Deep Reinforcement Learning - Deep Q Learning

Most of this material is from Naiyan Wang and Petar Veličković

Part 1

Q Learning

Q Learning

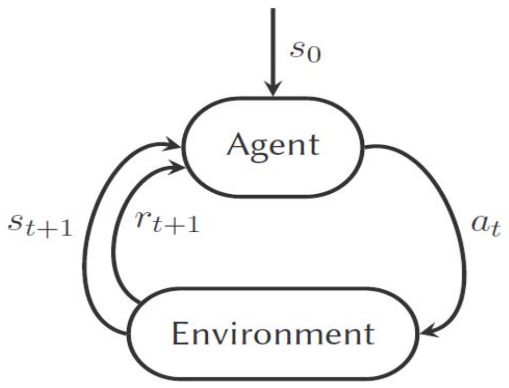
State Action Reward

Motivation

- In this talk I will provide a brief introduction to reinforcement learning, specifically focusing on the Deep Q-Network (DQN) and its variants.
- Deep reinforcement learning has become extremely popular in some of the recent major advances in artificial intelligence (*Atari* player, *AlphaGo*, . . .)

Context

We want to learn how to *act optimally* in a very general environment—performing state-changing actions on the environment which carry a reward signal.



Reinforcement learning

- The environment assumes a current state s out of a set S, and the agent may perform actions out of an action set A.
- There are also two functions: $\mathcal{S} : S \times A \rightarrow S$ and $\mathcal{R} : S \times A \rightarrow \mathbb{R}$, which the agent does not typically have access to (can be *stochastic*).
- After performing action a in state s, the environment assumes state $\mathcal{S}(s, a)$, and the agent gets a *reward* signal $\mathcal{R}(s, a)$.
- Our goal is to learn a *policy function*, $p : S \rightarrow A$, choosing actions that *maximise* expected future rewards.

Cumulative reward metric

- Aside from the aforementioned goal, it is also desirable to receive positive rewards *as soon as possible* (i.e. we assign larger importance to more *immediate* rewards).
- This is typically done by defining a *discount factor* $\gamma \in [0, 1)$, used to scale future rewards in the total value of a policy:
$$V^p(s) = \sum_{k=1}^{+\infty} \gamma^{k-1} r_k = r_1 + \gamma r_2 + \gamma^2 r_3 + \dots$$
where r_k is the reward obtained after k steps, starting from state s and following policy p thereafter.
- Such a definition is required, among other things, to assure *convergence* of the relevant learning algorithms.

Q function

- As mentioned, we want to find an optimal policy $p_{\text{opt}} : S \to A$, where $p_{\text{opt}}(s) = \operatorname{argmax}_p \{V^p(s)\}$ for any state s. We also denote $V_{\text{opt}} = V^{p_{\text{opt}}}$.
- Define a *Q* function, for all states s and actions a:
$$Q(s, a) = \mathcal{R}(s, a) + \gamma V_{\text{opt}}(\mathcal{S}(s, a))$$
corresponding to the reward obtained upon executing action a from state s and assuming *optimal behaviour* thereafter.
- Learning the Q function is sufficient for determining the optimal policy, as:
$$p_{\text{opt}}(s) = \operatorname*{argmax}_a \{Q(s, a)\}$$

