Reinforcement Learning
What is Reinforcement Learning?

- General purpose framework for learning Artificial Intelligence models
- RL assumes that the agent (our model) can take actions
- These actions affect the environment where the agent operates, more specifically the state of the environment and the state of the agent
- Given the state of the environment and the agent, an action taken from the agent causes a reward (can be positive or negative)
- Goal: the goal of an RL agent is to learn how to take actions that maximize future rewards
Some examples of RL
Some examples of RL

- Controlling physical systems
  - Robot walking, jumping, driving

- Logistics
  - Scheduling, bandwidth allocation

- Games
  - Atari, Go, Chess, Pacman

- Learning sequential algorithms
  - Attention, memory
Reinforcement Learning: An abstraction

Dynamical System ("The World")

State $s_t$

Reward $r_t$

Observation $o_t$

Action $a_t$

Learning Agent ("Our Model")
Experience is a series of observations, actions and rewards
\[ o_1, r_1, a_1, o_2, r_2, a_2, ..., o_t, r_t \]

The state is the summary of experience so far
\[ s_t = f(o_1, r_1, a_1, o_2, r_2, a_2, ..., o_t, r_t) \]

If we have fully observable environments, then
\[ s_t = f(o_t) \]
Policy

- Policy is the agent’s behavior function

- The policy function maps the state input $s_t$ to an action output $a_t$

- Deterministic policy: $a_t = f(s_t)$
- Stochastic policy: $\pi(a_t | s_t) = \mathbb{P}(a_t | s_t)$
Value function

- A value function is the prediction of the future reward
  - Given the state $s_t$ what will my reward be if I do action $a_t$

- The Q-value function gives the expected future reward

- Given state $s_t$, action $a_t$, a policy $\pi$ the Q-value function is $Q^\pi(s_t, a_t)$
How do we decide about actions, states, rewards?

- We model the policy and the value function as machine learning functions that can be optimized by the data.

- The **policy function** \( a_t = \pi(s_t) \) selects an action given the current state.

- The **value function** \( Q^\pi(s_t, a_t) \) is the expected total reward that we will receive if we take action \( a_t \) given state \( s_t \).

- What should our goal then be?
Goal: Maximize future rewards!

- Learn the policy and value functions such that the action taken at the $t$-th time step $a_t$ maximizes the expected sum of future rewards

$$Q^\pi(s_t, a_t) = \mathbb{E}(r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots | s_t, a_t)$$

- $\gamma$ is a discount factor. Why do we need it?
Goal: Maximize future rewards!

- Learn the policy and value functions such that the action taken at the $t$-th time step $a_t$ maximizes the expected sum of future rewards

$$Q^\pi(s_t, a_t) = \mathbb{E}(r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots | s_t, a_t)$$

- $\gamma$ is a discount factor. Why do we need it?
  - The further into the future we look $t + 1, \ldots, t + T$, the less certain we can be about our expected rewards $r_{t+1}, \ldots, r_{t+T}$
How can we rewrite the value function in more compact form

\[ Q^\pi(s_t, a_t) = \mathbb{E}(r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots | s_t, a_t) = ? \]
Bellman equation

- How can we rewrite the value function in more compact form
  
  \[ Q^\pi(s_t, a_t) = \mathbb{E}(r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots | s_t, a_t) = \mathbb{E}_{s', a'}(r + \gamma Q^\pi(s', a') | s_t, a_t) \]

- This is the **Bellman equation**

- How can we rewrite the optimal value function \( Q^*(s_t, a_t) \)?
How can we rewrite the value function in more compact form

\[
Q^\pi(s_t, a_t) = \mathbb{E}(r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots | s_t, a_t)
\]

\[
= \mathbb{E}_{s'}(r + \gamma Q^\pi(s', a')| s_t, a_t)
\]

This is the **Bellman equation**
Optimal value function

- Optimal value function $Q^*(s, a)$ is attained with the optimal policy $\pi^*$

$$Q^*(s, a) = \max_{\pi} Q^\pi(s, a) = Q^{\pi^*}(s, a)$$

- After we have found the optimal policy $\pi^*$ we do the optimal action

$$\pi^* = \arg\max_a Q^*(s, a)$$

- By expanding the optimal value function

$$Q^*(s, a) = r_{t+1} + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})$$

$$Q^*(s, a) = \mathbb{E}_{s'} \left( r + \gamma \max_{a'} Q^*(s', a') \middle| s, a \right)$$
Environment Models in RL

