Reinforcement Learning

an introduction

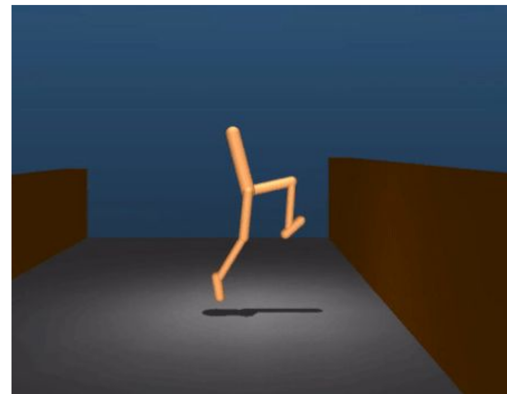
Nick Knowles
knowlen@wwu.edu

Why Reinforcement Learning?

An effective way to approach artificial intelligence (learning/adaptation)

A framework for applying deep learning to problems of general intelligence, planning, and control

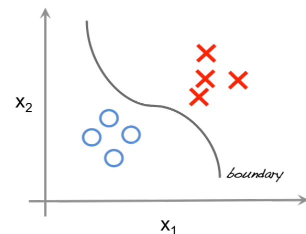
Relatively cheaper to implement



Machine Learning Paradigms

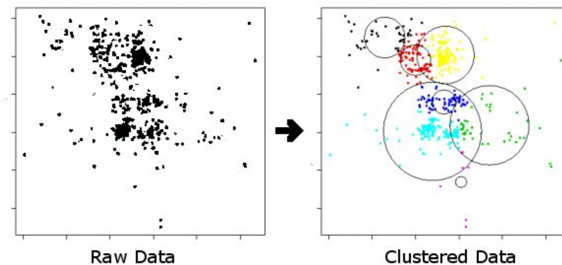
Supervised Learning

- Show what to do through labeled examples.
- Stops learning after initial training.
- Try to learn the underlying relationship between X and Y.



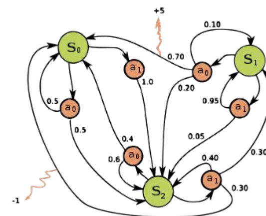
Unsupervised Learning

- Find patterns.
- No metric of right/wrong.
- Try to figure out the distribution of X.



Reinforcement Learning

- Let an agent learn through interaction with the environment.
- Reward signals instead of ground truth labels.
- Try to learn a good policy for how to act in various states.

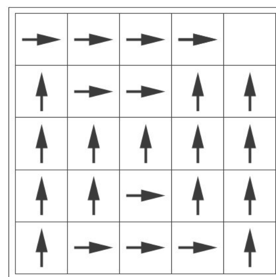


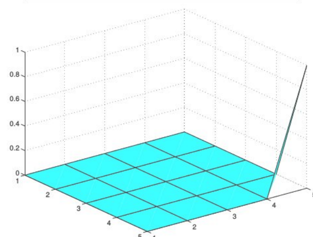
Common Limitations

Need to have an **environment** or **model** for the agent to carry out actions in.

Need access **to** a **reward signal.**

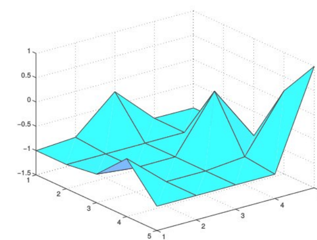
Optimal Policy

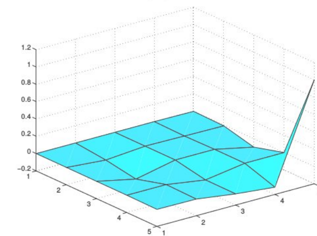




True reward function

Models





F

Concept of a state

Could bin "similar" states,

Observarion		State
[1, 0, 1, 0]	=>	2
[0, 0, 0, 0]	=>	0
[0, 1, 0, 1]	=>	2
[0, 1, 0, 0]	=>	1
[1, 0, 0, 0]	=>	1

eg; state = sum(Observation)

Or append a new state for every unique vector encountered,

Observation		State
[1, 0, 1]	=>	0
[0, 1, 0]	=>	1
[0, 1, 1]	=>	2
[1, 1, 1]	=>	3
[1, 1, 0]	=>	4

ect...

Concept of a state

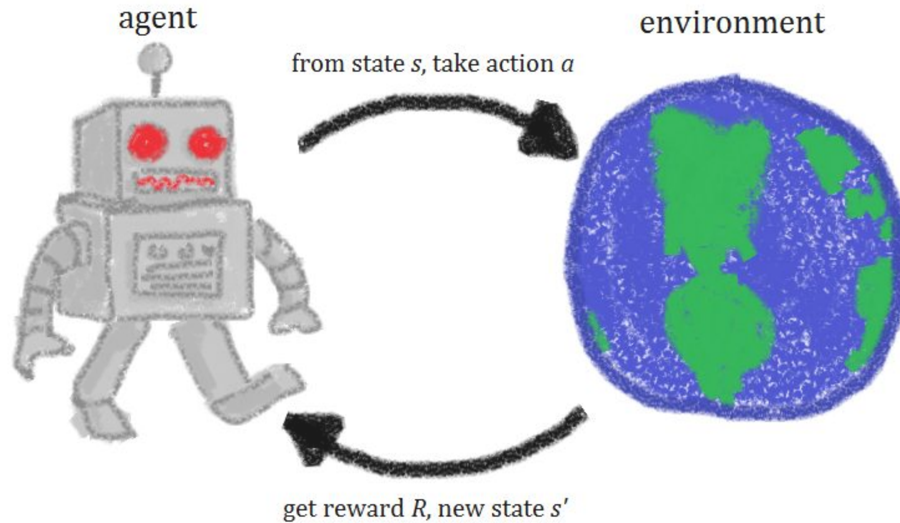
Usually have to bin continuous features

Example Bins:

Condition		Bin
$-1 < x < 1$	==	0
$x > 1$	==	1
$x < -1$	==	-1

Observation		Binned Observation		State
[0.2145, -1.8042, 9.0265, 0.8511]	=>	[0, -1, 1, 0]	=>	1

Learning through interaction



Notation

S: Current state

- Obtained from the environment at timestep 0.
- Obtained from **S'** after timestep 0

a: Action to be taken in S

- Obtained from a decision function.

r: Immediate reward

- Obtained from environment after taking **a** in **S**

S': Next state

- Obtained from the environment after taking **a** in **S**
- **S** = **S'** on next the timestep

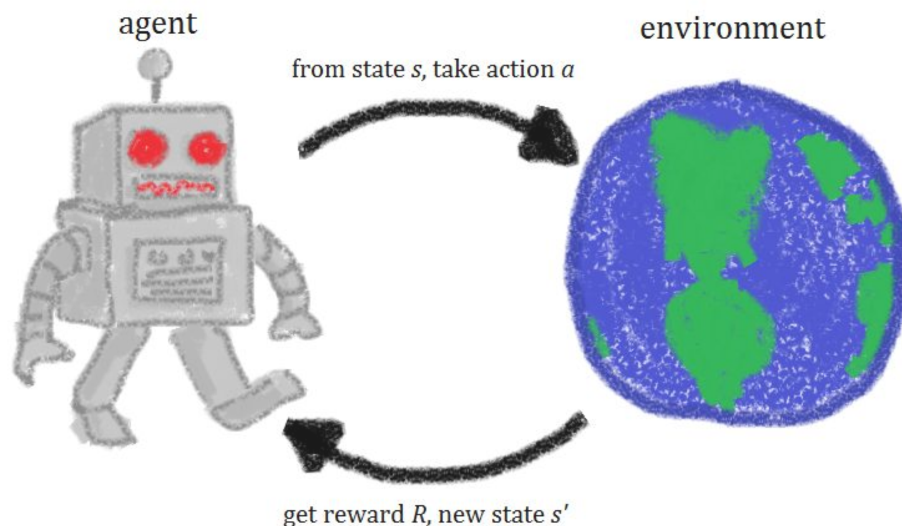
Learning through interaction

$$\ldots\ s_t,\ a_t,\ r_t,\ s_{t+1},\ a_{t+1},\ r_{t+1},\ s_{t+2},\ a_{t+2},\ r_{t+2},\ s_{t+3}\ \ldots$$

- Note the **state** *before* taking action.
- Execute an **action**.
- Note any **reward** received.
- Note the **state** *after* taking action.

Experience tuple: $<S_t, a_t, r_t, S_{t+1}>$

$== \ \mathbf{< S, a, r, S' >}$



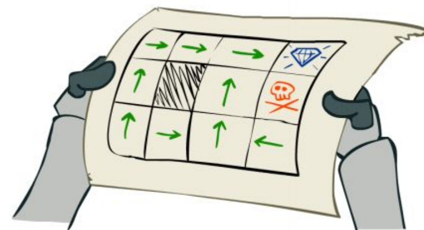
The goal

Use experience tuples **<S, a, r, S'>** to iteratively learn:

[S] -> **[a** that leads to good future states**]**

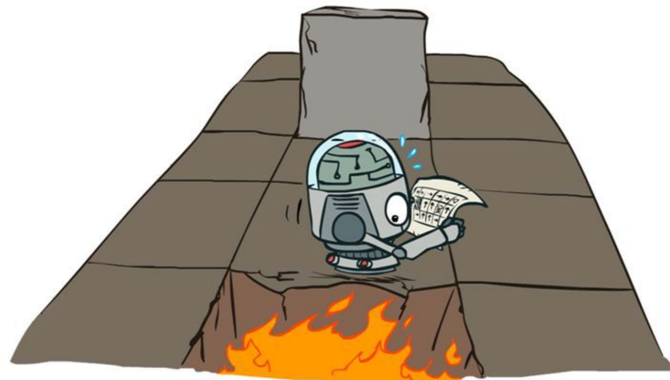
... a "Policy"

Policy Function

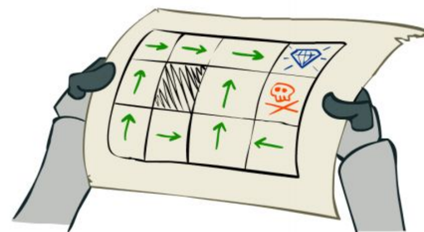


a **Policy function** dictates which actions will be selected by the agent.

π(S): [probability vector over actions]



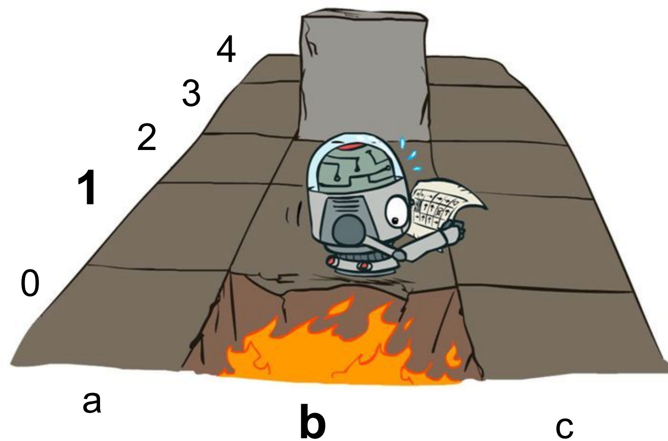
Policy Function (example)



Agent observes **state**$_{b1}$ from the environment.
It plugs the observation into π,

$$\pi(\mathbf{state}_{b1}) = [0, 0.8, 0.1, 0.1]$$

The policy function says;
take $\mathbf{a}_1$ in **state**$_{b1}$ 80%of the time,
take either $\mathbf{a}_2$ or $\mathbf{a}_3$ the other 20%,
but never take $\mathbf{a}_0$ in this state.



