

Self-Driving Cars

Lecture 4 – Reinforcement Learning

Prof. Dr.-Ing. Andreas Geiger

Autonomous Vision Group

University of Tübingen / MPI-IS

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN



e l i a s
European Laboratory for Learning and Intelligent Systems

Agenda

4.1 Markov Decision Processes

4.2 Bellman Optimality and Q-Learning

4.3 Deep Q-Learning

4.1

Markov Decision Processes

Reinforcement Learning

So far:

- ▶ Supervised learning, lots of expert demonstrations required
- ▶ Use of auxiliary, short-term loss functions
 - ▶ Imitation learning: per-frame loss on action
 - ▶ Direct perception: per-frame loss on affordance indicators

Now:

- ▶ Learning of models based on the loss that we actually care about, e.g.:
 - ▶ Minimize time to target location
 - ▶ Minimize number of collisions
 - ▶ Minimize risk
 - ▶ Maximize comfort
 - ▶ etc.

Types of Learning

Supervised Learning:

- ▶ Dataset: $\{(x_i, y_i)\}$ (x_i = data, y_i = label) Goal: Learn mapping $x \mapsto y$
- ▶ Examples: Classification, regression, imitation learning, affordance learning, etc.

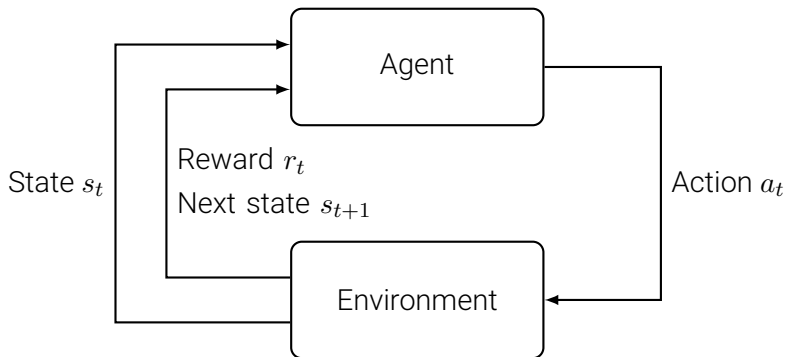
Unsupervised Learning:

- ▶ Dataset: $\{(x_i)\}$ (x_i = data) Goal: Discover structure underlying data
- ▶ Examples: Clustering, dimensionality reduction, feature learning, etc.

Reinforcement Learning:

- ▶ Agent interacting with environment which provides numeric reward signals
- ▶ Goal: Learn how to take actions in order to maximize reward
- ▶ Examples: Learning of manipulation or control tasks (everything that interacts)

Introduction to Reinforcement Learning

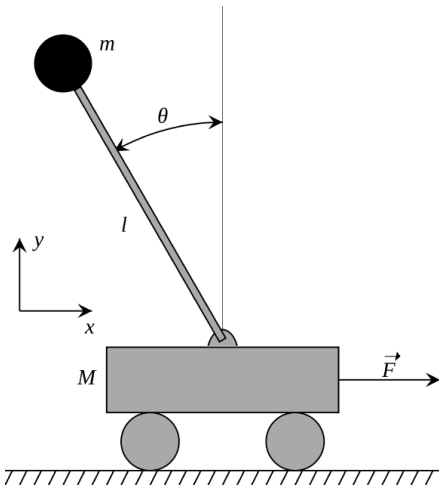


- ▶ Agent observes environment state s_t at time t
- ▶ Agent sends action a_t at time t to the environment
- ▶ Environment returns the reward r_t and its new state s_{t+1} to the agent

