



**BITS Pilani**  
Pilani | Dubai | Goa | Hyderabad

# Deep Reinforcement Learning

2022-23 Second Semester, M.Tech (AIML)

## Session #10-11-12: On Policy Prediction with Approximation

### Instructors :

1. Prof. S. P. Vimal ([vimalsp@wilp.bits-pilani.ac.in](mailto:vimalsp@wilp.bits-pilani.ac.in)),
2. Prof. Sangeetha Viswanathan ([sangeetha.viswanathan@pilani.bits-pilani.ac.in](mailto:sangeetha.viswanathan@pilani.bits-pilani.ac.in))

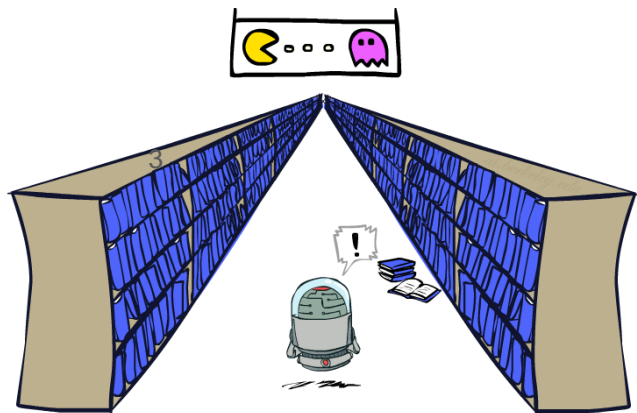
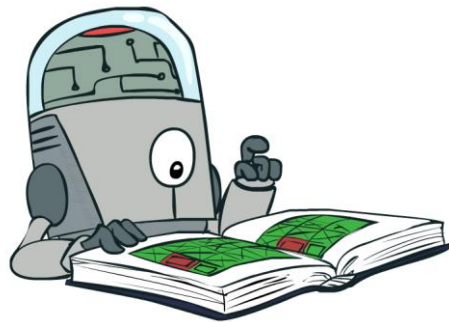


# Agenda for the classes

- Introduction
- Value Function Approximation
- Stochastic Gradient, Semi-Gradient Methods
- Role of Deep Learning for Function Approximation;
- Feature Construction Methods

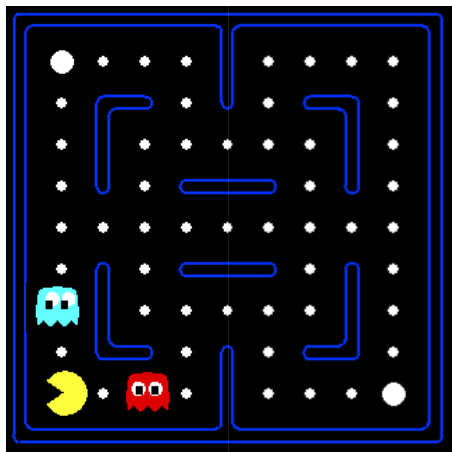
# Generalizing Across States

- 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:
  - 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

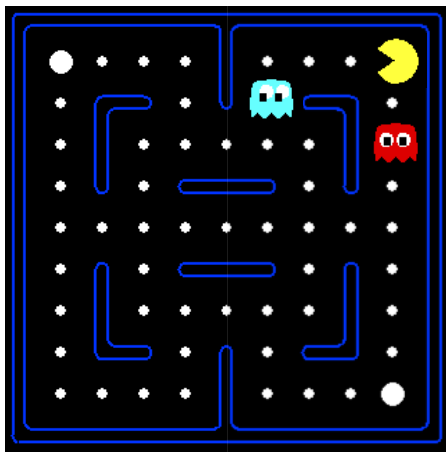


# Example: Pacman

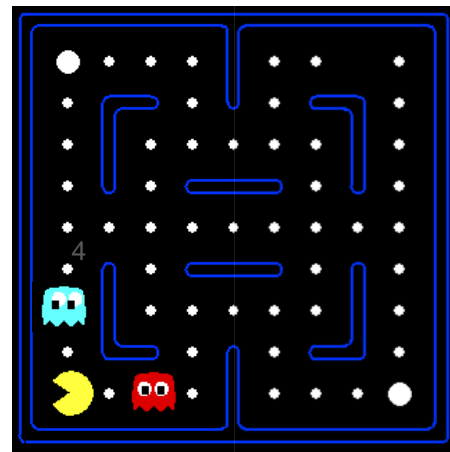
Let's say we discover through experience that this state is bad:



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



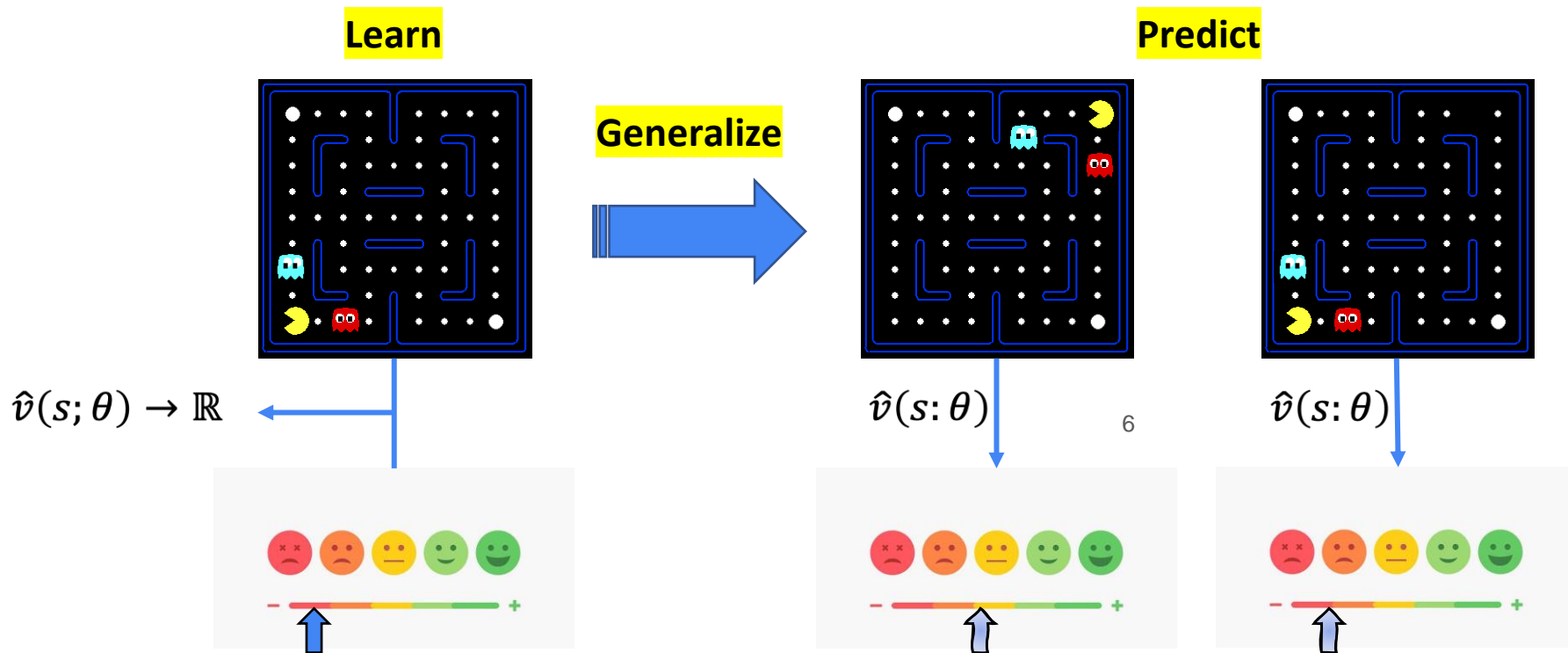
Or even this one!



- Naïve Q-learning
- After 50 training episodes



# Learn an approximation function

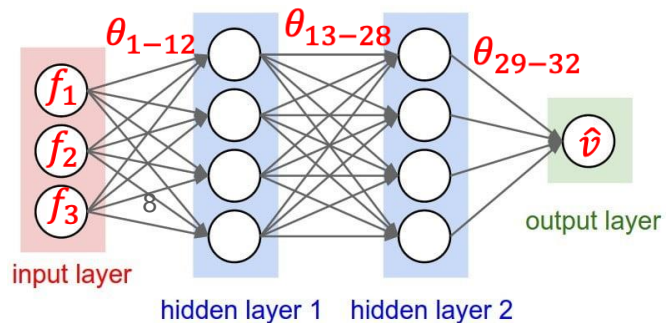


- Q-learning with function approximator



# Parameterized function approximator

- Assume that each state is vector of features  $(f_1, f_2, \dots, f_n)$ , e.g.,
  - Pacman 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 maybe like this  $= \prod_i f_i^{\theta_i}$  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  $\theta$  to reflect this information?





