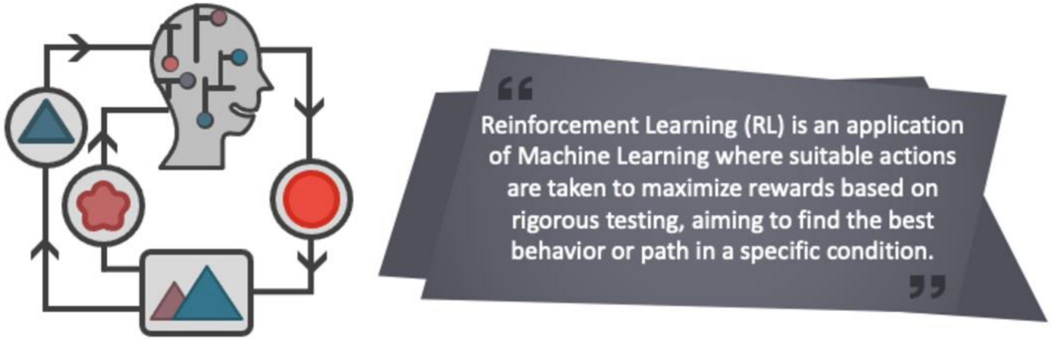
Week 8
Reinforcement Learning

Content

- Introduction to Reinforcement Learning
- Fundamentals of Reinforcement Learning
- Markov Decision Processes (MDPs)
- Q-Learning and Deep Q-Networks (DQNs)
- Policy Gradient Methods
- Python: Building a Simple Reinforcement Learning Agent

Introduction to Reinforcement Learning

- Learning through experience/data to make good decisions under uncertainty
- Essential part of intelligence
- Builds strongly from theory and ideas starting in the 1950s with Richard Bellman
- A number of impressive successes in the last decade



Introduction to Reinforcement Learning

- **Beyond Human Performance on the Board Game Go**

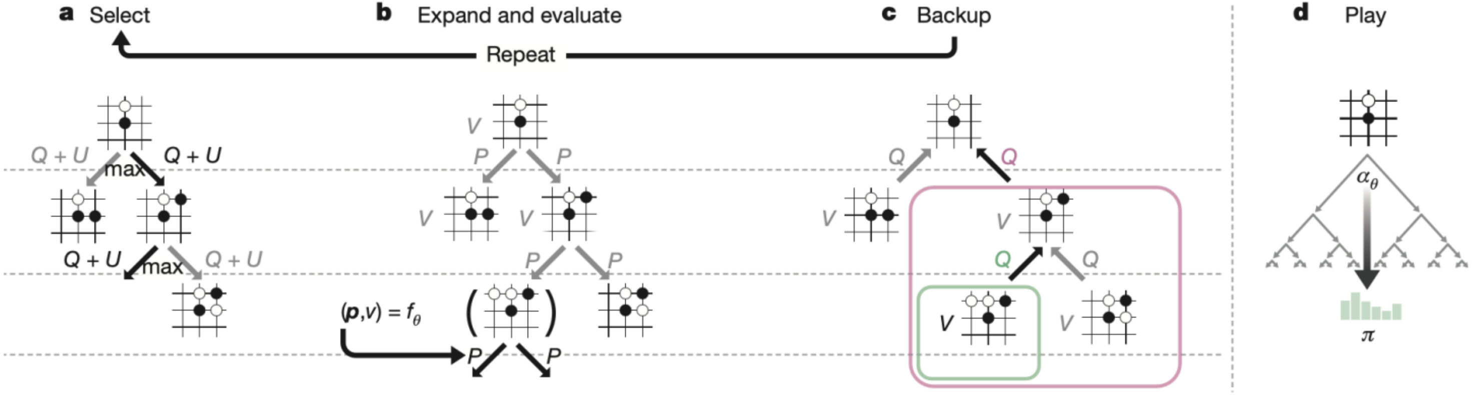


Image credits: Silver et al. Nature 2017
https://www.nature.com/articles/nature24270