- The model is learnt from experience
- The model acts as a replacement for the environment
- When planning, the agent can interact with the model
- For instance look ahead search to estimate the future states given actions
Approaches to Reinforcement Learning

- Policy-based
  - Learn directly the optimal policy $\pi^*$
  - The policy $\pi^*$ obtains the maximum future reward

- Value-based
  - Learn the optimal value function $Q^*(s, a)$
  - This value function applies for any policy

- Model-based
  - Build a model for the environment
  - Plan and decide using that model
How to make RL deep?
How to make RL deep?

- Use Deep Networks for the
  - Value function
  - Policy
  - Model

- Optimize final loss with SGD
How to make RL deep?
Deep Reinforcement Learning

- Non-linear function approximator: Deep Networks
- Input is as raw as possible, e.g. image frame
  ◦ Or perhaps several frames (When needed?)
- Output is the best possible action out of a set of actions for maximizing future reward
- **Important:** no need anymore to compute the actual value of the action-value function and take the maximum: \( \arg \max_{\alpha} Q_\theta(s, a) \)
  ◦ The network returns directly the optimal action
Value-based Deep RL
Q-Learning

- Optimize for Q value function
  \[
  Q^\pi(s_t, a_t) = \mathbb{E}_{s'}(r + \gamma Q^\pi(s', a')|s_t, a_t)
  \]
- In the beginning of learning the function \( Q(s, a) \) is incorrect
- We set \( r + \gamma \max_{a'} Q_t(s', a') \) to be the learning target
- Then we minimize the loss
  \[
  \min \left( r + \gamma \max_{a'} Q_t(s', a') - Q_t(s, a) \right)^2
  \]
Q-Learning

- Value iteration algorithms solve the Bellman equation

\[
Q_{t+1}(s, a) = \mathbb{E}_{s', r} \left( r + \gamma \max_{a'} Q_t(s', a') \middle| s, a \right)
\]

- In the simplest case \(Q_t\) is a table
  - To the limit iterative algorithms converge to \(Q^*\)

- However, a table representation for \(Q_t\) is not always enough
How to optimize?

- The objective is the mean squared-error in Q-values
  \[
  \mathcal{L}(\theta) = \mathbb{E} \left[ (r + \gamma \max_{a'} Q(s', a', \theta) - Q(s, a, \theta))^2 \right]
  \]

- The Q-Learning gradient then becomes
  \[
  \frac{\partial \mathcal{L}}{\partial \theta} = \mathbb{E} \left[ -2 \cdot (r + \gamma \max_{a'} Q(s', a', \theta) - Q(s, a, \theta)) \frac{\partial Q(s, a, \theta)}{\partial \theta} \right]
  \]

- Optimize end-to-end with SGD

- Scalar target value → Gradient 0
In practice

1. Do a feedforward pass for the current state \( s \) to get predicted Q-values for all actions

2. Do a feedforward pass for the next state \( s' \) and calculate maximum overall network outputs
   \[ \max_{a'} Q(s', a', \theta) \]

3. Set Q-value target to
   \[ r + \gamma \max_{a'} Q(s', a', \theta) \]
   - use the max calculated in step 2
   - For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs

4. Update the weights using backpropagation.
Deep Q Networks on Atari

- End-to-end learning from raw pixels
- Input: last 4 frames
- Output: 18 joystick positions
- Reward: change of score
Stability in Deep Reinforcement Learning
Stability problems

- Naively, Q-Learning oscillates or diverges with neural networks
- Why?
Stability problems

- Naively, Q-Learning oscillates or diverges with neural networks
- Why?
  - Sequential data breaks IID assumption
    - Highly correlated samples break SGD
- However, this is not specific to RL, as we have seen earlier
Stability problems

- Naively, Q-Learning oscillates or diverges with neural networks
- Why?
The learning objective is
\[ \mathcal{L}(\theta) = \mathbb{E}\left[ (r + \gamma \max_{a'} Q(s', a', \theta) - Q(s, a, \theta))^2 \right] \]

The target depends on the \( Q \) function also. This means that if we update the current \( Q \) function with backprop, the target will also change.

