## horizontal line

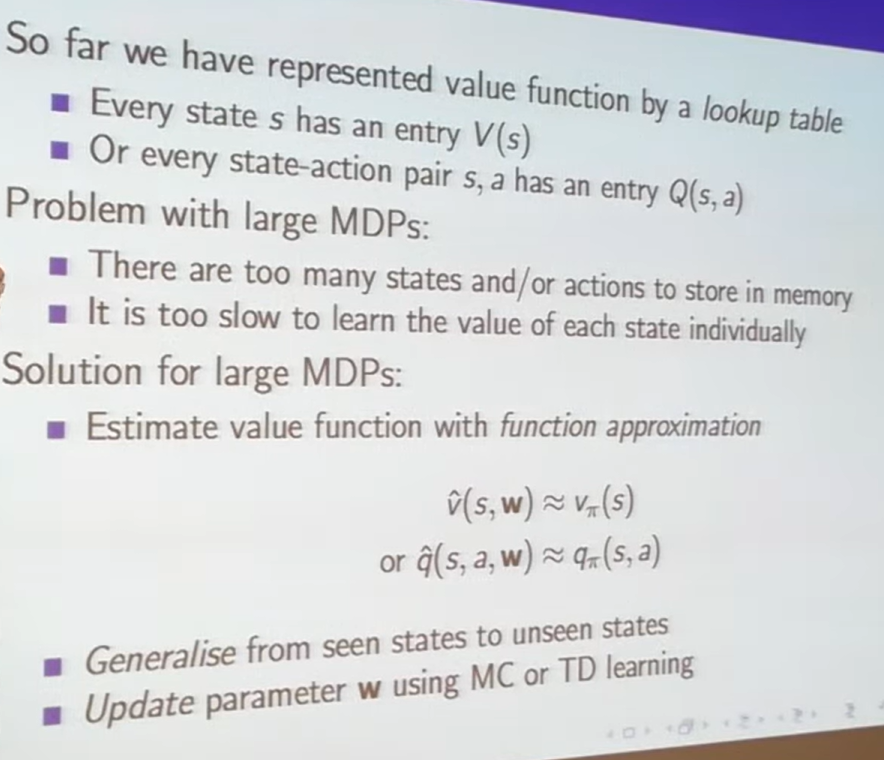
Value Function Approximation and AVFA

29.10.2025

**Value function approximation is a technique in reinforcement learning that replaces large or infinite value tables with parameterized functions, enabling agents to generalize across similar states and actions. It’s essential for scaling RL to complex or continuous environments.**

🧠 Why Use Value Function Approximation?

In simple environments, we can store value functions V(s) or Q(s, a) in a table. But in real-world tasks:

* **State/action spaces are huge or continuous** (e.g., robotics, games, telecom scheduling)
* **Tabular methods become infeasible** due to memory and sample inefficiency

**Function approximation** solves this by learning a mapping where w are parameters of the approximator (e.g., weights in a neural network).