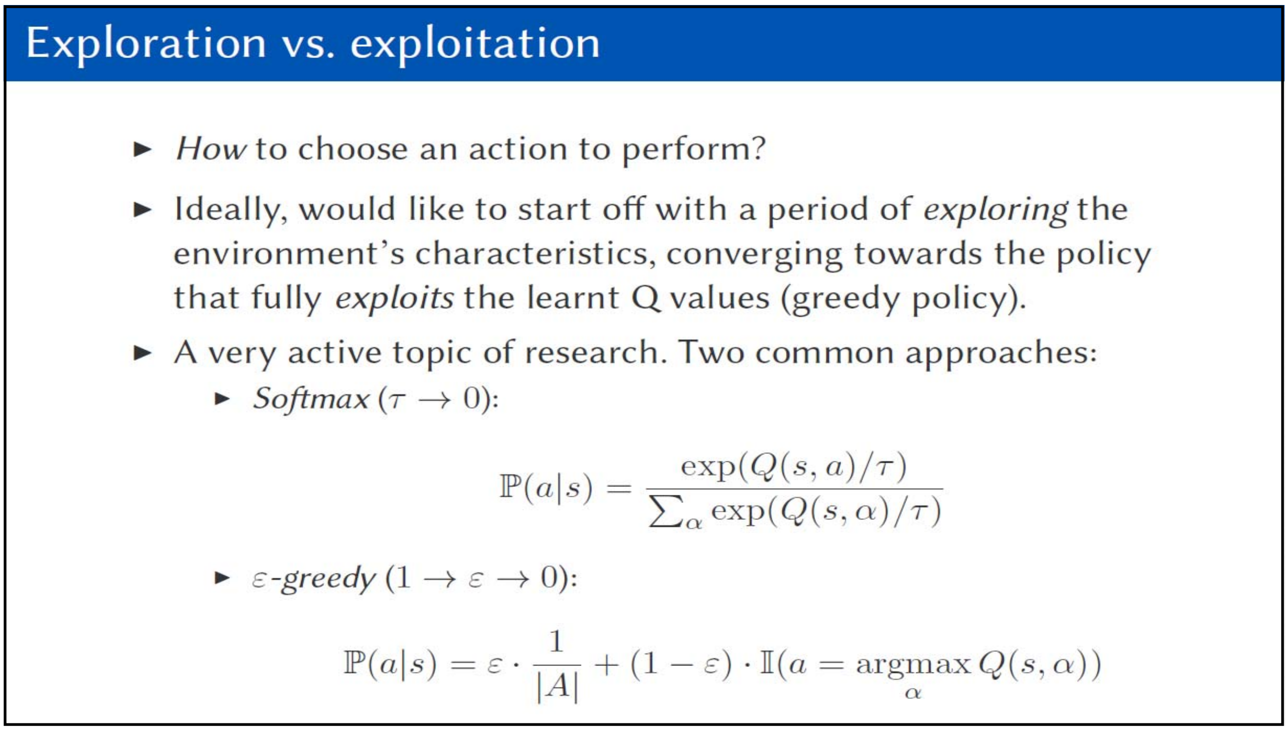
Towards an iterative method

- Note that, for any state s, $V_{\text{opt}}(s) = \max_\alpha \{Q(s, \alpha)\}$.
- Factoring this into the definition of Q:
$$Q(s, a) = \mathcal{R}(s, a) + \gamma \max_\alpha \{Q(\mathcal{S}(s, a), \alpha)\}$$
- This hints at the fact that, if Q' is an **estimate** of Q, then
$$\mathcal{R}(s, a) + \gamma \max_\alpha \{Q'(\mathcal{S}(s, a), \alpha)\}$$
will be a no-worse estimate of $Q(s, a)$ than $Q'(s, a)$.
- Thus far, we represent Q' as a table of current estimates of Q for all elements of $S \times A$.

Q-learning

Initialise the Q' table with random values.

1. Choose an action a to perform in the current state, s.
2. Perform a and receive reward $\mathcal{R}(s, a)$.
3. Observe the new state, $\mathcal{S}(s, a)$.
4. Update:
$$Q'(s, a) \leftarrow \mathcal{R}(s, a) + \gamma \max_\alpha \{Q'(\mathcal{S}(s, a), \alpha)\}$$
5. If the next state is not terminal, go back to step 1.



