CS 4803-DL / 7643-A: LECTURE 23
DANFEI XU

Topics:

- Reinforcement Learning Part 1
  - **Markov Decision Processes**
  - **Value Iteration**
  - **(Deep) Q Learning**

Administrative

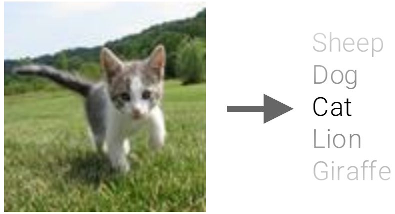
- HW4 is due EOD 11/12. Grace period ends 11/14

Reinforcement Learning Introduction

Types of Machine Learning

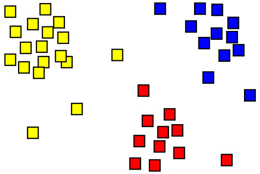
Supervised Learning

- Train Input: $\{X, Y\}$
- Learning output: $f : X \rightarrow Y,\ P(y|x)$
- e.g. classification



Unsupervised Learning

- Input: $\{X\}$
- Learning output: $P(x)$
- Example: Clustering, density estimation, generative modeling



Reinforcement Learning

- Evaluative feedback in the form of **reward**
- No supervision on the right action

Decision Making

- **Interactive Environment:** Unlike other ML paradigms, decision making is to act optimally in a dynamic, interactive environment.
- **Feedback Loop:** The agent's actions directly influence the future distribution of inputs, creating a continuous feedback loop.
- **Optimality:** The goal is to learn actions that maximize sum of future rewards, focusing on long-term outcomes.
- **Learn to predict:** The model must be able to predict, either implicitly or explicitly, how the environment changes in response to the agent's actions.

RL: Sequential decision making in an environment with evaluative feedback.

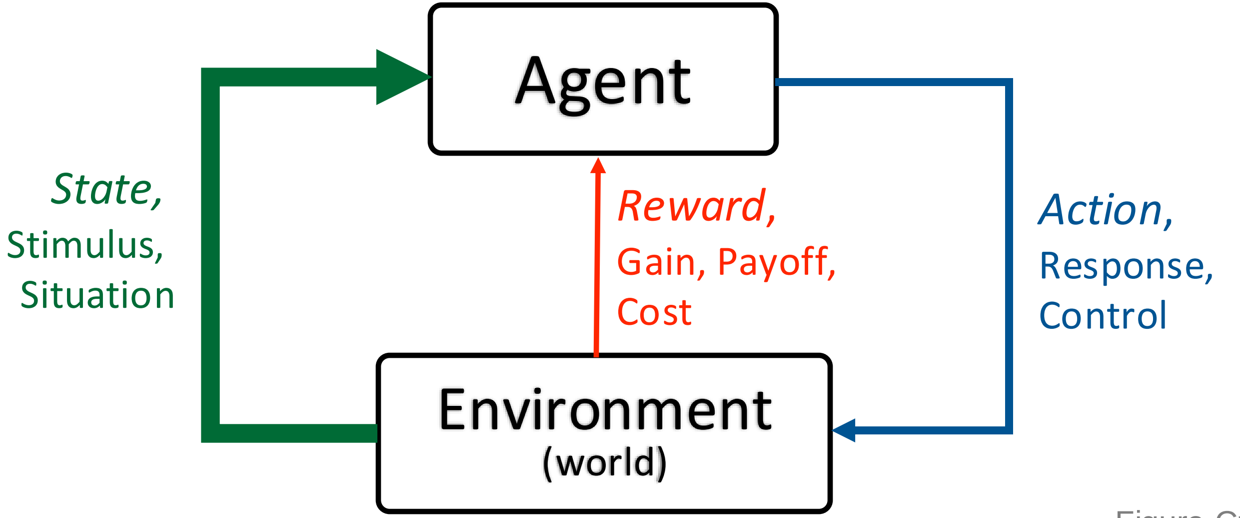
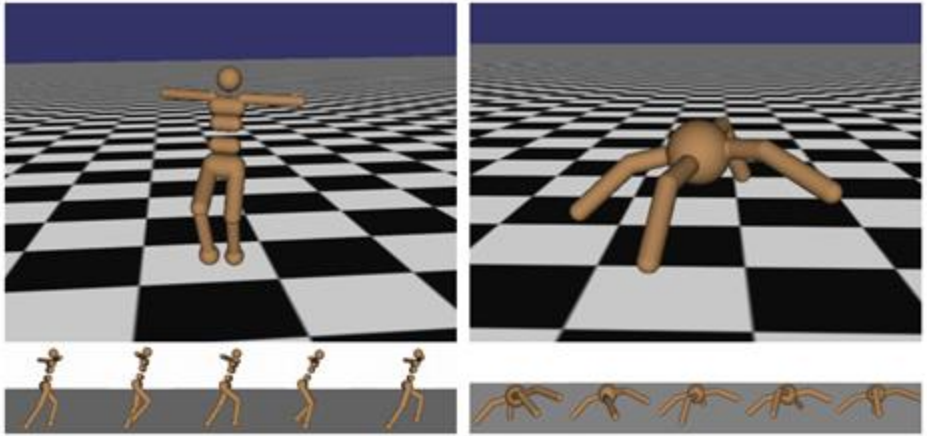


Figure Credit: Rich Sutton

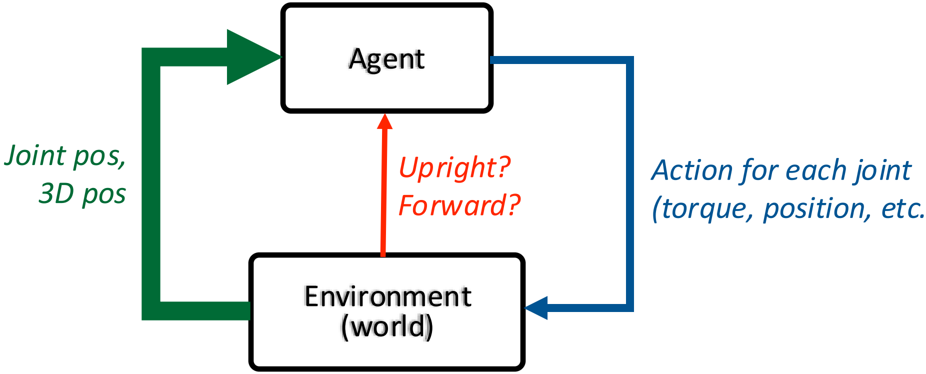
- **Environment** may be unknown, non-linear, stochastic and complex.
- **Agent** learns a **policy** to map states of the environments to actions.
  - Seeking to maximize cumulative reward in the long run.

Example: Robot Locomotion



Figures copyright John Schulman et al., 2016. Reproduced with permission.

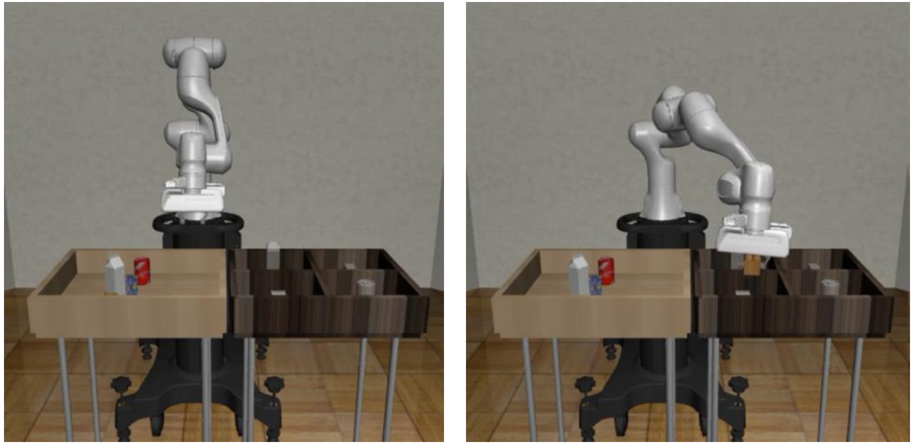
- **Objective**: Make the robot move forward without falling
- **State**: Angle and position of the joints
- **Action**: Torques applied on joints
- **Reward**: +1 at each time step upright and moving forward



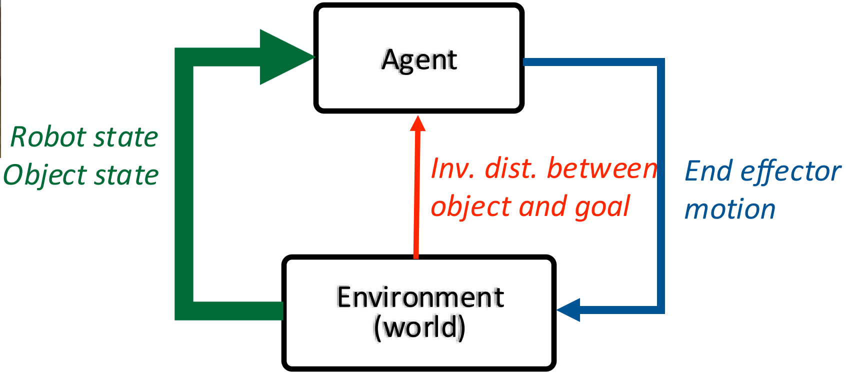
Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Examples of RL tasks

Example: Robot Manipulation



- **Objective**: Pick up object and place to sorting bin
- **State**: Pose of the object and the bin, joint state and velocity of robots
- **Action**: End effector motion
- **Reward**: inverse distance between the object and the bin

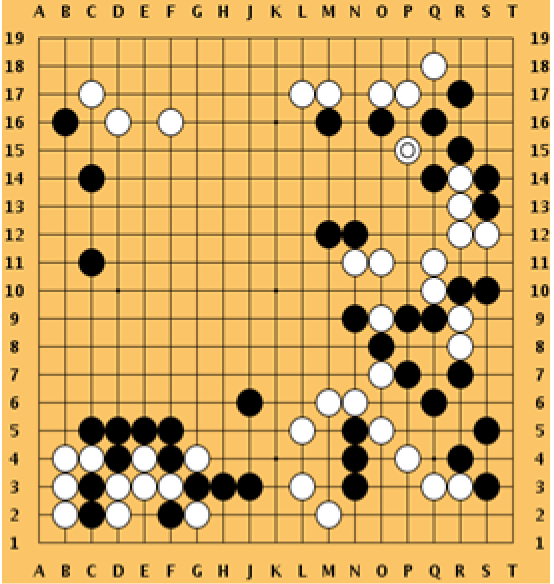


Example: Atari Games

- **Objective**: Complete the game with the highest score
- **State**: Raw pixel inputs of the game state
- **Action**: Game controls e.g. Left, Right, Up, Down
- **Reward**: Score increase/decrease at each time step

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Example: Go



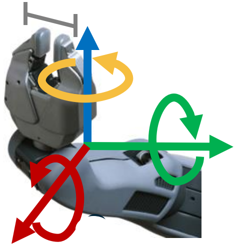
- **Objective**: Defeat opponent
- **State**: Board pieces
- **Action**: Where to put next piece down
- **Reward**: +1 if win at the end of game, 0 otherwise