# Gradient Descent

- Given:  $v(S_1) = 5, v(S_2) = 8, v(S_3) = 2$
- We want to set  $\theta$  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:  $\text{loss} = \sum_s |v(s) - \hat{v}(s; \theta)|$
  - Partial derivative of the loss with respect to  $\theta_i$  = how to change  $\theta_i$  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:  $\text{loss} = \frac{1}{2} \sum_s [v(s) - \hat{v}(s; \theta)]^2$  = squared loss function

# Gradient Decent

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

- For each  $i$

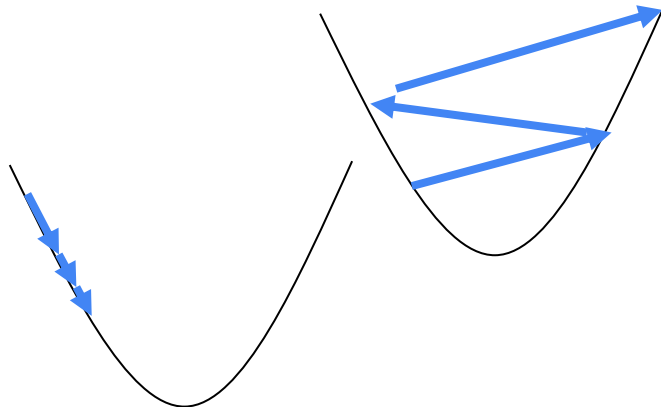
- Push  $\theta_i$  towards a direction that minimizes loss

- $\theta_i = \theta_i - \frac{\partial \text{loss}}{\partial \theta_i}$

- More generally  $\theta = \theta - \alpha \nabla \text{loss}$

- $\nabla \text{loss} = \left( \frac{\partial \text{loss}}{\partial \theta_1}, \frac{\partial \text{loss}}{\partial \theta_2}, \dots, \frac{\partial \text{loss}}{\partial \theta_n} \right)$

- $\alpha$  is the learning rate, requires tuning per domain, too large learning diverges to small results in slow learning or even premature convergence



# Stochastic Gradient Descent



# SGD for Monte Carlo estimation

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

Input: the policy  $\pi$  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  $S_0, A_0, R_1, S_1, A_1, \dots, R_T, S_T$  using  $\pi$

For  $t = 0, 1, \dots, T - 1$ :

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [G_t - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})$$

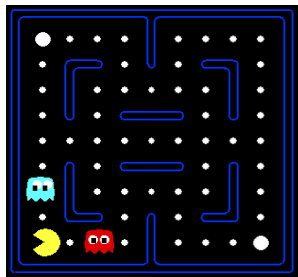
$\mathbf{w}$  are the tunable  
parameters of the value  
approximation function

18

- 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

$$\bullet S = \{f_1(S), f_2(S), f_3(S)\}$$

•  $f_{1,2}$ =distance to ghost 1,2,  $f_3$ =distance to food

$$\bullet \hat{v}(s) = \sum_i \theta_i f_i(s)$$

• init:  $\theta = [0, 0, 0]$

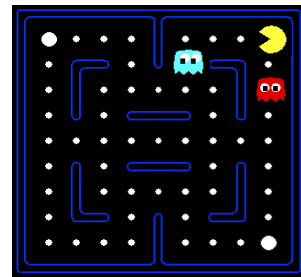
$$\bullet \theta = \theta + \alpha(G_t - \hat{v}(s; \theta)) \nabla \hat{v}(s; \theta)$$

$$\bullet \theta = [0, 0, 0] + 0.1(10 - [0, 0, 0] \cdot [2, 2, 1])[2, 2, 1]$$

•  $\theta = [2, 2, 1]$

$$\bullet \hat{v}(S') = f(S') \cdot \theta = [2, 4, 1] \cdot [2, 2, 1] = 13$$

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



# 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:  $\hat{v}(s_t) = \hat{v}(s_t) + \alpha(r_t + \gamma\hat{v}(s_{t+1}) - \hat{v}(s_t))$
  - Approximation TD learning:  $\theta = \theta + \alpha(r_t + \gamma\hat{v}(s_{t+1}; \theta) - \hat{v}(s_t; \theta))\nabla\hat{v}(s_t; \theta)$
- 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  $\pi$  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  $R, S'$

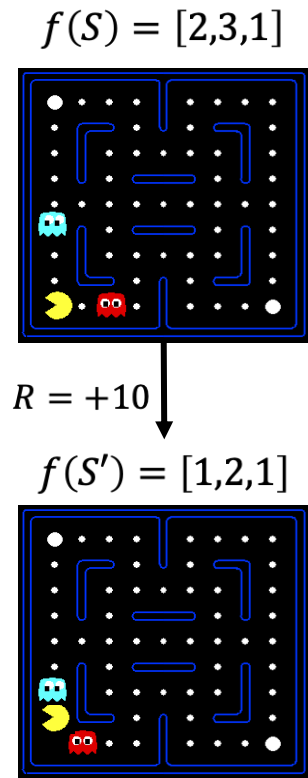
$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})] \nabla \hat{v}(S, \mathbf{w})$

$S \leftarrow S'$

    until  $S'$  is terminal

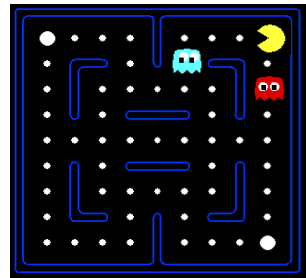
What's the difference  
from the tabular case?

# Example



- $S = \{f_1(S), f_2(S), f_3(S)\}$ 
  - $f_{1,2}$ =distance to ghost 1,2,  $f_3$ =distance to food
- $\hat{v}(s) = \sum_i \theta_i f_i(s)$ 
  - init:  $\theta = [0,0,0]$
- $\theta = \theta + \alpha(R + \gamma \hat{v}(S'; \theta) - \hat{v}(S; \theta)) \nabla \hat{v}(S; \theta)$
- $\theta = [0,0,0] + 0.1(10 + [1,2,1] \cdot [0,0,0] - [2,3,1] \cdot [0,0,0])[2,3,1]$ 
  - $\theta = [2,3,1]$
- $\hat{v}(U) = f(U) \cdot \theta = [2,4,1] \cdot [2,3,1] = 17^{23}$

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



# [Review] n-step TD Prediction

One-step return:

$$G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1})$$

Two-step return:

$$G_{t:t+2} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2})$$

1-step TD  
and TD(0)



2-step TD



3-step TD



...

n-step TD



∞-step TD  
and Monte Carlo



## [Review] n-step TD Prediction

One-step return:

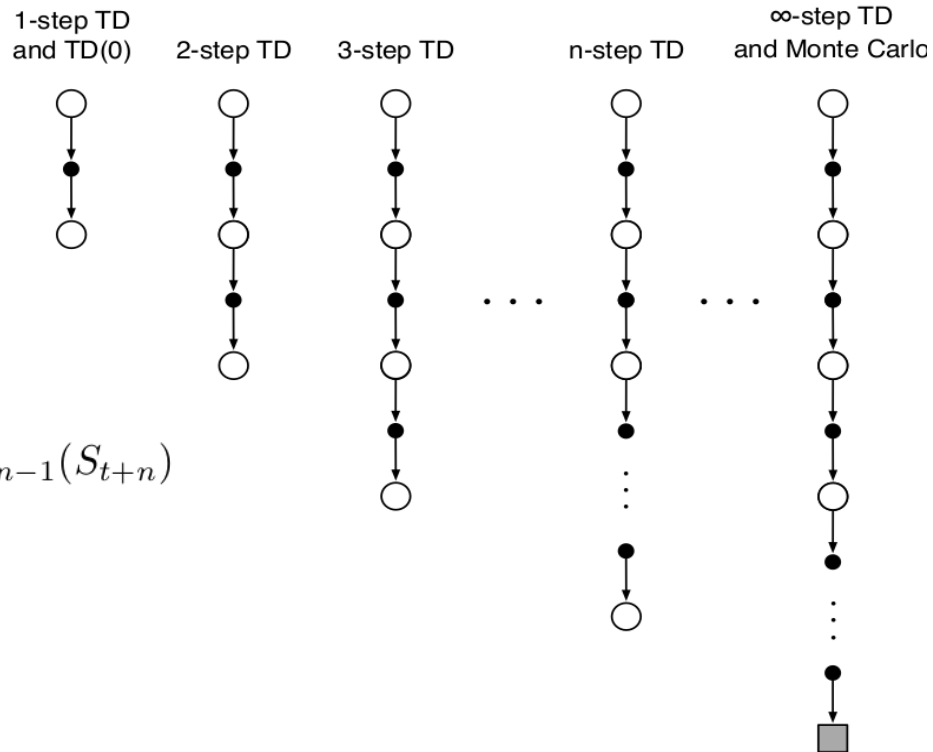
$$G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1})$$

