A Brief Introduction to Reinforcement Learning

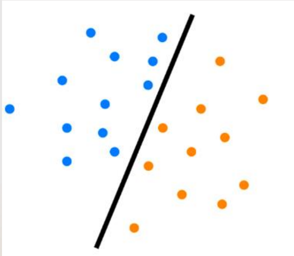
SAIL ON – 27 MAY 2017

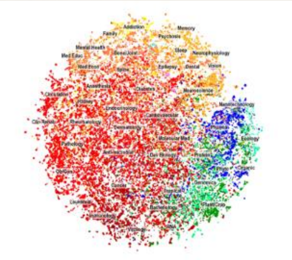
PTOMAN@STANFORD.EDU

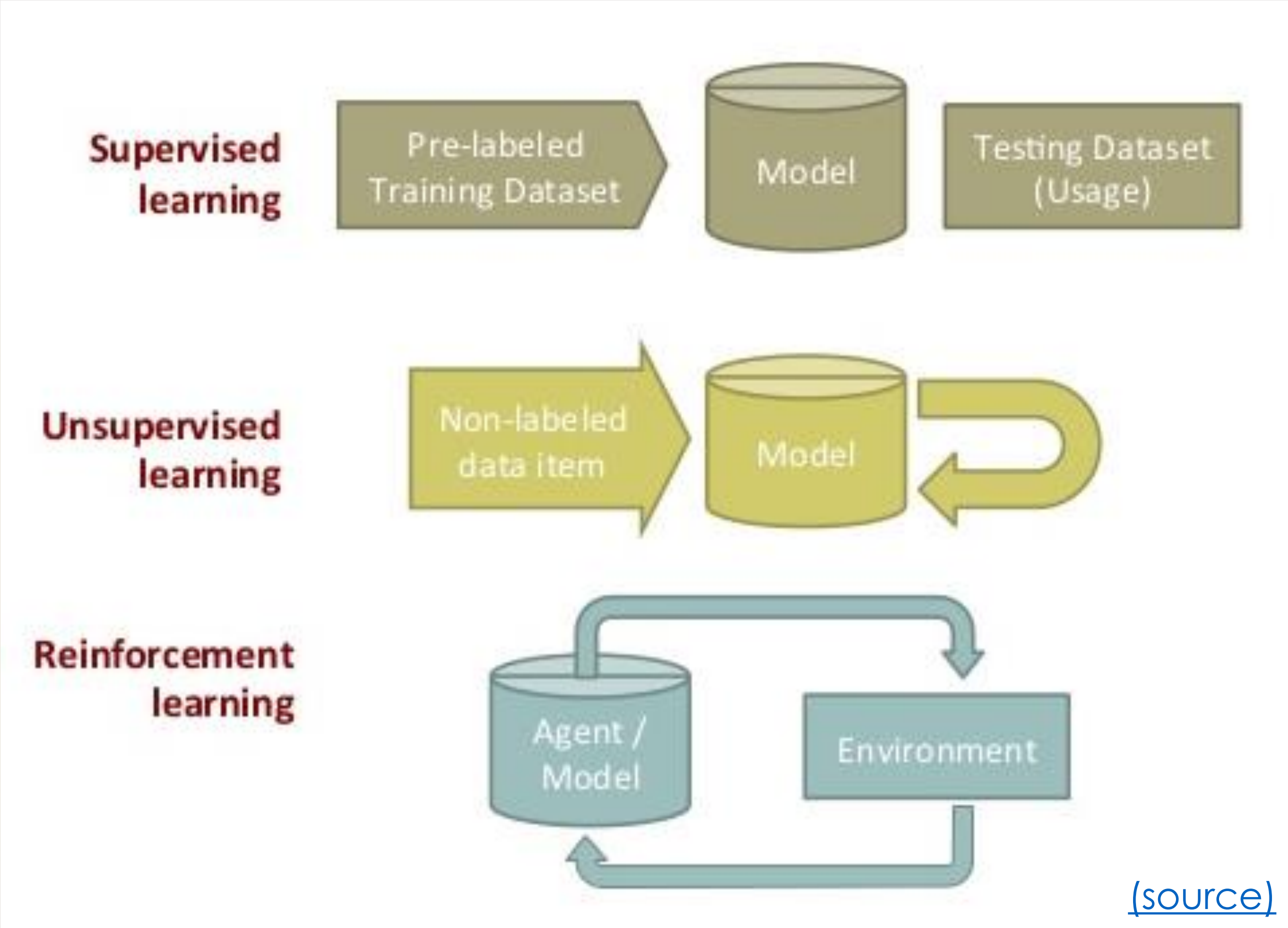
At 8:00 pm, you will be able to…

- **Competence** (able to perform on your own, with varying levels of perfection):
  - Describe how reinforcement learning works and how RL differs from supervised learning
  - Diagnose when RL vs. supervised learning is appropriate for a problem
  - Know that we can model the quality of an action with a lookup table or a more complex function like a neural network; name a weakness of lookup tables that models/functions fix
  - Explain an upper confidence bound and why it is better than choosing actions greedily
  - Articulate similarities and differences in how reinforcement learning and humans solve problems
- **Exposure** (aware):
  - Be aware of the three major paradigms of machine learning
  - Be aware of 1) how we mathematically formalize RL problems, and 2) how we map actual problems to the formalism
