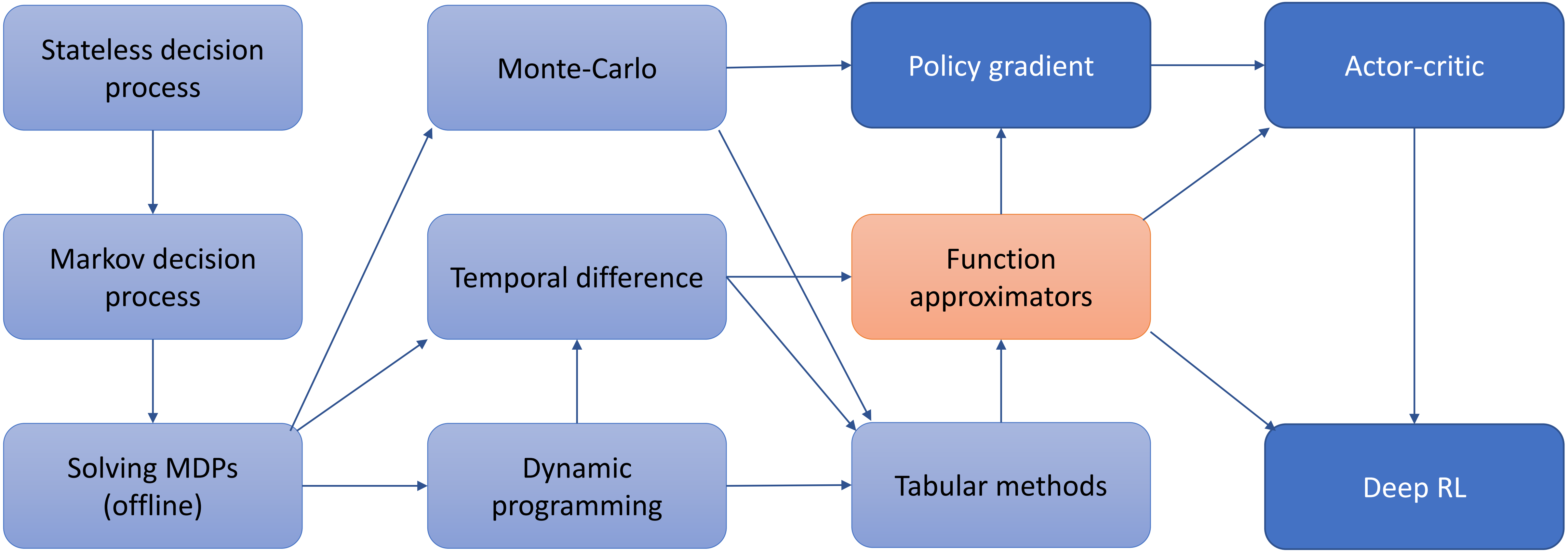
CSCE-642 Reinforcement Learning

Chapter 9: On-policy Prediction with Approximation

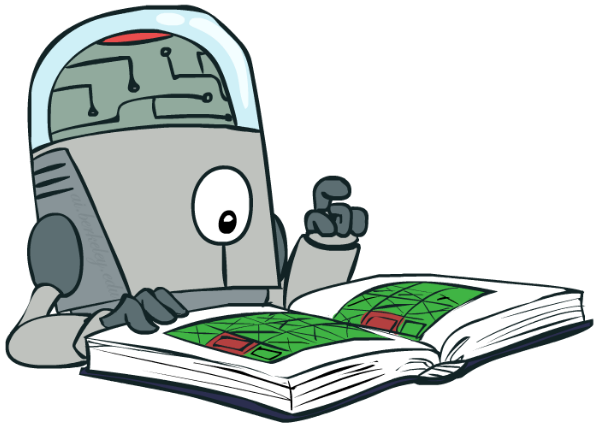
Instructor: Guni Sharon

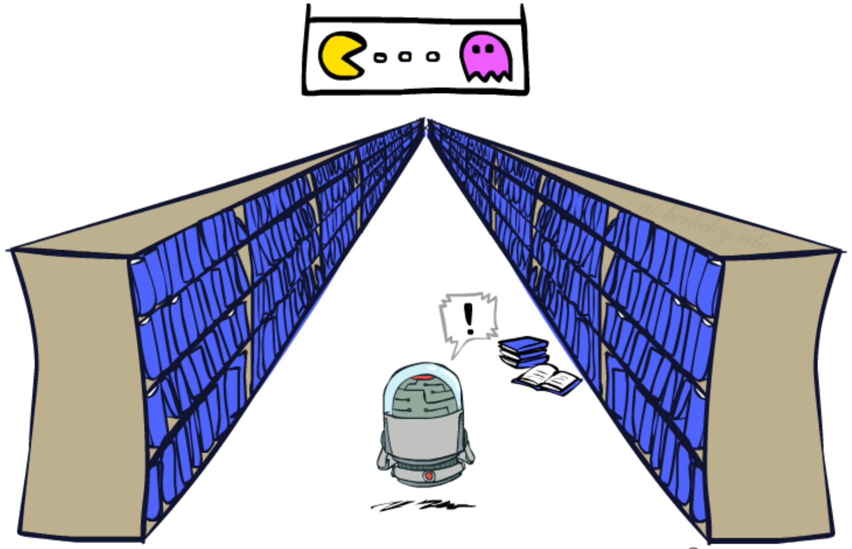
CSCE-689, Reinforcement Learning



Generalizing value-based learning

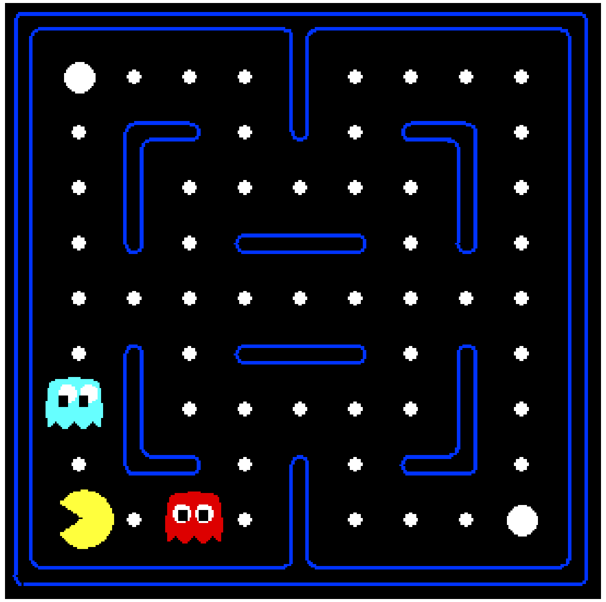
- Tabular Learning keeps a table of all state values
- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all during training
  - Too many states to hold a value table in memory
- Instead, we want to generalize:
  - Train on a small number of states from experience
  - Generalize that experience to new, similar situations
  - This is a fundamental idea in machine learning