### Two-step return:

$$G_{t:t+2} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2})$$

n-step return:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n})$$





# [Review] n-step TD Prediction

*n*-step TD for estimating  $V \approx v_\pi$

Input: a policy  $\pi$

Algorithm parameters: step size  $\alpha \in (0, 1]$ , a positive integer  $n$

Initialize  $V(s)$  arbitrarily, for all  $s \in \mathcal{S}$

All store and access operations (for  $S_t$  and  $R_t$ ) can take their index mod  $n + 1$

Loop for each episode:

    Initialize and store  $S_0 \neq \text{terminal}$

$T \leftarrow \infty$

    Loop for  $t = 0, 1, 2, \dots$ :

        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  $T \leftarrow t + 1$

$\tau \leftarrow t - n + 1$  ( $\tau$  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:  $G \leftarrow G + \gamma^n V(S_{\tau+n})$  ( $G_{\tau:\tau+n}$ )

$V(S_\tau) \leftarrow V(S_\tau) + \alpha [G - V(S_\tau)]$

    Until  $\tau = T - 1$

# $n$ -step return

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

Input: the policy  $\pi$  to be evaluated

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

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  $S_0 \neq \text{terminal}$

$T \leftarrow \infty$

    For  $t = 0, 1, 2, \dots$ :

        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  $T \leftarrow t + 1$

$\tau \leftarrow t - n + 1$  ( $\tau$  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:  $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n}, \mathbf{w})$  (28)

$\mathbf{w} \leftarrow \mathbf{w} + \alpha [G - \hat{v}(S_\tau, \mathbf{w})] \nabla \hat{v}(S_\tau, \mathbf{w})$

    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  $\pi$  to be evaluated

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

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  $S_0 \neq \text{terminal}$

$T \leftarrow \infty$

For  $t = 0, 1, 2, \dots$ :

    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  $T \leftarrow t + 1$

$\tau \leftarrow t - n + 1$  ( $\tau$  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:  $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n}, \mathbf{w})$  ( $G_{\tau:\tau+n}$ )

$\mathbf{w} \leftarrow \mathbf{w} + \alpha [G - \hat{v}(S_\tau, \mathbf{w})] \nabla \hat{v}(S_\tau, \mathbf{w})$

Until  $\tau = T - 1$

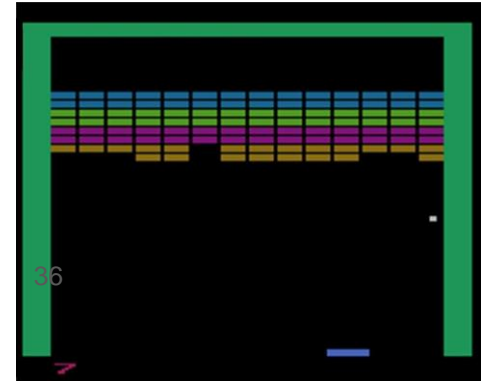
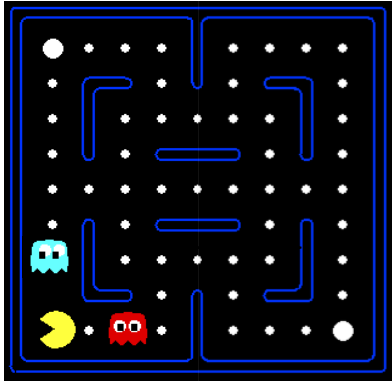
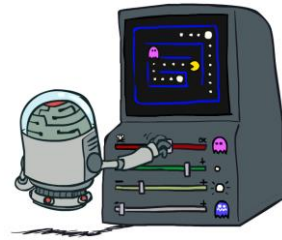
29

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



# Feature selection

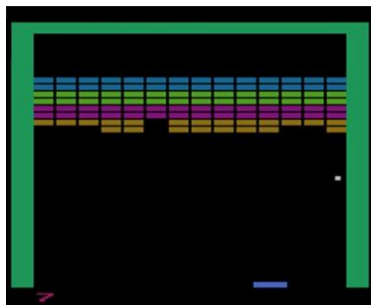
- Assume a linear function approximator  $\hat{v}(f(s); \theta) = f(s) \cdot \theta$
- What relevant features should represent states?



Features are domain depended  
requiring expert knowledge

# Automatic features extraction

- Consider a game state that is given as a bit map
- 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:**  $f_i(s) = \prod_{j=1}^k x_j^{c_{i,j}}$

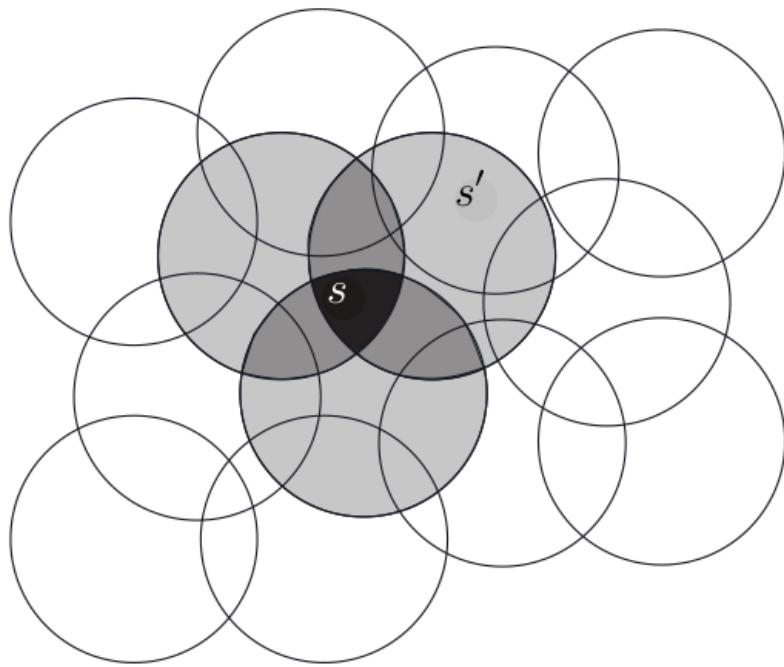
- where each  $c_{i,j}$  is an integer in the set  $\{0, 1, \dots, n\}$  for an integer  $n \geq 0$
- These features makeup the order- $n$  polynomial basis for dimension  $k$ , which contains  $(n + 1)^k$  different features

- **Fourier Basis:**  $f_i(s) = \cos(\pi X^\top c^i)$

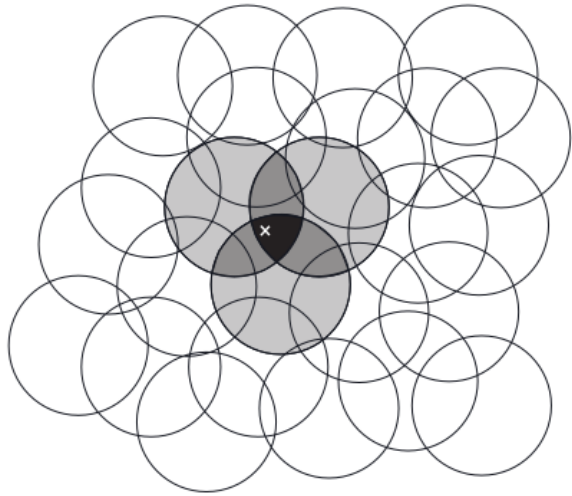
- Where  $c^i = (c_1^i, \dots, c_k^i)^\top$ , with  $c_j^i \in \{0, \dots, n\}$  for  $j = \{1, \dots, k\}$  and  $i = \{0, \dots, (n + 1)^k\}$
- This defines a feature for each of the  $(n + 1)^k$  possible integer vectors  $c^i$
- The inner product  $X^\top c^i$  has the effect of assigning an integer 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 - Coarse Coding

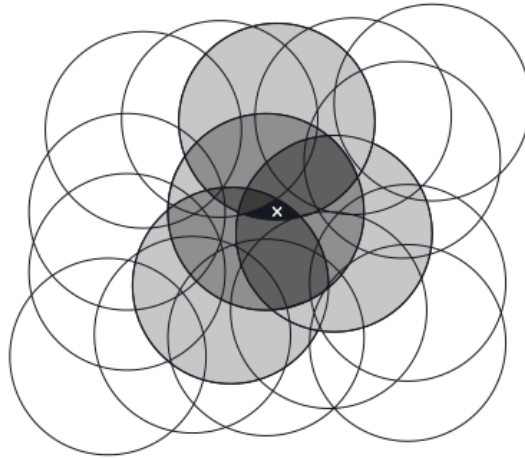
- Natural representation of the state set is continuous
- In 2-d, features corresponding to circles in state space
- Coding of a state:
  - If the state is inside a circle, then the corresponding feature has the value 1
  - otherwise the feature is 0
- Corresponding to each circle is a single weight (a component of  $w$ ) that is learned
  - Training a state affects the weights of all the intersecting circles.