  - Have been talked through the Q-learning update rule
  - Be familiar with terms from reinforcement learning: *agent, timestep, policy, Q-learning, discounting, discretization, rollout, exploration/exploitation tradeoff, upper confidence bound*
  - Be familiar with three touchstone RL problems: gridworlds/mazes, cart-pole, Atari games
  - Know where to go for additional RL resources and Jupyter notebooks

There are three types of machine learning for AI







- Labeled data
- Direct feedback
- Predict outcome/future

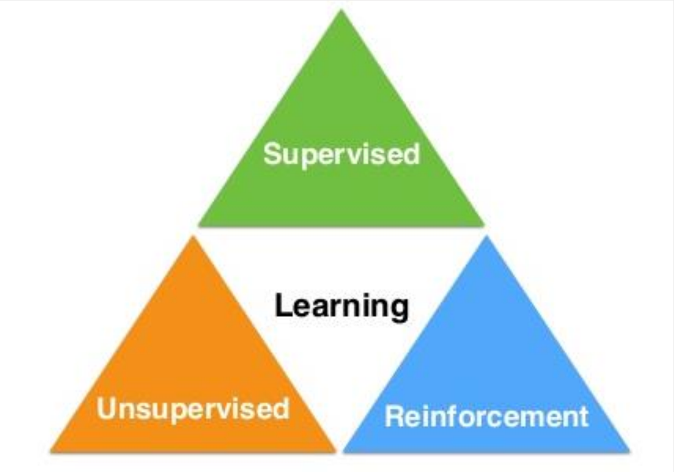
- No labels
- No feedback
- Find hidden structure

- Decision process
- Reward system
- Learn series of actions

How is reinforcement learning special?

- In reinforcement learning, agents take *actions* in their *environments* to maximize *rewards*
- This approach is based on behaviorist psychology

- There is no supervisor – only a *reward signal*
- Feedback is delayed, not instantaneous
- Time really matters (sequential, non-i.i.d. data)
- Agent's actions affect the subsequent data it receives