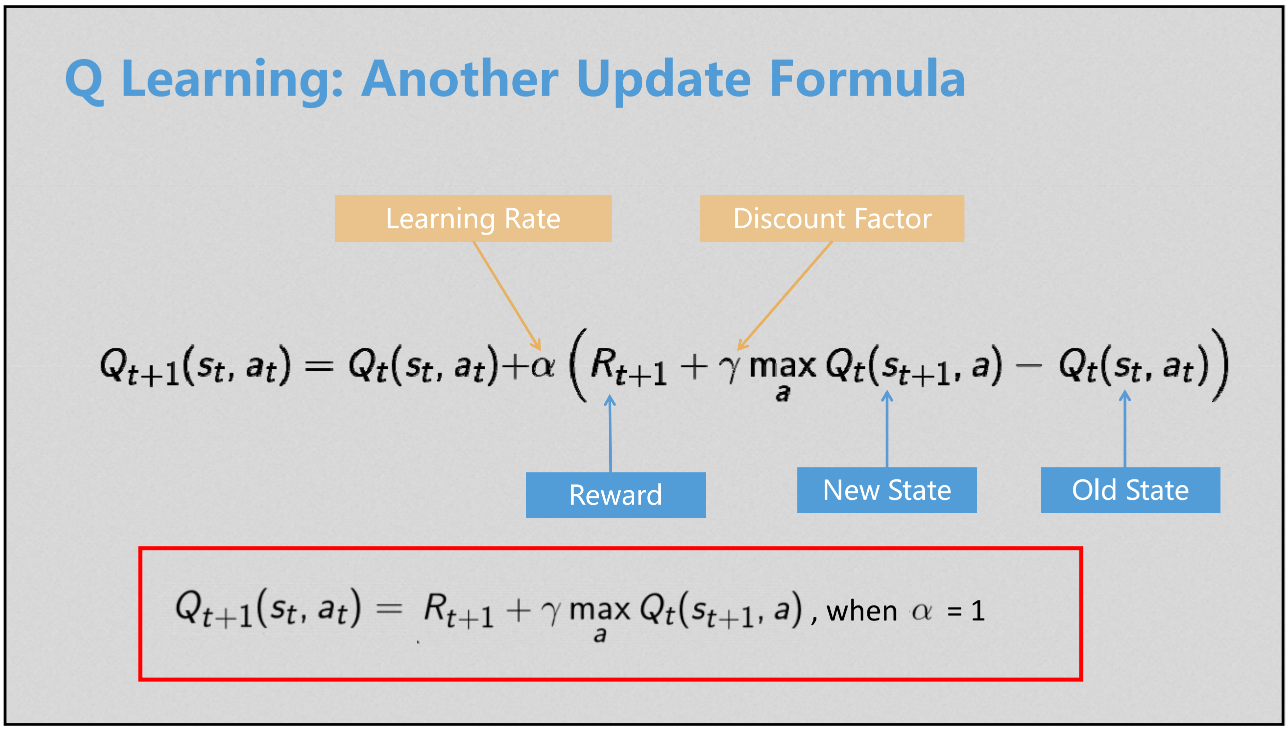
Exploration vs. exploitation

- *How* to choose an action to perform?
- Ideally, would like to start off with a period of *exploring* the environment's characteristics, converging towards the policy that fully *exploits* the learnt Q values (greedy policy).
- A very active topic of research. Two common approaches:
  - *Softmax* ($\tau \to 0$):

$$\mathbb{P}(a|s) = \frac{\exp(Q(s,a)/\tau)}{\sum_{\alpha} \exp(Q(s,\alpha)/\tau)}$$

  - *ε-greedy* ($1 \to \varepsilon \to 0$):

$$\mathbb{P}(a|s) = \varepsilon \cdot \frac{1}{|A|} + (1-\varepsilon) \cdot \mathbb{I}(a = \operatorname*{argmax}_{\alpha} Q(s,\alpha))$$



Q Learning: Another Update Formula

Learning Rate

Discount Factor

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha \left(R_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t) \right)$$

