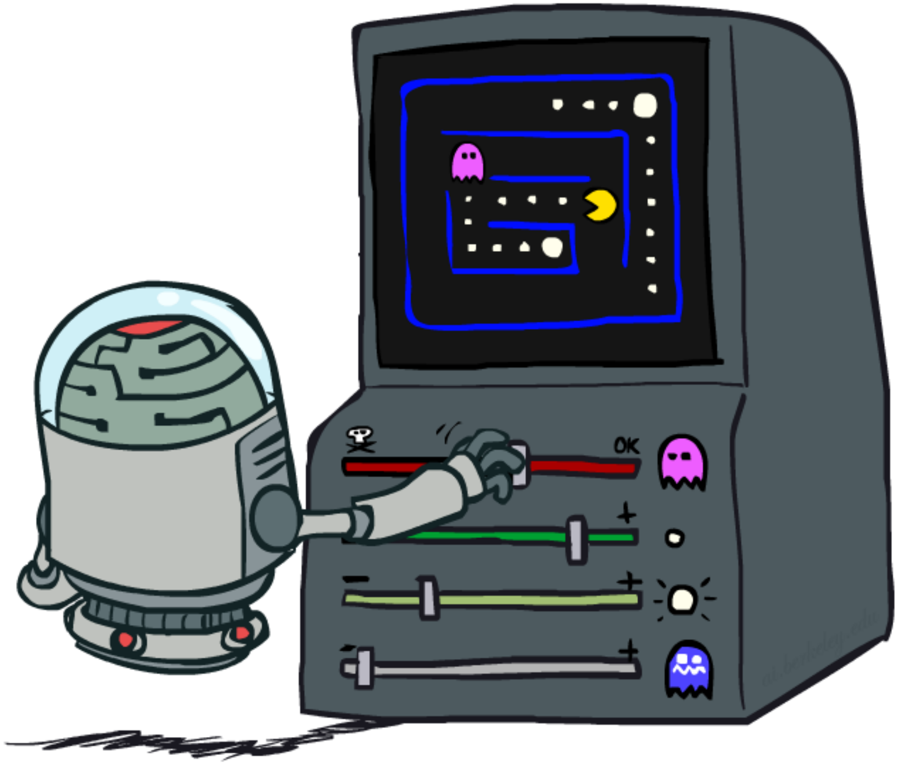
# CS 4100 // artificial intelligence

instructor: byron wallace




*Reinforcement learning II*

**Attribution**: many of these slides are modified versions of those distributed with the *UC Berkeley CS188* materials
Thanks to John DeNero and Dan Klein

# Reinforcement learning

Still assume an underlying Markov decision process (MDP) – we just don't know the parameters!:

- A set of states $s \in S$
- A set of actions (per state) A
- A model T(s,a,s')
- A reward function R(s,a,s')

And we're still looking for a policy $\pi(s)$

New twist: **don't know T or R**

- So we don't know which states are good or what the actions do
- Must actually try actions and states out to learn

# The Story So Far: MDPs and RL

## Known MDP: Offline Solution

| Goal | Technique |
|---|---|
| Compute V*, Q*, $\pi$* | Value / policy iteration |
| Evaluate a fixed policy $\pi$ | Policy evaluation |

## Unknown MDP: Model-Based

| Goal | Technique |
|---|---|
| Compute V*, Q*, $\pi$* | VI/PI on approx. MDP |
| Evaluate a fixed policy $\pi$ | PE on approx. MDP |

## Unknown MDP: Model-Free

| Goal | Technique |
|---|---|
| Compute V*, Q*, $\pi$* | Q-learning |
| Evaluate a fixed policy $\pi$ | Value Learning |

# The Story So Far: MDPs and RL

## Known MDP: Offline Solution

| Goal | Technique |
| --- | --- |
| Compute V*, Q*, $\pi$* | Value / policy iteration |
| Evaluate a fixed policy $\pi$ | Policy evaluation |

## Unknown MDP: Model-Based

| Goal | Technique |
| --- | --- |
| Compute V*, Q*, $\pi$* | VI/PI on approx. MDP |
| Evaluate a fixed policy $\pi$ | PE on approx. MDP |

## Unknown MDP: Model-Free

| Goal | Technique |
| --- | --- |
| Compute V*, Q*, $\pi$* | **Q-learning** |
| Evaluate a fixed policy $\pi$ | Value Learning |

# Model-Free Learning

Model-free (temporal difference) learning

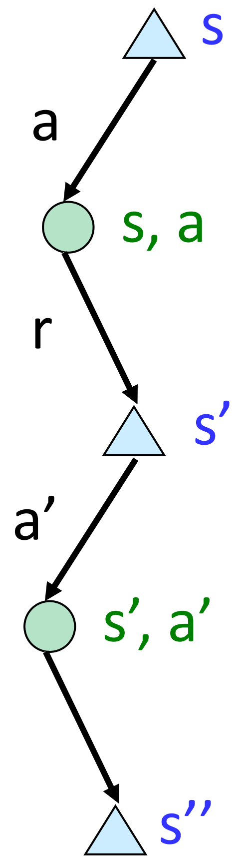
- Experience world through episodes

$$(s, a, r, s', a', r', s'', a'', r'', s'''' \ldots)$$

- Update estimates each transition

$$(s, a, r, s')$$

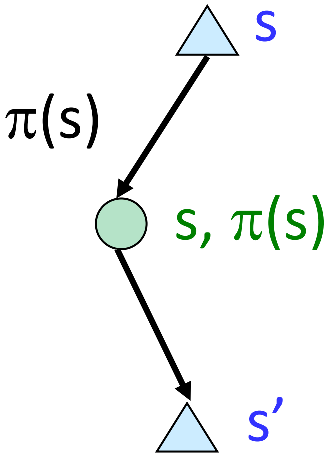
- Over time, updates will mimic Bellman updates



# Last time: Temporal Difference Learning (TDL)

Big idea: learn from every experience!

- Update V(s) each time we experience a transition (s, a, s’, r)
- Likely outcomes s’ will contribute updates more often




Temporal difference learning of values

- Policy still fixed, **still doing evaluation**!
- Move values toward value of whatever successor occurs: running average

Sample of V(s): $sample = R(s, \pi(s), s') + \gamma V^{\pi}(s')$

Update to V(s): $V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + (\alpha) sample$

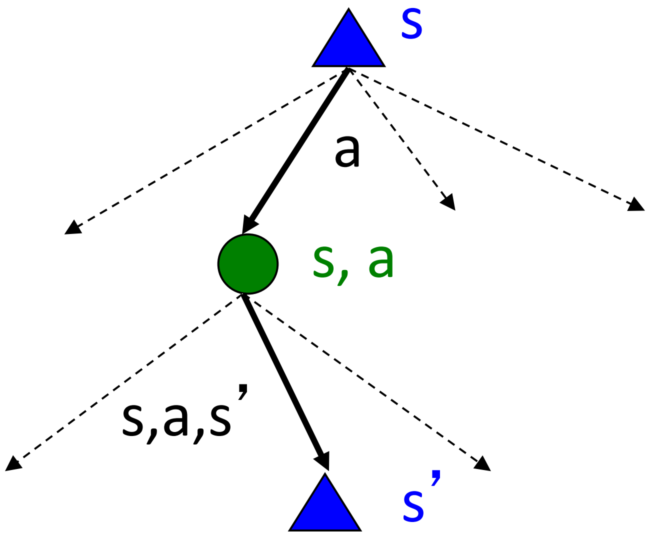
Can rewrite as: $V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$

# Problems with TD value learning

- TD value leaning is a model-free way to do policy evaluation, mimicking Bellman updates with *running sample averages*
- However, if we want to turn values into a (new) *policy*, we're sunk:

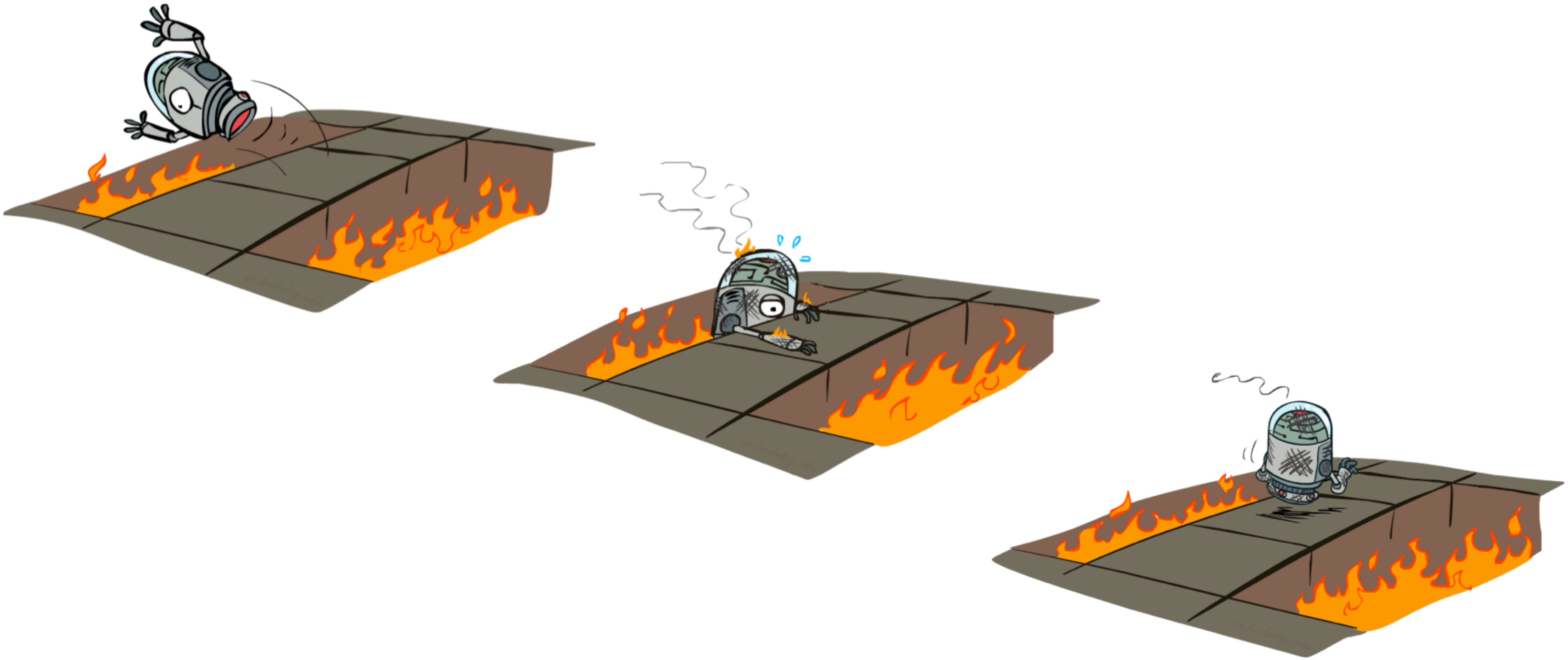
$$\pi(s) = \arg\max_a Q(s,a)$$

$$Q(s,a) = \sum_{s'} T(s,a,s') \left[ R(s,a,s') + \gamma V(s') \right]$$




- Idea: **learn Q-values**, **not values**
- Makes action selection model-free too!

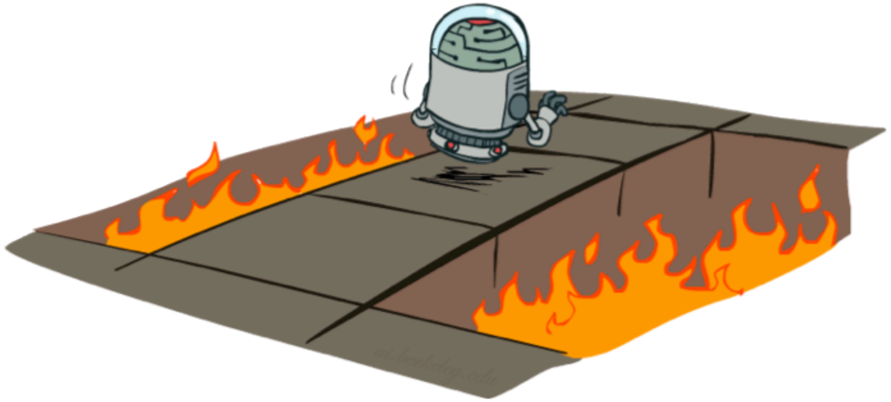
# Active reinforcement learning



# Active reinforcement learning

Full reinforcement learning: optimal policies (like value iteration)

- You don't know the transitions T(s,a,s')
- You don't know the rewards R(s,a,s')
- You choose the actions now
- **Goal: learn the optimal policy / values**



In this case:

- Learner makes choices!
- Fundamental tradeoff: **exploration** vs. **exploitation**
- This is NOT offline planning! You actually take actions in the world and find out what happens… May mean diving into a pit!

# Q-value iteration v. value iteration

Value iteration: find successive (depth-limited) values

- Start with $V_0(s) = 0$, which we know is right
- Given $V_k$, calculate the depth k+1 values for all states:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right]$$

But Q-values are more useful and are just averages! So compute them instead

- Start with $Q_0(s,a) = 0$, which we know is right
- Given $Q_k$, calculate the depth k+1 q-values for all q-states:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')\right]$$

# Q-Learning

Q-Learning: sample-based Q-value iteration

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[ R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$$

Idea: *Learn Q(s,a) values as you go*

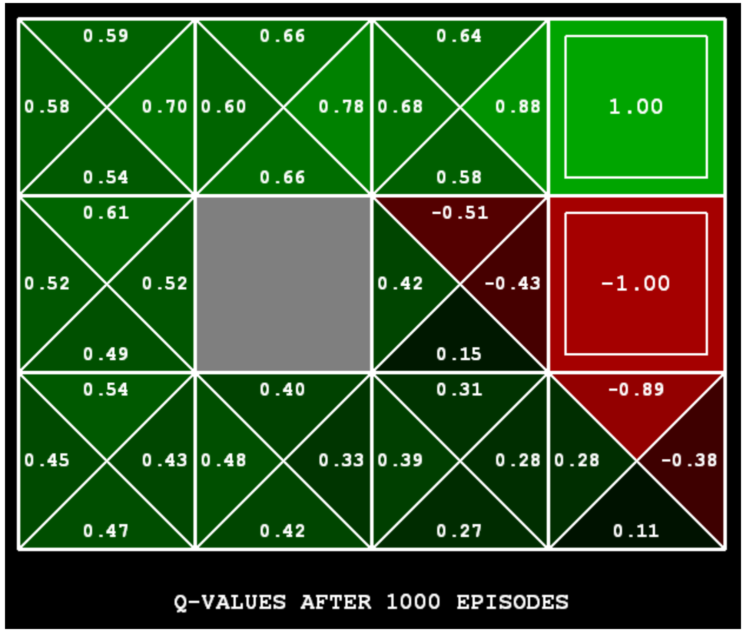
- Receive a sample (s,a,s',r)
- Consider your old estimate: $Q(s,a)$
- Consider your new sample estimate:

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

- Incorporate the new estimate into a running average:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\,[sample]$$

"Temporal difference"