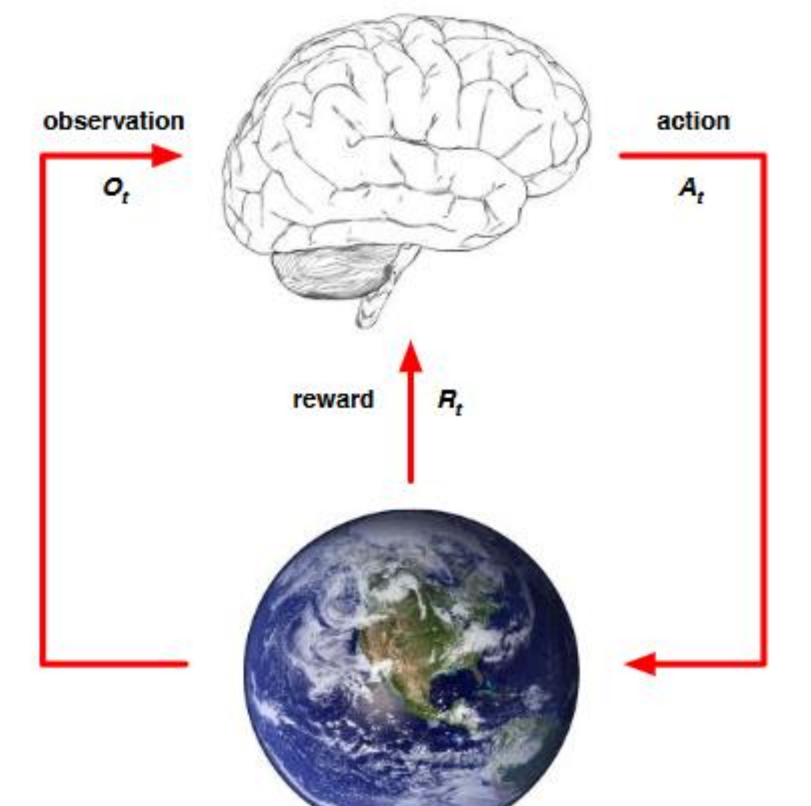


(source)

Silver, UCL

In reinforcement learning, we receive payoffs

- The reinforcement learning paradigm:
  - At timestep t, the agent is in state $s_{t.}$
  - It receives observation $o_{t.}$
  - It chooses action $a_{t.}$
  - After its action:
    - the agent is in the new state s_{t+1}
    - it also receives reward r_{t+1}
  - Then the agent receives a new observation o_{t+1}….



Silver, UCL

Agents learn to maximize rewards

- The reward r_t is a feedback signal
- It indicates how well the agent is doing at time t
- The agent's job is to maximize rewards

- RL is based on the *reward hypothesis*:

All goals can be described by
the maximization of cumulative reward.

(Do you agree?)

Silver, UCL

What approach should we use?

source

Which approach(es) can we choose if we want to….

1. Fly stunt maneuvers in an airplane
2. Transcribe speech
3. Manage an investment portfolio
4. Choose what ad to show someone
5. Rank potentially good-fit colleges for a student
6. Identify a treatment for a sick patient
7. Make a humanoid robot walk

Our (current) toolkit:

- reinforcement learning
- supervised learning
  - classification
    (Naïve Bayes, neural networks)
  - regression
    (linear regression, neural networks)

Silver, UCL

Agents make sequential (hopefully optimal) decisions

- *The agent's goal:* Choose actions that maximize total future rewards
- "Solving" a RL problem means "building an agent that acts optimally"

- Why is this hard?
  - Actions may have long-term consequences
  - Reward may be delayed
  - We may want to sacrifice immediately to get longer-term reward

One way to solve a RL problem: Q-learning

- Q-learning estimates the "quality" of *taking action* a_t *from state* s_t
- Q is a matrix that estimates quality

Q-learning Algorithm

1. Initialize a matrix Q(s,a)
2. Repeat (until convergence):
    1. Given s_t, choose an a_t by using Q
    2. Take action a_t
    3. Observe new state s_{t+1} and reward r_{t+1}
    4. Update Q(s_t, a_t)
    5. Set $s_t = s_{t+1}$

- Q-learning applied to a simple gridworld

Value iteration update rule

Q-learning Algorithm

1. Initialize a matrix Q(s,a)
2. Repeat (until convergence):
    1. Given s_t, choose an a_t by using Q
    2. Take action a_t
    3. Observe new state s_{t+1} and reward r_{t+1}
    4. Update Q(s_t, a_t)
    5. Set $s_t = s_{t+1}$

- The Q-learning update rule:

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

- What is going on here?
    - We increase the quality of the (a_t , s_t) pair
    - We use the amount of change, mediated by the learning rate α_t
    - The amount of change reflects the just-learned quality of landing in s_{t+1}: the immediate reward r_{t+1} plus our estimate of how useful being in s_{t+1} is for getting even more rewards in the future
- What is this gamma γ?

Discounting (γ)

- Would you rather have $10 now or in one month?

- People (and algorithms) value receiving the same good sooner more than they value receiving it later
- The *discount factor* gamma trades off the importance of sooner versus later rewards
- Gamma must be between 0 and 1; it's usually ~0.9 or ~0.99

- What happens if we don't have gamma?

How do we build a policy?