Introduction to Reinforcement Learning

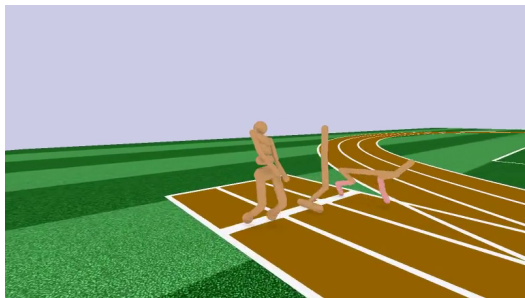
- ▶ Goal: Select actions to maximize total future reward
- ▶ Actions may have long term consequences
- ▶ Reward may be delayed, not instantaneous
- ▶ It may be better to sacrifice immediate reward to gain more long-term reward
- ▶ Examples:
 - ▶ Financial investment (may take months to mature)
 - ▶ Refuelling a helicopter (might prevent crash in several hours)
 - ▶ Sacrificing a chess piece (might help winning chances in the future)

Example: Cart Pole Balancing



- **Objective:** Balance pole on moving cart
- **State:** Angle, angular vel., position, vel.
- **Action:** Horizontal force applied to cart
- **Reward:** 1 if pole is upright at time t

Example: Robot Locomotion



<http://blog.openai.com/roboschool/>

- ▶ **Objective:** Make robot move forward
- ▶ **State:** Position and angle of joints
- ▶ **Action:** Torques applied on joints
- ▶ **Reward:** 1 if upright & forward moving

Example: Atari Games



- **Objective:** Maximize game score
- **State:** Raw pixels of screen (210x160)
- **Action:** Left, right, up, down
- **Reward:** Score increase/decrease at t

<http://blog.openai.com/gym-retro/>

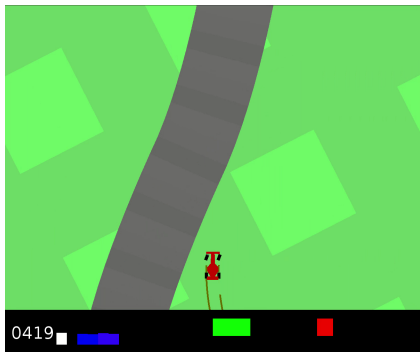
Example: Go



- ▶ **Objective:** Winning the game
- ▶ **State:** Position of all pieces
- ▶ **Action:** Location of next piece
- ▶ **Reward:** 1 if game won, 0 otherwise

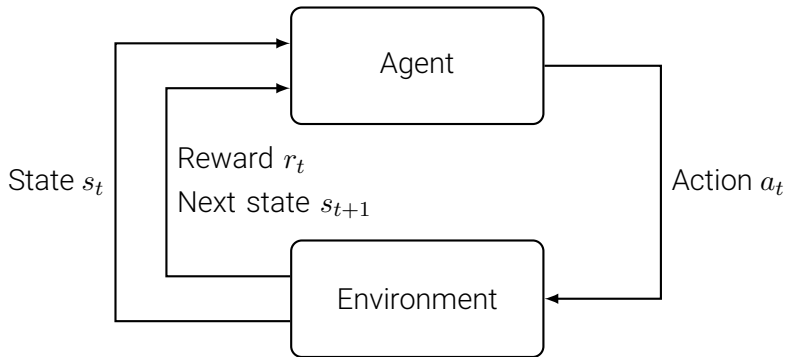
www.deepmind.com/research/alphago/

Example: Self-Driving



- ▶ **Objective:** Lane Following
- ▶ **State:** Image (96x96)
- ▶ **Action:** Acceleration, Steering
- ▶ **Reward:** - per frame, + per tile

Reinforcement Learning: Overview



- How can we mathematically formalize the RL problem?

Markov Decision Process

Markov Decision Process (MDP) models the environment and is defined by the tuple

$$(\mathcal{S}, \mathcal{A}, \mathcal{R}, P, \gamma)$$

with

- ▶ \mathcal{S} : set of possible states
- ▶ \mathcal{A} : set of possible actions
- ▶ $\mathcal{R}(r_t|s_t, a_t)$: distribution of current reward given (state,action) pair
- ▶ $P(s_{t+1}|s_t, a_t)$: distribution over next state given (state,action) pair
- ▶ γ : discount factor (determines value of future rewards)

Almost all reinforcement learning problems can be formalized as MDPs

Markov Decision Process

Markov property: Current state completely characterizes state of the world

- ▶ A state s_t is *Markov* if and only if

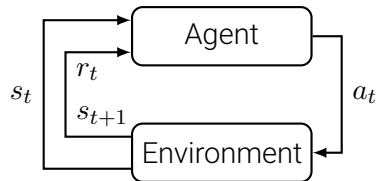
$$P(s_{t+1}|s_t) = P(s_{t+1}|s_1, \dots, s_t)$$