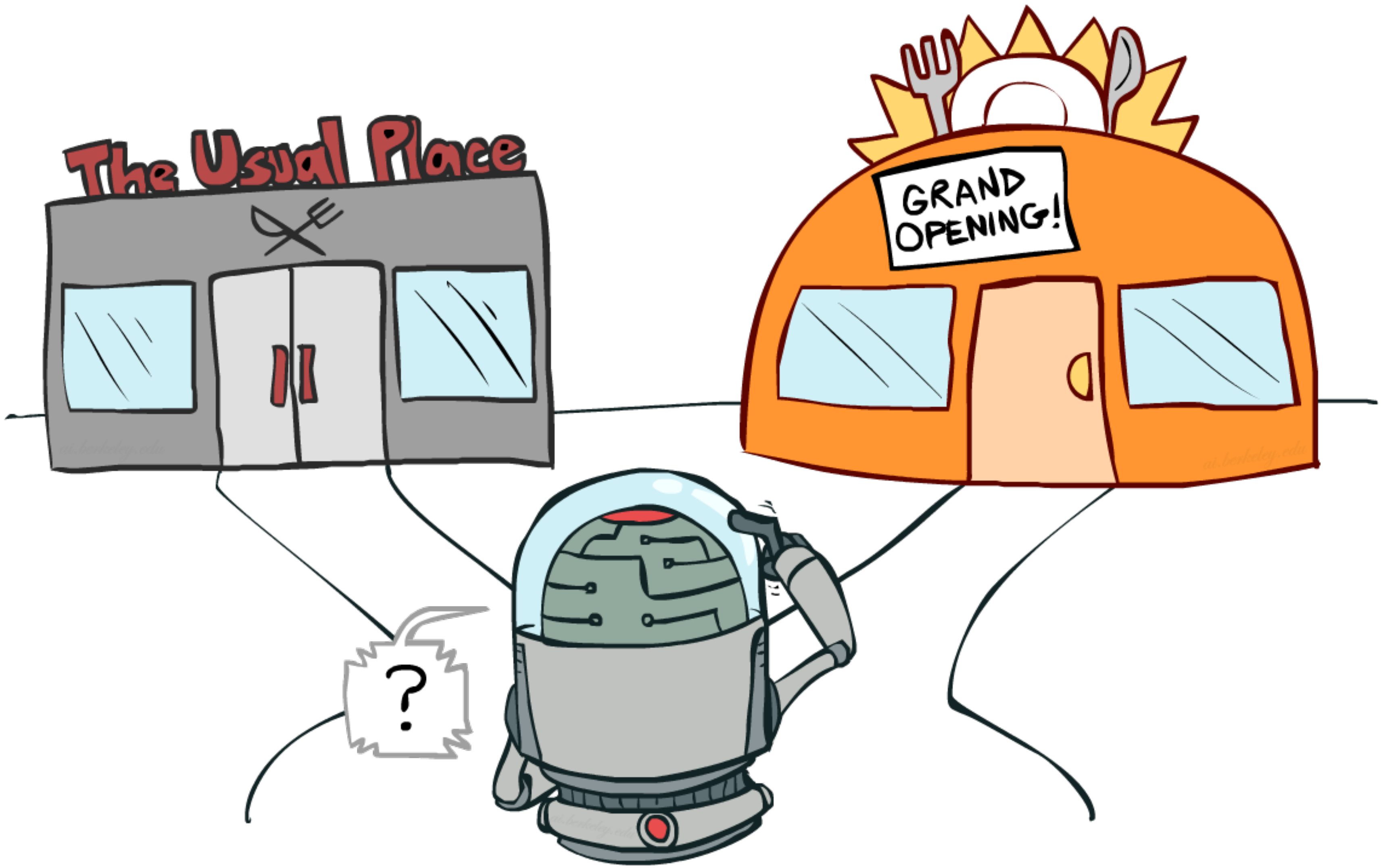
# Q-Learning properties

Q-learning converges to optimal policy -- even if you're acting suboptimally!

Caveats:

- You have to *explore enough*
- You have to eventually make the learning rate small enough
- … but not decrease it too quickly
- Basically, in the limit, it doesn't matter how you select actions (!)

# Exploration vs. exploitation



# How to explore?

Several schemes for forcing exploration

- Simplest: random actions (ε-greedy)
  - Every time step, flip a coin
  - With (small) probability ε, act randomly
  - With (large) probability 1-ε, act on current policy

- Problems with random actions?
  - You do eventually explore the space, but keep thrashing around once learning is done
  - One solution: lower ε over time
  - Another solution: exploration functions

# Exploration functions

## When to explore?

- Random actions: explore a fixed amount
- Better idea: explore areas whose badness is not (yet) established, eventually stop exploring


## Exploration function

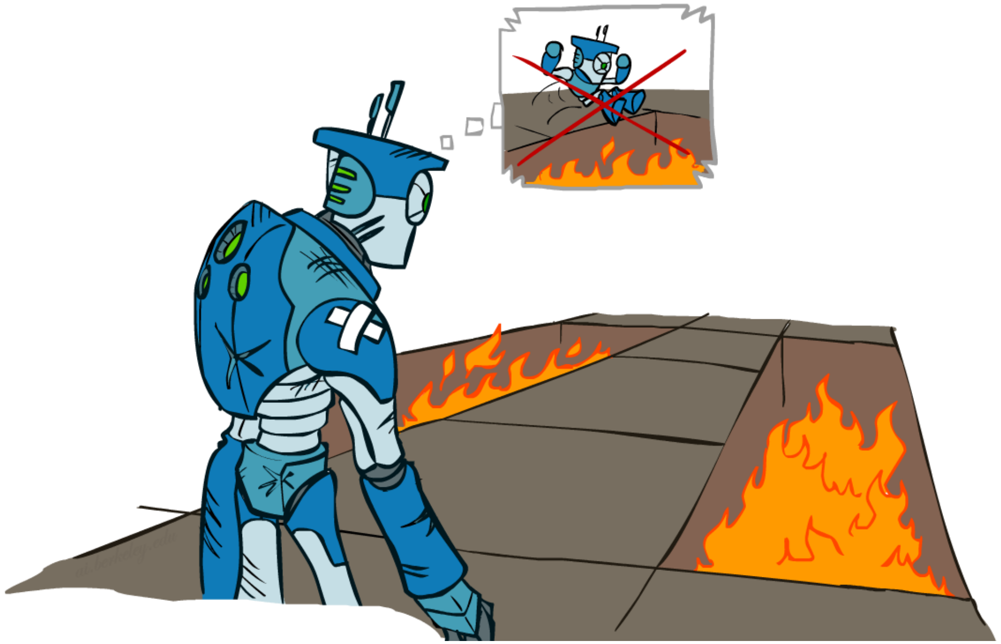
- Takes a value estimate u and a visit count n, and returns an opti utility, e.g. $f(u,n) = u + k/n$