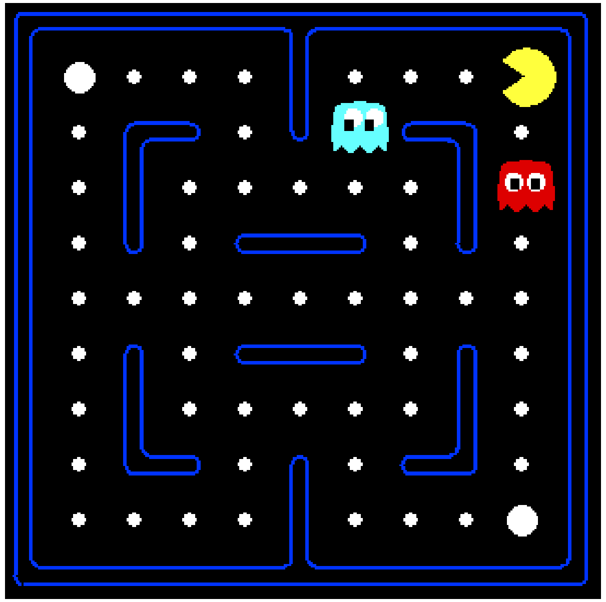


Example: Pacman

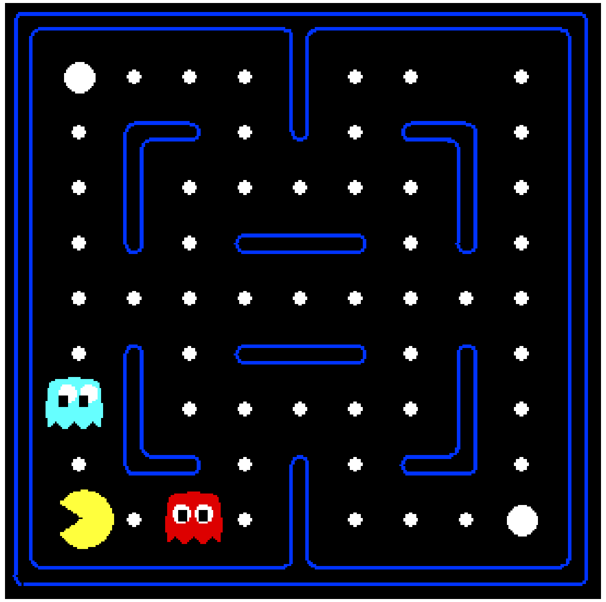
Let's say we discover through experience that this state is bad:



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

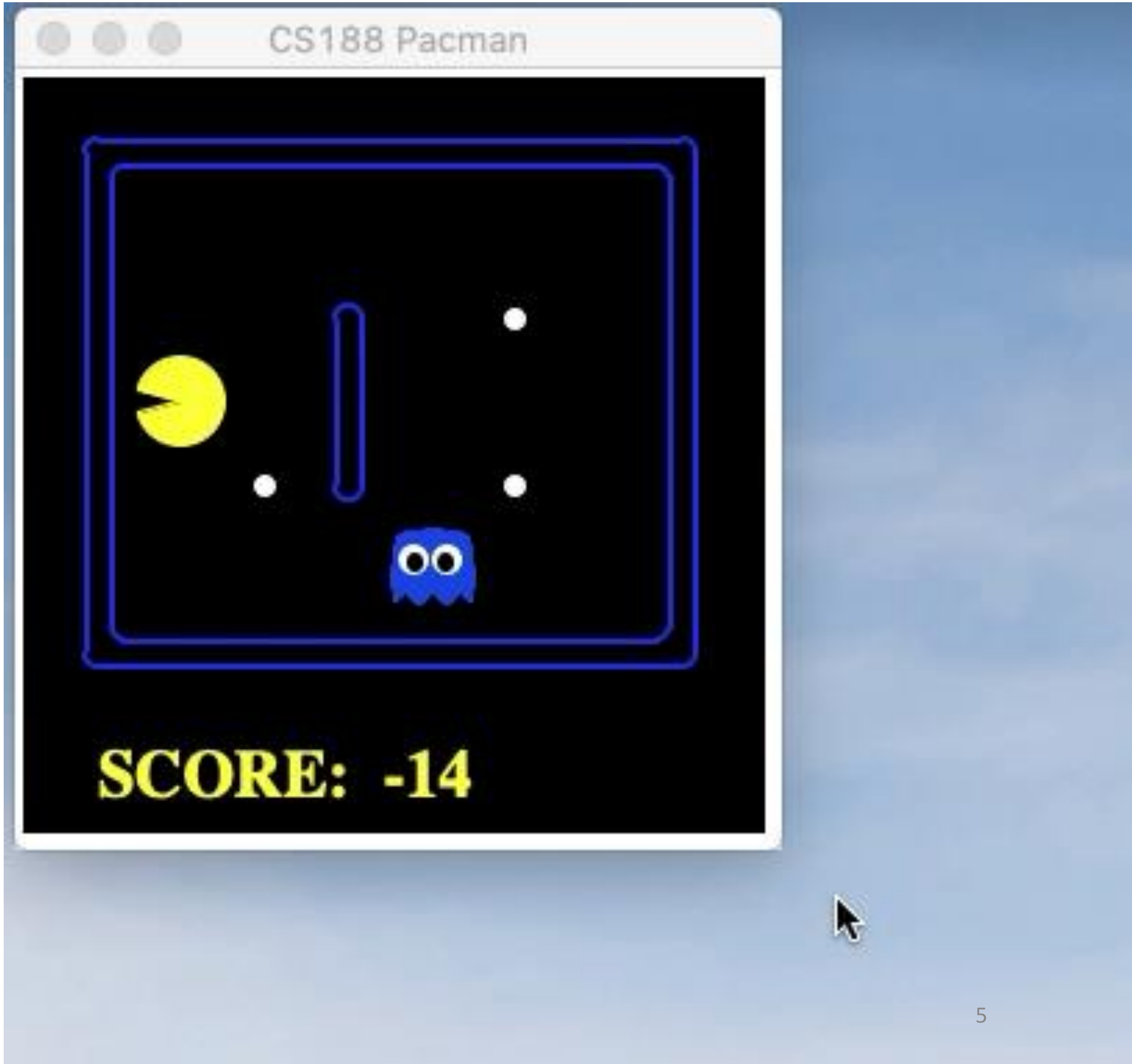


Or even this one!

