

CS-866 Deep Reinforcement Learning

Deep Value-Based Agents



Nazar Khan
Department of Computer Science
University of the Punjab

From Tabular to Deep Agents

- ▶ So far, we have studied **tabular methods** such as
 - ▶ Monte Carlo Sampling
 - ▶ SARSA
 - ▶ Q-learning
- ▶ These methods work well for **small, discrete environments**.
- ▶ But what happens when the state space becomes huge or continuous?

Main Challenge

Create an **agent algorithm** that can learn a good policy *by interacting with the world*, even in large, high-dimensional environments.

The Next Step: Deep Reinforcement Learning

- ▶ From now on, our agents will be **deep learning agents**.
- ▶ We combine:

Reinforcement Learning (decision making)

+ Deep Learning (function approximation)

⇒ Deep Reinforcement Learning (DRL)

- ▶ DRL allows us to handle:
 - ▶ Large or continuous state spaces
 - ▶ Complex perceptual inputs (e.g., images, audio)
 - ▶ High-dimensional control problems
-

Motivation: Beyond Toy Problems

- ▶ Simple grid worlds or taxi environments are **toy problems**.
- ▶ Real-world domains:
 - ▶ Robotics
 - ▶ Autonomous driving
 - ▶ Game playing (e.g., Go, Chess, Atari)
 - ▶ Financial decision-making
- ▶ These involve thousands or millions of states and actions.

Key Question

How can we scale from a small tabular $Q(s, a)$ to a powerful **neural-network-based** $Q_{\theta}(s, a)$?

From Tables to Parameterized Functions

Tabular Methods

- ▶ Store $Q(s, a)$ in a lookup table
- ▶ Feasible for small, discrete spaces
- ▶ Example: Taxi world with 500 states

Deep Methods

- ▶ Use a neural network $Q_\theta(s, a)$
- ▶ θ are trainable weights
- ▶ Approximates the value function for unseen states

Goal

Transform:

$$V, Q, \pi \rightarrow V_\theta, Q_\theta, \pi_\theta$$

so that our agent can generalize to **large or continuous spaces**.

Core Questions in Deep Value-Based RL

- ▶ How can we use deep learning for large-scale sequential decision-making?
 - ▶ How do we represent the value or policy functions with neural networks?
 - ▶ How can we train these networks stably and efficiently?
 - ▶ What challenges arise from using non-linear function approximators?
-

From Supervised to Reinforcement Learning

- ▶ **Deep supervised learning**¹ uses a *static dataset* $\{(x_i, y_i)\}$.
- ▶ The goal is to approximate a function $f_\theta(x)$ such that:

$$\theta^* = \arg \min_{\theta} \mathcal{L}(f_\theta(x_i), y_i)$$

where $\mathcal{L}$ is a loss function.

- ▶ The labels y_i are **fixed, static targets**.
- ▶ Learning proceeds via gradient descent until the loss converges.

Key Idea

Supervised learning optimizes against **known, static truths**.

¹See Appendix B of Plaat's book

Bootstrapping as a Minimization Process

- ▶ In reinforcement learning, **bootstrapping** plays a similar role.
- ▶ The agent learns from the difference between successive estimates:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

- ▶ This **temporal difference (TD) error** acts like a loss term that we try to minimize.
- ▶ Over time, this process converges to the true value functions:

$$V^*(s), \quad Q^*(s, a)$$

Analogy

Supervised learning: minimize $(y - \hat{y})^2$

Reinforcement learning: minimize $(r + \gamma V(s') - V(s))^2$

The Challenge: Moving Targets

- ▶ In Q-learning, the data samples are **not static**.
- ▶ The agent's actions generate new experience tuples:

$$(s_t, a_t, r_{t+1}, s_{t+1})$$

- ▶ The target in the loss function,

$$y_t = r_{t+1} + \gamma \max_a Q_\theta(s_{t+1}, a),$$

changes as the network parameters θ change!

- ▶ Hence, the **target itself moves** during training.

Consequence

The Q-network tries to predict targets that depend on itself — this creates potential instability and divergence.

Dynamic Data and Policy Coupling

- ▶ Unlike supervised learning, RL samples depend on the current policy π_θ .
- ▶ Thus, as π_θ improves, the distribution of states and rewards changes.
- ▶ The agent is both:
 - ▶ The **generator** of its own data, and
 - ▶ The **learner** from that data.
- ▶ This **circular dependency** makes convergence hard.