- ▶ "The future is independent of the past given the present"
- ▶ The state captures all relevant information from the history
- ▶ Once the state is known, the history may be thrown away
- ▶ The state is a sufficient statistics of the future

Markov Decision Process

Reinforcement learning loop:

- ▶ At time $t = 0$:
 - ▶ Environment samples initial state $s_0 \sim P(s_0)$
- ▶ Then, for $t = 0$ until done:
 - ▶ Agent selects action a_t
 - ▶ Environment samples reward $r_t \sim \mathcal{R}(r_t|s_t, a_t)$
 - ▶ Environment samples next state $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$
 - ▶ Agent receives reward r_t and next state s_{t+1}



How do we select an action?

Policy

A **policy** π is a function from \mathcal{S} to \mathcal{A} that specifies what action to take in each state:

- ▶ A policy fully defines the behavior of an agent
- ▶ Deterministic policy: $a = \pi(s)$
- ▶ Stochastic policy: $\pi(a|s) = P(a_t = a | s_t = s)$

Remark:

- ▶ MDP policies depend only on the **current state** and not the entire history
- ▶ However, the current state may include past observations

Policy

How do we learn a policy?

Imitation Learning: Learn a policy from **expert demonstrations**

- ▶ Expert demonstrations are provided
- ▶ Supervised learning problem

Reinforcement Learning: Learn a policy through **trial-and-error**

- ▶ No expert demonstrations given
- ▶ Agent discovers itself which actions **maximize the expected future reward**
 - ▶ The agent interacts with the environment and obtains reward
 - ▶ The agent discovers good actions and improves its policy π

Exploration vs. Exploitation

How do we discover good actions?

Answer: We need to explore the state/action space. Thus RL combines two tasks:

- ▶ **Exploration:** Try a novel action a in state s , observe reward r_t
 - ▶ Discovers more information about the environment, but sacrifices total reward
 - ▶ Game-playing example: Play a novel experimental move
- ▶ **Exploitation:** Use a previously discovered good action a
 - ▶ Exploits known information to maximize reward, but sacrifice unexplored areas
 - ▶ Game-playing example: Play the move you believe is best

Trade-off: It is important to explore and exploit simultaneously

Exploration vs. Exploitation

How to balance exploration and exploitation?

ϵ -**greedy** exploration algorithm:

- ▶ Try all possible actions with non-zero probability
- ▶ With probability ϵ choose an action at random (**exploration**)
- ▶ With probability $1 - \epsilon$ choose the best action (**exploitation**)
- ▶ Greedy action is defined as best action which was discovered so far
- ▶ ϵ is large initially and gradually annealed (=reduced) over time

Value Functions

How good is a state?

The **state-value function** V^π at state s_t is the expected cumulative discounted reward ($r_t \sim \mathcal{R}(r_t|s_t, a_t)$) when following policy π from state s_t :

$$V^\pi(s_t) = \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t, \pi] = \mathbb{E} \left[\sum_{k \geq 0} \gamma^k r_{t+k} \middle| s_t, \pi \right]$$

- ▶ The discount factor $\gamma < 1$ is the value of future rewards at current time t
 - ▶ Weights immediate reward higher than future reward
(e.g., $\gamma = \frac{1}{2} \Rightarrow \gamma^k = \frac{1}{1}, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \dots$)
 - ▶ Determines agent's far/short-sightedness
 - ▶ Avoids infinite returns in cyclic Markov processes

Value Functions

How good is a state-action pair?