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Deep Learning for Decision Making

state
input

Deep
Neural Nets



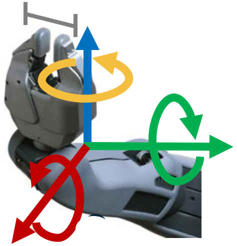
action
output

Deep Learning for Decision Making

Deep Learning for Decision Making

state
input

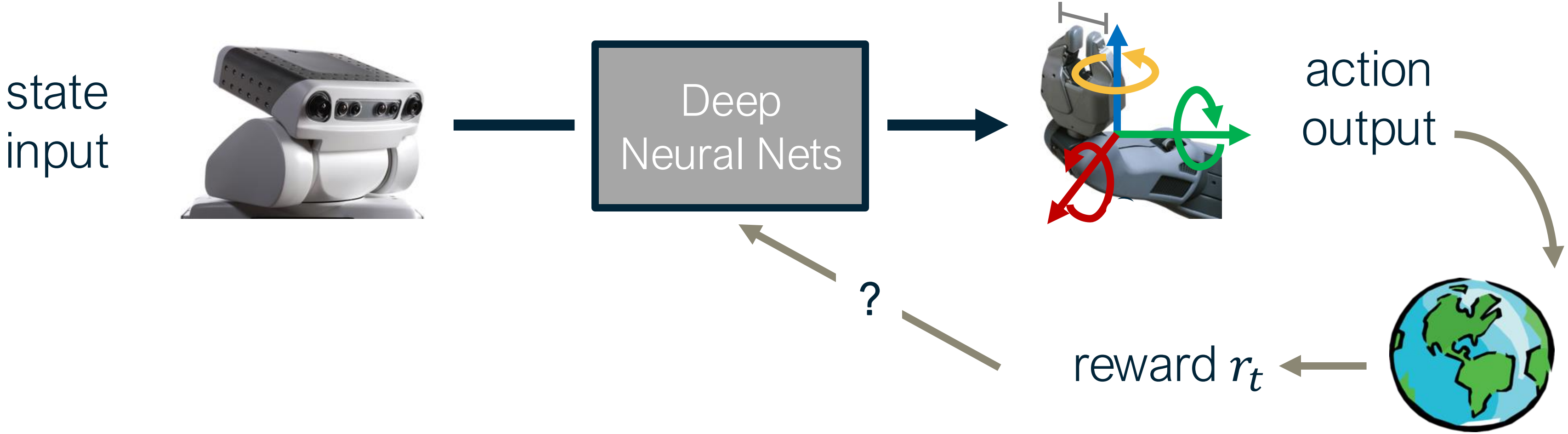
Deep
Neural Nets



action
output

Problem: we don't know the correct action label to supervise the output!

Deep Learning for Decision Making



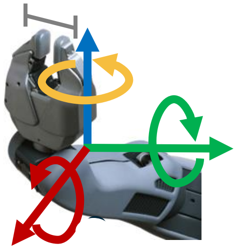
Problem: we don't know the correct action label to supervise the output!

All we know is the step-wise task reward

Deep Learning for Decision Making

state input

Deep Neural Nets



action output

?

reward r_t

How do we pose the learning problem?
(Deep) Reinforcement Learning!

Problem: we don’t know the correct action label to supervise the output!

All we know is the step-wise task reward

Markov Decision Processes

- **MDPs**: Theoretical framework underlying RL

- **MDPs**: Theoretical framework underlying RL
- An MDP is defined as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{T}, \gamma)$
  - $\mathcal{S}$: Set of possible states
  - $\mathcal{A}$: Set of possible actions
  - $\mathcal{R}(s, a, s')$: Distribution of reward
  - $\mathbb{T}(s, a, s')$: Transition probability distribution, also written as $p(s'|s, a)$
  - γ : Discount factor

Markov Decision Processes (MDPs)

- **MDPs**: Theoretical framework underlying RL
- An MDP is defined as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{T}, \gamma)$
  - $\mathcal{S}$: Set of possible states
  - $\mathcal{A}$: Set of possible actions
  - $\mathcal{R}(s, a, s')$: Distribution of reward
  - $\mathbb{T}(s, a, s')$: Transition probability distribution, also written as $p(s'|s, a)$
  - γ : Discount factor
- **Experience**: $\ldots s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, r_{t+2}, s_{t+2}, \ldots$

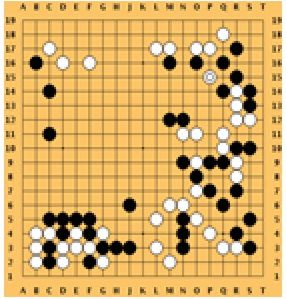
- **MDPs**: Theoretical framework underlying RL
- An MDP is defined as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{T}, \gamma)$
  - $\mathcal{S}$: Set of possible states
  - $\mathcal{A}$: Set of possible actions
  - $\mathcal{R}(s, a, s')$: Distribution of reward
  - $\mathbb{T}(s, a, s')$: Transition probability distribution, also written as $p(s'|s, a)$
  - γ : Discount factor
- **Experience**: $\ldots s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, r_{t+2}, s_{t+2}, \ldots$
- **Markov property**: Current state completely characterizes state of the environment
- **Assumption**: Most recent observation is a sufficient statistic of history

$$p\left(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, \ldots S_0 = s_0\right) = p\left(S_{t+1} = s' | S_t = s_t, A_t = a_t\right)$$

Markov Decision Processes (MDPs)

Fully observed MDP

- Agent receives the true state s_t at time t
- Example: Chess, Go



Partially observed MDP

- Agent perceives its own partial observation o_t of the state s_t at time t, using past states e.g. with an RNN
- Example: Poker, First-person games (e.g. Doom)

Source: https://github.com/mwydmuch/ViZDoom

MDP Variations

Georgia Tech

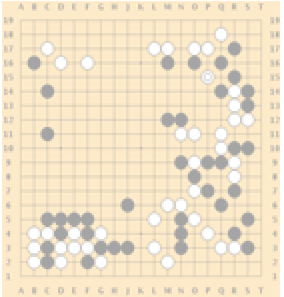
Fully observed MDP

- Agent receives the true state s_t at time t
- Example: Chess, Go

Partially observed MDP

- Agent perceives its own partial observation o_t of the state s_t at time t, using past

We will assume **fully observed MDPs** for this lecture



Source: https://github.com/mwydmuch/ViZDoom

MDP Variations

- In **Reinforcement Learning**, we assume an underlying **MDP** with unknown:
  - Transition probability distribution $\mathbb{T}$
  - Reward distribution $\mathcal{R}$

MDP
$(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{T}, \gamma)$

- In **Reinforcement Learning**, we assume an underlying **MDP** with unknown:
  - Transition probability distribution $\mathbb{T}$
  - Reward distribution $\mathcal{R}$

MDP
$(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{T}, \gamma)$

Put simply: without learning, the agent **doesn't know** how their actions will change the environment and what reward they will receive.

Reinforcement Learning is to learn to act optimally given experience data (transition, reward) from interacting with the environments.

The outcome is a control policy $\pi(a|s)$ that maps a state s to a (good) action a

MDPs in the context of RL

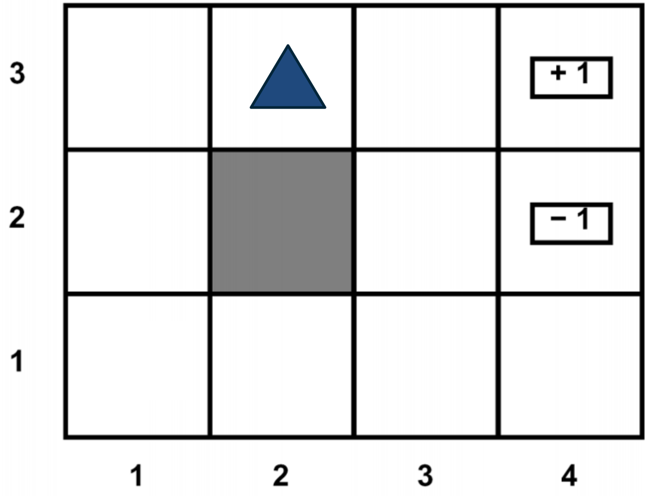


Figure credits: Pieter Abbeel

A Grid World MDP

- Agent lives in a 2D grid environment

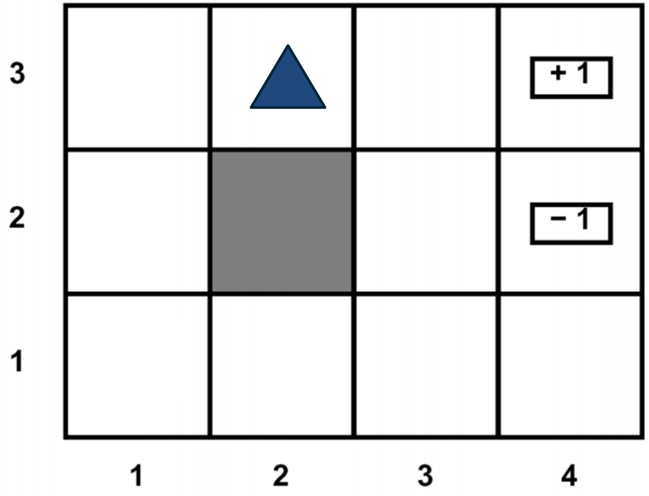


Figure credits: Pieter Abbeel

A Grid World MDP

- Agent lives in a 2D grid environment

- State: Agent's 2D coordinates
- Actions: N, E, S, W
- Rewards: +1/-1 at absorbing states

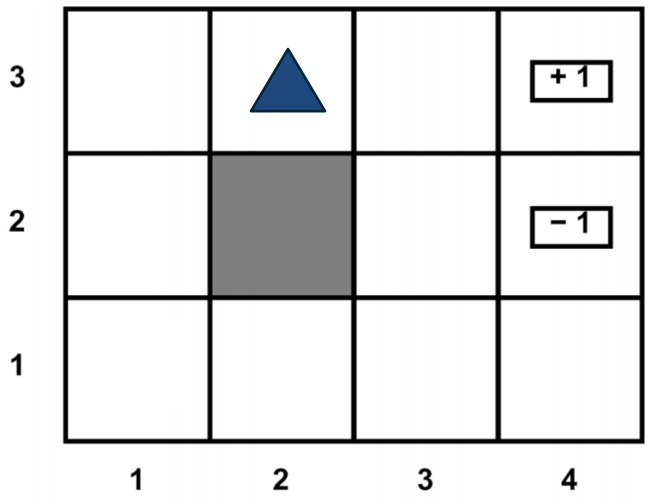


Figure credits: Pieter Abbeel

A Grid World MDP

- Agent lives in a 2D grid environment

- State: Agent's 2D coordinates
- Actions: N, E, S, W
- Rewards: +1/-1 at absorbing states

- Walls block agent's path
- Actions to not always go as planned $T(s, a, s')$
  - 20% chance that agent drifts one cell left or right of direction of motion (except when blocked by wall).