Plus, we know neural networks are highly non-convex.

Policy changes will change fast even with slight changes in the \( Q \) function.
  - Policy might oscillate
  - Distribution of data might move from one extreme to another

Stability problems
Stability problems

- Naively, Q-Learning oscillates or diverges with neural networks
- Why?
Stability problems

- Not easy to control the scale of the $Q$ values $\Rightarrow$ gradients are unstable $Q$
- Remember, the $Q$ function is the output of a neural network
- There is no guarantee that the outputs will lie in a certain range
  - Unless care is taken
- Naïve $Q$ gradients can be too large, or too small $\Rightarrow$ generally unstable and unreliable
- Where else did we observe a similar behavior?
Improving stability: Experience replay

- Replay memory/Experience replay
- Store memories $< s, a, r, s' >$
- Train using random stored memories instead of the latest memory transition
- Breaks the temporal dependencies – SGD works well if samples are roughly independent
- Learn from all past policies
Experience replay

- Take action $a_t$ according to $\varepsilon$-greedy policy
- 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 $D$
- Optimize mean squared error using the mini-batch

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s') \sim D} \left[ (r + \gamma \max_{a'} Q(s', a', \theta) - Q(s, a, \theta))^2 \right]$$

- Effectively, update your network using random past inputs (experience), not the ones the agent currently sees
Improving stability: Freeze target $Q$ network

- Instead of having “moving” targets, have two networks
  - One Q-Learning and one Q-Target networks

- Copy the $Q$ network parameters to the target network every $K$ iterations
  - Otherwise, keep the old parameters between iterations
  - The targets come from another (Q-Target) network with slightly older parameters

- Optimize the mean squared error as before, only now the targets are defined by the “older” $Q$ function
  $$\mathcal{L}(\theta) = \mathbb{E}[(r + \gamma \max_a Q(s', a', \theta_{old}) - Q(s, a, \theta))^2]$$

- Avoids oscillations
Improving stability: Take care of rewards

- Clip rewards to be in the range $[-1, +1]$
- Or normalize them to lie in a certain, stable range
- Can’t tell the difference between large and small rewards
## Results

<table>
<thead>
<tr>
<th>Game</th>
<th>Q-learning</th>
<th>Q-learning + Target Q</th>
<th>Q-learning + Replay</th>
<th>Q-learning + Target Q + Replay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakout</td>
<td>3</td>
<td>10</td>
<td>241</td>
<td>317</td>
</tr>
<tr>
<td>Enduro</td>
<td>29</td>
<td>142</td>
<td>831</td>
<td>1006</td>
</tr>
<tr>
<td>River Raid</td>
<td>1453</td>
<td>2868</td>
<td>4103</td>
<td>7447</td>
</tr>
<tr>
<td>Seaquest</td>
<td>276</td>
<td>1003</td>
<td>823</td>
<td>2894</td>
</tr>
<tr>
<td>Space Invaders</td>
<td>302</td>
<td>373</td>
<td>826</td>
<td>1089</td>
</tr>
</tbody>
</table>
Some extra tricks

- Skipping frames
  - Saves time and computation
  - Anyways, from one frame to the other there is often very little difference

- \( \varepsilon \)-greedy behavioral policy with annealed temperature during training
  - Select random action (instead of optimal) with probability \( \varepsilon \)
  - In the beginning of training our model is bad, no reason to trust the “optimal” action

- Alternatively: Exploration vs exploitation
  - early stages \(\rightarrow\) strong exploration
  - late stages \(\rightarrow\) strong exploitation
Policy-based Deep RL
Policy Optimization

- Problems with modelling the $Q$-value function
  - Often too expensive $\rightarrow$ must take into account all possible states, actions $\rightarrow$ Imagine when having continuous or high-dimensional action spaces
  - Not always good convergence $\leftarrow$ Oscillations

- Often learning directly a policy $\pi_\theta(a|s)$ that gives the best action without knowing what its expected future reward is easier

- Also, allows for stochastic policies $\leftarrow$ no exploration/exploitation dilemma