The **action-value function** Q^π at state s_t and action a_t is the expected cumulative discounted reward when taking action a_t in state s_t and then following the policy π :

$$Q^\pi(s_t, a_t) = \mathbb{E} \left[\sum_{k \geq 0} \gamma^k r_{t+k} \middle| s_t, a_t, \pi \right]$$

- ▶ The discount factor $\gamma \in [0, 1]$ is the value of future rewards at current time t
 - ▶ Weights immediate reward higher than future reward
(e.g., $\gamma = \frac{1}{2} \Rightarrow \gamma^k = \frac{1}{1}, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \dots$)
 - ▶ Determines agent's far/short-sightedness
 - ▶ Avoids infinite returns in cyclic Markov processes

Optimal Value Functions

The **optimal state-value function** $V^*(s_t)$ is the best $V^\pi(s_t)$ over all policies π :

$$V^*(s_t) = \max_{\pi} V^\pi(s_t) \qquad V^\pi(s_t) = \mathbb{E} \left[\sum_{k \geq 0} \gamma^k r_{t+k} \middle| s_t, \pi \right]$$

The **optimal action-value function** $Q^*(s_t, a_t)$ is the best $Q^\pi(s_t, a_t)$ over all policies π :

$$Q^*(s_t, a_t) = \max_{\pi} Q^\pi(s_t, a_t) \qquad Q^\pi(s_t, a_t) = \mathbb{E} \left[\sum_{k \geq 0} \gamma^k r_{t+k} \middle| s_t, a_t, \pi \right]$$

- ▶ The optimal value functions specify the best possible performance in the MDP
- ▶ However, searching over all possible policies π is computationally intractable

Optimal Policy

If $Q^*(s_t, a_t)$ would be known, what would be the **optimal policy**?

$$\pi^*(s_t) = \operatorname{argmax}_{a' \in \mathcal{A}} Q^*(s_t, a')$$

- ▶ Unfortunately, searching over all possible policies π is intractable in most cases
- ▶ Thus, determining $Q^*(s_t, a_t)$ is hard in general (for most interesting problems)
- ▶ Let's have a look at a simple example where the optimal policy is easy to compute

A Simple Grid World Example

actions = {

1. right →

2. left ←

3. up ↑

4. down ↓

}

states

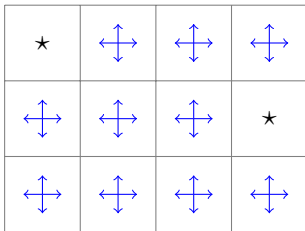
★			
			★

reward: $r = -1$ for
each transition

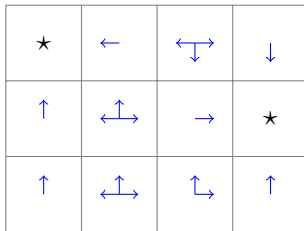
Objective: Reach one of terminal states (marked with '★') in least number of actions

- Penalty (negative reward) given for every transition made

A Simple Grid World Example



Random Policy



Optimal Policy

Solving for the Optimal Policy

Bellman Optimality Equation

- ▶ The **Bellman Optimality Equation** is named after Richard Ernest Bellman who introduced **dynamic programming** in 1953
- ▶ Almost any problem which can be solved using **optimal control theory** can be solved via the appropriate Bellman equation



Richard Ernest Bellman

Bellman Optimality Equation

The **Bellman Optimality Equation (BOE)** decomposes Q^* as follows:

$$\begin{aligned} Q^*(s_t, a_t) &= \mathbb{E} [r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t, a_t] \\ &\stackrel{BOE}{=} \mathbb{E} \left[r_t + \gamma \max_{a' \in \mathcal{A}} Q^*(s_{t+1}, a') \middle| s_t, a_t \right] \end{aligned}$$

This **recursive formulation** comprises two parts:

- ▶ Current reward: r_t
- ▶ Discounted optimal action-value of successor: $\gamma \max_{a' \in \mathcal{A}} Q^*(s_{t+1}, a')$

We want to **determine** $Q^*(s_t, a_t)$. How can we **solve** the BOE?

