CSCI E-82a
Probabilistic Programming and AI
Lecture 14
Reinforcement Learning with Function Approximation

Steve Elston

HARVARD
Extension School

RL with Function Approximation

- Why use function approximation for RL?
- Function approximators and basis functions
- Linear approximators and coding
- Linear functions and gradient descent
- The mountain car problem
- Solving the mountain car problem
- Q-learning and function approximation
- Deep Q Network algorithm
- Double DQN algorithm
- Prioritize replay

Why Use Function Approximation?

- Up to now, only used **tabular algorithms**
- Tabular algorithms have limits of scalability
  - Value function needs **table entry for every state**
  - Action-value function needs **table entry for every action-value pair**
- Problems with **large number of discrete states and actions**
  - Backgammon: 10^{20} states
  - Go: 10^{170} states
- Problems with **continuous variables**
  - Examples; temperature, pressure, acceleration, velocity, etc.
  - Have an **infinite number of states**

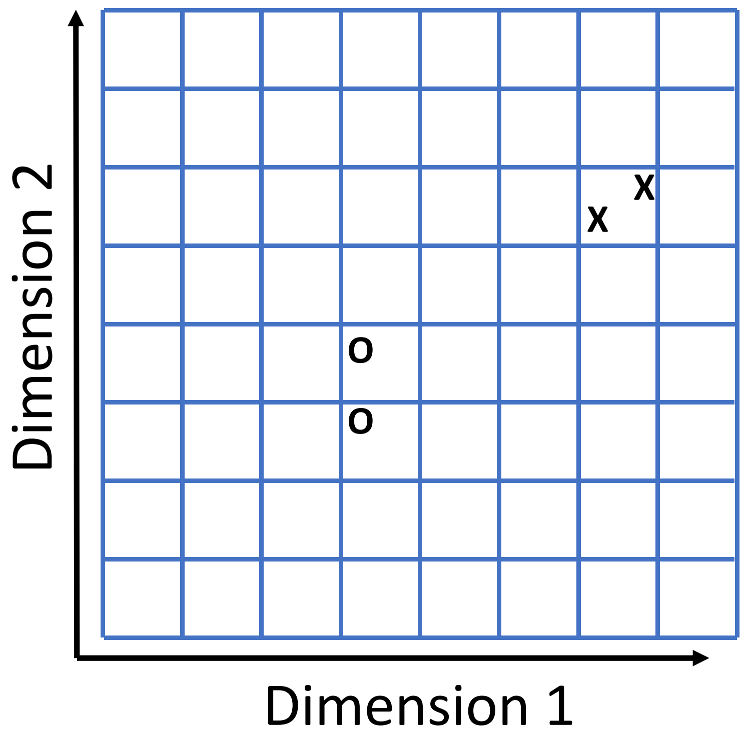
Function Approximators

- How to scale beyond **tabular algorithms?**
- Need a better **representation**
- Use **function approximation**
  - Agent **learns value function**
    v(s) ~ f(s,w) = function of features w
  - Or, agent **learns action-value function**
    q(s,a) ~ f(s,a,w) = function of features w
  - f(s,w), f(s,a,w) have **sparse number of parameters** compared to original space

Function Approximators

- Which **function approximators** to use?
- **Linear function approximators**
  - Grid coding
  - Coarse coding
  - Fourier and wavelet basis function
  - Radial basis function – e.g. Gaussian
  - Splines – generalized additive models (GAM)
- **Nonlinear function approximators**
  - Nearest neighbors
  - Decision trees
  - Deep neural networks
  - More on these later

Tile Coding



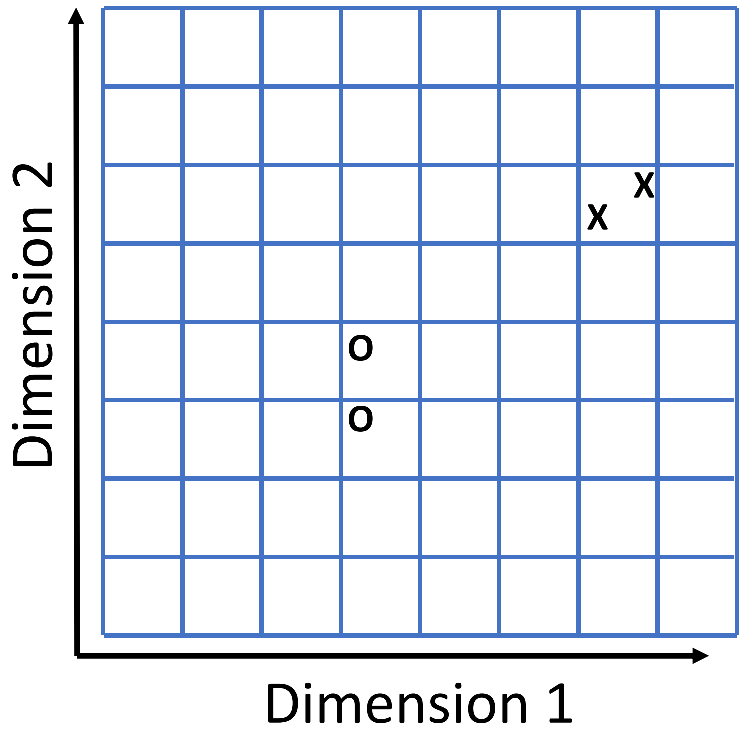
O coded in two tiles
Xs coded in one tile