- Model optimal action value with a non-linear function approximator
  \[ Q^*(s, a) \approx Q(s, a; w) \]
Policy Optimization

\[ \pi_\theta(a|s) \]

Slide inspired by P. Abbeel
Policy Optimization

- Train learning agent for the optimal policy $\pi_w(a|s)$ given states $s$ and possible actions $a$
- The policy class can be either deterministic or stochastic

$\pi_w(a|s)$

Slides inspired by P. Abbeel
Policy Optimization

- Use a deep network as a non-linear approximator that finds an optimal policy by maximizing $Q(s, a; \theta)$

$$
\mathcal{L}(w) = Q(s, a; w) = \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots | \pi_w(s_t, a_t)]
$$

- If policy is deterministic

$$
\frac{\partial \mathcal{L}}{\partial w} = \mathbb{E} \left[ \frac{\partial \log \pi(a | s, w)}{\partial w} Q^\pi(s, a) \right]
$$

- If policy is stochastic $a = \pi(s)$

$$
\frac{\partial \mathcal{L}}{\partial w} = \mathbb{E} \left[ \frac{\partial Q^\pi(s, a)}{\partial a} \frac{\partial a}{\partial w} \right]
$$

- To compute gradients use the log-derivative trick (REINFORCE algorithm (Williams, 1992))

$$
\nabla_\theta \log p(x; \theta) = \frac{\nabla_\theta p(x; \theta)}{p(x; \theta)}
$$
Asynchronous Advantage Actor-Critic (A3C)

- Estimate Value function
  \[ V(s, v) = \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \cdots | s] \]

- Estimate the Q value after \( n \) steps
  \[ q_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n}, v) \]

- Update actor by
  \[ \frac{\partial \mathcal{L}_{actor}}{\partial w} = \frac{\partial \log \pi(a_t | s_t, w)}{\partial w} (q_t - V(s_t, v)) \]
A3C in labyrinth

- End-to-end learning of softmax policy from pixels
- Observations are the raw pixels
- The state is implemented as an LSTM
- Outputs value $V(s)$ and softmax over actions $\pi(a|s)$
- Task
  - Collect apples (+1)
  - Escape (+10)
- Demo

Diagram:

- $\pi(a|s_{t-1}) V(s_{t-1})$
- $\pi(a|s_t) V(s_t)$
- $\pi(a|s_{t+1}) V(s_{t-1})$

Images:

- $s_{t-1}$
- $s_t$
- $s_{t+1}$

- $o_{t-1}$
- $o_t$
- $o_{t+1}$
Model-based Deep RL
Learning models of the environment

- Often quite challenging because of cumulative errors
- Errors in transition models accumulate over trajectory
- Planning trajectories are different from executed trajectories
- At the end of a long trajectory final rewards are wrong
- Can be better if we know the rules
At least $10^{10^{48}}$ possible game states
  - Chess has $10^{120}$

Monte Carlo Tree Search used mostly
  - Start with random moves and evaluate how often they lead to victory
  - Learn the value function to predict the quality of a move
  - Exploration-exploitation trade-off

Tic-Tac-Toe possible game states
AlphaGo

- AlphaGo relies on a tree procedure for search
- AlphaGo relies on ConvNets to guide the tree search
- A ConvNet trained to predict human moves achieved 57% accuracy
  - Humans make intuitive moves instead of thinking too far ahead
- For Deep RL we don’t want to predict human moves
  - Instead, we want the agent to learn the optimal moves
- Two policy networks (one per side) + One value network
- Value network trained on 30 million positions while policy networks play
Both humans and Deep RL agents play better end games
  • Maybe a fundamental cause?

In the end the value of a state is computed equally from Monte Carlo simulation and the value network output
  • Combining intuitive play and thinking ahead

Where is the catch?
Both humans and Deep RL agents play better end games
  - Maybe a fundamental cause?

In the end, the value of a state is computed equally from Monte Carlo simulation and the value network output
  - Combining intuitive play and thinking ahead

Where is the catch?

State is not the pixels but positions

Also, the game states and actions are highly discrete
Summary

- Reinforcement Learning
- Q-Learning
- Deep Q-Learning
- Policy-based Deep RL
- Model-based Deep RL
- Making Deep RL stable