- ▶ The BOE is non-linear (because of max-operator) \Rightarrow no closed form solution
- ▶ Several iterative methods have been proposed, most popular: Q-Learning

Proof of the Bellman Optimality Equation

Proof of the **Bellman Optimality Equation** for the **optimal action-value function** Q^* :

$$\begin{aligned}Q^*(s_t, a_t) &= \mathbb{E} [r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t, a_t] \\&= \mathbb{E} \left[\sum_{k \geq 0} \gamma^k r_{t+k} | s_t, a_t \right] \\&= \mathbb{E} \left[r_t + \gamma \sum_{k \geq 0} \gamma^k r_{t+k+1} | s_t, a_t \right] \\&= \mathbb{E} [r_t + \gamma V^*(s_{t+1}) | s_t, a_t] \\&= \mathbb{E} \left[r_t + \gamma \max_{a'} Q^*(s_{t+1}, a') | s_t, a_t \right]\end{aligned}$$

Bellman Optimality Equation

Why is it useful to solve the BOE?

- ▶ A greedy policy which chooses the action that maximizes the optimal action-value function Q^* or the optimal state-value function V^* takes into account the reward consequences of all possible future behavior
- ▶ Via Q^* and V^* the optimal expected long-term return is turned into a quantity that is locally and immediately available for each state / state-action pair
- ▶ For V^* , a one-step-ahead search yields the optimal actions
- ▶ Q^* effectively caches the results of all one-step-ahead searches

Q-Learning

Q-Learning: Iteratively solve for Q^*

$$Q^*(s_t, a_t) = \mathbb{E} \left[r_t + \gamma \max_{a' \in \mathcal{A}} Q^*(s_{t+1}, a') \middle| s_t, a_t \right]$$

by constructing an **update sequence** Q_1, Q_2, \dots using learning rate α :

$$\begin{aligned} Q_{i+1}(s_t, a_t) &\leftarrow (1 - \alpha)Q_i(s_t, a_t) + \alpha(r_t + \gamma \max_{a' \in \mathcal{A}} Q_i(s_{t+1}, a')) \\ &= Q_i(s_t, a_t) + \underbrace{\alpha(r_t + \gamma \max_{a' \in \mathcal{A}} Q_i(s_{t+1}, a') - Q_i(s_t, a_t))}_{\text{temporal difference (TD) error}} \end{aligned}$$

$\underbrace{\hspace{10em}}_{\text{target}} \quad \underbrace{\hspace{10em}}_{\text{prediction}}$

► Q_i will converge to Q^* as $i \rightarrow \infty$ Note: policy π learned implicitly via Q table!

Q-Learning

Implementation:

- ▶ Initialize Q table and initial state s_0 randomly
- ▶ Repeat:
 - ▶ Observe state s_t , choose action a_t according to ϵ -greedy strategy
(Q-Learning is “off-policy” as the updated policy is different from the behavior policy)
 - ▶ Observe reward r_t and next state s_{t+1}
 - ▶ Compute TD error: $r_t + \gamma \max_{a' \in \mathcal{A}} Q_i(s_{t+1}, a') - Q_i(s_t, a_t)$
 - ▶ Update Q table

What's the problem with using Q tables?

- ▶ **Scalability:** Tables don't scale to high dimensional state/action spaces (e.g., GO)
- ▶ **Solution:** Use a function approximator (neural network) to represent $Q(s, a)$

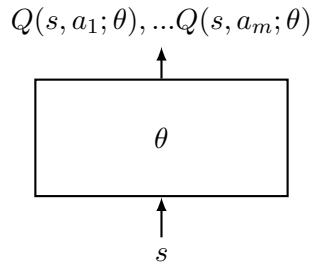
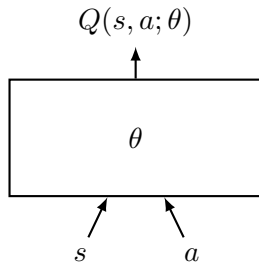
4.3

Deep Q-Learning

Deep Q-Learning

Use a **deep neural network** with weights θ as function approximator to estimate Q :

$$Q(s, a; \theta) \approx Q^*(s, a)$$



Training the Q Network

Forward Pass:

Loss function is the mean-squared error in Q-values:

$$\mathcal{L}(\theta) = \mathbb{E} \left[\left(\underbrace{r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta)}_{\text{target}} - \underbrace{Q(s_t, a_t; \theta)}_{\text{prediction}} \right)^2 \right]$$

Backward Pass:

Gradient update with respect to Q -function parameters θ :

$$\nabla_{\theta} \mathcal{L}(\theta) = \nabla_{\theta} \mathbb{E} \left[\left(r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) - Q(s_t, a_t; \theta) \right)^2 \right]$$

Optimize objective end-to-end with stochastic gradient descent (SGD) using $\nabla_{\theta} \mathcal{L}(\theta)$.