Two continuous variables

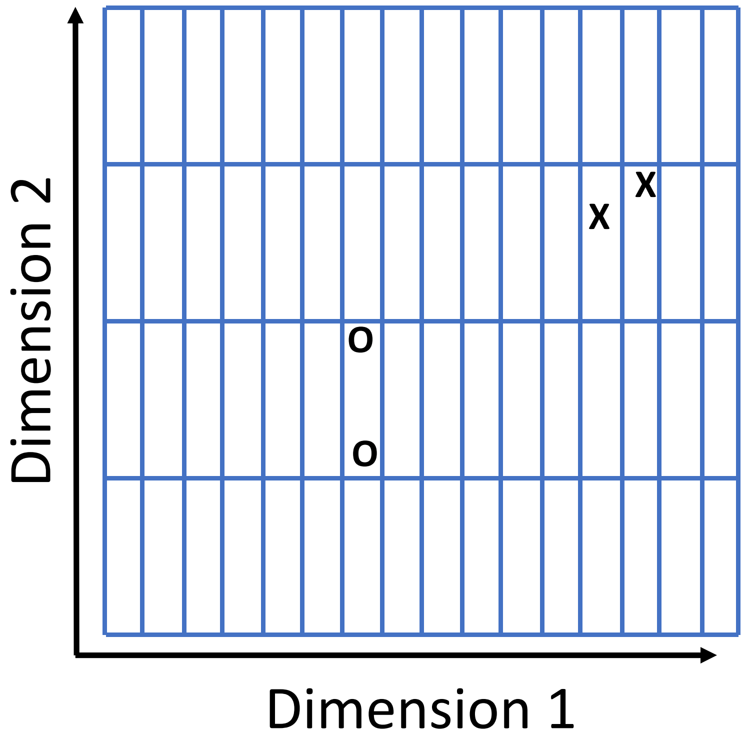
Divided values on rectangular grid

Values coded on the grid

Tile Coding



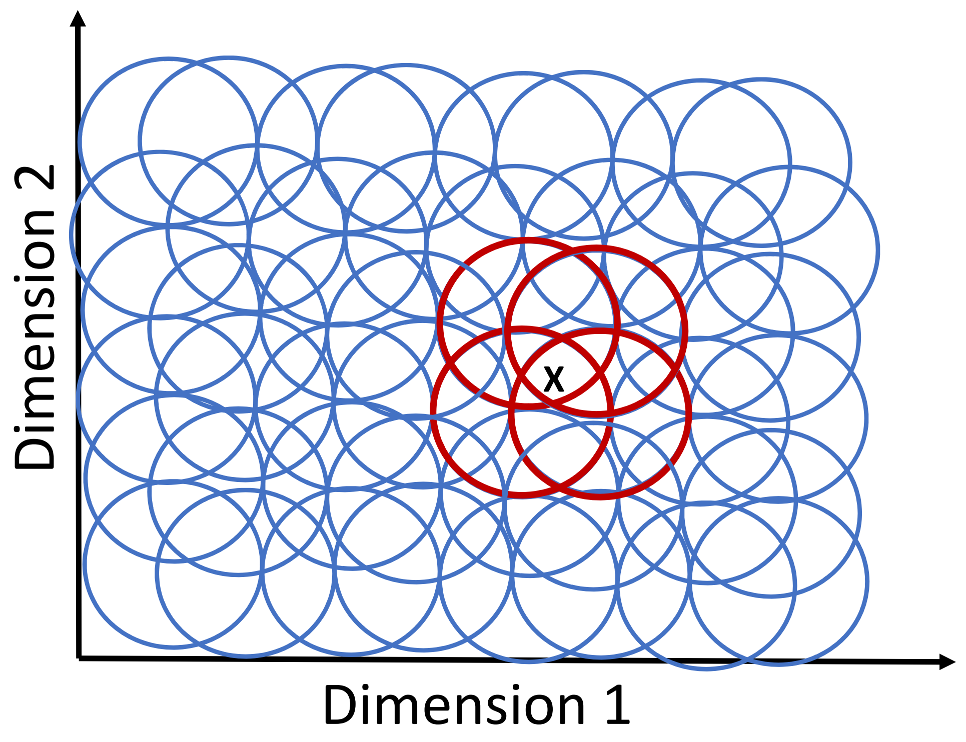
Os coded in two tiles
Xs coded in one tile



Os coded in one tile
Xs coded in one tile

Coarse Coding

Example**, overlapping coarse coding**



Point coded in 4 circles

Linear Function Coding

Grid coding is a linear function approximator

- Each **tile is a feature**
- Each feature is **linear in one parameter** or **model weight**
- Function approximator is a **linear model**

Linear Function Coding

The value function as a **linear function of model weights**

- The approximate value function: $\hat{v}(s, \mathbf{w})$
- The **weight vector**, **w**
- Coding is a **binary function** of state: $\mathbf{x}(s)$
- The coded approximate value function is then:

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s) = \sum_{i=1}^{d} w_i x_i(s)$$

Linear Functions and Gradient Descent

How good is the approximation?

- Like any approximator there is a difference between the estimated and actual values
- Can use **value function approximation**, $\hat{v}(S_t, \mathbf{w}_t)$, of the true state-value given a policy π, $v_\pi(s)$
- OR, an **action-value function approximation**, $\hat{q}(s, a)$, of the true action-value value given a policy π, $q_\pi(s, a)$

Linear Functions and Gradient Descent

How good is the approximation?