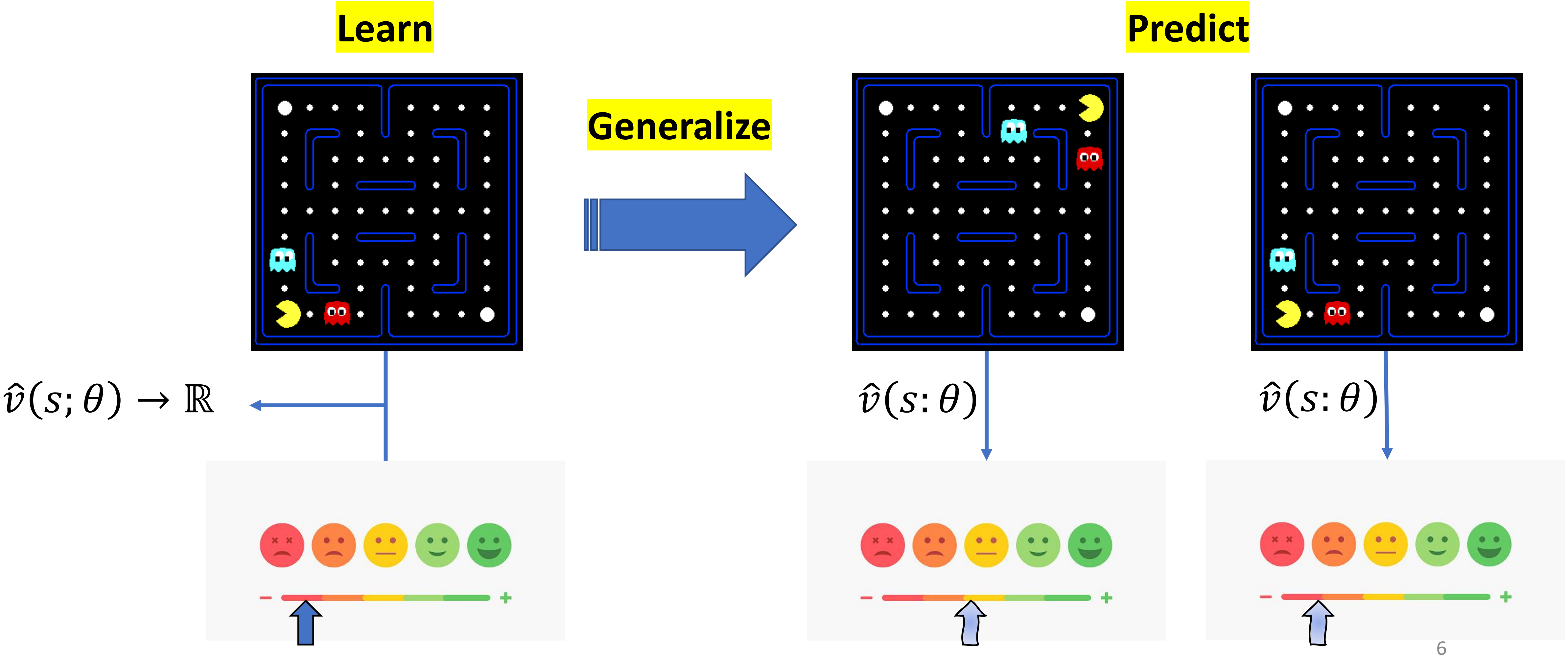


Demo

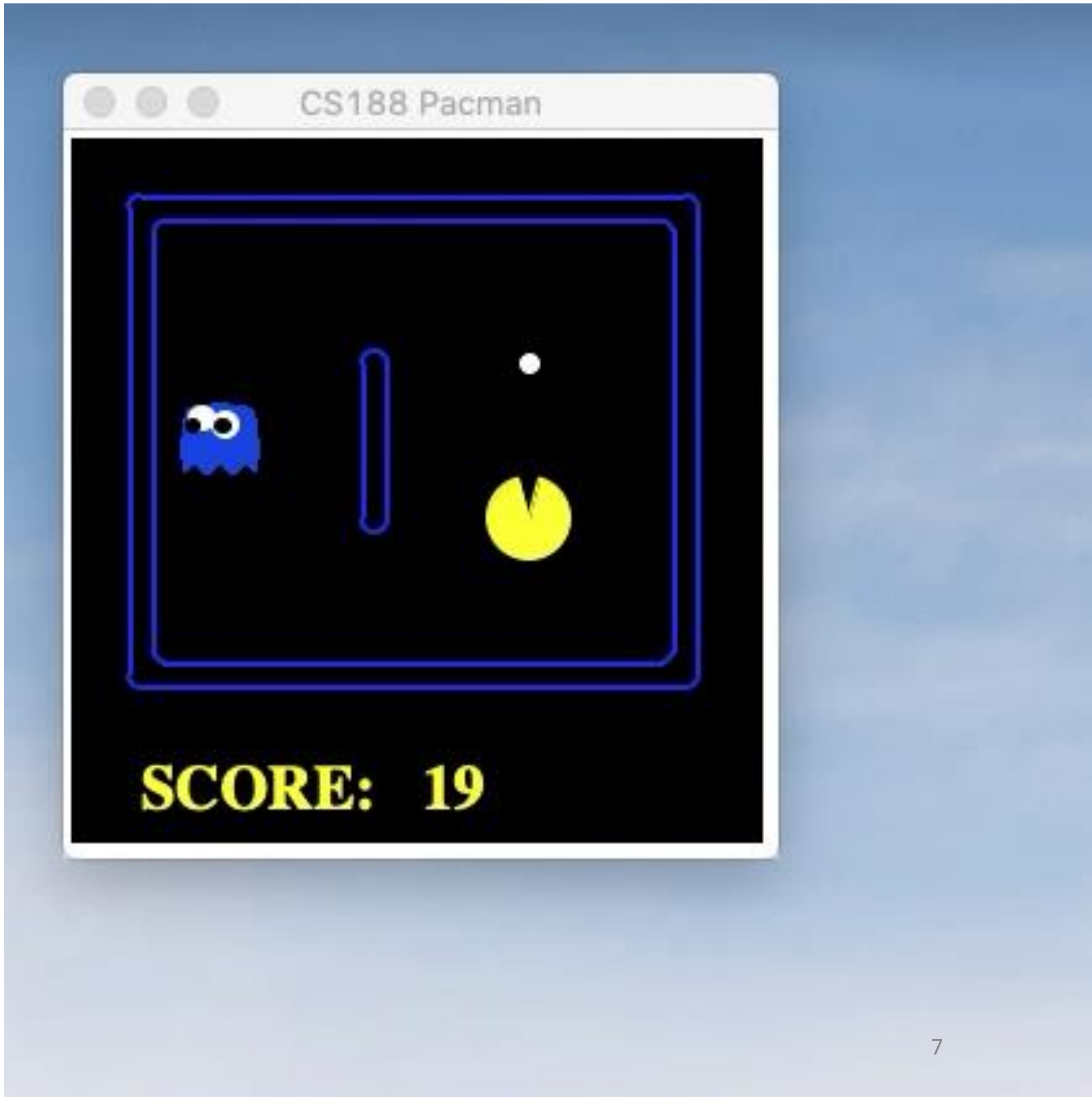
- Naïve Q-learning
- After 50 training episodes



