsilon$
(this is “proportional” sampling)

Problem: since we’re changing the distribution of updates performed, this is *off policy.*
– need to weight sample updates…

Question: qualitatively, how should we re-weight experiences?
– e.g. how should we re-weight an experience that prioritized replay does not sample often?

Prioritized Replay Buffer

<u>Idea</u>: sample elements of minibatch by drawing samples with probability: $P(i) = \frac{p_i}{\sum_k p_k}$

where p_i denotes the priority of a sample
– simplest case: $p_i = \mathrm{TD\ error} + \epsilon$
(this is "proportional" sampling)

<u>Problem</u>: since we're changing the distribution of updates performed, this is *off policy.*
– need to weight sample updates: $w_{s,a,s'} = \frac{1}{|B| P_{s,a,s'}}$

$$\nabla_w L(B; w, w^-) \approx -\frac{1}{|B|} \sum_{(s,a,s',r) \in B} w_{s,a,s'} \big(target(s', a'; w, w^-) - Q_w(s,a)\big) \nabla_w Q_w(s,a)$$

Prioritized Replay Buffer

<u>Idea</u>: sample elements of minibatch by drawing samples with probability: $P(i) = \frac{p_i}{\sum_k p_k}$

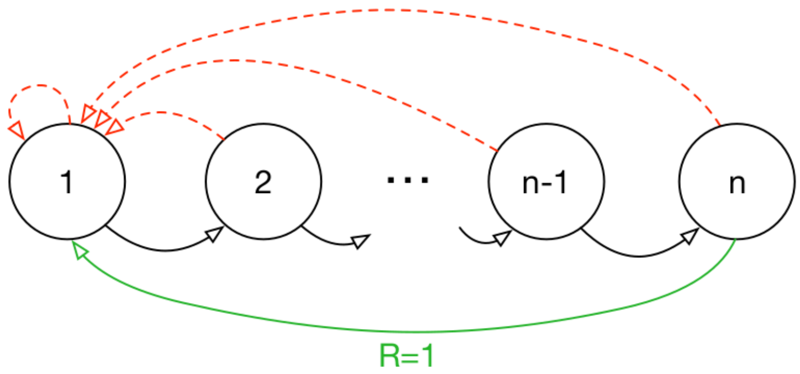
where p_i denotes the priority of a sample
– simplest case: $p_i = \text{TD error} + \epsilon$
(this is “proportional” sampling)

Why is epsilon needed?

<u>Problem</u>: since we’re changing the distribution of updates performed, this is *off policy.*
– need to weight sample updates: $w_{s,a,s'} = \frac{1}{|B| P_{s,a,s'}}$

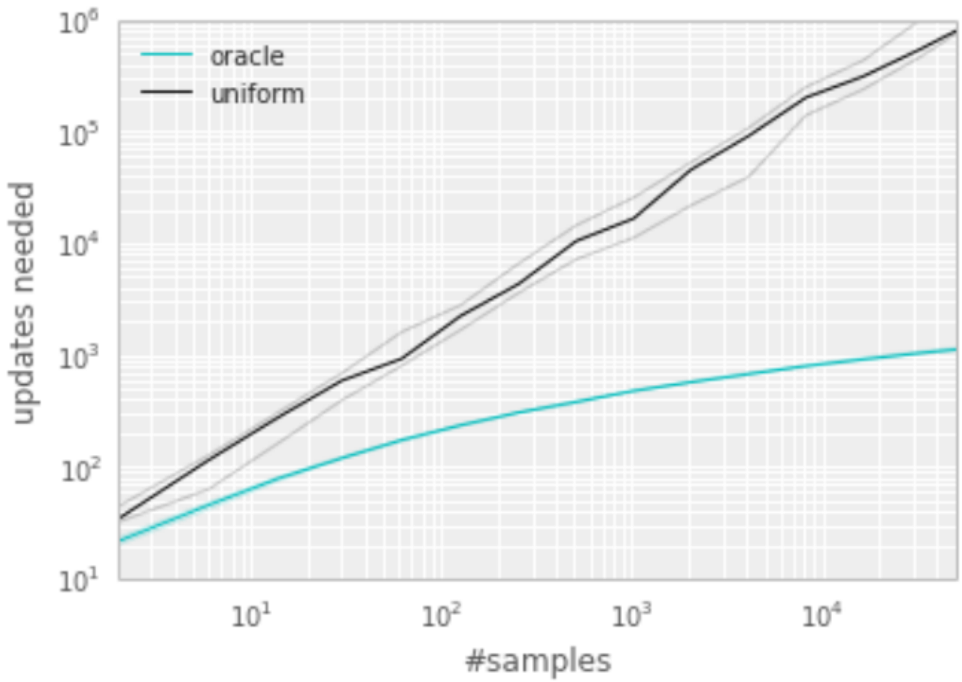
$$\nabla_w L(B; w, w^-) \approx -\frac{1}{|B|} \sum_{(s,a,s',r) \in B} w_{s,a,s'} \left(target(s', a'; w, w^-) - Q_w(s,a) \right) \nabla_w Q_w(s,a)$$