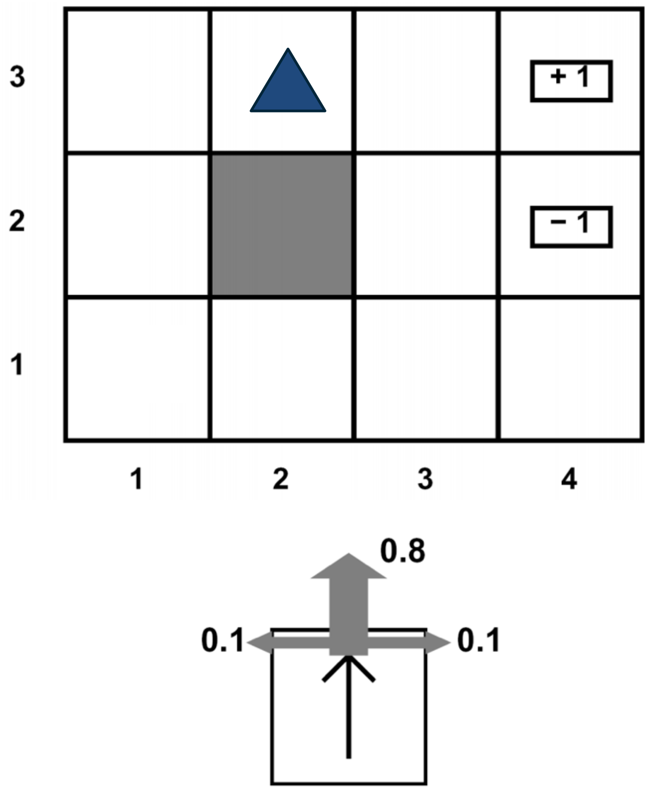


Figure credits: Pieter Abbeel

A Grid World MDP

- Solving MDPs by finding the **best/optimal policy**

- Solving MDPs by finding the **best/optimal policy**

- Formally, a **policy** is a mapping from states to actions

e.g.

State	Action
A →	2
B →	1

Solving MDPs: Optimal policy

- Solving MDPs by finding the **best/optimal policy**

- Formally, a **policy** is a mapping from states to actions
  - Deterministic $\pi(s) = a$

Solving MDPs: Optimal policy

- Solving MDPs by finding the **best/optimal policy**

- Formally, a **policy** is a mapping from states to actions
  - Deterministic $\pi(s) = a$
  - Stochastic $\pi(a|s) = \mathbb{P}(A_t = a | S_t = s)$

Solving MDPs: Optimal policy

- Solving MDPs by finding the **best/optimal policy**

- Formally, a **policy** is a mapping from states to actions
  - Deterministic $\pi(s) = a$
  - Stochastic $\pi(a|s) = \mathbb{P}(A_t = a|S_t = s)$

- What is a good policy?
  - Maximize **current reward**? Sum of all **future rewards**?

Solving MDPs: Optimal policy

- Solving MDPs by finding the **best/optimal policy**

- Formally, a **policy** is a mapping from states to actions
  - Deterministic $\pi(s) = a$
  - Stochastic $\pi(a|s) = \mathbb{P}(A_t = a | S_t = s)$

- What is a good policy?
  - Maximize **current reward**? Sum of all **future rewards**?
  - **Discounted sum of future rewards**!
  - Future is inherently uncertain!

Solving MDPs: Optimal policy

- Solving MDPs by finding the **best/optimal policy**

- Formally, a **policy** is a mapping from states to actions
  - Deterministic $\pi(s) = a$
  - Stochastic $\pi(a|s) = \mathbb{P}(A_t = a | S_t = s)$

Small γ → near-sighted

Large γ → far-sighted

- What is a good policy?
  - Maximize **current reward**? Sum of all **future rewards**?
  - **Discounted sum of future rewards**!
  - Future is inherently uncertain!
  - How much to value future rewards
    - Discount factor: γ
    - Typically 0.9 - 0.99

- Formally, the **optimal policy** is defined as:

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi\right]$$

- Formally, the **optimal policy** is defined as:

discounted sum of future rewards

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi\right]$$

Solving MDPs: Optimal policy

- Formally, the **optimal policy** is defined as:

discounted sum of future rewards

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | \pi\right]$$

?

Solving MDPs: Optimal policy

- Formally, the **optimal policy** is defined as:

discounted sum of future rewards

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi\right]$$

$$\boxed{s_0 \sim p(s_0), a_t \sim \pi(\cdot|s_t), s_{t+1} \sim p(\cdot|s_t, a_t)}$$

Expectation over initial state, actions from policy,
next states from transition distribution

We need a function to quantify the optimality of a policy!

- A **value function** predicts the sum of discounted future reward **for a given policy**

Value Function

- A **value function** predicts the sum of discounted future reward **for a given policy**

- **State** value function / **V**-function / $V : \mathcal{S} \rightarrow \mathbb{R}$
  - How good is this state?
  - Am I likely to win/lose the game from this state (reward-to-go)?

Value Function

Georgia Tech

- A **value function** predicts the sum of discounted future reward **for a given policy**

- **State** value function / **V**-function / $V : \mathcal{S} \rightarrow \mathbb{R}$
  - How good is this state under a policy?
  - Am I likely to win/lose the game from this state (reward-to-go)?

- **State-Action** value function / **Q**-function / $Q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$
  - How good is this state-action pair under a policy?
  - In this state, what is the impact of this action on my future?

Value Function

- A **value function** predicts the sum of discounted future reward **for a given policy**

- **State** value function / **V**-function / $V : \mathcal{S} \to \mathbb{R}$
  - How good is this state under a policy?
  - Am I likely to win/lose the game from this state (reward-to-go)?

- **State-Action** value function / **Q**-function / $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$
  - How good is this state-action pair under a policy?
  - In this state, what is the impact of this action on my future?

Value functions are measuring both the quality of a state (state-action pair) and the quality of a policy!

Value Function

- For a policy that produces a trajectory sample $(s_0, a_0, s_1, a_1, s_2 \cdots)$

- For a policy that produces a trajectory sample $(s_0, a_0, s_1, a_1, s_2 \cdots)$

- The **V-function** of the policy at state s, is the expected cumulative reward from state s:

$$V^\pi(s) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi\right]$$

Value Function

- For a policy that produces a trajectory sample $(s_0, a_0, s_1, a_1, s_2 \cdots)$

- The **V-function** of the policy at state s, is the expected cumulative reward from state s:

$$V^\pi(s) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi\right]$$

$$\boxed{\boldsymbol{s}_0 \sim p(\boldsymbol{s}_0), a_t \sim \pi(\cdot|\boldsymbol{s}_t), \boldsymbol{s}_{t+1} \sim p(\cdot|\boldsymbol{s}_t, a_t)}$$

Value Function

- For a policy that produces a trajectory sample $(s_0, a_0, s_1, a_1, s_2 \cdots)$

- The **Q-function** of the policy at state **s** and action **a**, is the expected cumulative reward upon taking action **a** in state **s** (and following policy thereafter):

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

$$\boxed{\boldsymbol{s}_0 \sim p(\boldsymbol{s}_0), a_t \sim \pi(\cdot|\boldsymbol{s}_t), \boldsymbol{s}_{t+1} \sim p(\cdot|\boldsymbol{s}_t, a_t)}$$

How do we learn a good policy?

Action-Value Function

- The V and Q functions corresponding to **the optimal policy** $\pi^{\star}$

$$V^*(s) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, \pi^*\right]$$

$$Q^*(s,a) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi^*\right]$$

$$\boxed{V^*(s) = \max_a Q^*(s,a)}$$

Optimal policy from Q value function: $\boxed{\pi^*(s) = \arg\max_a Q^*(s,a)}$

Optimal V & Q functions

- The V and Q functions corresponding to **the optimal policy** $\pi^\star$

$$V^*(s) = \mathbb{E}\left[\sum_{t\geq 0}\gamma^t r_t | s_0 = s, \pi^*\right]$$

$$Q^*(s,a) = \mathbb{E}\left[\sum_{t\geq 0}\gamma^t r_t | s_0 = s, a_0 = a, \pi^*\right]$$

$$\boxed{V^*(s) = \max_a Q^*(s,a)}$$

How do we learn the value functions?

Optimal policy from Q value function: $\boxed{\pi^*(s) = \arg\max_a Q^*(s,a)}$

Optimal V & Q functions

Bellman equation:

$$V^*(s) = \max_a \sum_{s'} p\left(s'|s,a\right)\left[r(s,a) + \gamma V^*\left(s'\right)\right]$$

Bellman equation:

$$\underbrace{V^*(s)}_{\text{Value of a given state}} = \underbrace{\max_a}_{\text{If we act optimally}} \underbrace{\sum_{s'} p(s'|s,a)}_{\text{Expectation over all possible next states if taking action } a} [\underbrace{r(s,a)}_{\text{Reward if taking action } a \text{ at current state}} + \underbrace{\gamma V^*(s')}_{\text{Discounted future value}}]$$

Bellman equation:

$$V^*(s) = \max_a \underbrace{\sum_{s'} p(s'|s,a)}_{} [r(s,a) + \gamma V^*(s')]$$

Value of a given state

If we act optimally

Expectation over all possible next states if taking action a

Reward if taking action a at current state

Discounted future value

Bellman equation: the optimal value of a state equals to the immediate reward plus discounted future rewards, when acting optimally

Bellman Optimality Equations

Bellman equation:

$$V^*(s) = \max_a \underbrace{\sum_{s'} p(s'|s,a)}_{} \left[r(s,a) + \gamma V^*(s')\right]$$

Value of a given state

If we act optimally

Expectation over all possible next states if taking action a

Reward if taking action a at current state

Discounted future value

Bellman equation: the optimal value of a state equals to the immediate reward plus discounted future rewards, when acting optimally

Can we use this equation to construct a learning algorithm of V*?

Bellman equation:

$$V^*(s) = \max_a \sum_{s'} p(s'|s,a)\left[r(s,a) + \gamma V^*(s')\right]$$

Goal: Learn a value function V that correctly maps states to optimal values.

Bellman Optimality Equations

Bellman equation:

$$V^*(s) = \max_a \sum_{s'} p(s'|s,a)\left[r(s,a) + \gamma V^*(s')\right]$$

Goal: Learn a value function V that correctly maps states to optimal values.

Facts:

- If a value function V is correct, then this equation should hold exactly.

Bellman equation:

$$V^*(s) = \max_a \sum_{s'} p(s'|s,a)\left[r(s,a) + \gamma V^*(s')\right]$$

Goal: Learn a value function V that correctly maps states to optimal values.

Facts:

- If a value function V is correct, then this equation should hold exactly.
- If the value function is incorrect, we can use this equation to update the value estimate.

Bellman Optimality Equations

Bellman equation:

$$V^*(s) = \max_a \sum_{s'} p(s'|s,a)\left[r(s,a) + \gamma V^*(s')\right]$$

Goal: Learn a value function V that correctly maps states to optimal values.