Moving Target Problem

$$y_t(\theta) = r_{t+1} + \gamma \max_a Q_\theta(s_{t+1}, a)$$

depends on θ itself — our target moves every time we update the network.

Stability: The Central Challenge of Deep RL

- ▶ Finding stable learning algorithms for moving targets took years of research.
- ▶ The key innovations:
 - ▶ **Experience Replay:** breaks correlation between consecutive samples.
 - ▶ **Target Networks:** stabilize the moving target by freezing parameters temporarily.
 - ▶ **Careful learning rate tuning.**

Why It's Hard

The optimization landscape is **non-stationary** and **self-referential**.

Connecting the Three Worlds

Aspect	Supervised Learning	Tabular Q-learning	Deep Q-learning
Data source	Static dataset	Sampled from environment	Sampled, stored, replayed
Targets	Fixed labels	Bootstrapped rewards	Moving bootstrapped rewards
Function type	$f_{\theta}(x)$	Table $Q(s, a)$	Network $Q_{\theta}(s, a)$
Loss	$\mathcal{L}(y, \hat{y})$	TD Error	TD Error with moving target
Stability	High	Moderate	Low (requires tricks)

Summary

Deep Q-learning generalizes Q-learning to large problems by using deep networks, but inherits instability from its **bootstrapped, moving-target** nature.

Overview of Supervised Training

- ▶ In supervised deep learning, training minimizes a loss function between predicted outputs and known targets.
 - ▶ The dataset is **static** — both inputs and labels remain fixed throughout training.
 - ▶ The optimization aims to approximate a target function through repeated forward and backward passes.
-

Typical Supervised Learning Algorithm

```
def train_sl(data, net, alpha=0.001):           # train classifier
    for epoch in range(max_epochs):             # an epoch is one
        pass
        sum_sq = 0                             # reset to zero for each
        pass
        for (image, label) in data:
            output = net.forward_pass(image) # predict
            sum_sq += (output - label)**2 # compute error
        grad = net.gradient(sum_sq)           # derivative of error
        net.backward_pass(grad, alpha)        # adjust weights
    return net
```

Typical Supervised Learning Algorithm

- ▶ The main components are:
 1. **Input dataset:** static pairs of inputs and target labels.
 2. **Forward pass:** compute network predictions.
 3. **Loss computation:** measure prediction error.
 4. **Backward pass:** compute gradients and update parameters.

Goal: minimize the loss function over the dataset to find parameters that best fit the data.

Structure of the Training Loops

- ▶ Training typically involves a **double loop**:
 1. **Outer loop**: controls the number of epochs.
 2. **Inner loop**: iterates through each example (or minibatch) of the dataset.
 - ▶ In each epoch:
 - ▶ Perform a forward approximation using current parameters.
 - ▶ Compute the loss and its gradient.
 - ▶ Adjust the parameters via backpropagation.
-

Inner Loop Dynamics

- ▶ The **inner loop** provides samples to the forward computation of:
 - ▶ Output value
 - ▶ Loss value
 - ▶ Gradient computation
 - ▶ The **backward pass** then adjusts the parameters accordingly.
 - ▶ The dataset is static, so each epoch processes the same data repeatedly until convergence.
-

Independence and Sampling of Data

- ▶ Each training sample is assumed to be **independent** of the others.
 - ▶ Samples are typically selected with **equal probability**.
 - ▶ For example:
 - ▶ After an image of a white horse is sampled,
 - ▶ The probability that the next image is a black grouse or a blue moon remains equally (un)likely.
 - ▶ This independence ensures that learning is based purely on the data distribution, not temporal correlations.
-

Key Characteristics of Supervised Training

- ▶ Static dataset $\Rightarrow$ fixed ground-truth targets.
 - ▶ Independent samples $\Rightarrow$ no temporal dependencies.
 - ▶ Objective $\Rightarrow$ minimize loss over dataset to achieve best approximation of the target function.
 - ▶ The learning process is stable because:
 - ▶ Targets do not change during training.
 - ▶ Gradients are computed against fixed labels.
-

Introduction to Bootstrapping Q-Values

- ▶ Q-learning is a foundational reinforcement learning (RL) algorithm.
- ▶ Unlike supervised learning, RL chooses its training examples **dynamically** through interaction with the environment.
- ▶ The algorithm learns by **bootstrapping** — updating estimates based partly on other learned estimates.
- ▶ For convergence, every state must eventually be sampled by the environment².

²Christopher JCH Watkins. 'Learning from Delayed Rewards'. PhD thesis. King's College, Cambridge, 1989.

