Reinforcement Learning

- Variation on Supervised Learning
- Exact target outputs are not given
- Some variation of reward is given either immediately or after some steps
  - Chess
  - Path Discovery
- RL systems learn a mapping from states to actions by trial-and-error interactions with a dynamic environment
- TD-Gammon (Neuro-Gammon)
- Deep RL – Showing tremendous potential
  - Atari games
difficult and engaging for human players. We used the same network architecture, hyperparameter values (see Extended Data Table 1) and learning procedure throughout—taking high-dimensional data (210 x 160 colour video at 60 Hz) as input—to demonstrate that our approach robustly learns successful policies over a variety of games based solely on sensory inputs with only very minimal prior knowledge (that is, merely the input data were visual images, and the number of actions available in each game, but not their correspondences; see Methods). Notably, our method was able to train large neural networks using a reinforcement learning signal and stochastic gradient descent in a stable manner—illustrated by the temporal evolution of two indices of learning (the agent’s average score-per-episode and average predicted Q-values; see Fig. 2 and Supplementary Discussion for details).

We compared DQN with the best performing methods from the reinforcement learning literature on the 49 games where results were available. In addition to the learned agents, we also report scores for a professional human games tester playing under controlled conditions and a policy that selects actions uniformly at random (Extended Data Table 2 and Fig. 3, denoted by 100% (human) and 0% (random) on y-axis; see Methods). Our DQN method outperforms the best existing reinforcement learning methods on 43 of the games without incorporating any of the additional prior knowledge about Atari 2600 games used by other approaches (for example, refs 12, 15). Furthermore, our DQN agent performed at a level that was comparable to that of a professional human games tester across the set of 49 games, achieving more than 75% of the human score on more than half of the games (29 games; see Convolution Convolution Fully connected Fully connected No input for details).

Figure 1 | Schematic illustration of the convolutional neural network. The details of the architecture are explained in the Methods. The input to the neural network consists of an 84 x 3 x 84 x 3 image produced by the preprocessing map \( w \), followed by three convolutional layers (note: snaking blue line symbolizes sliding of each filter across input image) and two fully connected layers with a single output for each valid action. Each hidden layer is followed by a rectifier nonlinearity (that is, \( \text{max}(0, x) \)).

Figure 2 | Training curves tracking the agent’s average score and average predicted action-value. a, Each point is the average score achieved per episode after the agent is run with \( \epsilon \)-greedy policy (\( \epsilon = 0.05 \)) for 520 k frames on Space Invaders. b, Average score achieved per episode for Seaquest. c, Average predicted action-value on a held-out set of states on Space Invaders. Each point on the curve is the average of the action-value \( Q \) computed over the held-out set of states. Note that \( Q \)-values are scaled due to clipping of rewards (see Methods).

c, Average predicted action-value on Seaquest. See Supplementary Discussion for details.
RL Basics

- Agent (sensors and actions)
- Can sense state of Environment (position, etc.)
- Agent has a set of possible actions
- Actual rewards for actions from a state are usually delayed and do not give direct information about how best to arrive at the reward
- RL seeks to learn the optimal policy: which action should the agent take given a particular state to achieve the agents goals (e.g. maximize reward)
Learning a Policy

- Find optimal policy $\pi: S \rightarrow A$
- $a = \pi(s)$, where $a$ is an element of $A$, and $s$ an element of $S$
- Which actions in a sequence leading to a goal should be rewarded, punished, etc. – Temporal Credit assignment problem
- Exploration vs. Exploitation – To what extent should we explore new unknown states (hoping for better opportunities) vs. taking the best possible action based on the knowledge already gained
  - The restaurant problem
- Markovian? – Do we just base action decision on current state or is there some memory of past states – Basic RL assumes Markovian processes (action outcome is only a function of current state, state fully observable) – Does not directly handle partially observable states (i.e. states which are not unambiguously identified) – can still approximate
  - Deep Reinforcement Learning can help overcome
Rewards