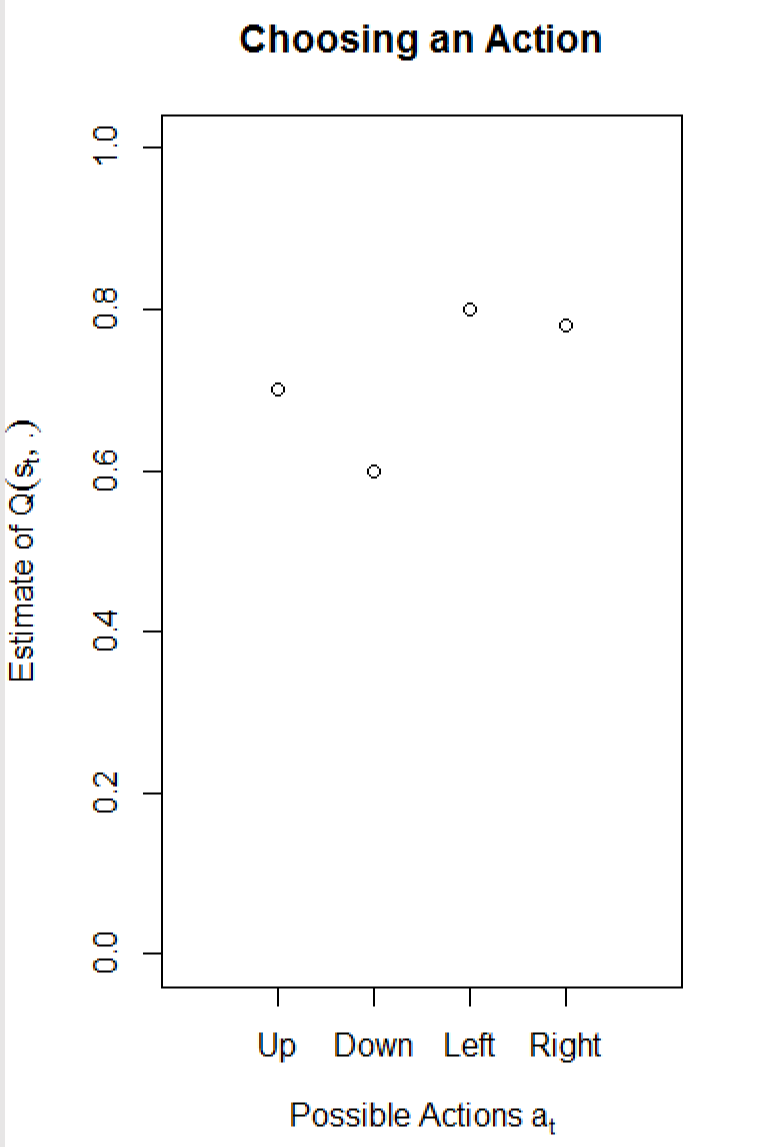
- Let's say from state s_t , we have the following row of Q:

Up	Down	Left	Right
0.7	0.6	0.8	0.78

- What should the agent's *policy* be?
(With what probability should it take each action?)