Experience Replay

To speed-up training we like to train on **mini-batches**:

- ▶ Problem: Learning from consecutive samples is inefficient
- ▶ Reason: Strong correlations between consecutive samples

Experience replay stores agent's experiences at each time-step

- ▶ Continually update a **replay memory** D with new experiences $e_t = (s_t, a_t, r_t, s_{t+1})$
- ▶ Train on samples $(s_t, a_t, r_t, s_{t+1}) \sim U(D)$ drawn uniformly at random from D
- ▶ Breaks correlations between samples
- ▶ Improves data efficiency as each sample can be used multiple times

In practice, a **circular replay memory** of finite memory size is used.

Fixed Q Targets

Problem: Non-stationary targets

- ▶ As the policy changes, so do our targets: $r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta)$
- ▶ This may lead to oscillation or divergence

Solution: Use fixed Q targets to stabilize training

- ▶ A target network Q with weights θ^- is used to generate the targets:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1}) \sim U(D)} \left[\left(r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-) - Q(s_t, a_t; \theta) \right)^2 \right]$$

- ▶ Target network Q is only updated every C steps by cloning the Q -network
- ▶ Effect: Reduces oscillation of the policy by adding a delay

Putting it together

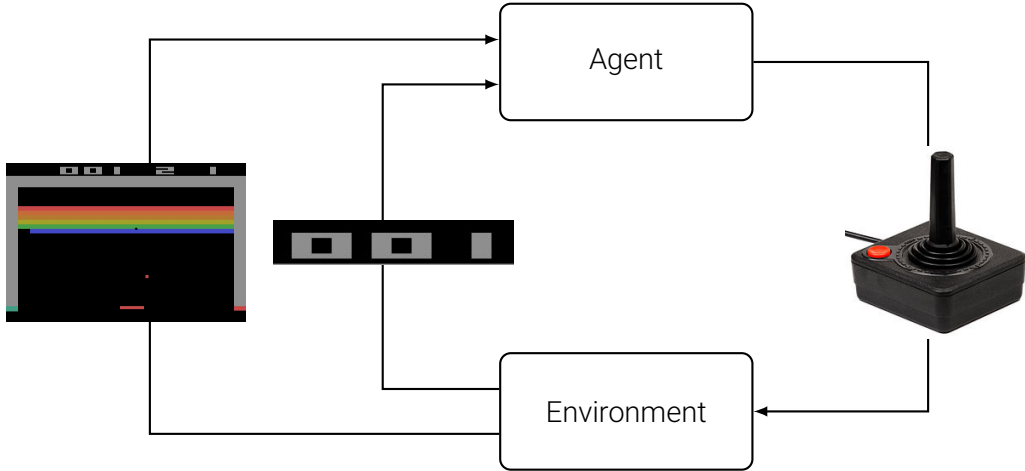
Deep Q-Learning using experience replay and fixed Q targets:

- ▶ Take action a_t according to ϵ -greedy policy
- ▶ Store transition (s_t, a_t, r_t, s_{t+1}) in replay memory D
- ▶ Sample random mini-batch of transitions (s_t, a_t, r_t, s_{t+1}) from D
- ▶ Compute Q targets using old parameters θ^-
- ▶ Optimize MSE between Q targets and Q network predictions

$$\mathcal{L}(\theta) = \mathbb{E}_{s_t, a_t, r_t, s_{t+1} \sim D} \left[\left(r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-) - Q(s_t, a_t; \theta) \right)^2 \right]$$

using stochastic gradient descent.