- Compute the **mean square value error** between $v_\pi(s)$ and $\hat{v}(S_t, \mathbf{w}_t)$ as the metric or **loss function**:

$$\overline{VE}(w) = \sum_{s \in S} \mu(s) \Big[v_\pi(s) - \hat{v}(s, \mathbf{w}) \Big]^2$$

- Where, $\mu(s)$ is the **probability of being in state, s**
- For on-policy algorithms $\mu(s)$ is known as the **on-policy distribution**

Linear Functions and Gradient Descent

Solve linear system of equations with **gradient descent**

- Must compute a **gradient with respect to the d-dimensional weight vector, w**

$$\nabla_w \hat{v}(S_t, \mathbf{w}_t) = \begin{bmatrix} \frac{\partial \hat{v}(S_t, \mathbf{w}_t)}{\partial w_1} \\ \frac{\partial \hat{v}(S_t, \mathbf{w}_t)}{\partial w_2} \\ \vdots \\ \frac{\partial \hat{v}(S_t, \mathbf{w}_t)}{\partial w_d} \end{bmatrix}$$

Linear Functions and Gradient Descent

Solution with **stochastic gradient descent**

- For a **random sample of the dat**a, $\hat{p}data$, and a **loss function** $J(\mathbf{w}_t)$ the **stochastic gradient descent update** is:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\, E_{\hat{p}data}\Big[\nabla_w J(\mathbf{w}_t)\Big]$$

- Using the estimated state-value function, $\hat{v}(S_t, \mathbf{w}_t)$, the stochastic gradient descent update becomes

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\Big[v_\pi(s) - \hat{v}(S_t, \mathbf{w}_t)\Big]\nabla_w \hat{v}(S_t, \mathbf{w}_t)$$

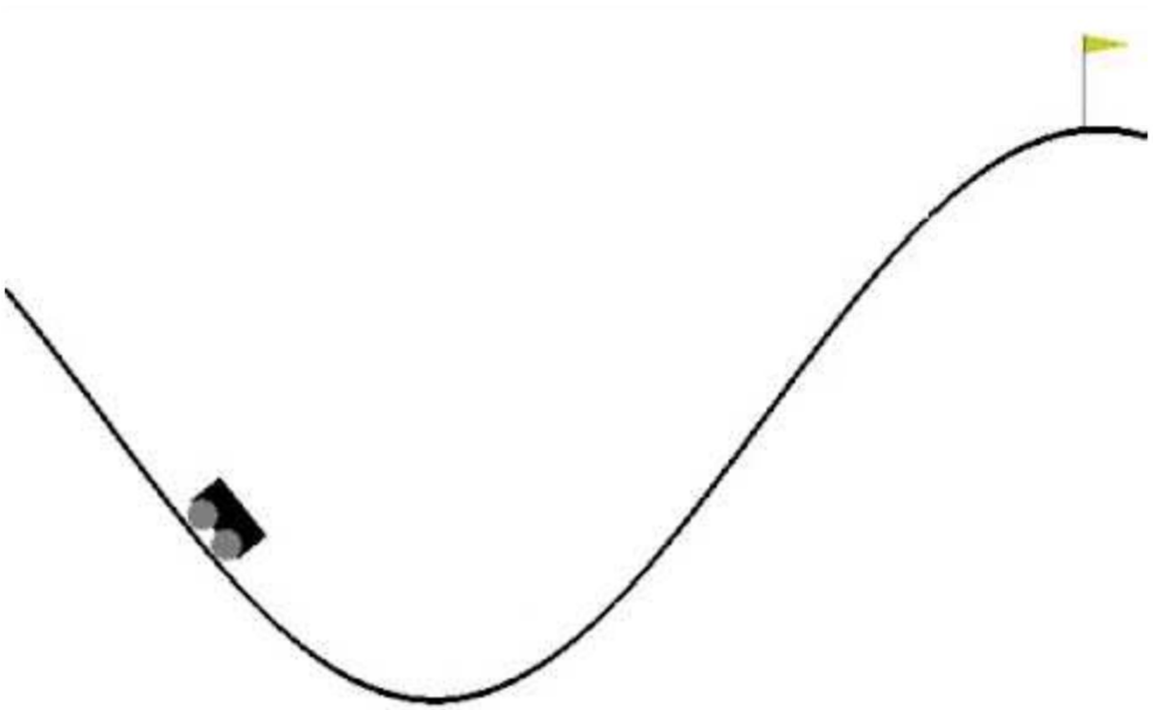
- Since $\hat{v}(S_t, \mathbf{w}_t)$ is a **bootstrap estimate** of $v_\pi(s)$, $\nabla_w \hat{v}(S_t, \mathbf{w}_t)$ is a **semi-gradient**

Mountain Car Problem

- The mountain car problem (Moore, 1990) is a canonical control problem used to test RL algorithms
  - An under-powered car must travel to the top of a hill
  - The car has 3 actions; accelerate forward, +1, backward, -1, and neutral, 0
  - The agent must learn a policy to get the car to the top of the hill
- There are two state variables
  - Position
  - Velocity

Mountain Car Problem

Under-powered car must gain momentum to get to the goal at the top of the mountain



Mountain Car Problem

- The state equation for car position is:

$$x' = x + \dot{x}$$

- The state equation for car velocity is:

$$\dot{x}' = \dot{x} + 0.001 * \ddot{x} - 0.0025 * cos(3 * x)$$

- Acceleration, is the action, determined by the agent from the set:

$$\ddot{x} = \{-1.0, 0.0, 1.0\}$$

Mountain Car Problem

- Car reward function:
  - Each time step, -1
  - At goal, 100
- The position and velocity of the car are limited

$$-1.2 \leq x \leq 0.5$$
$$-0.07 \leq \dot{x} \leq 0.07$$

- Starting position of car is random

$$p(x_0) = uniform(-0.6 \leq x_0 \leq -0.4)$$

Solving Mountain Car Problem is Hard!

Why is the Mountain Car Problem Hard?