Challenges of Large State Spaces

- ▶ In small environments, it is feasible for all states to be visited repeatedly.
 - ▶ However, for large or continuous state spaces:
 - ▶ This condition no longer holds.
 - ▶ Many states may never be visited.
 - ▶ Therefore, convergence to the true value function is not guaranteed.
 - ▶ This motivates the use of **function approximation** (e.g., deep networks) in modern RL.
-

Structure of the Q-Learning Algorithm³

- ▶ As with supervised training, Q-learning consists of a **double loop**:
 - 1. Outer loop:** Controls the number of episodes.
 - 2. Inner loop:** Iterates through steps within an episode.
- ▶ Each episode represents a trajectory from a start state to a terminal state.

```
def qlearn(environment, alpha=0.001, gamma=0.9, epsilon=0.05):  
    Q[TERMINAL, _] = 0 # policy  
    for episode in range(max_episodes):  
        s = s0  
        while s not TERMINAL: # perform steps of one full  
            episode  
            a = epsilongreedy(Q[s], epsilon)  
            (r, sp) = environment(s, a)  
            Q[s,a] = Q[s,a] + alpha*(r+gamma*max(Q[sp]) - Q[s,a])  
            s = sp  
    return Q
```

³Christopher JCH Watkins. 'Learning from Delayed Rewards'. PhD thesis. King's College, Cambridge, 1989.

Representation and Convergence

- ▶ Q-values are stored in a **Python array** indexed by (s, a) pairs:

$$Q[s, a]$$

- ▶ Q-function represents the expected return for taking action a in state s .
- ▶ Convergence is assumed to occur when enough episodes have been sampled.
- ▶ The update rule is **bootstrapped** from prior estimates:

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

Off-Policy Learning in Q-Learning

- ▶ Q-learning is an **off-policy** method.
 - ▶ The update target uses the **maximum future Q-value**, not the value of the action taken by the current policy.
 - ▶ This enables learning the optimal policy while following an exploratory behavior policy (e.g., ϵ -greedy).
 - ▶ Hence, even though behavior is stochastic, the learned Q-values reflect an optimal deterministic policy.
-

Differences from Supervised Learning

- ▶ In Q-learning:
 - ▶ Samples are **not independent**.
 - ▶ Each next action **depends** on the current policy.
 - ▶ Successive states are **highly correlated** in a trajectory.
 - ▶ Example:
 - ▶ If the agent samples a state where a ball is in the upper left corner,
 - ▶ The next state will likely also be near the upper left corner.
 - ▶ This temporal correlation violates the i.i.d. assumption of supervised learning.
-

Consequences of Sample Dependence

- ▶ Correlated samples can cause:
 - ▶ **Slow learning** due to redundant experiences.
 - ▶ **Instability** or divergence in training.
 - ▶ The network may overfit to local regions of the state space.
 - ▶ To mitigate this, RL uses:
 - ▶ **Exploration strategies** (ϵ -greedy, softmax, etc.).
 - ▶ **Experience replay** to decorrelate samples.
-

Need for Exploration

- ▶ Without sufficient exploration:
 - ▶ The agent may become trapped in local optima.
 - ▶ It may fail to discover high-reward states.
 - ▶ Exploration ensures coverage of diverse states, improving Q-value estimation.
 - ▶ Common strategies include:
 - ▶ ϵ -greedy policy: random action with probability ϵ .
 - ▶ Decaying ϵ : reduces randomness over time.
-

Summary of Bootstrapping in Q-Learning

- ▶ Q-learning updates are based on **bootstrapping previous estimates**.
 - ▶ The process is **dynamic**: training samples and targets change during learning.
 - ▶ Q-learning differs from supervised learning because:
 - ▶ Samples are temporally correlated.
 - ▶ Targets depend on current estimates (no static ground truth).
 - ▶ Learning is off-policy and depends on exploration.
 - ▶ These factors make convergence difficult, especially for large problems.
-

Deep Reinforcement Learning Target-Error

- ▶ Deep learning and Q-learning share a striking structural similarity.
 - ▶ Both algorithms consist of a **double loop**:
 - ▶ An **outer loop** over epochs or episodes.
 - ▶ An **inner loop** over samples or steps.
 - ▶ Each iteration minimizes an error or difference between a prediction and a target.
 - ▶ This similarity raises the question: **Can bootstrapping be combined with loss-function minimization?**
-

Combining Bootstrapping and Gradient Descent

- ▶ Mnih et al.⁴ demonstrated that the two processes **can indeed be combined**.
- ▶ The result is **Deep Q-Learning (DQN)** — a method that merges:
 - ▶ Bootstrapping from Q-learning, and
 - ▶ Gradient-based parameter optimization from deep learning.
- ▶ The key idea: train a **Q-network** that approximates $Q_{\theta}(s, a)$ using backpropagation.