# Types of Function Approximators :

# Value Function Approximation by Stochastic gradient descent

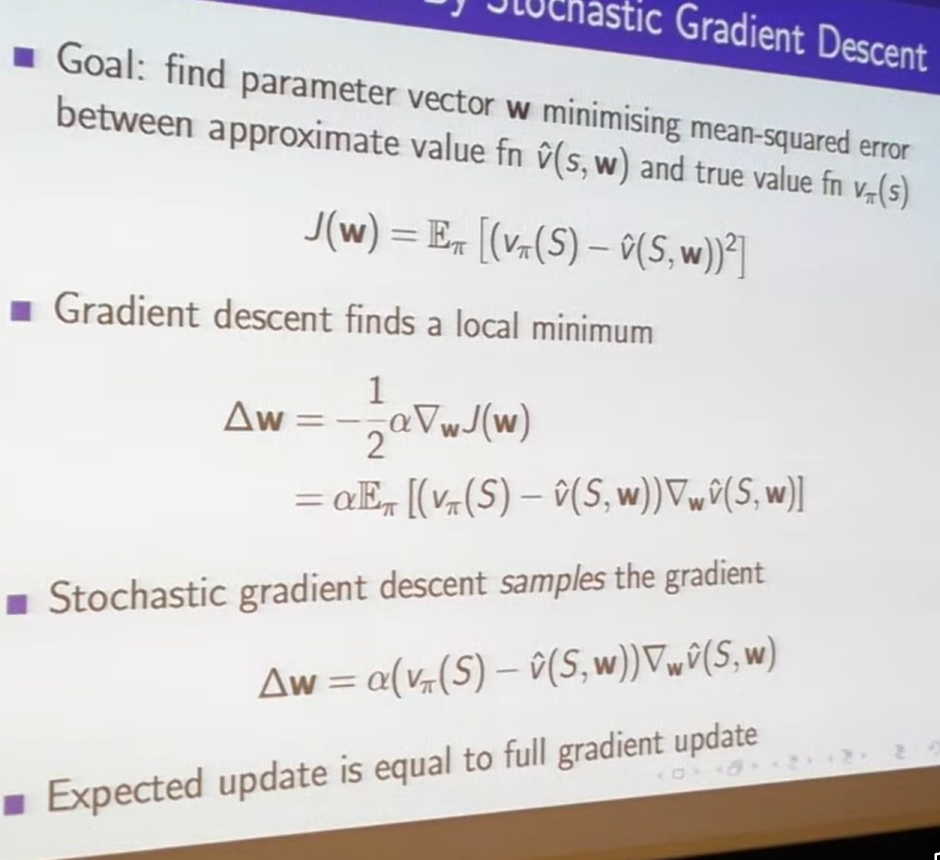
🔁 How SGD Updates the Approximator

At each time step, we:

1. **Sample a transition**: (s, a, r, s')
2. **Compute TD error**:
3. \delta = r + \gamma \hat{V}(s'; \theta) - \hat{V}(s; \theta)or for Q-values:
4. \delta = r + \gamma \hat{Q}(s', a'; \theta) - \hat{Q}(s, a; \theta)
5. **Update parameters** using gradient descent:

\theta \leftarrow \theta + \alpha \cdot \delta \cdot \nabla\_\theta \hat{V}(s; \theta)

This is called **semi-gradient TD learning**, because we only differentiate the current estimate, not the bootstrapped target.



# TD Target and TD Error

**The TD target is the estimated value of the next state (plus reward), and the TD error is the difference between this target and the current value estimate. Together, they drive learning in temporal-difference methods.**

🎯 TD Target

The TD target is what we *want* our current value estimate to move toward. It’s a one-step lookahead:

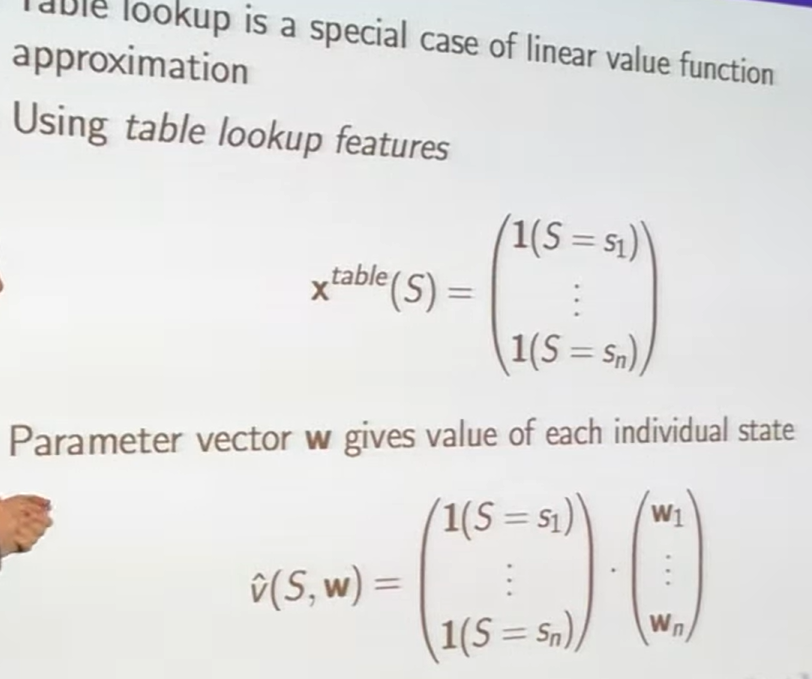
⚡ TD Error

The TD error is the difference between:

* What we *currently believe* (our estimate)
* What we *just learned* (the TD target)

# Linear Value Function Approximation

A **lookup table** is the simplest way to store value estimates in reinforcement learning. It’s like a dictionary:

* Each **state** s\_i has a corresponding **value** v(s\_i)

✅ **Conclusion**: A lookup table is just a linear approximator with one-hot features. It’s the most basic form of VFA — no generalization, just memorization.

# Incremental Prediction Algorithms

🧠 What Are Incremental Prediction Algorithms?