Introduction to Reinforcement Learning

- **Learning Plasma Control for Fusion Science**

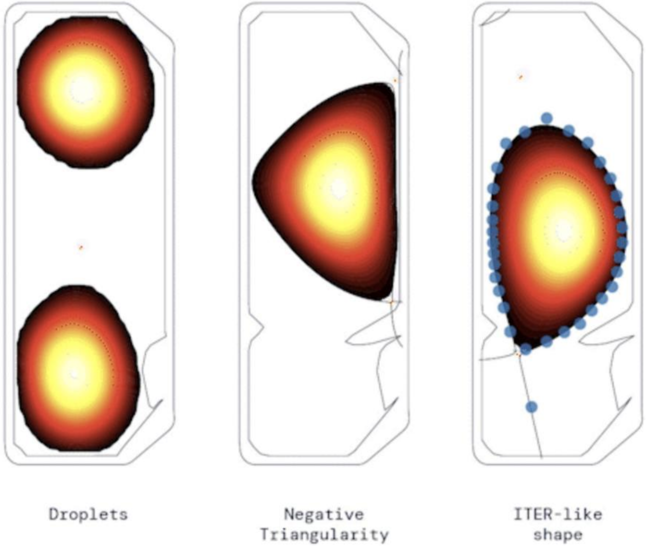
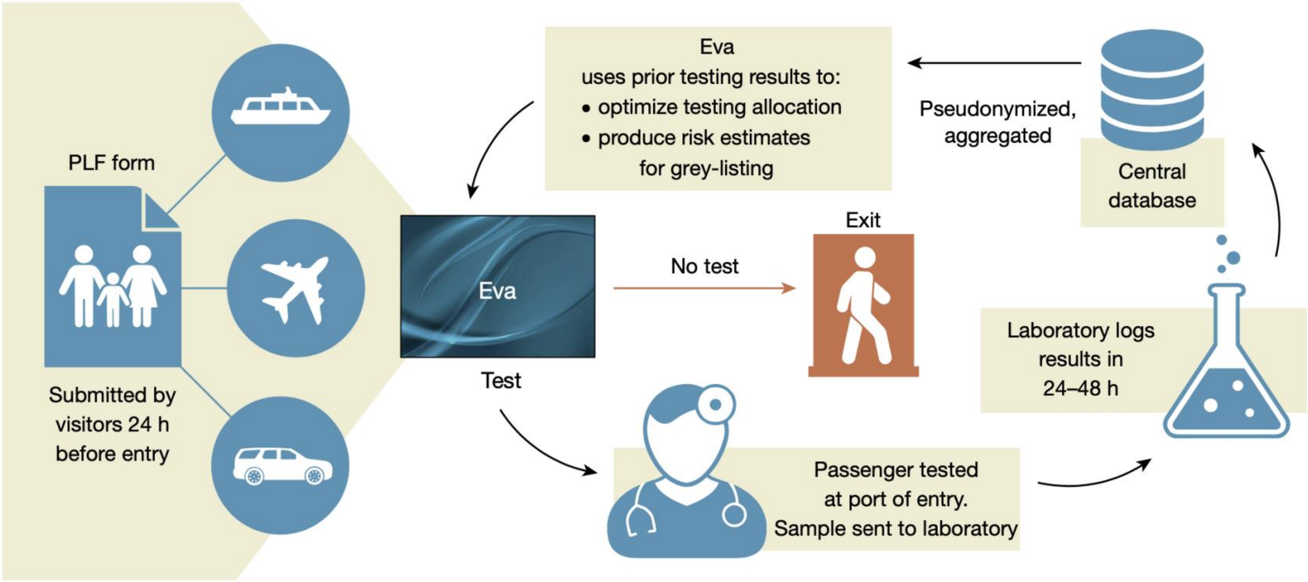


Image credits: left Alain Herzog / EPFL, right DeepMind & SPC/EPFL. Degrave et al. Nature 2022 https://www.nature.com/articles/s41586-021-04301-9

Introduction to Reinforcement Learning

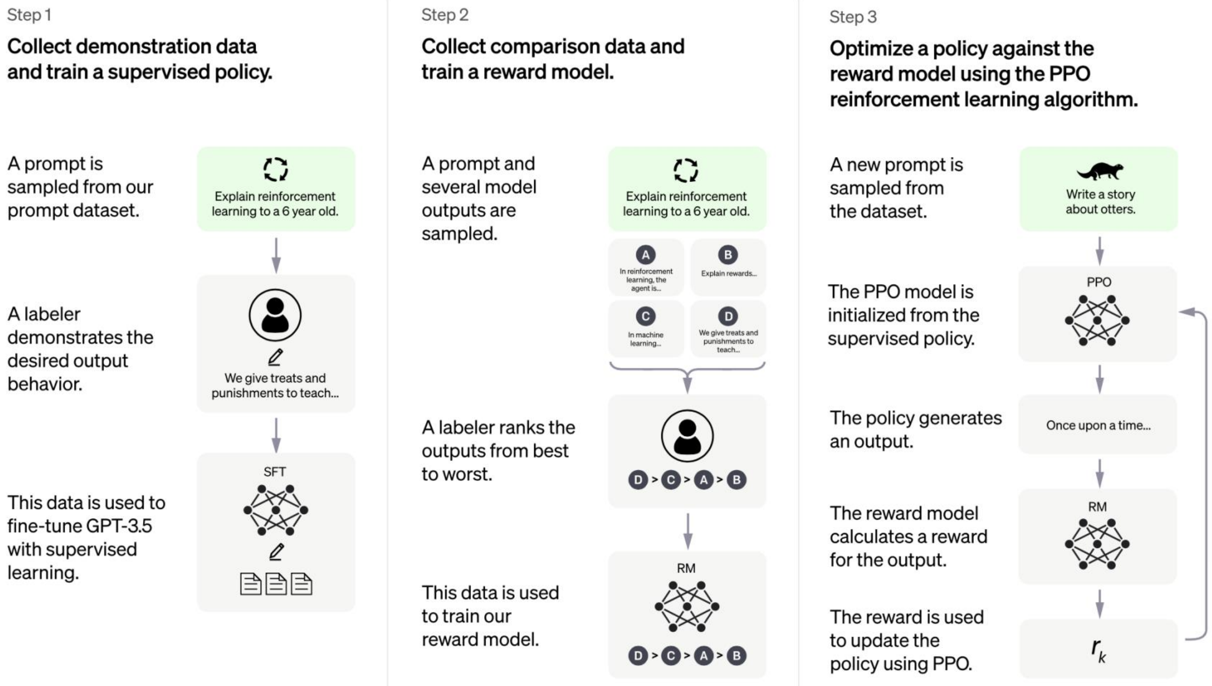
- **Efficient and targeted COVID-19 border testing via RL**