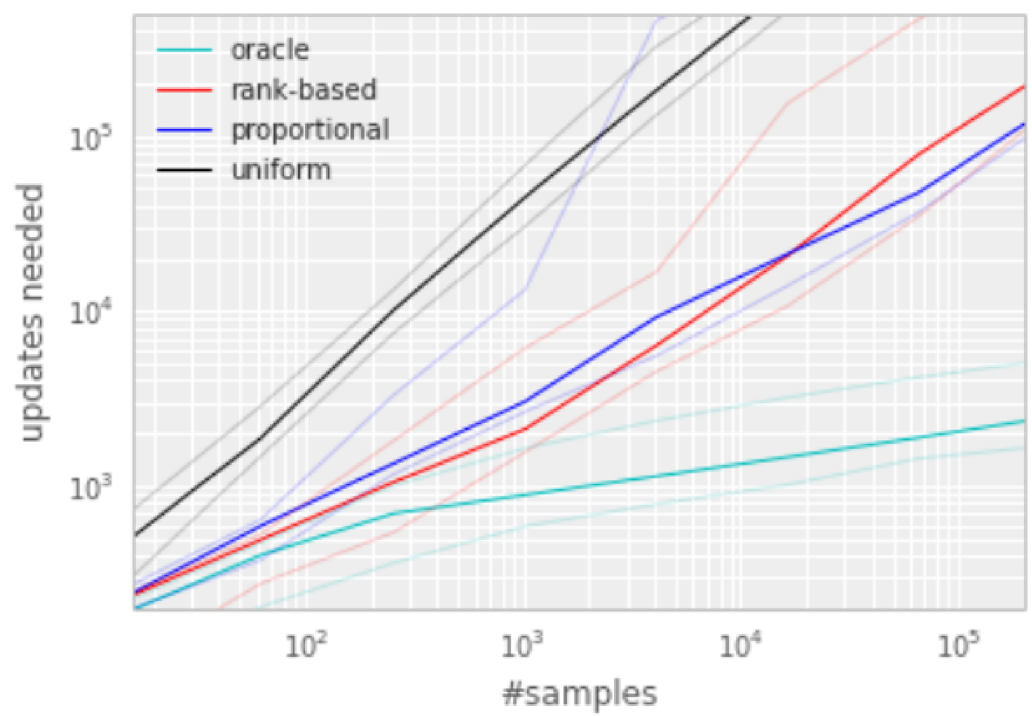
Prioritized Replay Buffer



– Left action transitions to state 1 w/ zero reward

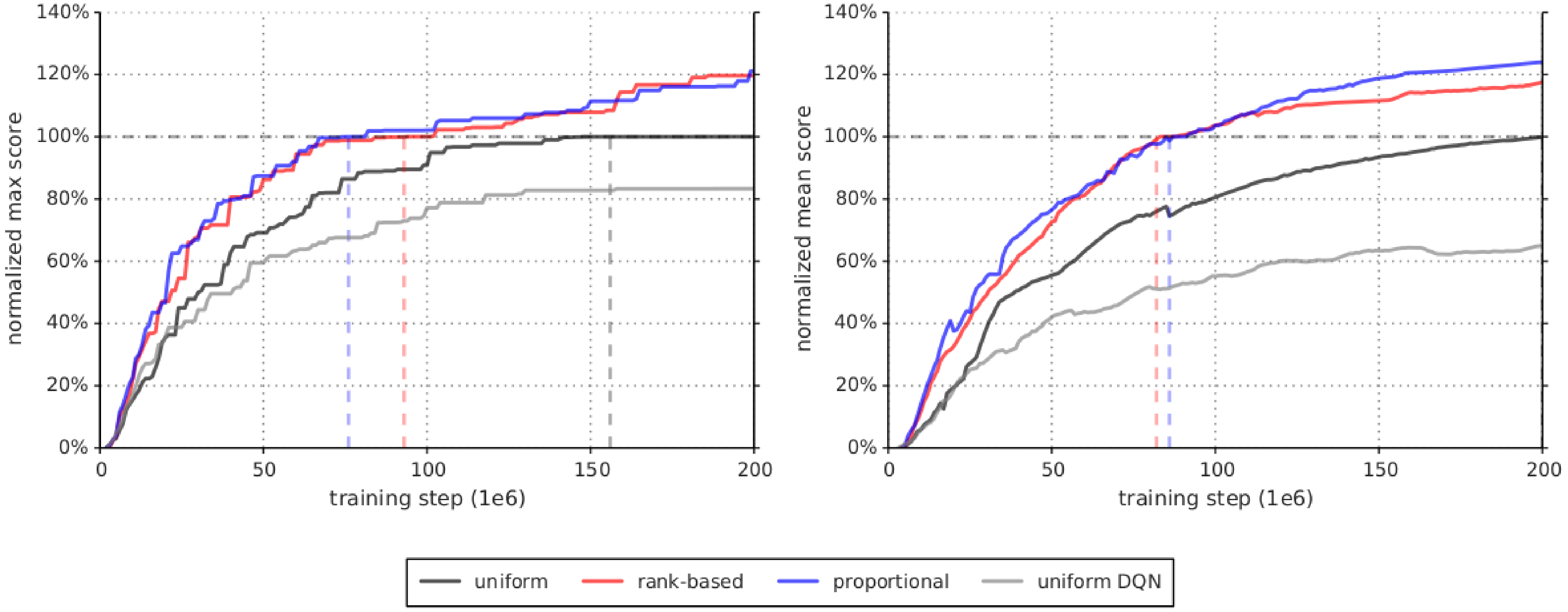
– Far right state gets reward of 1





Prioritized buffer is not as good as oracle, but it is better than uniform sampling...