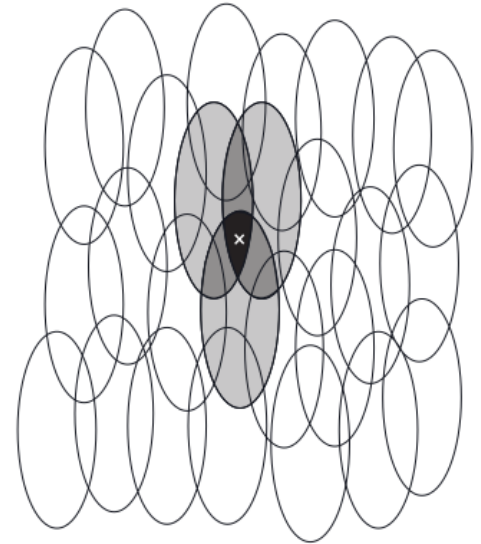
# Automatic features extraction for linear approximator - Coarse Coding



Narrow generalization

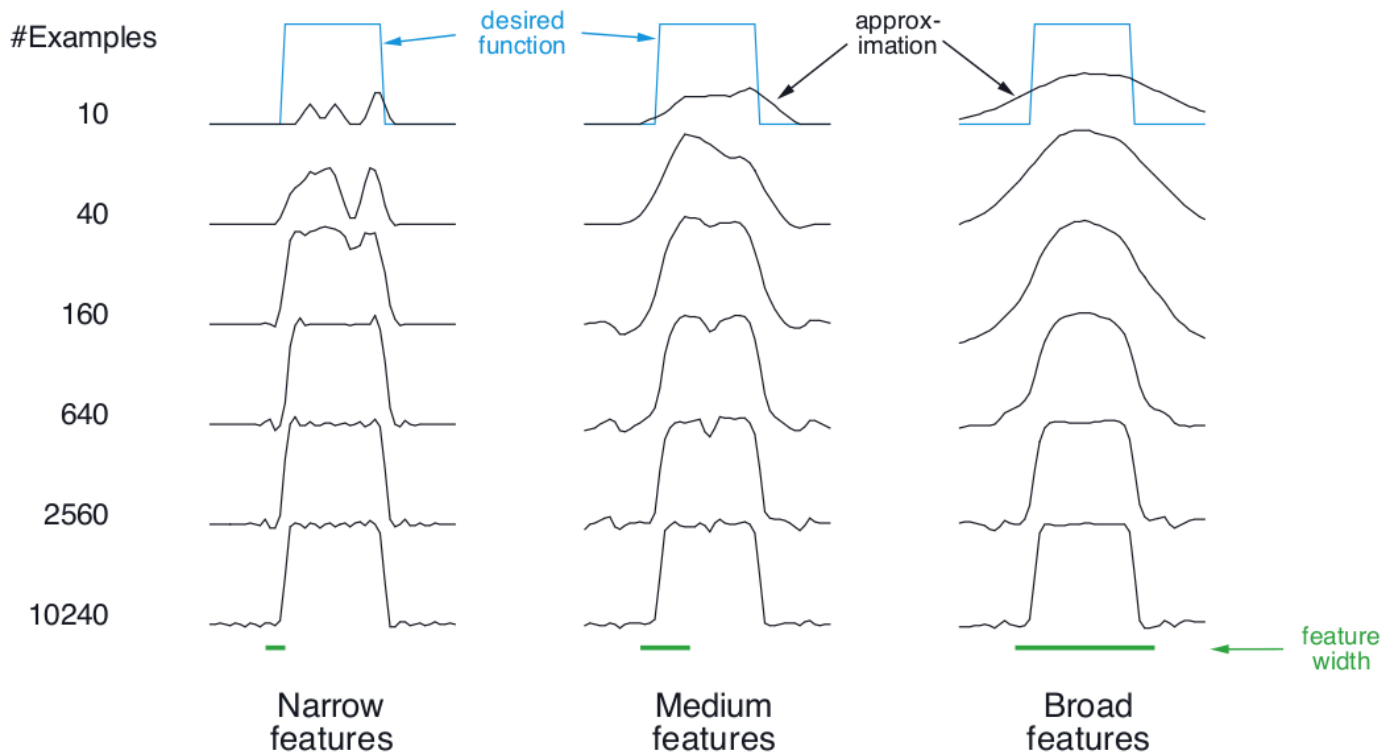


Broad generalization

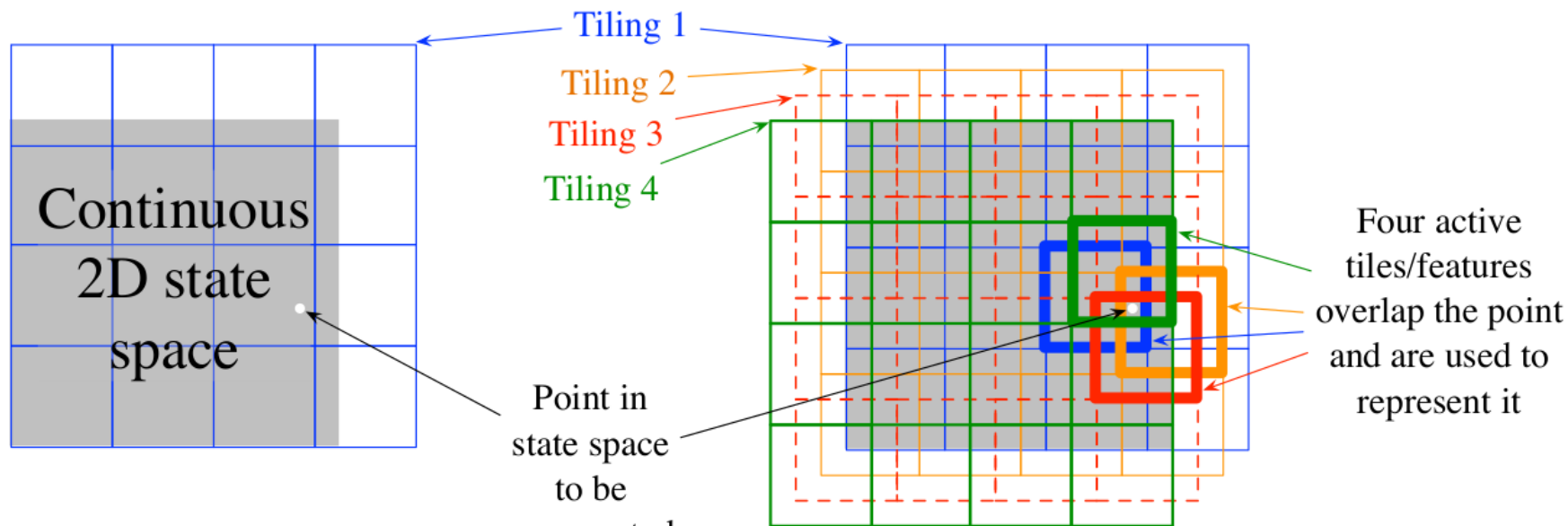


Asymmetric generalization

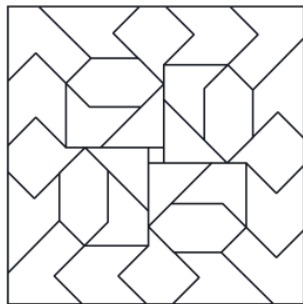
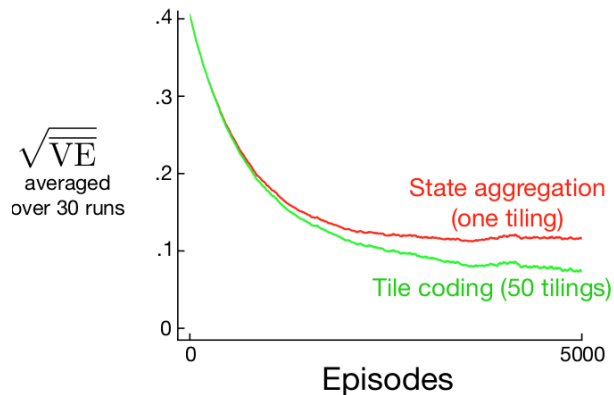
# Automatic features extraction for linear approximator - Coarse Coding



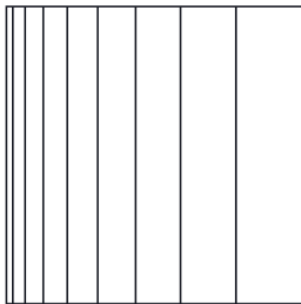
# Automatic features extraction for linear approximator - **Tile Coding**