These are algorithms that:

* **Predict the value of a state** v\_\pi(s)
* **Update their estimates step-by-step**, using sampled experience
* **Don’t require a full model or supervisor** — they learn from rewards

🔁 Three Incremental Prediction Algorithms

🔹 1. **Monte Carlo (MC)**

* **Target**: Full return from episode : G\_t = r\_t + r\_{t+1} + r\_{t+2} + \dots

**Update rule**: \Delta w = \alpha (G\_t - \hat{v}(S\_t, w)) \nabla \hat{v}(S\_t, w)

🔍 This means:

* Wait until the episode ends
* Use the total reward as the target
* Update weights to reduce the error between predicted and actual return

🔹 2. **TD(0)** — Temporal Difference Learning

* **Target**: One-step bootstrapped estimate : R\_{t+1} + \gamma \hat{v}(S\_{t+1}, w)

**Update rule**: \Delta w = \alpha (R\_{t+1} + \gamma \hat{v}(S\_{t+1}, w) - \hat{v}(S\_t, w)) \nabla \hat{v}(S\_t, w)

🔍 This means:

* Use immediate reward + estimated future value
* Update after every step — no need to wait for episode end

🔹 3. **TD(λ)** — Eligibility Traces

* **Target**: λ-return — a mix of MC and TD(0) : G\_t^\lambda = (1 - \lambda) \sum\_{n=1}^{\infty} \lambda^{n-1} G\_t^{(n)}

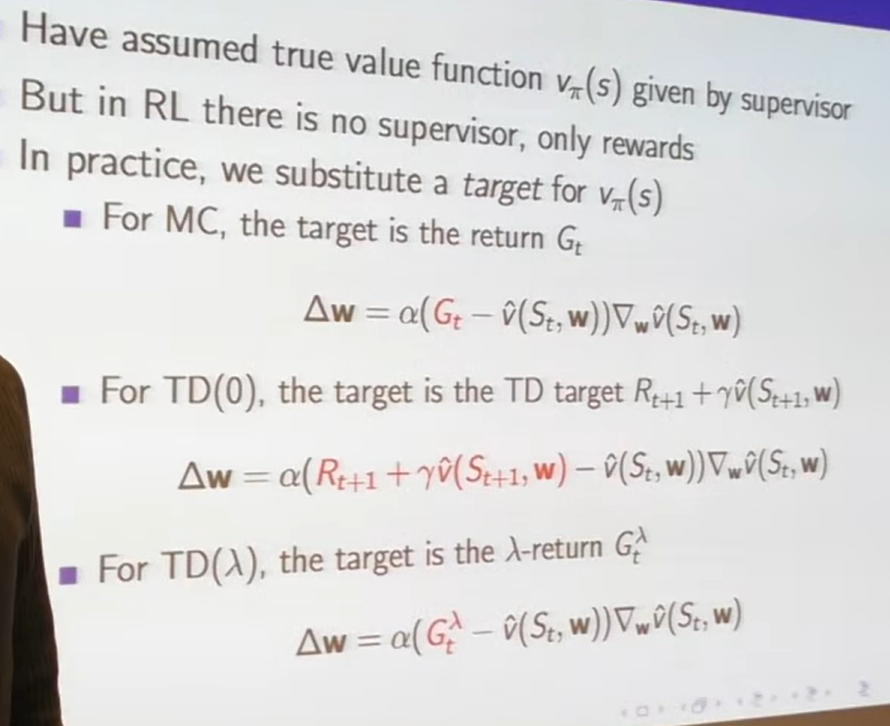
**Update rule**: \Delta w = \alpha (G\_t^\lambda - \hat{v}(S\_t, w)) \nabla \hat{v}(S\_t, w)

🔍 This means:

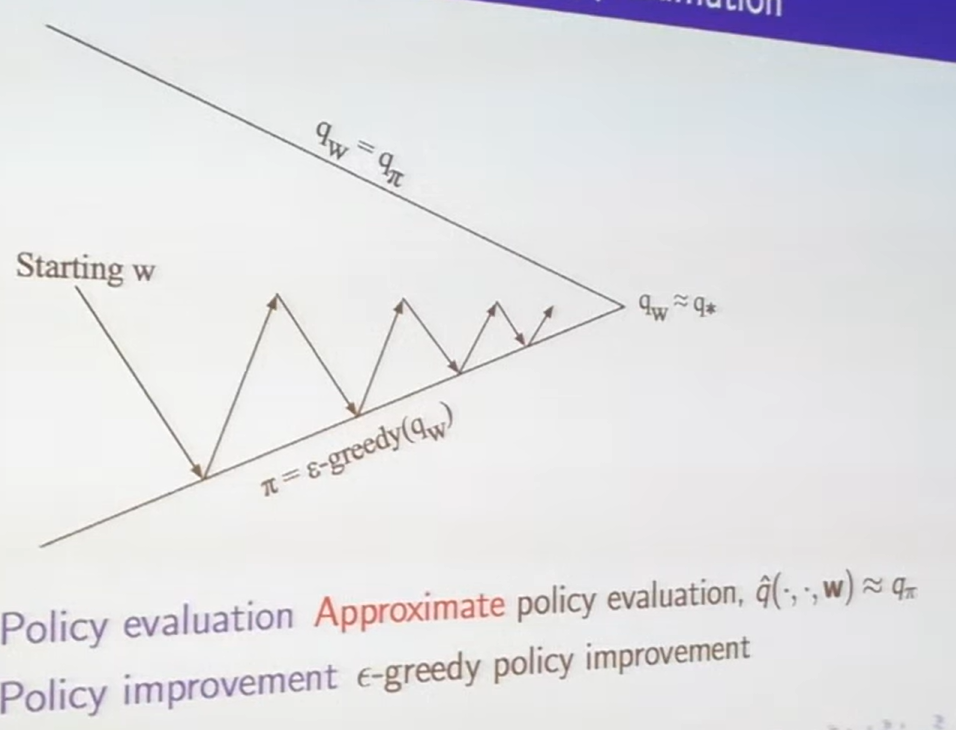
* Blend short-term and long-term predictions
* More flexible and often more stable

✅ Summary: What Makes These “Incremental”?

* They **learn step-by-step**, not all at once
* They **use sampled experience**, not full models
* They **update weights** using gradient descent based on prediction error



# Control with VFA



Control with Value Function Approximation (VFA) means learning a policy that maximizes long-term rewards using a function to estimate action values, instead of a lookup table. It enables reinforcement learning agents to scale to large or continuous environments.

🔧 Common Control Algorithms with VFA

1. **SARSA with VFA** (On-policy)

* Learns from the action actually taken
* Update rule:
* \delta = r + \gamma \hat{Q}(s', a'; \theta) - \hat{Q}(s, a; \theta)  
  \theta \leftarrow \theta + \alpha \cdot \delta \cdot \nabla\_\theta \hat{Q}(s, a; \theta)Uses the next action a' from the current policy

2. **Q-learning with VFA** (Off-policy)

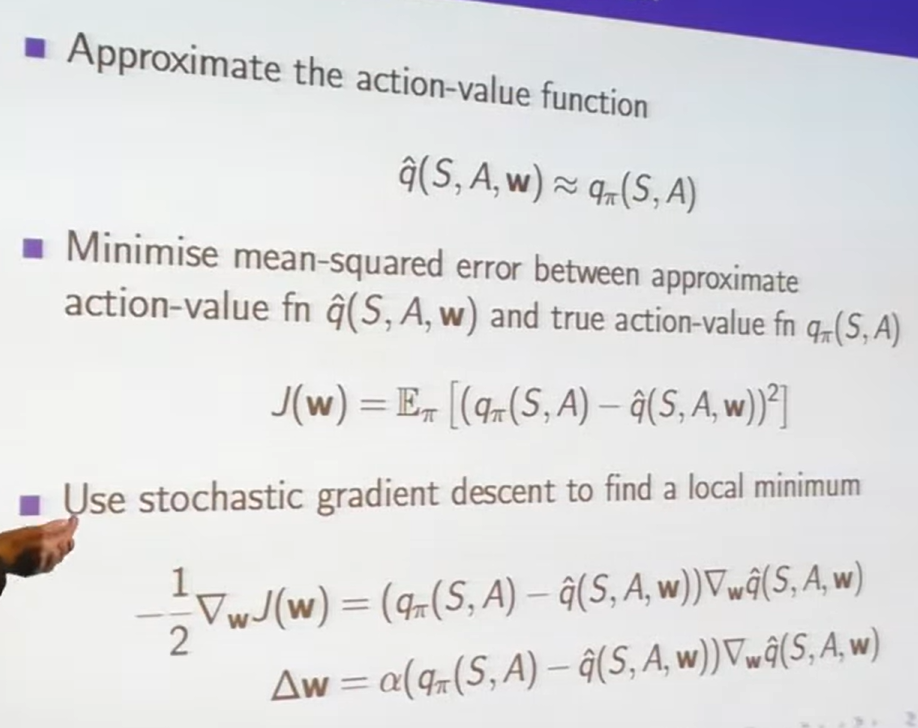
* Learns from the best possible action
* Update rule:
* \delta = r + \gamma \max\_{a'} \hat{Q}(s', a'; \theta) - \hat{Q}(s, a; \theta)  
  \theta \leftarrow \theta + \alpha \cdot \delta \cdot \nabla\_\theta \hat{Q}(s, a; \theta)Uses the greedy action a' that maximizes Q

3. **Expected SARSA with VFA**

* Uses the expected value over all possible actions
* More stable than SARSA or Q-learning

# Action Value Function Approximator

An Action-Value Function Approximator is a model that estimates the expected return for taking a specific action in a given state, using a parameterized function instead of a lookup table. It enables reinforcement learning agents to scale to large or continuous state-action spaces.