*Regular Q-Update:* $$Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

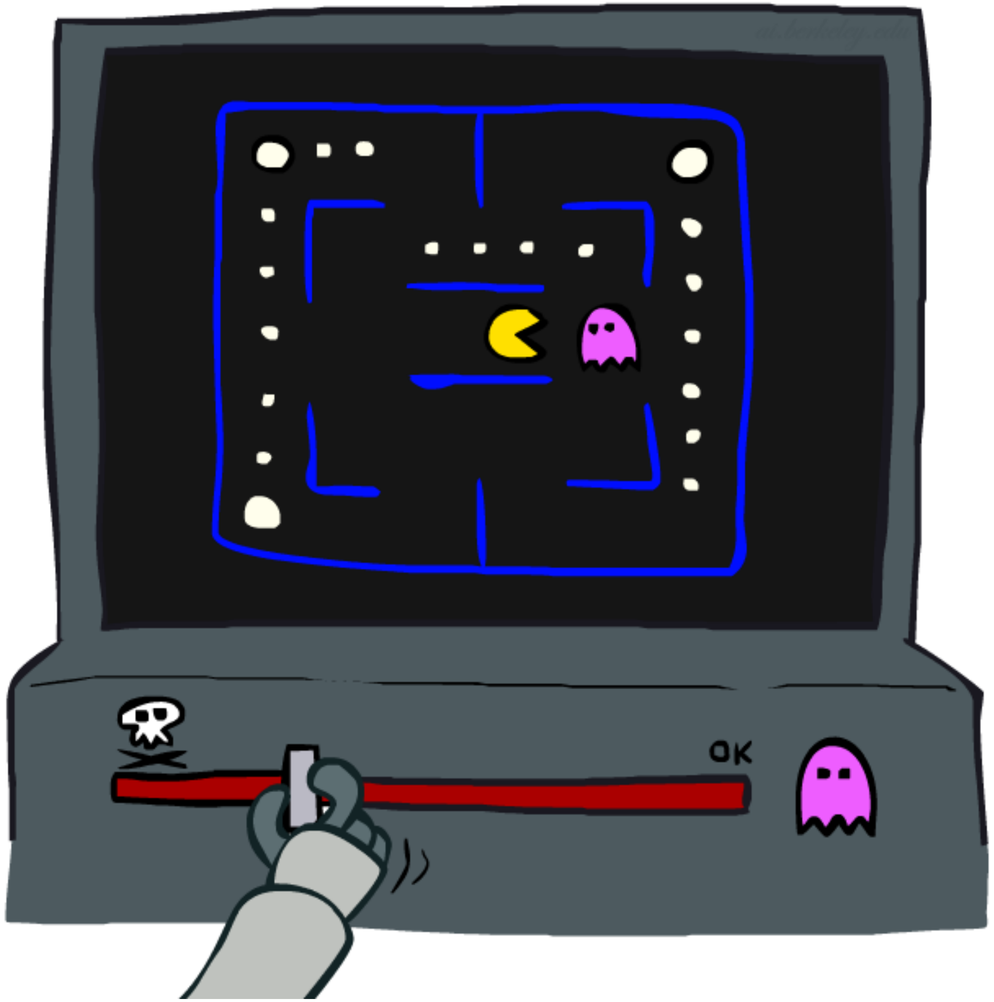
*Modified Q-Update:* $$Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} f(Q(s',a'),N(s',a'))$$

# Regret

- Even if you learn the optimal policy, you still make mistakes along the way!
- Regret is a measure of your total mistake cost: *the difference between your (expected) rewards, including youthful suboptimality, and optimal (expected) rewards*
- Minimizing regret goes beyond learning to be optimal – it requires optimally learning to be optimal
- Example: random exploration and exploration functions both end up optimal, but random exploration has higher regret



# Approximate Q-Learning



# Generalizing across states

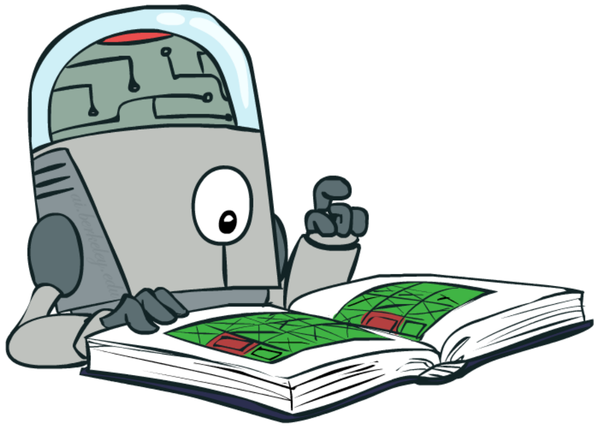
Basic Q-Learning keeps a table of all q-values

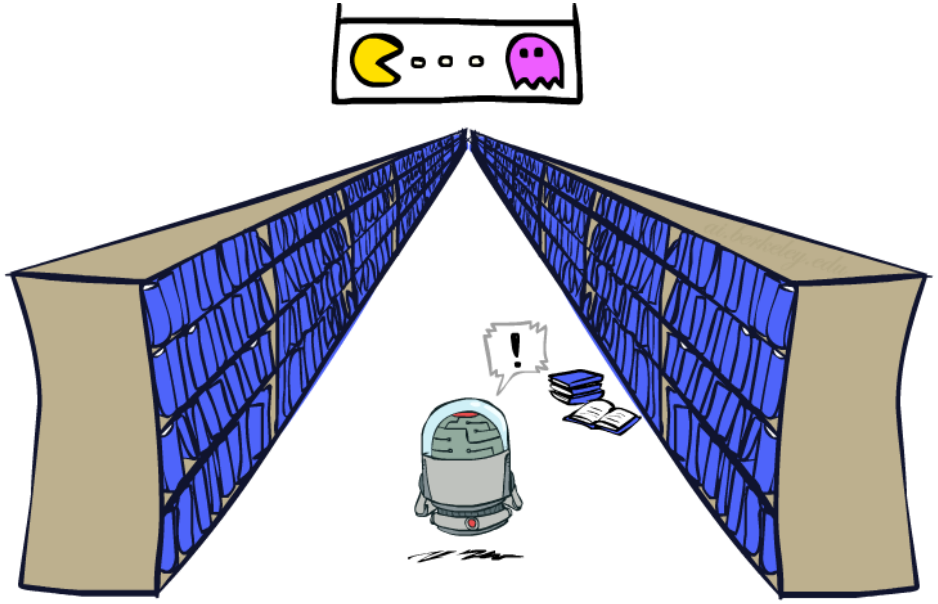
In realistic situations, we cannot possibly learn about every single state!

- Too many states to visit them all in training
- Too many states to hold the q-tables in memory

Instead, we want to generalize:

- Learn about some small number of training states from experience
- Generalize that experience to new, similar situations
- This is a fundamental idea in machine learning, and we'll see it over and over again

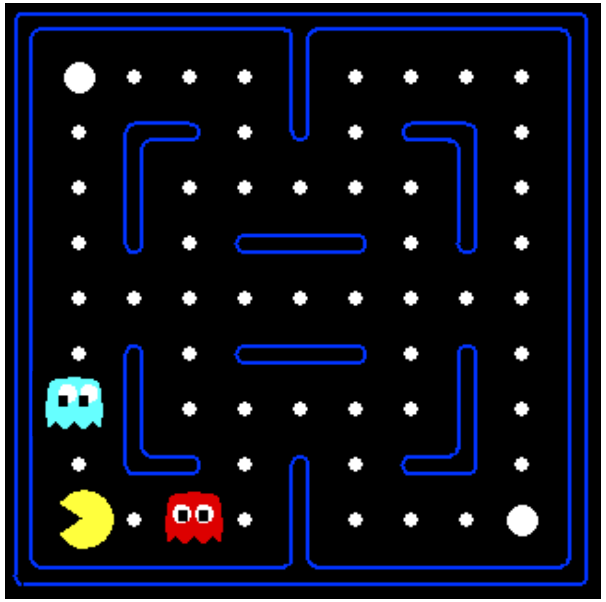




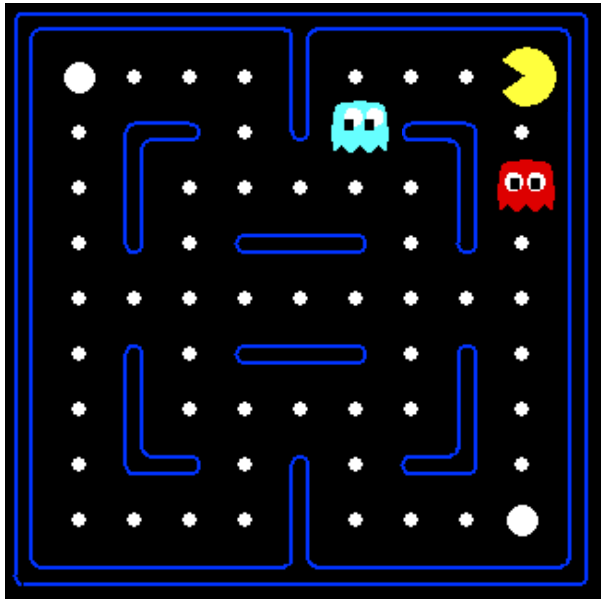
[demo – RL pacman]