Bastani et al. Nature 2021
https://www.nature.com/articles/s41586-021-04014-z

Introduction to Reinforcement Learning

- **ChatGPT (https://openai.com/blog/chatgpt/)**

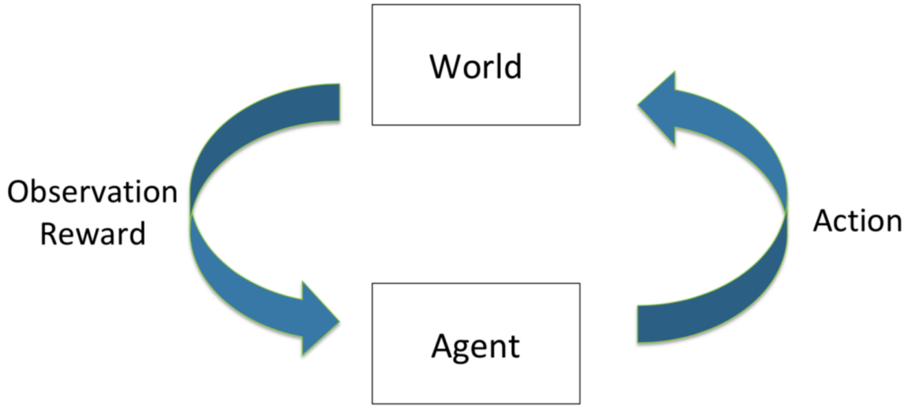


Fundamentals of Reinforcement Learning

- **Basic Concepts**
  - **Agent**: The entity that learns and makes decisions.
  - **Environment**: The system within which the agent operates.
  - **State (S)**: A representation of the environment at a specific point in time.
  - **Action (A)**: Choices available to the agent to interact with the environment.
  - **Reward (R)**: A scalar feedback signal received from the environment after taking an action.
  - **Policy (π)**: The strategy the agent uses to decide which action to take in a given state.
  - **π**(S, A)

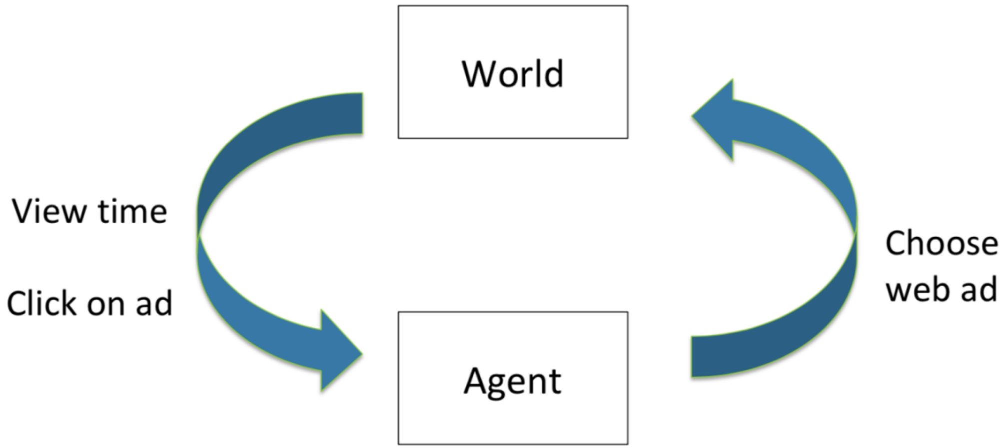
Fundamentals of Reinforcement Learning

- **Sequential Decision Making**
  - **Goal**: Select actions to maximize total expected future reward.
  - May require balancing immediate & long term rewards