- Several characteristics make this problem difficult for an agent to learn a good policy
- **Positive reward delayed** to end of episode
- **State variables**, position and velocity, are **continuous**
- The **relationship** between the state variables is **nonlinear**

Solving the Mountain Car Problem

Use 3-dimensional tile coding

- First dimension divides the position interval, $-1.2 \leq x \leq 0.5$
- Section dimension divides the velocity interval, $-0.07 \leq \dot{x} \leq 0.07$
- Third dimension has three steps for the acceleration state - action, $\{-1.0, 0.0, 1.0\}$

- Coding function $x_i(s, a)$ has values:

$$x_i(s, a) = 1 \textit{ if in tile } i$$
$$x_i(s, a) = 0 \textit{ otherwise}$$

Solving the Mountain Car Problem

Use a d x d tile approximate linear function for action-values

$$q(s,a) \approx \hat{q}(s,a,\mathbf{w}) = \sum_{i=1}^{d} \sum_{j=1}^{d} w_{i,j} x_{i.j}(s,a)$$

- Where,

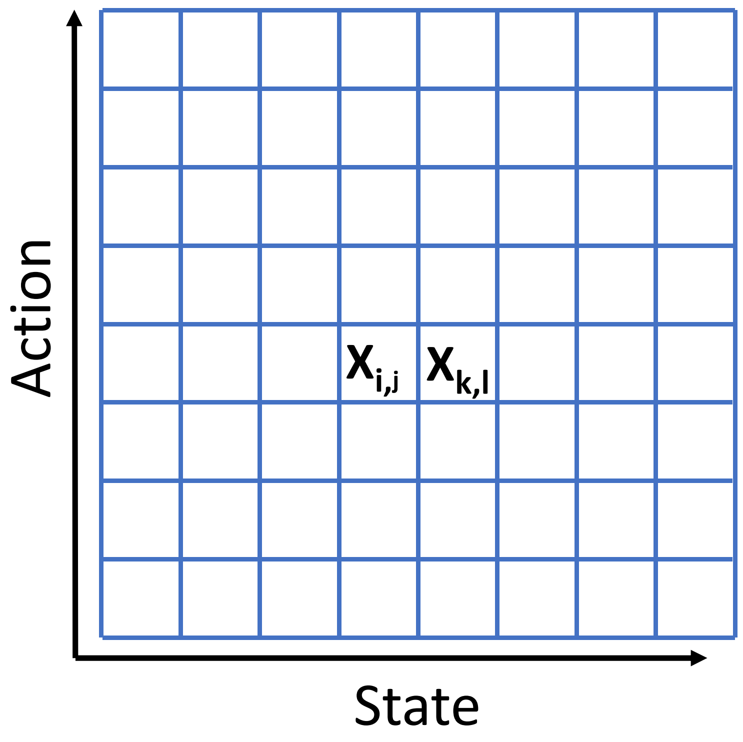
$\hat{q}(s,a,\mathbf{w})$ = the approximate state-value function

$x_{i.j}(s,a)$ = state-action tile coding function, {0,1}

$w_{i,j}$ = function weights, which must be learned

s = state variable tuple, position and velocity

Independence in Tile Coding



- Consider state action grid coding
- Grid coding is binary, $x_{i,j} = \{0,1\}$
- Grid coding is independent
- A state action is in one grid cell or the other

  Coding i,j independent of coding k,l

Solving the Mountain Car Problem

Need to find the semi-gradient of the linear function approximation

$$\nabla_w \hat{q}(S_t, A_t, \mathbf{w}_t) = \begin{bmatrix} \frac{\partial \hat{q}(S_t, A_t, \mathbf{w}_t)}{\partial w_{1,1}} \\ \frac{\partial \hat{q}(S_t, A_t, \mathbf{w}_t)}{\partial w_{1,2}} \\ \vdots \\ \frac{\partial \hat{q}(S_t, A_t, \mathbf{w}_t)}{\partial w_{d,d}} \end{bmatrix} = \begin{bmatrix} \frac{\partial x_{1,1} w_{1,1}}{\partial w_{1,1}} \\ \frac{\partial x_{1,2} w_{1,2}}{\partial w_{1,2}} \\ \vdots \\ \frac{\partial x_{d,d} w_{d,d}}{\partial w_{d,d}} \end{bmatrix}$$

Solving the Mountain Car Problem

Need to find the semi-gradient of the linear function approximation

The gradient is linear in the weights, $w_{i,j}$, and all $x_{i,j}$ = {0,1} are independent, so:

$$\nabla_w \hat{q}(S_t, A_t, \mathbf{w}) = \begin{cases} 1, \text{ if } x_{i,j}(s,a) = 1 \\ 0, \text{ if } x_{i,j}(s,a) = 0 \end{cases}$$

Solving the Mountain Car Problem

n-step SARSA for control

- The n-step bootstrapped gain with function approximation:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^{n} \hat{q}(S_{t+n}, A_{t+n}, w_{t+n-1})$$

- The weight update:

$$w_{t+n}(S_t, A_t) = w_{t+n-1} + \alpha \Big[G_{t:t+n} - \hat{q}(S_{t+n}, A_{t+n}, w_{t+n-1}) \Big] \nabla \hat{q}(S_t, A_t, w_{t+n-1})$$

Where:

$$\delta_t = G_{t:t+n} - \hat{q}(S_{t+n}, A_{t+n}, w_{t+n-1}) = \text{the n-step TD error}$$

$$\nabla \hat{q}(S_t, A_t, w_{t+n-1}) = \text{the gradient of the action value approximation}$$

Q-Learning and Function Approximation

What happens if Q-learning is used in a tabular case?