Value Function

Value functions return the long term value of being in some state.

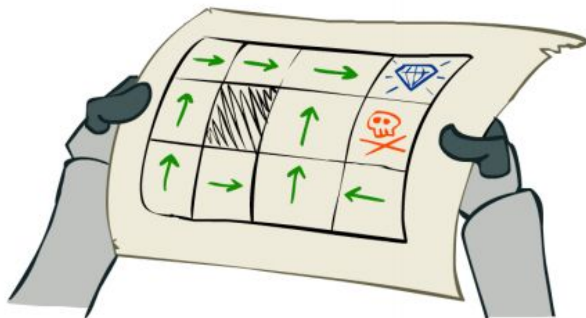
V(S): cumulative reward if taking the 'best' action in this & all future states

.. can also be based **on policy**,

$V^{\pi}(S_t)$: cumulative reward if we only take actions given by $\pi(S_t)$ to S_{t+n}

If we always take the most probable or highest value action, then
V(S) is considered **off policy**.

DQN and Q-learning are off policy.



Value Functions

Value functions: take a state and return a (long term) value.

V(S): cumulative reward if take the 'best' action in this & all future states

.. can also be based **on policy**,

$V^{\pi}(S_t)$: cumulative reward if we only take actions given by $\pi(S_t)$ to S_{t+n}

But how can we know these?

$$V(S) = V(S) + \text{learn_rate}(r + \alpha V(S') - V(S))$$

where α is "discount" weight on $V(S_{t+1})$

This propagates recursively to either a terminal state or a cycle,

$$V(S) = E[r_S + \alpha r_{S'} + \alpha^2 r_{S''} + \ldots]$$

Value and Policy Functions

Scaling to infinite horizons: Having a *discount factor* ensures that values do not scale to infinity when the number of decisions is unbounded.

$$\sum_{t=0}^{\infty} \gamma^t r_t$$

$$0 < \gamma < 1$$

Temporal Difference

We don't need to store previous events if we can model the change in value when transitioning between any two states

$V(S_t)$ becomes a function of $V(S_{t+n})$ by passing through the state space multiple times over.

And S is invariant to $S_{0:t-2}$ (markov property)

Allows the abstraction,

Expected future rewards can steer decisions made in the present

Temporal Difference Learning

Sometimes ignore the future by taking random actions

(Exploration vs Exploitation)

/knowlen/RL_fun/simple_rl.py

Grid World Demo

http://cs.stanford.edu/people/karpathy/reinforcejs/gridworld_td.html

The Q function

Maps state action pairs to value.

Q(S,a): the long term value of taking action a in state S

```
Q(S,a) += learn_rate * (r + discount * V(S') - Q(S,a))
```

Q learning algorithm

S $state_t$
S' $state_{t+1}$
a action taken in **S**
r reward observed after taking **a** in **S**

```
while not stopped:
    Q(S,a) += lr * (r + (d * V(S')) - Q(S,a))
    S = S'
    a = [argmax Q(S,a[0..n_act])] or random action
         action
    S', r = environment.do(action)
```

The Q function

Traditionally implemented as a table.

"optimize productivity"

Actions

States

Q Table	a_0: work	a_1: eat (30 min)	a_3: sleep (8 hr)
S_0: hungry	0	0	0
S_1: tired	0	0	0
S_2: caffeinated	0	0	0
S_3: other	0	0	0

The Q function

Traditionally implemented as a table.

"optimize productivity"

Actions

States

Q Table	a_0: work	a_1: eat (30 min)	a_3: sleep (8 hr)
S_0: hungry	0	0	0
S_1: tired	0	0	0
S_2: caffeinated	10	0	-10
S_3: other	5	0	0

The Q function

Traditionally implemented as a table.

"optimize productivity"

Actions

States

Q Table	a_0: work	a_1: eat (30 min)	a_3: sleep (8 hr)
S_0: hungry	0	8	0
S_1: tired	-8	0	0
S_2: caffeinated	20	0	-20
S_3: other	10	-1	0

The Q function

Traditionally implemented as a table

"optimize productivity"

Actions

States

Q Table	a_0: work	a_1: eat (30 min)	a_3: sleep (8 hr)
S_0: hungry	-6	23	1
S_1: tired	-23	-1	6
S_2: caffeinated	40	0	-40
S_3: other	20	-1	-16

The Q function

Traditionally implemented as a table.

"optimize productivity"

Actions

States

Q Table	a_0: work	a_1: eat (30 min)	a_3: sleep (8 hr)
S_0: hungry	-15	68	30
S_1: tired	-50	-15	42
S_2: caffeinated	80	0	-100
S_3: other	40	-10	-20

Limitation of Q Learning

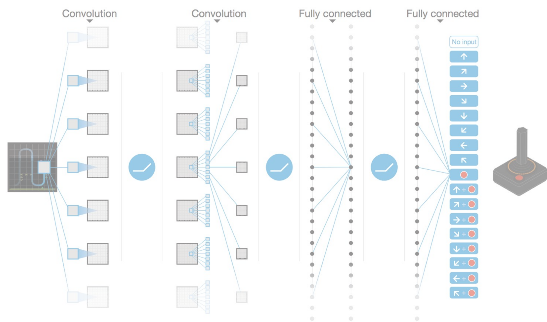
Q	a_1	a_2
s_1		
s_2		

Constrained to **discrete state space**

-Table blows up

Limitation of Q Learning

Constrained to **discrete state space**

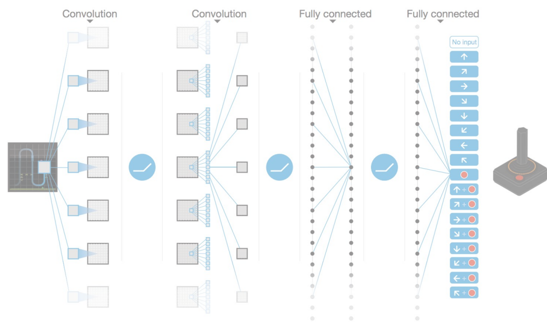


... solved by using neural nets

Q Learning: Questions?