# Linear AVFA

**Linear Action-Value Function Approximation (Linear AVFA) uses a weighted sum of features to estimate the expected return for taking an action in a given state. It’s a scalable alternative to tabular methods and is especially useful in large or continuous environments.**

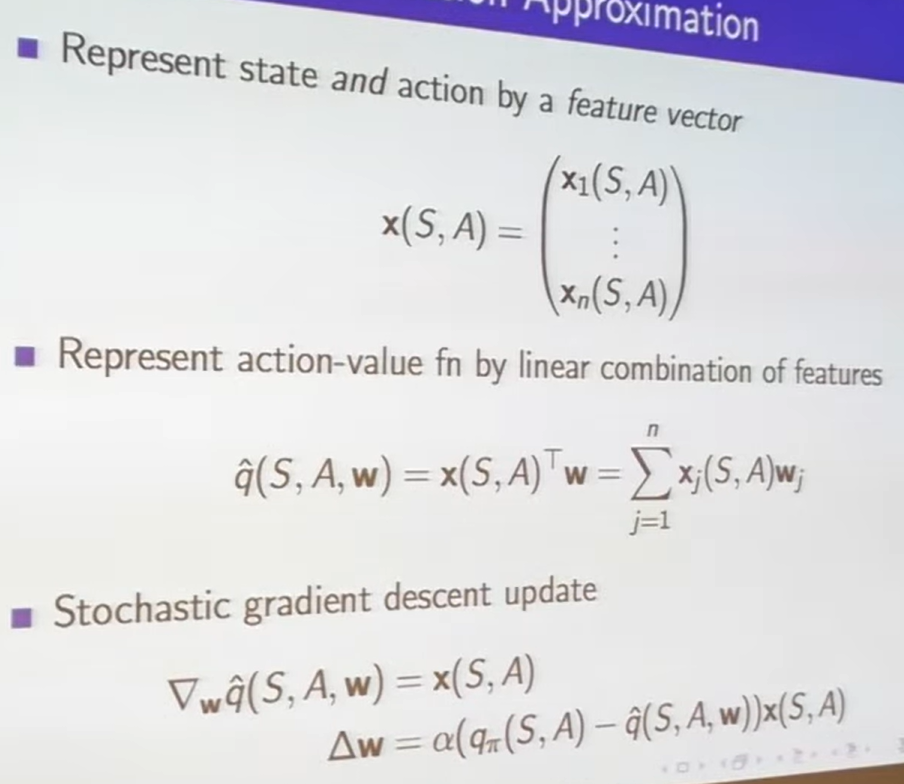
🧠 What Is Linear AVFA?

In reinforcement learning, the **action-value function** Q(s, a) estimates how good it is to take action in states.  
Linear AVFA approximates this using a **linear model**:

\hat{Q}(s, a; \mathbf{w}) = \phi(s, a)^T \mathbf{w} = \sum\_{i=1}^{d} w\_i \cdot \phi\_i(s, a)

* \phi(s, a): feature vector for the state-action pair
* \mathbf{w}: weight vector (parameters to learn)
* d: number of features

This replaces the need for a lookup table with a generalizable function.



# Incremental Control Algorithms

Incremental control algorithms in reinforcement learning are methods that learn optimal policies step-by-step using sampled experience, without requiring full knowledge of the environment or waiting for episodes to finish. They update value estimates and policies continuously as the agent interacts with the environment.