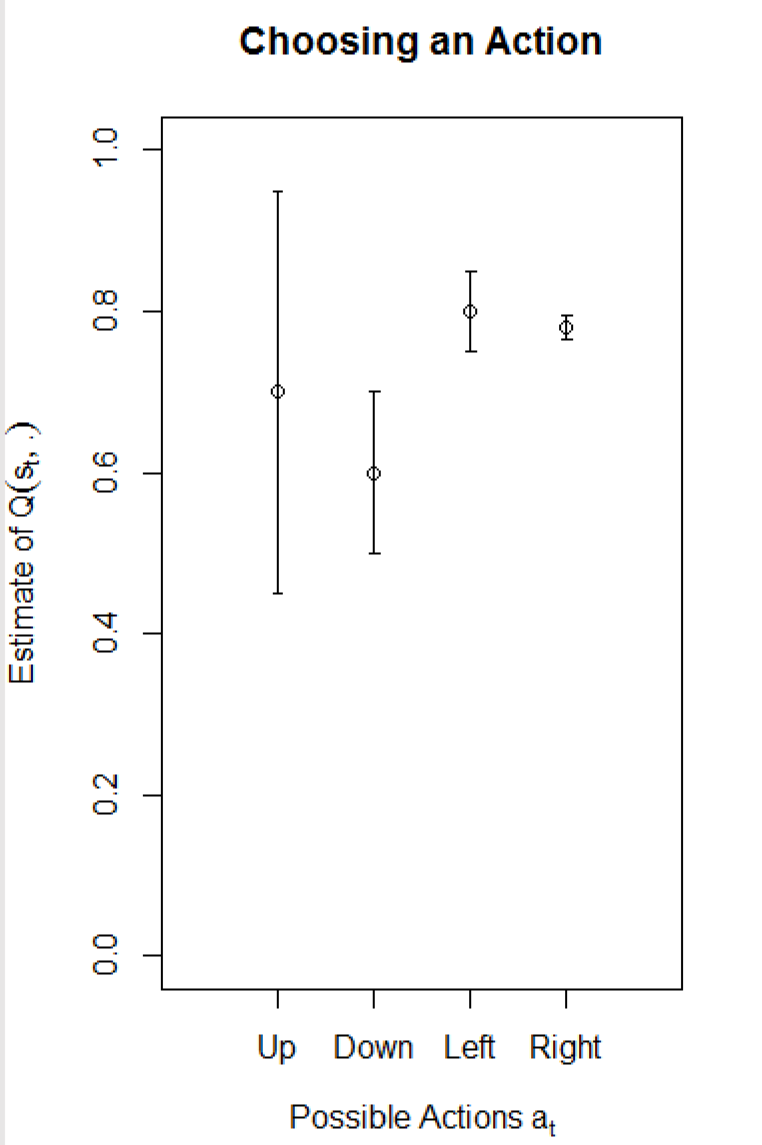
Q-learning Algorithm

1. Initialize a matrix Q(s,a)
2. Repeat (until convergence):
    1. Given s_t, choose an a_t by using Q
    2. Take action a_t
    3. Observe new state s_{t+1} and reward r_{t+1}
    4. Update Q(s_t, a_t)
    5. Set $s_t = s_{t+1}$



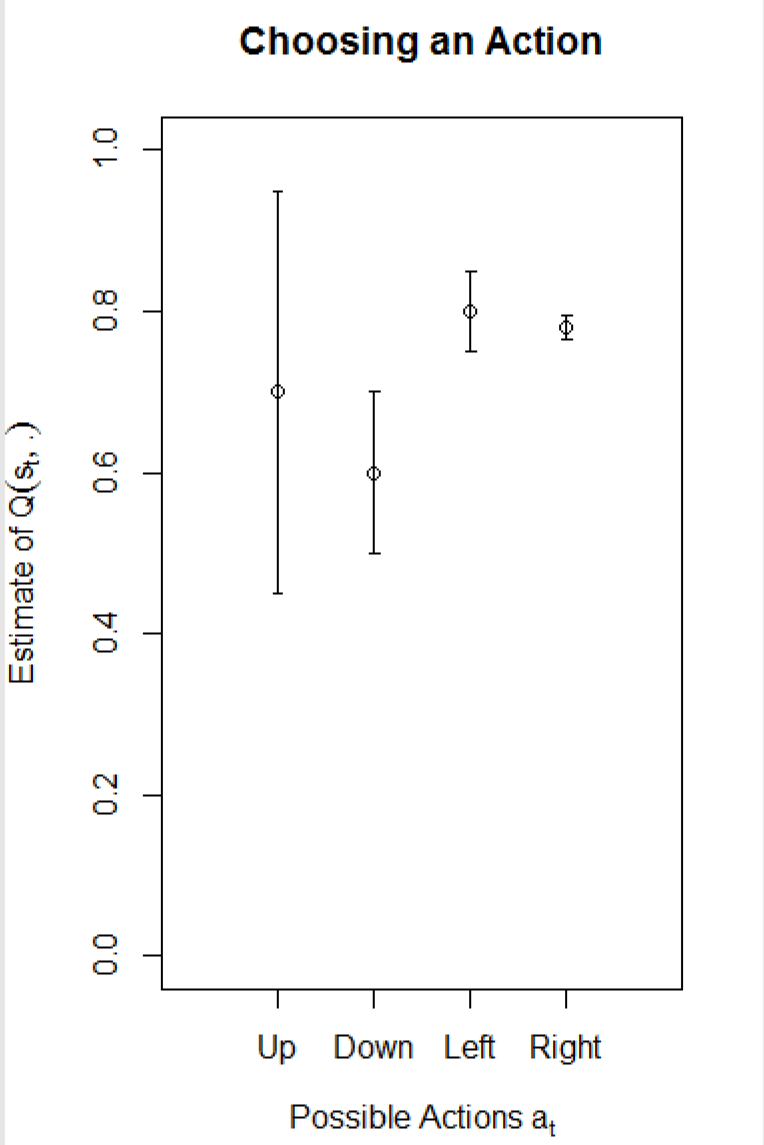
Exploration vs. exploitation

- There is a tradeoff between *exploration* and *exploitation*
  - *Exploration*: learn more information about the environment
  - *Exploitation*: exploit known information to maximize reward
- What are some real-life examples?
- Sometimes we have principled methods to make this tradeoff
- Sometimes we use heuristics
- Given Q estimates at right, what should agent do?