Learn an approximation function

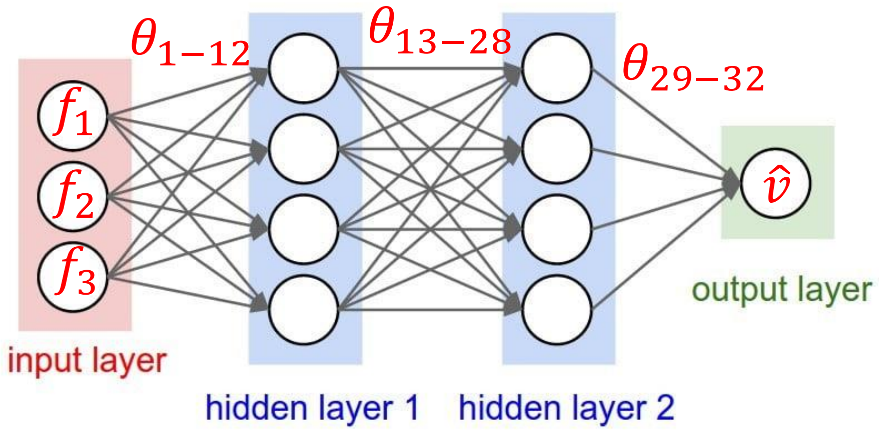


- Q-learning with function approximator



Parametrized function approximator

- Assume that each state is vector of features $(f_1, f_2, \dots, f_n)$, e.g.,
  - Packman_location, Ghost1_location , Ghost2_location, food_location
  - Or even screen pixels
- A parametrized value approximator $\hat{v}(s;\theta)$ might look like this:
  - $= \sum_i \theta_i f_i$ or this: $= \sum_i \theta_i \sin(i f_i)$ or even this:

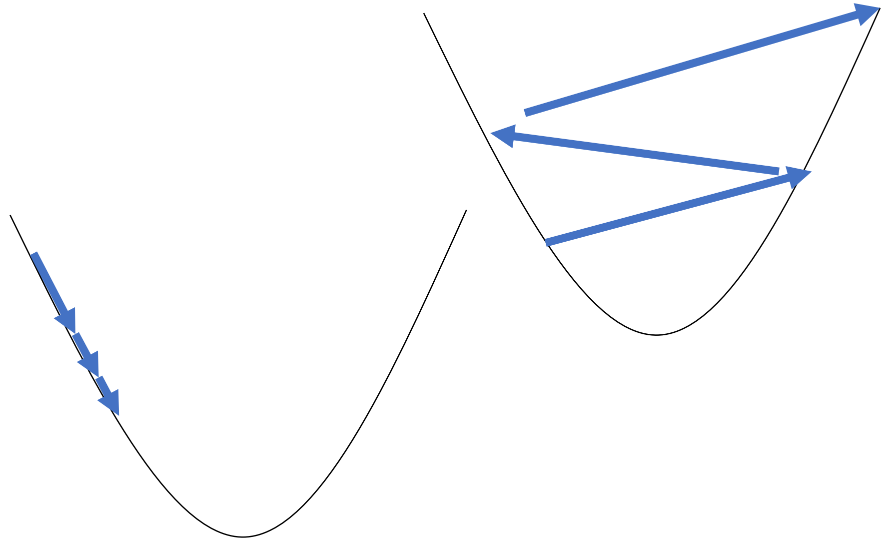


- Assume we know the true value for a set of states:
  - $v(S_1) = 5, v(S_2) = 8, v(S_3) = 2$
  - How can we update θ to reflect this information?

Gradient Decent

- Given: $v(S_1) = 5, v(S_2) = 8, v(S_3) = 2$
- We want to set θ such that $\forall s, \hat{v}(s;\theta) = v(s)$
    - Not possible in the general case, why?
    - Instead we'll try to minimize the errors: $\mathrm{loss} = \sum_s |v(s) - \hat{v}(s;\theta)|$
    - Partial derivative of the loss with respect to θ_i = how to change θ_i such that loss will increase the most
    - Go the other way -> decrease loss
    - Ooops! Absolute value is not differentiable -> can't compute gradients
    - Simple fix: $\mathrm{loss} = \frac{1}{2}\sum_s [v(s) - \hat{v}(s;\theta)]^2$ = squared loss function

Gradient Decent



- $\text{loss} = \frac{1}{2}\sum_s [v(s) - \hat{v}(s;\theta)]^2$
- For each i
  - Push θ_i towards a direction that minimizes loss
  - $\theta_i = \theta_i - \frac{\partial loss}{\partial \theta_i}$
- More generally $\theta = \theta - \alpha \nabla loss$
  - $\nabla loss = \left(\frac{\partial loss}{\partial \theta_1}, \frac{\partial loss}{\partial \theta_2}, \dots, \frac{\partial loss}{\partial \theta_n}\right)$
  - α is the learning rate, requires tuning per domain, too large causes divergence to small results in slow learning or even premature convergence

Gradient Decent

Chain rule

- $\text{loss} = \frac{1}{2}\sum_s [v(s) - \hat{v}(s;\theta)]^2$
- Gradient Decent: $\theta = \theta - \alpha \nabla_\theta loss = \theta - \alpha \nabla_{\hat{v}} loss \cdot \nabla_\theta \hat{v}$
  - $\theta = \theta + \alpha \sum_s \underbrace{\underbrace{[v(s) - \hat{v}(s;\theta)]}_{\text{The error in } \hat{v}(s;\theta)} \underbrace{\nabla \hat{v}(s;\theta)}_{\text{Change to } \theta \text{ such that } \hat{v}(s) \text{ is increased}}}_{\text{Change to } \theta \text{ such that the error in } \hat{v}(s;\theta) \text{ is decreased}}$