Facts:

- If a value function V is correct, then this equation should hold exactly.
- If the value function is incorrect, we can use this equation to update the value estimate.

$$\boxed{V^{i+1}(s) \leftarrow \max_a \sum_{s'} p(s'|s,a)\left[r(s,a) + \gamma V^i(s')\right]}$$

Value Iteration

Bellman Optimality Equations

$$V^{i+1}(s) \leftarrow \max_a \sum_{s'} p(s'|s,a) \left[r(s,a) + \gamma V^i(s') \right]$$

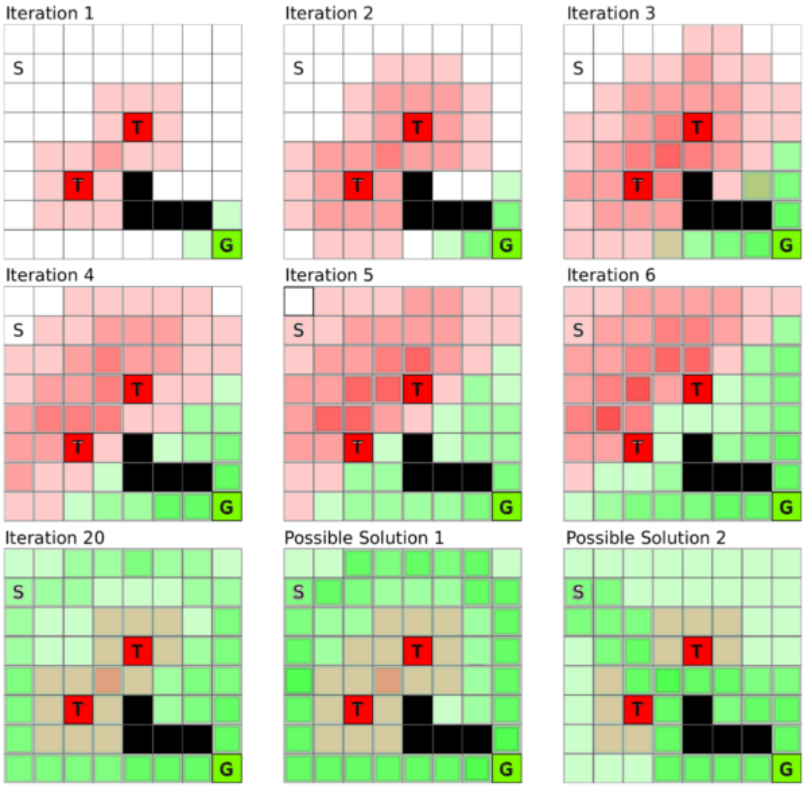
Initialize Value Function table

For each iteration i:

- For each state s:
  - For each action a:
    - Get reward $r(s,a)$
    - For each possible future states s':
      - Get current $V(s')$ from table
    - Compute the expectation term
  - Select the highest future value
  - Update new $V(s)$

This algorithm looks familiar …

It's dynamic programming!



https://developer.nvidia.com/blog/deep-learning-nutshell-reinforcement-learning/

Value Iteration

Algorithm: Value Iteration

- Initialize values of all states to arbitrary values, e.g., all 0's.
- While not converged:
  - For each state:

$$\boxed{V^{i+1}(s) \leftarrow \max_a \sum_{s'} p(s'|s,a)\left[r(s,a) + \gamma V^i(s')\right]}$$

- Repeat until convergence (no change in values)

$$V^0 \rightarrow V^1 \rightarrow V^2 \rightarrow \quad \cdots \rightarrow V^i \rightarrow \ldots \quad \rightarrow V^*$$

Q: What's the time complexity per iteration?

Time complexity per iteration $O(|\mathcal{S}|^2|\mathcal{A}|)$

Value Iteration Update:

$$V^{i+1}(s) \leftarrow \max_a \sum_{s'} p(s'|s,a)\left[r(s,a) + \gamma V^i(s')\right]$$

Q-Iteration Update:

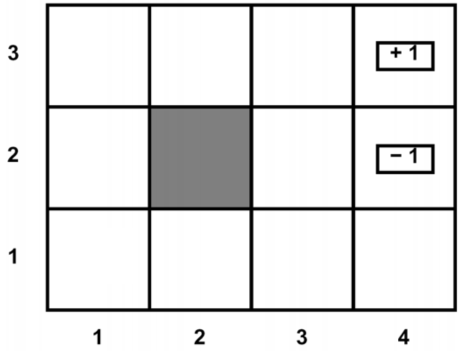
$$Q^{i+1}(s,a) \leftarrow \sum_{s'} p(s'|s,a)\left[r(s,a) + \gamma \max_{a'} Q^i(s',a')\right]$$

Given a learned Q function, we can derive the *optimal policy*:

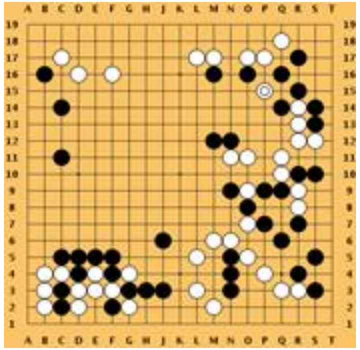
$$\pi(s) = argmax_a Q(s,a)$$

Value iteration is almost never used in practice!

$\boxed{\text{Time complexity per iteration } O(|\mathcal{S}|^2|\mathcal{A}|)}$



$|S| = 11, |A| = 4$



$|S| \cong 3^{361}, |A| \cong 361$

$|S| \cong ?, |A| = ?$

Can't iterate over all (s, a) pairs -> need approximation!

We also don't know the transition function (model) -> need a *model-free* method!

Q-Learning

- We'd like to do Q-value updates to each Q-state:

$$Q'(s_t, a_t) \cong \sum_{s'} T(s_{t+1}|s_t, a_t)[r_t + \gamma \max_a Q(s_{t+1}, a)]$$

  - But can't compute this update without knowing the transition function and enumerate all possible next states s'!

- Instead, approximate the expectation (sum over next states) with (lots of) experience samples
  - Take an action in the environment following *policy* $\mathrm{argmax}_{\mathrm{a}} Q(s, a)$
  - receive a sample transition (s_t, a_t, r_t, s_{t+1})
  - This sample suggests: $Q(s_t, a_t) \cong r_t + \gamma \max_a Q(s_{t+1}, a)$
  - Keep a running average to approximate the expectation:

$$Q'(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a)]$$

Old estimates

New estimates

Q-Learning

Approximate the expectation (sum over next states) with (lots of) experience samples

- Take an action in the environment following *policy* $\mathrm{argmax}_{\mathrm{a}} Q(s, a)$
- receive a sample transition (s_t, a_t, r_t, s_{t+1})
- This sample suggests: $Q(s_t, a_t) \cong r_t + \gamma \max_a Q(s_{t+1}, a)$
- Keep a running average to approximate the expectation:

$$Q'(s_t, a_t) = (1 - \alpha) Q(s_t, a_t) + \alpha [r_t + \gamma \max_a Q(s_{t+1}, a)]$$

- We can now learn Q values without having access to a transition model
- Getting experience data through interaction instead of assuming access to all states: more practical in real-world situation (e.g., robots learning through trial-and-error)
- Still need to represent all (s, a) pairs in a Q value table!

Q-Learning

Idea: represent the Q value table as a **parametric function** $Q_\theta(s, a)$!

How do we learn the function? We need a **loss metric**!

$$
\begin{aligned}
Q'(s_t, a_t) &= (1-\alpha)Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a)] \\
&= Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))
\end{aligned}
$$

Now, at optimum, $Q(s_t, a_t) = Q'(s_t, a_t) = Q^*(s_t, a_t)$; This gives us:

$$
0 = 0 + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))
$$

Learning problem:

$$
\mathrm{argmin}_\theta || \underbrace{r_t + \gamma \max_a Q_\theta(s_{t+1}, a)}_{\text{Target Q value}} - Q_\theta(s_t, a_t)) ||
$$

How to model Q?

Deep Q-Learning

- **Q-Learning** with **linear function approximators**

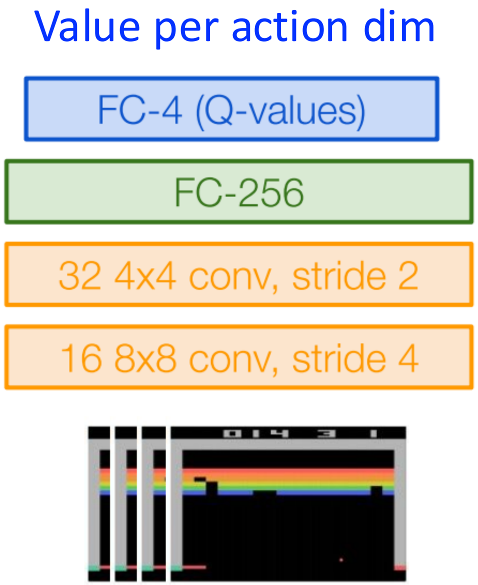
$$Q(s, a; w, b) = w_a^\top s + b_a$$

  - Has some theoretical guarantees

- **Deep Q-Learning**: Fit a **deep Q-Network**

$$Q(s, a; \theta)$$

  - Works well in practice
  - Q-Network can take arbitrary input (e.g. RGB images)
  - Assume discrete action space (e.g., left, right)



Deep Q-Learning

- Assume we have collected a dataset:

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

- We want a Q-function that satisfies bellman optimality (Q-value)

$$\boxed{Q^*(s,a) = \mathop{\mathbb{E}}_{s' \sim p(s'|s,a)} \left[r(s,a) + \gamma \max_{a'} Q^*(s', a') \right]}$$

- Loss for a single data point:

$$\text{MSE Loss} := \Big(\underbrace{Q_{new}(s,a)}_{\text{Predicted Q-Value}} - \underbrace{(r + \gamma \max_{a} Q_{old}(s', a))}_{\text{Target Q-Value}} \Big)^2$$

Deep Q-Learning

- Minibatch of $\{(s, a, s', r)_i\}_{i=1}^{B}$

- Forward pass: State $\rightarrow$ Q-Network $\rightarrow$ Q-Values per action

  $B \times D$ $\qquad$ $B \times n_{actions}$

- Compute loss:

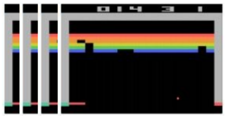
$$\Big(\underbrace{Q_{new}(s,a)}_{\theta_{new}} - (r + \gamma \max_a \underbrace{Q_{old}(s',a)}_{\theta_{old}})\Big)^2$$

- Backward pass:

$$\frac{\partial Loss}{\partial \theta_{new}}$$

Q-Network:
- FC-4 (Q-values)
- FC-256
- 32 4x4 conv, stride 2
- 16 8x8 conv, stride 4

State:



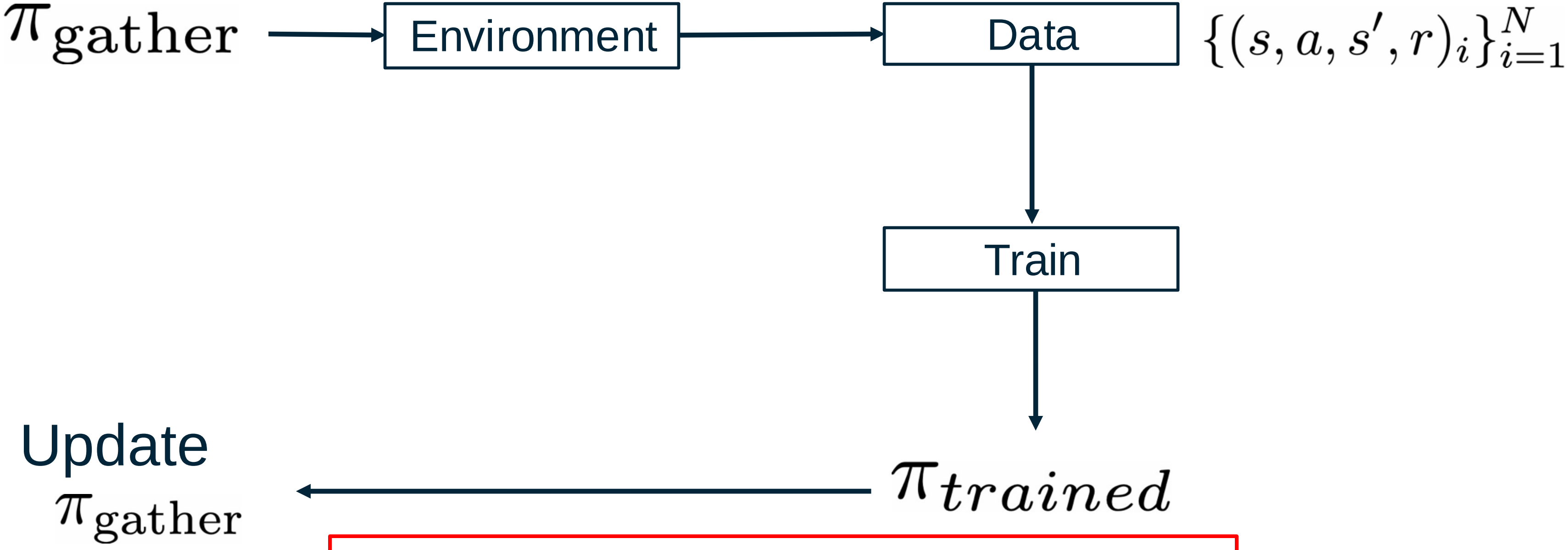
$$\text{MSE Loss} := \left(Q_{new}(s,a) - (r + \max_{a} Q_{old}(s',a))\right)^2$$