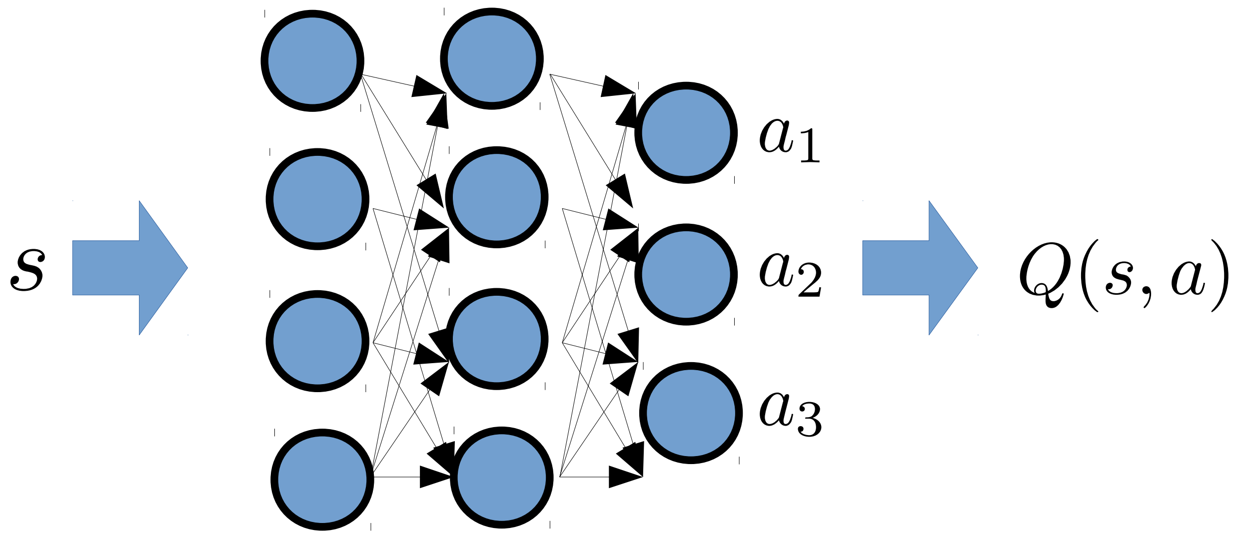
Prioritized Replay Buffer



– averaged results over 57 atari games

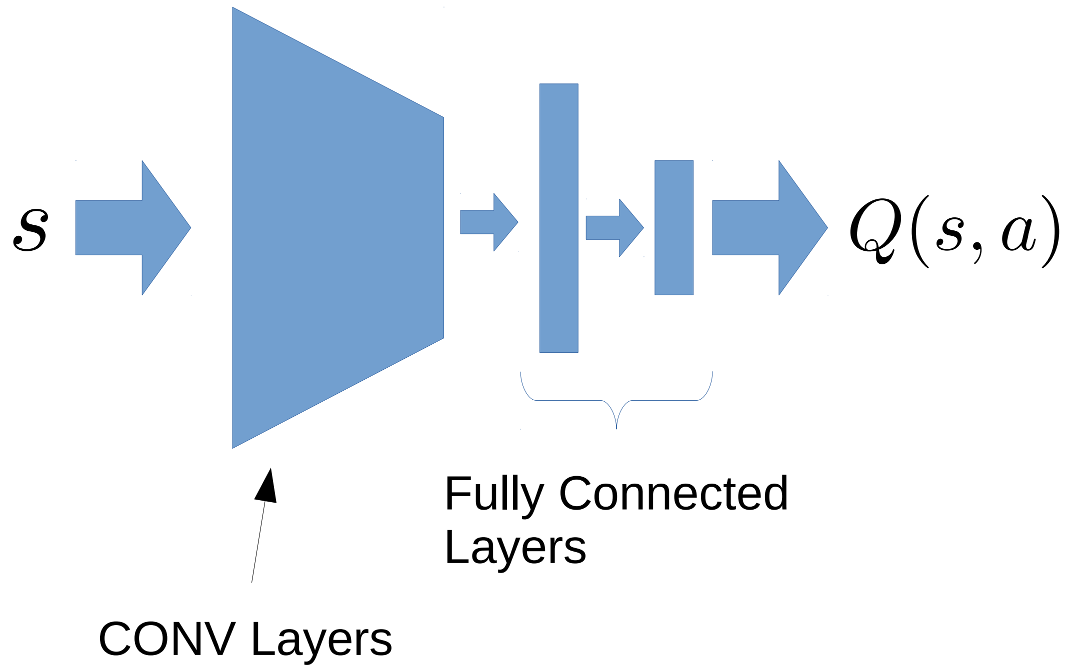
Dueling networks for Q-learning

Recall architecture of Q-network:



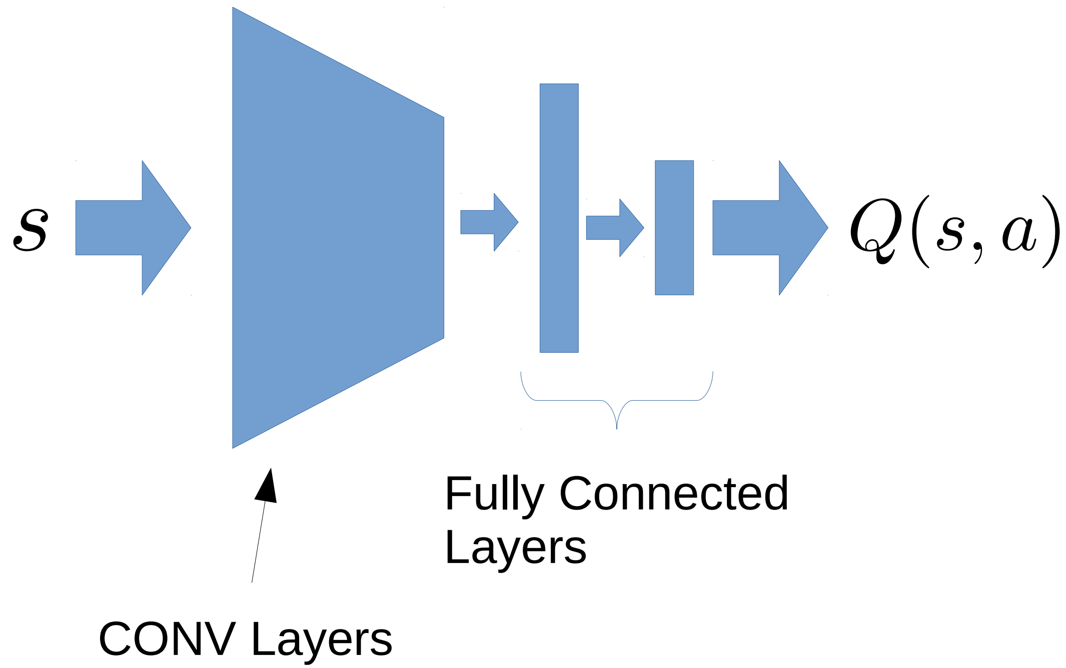
Dueling networks for Q-learning

This is a more common way of drawing it:



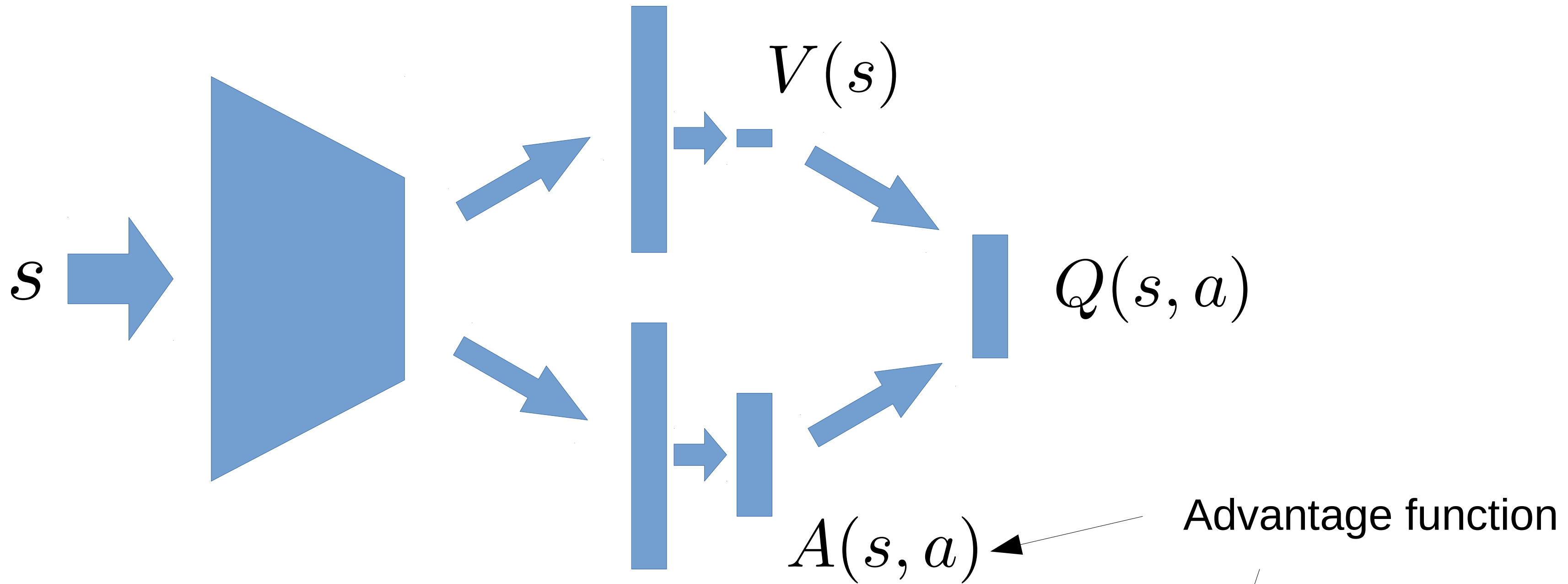
Dueling networks for Q-learning

This is a more common way of drawing it:



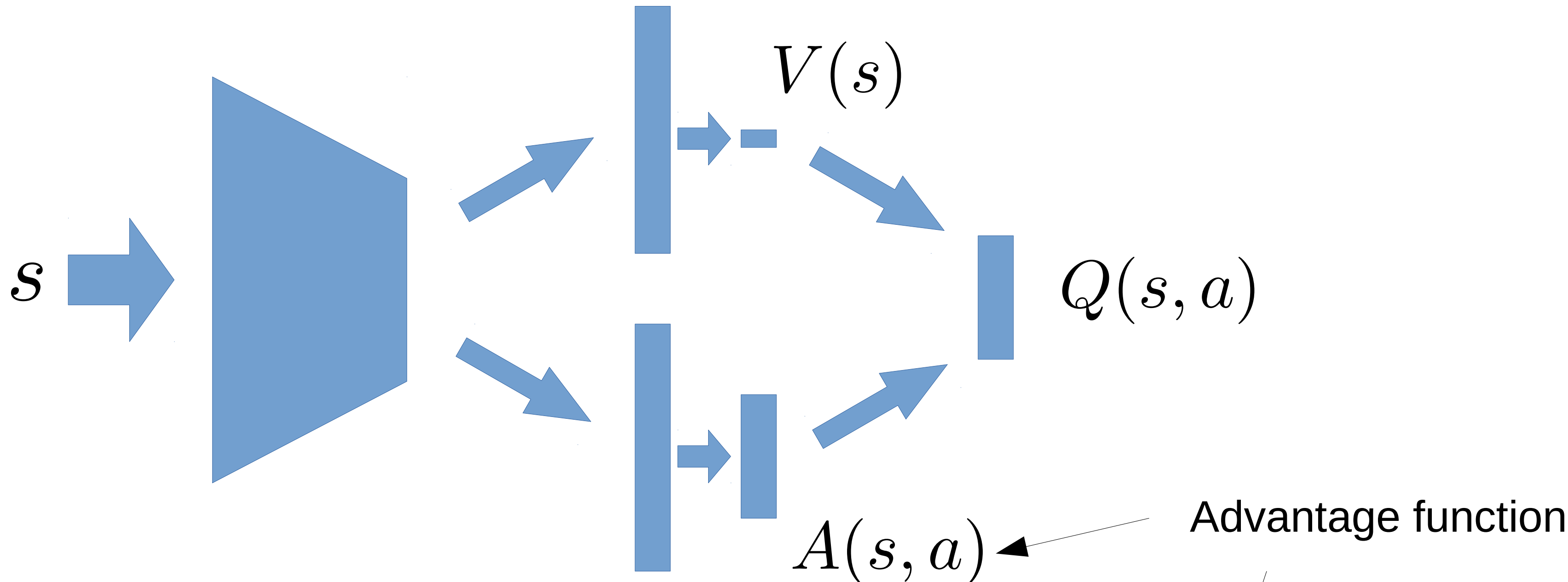
We're going to express the q-function using a new network architecture