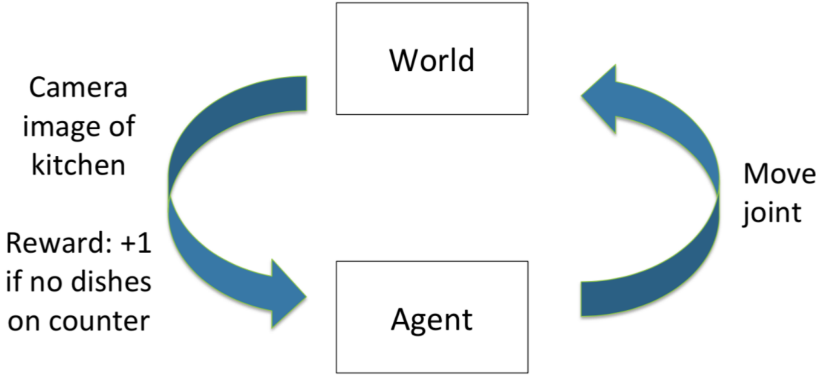
Fundamentals of Reinforcement Learning

- **Example: Web Advertising**
  - **Goal**: Select actions to maximize total expected future reward.
  - May require balancing immediate & long term rewards



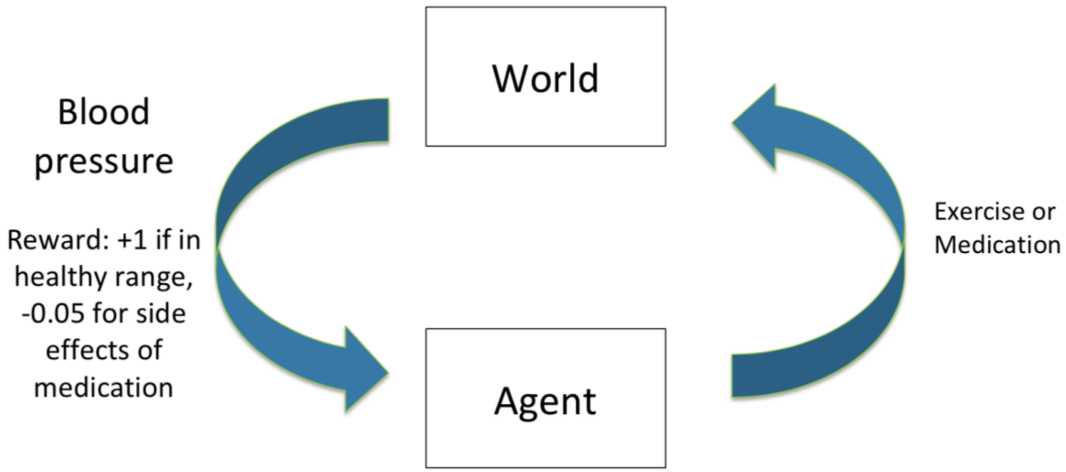
Fundamentals of Reinforcement Learning

- **Example: Robot Unloading Dishwasher**
    - **Goal**: Select actions to maximize total expected future reward.
    - May require balancing immediate & long term rewards



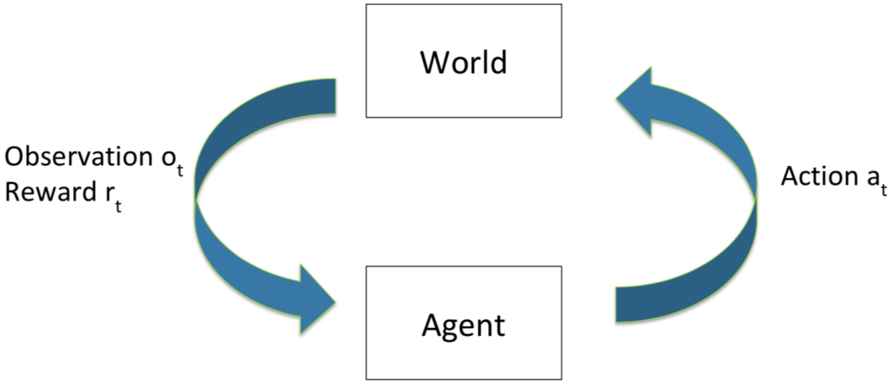
Fundamentals of Reinforcement Learning

- **Example: Blood Pressure Control**
  - **Goal**: Select actions to maximize total expected future reward.
  - May require balancing immediate & long term rewards



Fundamentals of Reinforcement Learning

- **Sequential Decision Process: Agent & the World (Discrete Time)**
  - Each time step t:
    - Agent takes an action a_t
    - World updates given action a_t, emits observation o_t and reward r_t
    - Agent receives observation o_t and reward r_t



Markov Decision Processes (MDPs)

- **Markov Assumption**
  - Information state: sucient statistic of history
  - State s_t is Markov if and only if:

$$p(s_{t+1}|s_t, a_t) = p(s_{t+1}|h_t, a_t)$$

  - Future is independent of past given present

Markov Decision Processes (MDPs)