- We don't want the policy to change its behavior too frequently
  - Freeze Q_{old} and update Q_{new} parameters
  - Set $Q_{old} \leftarrow Q_{new}$ at regular intervals or update as running average
    - $\theta_{old} = \beta\theta_{old} + (1-\beta)\theta_{new}$

How to gather experience?

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

This is why RL is hard



Challenge 1: Exploration vs Exploitation

Challenge 2: Non iid, highly correlated data

How to gather experience?

- What should π_{gather} be?
  - Greedy? -> no exploration, always choose the most confident action

    $$\arg\max_a Q(s, a; \theta)$$

- An exploration strategy:
  - ϵ-greedy

$$a_t = \begin{cases} \arg\max_a Q(s, a) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases}$$

- Samples are correlated => high variance gradients => **inefficient learning**
- Current Q-network parameters determines next training samples => can lead to **bad feedback loops**
  - e.g. if maximizing action is to move right, training samples will be dominated by samples going right, may fall into local minima

Correlated Data Problem

- Correlated data: addressed by using **experience replay**
  - A replay buffer stores transitions (s, a, s', r)
  - Continually update replay buffer as game (experience) episodes are played, older samples discarded
  - Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples
- Larger the buffer, lower the correlation

Experience Replay

Deep Q-Learning Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    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 $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Experience Replay

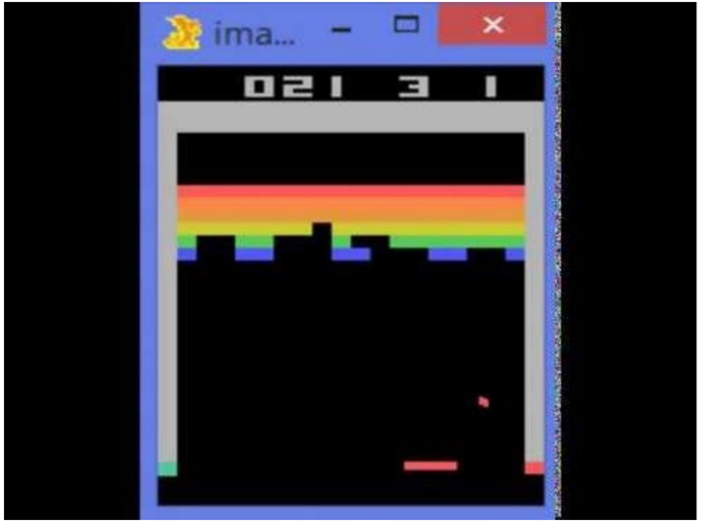
Epsilon-greedy

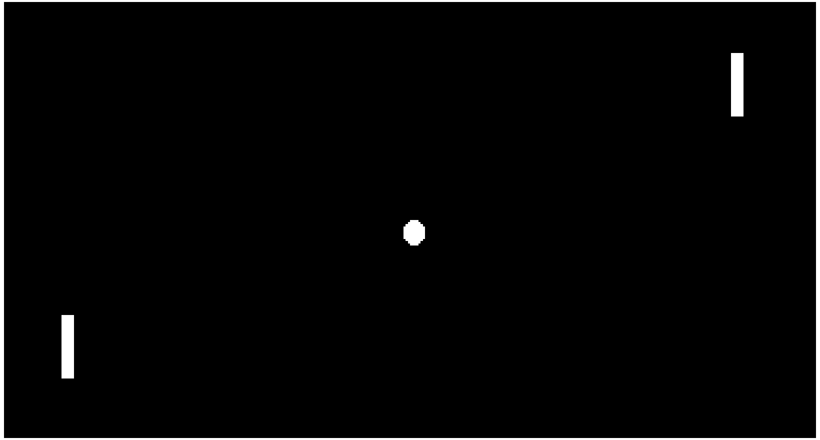
Q Update

Atari Games

- **Objective**: Complete the game with the highest score
- **State**: Raw pixel inputs of the game state
- **Action**: Game controls e.g. Left, Right, Up, Down
- **Reward**: Score increase/decrease at each time step

Atari Games





https://www.youtube.com/watch?v=V1eYniJ0Rnk

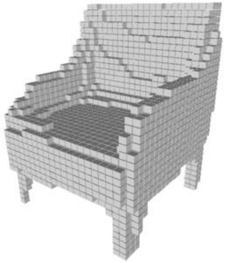
Different RL Paradigms

- **Value-based RL**
  - (Deep) Q-Learning, approximating $Q^*(s, a)$ with a deep Q-network
- **Policy-based RL**
  - Directly approximate optimal policy π^* with a parametrized policy π_θ^*
- **Model-based RL**
  - Approximate transition function $T(s', a, s)$ and reward function $\mathcal{R}(s, a)$
  - Plan by looking ahead in the (approx.) future!

Next Time: RL continued --- Policy Gradient and Actor-Critic

What is an implicit representation for 3D data?

Example: representing a 3D occupancy grid



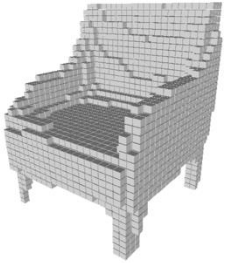
Explicit: A tensor of **3D voxel grid** $V \in \{0,1\}^{[H,W,L]}$

Implicit: A **function** that maps locations to occupancies

$$F_\theta : x, y, z \to \{0,1\}$$

What is an implicit representation for 3D data?

Example: representing a 3D occupancy grid



Explicit: A tensor of **3D voxel grid** $V \in \{0, 1\}^{[H,W,L]}$

Implicit: A **function** that maps locations to occupancies

$$F_\theta : x, y, z \rightarrow \{0, 1\}$$

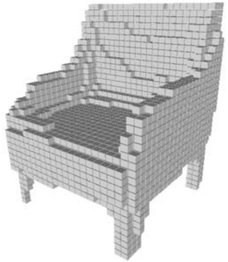
Implicit representation describes 3D shapes using **mathematical functions** rather than explicit voxels, points, or mesh.

Example: Signed Distance Function

$$F_\theta : \mathbb{R}^3 \rightarrow \mathbb{R}$$

What is an implicit representation for 3D data?

Example: representing a 3D occupancy grid



Explicit: A tensor of **3D voxel grid** $V \in \{0,1\}^{[H,W,L]}$

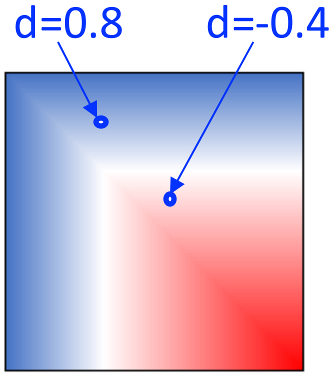
Implicit: A **function** that maps locations to occupancies

$$F_\theta: x, y, z \rightarrow \{0,1\}$$

Implicit representation describes 3D shapes using **mathematical functions** rather than explicit voxels, points, or mesh.
Example: Signed Distance Function

$$F_\theta: \mathbb{R}^N \rightarrow \mathbb{R}$$

Can we represent more than just geometry?



SDF distance map

How far is a point from the nearest surface, and is the point *inside or outside* of the shape?

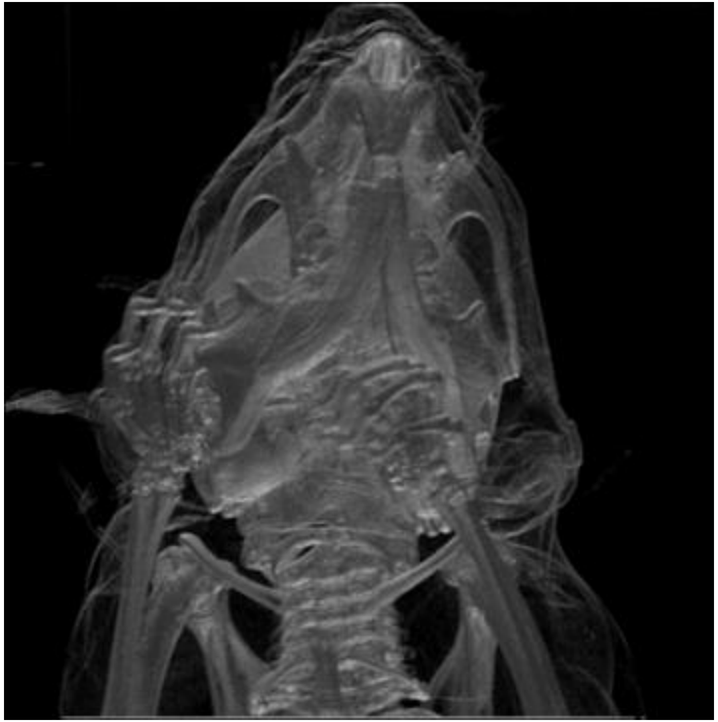
Implicit 3D Representation: Beyond Geometry

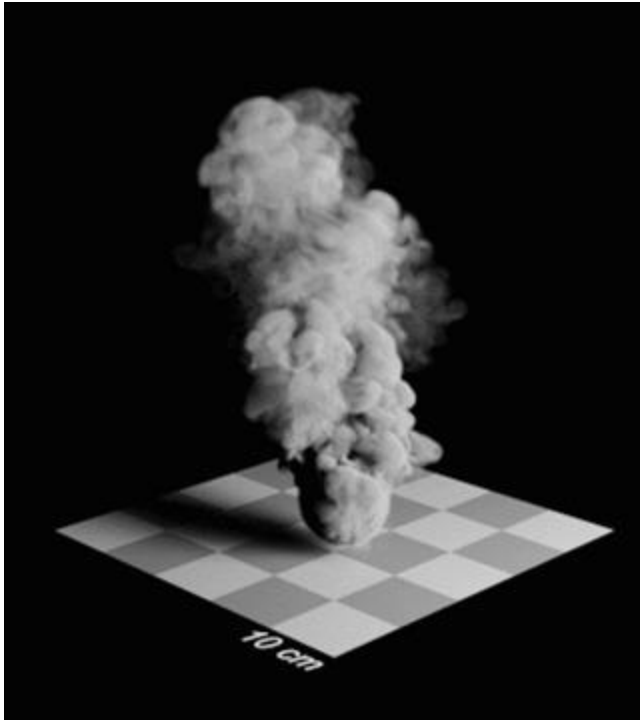
$$f_\theta(viewpoint) = Image$$

Goal: Learn an implicit 3D representation function that maps any camera viewpoint to full RGB images

Can we implicitly represent a full 3D scene, including its fine-grained **geometry** (e.g., surface occupancy) and **appearance**?