Case Study: Playing Atari Games



Objective: Complete the game with the highest score

Case Study: Playing Atari Games

$Q(s, a; \theta)$: Neural network with weights θ

FC-Out (Q values)

FC-256

32 4x4 conv, stride 2

16 8x8 conv, stride 2



Output: Q values for all (4 to 18) Atari actions
(efficient: single forward pass computes Q for all actions)

Input: $84 \times 84 \times 4$ stack of last 4 frames
(after grayscale conversion, downsampling, cropping)

Case Study: Playing Atari Games



Deep Q-Learning Shortcomings

Deep Q-Learning suffers from several **shortcomings**:

- ▶ Long training times
- ▶ Uniform sampling from replay buffer \Rightarrow all transitions equally important
- ▶ Simplistic exploration strategy
- ▶ Relies on fully-observable states
- ▶ Action space is limited to a discrete set of actions

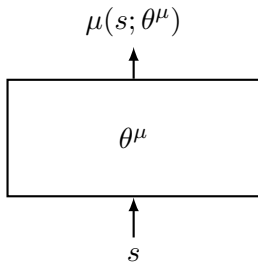
Various **improvements** over the original algorithm have been explored.

Deep Deterministic Policy Gradients

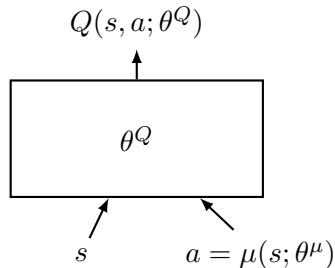
DDPG addresses the problem of continuous action spaces.

Problem: Finding a continuous action requires optimization at every timestep.

Solution: Use two networks, an **actor** (deterministic policy) and a **critic**.



Actor



Critic

Deep Deterministic Policy Gradients

- ▶ **Actor** network with weights θ^μ estimates agent's deterministic policy $\mu(s; \theta^\mu)$
 - ▶ Update deterministic policy $\mu(\cdot)$ in direction that most improves Q
 - ▶ Apply chain rule to the **expected return** (this is the policy gradient):

$$\nabla_{\theta^\mu} \mathbb{E}_{s_t, a_t, r_t, s_{t+1} \sim D} [Q(s_t, \mu(s_t; \theta^\mu); \theta^Q)] = \mathbb{E} [\nabla_{a_t} Q(s_t, a_t; \theta^Q) \nabla_{\theta^\mu} \mu(s_t; \theta^\mu)]$$

- ▶ **Critic** estimates value of current policy $Q(s, a; \theta^Q)$
 - ▶ Learned using the **Bellman Optimality Equation** as in Q Learning:

$$\nabla_{\theta^Q} \mathbb{E}_{s_t, a_t, r_t, s_{t+1} \sim D} \left[(r_t + \gamma Q(s_{t+1}, \mu(s_{t+1}; \theta^{\mu-}); \theta^{Q-}) - Q(s_t, a_t; \theta^Q))^2 \right]$$

- ▶ Remark: No maximization over actions required as this step is now learned via $\mu(\cdot)$

Deep Deterministic Policy Gradients

Experience replay and **target networks** are again used to stabilize training:

- ▶ Replay memory D stores transition tuples (s_t, a_t, r_t, s_{t+1})
- ▶ Target networks are updated using “soft” target updates
 - ▶ Weights are not directly copied but slowly adapted:

$$\begin{aligned}\theta^{Q-} &\leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q-} \\ \theta^{\mu-} &\leftarrow \tau \theta^{\mu} + (1 - \tau) \theta^{\mu-}\end{aligned}$$

where $0 < \tau \ll 1$ controls the tradeoff between speed and stability of learning

Exploration is performed by adding noise $\nabla_{\theta^{\mu}}$ to the policy $\mu(s)$:

$$\mu(s; \theta^{\mu}) + \mathcal{N}$$

Prioritized Experience Replay

Prioritize experience to replay important transitions more frequently

- ▶ Priority δ is measured by magnitude of temporal difference (TD) error:

$$\delta = |r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^{Q-}) - Q(s_t, a_t; \theta^Q)|$$

- ▶ TD error measures how “surprising” or unexpected the transition is
- ▶ Stochastic prioritization avoids overfitting due to lack of diversity
- ▶ Enables learning speed-up by a factor of 2 on Atari benchmarks