# Example: Pacman

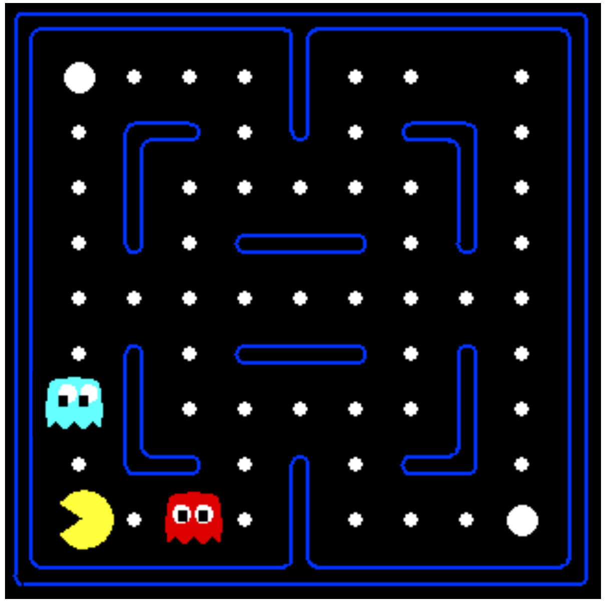
Let's say we discover through experience that this state is bad:



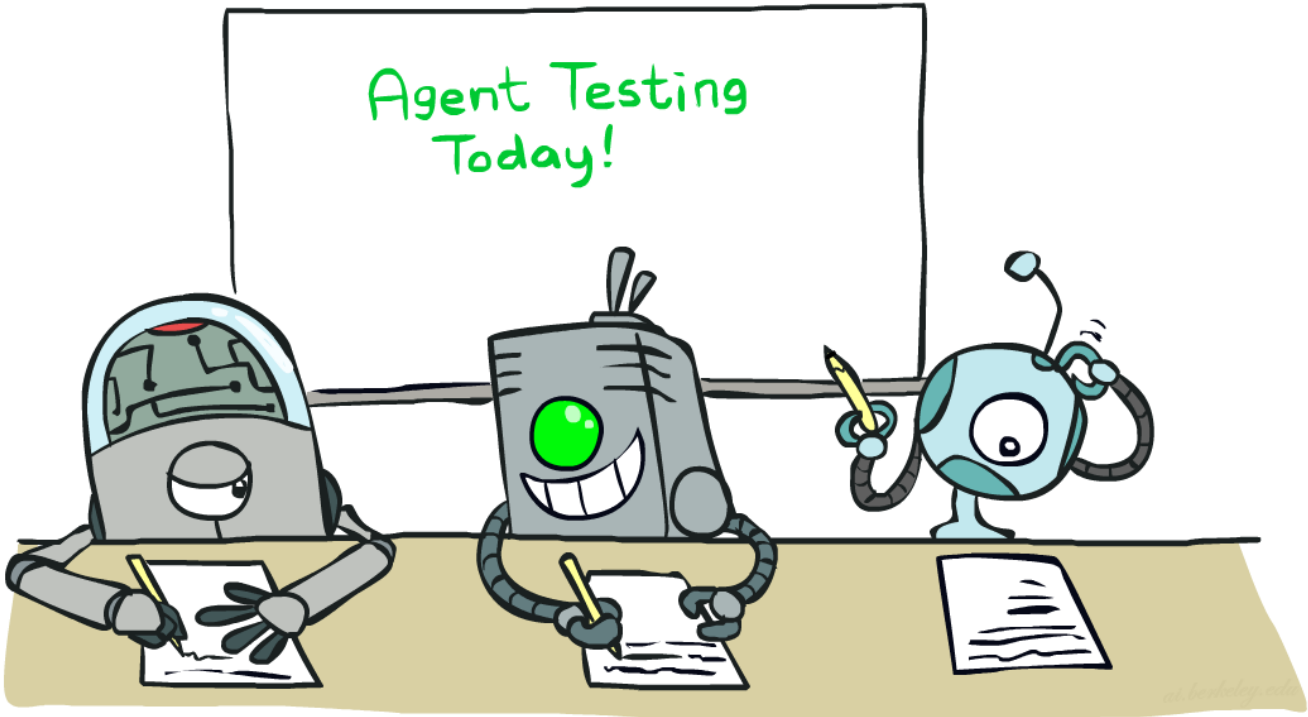
In naïve q-learning, we know nothing about this state:



Or even this one!



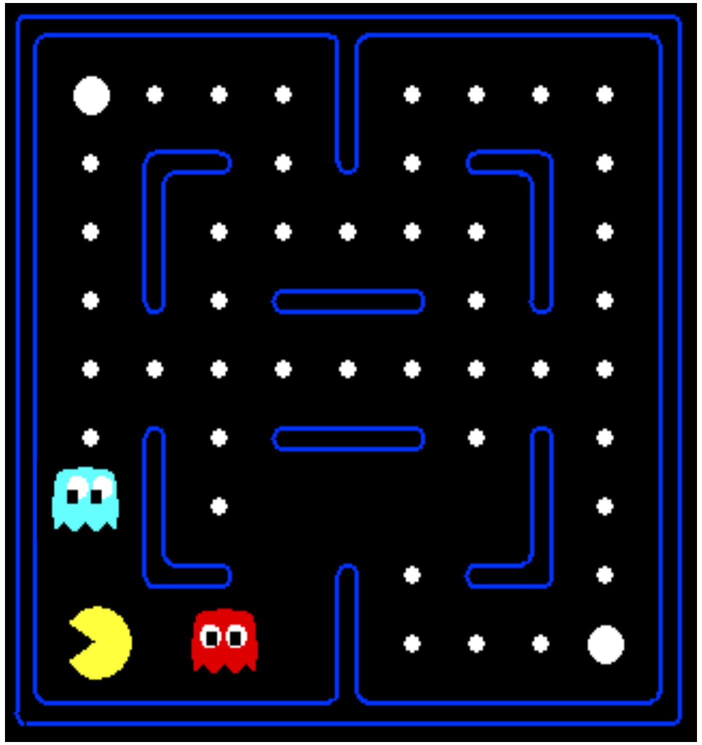
# Enter *machine learning*



# Feature-based representations

Idea: describe a state using a vector of *features* (properties)

- Features are functions from states to real numbers (often 0/1) that capture important properties of the state
- Example features:
  - Distance to closest ghost
  - Distance to closest dot
  - Number of ghosts
  - $1 / (\text{dist to dot})^2$
  - Is Pacman in a tunnel? (0/1)
  - …… etc.
  - Is it the exact state on this slide?
- Can also describe a q-state (s, a) with features (e.g. action moves closer to food)



# Linear value functions

Using a feature representation, we can write a q function (or value function) for any state using a few weights:

$$V(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

Advantage: our experience is summed up in a few powerful numbers

Disadvantage: states may share features but actually be very different in value!