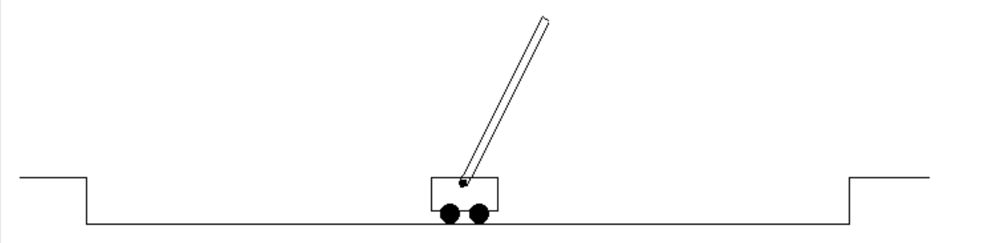


Confidence bounds are more informative than points

- It is common to use the *Upper Confidence Bound (UCB)*
- UCB balances exploration and exploitation:
  - Actions we know little about have *huge* confidence bounds, so by UCB, they look great – we explore them
  - Whenever we try an action, our confidence bounds shrink: we become more and more sure of its true quality
  - Eventually the bounds are pretty tight, and the agent consistently exploits the optimal action
- Other options:
  - Risk-averse problems might want best Lowest Conf. Bound
  - A greedy approach is to use the best single *point* (mean)
- "Dominated" options are worse than the alternatives no matter how we slice it



Introduction to cart-pole



- Consider cart-pole:
  - *Goal*: Balance the pendulum on the cart
  - *Observations are in 4-D space*: position, velocity, angle, angular velocity
  - *Actions are in 2-D space*: move cart right (+1), move cart left (-1)
  - *Rewards*: +1 for every timestep the pole does not fall
- How can we use Q-learning for this problem?

What options do we have for getting Q?

- One option is to *discretize* the state space & run Q-learning as before
  - We can make *k* buckets each for position, velocity, angle, angular velocity
  - We develop a policy for how to act given the probabilities on each row of Q
  - We run that policy & update Q as before
- What makes using this approach hard?
- Do we have any alternatives?

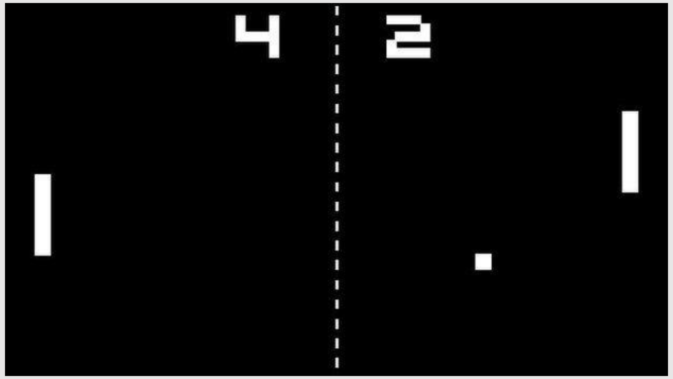
A neural network or other approach can model Q

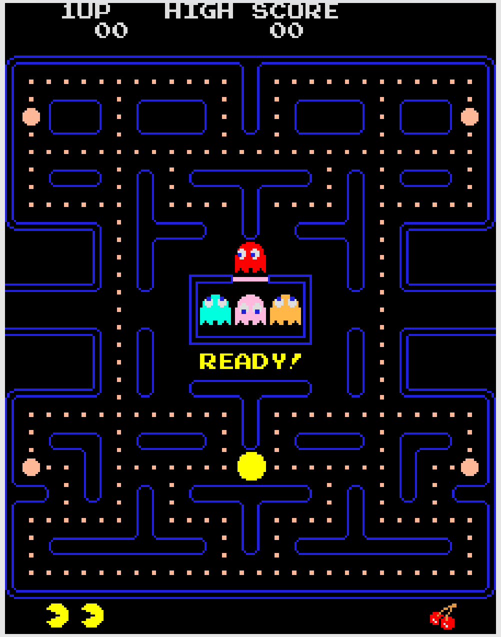
- Another option is to model Q as a function: given the 4-D input state, we build a function that estimates how good each action is
- With this approach…
  - No discretization necessary! (Not even a *path* to discretization is necessary!)
  - We can use whatever model we like to produce probabilities for Q(s_t, •)!
  - The model can be arbitrarily complex!
  - We still use the same update rule – only with a model for Q rather than lookups!