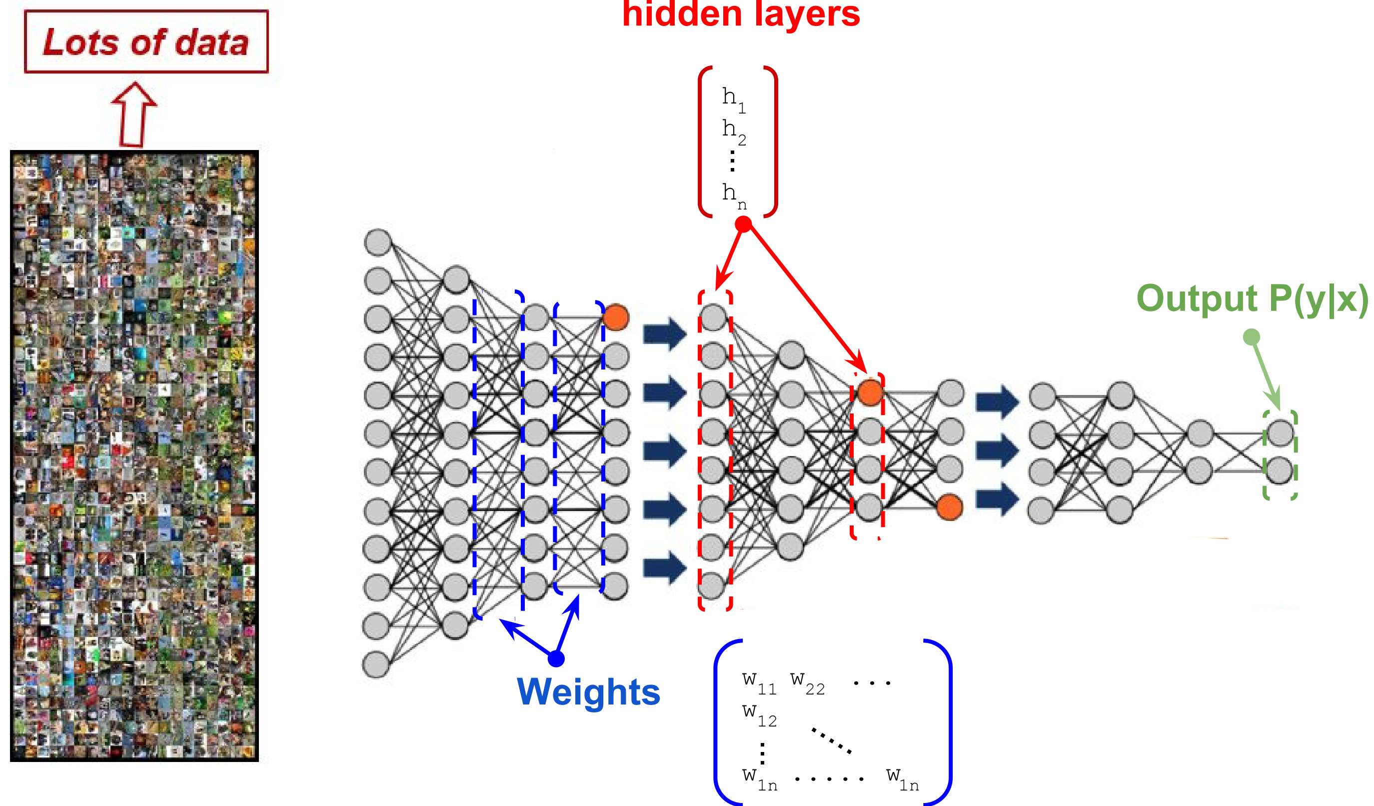
Limitation of Q Learning

Constrained to **discrete state space**

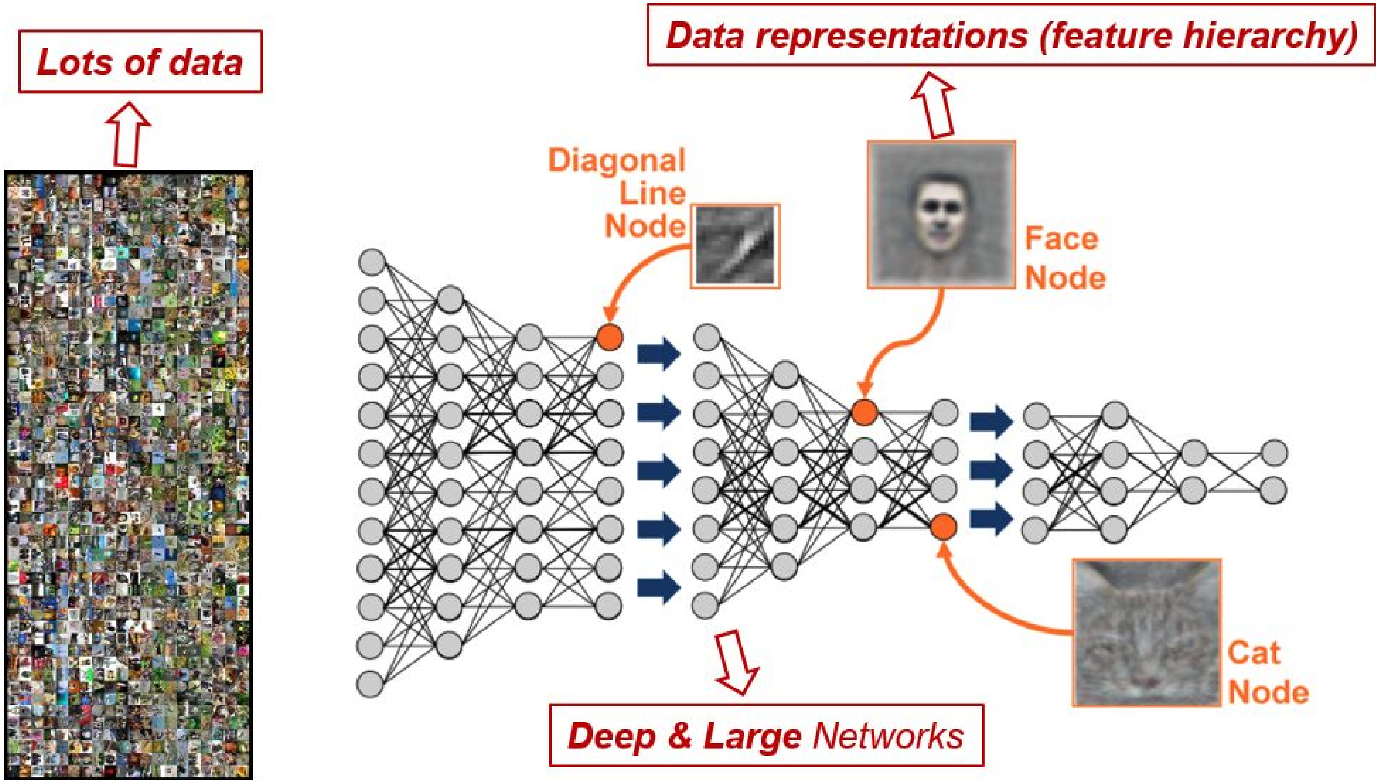


... solved by using neural nets