Dueling networks for Q-learning



Decompose *Q* as: $$Q(s,a) = V(s) + A(s,a)$$

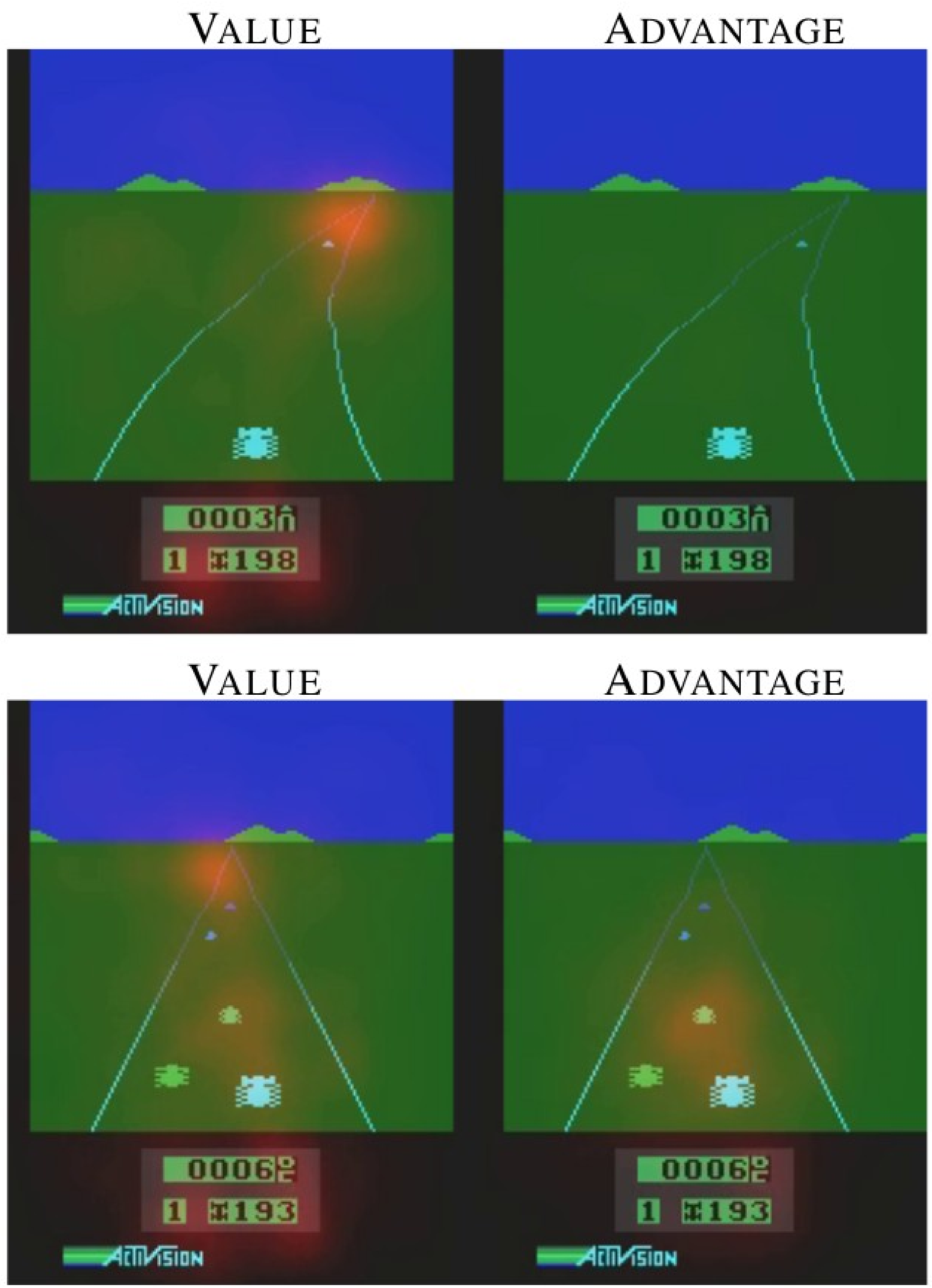
Think-pair-share



Decompose *Q* as: $Q(s,a) = V(s) + A(s,a)$

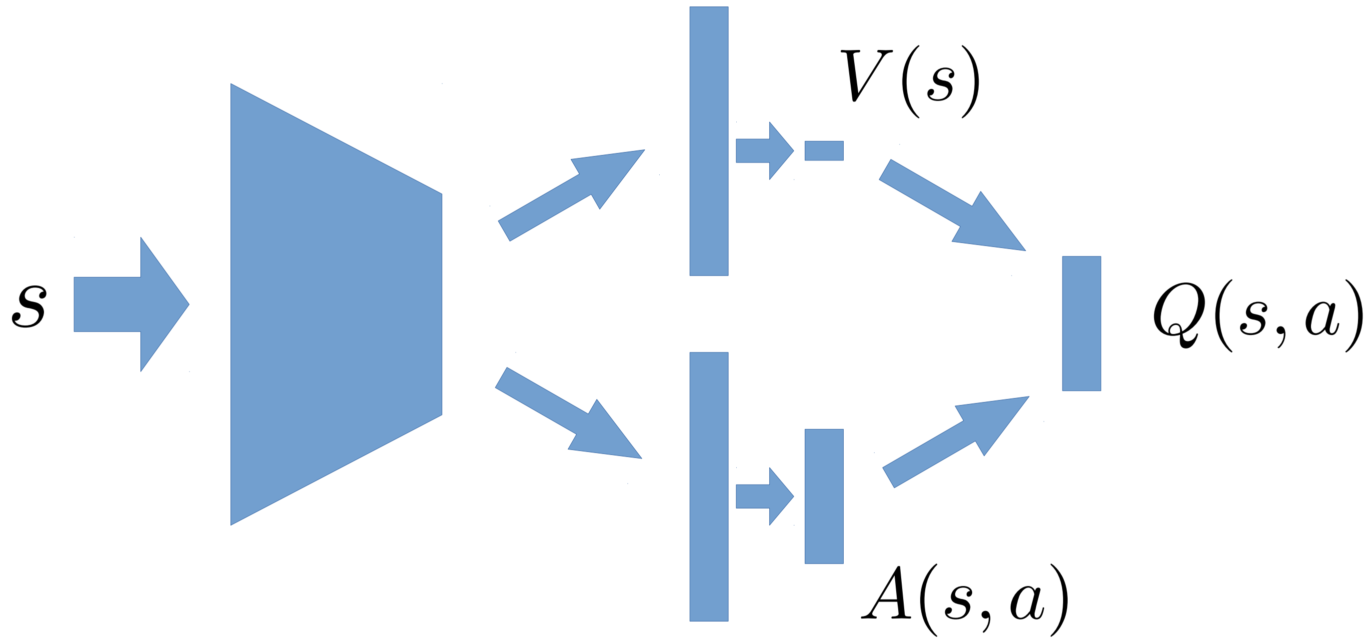
1. Why might this decomposition be better?
2. is A always positive, negative, or either? Why?