Reward

New State

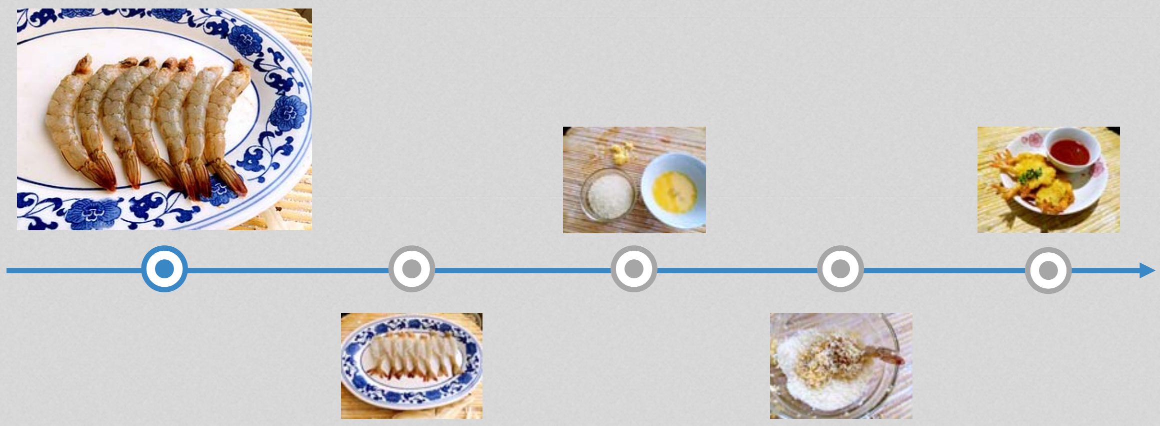
Old State

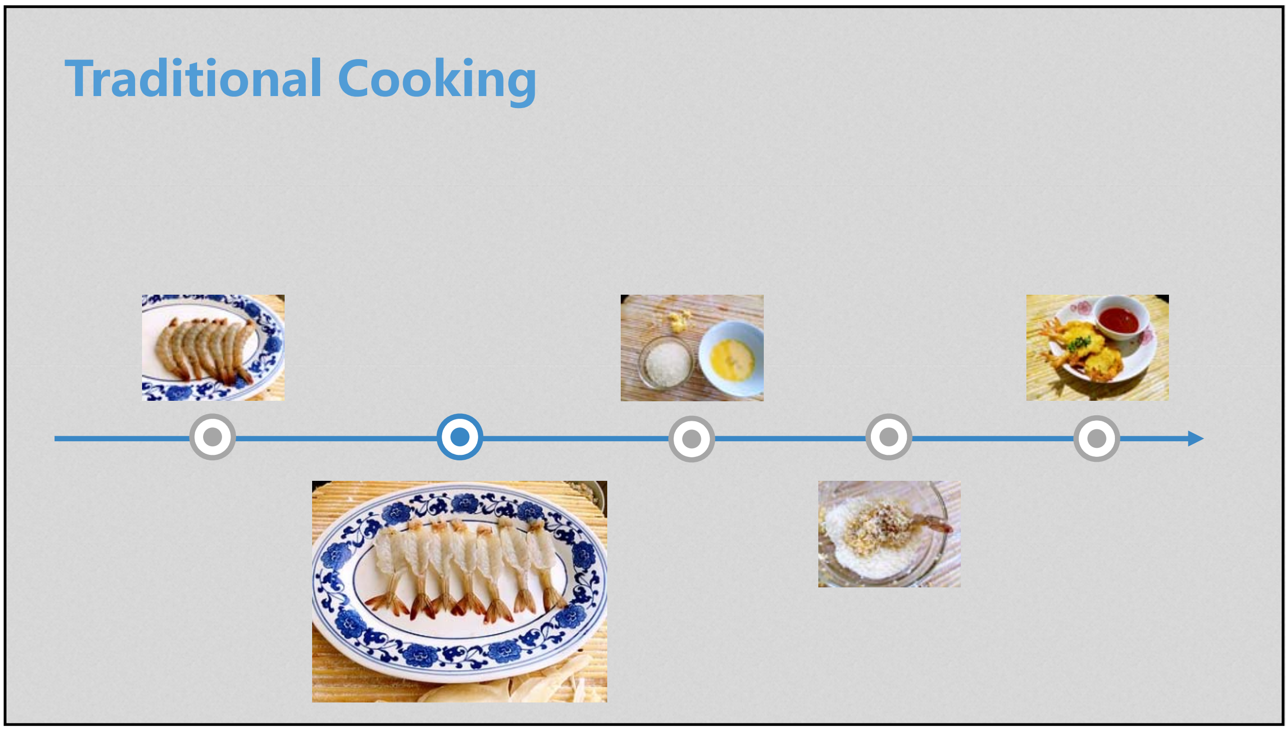
$$\boxed{Q_{t+1}(s_t, a_t) = R_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) \text{, when } \alpha = 1}$$