- Recall basic Q-learning update

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma\ max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

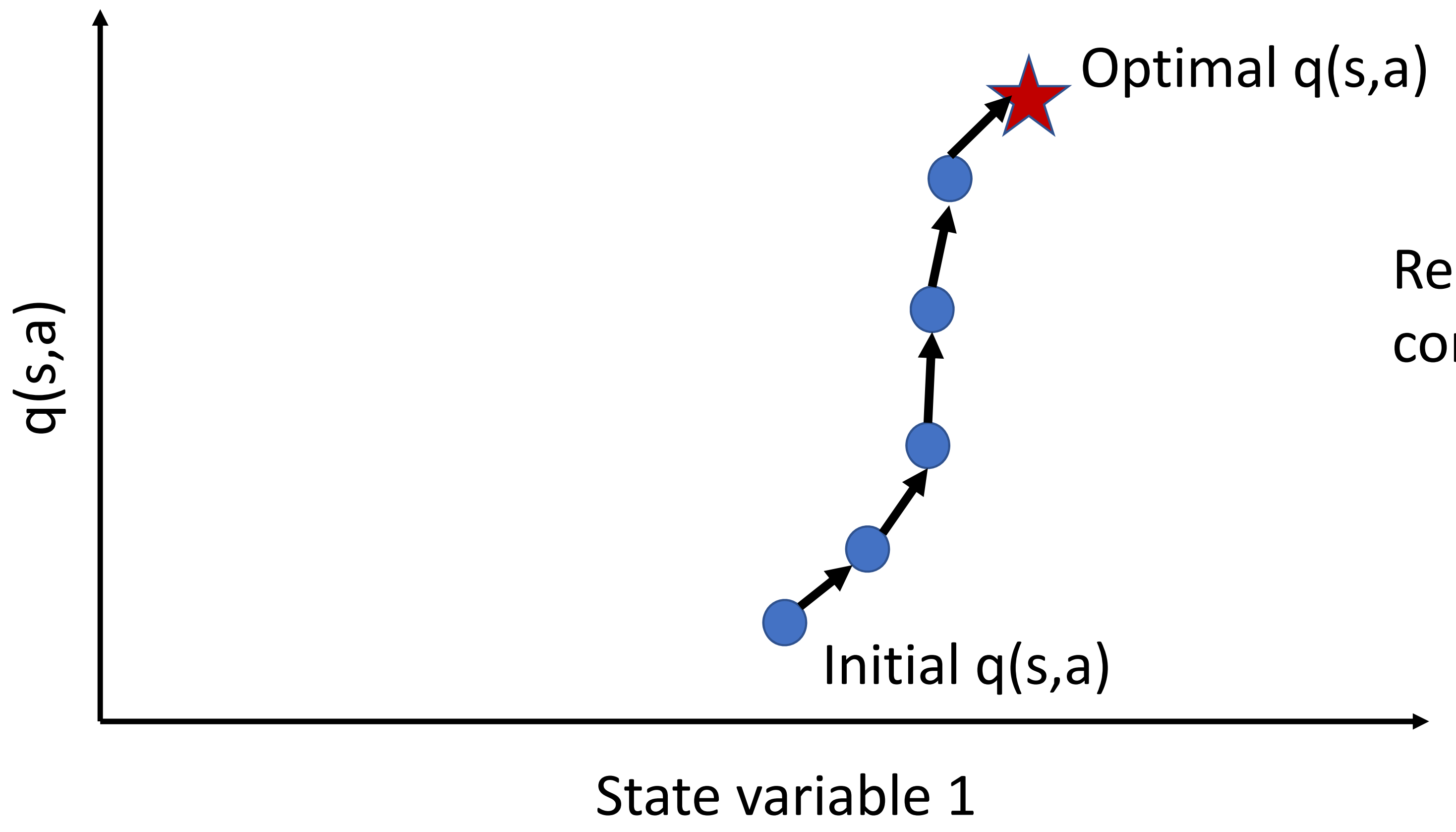
- The TD error is then,

$$\delta_t = R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)$$

- For tabular case, convergence as δ_t approaches 0

Q-Learning and Function Approximation

Q-learning converges in the tabular case – discrete state-actions



Reduce TD error until convergence

Q-Learning and Function Approximation

What happens if Q-learning is used with function approximation?

- The TD error uses an approximation of action-value function, $\hat{Q}(s_t, a_t, \mathbf{w}_t)$

$$\delta_t = G_t - \hat{Q}(s_t, a_t, \mathbf{w}_t)$$

- The gain depends on the nonlinear $\underset{a}{max}$ operator

$$G_t = R_{t+1} + \gamma \underset{a}{max}\, Q_\pi(s_{t+1}, a, \mathbf{w})$$

- But, the action-value function approximator minimizes least squares error

$$\overline{VE}(\mathbf{w}) = \sum_{s \in S} \mu(s) \Big[q_\pi(s, a) - \hat{q}(s, a, \mathbf{w}) \Big]^2$$