Intuition



VALUE
ADVANTAGE
0003
1
198
ACTIVISION
VALUE
ADVANTAGE
0006
1
193
ACTIVISION

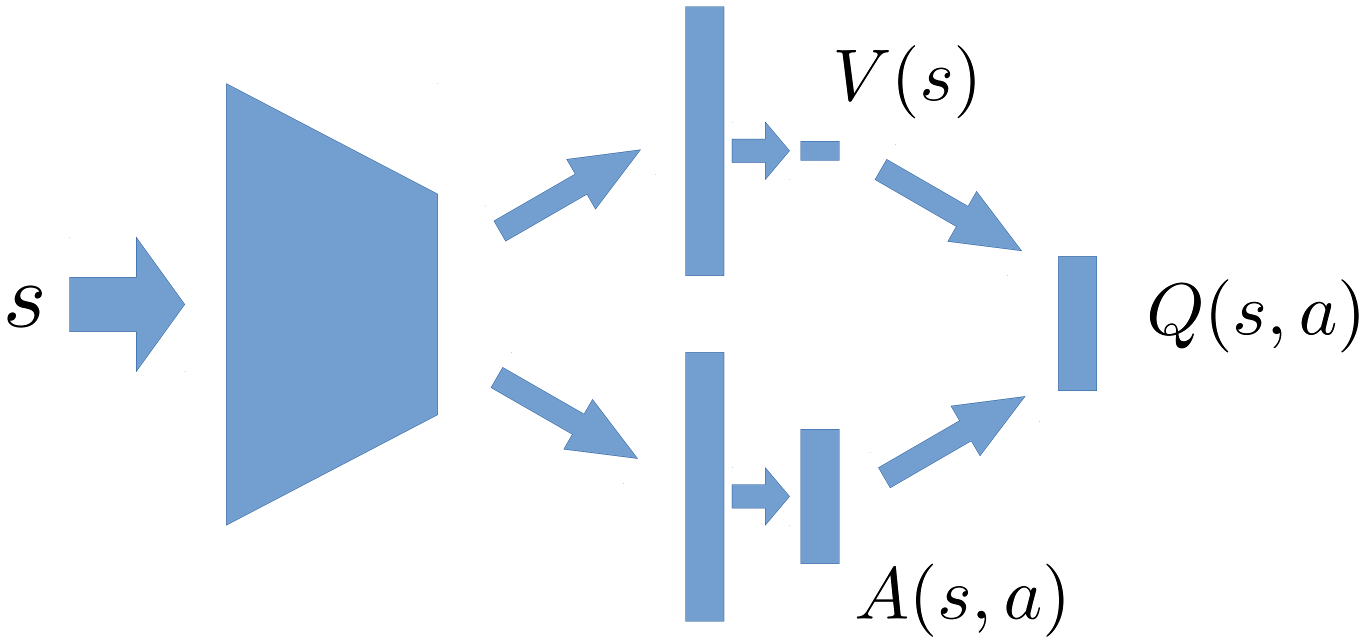
Dueling networks for Q-learning



Notice that the V/Q decomposition is not unique, given Q targets only

Therefore: $Q(s, a) = V(s) + A(s, a) - \max_a A(s, a)$

Question

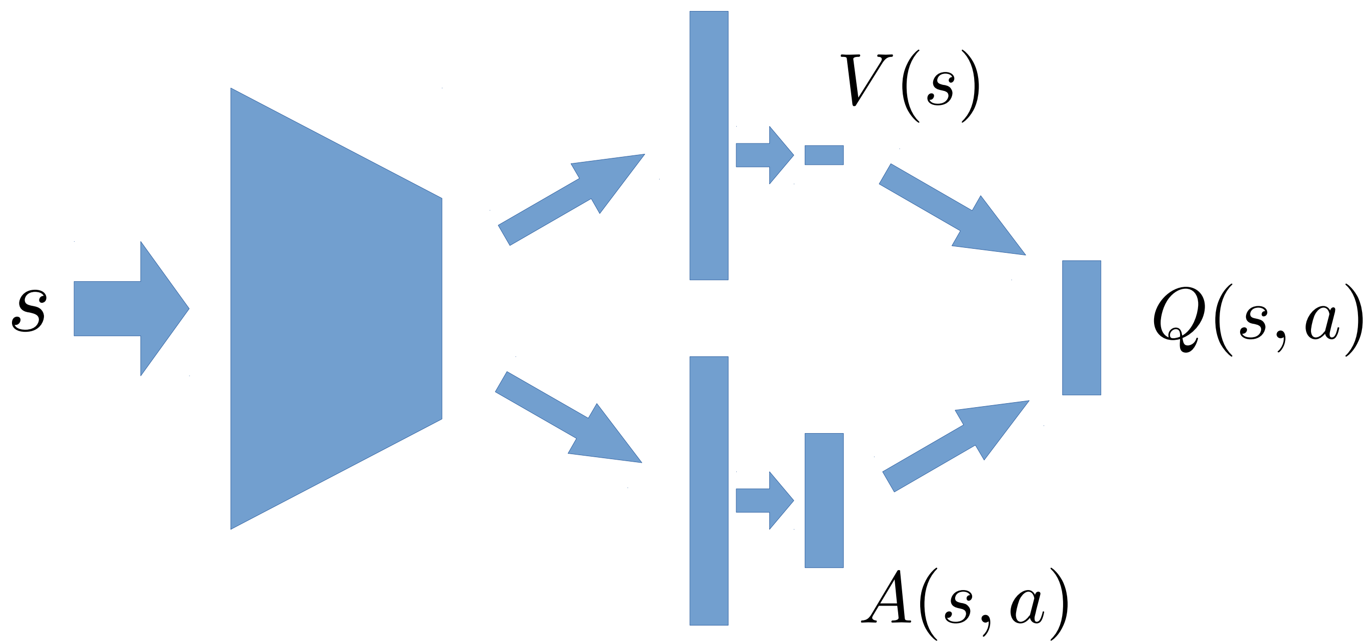


Notice that the V/Q decomposition is not unique, given Q targets only

Therefore: $Q(s,a) = V(s) + A(s,a) - \max_a A(s,a)$

Why does this help?