Gradient Descent

- Idea:
  - Start somewhere
  - Repeat: Take a step in the gradient direction

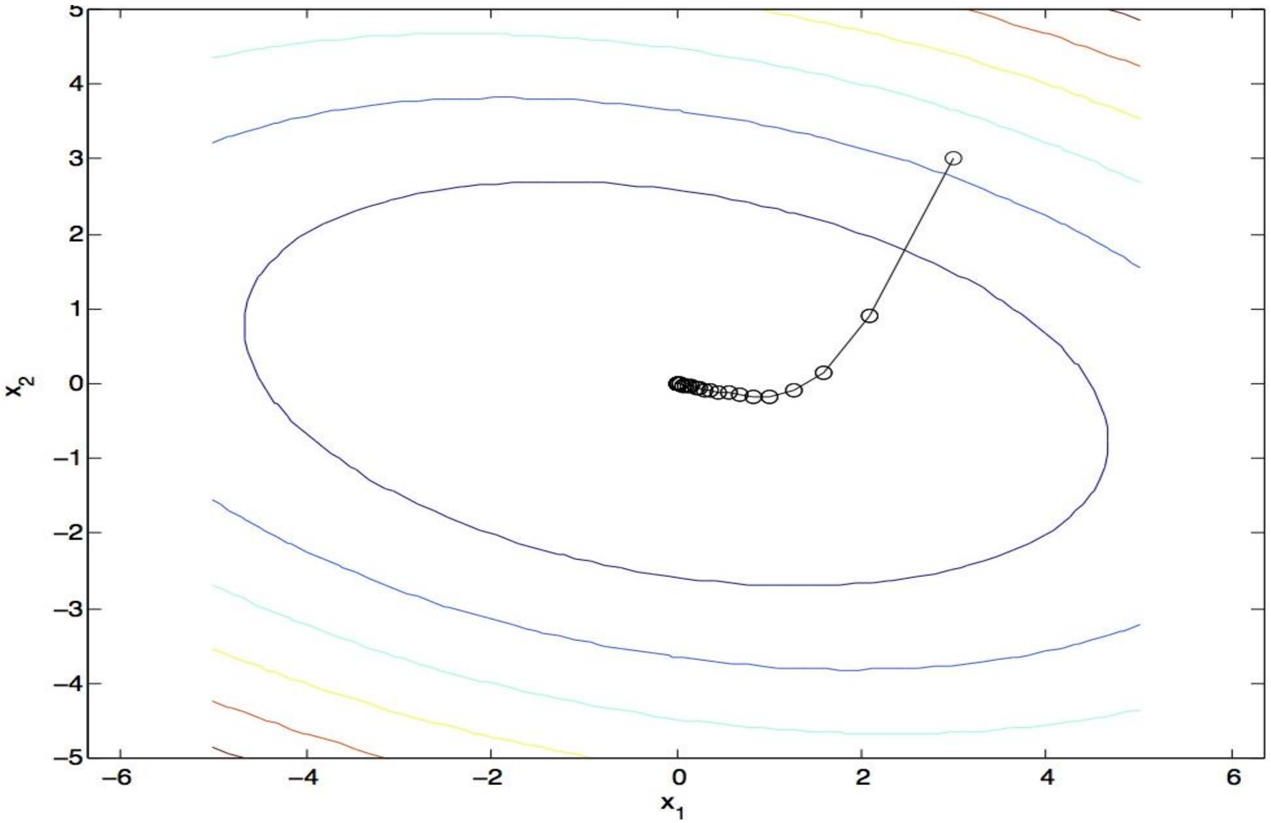


Figure source: Mathworks

Batch Gradient Decent

Minimize squared loss: $l(\theta) = \frac{1}{2}\sum_s [v(s) - \hat{v}(s;\theta)]^2$

- `init`
- `for iter = 1, 2, …`
  - $\theta = \theta + \alpha \sum_s [v(s) - \hat{v}(s;\theta)] \nabla \hat{v}(s;\theta)$

Stochastic Gradient Decent (SGD)

Minimize squared loss: $l(\theta) = \frac{1}{2}\sum_s [v(s) - \hat{v}(s;\theta)]^2$

Observation: once gradient on one training example has been computed, might as well incorporate before computing next one

- `init` θ
- `for iter = 1, 2, …`
  - `pick random j`
  - $\theta = \theta + \alpha\left[v(s_j) - \hat{v}(s_j;\theta)\right]\nabla\hat{v}(s_j;\theta)$

Mini-Batch Gradient Decent

Minimize squared loss: $l(\theta) = \frac{1}{2}\sum_s [v(s) - \hat{v}(s;\theta)]^2$

Observation: gradient over small set of training examples (=mini-batch) can be computed in parallel, might as well do that instead of a single one

- `init` θ
- `for iter = 1, 2, …`
  - `pick random subset of training examples J`
  - $\theta = \theta + \alpha \sum_{s \in j} [v(s) - \hat{v}(s;\theta)] \nabla \hat{v}(s;\theta)$

SGD for Monte Carlo estimation

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \rightarrow \mathbb{R}$

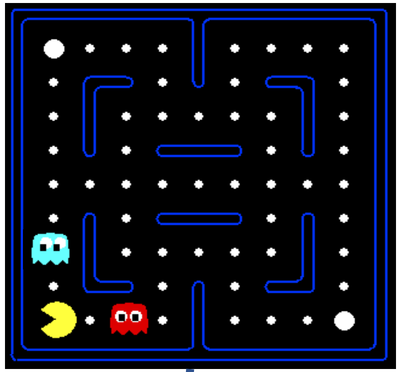
Initialize value-function weights $\mathbf{w}$ as appropriate (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat forever:
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  For $t = 0, 1, \ldots, T-1$:
    $\boxed{\mathbf{w} \leftarrow \mathbf{w} + \alpha \big[G_t - \hat{v}(S_t,\mathbf{w})\big] \nabla \hat{v}(S_t,\mathbf{w})}$

w are the tunable parameters of the value approximation function

- Guaranteed to converge to a local optimum because G_t is an unbiased estimate of $v_\pi(S_t)$

Example

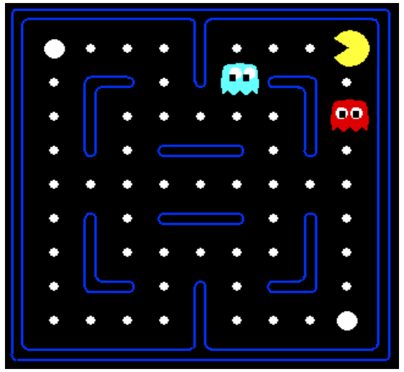
$f(S) = [2,2,1]$



10

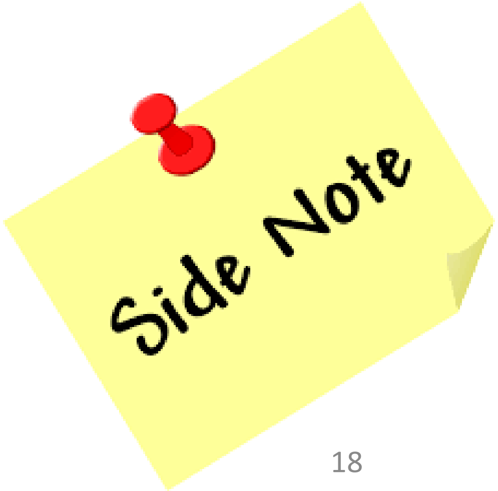
- $S = \{f_1(S), f_2(S), f_3(S)\}$
  - $f_{1,2}$=distance to ghost 1,2, f_3=distance to food
- $\hat{v}(s) = \sum_i \theta_i f_i(s)$
  - init: $\theta = [0,0,0]$
- $\theta = \theta + \alpha\big(G_t - \hat{v}(s;\theta)\big)\nabla\hat{v}(s;\theta)$
- $\theta = [0,0,0] + 0.1(10 - [0,0,0] \cdot [2,2,1])[2,2,1]$
  - $\theta = [2,2,1]$
- $\hat{v}(S') = f(S') \cdot \theta = [2,4,1] \cdot [2,2,1] = 13$

$f(S') = [2,4,1]$



Side note

- Should we care about on-policy value approximation?
  - Once the policy changes the approximated values become irrelevant
- Yes! This will be useful for Actor-Critic methods which will be discussed later



Learning approximation with bootstrapping

- Can we update the value approximation function at every step?
- Yes, define SGD as a function of the TD error
  - Tabular TD learning: $\hat{v}(s_t) = \hat{v}(s_t) + \alpha\big(r_t + \gamma\hat{v}(s_{t+1}) - \hat{v}(s_t)\big)$
  - Approximation TD learning: $\theta = \theta + \alpha\big(r_t + \gamma\hat{v}(s_{t+1};\theta) - \hat{v}(s_t;\theta)\big)\nabla\hat{v}(s_t;\theta)$
- Known as Semi-gradient methods
- **NOT** guaranteed to converge to a local optimum because $\hat{v}(s_{t+1};\theta)$ is a biased estimate of $v_\pi(s_{t+1})$
- Semi-gradient (bootstrapping) methods do not converge as robustly as (full) gradient methods

Semi-gradient methods

- They do converge reliably in important cases such as the linear approximation case
- They offer important advantages that make them often clearly preferred
- They typically enable significantly faster learning, as we have seen in Chapters 6 and 7
- They enable learning to be continual and online, without waiting for the end of an episode
- This enables them to be used on continuing problems and provides computational advantages

Semi-gradient TD(0)

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$

Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        Choose $A \sim \pi(\cdot|S)$
        Take action A, observe R, S'
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
        $S \leftarrow S'$
    until S' is terminal

What's the difference from the tabular case?