⁴Volodymyr Mnih et al. 'Playing Atari with Deep Reinforcement Learning'. In: (2013).
cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013. URL:
<http://arxiv.org/abs/1312.5602>.

Naive Deep Learning Version of Q-Learning

- ▶ The structure is still a double loop:
 1. Outer loop: controls episodes or training iterations.
 2. Inner loop: bootstraps Q-values by minimizing a loss function.
 - ▶ The parameters θ of the Q-network are updated via **stochastic gradient descent**.
-

Naive Deep Learning Version of Q-Learning

```
def train_qlearn(environment, Qnet, alpha=0.001, gamma=0.0,
epsilon=0.05
    s = s0                # initialize start state
    for epoch in range(max_epochs): # an epoch is one pass
        sum_sq = 0         # reset to zero for each pass
        while s not TERMINAL: # perform steps of one full
            episode
                a = epsilon_greedy(Qnet(s,a)) # net: Q[s,a]-values
                (r, sp) = environment(a)
                output = Qnet.forward_pass(s, a)
                target = r + gamma * max(Qnet(sp))
                sum_sq += (target - output)**2
                s = sp
        grad = Qnet.gradient(sum_sq)
        Qnet.backward_pass(grad, alpha)
    return Qnet           # Q-values
```

The Deep Q-Learning Loss Function

- ▶ Deep Q-learning minimizes a loss based directly on the Q-learning update rule.
- ▶ The loss at iteration t is:

$$\mathcal{L}(\theta_t) = \mathbb{E}_{s,a,r,s'} \left[\left(r + \gamma \max_{a'} Q_{\theta_{t-1}}(s', a') - Q_{\theta_t}(s, a) \right)^2 \right]$$

- ▶ This is the squared difference between:
 - ▶ The new Q-value $Q_{\theta_t}(s, a)$ (forward pass), and
 - ▶ The old bootstrapped target $r + \gamma \max_{a'} Q_{\theta_{t-1}}(s', a')$.
-

Gradient of the Deep Q-Learning Loss

- ▶ The gradient for the parameter update is given by:

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

- ▶ Here:
 - ▶ ρ : behavior distribution (policy used for exploration)
 - ▶ $\mathcal{E}$: environment dynamics (e.g., Atari emulator)
- ▶ This defines a **fixed-point iteration** process⁵.

⁵Francisco S Melo and M Isabel Ribeiro. 'Convergence of Q-learning with linear function approximation'. In: *2007 European Control Conference (ECC)*. IEEE. 2007, pp. 2671–2678.

Moving Targets and Instability

- ▶ A crucial distinction from supervised learning:
 - ▶ In supervised learning, targets are **fixed**.
 - ▶ In deep reinforcement learning, targets are **moving**.
- ▶ The target values depend on the previous parameters θ_{t-1} :

$$y_t = r + \gamma \max_{a'} Q_{\theta_{t-1}}(s', a')$$

- ▶ Since both prediction and target evolve as learning progresses, **the optimization target moves during training**.
-

Implications of Moving Targets

- ▶ Moving targets can lead to:
 - ▶ **Instability**: network weights chase a shifting objective.
 - ▶ **Divergence**: updates may amplify errors instead of reducing them.
 - ▶ **Non-stationary training signals**.
 - ▶ To mitigate these problems, DQN introduced:
 - ▶ **Target networks** — to stabilize the bootstrapped targets.
 - ▶ **Experience replay** — to decorrelate samples.
 - ▶ These innovations made deep reinforcement learning feasible for large-scale tasks such as **Atari games**.
-

Summary: Deep RL Target-Error Concept

- ▶ Deep RL integrates:
 1. **Bootstrapping** from temporal-difference learning.
 2. **Loss minimization** through gradient descent.
 - ▶ The target depends on older network weights, making it a **moving target**.
 - ▶ Despite this challenge, stability can be achieved with architectural innovations (target networks, replay buffers).
 - ▶ Deep Q-learning thus bridges the gap between tabular Q-learning and deep neural function approximation.
-

Three Core Challenges

- ▶ Our naive deep Q-learner faces three fundamental problems:
 1. **Coverage:** The state space is too large to sample fully.
 2. **Correlation:** Subsequent samples are highly correlated.
 3. **Convergence:** The optimization target moves during learning.
 - ▶ These issues threaten convergence, stability, and generalization of deep RL agents.
-

Challenge 1: Coverage

- ▶ Proofs of Q-learning's convergence rely on a key assumption:

All state-action pairs (s, a) must eventually be sampled.