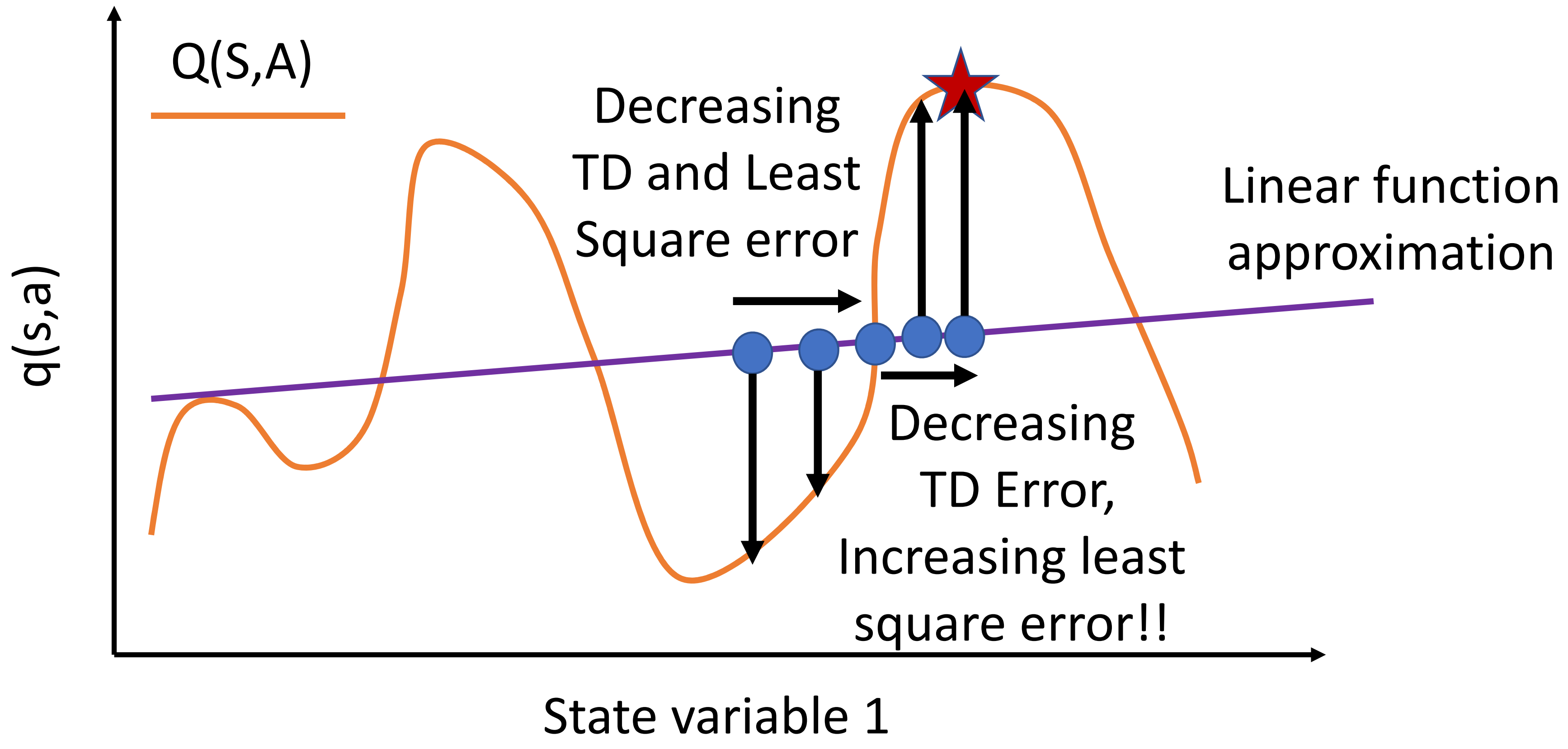
Q-Learning and Function Approximation

What happens if Q-learning is used with function approximation?

- Using TD error based on max operator with function approximation using least squares error leads to **instability and poor convergence**!
- Problem arises if three conditions are met:
    - **Function approximation**; $\hat{Q}(s_t, a_t, \mathbf{w}_t)$
    - **Off-policy algorithm**
    - **Bootstrapping** with an approximate action-value
    - The **deadly triad!**
- Monte Carlo control algorithms do not bootstrap and always converge, *eventually*

Q-Learning and Function Approximation

Q-learning will not converge with function approximation


Q(S,A)
Decreasing TD and Least Square error
Linear function approximation
q(s,a)
Decreasing TD Error, Increasing least square error!!
State variable 1

Q-Learning and Function Approximation

Convergence of control algorithms

Algorithm	Tabular	Linear	Nonlinear
Monte Carlo Control	✓	✓	✓
SARSA	✓	✓	✗
Q-learning	✓	✗	✗

DQN Algorithm

How can deep Q-Learning be applied to function approximation given the convergence problems?

- Use **two deep neural networks**
- The **online networks** updates the model weights as a regression problem
  - Not Q-learning
- The **target network** computes the bootstrap estimates, $Q_\pi(S, A, \mathbf{w}^-)$, using fixed weights ***w***$^-$
- Weights of the target network are updated periodically
- Indirect use of Q-learning algorithm!

DQN Algorithm

Deep Q Network as a function approximator

- The **DQN algorithm** learns the weights of $\hat{Q}(s_t, a_t, \mathbf{w}_t)$ with **online model** as a **regression problem**
- The regression estimator learns $\mathbf{w}_t$ minimizing loss function

$$J(\mathbf{w}_t) = \frac{1}{2} \parallel G_t - \hat{Q}(s_t, a_t, \mathbf{w}_t) \|^2$$

Where

$$G_t = R_{t+1} + \gamma \max_a Q_\pi(s_{t+1}, a, \mathbf{w})$$

- $\mathbf{w}_t$ is updated on each training epoch, as typical with deep neural networks

DQN Algorithm

Deep neural network as a function approximator for differentiable function

- The gradient descent weight update is:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\left[R_{t+1} + \gamma\, max\; Q_\pi(s_{t+1}, a_{t+1}) - \hat{Q}(s_t, a_t, \mathbf{w}_t)\right]\nabla_w \hat{Q}(s_t, a_t, \mathbf{w}_t)$$