- Assume a reward function $r(s,a)$ – Common approach is a positive reward for entering a goal state (win the game, get a resource, etc.), negative for entering a bad state (lose the game, lose resource, etc.), 0 for all other transitions.
- Could also make all reward transitions -1, except for 0 going into the goal state, which would lead to finding a minimal length path to a goal
- Discount factor $\gamma$: between 0 and 1, future rewards are discounted
- Value Function $V(s)$: The value of a state is the sum of the discounted rewards received when starting in that state and following a fixed policy until reaching a terminal state
- $V(s)$ also called the Discounted Cumulative Reward

\[
V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{i=0}^{\infty} \gamma^i r_{t+i}
\]
4 possible actions: N, S, E, W

<table>
<thead>
<tr>
<th>Reward Function</th>
<th>V(s) with optimal policy and $\gamma = 1$</th>
<th>V(s) with optimal policy and $\gamma = .9$</th>
<th>One Optimal Policy</th>
<th>V(s) with random policy and $\gamma = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0   -1  -1  -1</td>
<td>0   0   -1  -2</td>
<td>0   1  .90  .81</td>
<td><img src="image1" alt="One Optimal Policy" /></td>
<td><img src="image2" alt="V(s) with random policy and $\gamma = 1$" /></td>
</tr>
<tr>
<td>-1  -1  -1  -1</td>
<td>0   -1  -2  -1</td>
<td>1   .90 .81 .90</td>
<td><img src="image3" alt="One Optimal Policy" /></td>
<td><img src="image4" alt="V(s) with random policy and $\gamma = 1$" /></td>
</tr>
<tr>
<td>-1  -1  -1  -1</td>
<td>-1  -2  -1  0</td>
<td>.90  .81 .90  1</td>
<td><img src="image5" alt="One Optimal Policy" /></td>
<td><img src="image6" alt="V(s) with random policy and $\gamma = 1$" /></td>
</tr>
<tr>
<td>-1  -1  -1  0</td>
<td>-2  -1  0  0</td>
<td>.81  .90  1  0</td>
<td><img src="image7" alt="One Optimal Policy" /></td>
<td><img src="image8" alt="V(s) with random policy and $\gamma = 1$" /></td>
</tr>
</tbody>
</table>

$V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$
4 possible actions: N, S, E, W

<table>
<thead>
<tr>
<th>Reward Function</th>
<th>V(s) with random policy and $\gamma = 1$</th>
<th>V(s) with optimal policy and $\gamma = 1$</th>
<th>One Optimal Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>Reward Function</td>
<td>V(s) with random policy and $\gamma = .9$</td>
<td>V(s) with optimal policy and $\gamma = .9$</td>
<td>One Optimal Policy</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------------------------</td>
<td>------------------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

$$V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$
Policy vs. Value Function

- Goal is to learn the optimal policy

$$\pi^* \equiv \arg \max_{\pi} V^\pi(s), (\forall s)$$

- $V^*(s)$ is the value function of the optimal policy. $V(s)$ is the value function of the current policy.
- $V(s)$ is fixed for the current policy and discount factor
- Typically start with a random policy – Effective learning happens when rewards from terminal states start to propagate back into the value functions of earlier states
- $V(s)$ can be represented with a lookup table and will be used to iteratively update the policy (and thus update $V(s)$ at the same time)
- For large or real valued state spaces, lookup table is too big, thus must approximate the current $V(s)$. Any adjustable function approximator (e.g. neural network) can be used.
Policy Iteration

Let $\pi$ be an arbitrary initial policy

Repeat until $\pi$ unchanged

For all states $s$

$$V^\pi(s) = \sum_{s'} P(s' | s, \pi(s), s') \cdot [R(s, \pi(s), s') + \gamma V^\pi(s')]$$

For all states $s$

$$\pi'(s) = \arg \max_a \sum_{s'} P(s' | s, a) \cdot [R(s, a, s') + \gamma V^\pi(s')]$$

- In policy iteration the equations just calculate one state ahead rather than continue to an absorbing state
- To execute directly, must know the probabilities of state transition function and the exact reward function
- Also usually must be learned with a model doing a simulation of the environment. If not, how do you do the argmax which requires trying each possible action. In the real world, you can’t have a robot try one action, backup, try again, etc. (e.g. environment may change because of the action, etc.)
Q-Learning