- ▶ This ensures that $Q(s, a)$ converges to the optimal $Q^*(s, a)$.
 - ▶ However, in large or continuous environments:
 - ▶ Full state coverage is **impossible**.
 - ▶ Many states may never be visited.
 - ▶ $\Rightarrow$ No theoretical guarantee of convergence to the optimal policy.
-

Coverage in Practice

- ▶ Example: Atari game with millions of unique screen states.
 - ▶ Even after millions of steps, the agent may have visited only a fraction.
 - ▶ Consequently:
 - ▶ Q-values for unseen states remain inaccurate.
 - ▶ Policy may fail catastrophically in novel or rare situations.
 - ▶ This is a form of **out-of-distribution generalization** failure.
-

Challenge 2: Correlation

- ▶ In reinforcement learning, samples are **not independent**.
- ▶ Each state s_{t+1} is generated from s_t by one action:

$$s_{t+1} = f(s_t, a_t)$$

- ▶ Hence, consecutive samples (s_t, a_t, r_t, s_{t+1}) are **highly correlated**.
 - ▶ This violates the i.i.d. assumption of stochastic gradient descent.
-

Consequences of Sample Correlation

- ▶ Correlated samples can cause:
 - ▶ **Biased training**: updates reflect a narrow part of the state space.
 - ▶ **Local minima**: the policy becomes specialized to a small region.
 - ▶ **Feedback loops**: policy reinforces its own biases.
 - ▶ Example:
 - ▶ A chess agent always plays one opening.
 - ▶ It learns strong Q-values only for that opening.
 - ▶ When the opponent plays a different opening performance collapses.
-

The Specialization Trap

- ▶ When exploitation dominates exploration:
 - ▶ The agent repeatedly selects the same actions.
 - ▶ State trajectories become repetitive.
 - ▶ The agent gets stuck in a “specialization trap.”
 - ▶ This worsens both:
 - ▶ **Coverage**: fewer distinct states sampled.
 - ▶ **Convergence**: biased gradients lead to overfitting.
 - ▶ The result: poor generalization and unstable learning.
-

Challenge 3: Convergence

- ▶ In supervised learning:
 - ▶ Targets y are **fixed**.
 - ▶ Loss $\mathcal{L}(\theta) = (y - f_{\theta}(x))^2$ minimizes toward a stable solution.
- ▶ In deep reinforcement learning:
 - ▶ Targets **move** because they depend on θ_{t-1} .
 - ▶ Bootstrapped target:

$$y_t = r + \gamma \max_{a'} Q_{\theta_{t-1}}(s', a')$$

Moving Targets and Instability

- ▶ The loss at time t is:

$$\mathcal{L}(\theta_t) = \left(r + \gamma \max_{a'} Q_{\theta_{t-1}}(s', a') - Q_{\theta_t}(s, a) \right)^2$$

- ▶ Both prediction and target depend on parameters being optimized.
 - ▶ $\Rightarrow$ Risk of:
 - ▶ **Overshooting** the target.
 - ▶ **Oscillation** or even **divergence**.
 - ▶ Gradient descent “chases” a target that moves with every update.
-

Why Convergence is Difficult

- ▶ Reinforcement learning optimizes a function that depends on itself:

$$Q_{\theta}(s, a) \approx r + \gamma \max_{a'} Q_{\theta}(s', a')$$

- ▶ This circular dependency causes:
 - ▶ **Non-stationary targets**
 - ▶ **Instability** in gradient-based updates
 - ▶ Considerable research effort has gone into finding algorithms that:
 - ▶ Break this circular dependency,
 - ▶ And stabilize learning despite moving targets.
-

Summary: The Three Challenges

1. Coverage

Large state spaces prevent full sampling $\Rightarrow$ incomplete Q-values.

2. Correlation

Sequential samples are correlated $\Rightarrow$ biased updates and specialization traps.

3. Convergence

Targets move with parameters $\Rightarrow$ instability and potential divergence.

- Overcoming these challenges led to key innovations: **Experience Replay** and **Target Networks**.
-

The Deadly Triad: Overview

- ▶ Combining **off-policy learning** with **nonlinear function approximation** can cause Q-values to diverge.^{6,7,8}
- ▶ Three interacting elements make reinforcement learning unstable:
 1. Function approximation
 2. Bootstrapping
 3. Off-policy learning
- ▶ Together, they form the **Deadly Triad**⁹.