Basics: Volume Rendering

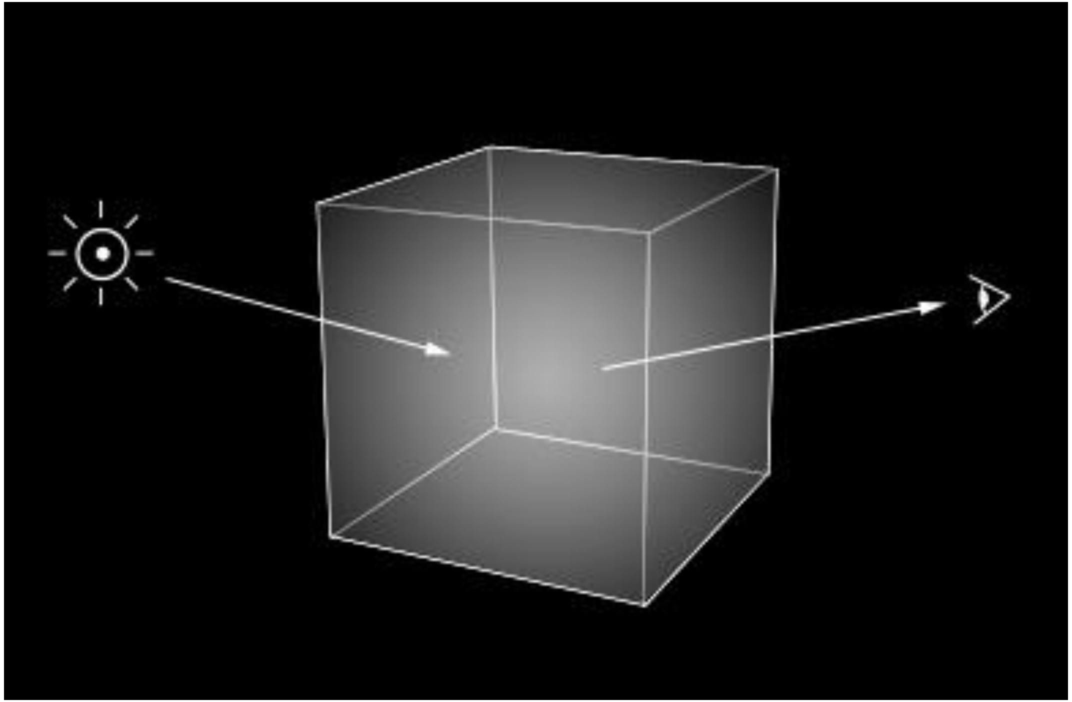




https://en.wikipedia.org/wiki/Volume_rendering

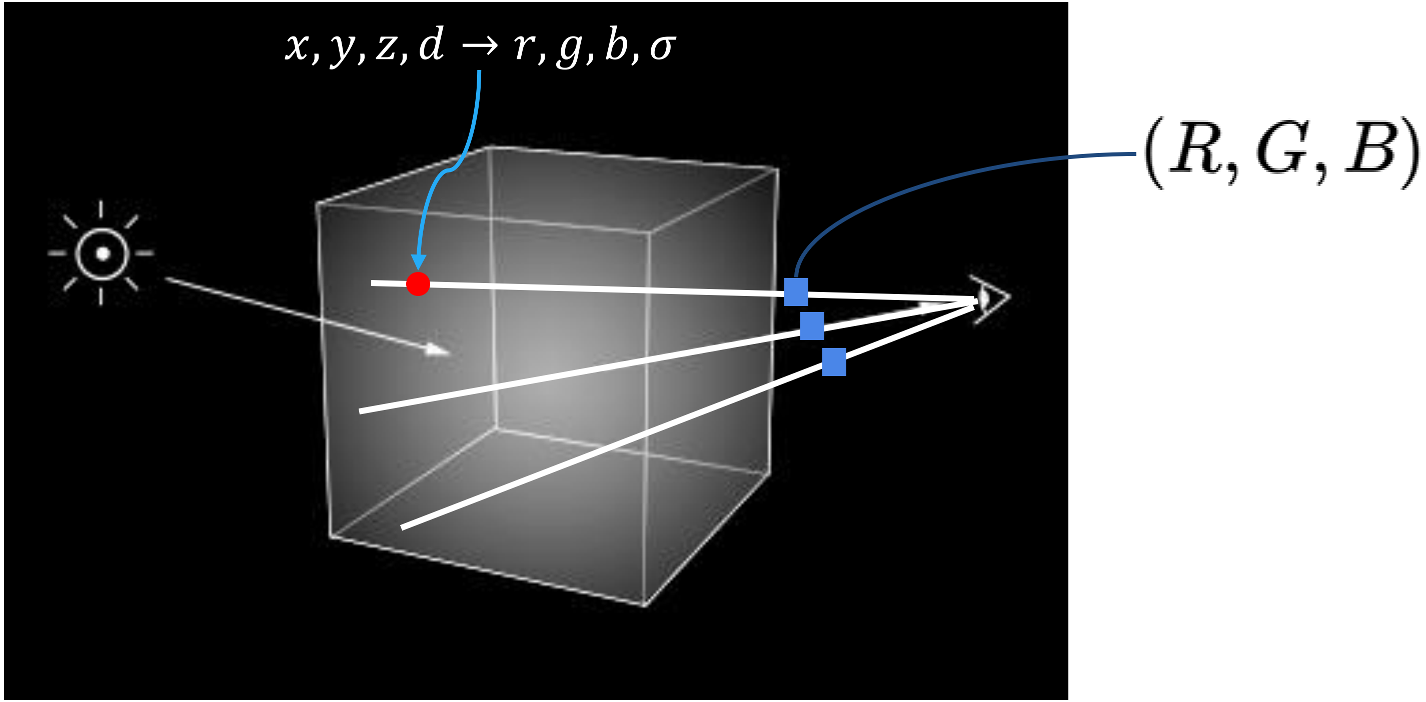
https://coronarenderer.freshdesk.com/support/solutions/articles/12000045276-how-to-use-the-corona-volume-grid-

Volume Rendering: Scene Representation



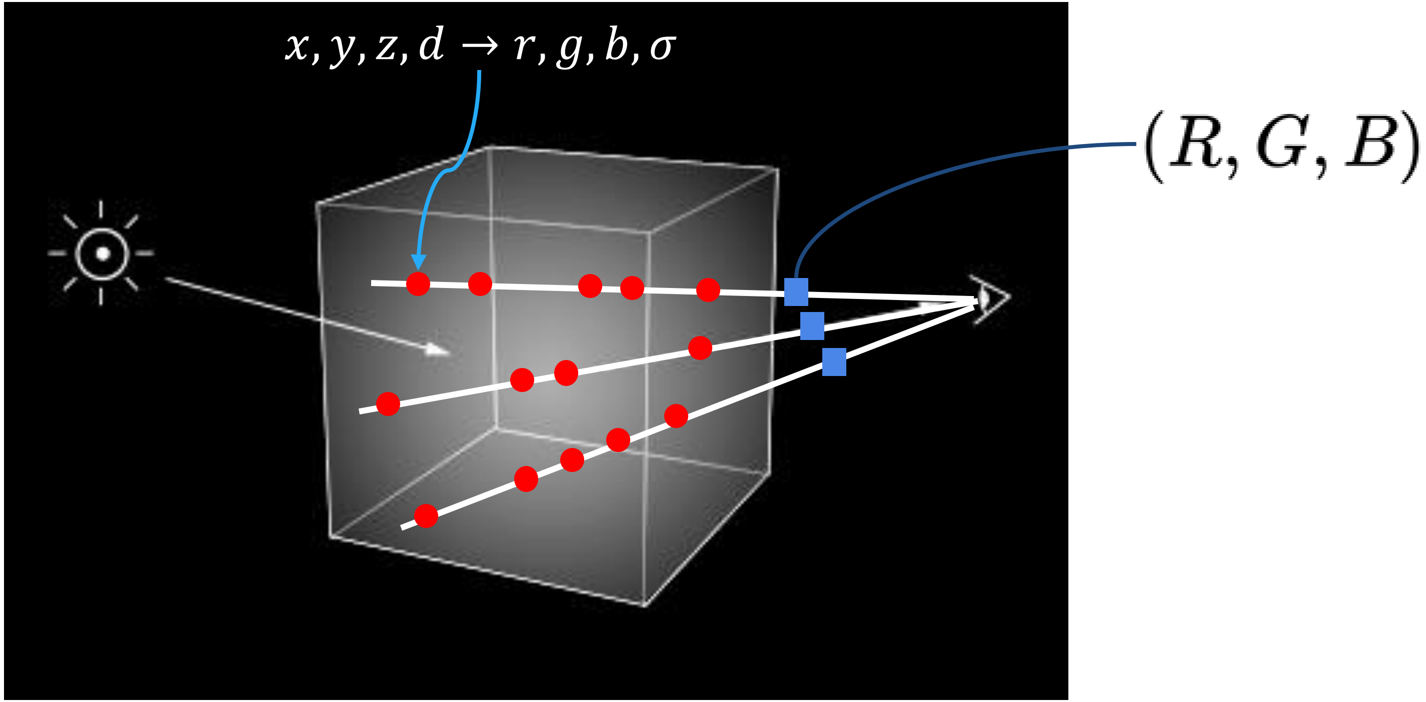
Volume Rendering: Scene Representation

Each location (x, y, z) emits certain color r, g, b when viewed with direction d.
We represent point occupancy continuously as density σ.