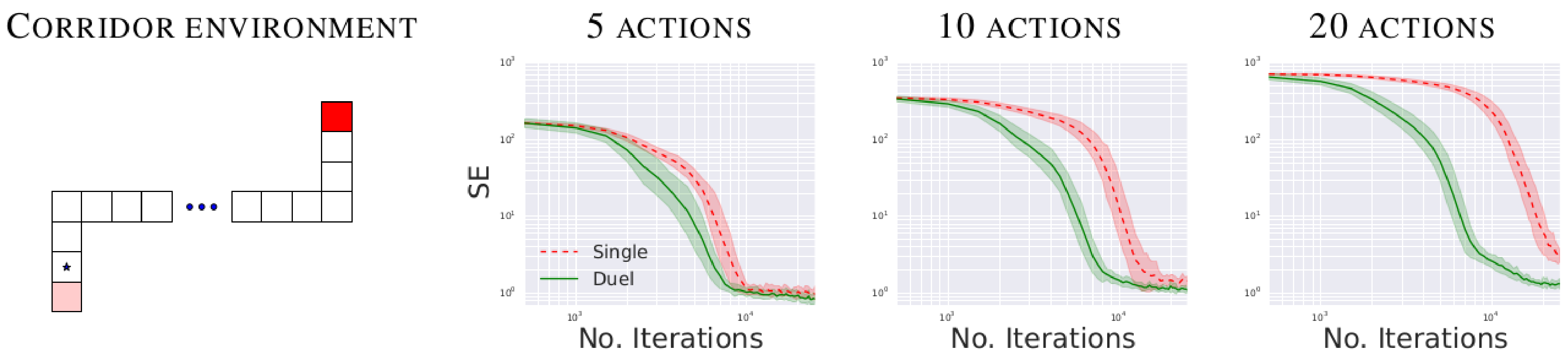
Dueling networks for Q-learning



Notice that the V/Q decomposition is not unique, given Q targets only

Actually: $$Q(s,a) = V(s) + A(s,a) - \sum_a A(s,a)$$

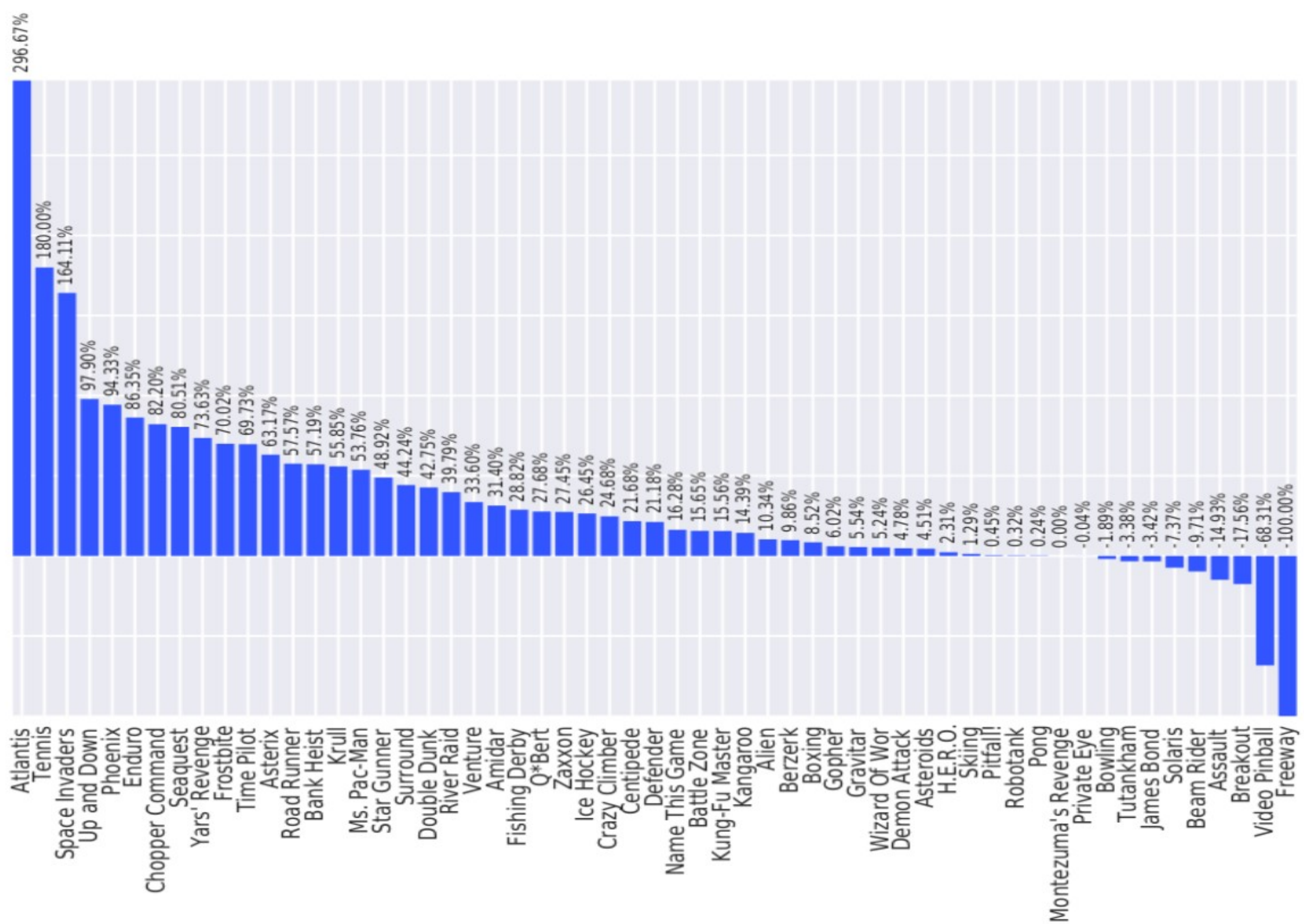
Dueling networks for Q-learning



Action set: left, right, up, down, no-op (arbitrary number of no-op actions).
SE: squared error relative to true value function
Compare dueling w/ single stream networks (all networks are three-layer MLPs)
Increasing number of actions in above corresponds to increases in no-op actions

Conclusion: Dueling networks can help a lot for large numbers of actions.

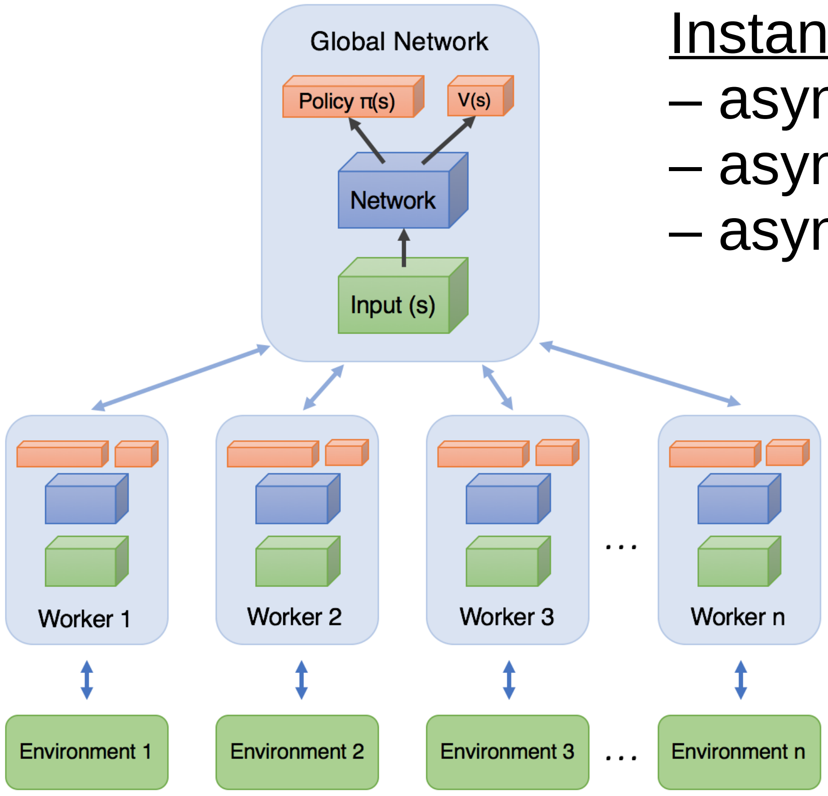
Dueling networks for Q-learning



Change in avg rewards for 57 ALE domains versus DQN w/ single network.

Asynchronous methods

Idea: run multiple RL agents in parallel
- – all agents run against their own environments and Q fn
- – periodically, all agents synch w/ a global Q fn.



Instantiations of the idea:
- – asynchronous Q-learning
- – asynchronous SARSA
- – asynchronous advantage actor critic (A3C)

Asynchronous Q-learning

Algorithm 1 Asynchronous one-step Q-learning - pseudocode for each actor-learner thread.

// Assume global shared θ, θ^-, and counter $T = 0$. ← Shared Q functions

Initialize thread step counter $t \leftarrow 0$
Initialize target network weights $\theta^- \leftarrow \theta$
Initialize network gradients $d\theta \leftarrow 0$
Get initial state s
repeat
  Take action a with ϵ-greedy policy based on $Q(s, a; \theta)$
  Receive new state s' and reward r

$$y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s' \end{cases}$$

  Accumulate gradients wrt θ: $d\theta \leftarrow d\theta + \frac{\partial (y - Q(s,a;\theta))^2}{\partial \theta}$ ← Accumulate gradients
  $s = s'$
  $T \leftarrow T + 1$ and $t \leftarrow t + 1$
  if $T \mod I_{target} == 0$ **then**
    Update the target network $\theta^- \leftarrow \theta$ ← Update target network
  end if
  if $t \mod I_{AsyncUpdate} == 0$ or s is terminal **then**
    Perform asynchronous update of θ using $d\theta$.
    Clear gradients $d\theta \leftarrow 0$.
  end if
until $T > T_{max}$

Periodically, apply batch of weight updates

Asynchronous Q-learning

Why does this approach help?