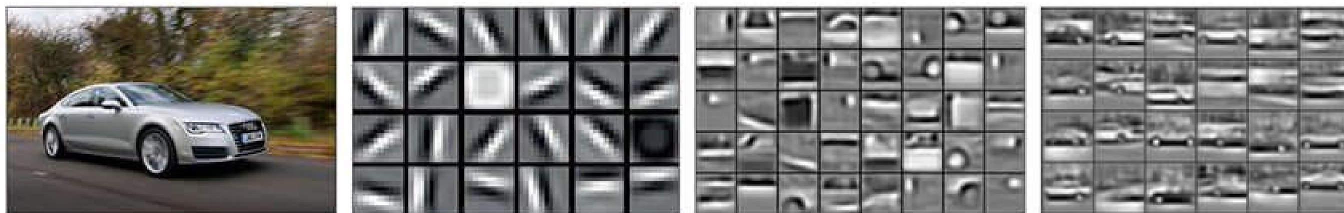
Neural Networks

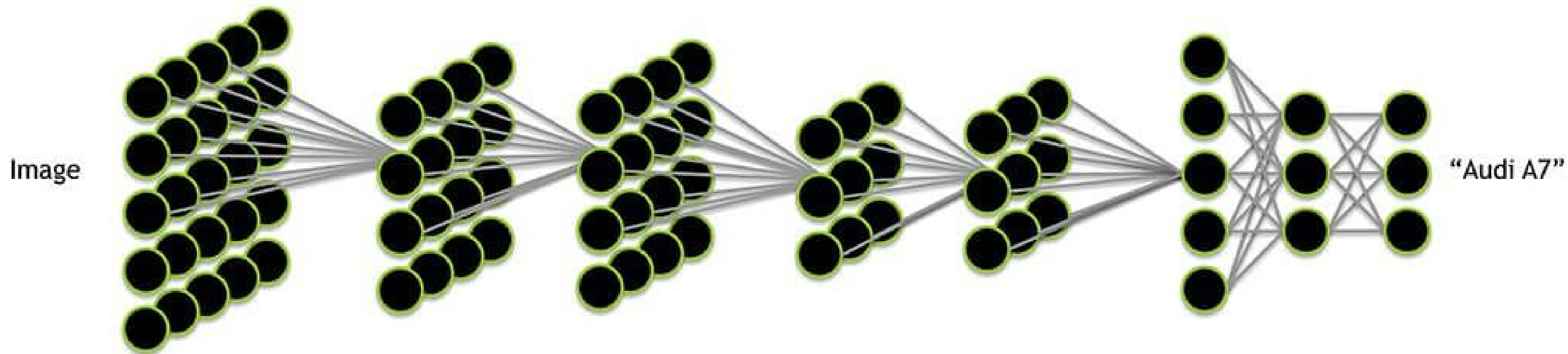


Neural Networks

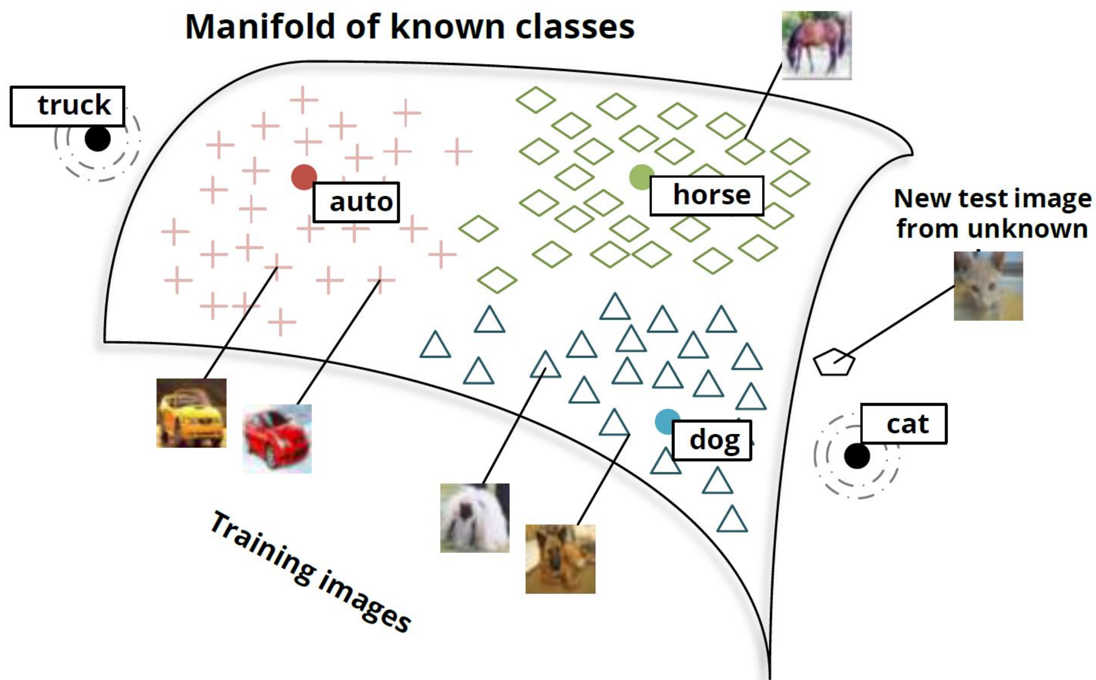


Neural Networks