Learning to Drive in a Day

Real-world RL demo by Wayve:

- ▶ Deep Deterministic Policy Gradients with Prioritized Experience Replay
- ▶ **Input:** Single monocular image
- ▶ **Action:** Steering and speed
- ▶ **Reward:** Distance traveled without the safety driver taking control (requires no maps / localization)
- ▶ 4 Conv layers, 2 FC layers
- ▶ Only 35 training episodes

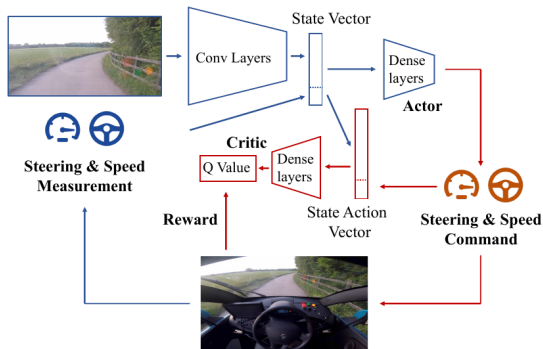


Fig. 1: We design a deep reinforcement learning algorithm for autonomous driving. This figure illustrates the actor-critic algorithm which we use to learn a policy and value function for driving. Our agent maximises the reward of distance travelled before intervention by a safety driver.

Learning to Drive in a Day



Other flavors of Deep RL

Asynchronous Deep Reinforcement Learning

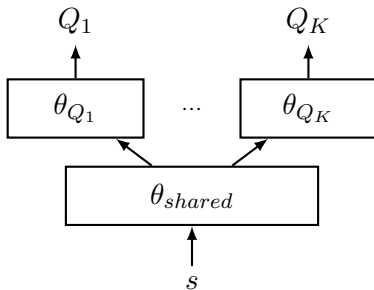
Execute multiple agents in separate environment instances:

- ▶ Each agent interacts with its own environment copy and collects experience
- ▶ Agents may use different exploration policies to maximize experience diversity
- ▶ Experience is not stored but directly used to update a shared global model
- ▶ Stabilizes training in similar way to experience replay by decorrelating samples
- ▶ Leads to reduction in training time roughly linear in the number of parallel agents

Bootstrapped DQN

Bootstrapping for efficient exploration:

- ▶ Approximate a distribution over Q values via K bootstrapped "heads"
- ▶ At the start of each epoch, a single head Q_k is selected uniformly at random
- ▶ After training, all heads can be combined into a single ensemble policy



Double Q-Learning

Double Q-Learning

- Decouple Q function for selection and evaluation of actions to avoid Q overestimation and stabilize training. Target:

$$DQN \quad : \quad r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-)$$

$$DoubleDQN \quad : \quad r_t + \gamma Q(s_{t+1}, \operatorname{argmax}_{a'} Q(s_{t+1}, a'; \theta); \theta^-)$$

- Online network with weights θ is used to determine greedy policy
- Target network with weights θ^- is used to determine corresponding action value
- Improves performance on Atari benchmarks

Deep Recurrent Q-Learning

Add recurrency to a deep Q-network to handle *partial observability* of states:

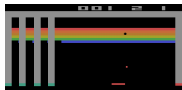
FC-Out (Q-values)

LSTM

Replace fully-connected layer with **recurrent LSTM layer**

32 4x4 conv, stride 2

16 8x8 conv, stride 2



Faulty Reward Functions



Summary

- ▶ Reinforcement learning learns through **interaction** with the environment
- ▶ The environment is typically modeled as a **Markov Decision Process**
- ▶ The goal of RL is to **maximize the expected future reward**
- ▶ Reinforcement learning requires trading off **exploration** and **exploitation**
- ▶ **Q-Learning** iteratively solves for the optimal action-value function
- ▶ The policy is learned implicitly via the **Q table**
- ▶ **Deep Q-Learning** scales to continuous/high-dimensional state spaces
- ▶ **Deep Deterministic Policy Gradients** scales to continuous action spaces
- ▶ Experience replay and target networks are necessary to stabilize training