- With semi-gradient $\nabla_w \hat{Q}(S_t, A_t, \mathbf{w}_t) = \begin{bmatrix} \frac{\partial \hat{Q}(S_t, A_t, \mathbf{w}_t)}{\partial w_1} \\ \frac{\partial \hat{Q}(S_t, A_t, \mathbf{w}_t)}{\partial w_2} \\ \vdots \\ \frac{\partial \hat{Q}(S_t, A_t, \mathbf{w}_t)}{\partial w_d} \end{bmatrix}$

DQN Algorithm

Deep neural network as a function approximator

- The regression estimator learns **w**$_t$ minimizing loss function

$$J(\mathbf{w}_t) = \frac{1}{2} \parallel G_t - \hat{Q}(s_t, a_t, \mathbf{w}_t)\|^2$$

- The gain is: $G_t^{DQN} = R_{t+1} + \gamma \, max \, Q_\pi(s_{t+1}, a_{t+1}, \mathbf{w}_t^-)$
- $Q_\pi(s_{t+1}, a_{t+1}, \mathbf{w}_t^-)$ is computed with the **target model** with **fixed weights, $\mathbf{w}_t^-$**
- Weights, $\mathbf{w}_t^-$, must be frozen so gain bootstrap estimate, G_t^{DQN}, is stable
- Every T epochs the target model weights are updated, $\mathbf{w} \rightarrow \mathbf{w}^-$

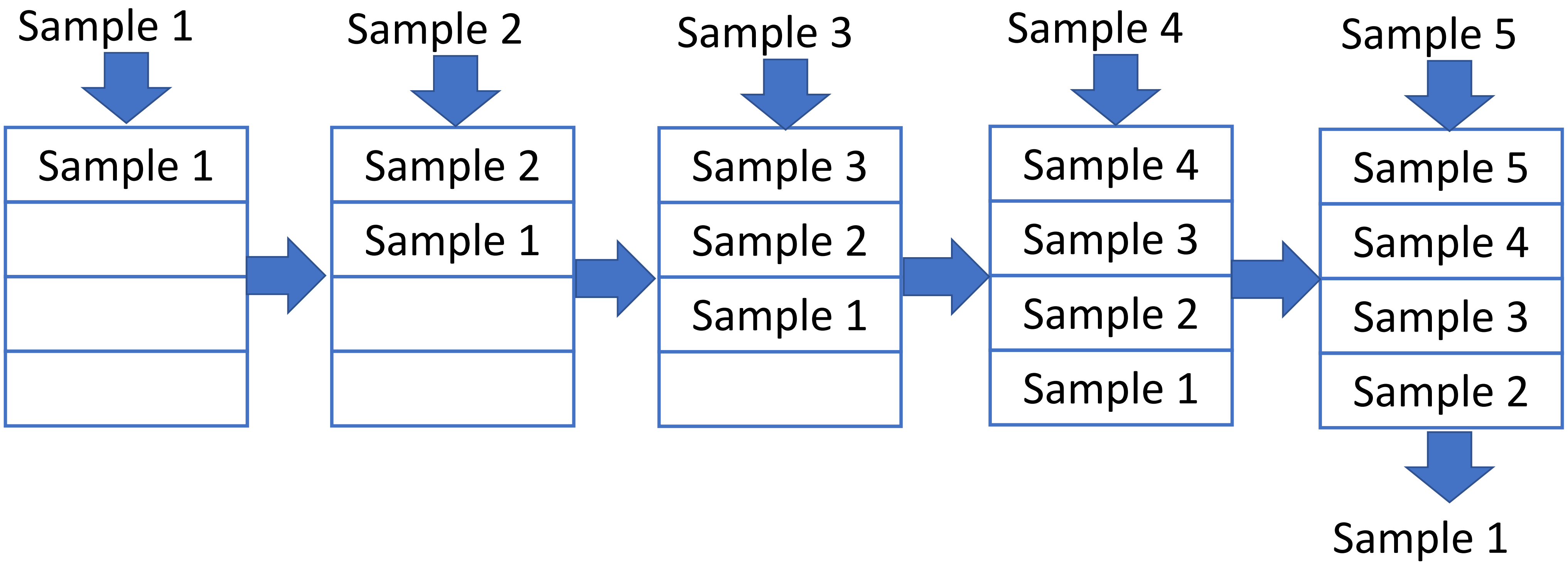
Data Replay

The Replay Buffer

- Deep neural networks **trained with stochastic gradient descent**
- Gradient descent requires **mini-batch samples** from data
- How is this done?
- Use **replay buffer**