🔧 Key Incremental Control Algorithms

1. **SARSA (State-Action-Reward-State-Action)**

* **On-policy**: learns from the actions actually taken
* **Update rule**:
* Q(s, a) \leftarrow Q(s, a) + \alpha \left[ r + \gamma Q(s', a') - Q(s, a) \right]Updates after every transition

2. **Q-learning**

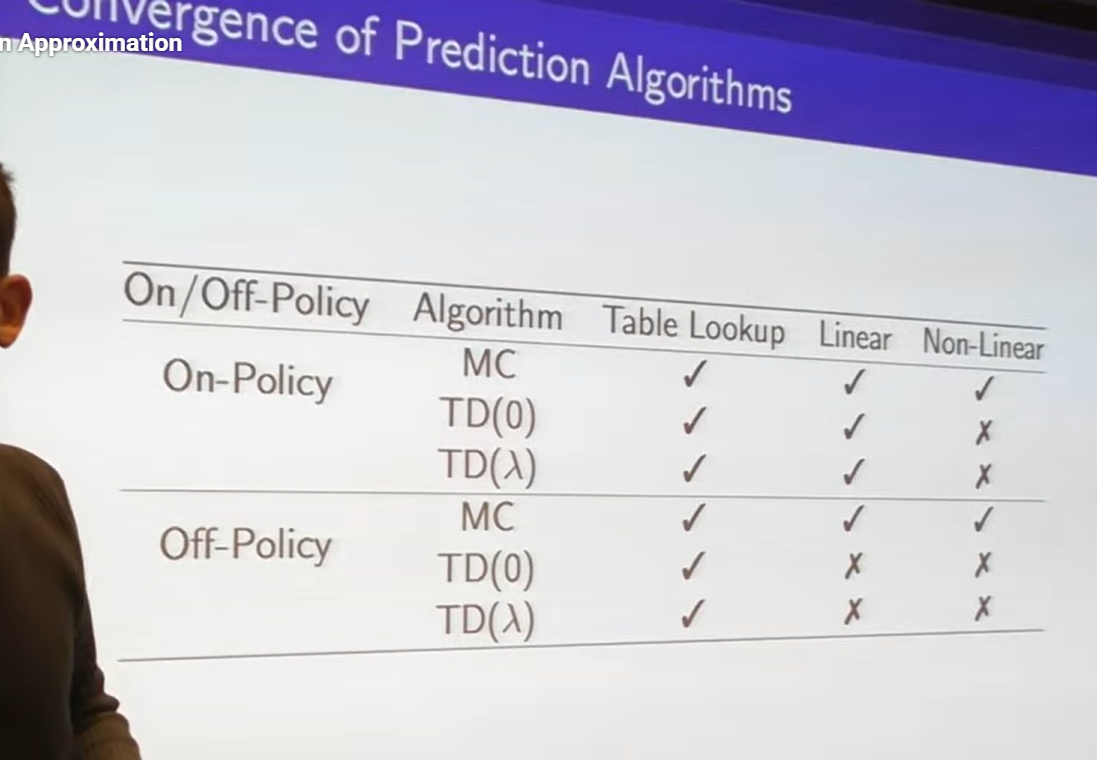
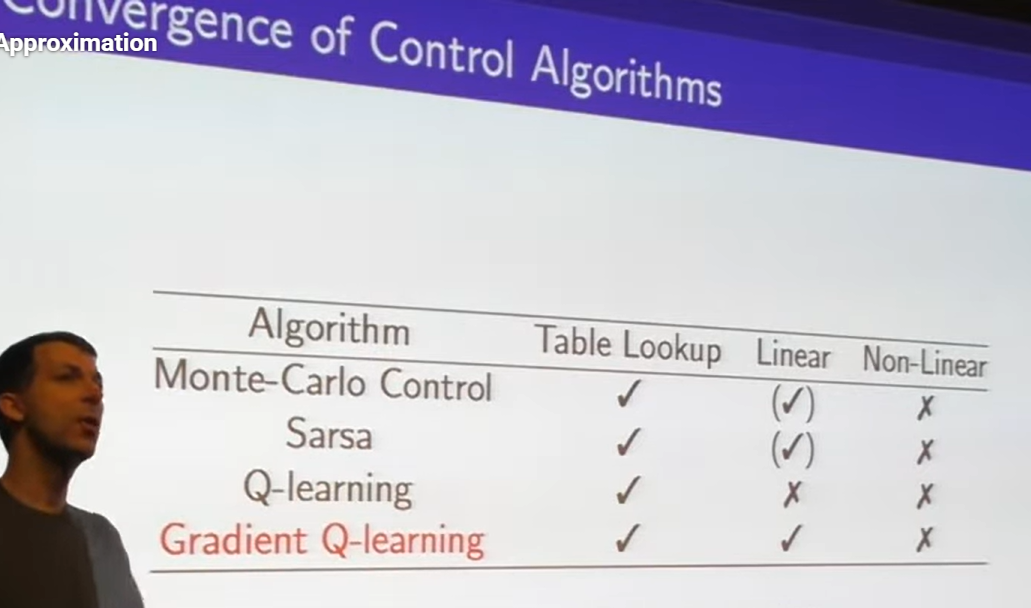
* **Off-policy**: learns from the best possible action
* **Update rule**:
* Q(s, a) \leftarrow Q(s, a) + \alpha \left[ r + \gamma \max\_{a'} Q(s', a') - Q(s, a) \right] . More aggressive policy improvement