⁶Leemon Baird. 'Residual algorithms: Reinforcement learning with function approximation'. In: *Machine Learning Proceedings 1995*. Elsevier, 1995, pp. 30–37.

⁷Geoffrey J Gordon. *Approximate solutions to Markov decision processes*. Carnegie Mellon University, 1999.

⁸John N Tsitsiklis and Benjamin Van Roy. 'Analysis of temporal-difference learning with function approximation'. In: *Advances in Neural Information Processing Systems*. 1997, pp. 1075–1081.

⁹Richard S Sutton and Andrew G Barto. *Reinforcement learning, An Introduction, Second Edition*. MIT Press, 2018.

Function Approximation

- ▶ Function approximators (e.g., neural networks) estimate $Q(s, a)$ using shared features between states.
- ▶ Unlike exact tabular methods, deep networks generalize over state features:

$$Q(s, a) \approx f_{\theta}(\phi(s), a)$$

- ▶ Errors in shared features can cause **misidentification of states**.
- ▶ Reward values or Q-values can then be attributed incorrectly to unrelated states.

Implication

Misassigned values can cause **instability or divergence** during learning.

Bootstrapping

- ▶ In temporal-difference and Q-learning, current estimates depend on previous estimates:

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

- ▶ Bootstrapping speeds up training since values need not be computed from scratch.
- ▶ However, errors in early estimates can **propagate and amplify**.
- ▶ With function approximation, these errors can affect multiple states that share features.

Key Issue

Bootstrapping + Function Approximation $\Rightarrow$ Persistent and spreading errors.

Off-Policy Learning

- ▶ Off-policy methods (e.g., Q-learning) learn from a **behavior policy** π_b that differs from the **target policy** π .
- ▶ The learning update uses:

$$Q^\pi(s, a) = \mathbb{E} \left[r + \gamma \max_{a'} Q^\pi(s', a') \right]$$

- ▶ The policy used for exploration may not generate data representative of the optimal policy's state distribution.
- ▶ This can cause poor convergence or divergence, especially when combined with function approximation.

Observation

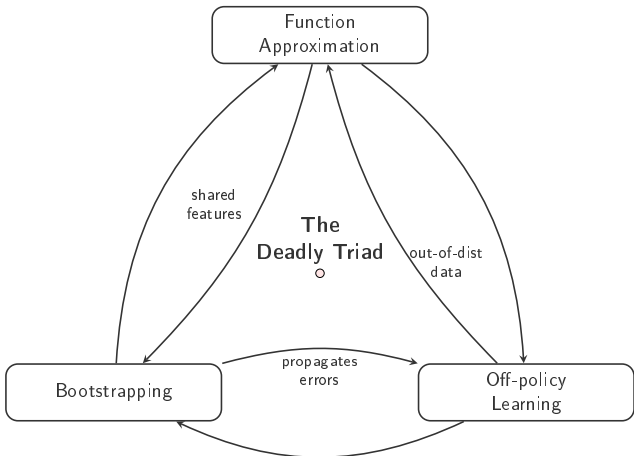
Off-policy learning is less stable than on-policy learning, and stability worsens with nonlinear function approximators.

Interaction of the Triad

- ▶ Each element of the triad alone can cause instability.
- ▶ Together, they can result in:
 - ▶ Divergent Q-values
 - ▶ Oscillatory learning
 - ▶ Poor convergence
- ▶ Example:

function approx.+bootstrapping+off-policy data $\Rightarrow$ divergence

Illustration of the deadly triad interaction.



Risks: Divergence / Oscillations / Instability

Mitigations: Experience Replay, Target Networks, Double Q, On-policy methods

Avoiding the Deadly Triad

- ▶ Several techniques have been developed to mitigate instability:
 - ▶ Experience Replay (reduces correlation)
 - ▶ Target Networks (stabilize bootstrapping)
 - ▶ On-policy algorithms (e.g., SARSA, A3C)
 - ▶ Double Q-learning (reduces overestimation bias)
- ▶ Stable deep RL became possible with these methods¹⁰.

Key Idea

Breaking at least one link in the triad (approximation, bootstrapping, or off-policy) helps achieve convergence.

¹⁰Volodymyr Mnih et al. 'Human-level control through deep reinforcement learning'. In: *Nature* 518.7540 (2015), pp. 529–533.