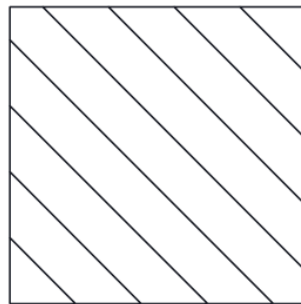
# Automatic features extraction for linear approximator - **Tile Coding**



Irregular



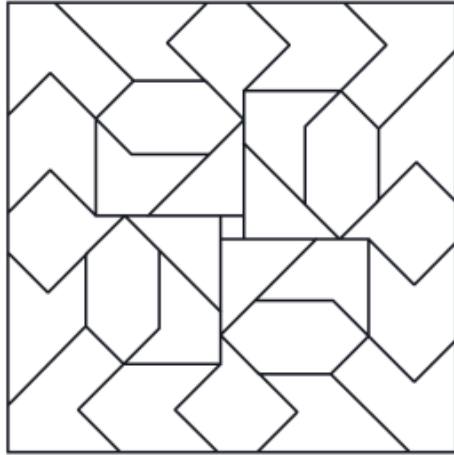
Log stripes



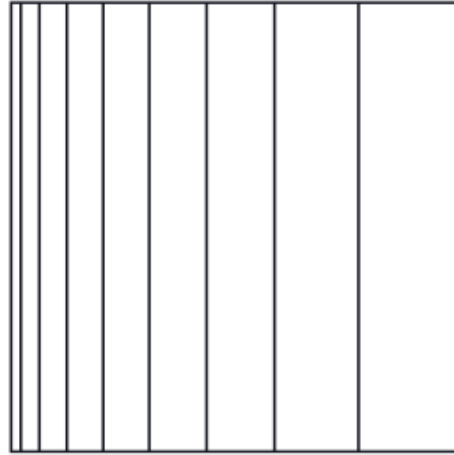
Diagonal stripes



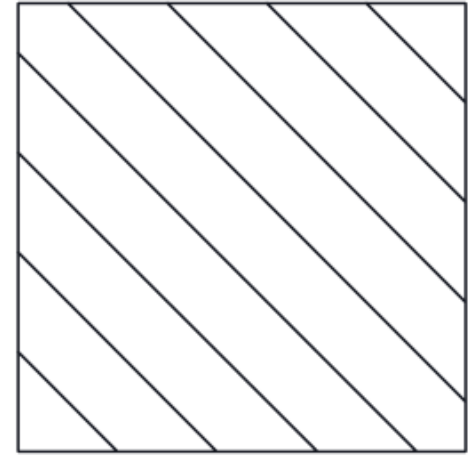
# Automatic features extraction for linear approximator - **Tile Coding**



Irregular



Log stripes

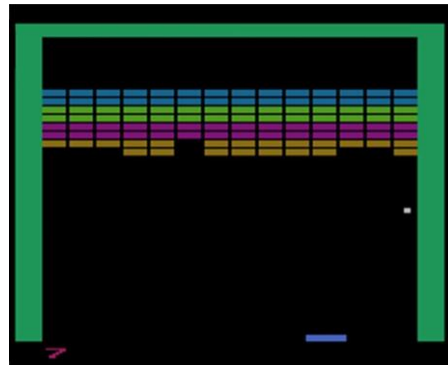


Diagonal stripes

# 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...

45



# 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



# Part-2 DQN

# Mnih et al. 2015

- First deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning
- The model is a convolutional neural network, trained with a variant of Q-learning
- Input is raw pixels and output is an action-value function estimating future rewards
- Surpassed a human expert on various Atari video games

# The age of deep learning

- Previous models relied on hand-crafted features combined with linear value functions
  - The performance of such systems heavily relies on the quality of the feature representation
- Advances in deep learning have made it possible to automatically extract high-level features from raw sensory data

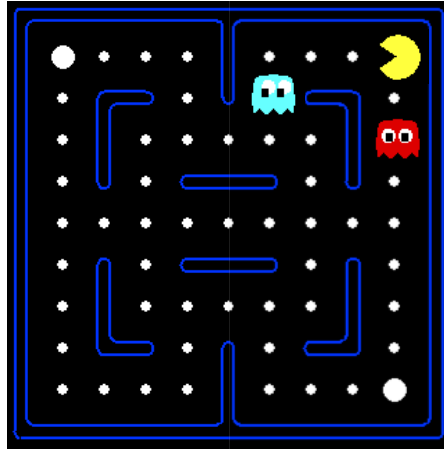


# Example: Pacman

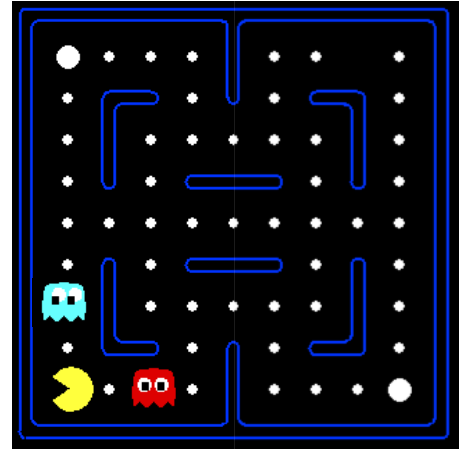
Let's say we discover through experience that this state is bad:



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



Or even this one!



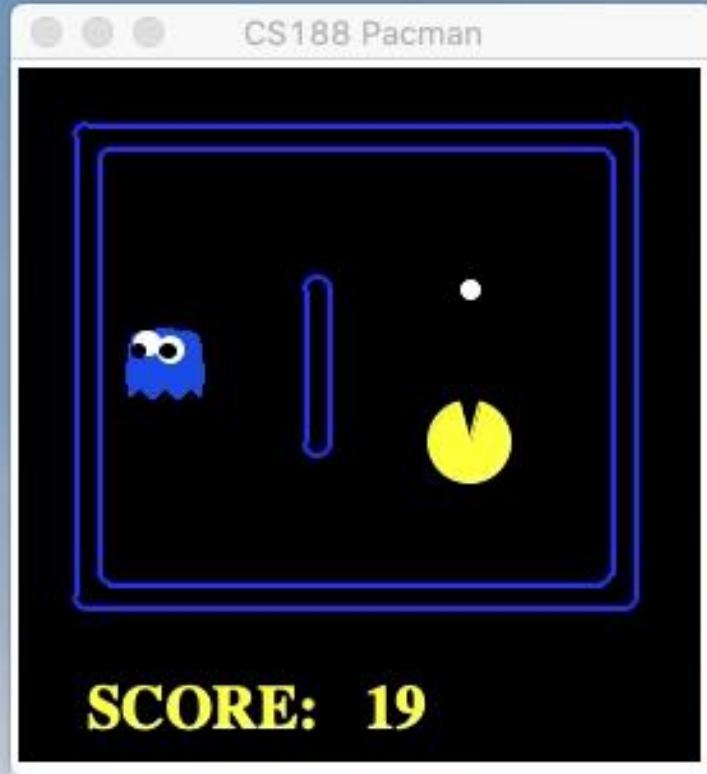
**We must generalize our knowledge!**

- Naïve Q-learning
- After 50 training episodes





- Generalize Q-learning with function approximator



- Generalizing knowledge results in efficient learning
- E.g., learn to avoid the ghosts



# Generalizing with Deep learning

- **Supervised:** Require large amounts of hand-labelled training data
  - **RL** on the other hand, learns from a scalar reward signal that is frequently sparse, noisy, and delayed
- **Supervised:** Assume the data samples are independent
  - In **RL** one typically encounters sequences of highly correlated state
- **Supervised:** Assume a fixed underlying distribution
  - In **RL** the data distribution changes as the algorithm learns new behaviors
- DQN was first to demonstrate that a convolutional neural network can overcome these challenges to learn successful control policies from raw video data in complex RL environments

# Deep Q learning [Mnih et al. 2015]

- Trains a generic neural network-based agent that successfully learns to operate in as many domains as possible
- The network is not provided with any domain-specific information or hand-designed features
- Must learn from nothing but the raw input (pixels), the reward, terminal signals, and the set of possible actions

# Original Q-learning

Q-learning (off-policy TD control) for estimating  $\pi \approx \pi_*$

Repeat (for each episode):

Initialize  $S$

Repeat (for each step of episode):

Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

Take action  $A$ , observe  $R, S'$

$$\theta = \theta + \alpha \left( R + \gamma \max_a \hat{Q}(S', a; \theta) - \hat{Q}(S, A; \theta) \right) \nabla_{\theta} \hat{Q}(S, A; \theta)$$

$S \leftarrow S'$

until  $S$  is terminal

# Deep Q learning [Mnih et al. 2015]

- DQN addresses problems of correlated data and non-stationary distributions
  - Use an experience replay mechanism
  - Randomly samples and trains on previous transitions
  - Results in a smoother training distribution over many past behaviors