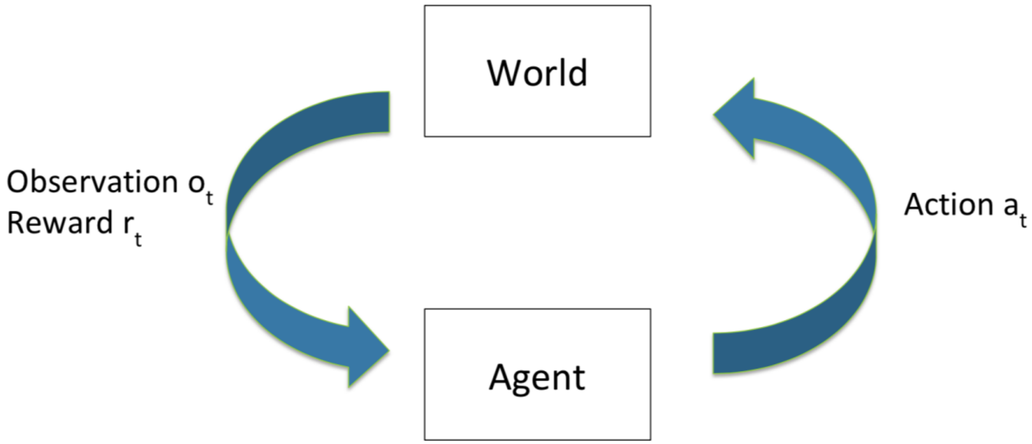
- **Why is Markov Assumption Popular?**
  - Simple and often can be satisfied if include some history as part of the state
  - In practice often assume most recent observation is sucient statistic of history

$$s_t = o_t$$

  - State representation has big implications for:
    - Computational complexity
    - Data required
    - Resulting performance

Markov Decision Processes (MDPs)

- **Why is Markov Assumption Popular?**
  - Is state Markov? Is world partially observable? (POMDP)
  - Are dynamics deterministic or stochastic?
  - Do actions influence only immediate reward (bandits) or reward and next state ?



Markov Decision Processes (MDPs)

- **Example: Mars Rover as a Markov Decision Process**
  - States: Location of rover (s1,...,s7)
  - Actions: TryLeft or TryRight
  - Rewards:
    - +1 in state s1
    - +10 in state s7
    - 0 in all other states

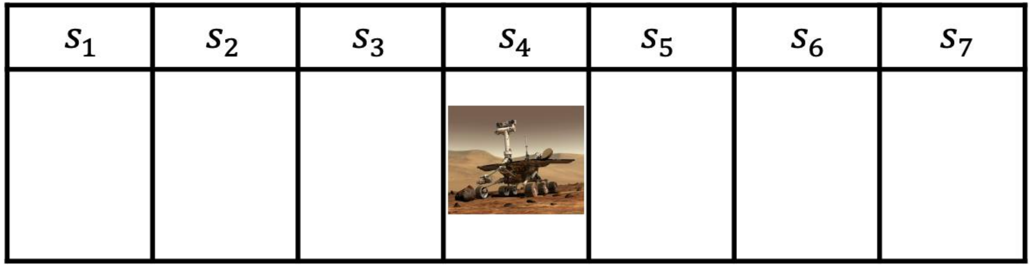


Figure: Mars rover image: NASA/JPL-Caltech

Markov Decision Processes (MDPs)

- **MDP Model**
  - Agent's representation of how world changes given agent's action
  - Transition / dynamics model predicts next agent state

$$p(s_{t+1} = s' | s_t = s, a_t = a)$$

  - Reward model predicts immediate reward

$$r(s_t = s, a_t = a) = \mathbb{E}[r_t | s_t = s, a_t = a]$$

Markov Decision Processes (MDPs)

- **Example: Mars Rover Stochastic Markov Model**
  - Numbers above show RL agent's reward model
  - Part of agent's transition model:

$$0.5 = P(s_1|s_1, \text{TryRight}) = P(s_2|s_1, \text{TryRight})$$
$$0.5 = P(s_2|s_2, \text{TryRight}) = P(s_3|s_2, \text{TryRight}) \cdots$$

  - Model may be wrong

s_1	s_2	s_3	s_4	s_5	s_6	s_7
$\hat{r} = 0$	$\hat{r} = 0$	$\hat{r} = 0$	$\hat{r} = 0$	$\hat{r} = 0$	$\hat{r} = 0$	$\hat{r} = 0$

Markov Decision Processes (MDPs)

- **Policy**
  - Policy π determines how the agent chooses actions
  - π : S -> A, mapping from states to actions
  - Deterministic policy:

$$\pi(s) = a$$

  - Stochastic policy:

$$\pi(a|s) = Pr(a_t = a|s_t = s)$$

Markov Decision Processes (MDPs)

- **Example: Mars Rover Policy**
  - π (s1) = π (s2) = $\cdots$ = π (s7) = TryRight
  - Q: is this a deterministic policy or a stochastic policy?