# Approximate Q-Learning

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

Q-learning with linear Q-functions:

$$\text{transition} \;= (s, a, r, s')$$

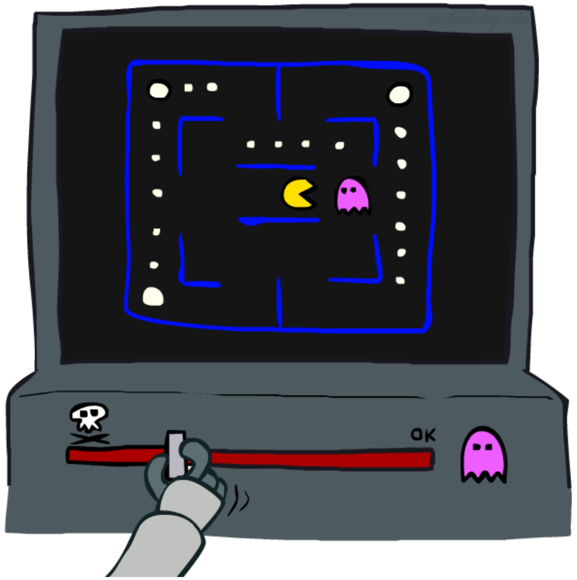
$$\text{difference} = \left[r + \gamma \max_{a'} Q(s', a')\right] - Q(s,a)$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha \,[\text{difference}]$$

Exact Q's

$$w_i \leftarrow w_i + \alpha \,[\text{difference}]\, f_i(s,a)$$

**Approximate** Q's



# Approximate Q-Learning

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

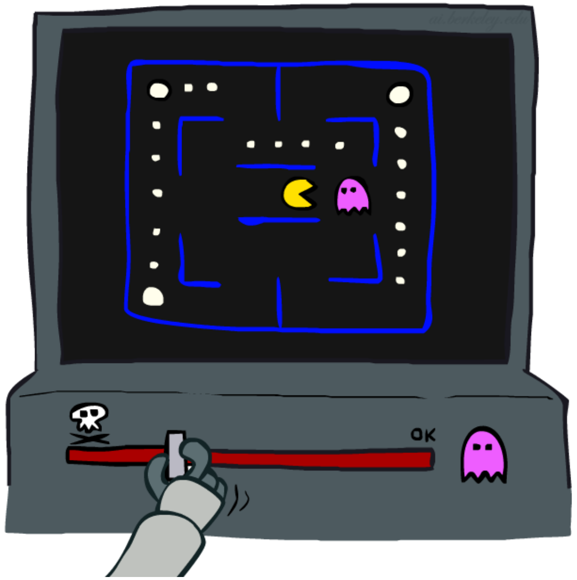
Q-learning with linear Q-functions:

$$\text{transition } = (s,a,r,s')$$

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

$$Q(s,a) \leftarrow Q(s,a) + \alpha\,[\text{difference}] \qquad \text{Exact Q's}$$

$$w_i \leftarrow w_i + \alpha\,[\text{difference}]\, f_i(s,a) \qquad \textbf{Approximate}\text{ Q's}$$




Intuitive interpretation:

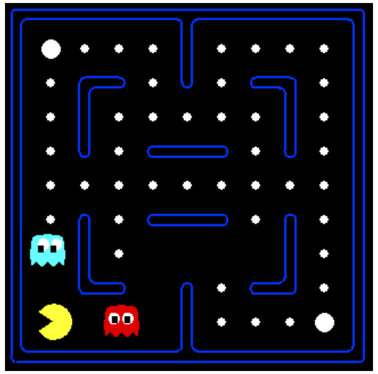
- Adjust weights of active features
- E.g., if something unexpectedly bad happens, blame the features that were on: disprefer all states with that state's features

Formal justification: online least squares (will revisit in a moment!)