# Deep Q learning [Mnih et al. 2015]

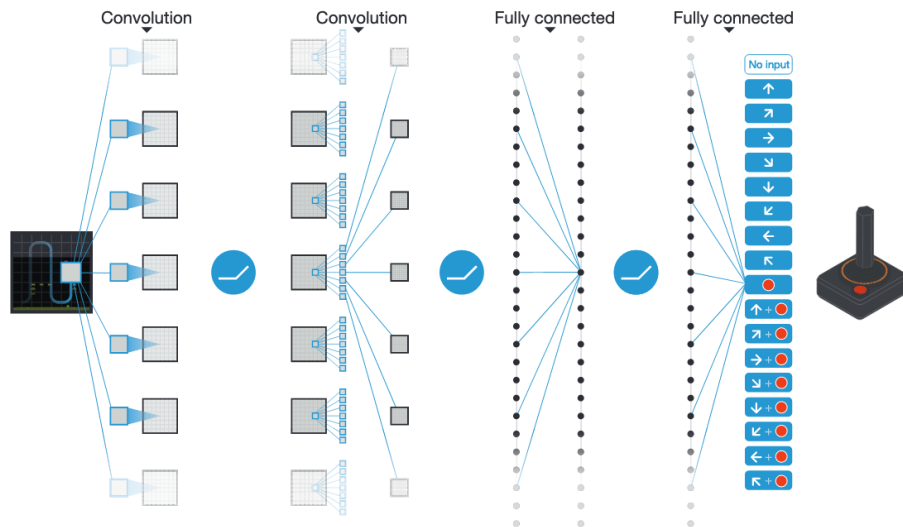
- Learn a function approximator (Q network),  $\hat{Q}(s, a; \theta) \approx Q^*(s, a)$
- Value propagation:  $Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[ r + \gamma \max_{a'} Q^*(s', a') \right]$ 
  - $\mathcal{E}$  represents the environment (underlying MDP)
- Update  $\hat{Q}(s, a; \theta)$  at each step  $i$  using SGD with squared loss:
- $L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} \left[ \left( y_i - \hat{Q}(s, a; \theta_i) \right)^2 \right]$ 
  - $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[ r + \gamma \max_{a'} \hat{Q}(s', a'; \theta_{i-1}) \right]$
  - $\rho(s, a)$  is the behavior distribution, e.g., epsilon greedy
  - $\theta_{i-1}$  is considered fix when optimizing the loss function (helps when taking the derivative with respect to  $\theta$  and with stability)

# Deep Q learning [Mnih et al. 2015]

$$\bullet L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)} \left[ \left( y_i - \hat{Q}(s, a; \theta_i) \right)^2 \right]$$

Independent of  $\theta_i$  (because  $\theta_{i-1}$  is considered fix )

$$\bullet \nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot), s' \sim \mathcal{E}} \left[ (r + \gamma \max_{a'} \hat{Q}(s', a'; \theta_{i-1}) - \hat{Q}(s, a; \theta_i)) \nabla_{\theta_i} \hat{Q}(s, a; \theta_i) \right]$$





# Q-learning with experience replay

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$

Initialize the replay memory and two identical Q approximators (DNN).  $\hat{Q}$  is our target approximator.

**For** episode = 1,  $M$  **do**

Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$

**For**  $t = 1, T$  **do**

With probability  $\varepsilon$  select a random action  $a_t$

otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

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  $D$

Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$

Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the network parameters  $\theta$

Every  $C$  steps reset  $\hat{Q} = Q$

**End For**

**End For**

# Q-learning with experience replay

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$

**For** episode = 1,  $M$  **do** ← Play  $m$  episodes (full games)

Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$

**For**  $t = 1, T$  **do**

With probability  $\varepsilon$  select a random action  $a_t$

otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

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  $D$

Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$

Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the network parameters  $\theta$

Every  $C$  steps reset  $\hat{Q} = Q$

**End For**

**End For**

# Q-learning with experience replay

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$

**For** episode = 1,  $M$  **do**

Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$

**For**  $t = 1, T$  **do**

With probability  $\varepsilon$  select a random action  $a_t$

otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

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  $D$

Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$

Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the network parameters  $\theta$

Every  $C$  steps reset  $\hat{Q} = Q$

**End For**

**End For**

Start episode from  $x_1$  (pixels at the starting screen).

Preprocess the state (include 4 last frames, RGB to grayscale conversion, downsampling, cropping)

# Q-learning with experience replay

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$

**For** episode = 1,  $M$  **do**

Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$

**For**  $t = 1, T$  **do** ← For each time step during the episode

With probability  $\varepsilon$  select a random action  $a_t$

otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

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  $D$

Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$

Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the network parameters  $\theta$

Every  $C$  steps reset  $\hat{Q} = Q$

**End For**

**End For**

# Q-learning with experience replay

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$

**For** episode = 1,  $M$  **do**

Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$

**For**  $t = 1, T$  **do**

With probability  $\varepsilon$  select a random action  $a_t$   
otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$  }

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  $D$

Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$

Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the network parameters  $\theta$

Every  $C$  steps reset  $\hat{Q} = Q$

**End For**

**End For**

With small probability select a random action (explore), otherwise select the, currently known, best action (exploit).

# Q-learning with experience replay

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$

**For** episode = 1,  $M$  **do**

Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$

**For**  $t = 1, T$  **do**

With probability  $\varepsilon$  select a random action  $a_t$

otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

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  $D$

Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$

Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the network parameters  $\theta$

Every  $C$  steps reset  $\hat{Q} = Q$

**End For**

**End For**

Execute the chosen action and store the (processed) observed transition in the replay memory

# Q-learning with experience replay

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$

**For** episode = 1,  $M$  **do**

Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$

**For**  $t = 1, T$  **do**

With probability  $\varepsilon$  select a random action  $a_t$

otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

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  $D$

Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$

Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the network parameters  $\theta$

Every  $C$  steps reset  $\hat{Q} = Q$

**End For**

**End For**

Experience replay:  
Sample a random minibatch of transitions from replay memory and perform gradient decent step on  $Q$  (not on  $\hat{Q}$ )

# Q-learning with experience replay

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$

**For** episode = 1,  $M$  **do**

Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$

**For**  $t = 1, T$  **do**

With probability  $\varepsilon$  select a random action  $a_t$

otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

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  $D$

Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$

Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the network parameters  $\theta$

Every  $C$  steps reset  $\hat{Q} = Q$

Once every several steps set the target function,  $\hat{Q}$ , to equal  $Q$

**End For**

**End For**



# Q-learning with experience replay

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$

**For** episode = 1,  $M$  **do**

Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$

**For**  $t = 1, T$  **do**

With probability  $\varepsilon$  select a random action  $a_t$

otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

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  $D$

Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$

Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the network parameters  $\theta$

Every  $C$  steps reset  $\hat{Q} = Q$

**End For**

**End For**

Such delayed online learning helps in practice:

“This modification makes the algorithm more stable compared to standard online Q-learning, where an update that increases  $Q(s_t, a_t)$  often also increases  $Q(s_{t+1}, a)$  for all  $a$  and hence also increases

the target  $y_j$ , possibly leading to oscillations or divergence of the policy”

[Human-level control through deep reinforcement learning. Nature 518.7540 (2015): 529.]

# Deep Q learning

- *model-free*: solves the reinforcement learning task directly using samples from the emulator  $\mathcal{E}$ , without explicitly learning an estimate of  $\mathcal{E}$
- *off-policy*: learns the optimal policy,  $a = \underset{a}{\operatorname{argmax}} Q(s, a; \theta)$ , while following a different behavior policy
  - One that ensures adequate exploration of the state space