Markov Decision Processes (MDPs)

- **Evaluation and Control**
  - Evaluation
    - Estimate/predict the expected rewards from following a given policy
  - Control
    - Optimization: find the best policy

Markov Decision Processes (MDPs)

- **Making Sequences of Good Decisions Given a Model of the World**
  - Assume finite set of states and actions
  - Given models of the world (dynamics and reward)
  - Evaluate the performance of a particular decision policy
  - Compute the best policy
  - This can be viewed as an AI planning problem

Markov Decision Processes (MDPs)

- **Making Sequences of Good Decisions Given a Model of the World**
    - Markov Processes Markov
    - Reward Processes (MRPs)
    - Markov Decision Processes (MDPs)
    - Evaluation and Control in MDPs

Q-Learning and Deep Q-Networks (DQNs)

- Reinforcement Learning (Generally) Involves
- Optimization
- Delayed consequences
- Exploration
- Generalization

Q-Learning and Deep Q-Networks (DQNs)

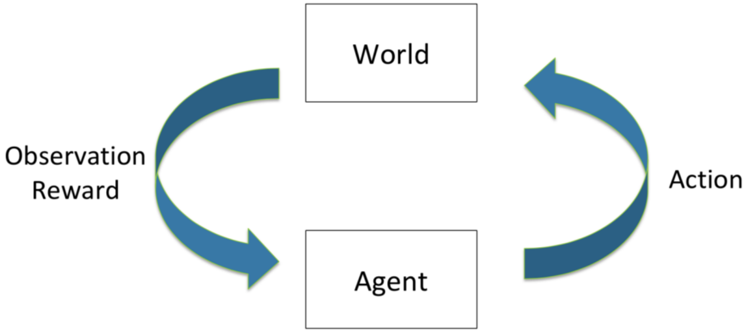
- **Reinforcement Learning (Generally) Involves**
  - Optimization
    - Goal is to find an optimal way to make decisions
    - Explicit notion of decision utility
  - Delayed consequences
    - Decisions now can impact things much later...
    - Saving for retirement Finding a key in video game Montezuma’s revenge
  - Exploration
    - Learning about the world by making decisions
    - Decisions impact what we learn about
  - Generalization
    - Policy is mapping from past experience to action

Q-Learning and Deep Q-Networks (DQNs)

- **Generalized Policy Improvement**
  - Model-free Policy Iteration
    - Initialize policy π
    - Repeat:
      - Policy evaluation: compute Q^π
      - Policy improvement: update π given Q^π
    - May need to modify policy evaluation:
      - If π is deterministic, can't compute $Q(s, a)$ for any $a \neq \pi(s)$
    - How to interleave policy evaluation and improvement?
      - Policy improvement is now using an estimated Q

Q-Learning and Deep Q-Networks (DQNs)

- **Generalized Policy Improvement**
  - The Problem of Exploration
    - Goal: Learn to select actions to maximize total expected future reward
    - Problem: Can't learn about actions without trying them (need to explore)
    - Problem: But if we try new actions, spending less time taking actions that our past experience suggests will yield high reward (need to exploit knowledge of domain to achieve high rewards)



Q-Learning and Deep Q-Networks (DQNs)

- **Generalized Policy Improvement**
  - ϵ-greedy Policies
    - Simple idea to balance exploration and achieving rewards
    - Let |A| be the number of actions
    - Then an ϵ-greedy policy w.r.t. a state-action value Q(s, a) is $\pi(a|s)$ =

$$\begin{array}{l} \arg\max_a Q(s,a), \text{ w. prob } 1-\epsilon+\frac{\epsilon}{|A|} \\ a' \neq \arg\max Q(s,a) \text{ w. prob } \frac{\epsilon}{|A|} \end{array}$$

    - In words: select argmax action with probability $1 - \epsilon$, else select action uniformly at random

Q-Learning and Deep Q-Networks (DQNs)

- **Generalized Policy Improvement**
  - Policy Improvement with ϵ-greedy policies
    - Recall we proved that policy iteration using given dynamics and reward models, was guaranteed to monotonically improve
    - That proof assumed policy improvement output a deterministic policy
    - Same property holds for ϵ-greedy policies
    - Monotonic ϵ-greedy Policy Improvement
      - Theorem

For any ϵ-greedy policy π_i, the ϵ-greedy policy w.r.t. Q^{π_i}, π_{i+1} is a monotonic improvement $V^{\pi_{i+1}} \geq V^{\pi_i}$

$$
\begin{aligned}
Q^{\pi_i}(s, \pi_{i+1}(s)) &= \sum_{a \in A} \pi_{i+1}(a|s) Q^{\pi_i}(s, a) \\
&= (\epsilon / |A|) \left[\sum_{a \in A} Q^{\pi_i}(s, a) \right] + (1 - \epsilon) \max_a Q^{\pi_i}(s, a)
\end{aligned}
$$

Q-Learning and Deep Q-Networks (DQNs)

- **Recall Monte Carlo Policy Evaluation, Now for Q**

Initialize $Q(s,a) = 0, N(s,a) = 0 \; \forall (s,a)$, $k = 1$, Input $\epsilon = 1$, π
loop
  Sample k-th episode $(s_{k,1}, a_{k,1}, r_{k,1}, s_{k,2}, \ldots, s_{k,T})$ given π
  Compute $G_{k,t} = r_{k,t} + \gamma r_{k,t+1} + \gamma^2 r_{k,t+2} + \cdots \gamma^{T_i - 1} r_{k,T_i} \; \forall t$
  for $t = 1, \ldots, T$ **do**
    if First visit to **(s,a)** in episode k **then**
      $N(s,a) = N(s,a) + 1$
      $Q(s_t, a_t) = Q(s_t, a_t) + \frac{1}{N(s,a)}(G_{k,t} - Q(s_t, a_t))$
    end if
  end for
  $k = k + 1$
end loop