- No model of the world required – Just try one action and see what state you end up in and what reward you get. Update the policy based on these results. This can be done in the real world and is thus more widely applicable.
- Rather than find the value function of a state, find the value function of a \((s,a)\) pair and call it the Q-value.
- Only need to try actions from a state and then incrementally update the policy.
- \(Q(s,a) = \text{Sum of discounted reward for doing } a \text{ from } s \text{ and following the optimal policy thereafter}\)

\[
Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a)) = r(s,a) + \gamma \max_{a'} Q(s',a')
\]

\[
\pi^*(s) = \arg \max_a Q(s,a)
\]
\( r(s, a) \) (immediate reward) values

\( Q(s, a) \) values \hspace{1cm} V^*(s) \) values

One optimal policy

FIGURE 13.2
A simple deterministic world to illustrate the basic concepts of \( Q \)-learning. Each grid square represents a distinct state, each arrow a distinct action. The immediate reward function, \( r(s, a) \) gives reward 100 for actions entering the goal state \( G \), and zero otherwise. Values of \( V^*(s) \) and \( Q(s, a) \) follow from \( r(s, a) \), and the discount factor \( \gamma = 0.9 \). An optimal policy, corresponding to actions with maximal \( Q \) values, is also shown.
Learning Algorithm for Q function

- Create a table with a cell for every state and \((s,a)\) pair with zero or random initial values for the hypothesis of the \(Q\) values which we represent by \(\hat{Q}\).
- Iteratively try different actions from different states and update the table based on the following learning rule (for Deterministic environment):

\[
\hat{Q}(s, a) = r(s,a) + \gamma \max_{a'} \hat{Q}(s', a')
\]

- Note that this slowly adjusts the estimated Q-function towards the true Q-function. Iteratively applying this equation will in the limit converge to the actual Q-function if:
  - The system can be modeled by a deterministic Markov Decision Process – action outcome depends only on current state (not on how you got there).
  - \(r\) is bounded \((r(s,a) < c\) for all transitions).
  - Each \((s,a)\) transition is visited infinitely many times.
Learning Algorithm for Q function

Until Convergence (Q-function not changing or changing very little)
Start in an arbitrary \( s \)
Select an action \( a \) and execute (exploitation vs. exploration)
Update the Q-function table entry

\[
\hat{Q}(s, a) = r(s, a) + \gamma \max_{a'} \hat{Q}(s', a')
\]

- Could also continue \( (s \rightarrow s') \) until an absorbing state is reached (episode) at which point can start again at an arbitrary \( s \).
- But sufficient to choose a new \( s \) at each iteration and just go one step.
- Do not need to know the actual reward and state transition functions. Just sample them (Model-less).
\[ \hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \]
\[ \leftarrow 0 + 0.9 \max\{63, 81, 100\} \]
\[ \leftarrow 90 \]

**FIGURE 13.3**
The update to \( \hat{Q} \) after executing a single action. The diagram on the left shows the initial state \( s_1 \) of the robot (R) and several relevant \( \hat{Q} \) values in its initial hypothesis. For example, the value \( \hat{Q}(s_1, a_{right}) = 72 \), where \( a_{right} \) refers to the action that moves R to its right. When the robot executes the action \( a_{right} \), it receives immediate reward \( r = 0 \) and transitions to state \( s_2 \). It then updates its estimate \( \hat{Q}(s_1, a_{right}) \) based on its \( \hat{Q} \) estimates for the new state \( s_2 \). Here \( \gamma = 0.9 \).
Example - Chess

- Assume reward of 0’s except win (+10) and loss (-10)
- Set initial Q-function to all 0’s
- Start from any initial state (could be normal start of game) and choose transitions until reaching an absorbing state (win or lose)
- During all the earlier transitions the update was applied but no change was made since rewards were all 0.
- Finally, after entering absorbing state, \( Q(s_{pre}, a_{pre}) \), the preceding state-action pair, gets updated (positive for win or negative for loss).
- Next time around a state-action entering \( s_{pre} \) will be updated and this progressively propagates back with more iterations until all state-action pairs have the proper Q-function.
- If other actions from \( s_{pre} \) also lead to the same outcome (e.g. loss) then Q-learning will learn to avoid this state altogether (however, remember it is the max action out of the state that sets the actual Q-value)
Possible States for Chess
Exploration vs Exploitation