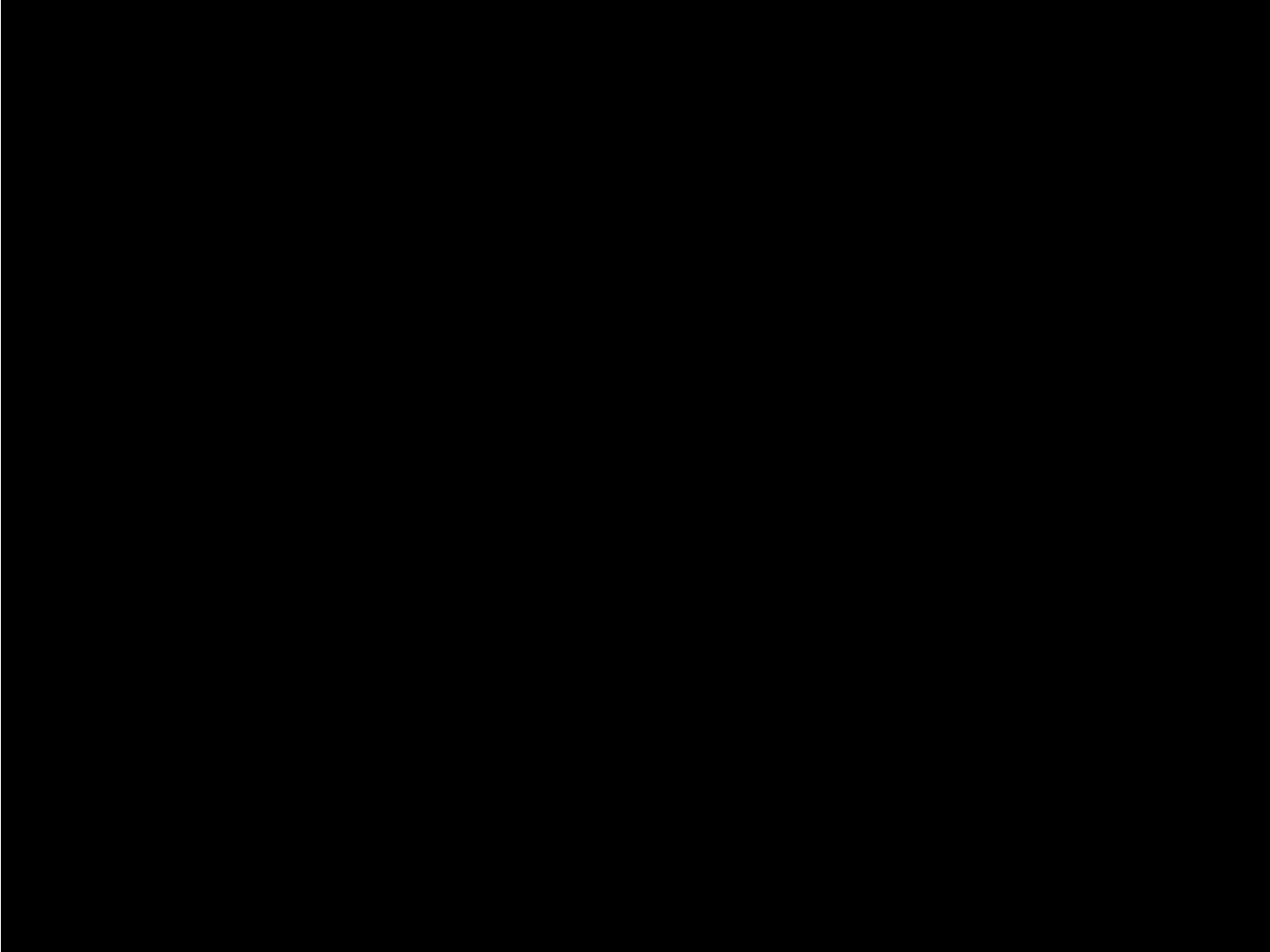
# Experience replay

- Utilizing experience replay has several advantages
  - Each step of experience is potentially used in many weight updates, which allows for greater data efficiency
  - Learning directly from consecutive samples is inefficient, due to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the variance of the updates
  - The behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters
- Note that when learning by experience replay, it is necessary to learn off-policy (because our current parameters are different to those used to generate the sample), which motivates the choice of Q-learning

# Experience replay

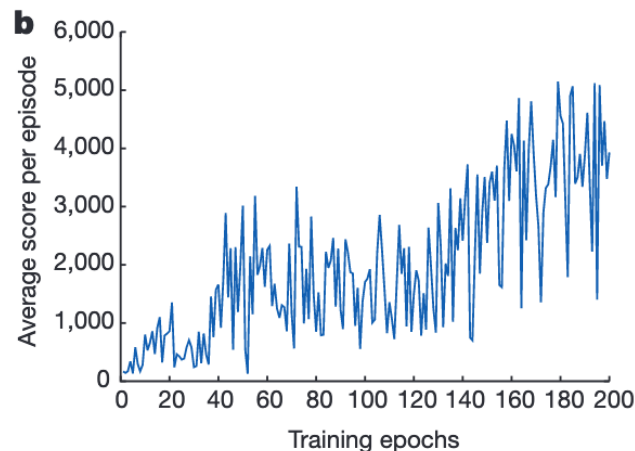
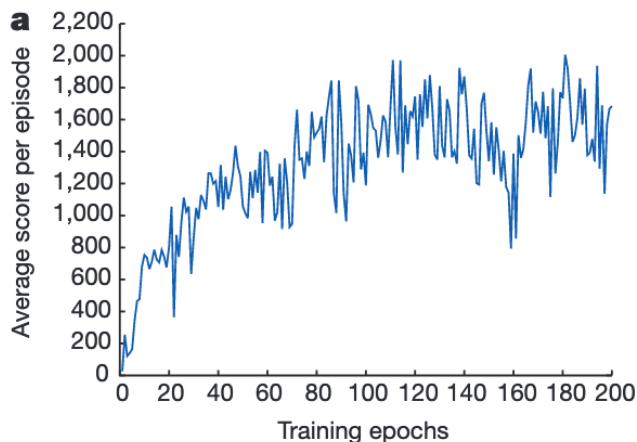
- DQN only stores the last N experience tuples in the replay memory
  - Old transitions are overwritten
- Samples uniformly at random from the buffer when performing updates
- Is there room for improvement?
  - Important transitions?
    - Prioritized sweeping
  - Prioritize deletions from the replay memory
  - see prioritized experience replay, <https://arxiv.org/abs/1511.05952>

**Before training**  
peaceful swimming



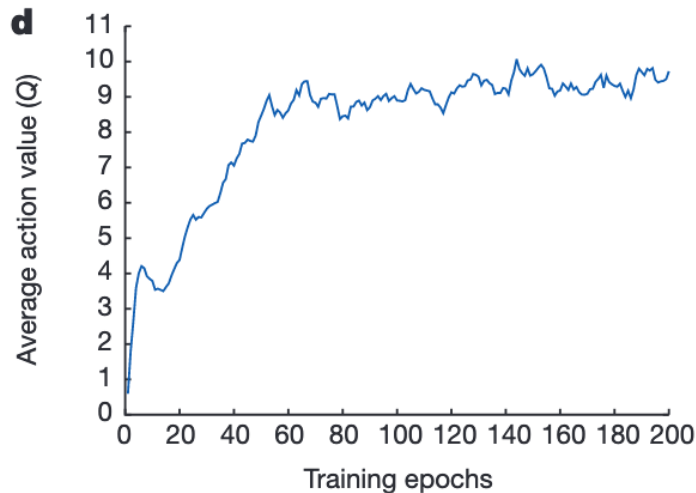
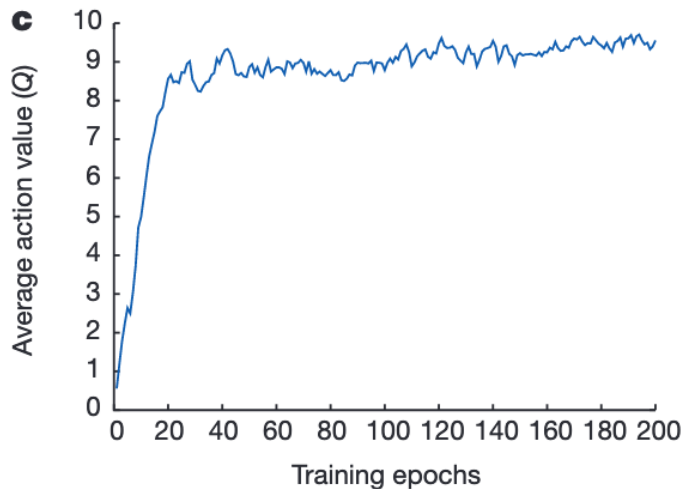
# Results: DQN

- Each point is the average score achieved per episode after the agent is run with an epsilon-greedy policy ( $\epsilon = 0.05$ ) for 520k frames on SpaceInvaders (a) and Seaquest (b)