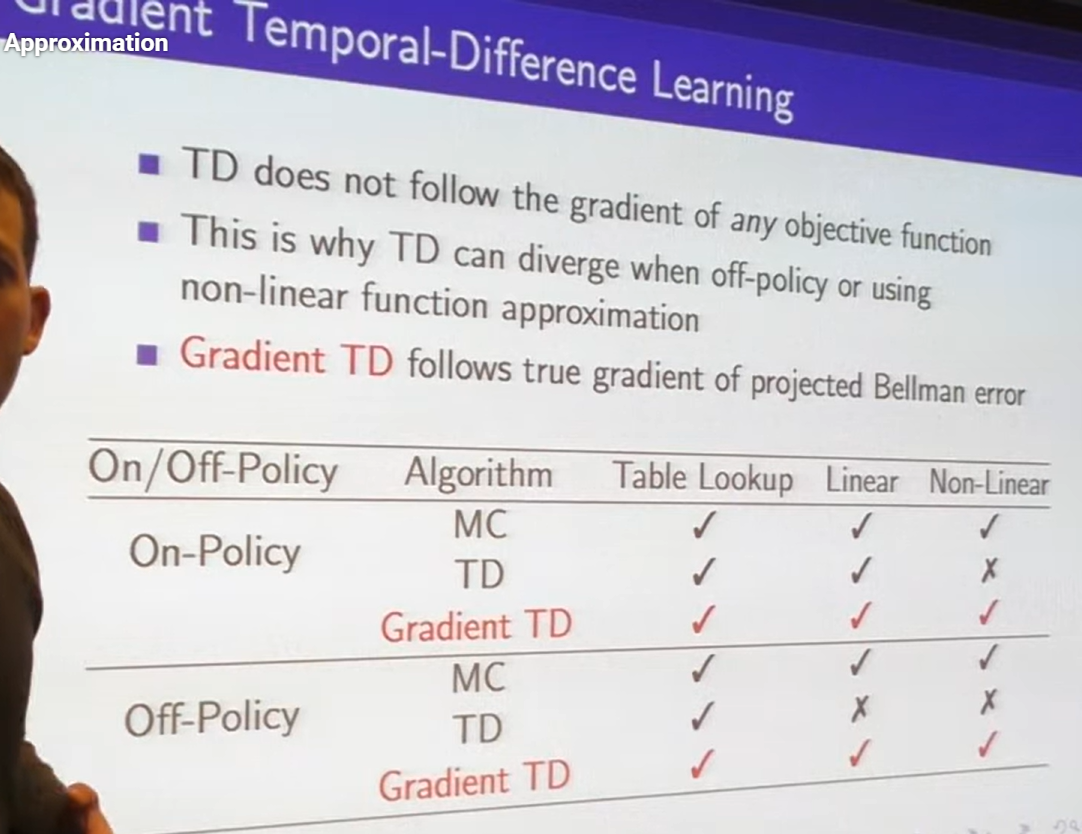
3. **Expected SARSA**

* Uses expected value over all actions under the current policy
* More stable than SARSA or Q-learning

**4. Actor-Critic**

* Separates value estimation (critic) and policy (actor)
* Critic updates incrementally using TD error
* Actor updates policy parameters using gradient ascent





🔹 “TD does not follow the gradient of any objective function”

* Traditional TD methods (like TD(0)) update value estimates using bootstrapped targets.
* But they don’t minimize a well-defined loss function — which can cause instability.

🔹 “This is why TD can diverge when off-policy or using non-linear function approximation”

* TD can **diverge** (fail to learn) when:
* The agent learns from a different policy than it behaves with (off-policy)
* The value function is approximated with a non-linear model (e.g., neural networks)

🔹 “Gradient TD follows true gradient of projected Bellman error”

* Gradient TD methods (e.g., GTD, GTD2, TDC) **do** minimize a well-defined objective.
* They are designed to be **stable** even with off-policy learning and non-linear approximators.