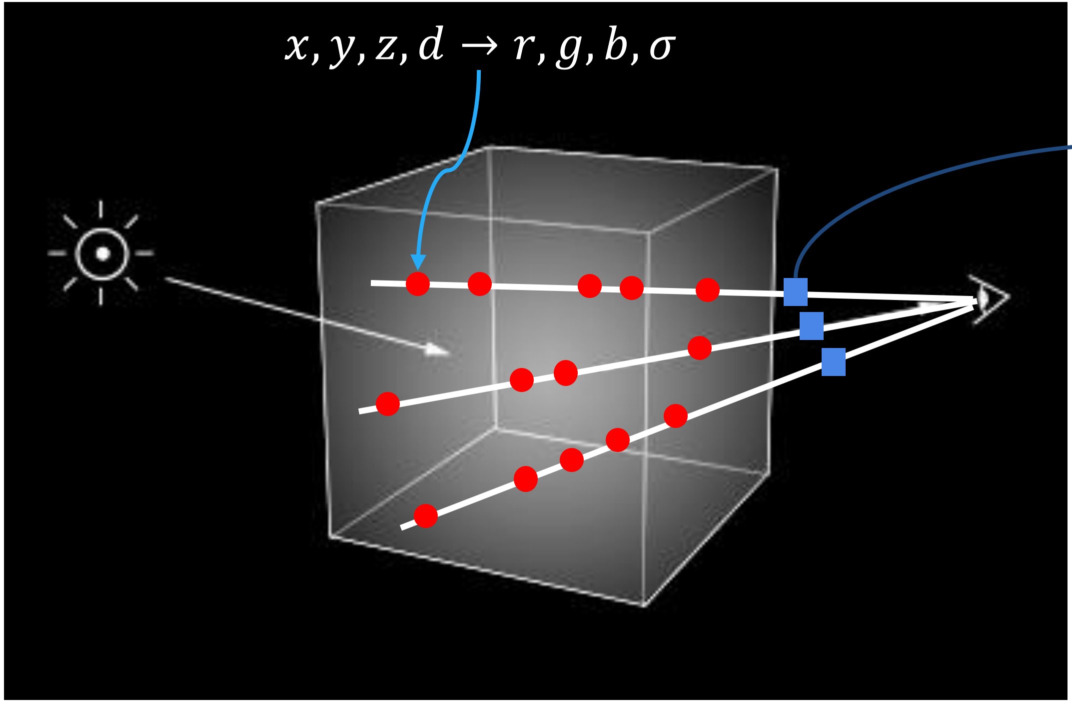
Volume Rendering: Scene Representation

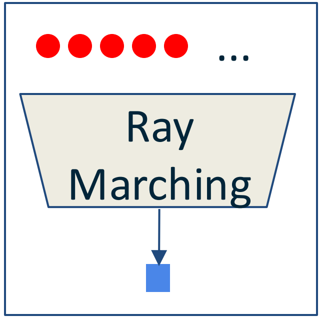
Each location (x, y, z) emits certain color r, g, b when viewed with direction d.
We represent point occupancy continuously as density σ.



Volume Rendering: Ray Marching

Ray Marching: Integrate color and density of points along a ray (via discretization) to render an RGB value. Render many points -> An image!

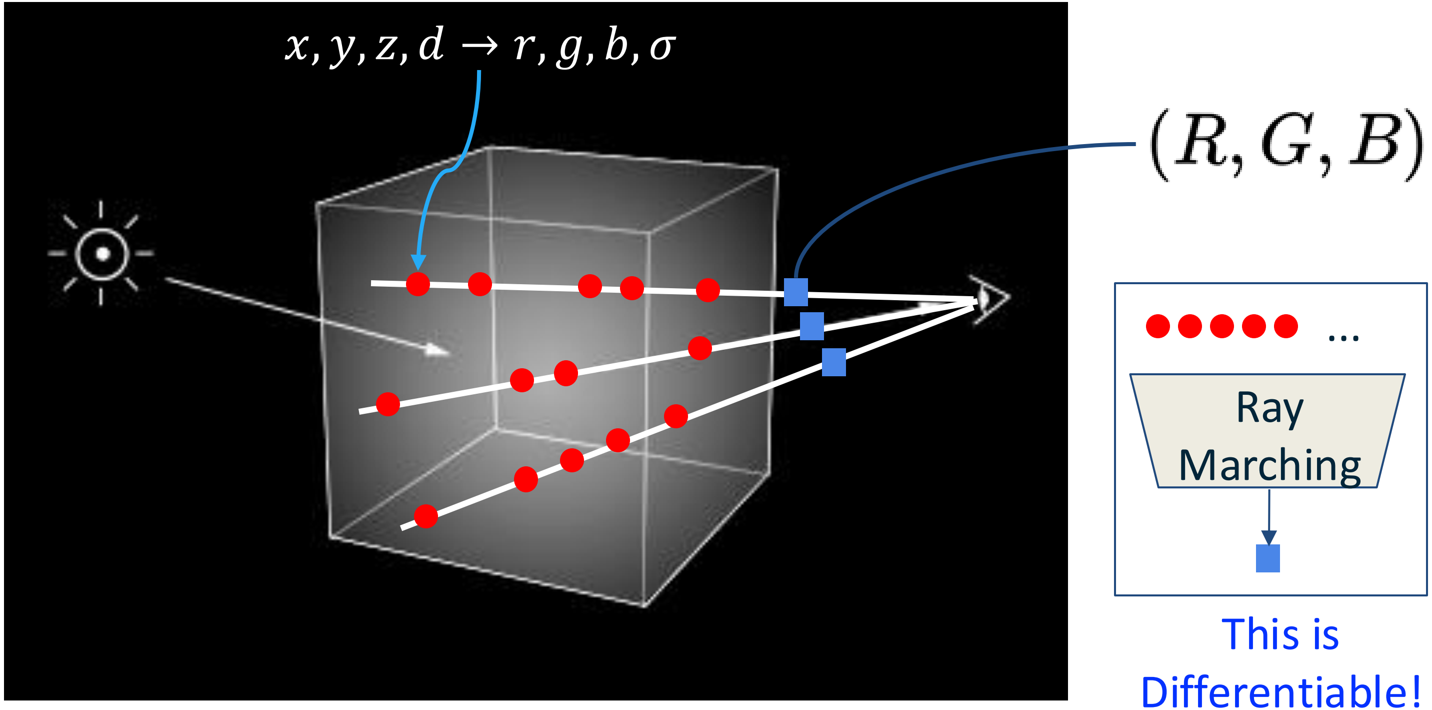




This is Differentiable!

Volume Rendering: Ray Marching

Neural Radiance Field (NeRF): Train a neural network to represent the scene volume: $F_\theta(x, y, z, d) = (r, g, b, \sigma)$. **Each NN encodes a 3D scene**.



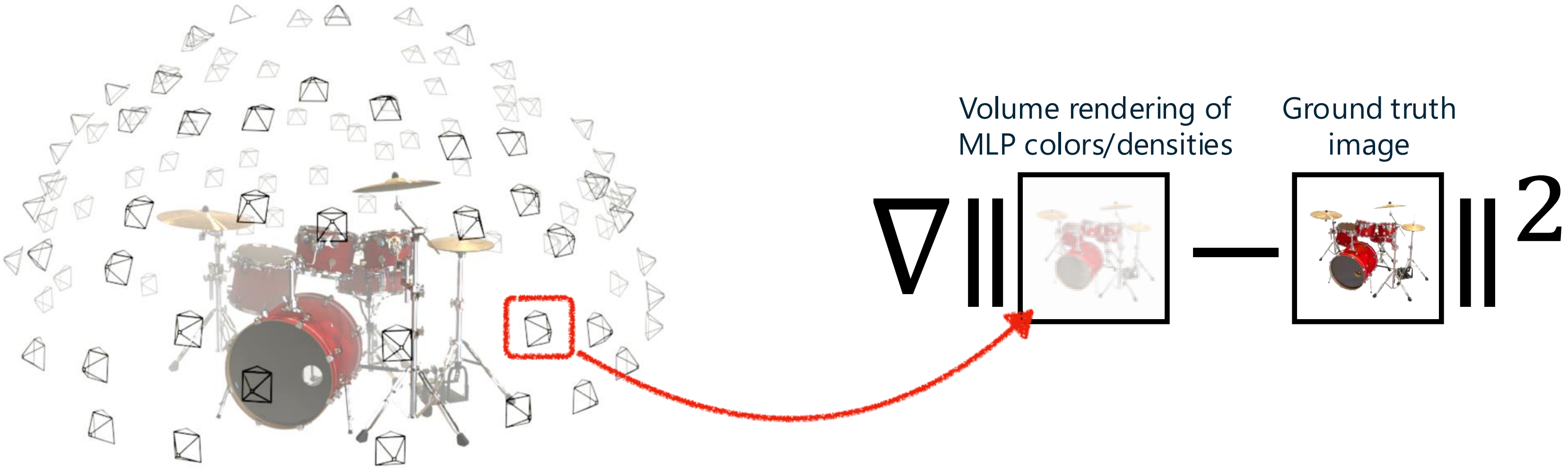
This is Differentiable!

NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis

Ben Mildenhall[1*] Pratul P. Srinivasan[1*] Matthew Tancik[1*]
Jonathan T. Barron[2] Ravi Ramamoorthi[3] Ren Ng[1]

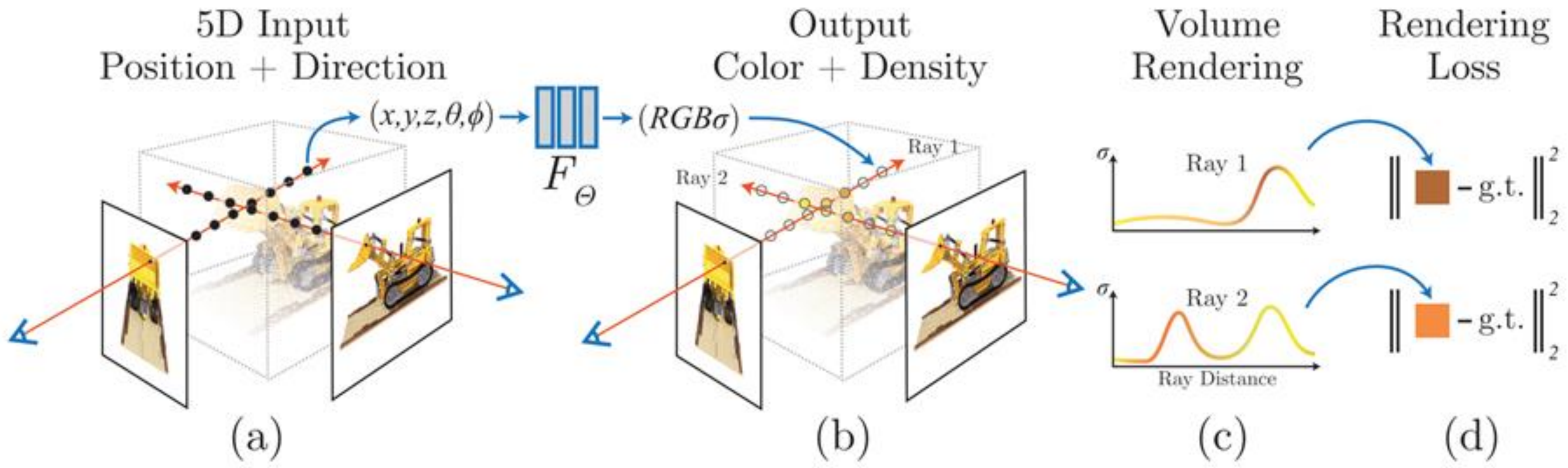
[1]UC Berkeley [2]Google Research [3]UC San Diego

Train a Single Neural Network to Reproduce the Ground Truth Images of a Scene



Adapted from material from Pratul Srinivasan

NeRF Overview



NeRF: Optimization

The volume density $\sigma(\mathbf{x})$ can be interpreted as the differential probability of a ray terminating at an infinitesimal particle at location $\mathbf{x}$. The expected color $C(\mathbf{r})$ of camera ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ with near and far bounds t_n and t_f is:

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt\,, \text{ where } T(t) = \exp\left(-\int_{t_n}^{t} \sigma(\mathbf{r}(s))ds\right). \quad (1)$$

Solution: Numerically estimate the integral (quadrature).