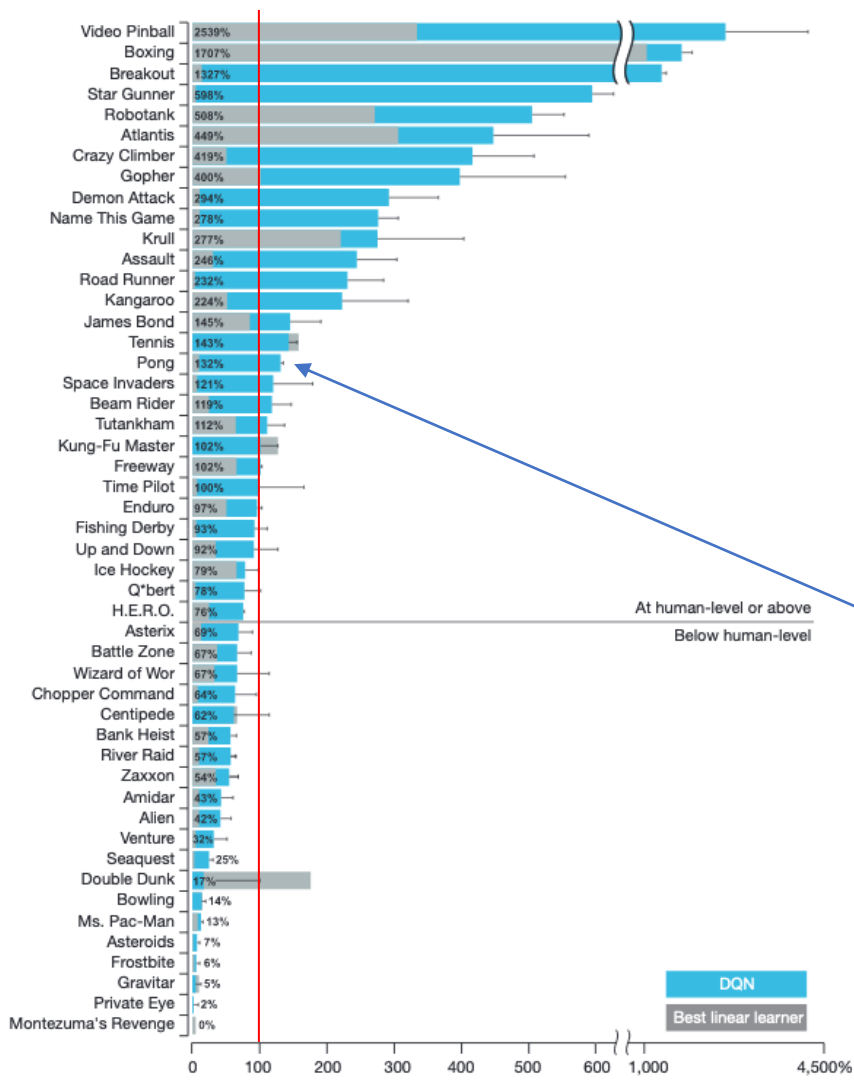


# Results: DQN

- Average predicted action-value on a held-out set of states on Space Invaders (c) and Seaquest (d)







- Normalized between a professional human games tester (100%) and random play (0%)
- E.g., in Pong, DQN achieved a factor of 1.32 higher score on average when compared to a professional human player

# Maximization bias

- See lecture 5TD\_learning.pptx, slide 41
- The max operator in Q-learning uses the same values both to select and to evaluate an action
  - $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[ r + \gamma \max_{a'} \hat{Q}(s', a'; \theta_{i-1}) \right]$
  - Makes it more likely to select overestimated values, resulting in overoptimistic value estimates
- We can use a technique similar to the previously discussed Double Q-learning (lecture 5TD\_learning.pptx, slide 43)
  - Two value functions are trained. Update only one of the two value functions at each step while considering the  $\max_a$  from the other

# Double Deep Q networks (DDQN)

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$

**For** episode = 1,  $M$  **do**

Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$

**For**  $t = 1, T$  **do**

With probability  $\varepsilon$  select a random action  $a_t$

otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

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  $D$

Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$

Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$


Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the network parameters  $\theta$

Every  $C$  steps reset  $\hat{Q} = Q$

**End For**

**End For**

We have two available  $Q$  networks. Let's use them for double learning



# Double Deep Q networks (DDQN)

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$

**For** episode = 1,  $M$  **do**

Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$

**For**  $t = 1, T$  **do**

With probability  $\epsilon$  select a random action  $a_t$

otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

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  $D$

Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$

Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the network parameters  $\theta$

Every  $C$  steps reset  $\hat{Q} = Q$

**End For**

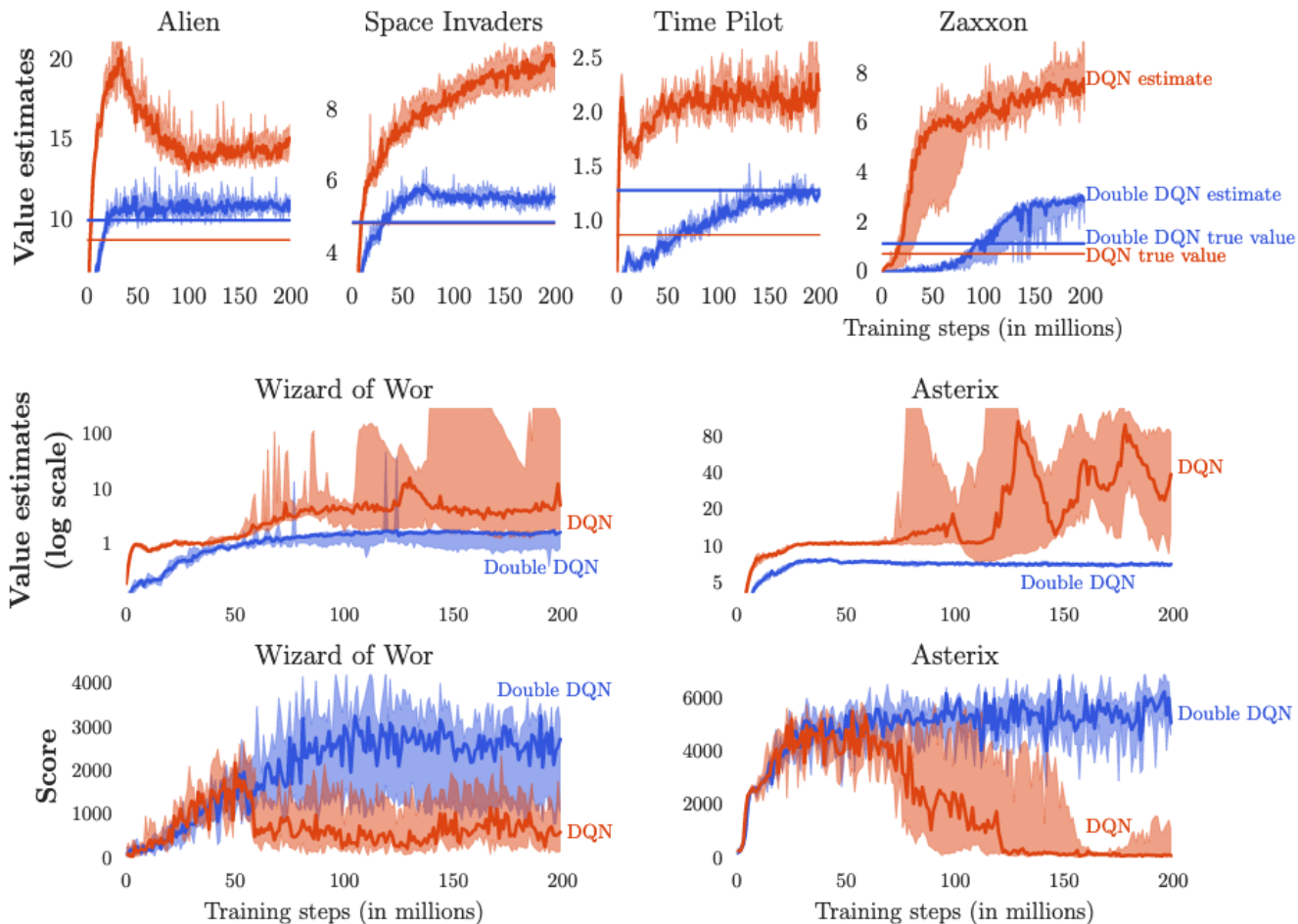
**End For**

We have two available Q networks. Let's use them for double learning

$$r_j + \gamma \hat{Q}(\phi_{j+1}, \operatorname{argmax}_{a'} Q(\phi_{j+1}, a'; \theta), \theta^-)$$

- Hasselt et al. 2015

- The straight horizontal orange (for DQN) and blue (for Double DQN) lines in the top row are computed by running the corresponding agents after learning concluded, and averaging the actual discounted return obtained from each visited state. These straight lines would match the learning curves at the right side of the plots if there is no bias.



# What did we learn?

- Using deep neural networks as function approximators in RL is tricky
  - Sparse samples
  - Correlated samples
  - Evolving policy (nonstationary sample distribution)
- DQN attempts to address these issues
  - Reuse previous transitions at each training (SGD) step
  - Randomly sample previous transitions to break correlation
  - Use off-policy, TD(0) learning to allow convergence to the true target values ( $Q^*$ )
    - No guarantees for non-linear (DNN) approximators



## Required Readings

1. Chapter-6 of Introduction to Reinforcement Learning, 2<sup>nd</sup> Ed., Sutton & Barto



**BITS Pilani**  
Pilani | Dubai | Goa | Hyderabad

Thank you