Semi-gradient TD(0)

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$

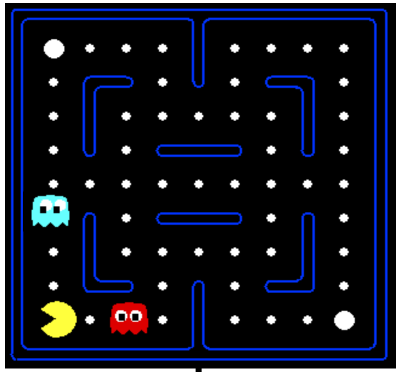
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        Choose $A \sim \pi(\cdot|S)$
        Take action A, observe R, S'
        $\boxed{\mathbf{w} \leftarrow \mathbf{w} + \alpha \big[R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big] \nabla \hat{v}(S,\mathbf{w})}$
        $S \leftarrow S'$
    until S' is terminal

What's the difference from the tabular case?

$\hat{v}(S) = \hat{v}(S) + \alpha[R + \gamma \hat{v}(S') - \hat{v}(S)]$

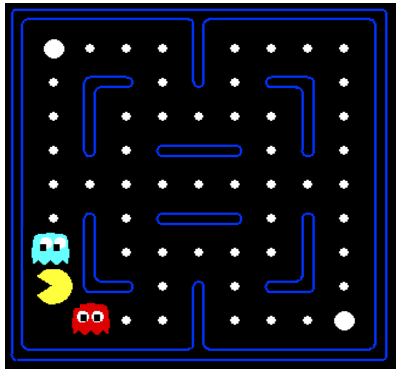
Example

$f(S) = [2,3,1]$



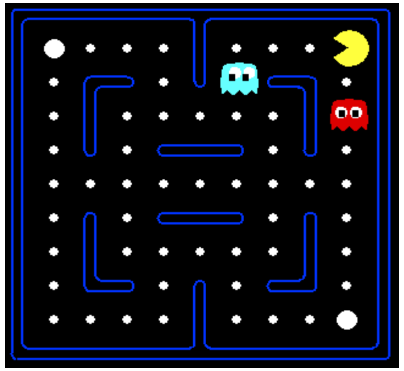
$R = +10$

$f(S') = [1,2,1]$



- $S = \{f_1(S), f_2(S), f_3(S)\}$
  - $f_{1,2}$=distance to ghost 1,2, f_3=distance to food
- $\hat{v}(s) = \sum_i \theta_i f_i(s)$
  - init: $\theta = [0,0,0]$
- $\theta = \theta + \alpha\big(R + \gamma\hat{v}(S';\theta) - \hat{v}(S;\theta)\big)\nabla\hat{v}(S;\theta)$
- $\theta = [0,0,0] + 0.1(10 + [1,2,1]\cdot[0,0,0] - [2,3,1]\cdot[0,0,0])[2,3,1]$
  - $\theta = [2,3,1]$
- $\hat{v}(U) = f(U)\cdot\theta = [2,4,1]\cdot[2,3,1] = 17$

$f(U) = [2,4,1]$



n-step return

n-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Parameters: step size $\alpha \in (0, 1]$, a positive integer n
All store and access operations (S_t and R_t) can take their index mod n

Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  For $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n},\mathbf{w})$ $(G_{\tau:\tau+n})$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[G - \hat{v}(S_\tau,\mathbf{w})\right] \nabla \hat{v}(S_\tau,\mathbf{w})$
  Until $\tau = T - 1$

- Again, only a simple modification over the tabular setting

n-step return

n-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

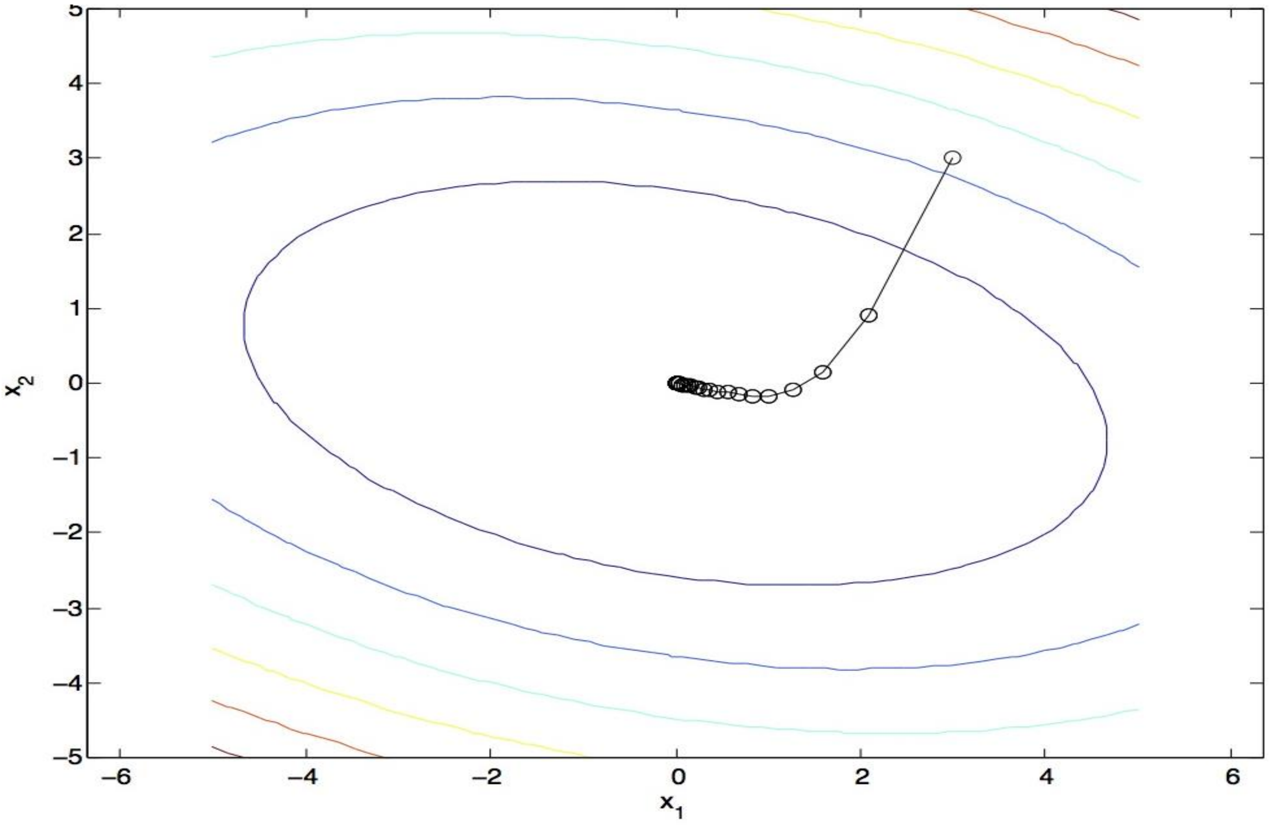
Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Parameters: step size $\alpha \in (0, 1]$, a positive integer n
All store and access operations (S_t and R_t) can take their index mod n

Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  For $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n},\mathbf{w})$ $(G_{\tau:\tau+n})$
  | $\boxed{\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[G - \hat{v}(S_\tau,\mathbf{w})\right] \nabla \hat{v}(S_\tau,\mathbf{w})}$
  Until $\tau = T - 1$

- Again, only a simple modification over the tabular setting
- Weight update instead of tabular entry update

Another optimization approach

- Approach1: Gradient decent
  - Update θ in iterations
  - Find a fixed point
  - If:$\theta_{t+1} = \theta_t$
    - Then: converged to a local optimum
- Approach2: Compute the fixed point directly
  - Solve: $\theta_{t+1} = \theta_t$



Compute fixed point over θ

- Assume a linear function approximator $\boxed{\hat{v}(f(s);\theta) = f(s)\cdot\theta}$
- TD update: $\theta_{t+1} = \theta_t + \alpha[R_{t+1} + \gamma\hat{v}(S_{t+1}) - \hat{v}(S_t)]\underbrace{\nabla\hat{v}(S_t)}_{f(S_t)}$

For a linear approximator

- $= \theta_t + \alpha\left[R_{t+1} + \big(\gamma f(S_{t+1}) - f(S_t)\big)\theta_t\right] f(S_t)$
- $= \theta_t + \alpha(\mathrm{b} - \mathrm{A}\theta_t)$

Vector Matrix

- Where: $\mathrm{b} = \mathbb{E}[R_{t+1} f(S_t)]$, $\mathrm{A} = \mathbb{E}\left[f(S_t)\big(f(S_t) - \gamma f(S_{t+1})\big)^{\top}\right]$
- Fixed point at: $\mathbb{E}[\theta_{t+1}] = \mathbb{E}[\theta_t]$
  - $\mathrm{b} - \mathrm{A}\theta_t = 0$
  - $\mathrm{b} = \mathrm{A}\theta_t$
  - $\theta_t = \mathrm{A}^{-1}\mathrm{b}$

TD-error = 0

Least squares TD

- Approximate A and b online, solve $\theta = \mathrm{A}^{-1}\mathrm{b}$
- Where: $\mathrm{b} = \mathbb{E}[R_{t+1} f(S_t)]$, $\mathrm{A} = \mathbb{E}\left[f(S_t)\big(f(S_t) - \gamma f(S_{t+1})\big)^\top\right]$
- $\widehat{\mathrm{A}} = \sum_{k=0}^{t-1} f(S_t)\big(f(S_t) - \gamma f(S_{t+1})\big)^\top$
- But $\widehat{\mathrm{A}}$ is not guaranteed to be invertible (it might have '0' on diagonal)
- So add a small constant (ε) to the diagonal
  - $\widehat{\mathrm{A}} = \sum_t f(S_t)\big(f(S_t) - \gamma f(S_{t+1})\big)^\top + \varepsilon \mathrm{I}$
- $\hat{\mathrm{b}} = \sum_t R_{t+1} f(S_t)$

Least squares TD

LSTD for estimating $\hat{v} \approx v_\pi$ ($O(d^2)$ version)

Input: feature representation $\mathbf{x}(s) \in \mathbb{R}^d$, for all $s \in \mathcal{S}, \mathbf{x}(\text{terminal}) \doteq \mathbf{0}$

$\widehat{\mathbf{A}^{-1}} \leftarrow \varepsilon^{-1}\mathbf{I}$

Store the inverse of A instead of A

An $d \times d$ matrix

$\widehat{\mathbf{b}} \leftarrow \mathbf{0}$

An d-dimensional vector

Repeat (for each episode):
  Initialize S; obtain corresponding $\mathbf{x}$
  Repeat (for each step of episode):
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'; obtain corresponding $\mathbf{x}'$
    $\mathbf{v} \leftarrow \widehat{\mathbf{A}^{-1}}^\top (\mathbf{x} - \gamma\mathbf{x}')$
    $\widehat{\mathbf{A}^{-1}} \leftarrow \widehat{\mathbf{A}^{-1}} - (\widehat{\mathbf{A}^{-1}}\mathbf{x})\mathbf{v}^\top / (1 + \mathbf{v}^\top \mathbf{x})$
    $\widehat{\mathbf{b}} \leftarrow \widehat{\mathbf{b}} + R\mathbf{x}$
    $\boldsymbol{\theta} \leftarrow \widehat{\mathbf{A}^{-1}}\widehat{\mathbf{b}}$
    $S \leftarrow S'$; $\mathbf{x} \leftarrow \mathbf{x}'$
  until S' is terminal

Least squares TD

LSTD for estimating $\hat{v} \approx v_\pi$ ($O(d^2)$ version)

Input: feature representation $\mathbf{x}(s) \in \mathbb{R}^d$, for all $s \in \mathcal{S}$, $\mathbf{x}(\text{terminal}) \doteq \mathbf{0}$

$\widehat{\mathbf{A}^{-1}} \leftarrow \varepsilon^{-1}\mathbf{I}$ — An $d \times d$ matrix

$\widehat{\mathbf{b}} \leftarrow \mathbf{0}$ — An d-dimensional vector

Repeat (for each episode):

    Initialize S; obtain corresponding $\mathbf{x}$

    Repeat (for each step of episode):

        Choose $A \sim \pi(\cdot|S)$

        Take action A, observe R, S'; obtain corresponding $\mathbf{x}'$

        $\mathbf{v} \leftarrow \widehat{\mathbf{A}^{-1}}^{\top}(\mathbf{x} - \gamma\mathbf{x}')$

$$\left.\begin{array}{l}\widehat{\mathbf{A}^{-1}} \leftarrow \widehat{\mathbf{A}^{-1}} - (\widehat{\mathbf{A}^{-1}}\mathbf{x})\mathbf{v}^{\top}/(1 + \mathbf{v}^{\top}\mathbf{x}) \\ \widehat{\mathbf{b}} \leftarrow \widehat{\mathbf{b}} + R\mathbf{x}\end{array}\right\} \longleftarrow \text{Incremental updates (no need to store all previous transitions)}$$