# Batch RL

**Batch Reinforcement Learning (Batch RL), also known as Offline RL, is a learning paradigm where the agent learns an optimal policy from a fixed dataset of past interactions, without further exploration or interaction with the environment.**

🧠 Core Idea

In **Batch RL**, the agent is given a **static batch of experience**:

\mathcal{D} = \{(s\_i, a\_i, r\_i, s'\_i)\}\_{i=1}^N

* These transitions are collected beforehand
* The agent **cannot interact** with the environment during learning
* The goal is to learn a policy \pi that performs well using only this data

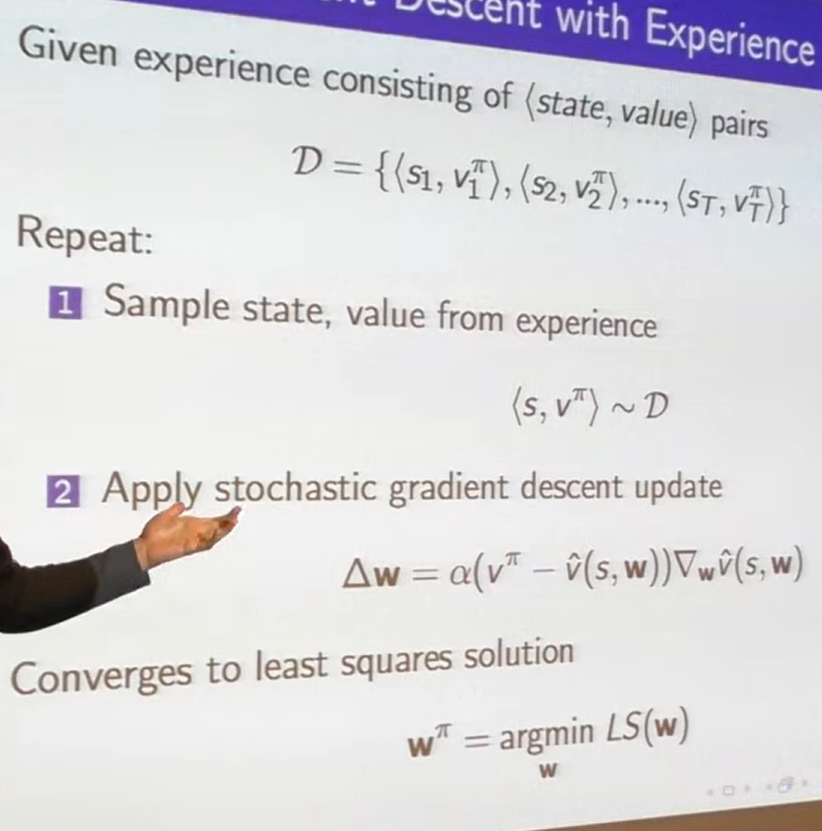
This is different from **online RL**, where the agent continuously explores and updates its policy.

🔍 Why Use Batch RL?

* **Safe learning**: No risky exploration (e.g., in healthcare or robotics)
* **Data efficiency**: Reuses existing logs (e.g., customer behavior, industrial logs)
* **Offline evaluation**: Useful when environment access is expensive or limited

# Least Square Prediction

# SGD with Experience Replay



# Experience Replay in DQN

**Deep Q-Network (DQN) is a reinforcement learning algorithm that uses a neural network to approximate the Q-value function, enabling agents to learn optimal policies in high-dimensional or continuous state spaces. Experience replay is a key technique in DQN that improves learning stability and efficiency by reusing past experiences.**

🧠 What Is DQN?

DQN combines **Q-learning** with **deep learning**:

* Instead of a lookup table, it uses a **neural network** to estimate Q(s, a)
* This allows it to handle complex inputs like images, sensor data, or large state spaces

🔧 Core Components:

* **Q-network**: predicts action values Q(s, a; \theta)
* **Target network**: a copy of the Q-network used to compute stable targets
* **Experience replay**: stores past transitions to break correlation and improve sample efficiency

🔁 Update Rule:

\theta \leftarrow \theta + \alpha \cdot \left( r + \gamma \max\_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \right) \cdot \nabla\_\theta Q(s, a; \theta)

* \theta: parameters of the Q-network
* \theta^-: parameters of the target network (updated periodically)

🔄 What Is Experience Replay?

**Experience replay** is a buffer that stores past transitions:

\mathcal{D} = \{(s\_t, a\_t, r\_t, s\_{t+1})\}

🔹 How It Works:

1. Every time the agent interacts with the environment, the transition is added to the buffer
2. During training, the agent samples **random mini-batches** from the buffer
3. These samples are used to update the Q-network

✅ Benefits:

* **Breaks correlation** between consecutive samples
* **Improves data efficiency** by reusing past experiences
* **Stabilizes training** by smoothing out updates