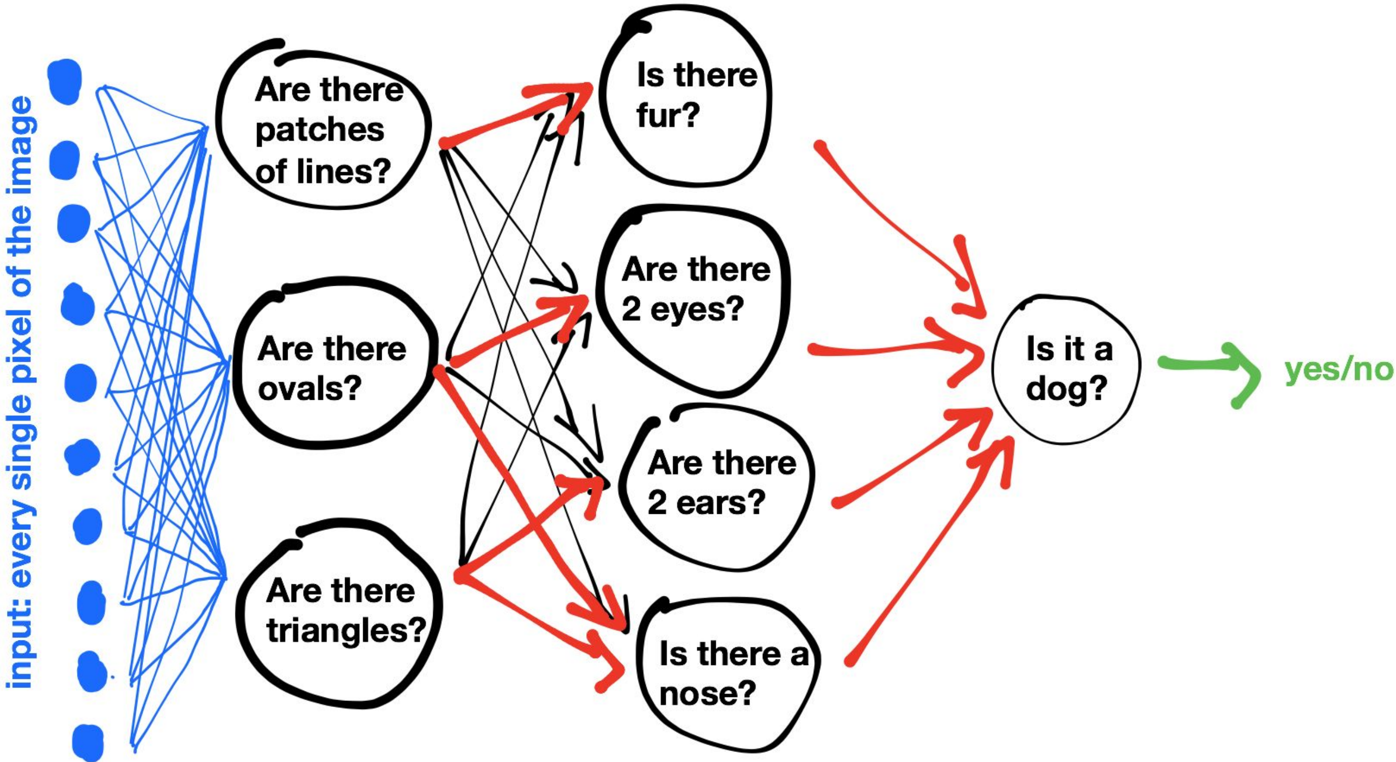




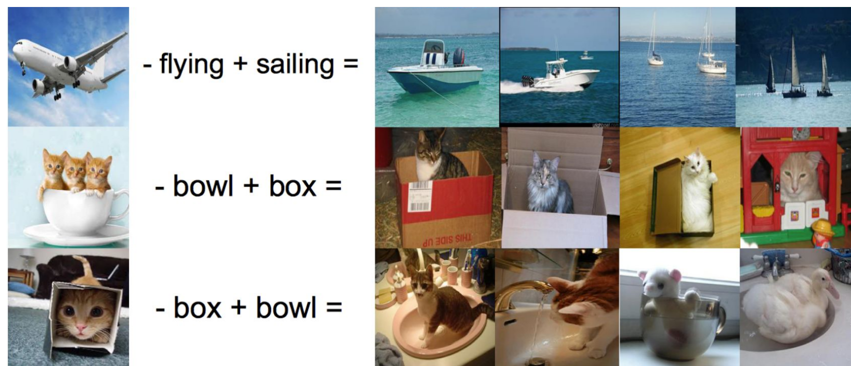
Neural Networks



Neural Networks



Neural Networks



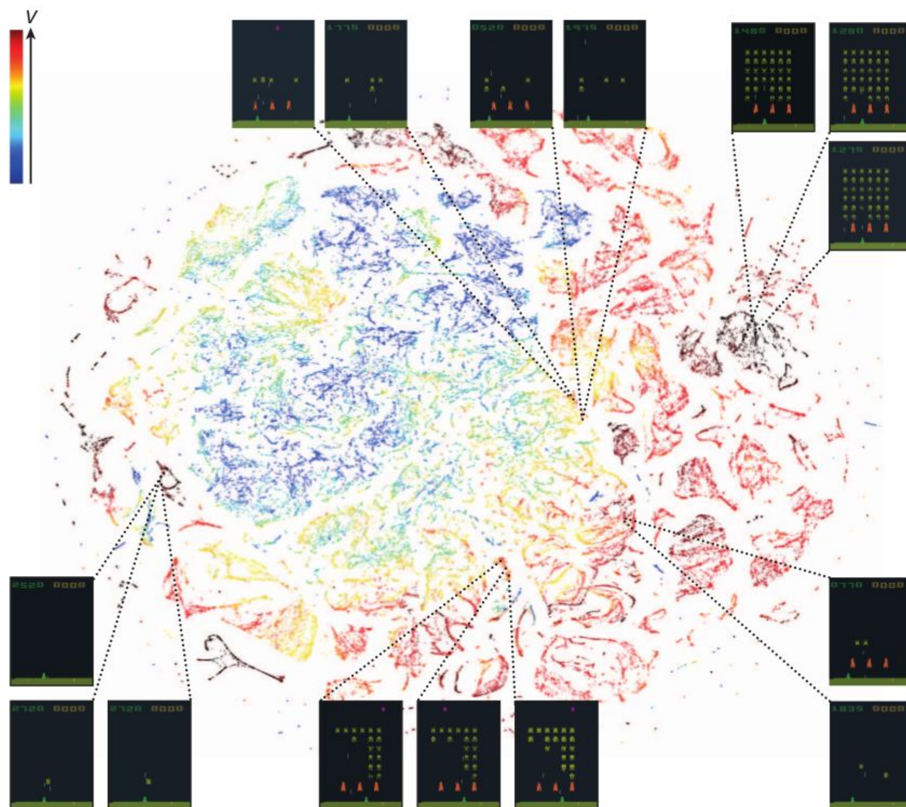
(Kiros, Salakhutdinov, Zemel, TACL 2015)

Neural Networks

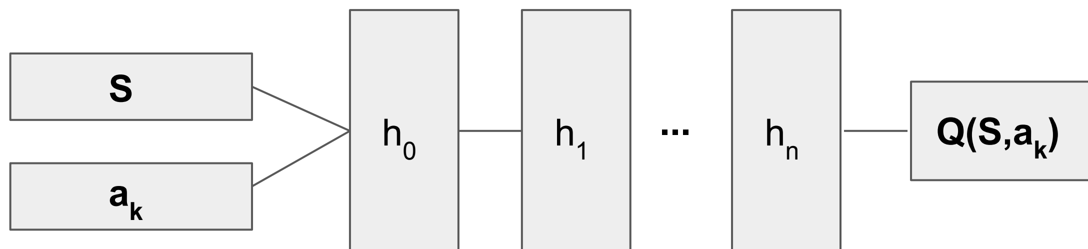
Original photo (raw input) ~ Reference photo (latent representation) = Result