        $\boldsymbol{\theta} \leftarrow \widehat{\mathbf{A}^{-1}}\widehat{\mathbf{b}}$

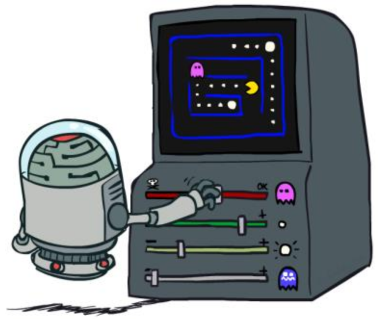
        $S \leftarrow S'$; $\mathbf{x} \leftarrow \mathbf{x}'$

    until S' is terminal

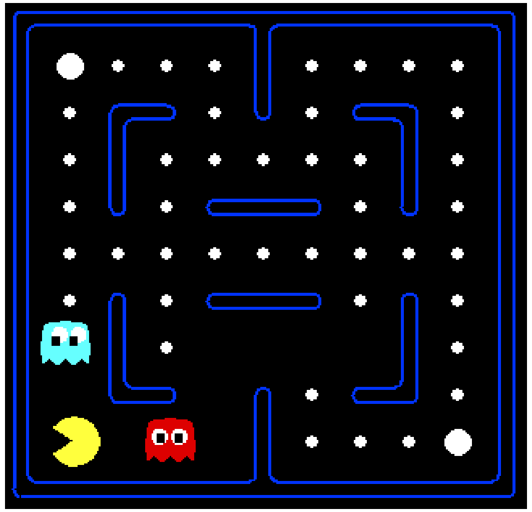
Least squares TD

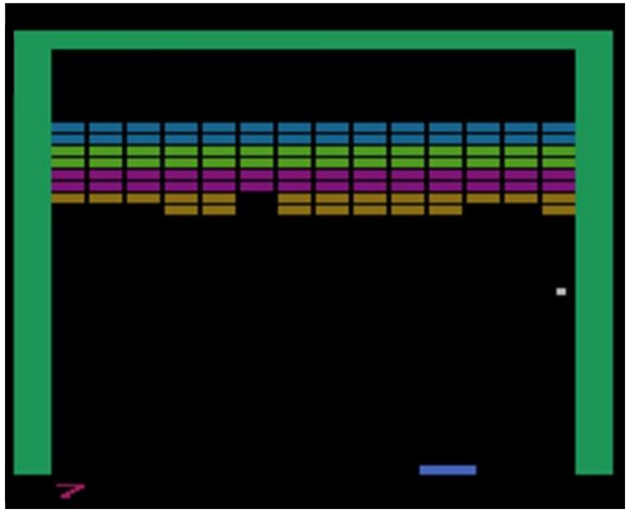
- Directly computing the TD fixed point
- Most data efficient form of linear TD(0), but it is also more expensive computationally
  - Semi-gradient linear TD(0) requires memory and per-step computation that is only $O(d)$ where d is the number of state features
- In the incremental update version, $\widehat{\mathrm{A}}$ is an outer product (a column vector times a row vector) and thus requires a matrix update
- The update computational complexity is $O(d^2)$, and the memory complexity is $O(d^2)$

Feature selection



- Assume a linear function approximator $\hat{v}(f(s);\theta) = f(s) \cdot \theta$
- What relevant features should represent states?

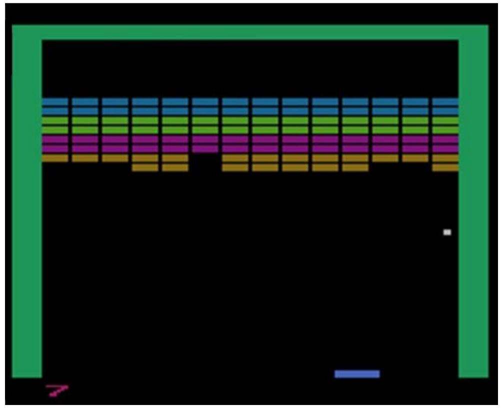




Features are domain depended
requiring expert knowledge

Automatic features extraction

- Consider a game state as a pixel matrix
- Raw data of type: $pixel(7{,}3) = [0{,}0{,}0]$ (black)
- Desired features = {ball location, ball speed, ball direction, pan location…}
- How can we translate pixels to the relevant features?

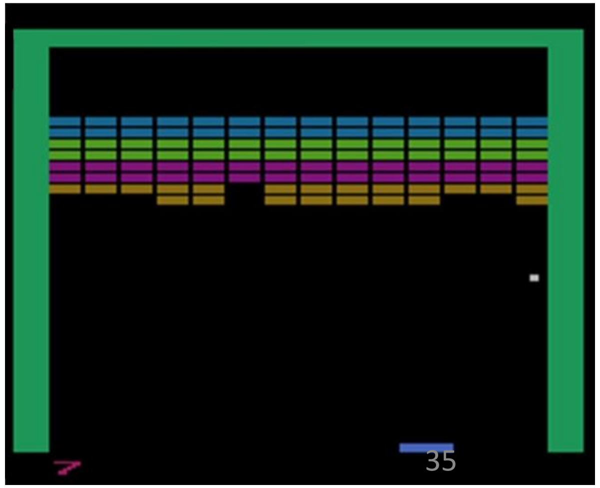


Automatic features extraction for linear approximator

- **Polynomials**: $f_i(s) = \prod_{j=1}^{k} x_j^{c_{i,j}}$
  - where each $c_{i,j}$ is an integer in the set $\{0,1,\dots,n\}$ for an integer $n \geq 0$
  - These features makeup the order-n polynomial basis for dimension k, which contains $(n+\ 1)^k$ different features
- **Fourier Basis**: $f_i(s) = \cos\left(\pi X^\top c^i\right)$
  - Where $c^i = \left(c_1^i,\dots,c_k^i\right)^\top$, with $c_j^i \in \{0,\dots,n\}$ for $j = \{1,\dots,k\}$ and $i = \{0,\dots,(n+1)^k\}$
  - This defines a feature for each of the $(n+1)^k$ possible integer vectors c^i
  - The inner product $X^\top c^i$ has the effect of assigning an integer in $\{0,\dots,n\}$ to each dimension of X
  - This integer determines the feature's frequency along that dimension
  - The features can be shifted and scaled to suit the bounded state space of a particular application

Automatic features extraction for linear approximator

- Other approaches include: Coarse Coding, Tile Coding, Radial Basis Functions (See chapter 9.5 in textbook)
- Each of these approaches defines a set of features, some useful yet most are not
  - E.g., is there a polynomial/Fourier function that translates pixels to pan location?
  - Probably but it's a needle in a (combinatorial) haystack
- Can we do better (generically)
  - Yes, using deep neural networks…



What did we learn?

- Reinforcement learning must generalize on observed experience if it is to be applicable to real world domains
- We can use parameterized function approximation to represent our knowledge about the domain state/action values
- Use stochastic gradient descend to update the tunable parameters such that the observed (TD, rollout) error is reduced
- When using a linear approximator, the Least squares TD method provides the most sample efficient approximation

What next?

- **Lecture**: Deep Neural Networks as function approximators
- **Assignments**:
  - Value Iteration, by September-23, EOD
  - Asynchronous Value Iteration, by September-23, EOD
  - Policy Iteration, by September-23, EOD
  - Monte-Carlo Control by September-30, EOD
  - Monte-Carlo Control with Importance Sampling by September-30, EOD
  - Tabular Q-Learning, by Oct. 7, EOD
  - SARSA, by Oct. 7, EOD
  - Q-Learning with Approximation, by Oct. 14, EOD
- **Quiz (on Canvas)**:
  - n-step Bootstrapping, by Sep. 23, EOD
  - Value approximation, by Oct. ?, EOD
- **Project**:
  - Project proposal, by Sep. 30 EOD