Part 2

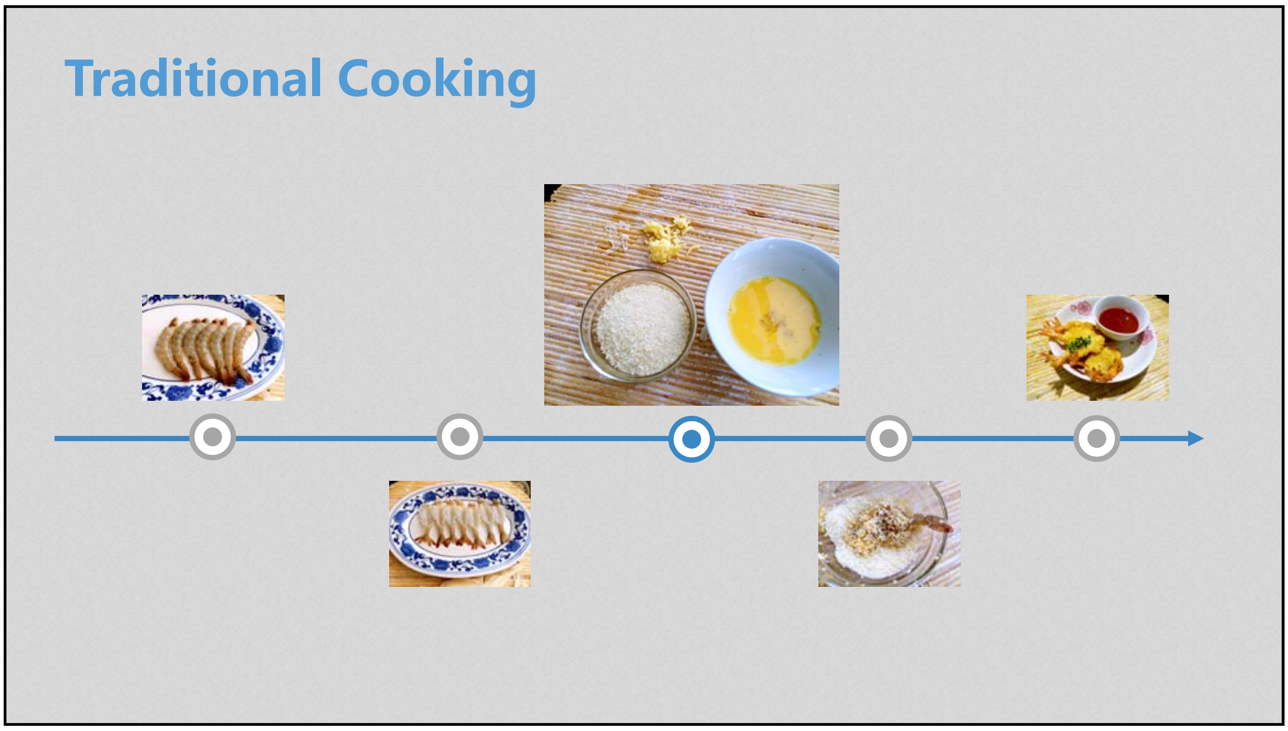
Deep Q Learning

Traditional Cooking



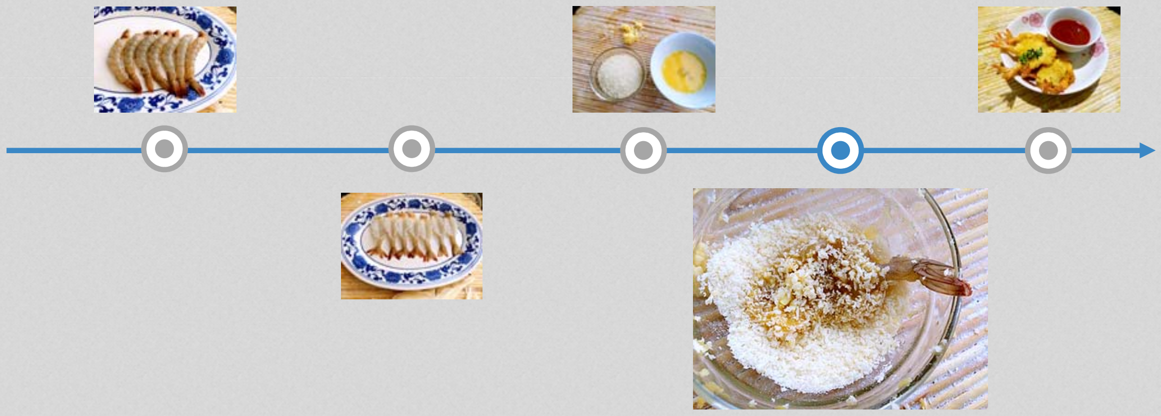


Traditional Cooking

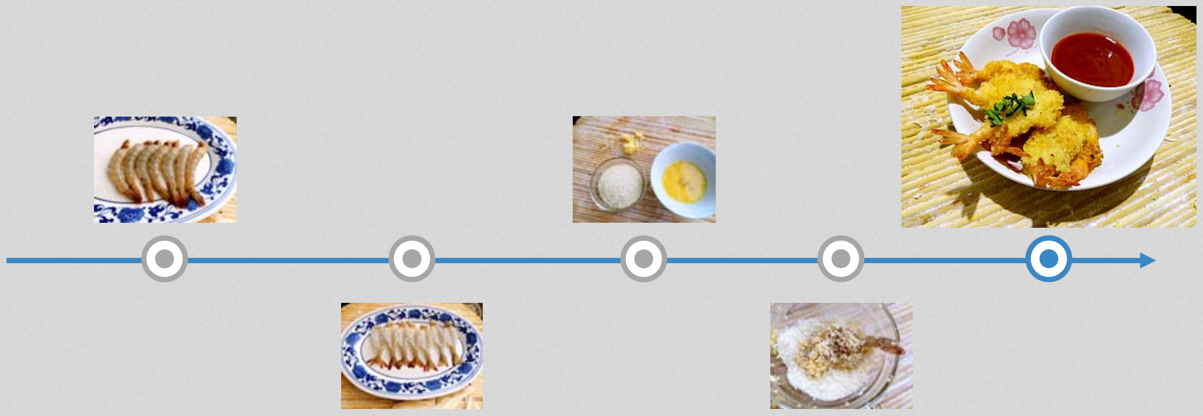


Traditional Cooking