Q-Learning and Deep Q-Networks (DQNs)

- **Monte Carlo Online Control / On Policy Improvement**

Initialize $Q(s,a) = 0, N(s,a) = 0\ \forall(s,a)$, Set $\epsilon = 1,\ k = 1$
$\pi_k = \epsilon\text{-greedy}(Q)$ // Create initial ϵ-greedy policy
loop
    Sample k-th episode $(s_{k,1}, a_{k,1}, r_{k,1}, s_{k,2}, \ldots, s_{k,T})$ given π_k
    $G_{k,t} = r_{k,t} + \gamma r_{k,t+1} + \gamma^2 r_{k,t+2} + \cdots \gamma^{T_i - 1} r_{k,T_i}$
    for $t = 1, \ldots, T$ **do**
        if First visit to (s,a) in episode k **then**
            $N(s,a) = N(s,a) + 1$
            $Q(s_t, a_t) = Q(s_t, a_t) + \frac{1}{N(s,a)}(G_{k,t} - Q(s_t, a_t))$
        end if
    end for
    $k = k + 1,\ \epsilon = 1/k$
    $\pi_k = \epsilon\text{-greedy}(Q)$ // Policy improvement
end loop

Q-Learning and Deep Q-Networks (DQNs)

- **Greedy in the Limit of Infinite Exploration (GLIE)**
  - Definition of GLIE
    - All state-action pairs are visited an infinite number of times

$$\lim_{i \to \infty} N_i(s, a) \to \infty$$

    - Behavior policy (policy used to act in the world) converges to greedy policy
  - GLIE Monte-Carlo Control using Tabular Representations
    - GLIE Monte-Carlo control converges to the optimal state-action value function

      $Q(s, a) \to Q*(s, a)$

Q-Learning and Deep Q-Networks (DQNs)

- **Model-free Policy Iteration with TD Methods**
  - Initialize policy π
  - Repeat:
    - Policy evaluation: compute Qπ using temporal difference updating with ϵ-greedy policy
    - Policy improvement: Same as Monte carlo policy improvement, set π to ϵ-greedy (Qπ) Method 1: SARSA
  - On policy: SARSA computes an estimate Q of policy used to act

Q-Learning and Deep Q-Networks (DQNs)

- **General Form of SARSA(What is SARSA) Algorithm**

```
Set initial ε-greedy policy π randomly, t = 0, initial state s_t = s_0
Take a_t ~ π(s_t)
Observe (r_t, s_{t+1})
loop
    Take action a_{t+1} ~ π(s_{t+1}) // Sample action from policy
    Observe (r_{t+1}, s_{t+2})
    Update Q given (s_t, a_t, r_t, s_{t+1}, a_{t+1}):

    Perform policy improvement:

    t = t + 1
end loop
```

Q-Learning and Deep Q-Networks (DQNs)

- **Q-Learning: Learning the Optimal State-Action Value**
  - SARSA is an on-policy learning algorithm
  - SARSA estimates the value of the current behavior policy (policy using to take actions in the world)
  - And then updates that (behavior) policy
  - Alternatively, can we directly estimate the value of $\pi *$ while acting with another behavior policy πb?
  - Yes! Q-learning, an off-policy RL algorithm

Q-Learning and Deep Q-Networks (DQNs)

- **Q-Learning: Learning the Optimal State-Action Value**
  - SARSA is an on-policy learning algorithm
    - Estimates the value of behavior policy (policy using to take actions in the world)
    - And then updates the behavior policy Q-learning estimate the Q value of $\pi *$ while acting with another behavior policy πb
  - Key idea: Maintain Q estimates and bootstrap for best future value
  - Recall SARSA

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha((r_t + \gamma Q(s_{t+1}, a_{t+1})) - Q(s_t, a_t))$$

  - Q-learning:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha((r_t + \gamma \max_{a'} Q(s_{t+1}, a')) - Q(s_t, a_t))$$

Q-Learning and Deep Q-Networks (DQNs)

- **Q-Learning with ϵ-greedy Exploration**

Initialize $Q(s, a), \forall s \in S, a \in A$ $t = 0$, initial state $s_t = s_0$
Set π_b to be ϵ-greedy w.r.t. Q
loop
    Take $a_t \sim \pi_b(s_t)$ // Sample action from policy
    Observe (r_t, s_{t+1})
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$
    $\pi(s_t) = \arg\max_a Q(s_t, a)$ w.prob $1 - \epsilon$, else random
    $t = t + 1$
end loop

Q-Learning and Deep Q-Networks (DQNs)