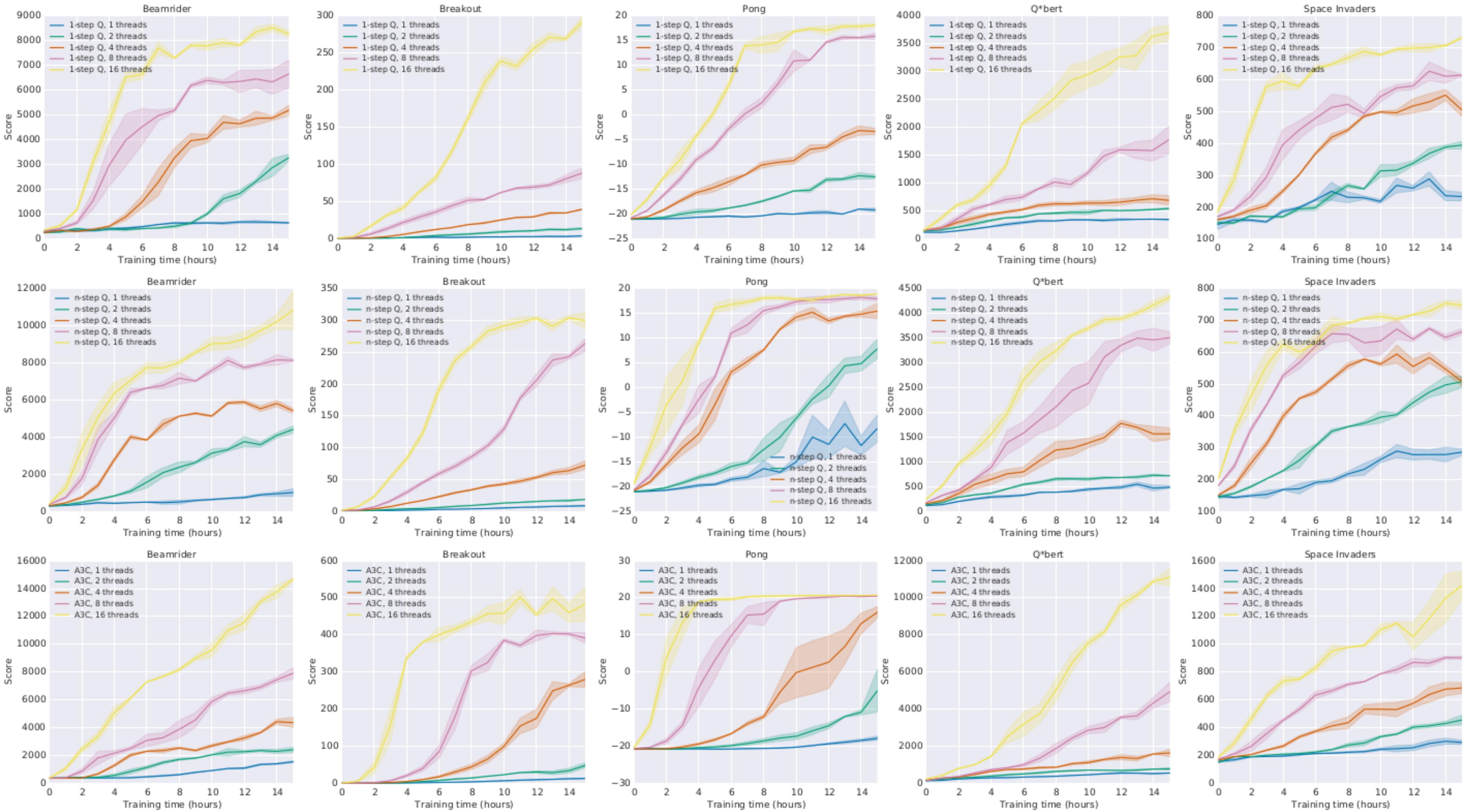
Asynchronous Q-learning

Why does this approach help?

It helps decorrelate training data

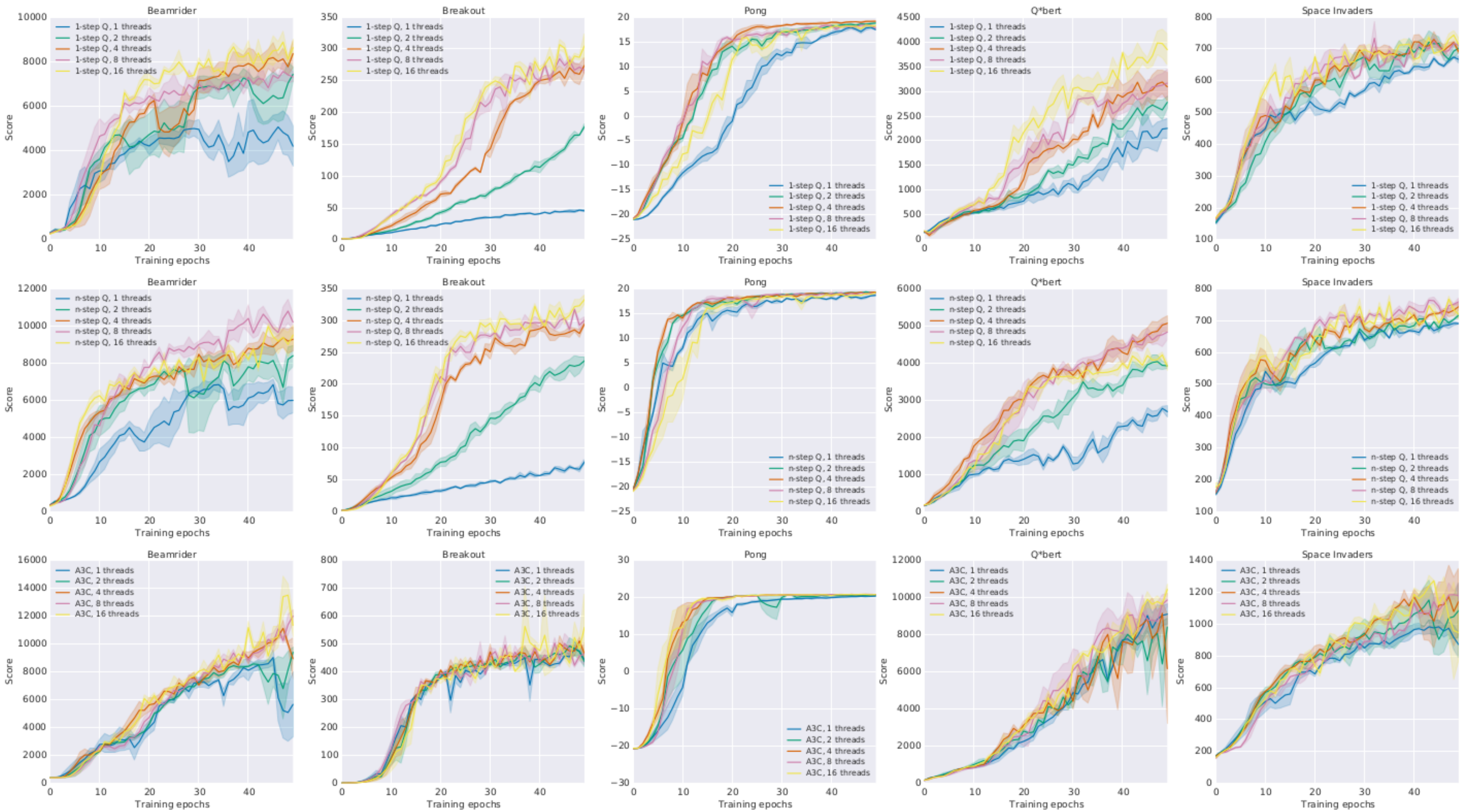
- – standard DQN relies on the replay buffer and the target network to decorrelate data
- – asynchronous methods accomplish the same thing by having multiple learners
- – makes it feasible to use on-policy methods like SARSA (why?)

Asynchronous Q-learning



Different numbers of learners versus wall clock time

Asynchronous Q-learning



Different numbers of learners versus number of SGD steps across all threads – speedup is not just due to greater computational efficiency

Combine all these ideas!