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

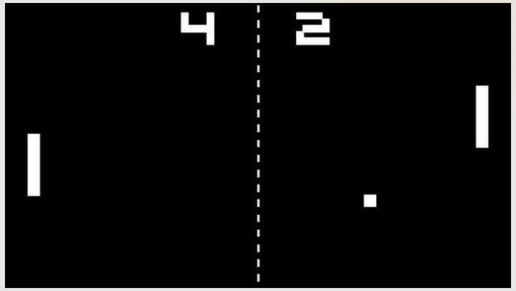
Q-as-function works well as problems get more complex

- How could we represent the state space for Atari games?
- As the problems get more complex, treating Q as a function is useful





Let's formulate Pong with RL….



- What is the space of observations?
- How do we turn observations into states?
  - Convolutional neural networks for vision, whose outputs are probabilities for actions instead of probabilities for Pekingese, Afghans, tables, etc.
  - Motion is key → input is 2 frames, and/or differences of frames
- What is the space of actions?
  - Up/down (maybe also stay put?)
- How do we formulate rewards?
  - -100 if ball passes (maybe also -1 for each move?)
  - The computer will figure out how to satisfy the reward function in unintuitive and perhaps unwanted ways…

Training a reinforcement learning model with rollouts

- Each time we ask the agent to make decisions until the game is over/goal is reached, we call it a *(policy) rollout*

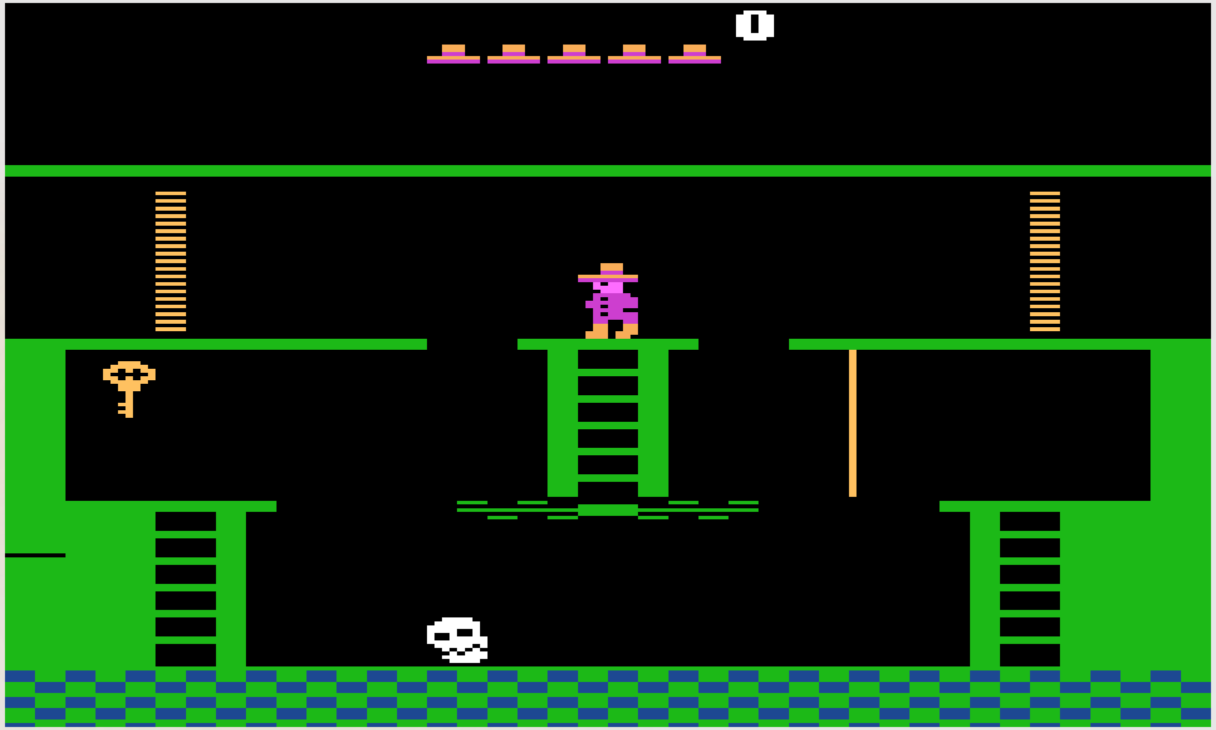
- Training works like this:
  1. Using a single model, we do *n* rollouts
  2. We update the model for Q based on which moves turned out to be good/bad
  3. We repeat until the model performs well