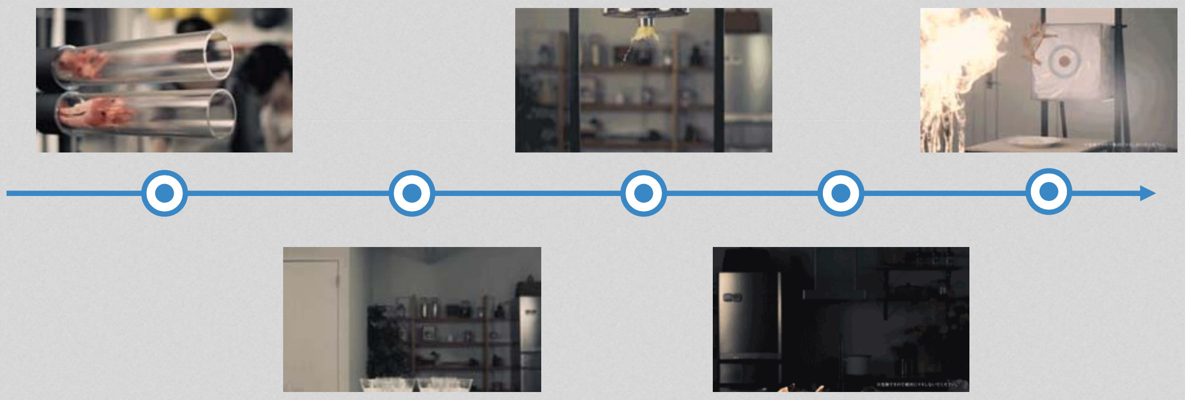
Traditional Cooking



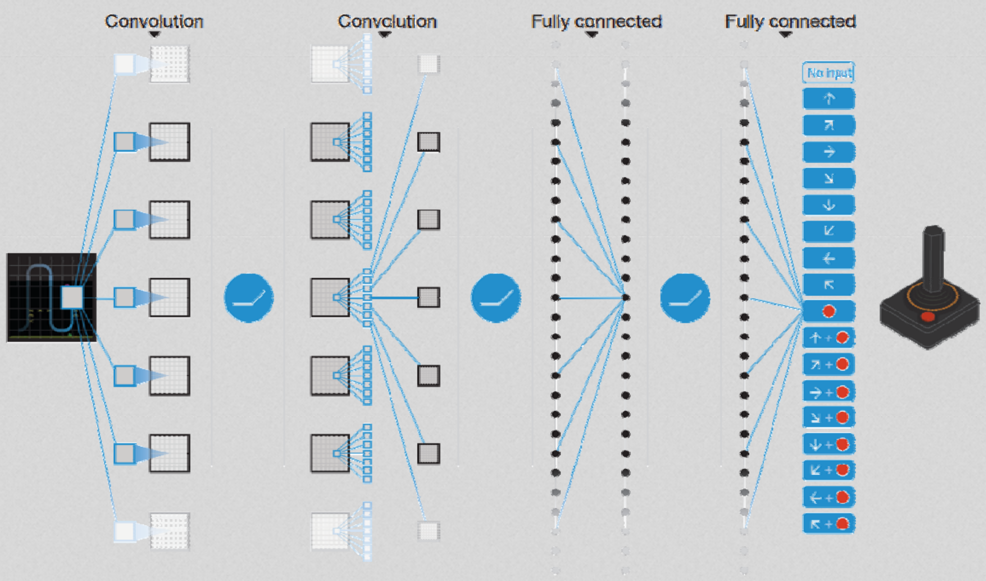
Traditional Cooking



End-to-End Cooking



End-to-End Deep Q Learning



Playing Atari

- We now turn our attention to the initial success story that linked deep and reinforcement learning, allowing multiple Atari games to be played at superhuman level using the same basic algorithm. (Mnih *et al.* (2013), *NIPS*)