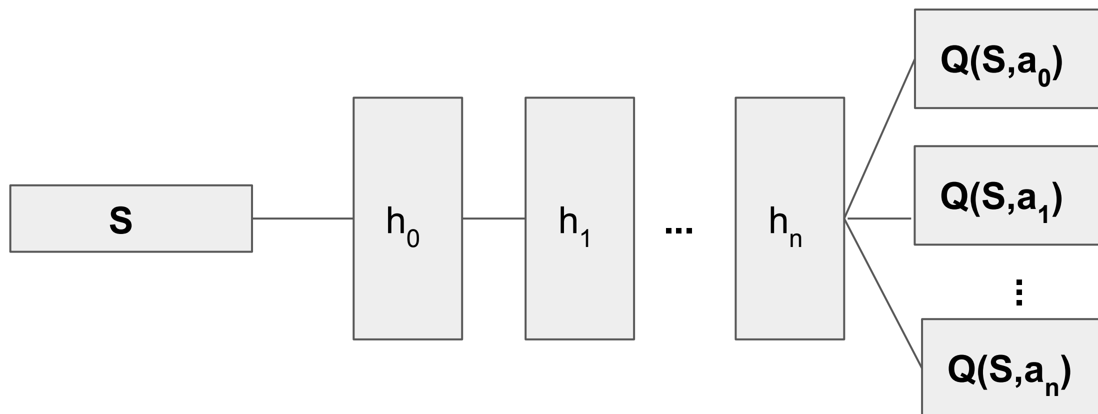
Neural Networks

Can "bin" states better than us



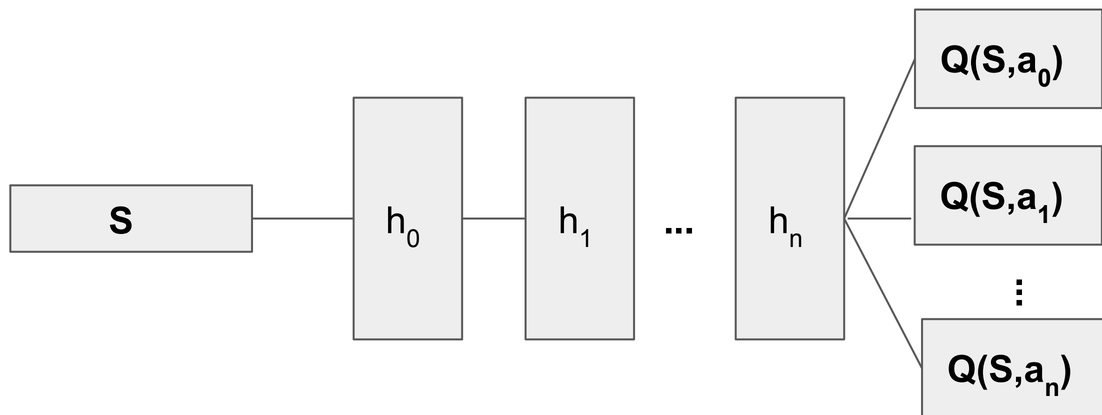
Deep Reinforcement Learning





DQN

- Experience Replay
- Target Network

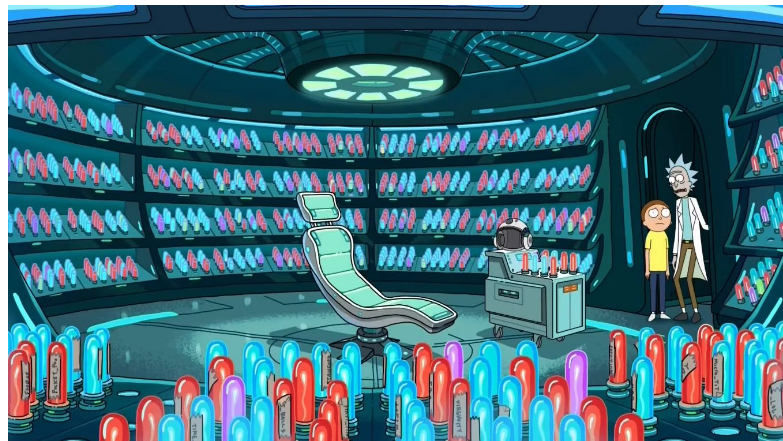


DQN: Experience Replay

Store the last K Experience Tuples,

....

$S_{111}, a_{111}, r_{111}, S'_{111}$
$S_{112}, a_{112}, r_{112}, S'_{112}$
$S_{113}, a_{113}, r_{113}, S'_{113}$
$S_{114}, a_{114}, r_{114}, S'_{114}$
$S_{115}, a_{115}, r_{115}, S'_{115}$
$S_{116}, a_{116}, r_{116}, S'_{116}$
$S_{117}, a_{117}, r_{117}, S'_{117}$
$S_{118}, a_{118}, r_{118}, S'_{118}$
$S_{119}, a_{119}, r_{119}, S'_{119}$
$S_{120}, a_{120}, r_{120}, S'_{120}$
S_n, a_n, r_n, S'_n



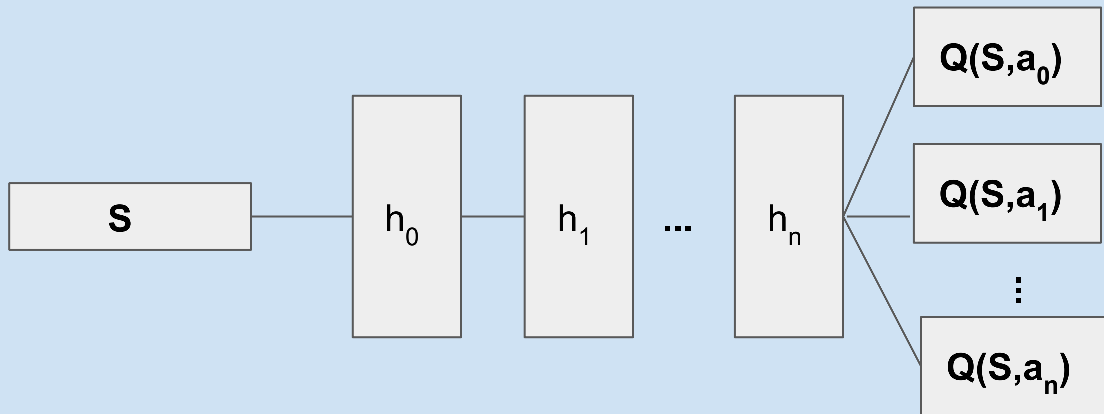
DQN algorithm

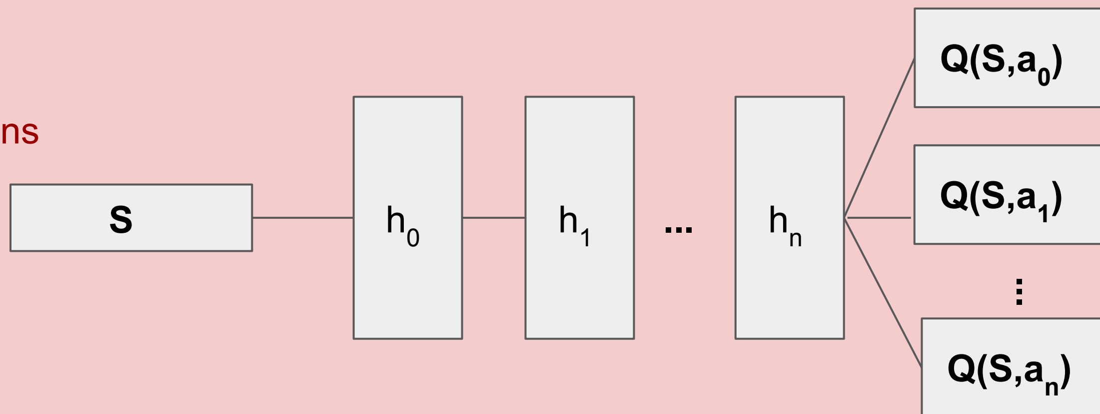
Algorithm 1 Deep Q-learning with Experience Replay

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

DQN: Target Network

Copy weights over every N timesteps





Questions?

Environments

Usually tied to specific tasks.

Episodic (has terminal state)

-The env can be "played through" repeatedly to completion.

-Each complete play through is called an episode.

-Eg; Atari, college, picking up objects in the physical world.

Continual (no clear terminal state)

-Never really ends.

-eg; financial markets, sandbox/exploration games, world wide web.