- Choosing action during learning (Exploitation vs. Exploration) – 2 Common approaches
- Softmax

\[ P(a_i | s) = \frac{k \hat{Q}(s,a_i)}{\sum_j k \hat{Q}(s,a_j)} \]

- Can increase \( k \) (constant \( >1 \)) over time to move from exploration to exploitation
- \( \varepsilon \)-greedy – With probability \( \varepsilon \) randomly choose any action, else greedily take the action with the best current Q value.
  - Start \( \varepsilon \) at 1 and then decrease with time
Sequence of Update – Note that much efficiency could be gained if you worked back from the goal state, etc. However, with model free learning, we do not know where the goal states are, or what the transition function is, or what the reward function is. We just sample things and observe. If you do know these functions then you can simulate the environment and come up with more efficient ways to find the optimal policy with standard DP algorithms (e.g. policy iteration).

One thing you can do for Q-learning is to store the path of an episode and then when an absorbing state is reached, propagate the discounted Q-function update all the way back to the initial starting state. This can speed up learning at a cost of memory.

- **TD(λ):** TD(λ) updates back to beginning of episode using normal discount $\gamma$ but also multiplies updates before the final update by a discount of $\lambda$ at each step. TD(0) just updates one state back, TD(1) updates all the way, but just using the base discount value of $\gamma$.

- Monotonic Convergence
Q-Learning in Non-Deterministic Environments

Both the transition function and reward functions could be non-deterministic

\[ Q^*(s, a) = E[r(s, a) + \gamma \max_{a'} Q^*(s', a')] \]

In this case the previous algorithm will not monotonically converge

Though more iterations may be required, we simply replace the update function with

\[ \hat{Q}_n(s, a) = (1 - \alpha_n) \hat{Q}_{n-1}(s, a) + \alpha_n [r(s, a) + \gamma \max_{a'} \hat{Q}_{n-1}(s', a')] \]

\[ = \hat{Q}_{n-1}(s, a) + \alpha_n [r(s, a) + \gamma \max_{a'} \hat{Q}_{n-1}(s', a') - \hat{Q}_{n-1}(s, a)] \]

where \( \alpha_n \) starts at 1 and decreases over time and \( n \) stands for the \( n^{th} \) iteration. An example of \( \alpha_n \) is

\[ \alpha_n = \frac{1}{1 + \text{# of visits}(s, a)} \]

Large variations in the non-deterministic function are muted and an overall averaging effect is attained (like a small learning rate in neural network learning)
Replace Q-table with a Function Approximator

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

- Train a function approximator (e.g. neural network) to output approximate Q-values
  - Avoid huge lookup table
  - Allows generalization from all states, not just those seen during training

- To overcome Markov limitation (partially observed states) the function approximator can be given an input made up of \( m \) consecutive proceeding states (Atari approach) or have memory (e.g. recurrent NN), etc.
  - Early Q learning used linear models or shallow neural networks

- Using deep networks as the approximator is shown to lead to accurate stable learning
Deep Q Network – 49 Classic Atari Games
Reinforcement Learning Summary

- Learning can be slow even for small environments
- Large and continuous spaces are difficult (need to generalize on states not seen before) – Use a function approximator
  - One common approach is to use a neural network in place of the lookup table, where it is trained with the inputs $s$ and $a$ and the goal Q-value as output. It can then generalize to cases not seen in training. Can also use real valued states and actions.
  - Latest – Deep Q learning, states and Q function represented by a deep neural network. Learns all 49 classic Atari games with the only inputs being pixels from the screen and the score, at above standard human playing level.
- Q-learning lets you do RL without any pre-knowledge of the environment
- Partially observable states – There are many Non-Markovian problems (“there is a wall in front of me” could represent many different states), Could keep some past memory to disambiguate states, etc. Deep RL overcomes this by having the input made up of multiple contextual states
- Don’t need pre-labeled data. Just experiment and learn!