# Example: Q-Pacman

$$Q(s,a) = 4.0 f_{DOT}(s,a) - 1.0 f_{GST}(s,a)$$

$s$



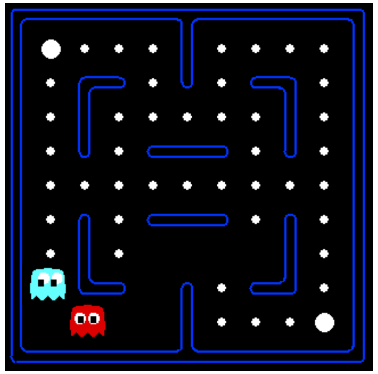
$f_{DOT}(s, \text{NORTH}) = 0.5$

$f_{GST}(s, \text{NORTH}) = 1.0$

$a = \text{NORTH}$

$r = -500$

$s'$



$Q(s, \text{NORTH}) = +1$

$r + \gamma \max_{a'} Q(s', a') = -500 + 0$

$Q(s', \cdot) = 0$

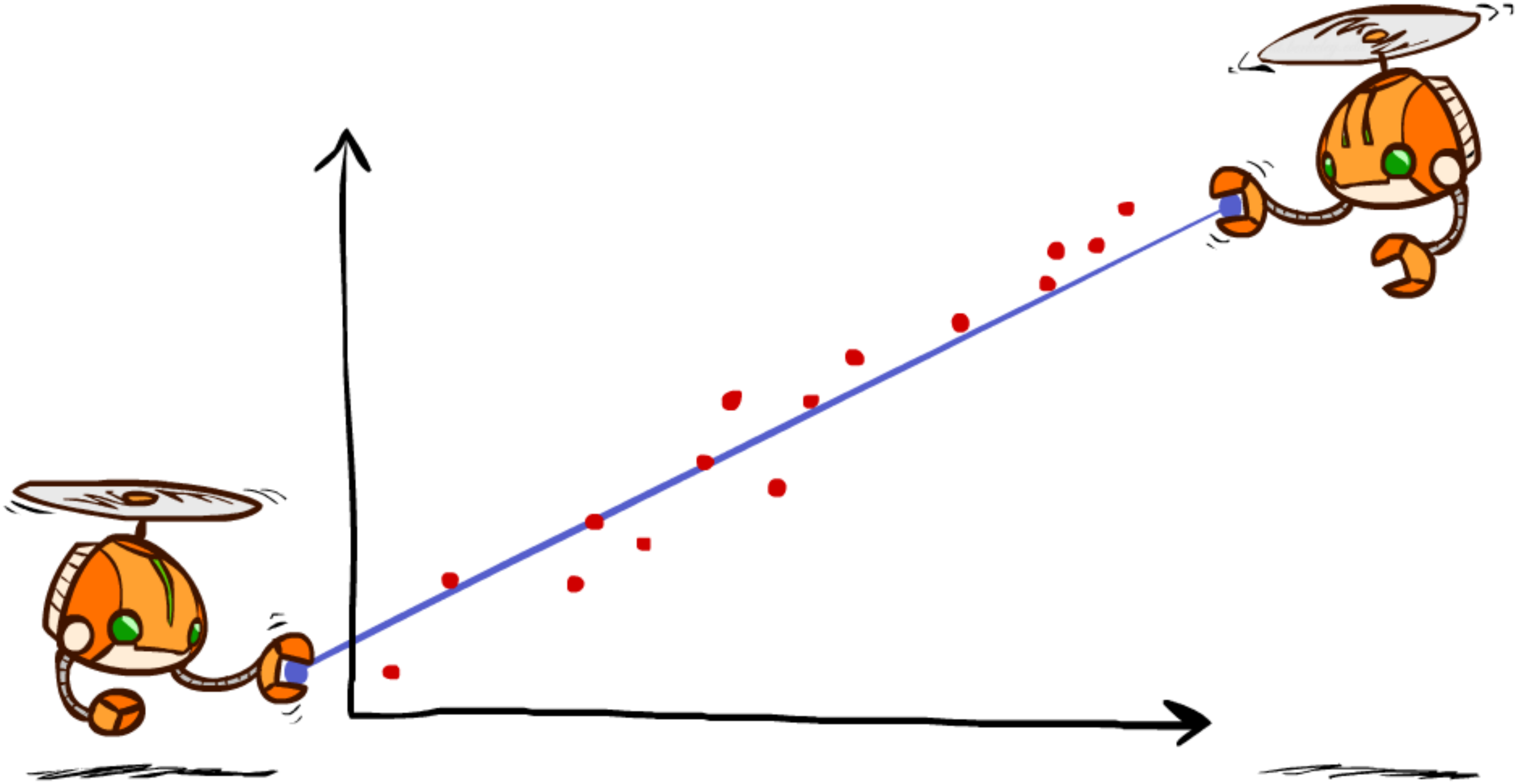
difference $= -501$

$w_{DOT} \leftarrow 4.0 + \alpha \left[-501\right] 0.5$

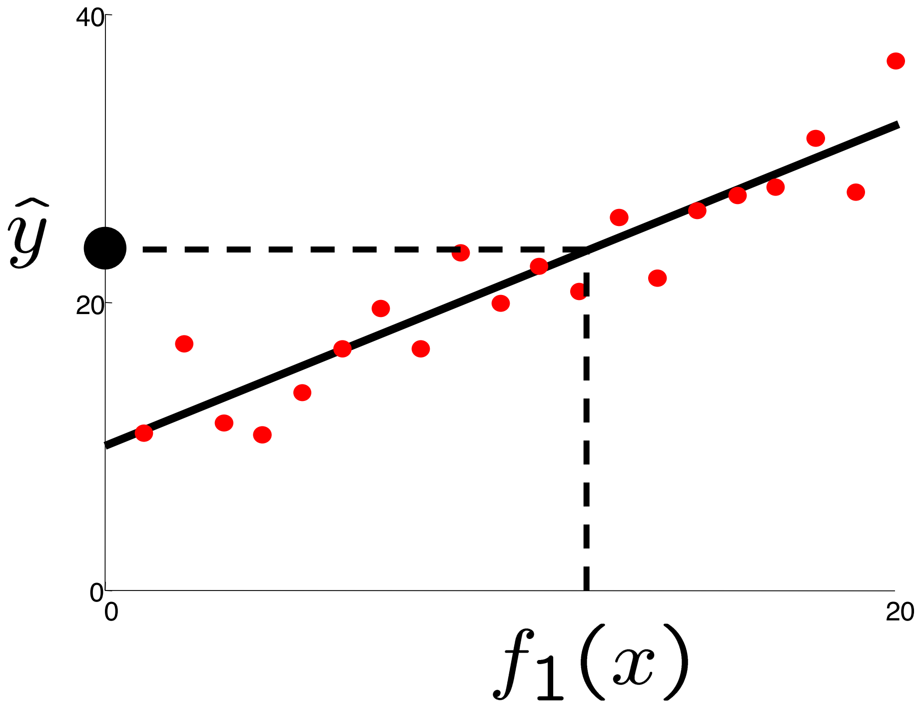
$w_{GST} \leftarrow -1.0 + \alpha \left[-501\right] 1.0$

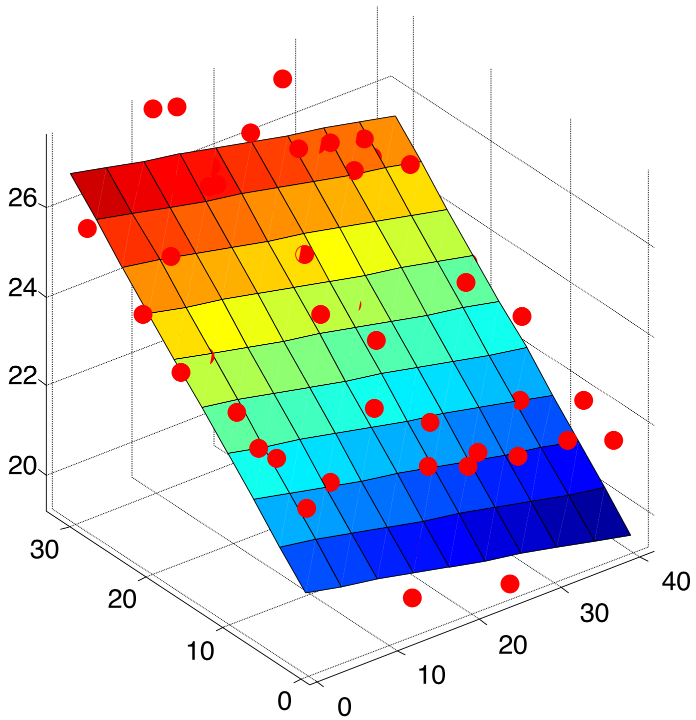
$$Q(s,a) = 3.0 f_{DOT}(s,a) - 3.0 f_{GST}(s,a)$$

# Formal justification: Q-Learning and least squares



# Linear approximation: regression




Prediction:

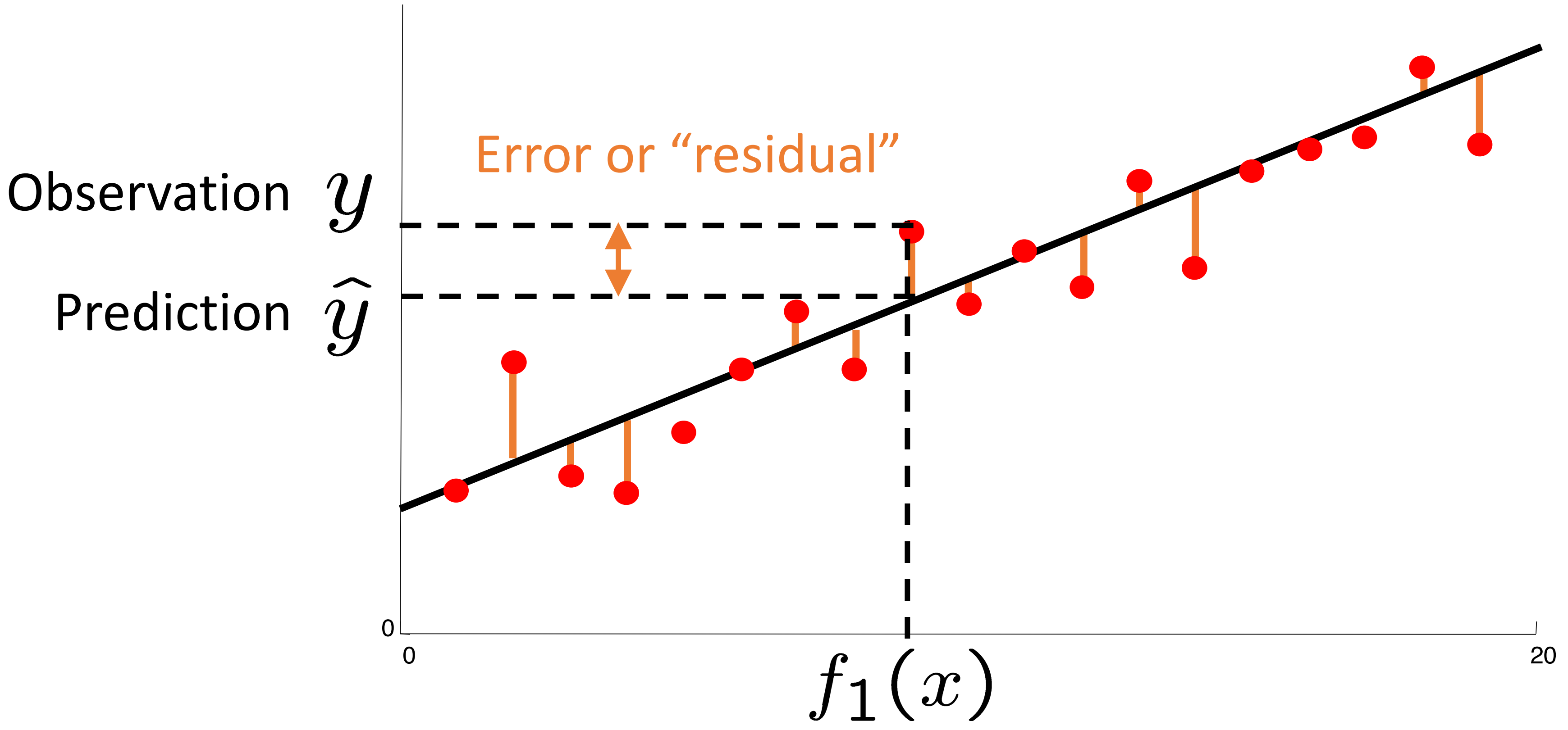
$$\hat{y} = w_0 + w_1 f_1(x)$$

Prediction:

$$\hat{y}_i = w_0 + w_1 f_1(x) + w_2 f_2(x)$$

# Optimization: least squares*

$$\text{total error} = \sum_i (y_i - \hat{y}_i)^2 = \sum_i \left( y_i - \sum_k w_k f_k(x_i) \right)^2$$