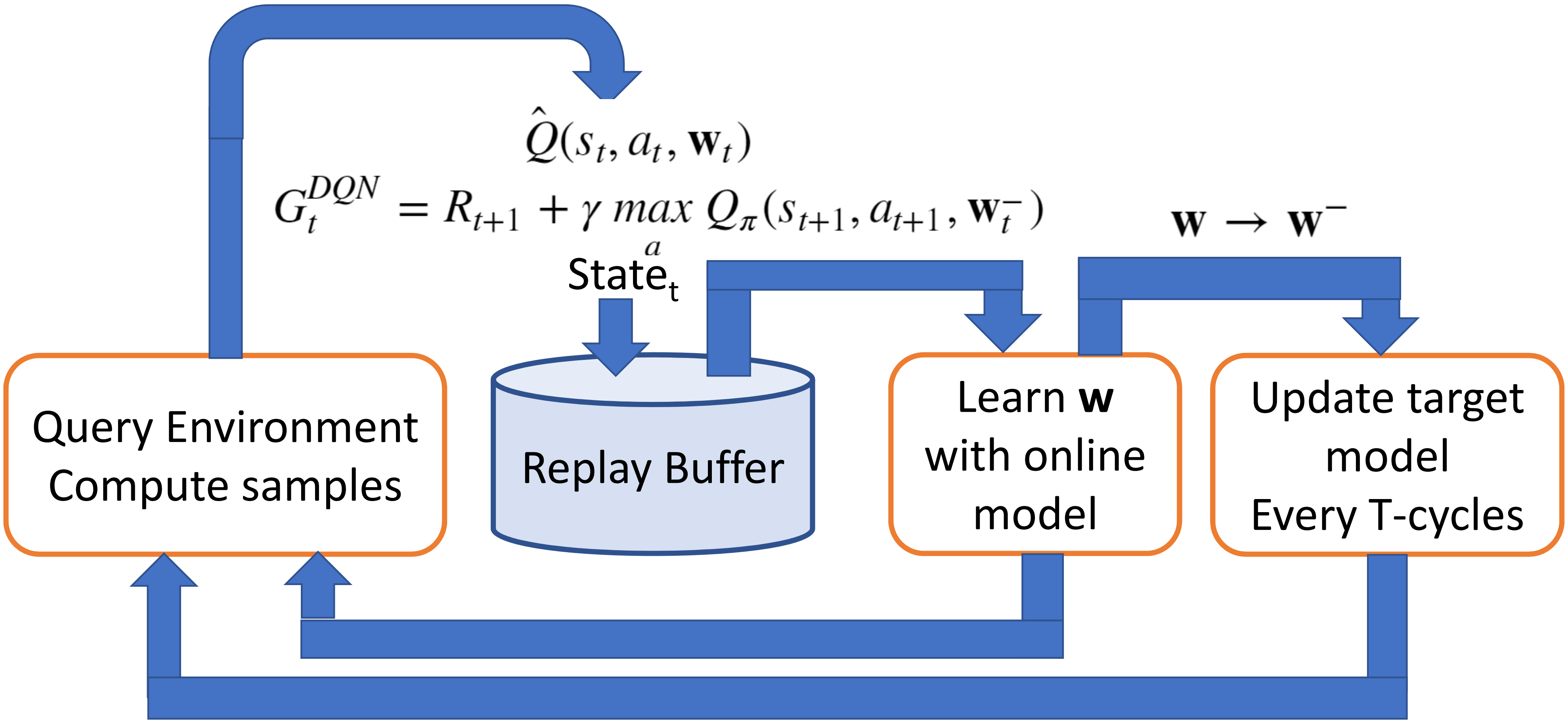
Data Replay

The Replay Buffer



DQN Algorithm

Putting everything together



$\hat{Q}(s_t, a_t, \mathbf{w}_t)$
$G_t^{DQN} = R_{t+1} + \gamma \max_a Q_\pi(s_{t+1}, a_{t+1}, \mathbf{w}_t^-)$
$\mathbf{w} \rightarrow \mathbf{w}^-$
State$_t$
Query Environment
Compute samples
Replay Buffer
Learn **w** with online model
Update target model
Every T-cycles

Double DQN Algorithm

- How to deal with the **bias in the DQN algorithm?**
- The **Double DQN algorithm** eliminates the bias
- The DDQN algorithm uses **two online models**
  1. Initially one model acts as the **online model** and the other as the **target model**
  2. Samples are added to the same **replay buffer**
  3. The **online model is updated**
  4. The **roles of the models are switched** and return to step 1
- Alternating models and sampling eliminates bias

Key Improvements in DQN

- Considerable research is improving the **sample efficiency** of DQN algorithms
- Three key improvements for DQN:

  Prioritized Experience Relay, Schaul, et. al., 2016

  Dueling Network Architectures for Deep Reinforcement Learning, Wang et. al., 2016

  Noisy Networks for Exploration, Fortunato, et. al., 2018

Prioritized Replay

- For DQN and DDQN is **uniform probability of sampling** (learning) from cases in replay buffer
- Uniform sampling can lead to slow learning
  - Samples with little information sampled
  - Samples with high information may not be sampled
- Prioritized replay buffer implements importance sampling
  - Samples with high information more likely to be sampled

Prioritized Replay

Importance sampling for prioritized replay

- Unfortunately no direct way to measure importance or information
- Use **absolute value of TD error as proxy:**

$$p_i = |\delta_t| = |R_{t+1} + \gamma \max_a Q_\pi(s_{t+1}, a_{t+1}) - \hat{Q}(s_t, a_t, \mathbf{w}_t)|$$

- This importance measure is both **deterministic** and **subject to noise**

Prioritized Replay

Importance sampling for prioritized replay

- Use **absolute value of TD error as proxy:**

$$p_i = |\delta_t| = |R_{t+1} + \gamma \max_a Q_\pi(s_{t+1}, a_{t+1}) - \hat{Q}(s_t, a_t, \mathbf{w}_t)|$$

- Use probability, P(i):

$$P(i) = \frac{p_i^a}{\sum_k p_k^a}$$

- Exponent, α, determines degree of prioritization
  - $\alpha = 0$, sampling is uniform
  - $\alpha = 1$, sampling sensitive to TD error and P(i) distribution is softmax

Prioritized Replay

Importance sampling for prioritized replay

- Importance sampling introduces a biased estimate of the TD errors
  - Cases with larger error more likely to be sampled
- Use **normalized importance sample weighting**:

$$w_i = \frac{1}{max_i(w_i)} \left(\frac{1}{N\ P(i)} \right)^{\beta}$$

- With normalization, $\frac{1}{max_i(w_i)}$
- The adjusted TD error is then, $w_i \delta_i$
- If exponent β = 1, bias is fully adjusted