- How big should *n* be?
- How fast does this approach learn?

How do we deal with slowness in training RL models?

- Reframe the setup to address sparsity of rewards:
  - Pre-train with supervised learning, then RL to fine-tune
  - "Traces" back over time to every state that contributed along a path
  - Focus exploration on areas that turned out to be *unexpectedly* good
  - Use an *actor-critic algorithm* instead of sampling a single action:
    - Model both the "goodness" of actions (actor) and the "goodness" of each outcome state (critic)
    - Apply updates for *all* actions based on estimates of *all* their goodnesses
- Use more compute power:
  - Ask the agent to play itself and/or have multiple copies simultaneously

Some games are harder than others…



Montezuma's Revenge

Frostbite

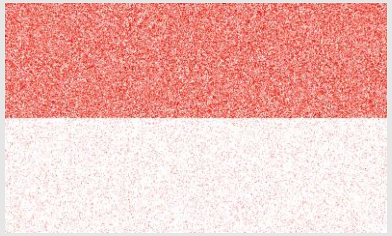
Karpathy

Do RL methods learn like humans?

- In RL, the reward function is *discovered* – humans are told
- In RL, the agent starts *from scratch* each time – humans have background knowledge (physics, psychology, context)
- In RL, the solution is found via *brute force* repetitive experiences of both good and bad outcomes – humans make inferences

- RL could work equally well if pixels were permuted or the reward function was chosen at random – humans would fail

- So… what does this all mean for "artificial intelligence"?
- When does each approach have an advantage?



Art Karpathy

Reinforcement learning is popular and growing… (but not the only AI game in town)

- Some visual demos of RL math in practice:
  - Helicopter stunts (2008)
  - Follow a road (2011)
  - Avoid obstacles (2012)
  - Breakout (computer game) (2014)
  - Robot playing with Legos (2015)
  - Quadruped locomotion (2016)
  - Doom (computer game) (2016)
  - Biped locomotion (2017)
- Traditional robotics (control theory & kinematics math – not RL math):
  - BigDog (2010)
  - SpotMini (2016)

Resources

- Explanations with code:
  - Johannes Rieke's Jupyter notebook to solve a maze with RL
  - Aurélien Géron's Jupyter notebook to accompany a book teaching RL through cart-pole and PacMan
  - Andrej Karpathy's blog post on RL with neural networks to play Pong
- Prose explanations:
  - David Silver's introductory lecture slides in a course on general RL
  - Nervana post on Deep Q-learning
  - Solving Montezuma's Revenge: Kulkarni and Narasimhan et al. (2016)
- OpenAI Gym, for exploring RL and comparing your implementation's performance on benchmark problems

At 8:00 pm, you are able to…

- **Competence** (able to perform on your own, with varying levels of perfection):
  - Describe how reinforcement learning works and how RL differs from supervised learning
  - Diagnose when RL vs. supervised learning is appropriate for a problem
  - Know that we can model the quality of an action with a lookup table or a more complex function like a neural network; name a weakness of lookup tables that models/functions fix
  - Explain an upper confidence bound and why it is better than choosing actions greedily
  - Articulate similarities and differences in how reinforcement learning and humans solve problems
- **Exposure** (aware):
  - Be aware of the three major paradigms of machine learning
  - Be aware of 1) how we mathematically formalize RL problems, and 2) how we map actual problems to the formalism
  - Have been talked through the Q-learning update rule
  - Be familiar with terms from reinforcement learning: *agent, timestep, policy, Q-learning, discounting, discretization, rollout, exploration/exploitation tradeoff, upper confidence bound*
  - Be familiar with three touchstone RL problems: gridworlds/mazes, cart-pole, Atari games
  - Know where to go for additional RL resources and Jupyter notebooks