1. Discretize the ray into bins.
2. Sample point in each bin.
3. Compute numerical integration.

NeRF: Optimization

The volume density $\sigma(\mathbf{x})$ can be interpreted as the differential probability of a ray terminating at an infinitesimal particle at location $\mathbf{x}$. The expected color $C(\mathbf{r})$ of camera ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ with near and far bounds t_n and t_f is:

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt\,, \text{ where } T(t) = \exp\left(-\int_{t_n}^{t} \sigma(\mathbf{r}(s))ds\right). \quad (1)$$

Solution: Numerically estimate the integral (quadrature).

1. Discretize the ray into bins.
2. Sample point in each bin.
3. Compute numerical integration.

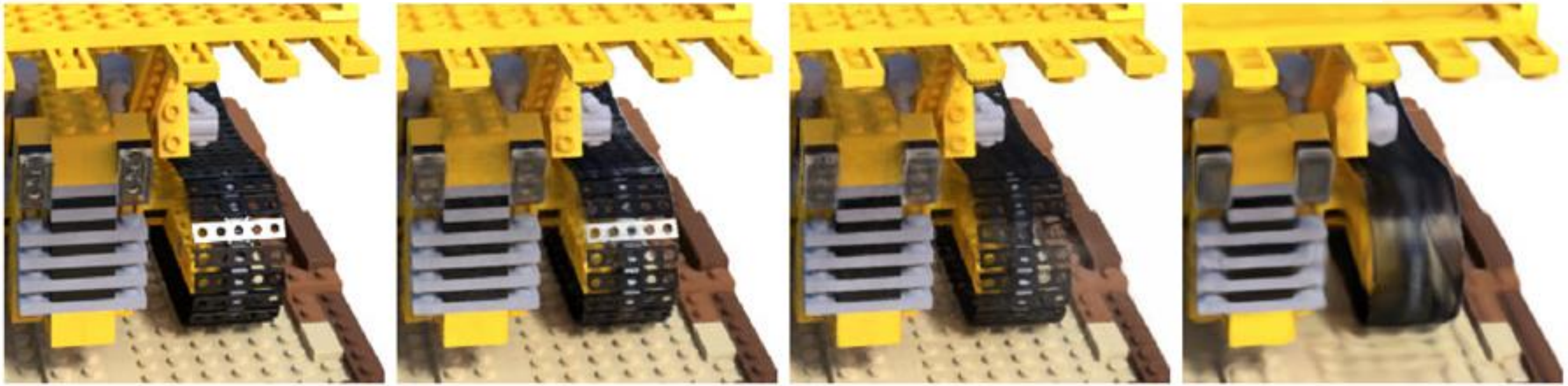
$$\hat{C}(\mathbf{r}) = \sum_{i=1}^{N} T_i(1 - \exp(-\sigma_i\delta_i))c_i\,, \text{ where } T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma_j\delta_j\right)$$

Key Insight 1: Positional Encoding

Challenge: Having F_θ operate directly on (x, y, z, d) performs poorly.

Solution: Positional encoding

$$\gamma(p) = \left(\sin\left(2^0 \pi p\right), \cos\left(2^0 \pi p\right), \cdots, \sin\left(2^{L-1} \pi p\right), \cos\left(2^{L-1} \pi p\right) \right)$$



Ground Truth | Complete Model | No View Dependence | No Positional Encoding

Key Insight 2: Hierarchical Volume Rendering

Challenge: Waste of compute on empty space.

Solution: coarse-to-fine prediction.

$$\hat{C}_c(\mathbf{r}) = \sum_{i=1}^{N_c} w_i c_i \,, \qquad w_i = T_i(1 - \exp(-\sigma_i \delta_i))\,. \tag{5}$$

Normalizing these weights as $\hat{w}_i = w_i / \sum_{j=1}^{N_c} w_j$ produces a piecewise-constant PDF along the ray. We sample a second set of N_f locations from this distribution using inverse transform sampling, evaluate our "fine" network at the union of the first and second set of samples, and compute the final rendered color of the ray $\hat{C}_f(\mathbf{r})$ using Eqn. 3 but using all $N_c + N_f$ samples. This procedure allocates more


NeRF encodes convincing view-dependent effects using directional dependence

Slide credit: Noah Snavely

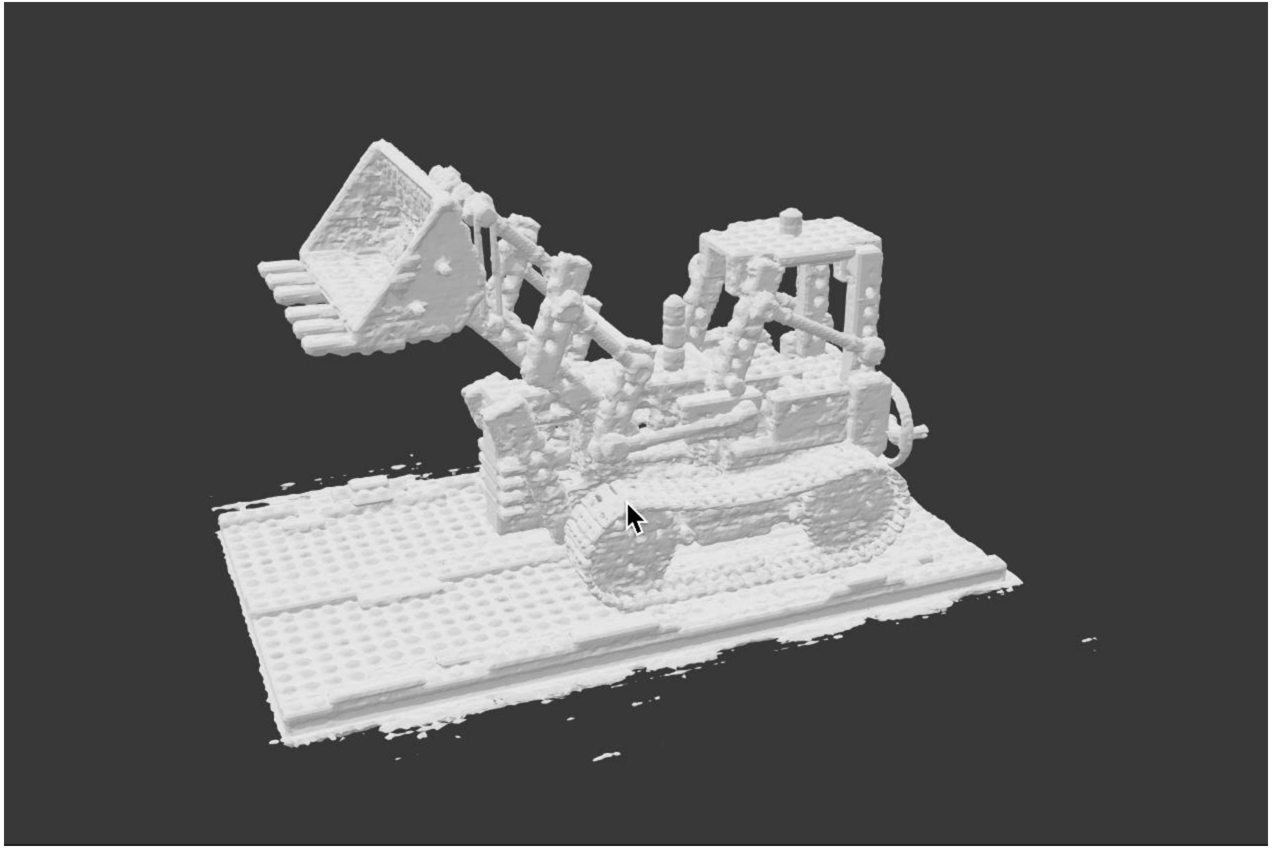
NeRF encodes convincing view-dependent effects using directional dependence

Slide credit: Noah Snavely

NeRF encodes detailed scene geometry with occlusion effects

Slide credit: Noah Snavely

NeRF encodes detailed scene geometry



Slide credit: Noah Snavely

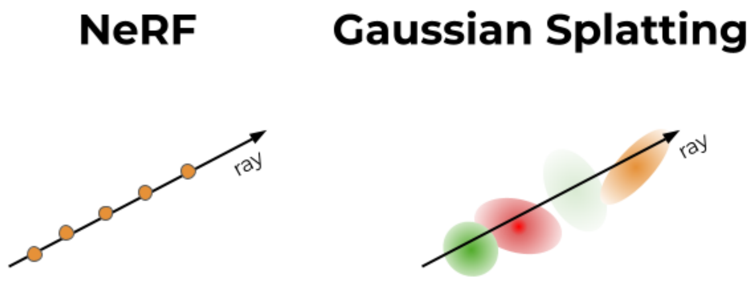
Space vs. Time Tradeoff

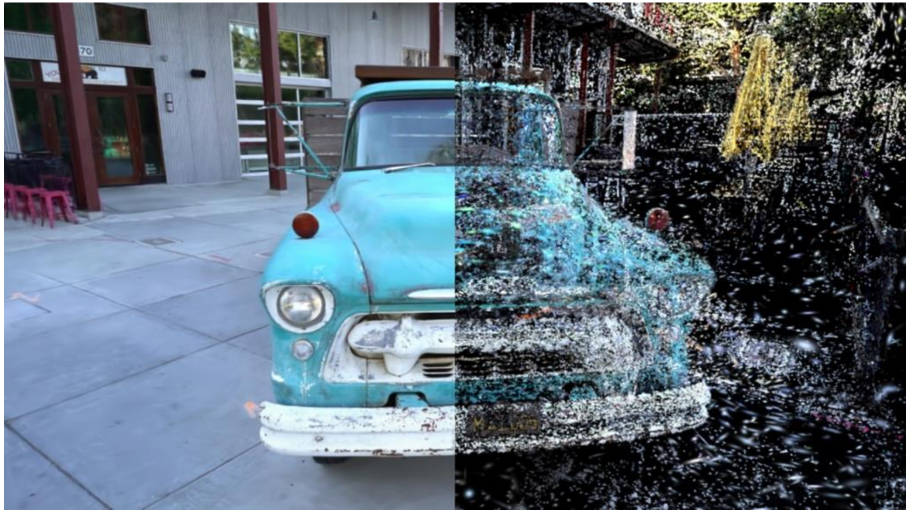
The biggest practical tradeoffs between these methods are time versus space. All compared single scene methods take at least 12 hours to train per scene. In contrast, LLFF can process a small input dataset in under 10 minutes. However, LLFF produces a large 3D voxel grid for every input image, resulting in enormous storage requirements (over 15GB for one "Realistic Synthetic" scene). Our method requires only 5 MB for the network weights (a relative compression of $3000\times$ compared to LLFF), which is even less memory than the *input images alone* for a single scene from any of our datasets.

3D Gaussian Splatting (Kerbl and Kopanas et al., 2023)

Key idea: 3D Gaussians as an **explicit representation** of a scene

- Train Gaussian blobs via inverse rendering (similar to NeRF)
- Store scene as Gaussian blobs instead of neural network weights (NeRF)
- Much faster during inference, but takes a lot of space to store





Summary: 3D Representation and Neural Rendering

- Representation matters a lot for 3D computer vision tasks (detection, reconstruction, etc.)
- 3D Voxels are intuitive representation of space but struggles with high-resolution shape and large scenes
- Implicit function emerge as a new paradigm in representing scenes with Neural Networks
- Neural volume rendering: represent scenes implicit as point-direction to color-density neural networks. Photorealistic rendering, slow to train and evaluate
- More recent works on trading off space and time