- **Q-Learning with Neural Networks**
  - Q-learning converges to optimal $Q*(s, a)$ using tabular representation
  - In value function approximation Q-learning minimizes MSE loss by stochastic gradient descent using a target Q estimate instead of true Q
  - But Q-learning with VFA can diverge
  - Two of the issues causing problems:
    - Correlations between samples
    - Non-stationary targets
  - Deep Q-learning (DQN) addresses these challenges by using
    - Experience replay
    - Fixed Q-targets

Q-Learning and Deep Q-Networks (DQNs)

- **DQNs: Experience Replay**
  - To help remove correlations, store dataset (called a replay buffer) D from prior experience
  - To perform experience replay, repeat the following:

s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
s_3, a_3, r_4, s_4
$\ldots$
$s_t, a_t, r_{t+1}, s_{t+1}$

$\rightarrow \quad s, a, r, s'$

$(s, a, r, s') \sim \mathcal{D}$: sample an experience tuple from the dataset
Compute the target value for the sampled s: $r + \gamma \max_{a'} \hat{Q}(s', a'; \boldsymbol{w})$
Use stochastic gradient descent to update the network weights

$$\Delta \boldsymbol{w} = \alpha(r + \gamma \max_{a'} \hat{Q}(s', a'; \boldsymbol{w}) - \hat{Q}(s, a; \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{Q}(s, a; \boldsymbol{w})$$

  - Uses target as a scalar, but function weights will get updated on the next round, changing the target value

Q-Learning and Deep Q-Networks (DQNs)

- **DQN Pseudocode**

Input $\boldsymbol{C}$, $\boldsymbol{\alpha}$, $D = \{\}$, Initialize $\boldsymbol{w}$, $\boldsymbol{w}^- = \boldsymbol{w}$, $t = 0$
Get initial state s_0
loop
    Sample action a_t given ϵ-greedy policy for current $\hat{Q}(s_t, a; \boldsymbol{w})$
    Observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in replay buffer D
    Sample random minibatch of tuples (s_i, a_i, r_i, s_{i+1}) from D
    for j in minibatch **do**
        if episode terminated at step $i + 1$ **then**
            $y_i = r_i$
        else
            $y_i = r_i + \gamma \max_{a'} \hat{Q}(s_{i+1}, a'; \boldsymbol{w}^-)$
        end if
        Do gradient descent step on $(y_i - \hat{Q}(s_i, a_i; \boldsymbol{w}))^2$ for parameters $\boldsymbol{w}$: $\Delta \boldsymbol{w} = \alpha(y_i - \hat{Q}(s_i, a_i; \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{Q}(s_i, a_i; \boldsymbol{w})$
    end for
    $t = t + 1$
    if mod(t,C) $==$ 0 **then**
        $\boldsymbol{w}^- \leftarrow \boldsymbol{w}$
    end if
end loop

Note there are several hyperparameters and algorithm choices. One needs to choose the neural network architecture, the learning rate, and how often to update the target network. Often a fixed size replay buffer is used for experience replay, which introduces a parameter to control the size, and the need to decide how to populate it.

Q-Learning and Deep Q-Networks (DQNs)

- **DQNs Summary (Other RF model)**
  - DQN uses experience replay and fixed Q-targets
  - Store transition (s_t , a_t ,r_{t+1},s_{t+1}) in replay memory D
  - Sample random mini-batch of transitions (s, a,r,s ′) from D
  - Compute Q-learning targets w.r.t. old, fixed parameters w^-
  - Optimizes MSE between Q-network and Q-learning targets
  - Uses stochastic gradient descent

Q-Learning and Deep Q-Networks (DQNs)

- **Which Aspects of DQN were Important for Success?**

Game	Linear	Deep Network	DQN w/ fixed Q	DQN w/ replay	DQN w/replay and fixed Q
Breakout	3	3	10	241	317
Enduro	62	29	141	831	1006
River Raid	2345	1453	2868	4102	7447
Seaquest	656	275	1003	823	2894
Space Invaders	301	302	373	826	1089

Replay is hugely important
Why? Beyond helping with correlation between samples, what does replaying do?

Python: Building a Simple Reinforcement Learning Agent

```python
class QLearningAgent:
    def __init__(self, env, learning_rate=0.1, discount_factor=0.99, exploration_rate=1.0, exploration_decay=0.995):
        self.env = env
        self.q_table = np.zeros((env.size, env.size, 4))  # Q-table initialized to zeros
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_rate = exploration_rate
        self.exploration_decay = exploration_decay

    def choose_action(self, state):
        if random.uniform(0, 1) < self.exploration_rate:
            return random.choice([0, 1, 2, 3])  # Explore
        else:
            x, y = state
            return np.argmax(self.q_table[x, y])  # Exploit

    def learn(self, state, action, reward, next_state):
        x, y = state
        next_x, next_y = next_state
        best_next_action = np.argmax(self.q_table[next_x, next_y])
        td_target = reward + self.discount_factor * self.q_table[next_x, next_y, best_next_action]
        td_error = td_target - self.q_table[x, y, action]
        self.q_table[x, y, action] += self.learning_rate * td_error
```