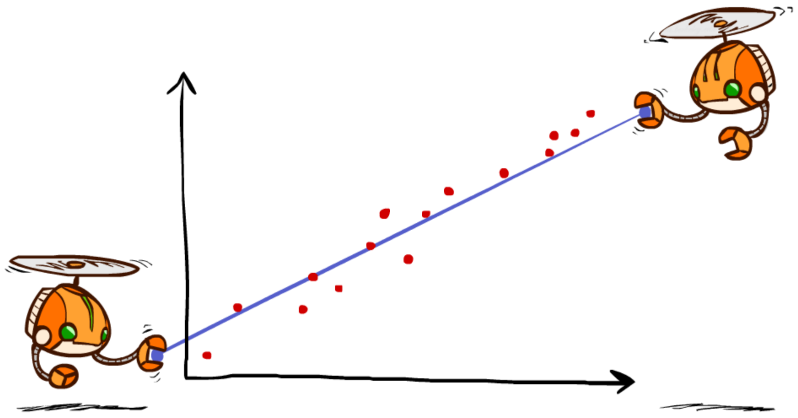
# Minimizing error

Imagine we had only one point x, with features f(x), target value y, and weights w:

$$\text{error}(w) = \frac{1}{2}\left(y - \sum_k w_k f_k(x)\right)^2$$

$$\frac{\partial\ \text{error}(w)}{\partial w_m} = -\left(y - \sum_k w_k f_k(x)\right) f_m(x)$$

$$w_m \leftarrow w_m + \alpha \left(y - \sum_k w_k f_k(x)\right) f_m(x)$$



Approximate q update explained:

$$w_m \leftarrow w_m + \alpha \left[r + \gamma \max_a Q(s', a') - Q(s, a)\right] f_m(s, a)$$

“target” “prediction”

# Let's think about Q-learning for SF

# Let's think about Q-learning for SF

Assume S = {*punch, block, move left, move right*}. So want to learn something like:

$$Q(s, punch) = w_1 \cdot f_1(s, punch) + ... + w_n \cdot f_n(s, punch)$$

- What are some features we might use here?

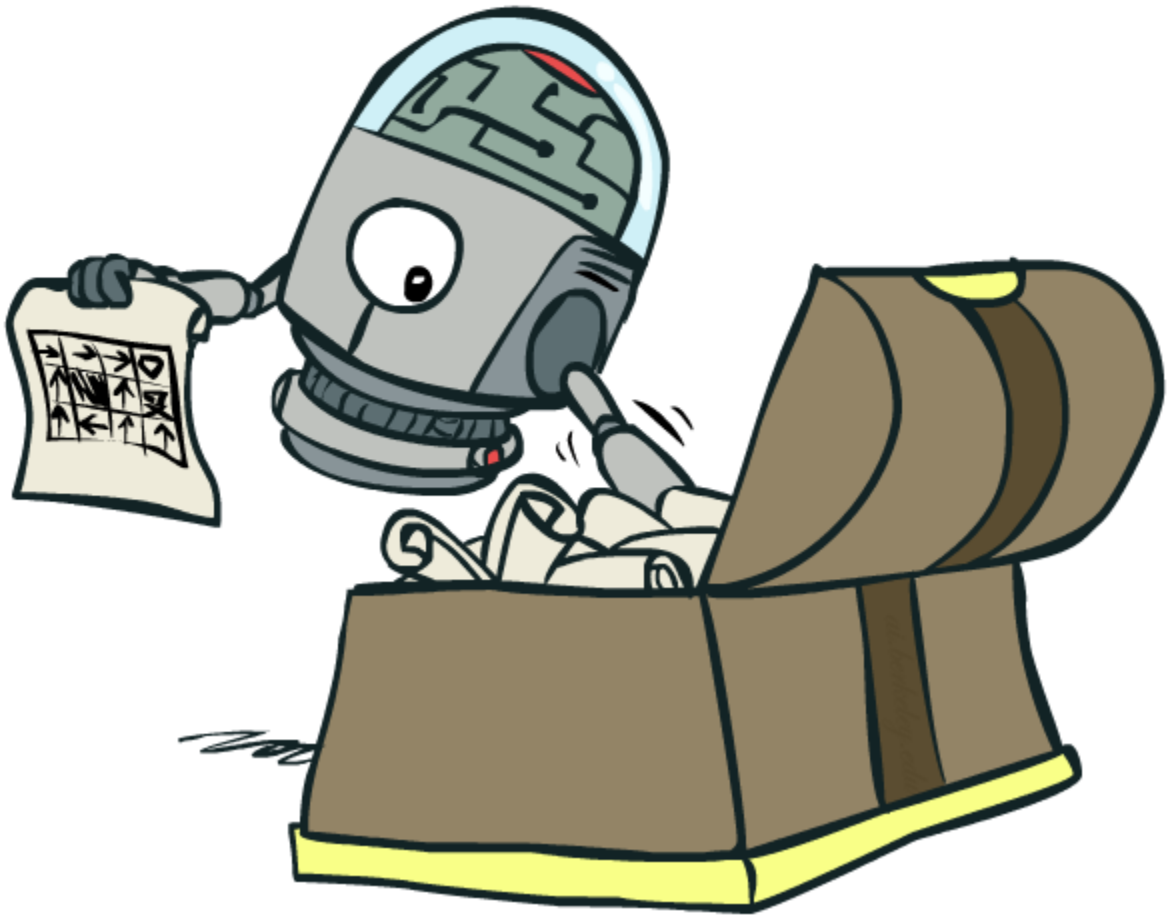
# Let's think about Q-learning for SF

Assume S = {*punch, block, move left, move right*}. So want to learn something like:

$$Q(s, punch) = w_1 \cdot f_1(s, punch) + ... + w_n \cdot f_n(s, punch)$$

- What are some features we might use here?
- What would we expect their values to look like (direction / order of magnitude)?

# Policy search



# Policy search

Note: often the feature-based policies that work well (win games, maximize utilities) *aren't the ones that approximate V / Q best*

- Q-learning's priority: get Q-values close (modeling)
- Action selection priority: get ordering of Q-values right (prediction)
- We'll see this distinction between modeling and prediction again later in the course

Solution: learn policies that maximize rewards, not the values that predict them

Policy search: start with an ok solution (e.g. Q-learning) then fine-tune by hill climbing on feature weights

# Policy search

Simplest policy search:

- Start with an initial linear value function or Q-function
- Nudge each feature weight up and down and see if your policy is better than before

Problems:

- How do we tell the policy got better?
- Need to run many sample episodes!
- If there are a lot of features, this can be impractical

# Policy search: stochastic policy

$$\pi_\theta(s, a) = e^{\hat{Q}_\theta(s,a)} / \sum_{a'} e^{\hat{Q}_\theta(s,a')}$$

This is a "softmax" function; we'll see it again!

# Policy search: REINFORCE

$$\nabla_\theta \rho(\theta) = \sum_a \pi_\theta(s_0, a) \cdot \frac{(\nabla_\theta \pi_\theta(s_0, a))R(a)}{\pi_\theta(s_0, a)} \approx \frac{1}{N} \sum_{j=1}^{N} \frac{(\nabla_\theta \pi_\theta(s_0, a_j))R(a_j)}{\pi_\theta(s_0, a_j)}$$

This is an unbiased estimate of the policy gradient

https://www.youtube.com/watch?v=0JL04JJjocc

# Conclusion

We're done with Part I: Search and Planning!

We've seen how AI methods can solve problems in:

- *Search*
- *Constraint Satisfaction Problems*
- *Games*
- *Markov Decision Problems*
- *Reinforcement Learning*

Next up: Part II: Uncertainty and Learning!