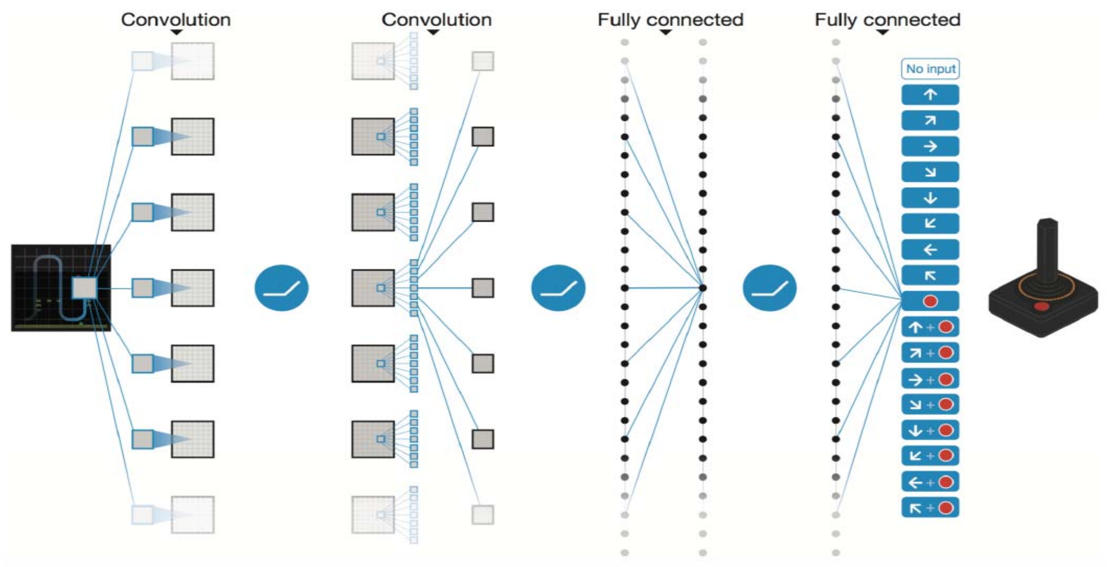
Setup

- No prior knowledge about the game itself is incorporated—the algorithm perceives solely the RGB matrices of pixels in the framebuffer as states, and changes in score as rewards.
- Actions correspond to joypad inputs (along with no-op).
- Further modifications:
  - The colour information is discarded—the matrices are **converted to grayscale** before further processing;
  - A sequence of **six previous frames** is used as a single state (to account for the fact that the system is only partially observed by the current frame);
  - The rewards are all **represented as** ± 1 (to avoid scaling issues between different games).

Q-learning, revisited

- Should we be able to successfully learn the values of the Q function for all pixel matrices and joypad input pairs, we become capable of playing any Atari game.
- **Major issue**: *we cannot store the entire Q table in memory!*
- It needs to be **approximated** somehow...
- The authors used a deep neural network (dubbed a "Deep Q-Network") as the approximator. This builds up on successes of a shallow neural network as an approximator for playing backgammon (TD-gammon).

Deep Q-Network



Conv [16] $\rightarrow$ ReLU $\rightarrow$ Conv [32] $\rightarrow$ ReLU $\rightarrow$ FC [256] $\rightarrow$ ReLU $\rightarrow$ FC [|A|]

Deep Q-learning

Initialise an empty replay memory.
Initialise the DQN with random (small) weights.

1. Choose an action a to perform in the current state, s, using an ε-greedy strategy (with ε annealed from 1.0 to 0.1).
2. Perform a and receive reward $\mathcal{R}(s, a)$.
3. Observe the new state, $\mathcal{S}(s, a)$.
4. Add $(s, a, \mathcal{R}(s, a), \mathcal{S}(s, a))$ to the replay memory.
5. Sample a minibatch of tuples (s_i, a_i, r_i, s_{i+1}) from the replay memory, and perform stochastic gradient descent on the DQN, based on the loss function

$$\Big(Q'(s_i, a) - \Big(r_i + \gamma \max_{\alpha} Q'(s_{i+1}, \alpha)\Big)\Big)^2$$

Variable

Target

   where $Q'(s, \cdot)$ is computed by feeding s into the DQN.
6. If the next state is not terminal, go back to step 1.

Results Analysis

DQN is good at …

DQN is bad at …

Part 3

Discussion

Leveraging GPU

- The discussed neural network architecture represents a common architecture used in computer vision, relying on convolutional layers followed by fully connected layers, with rectifier activations.

- Such operations are typically implemented as matrix operations, and they may be extremely sped up using SIMD instructions.

- The DQN is easily implementable in deep learning frameworks (e.g. TensorFlow, Theano, Torch, . . .), which automatically leverage the available GPUs in the system to facilitate this.

Double Q-learning

- Original paper by van Hasselt *et al.* (2015).
- Q-learning known to have tendency to overestimate action values as the training progresses—this is often due to the fact that the same table is used for *action selection* and *evaluation*.
- *Double Q-learning* addresses this issue by using *two sets of DQN weights*—one used for evaluation, and the other for selection. They are symmetrically updated by switching their roles after each training step of the algorithm.

Three-dimensional setting

Currently evaluating the feasibility of DQN (along with several extensions) in the context of 3D games.

Ideas

- Investigating more powerful neural network *architectures* for the DQN;
- Leveraging *unsupervised methods* to extract the “most important” regions of the frame as the training progresses.
- Incorporating *sound information* (completely ignored in prior work)—requiring *recurrent neural networks* (RNNs).
- All of the above probably infeasible only a few years ago! The development of GPUs made it all easier for us.

Final Wrap-up

Q: What is the key contributing factor?

A: Almost unlimited training data

Q: How to account for long term dependency ?

A: Long short term memory may be the solution

- End -