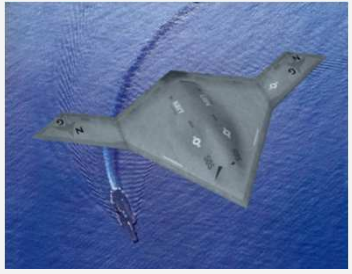
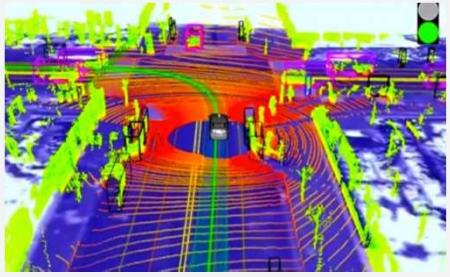
Intelligent Image and Graphics Processing

Deep Reinforcement Learning





Reinforcement Learning

强化学习的基本思想与动物心理学有关“试错法”学习的研究密切相关，即强调在与环境的交互中学习，通过环境对不同行为的评价性反馈信号来改变行为选择策略以实现学习目标。来自环境的评价性反馈信号通常称为回报或强化信号，强化学习系统的目标就是极大化期望回报信号。

Definition (Reinforcement Learning [Sutton and Barto, 1998])

“Reinforcement learning is learning what to do–how to map situations to actions–so as to maximize a numerical reward signal. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them.”

Agent and Environment

- Agent:
  - Situated in an environment
  - Subject of learning
  - Perceives (probably only a portion) of the environment's state
    → e.g. sonar, camera, ...
  - Can perform actions to act in or change the environment
    → e.g. move, turn, ...

- Environment:
  - Everything outside the agent
  - Observable state
  - Offers reward / punishment (RL)

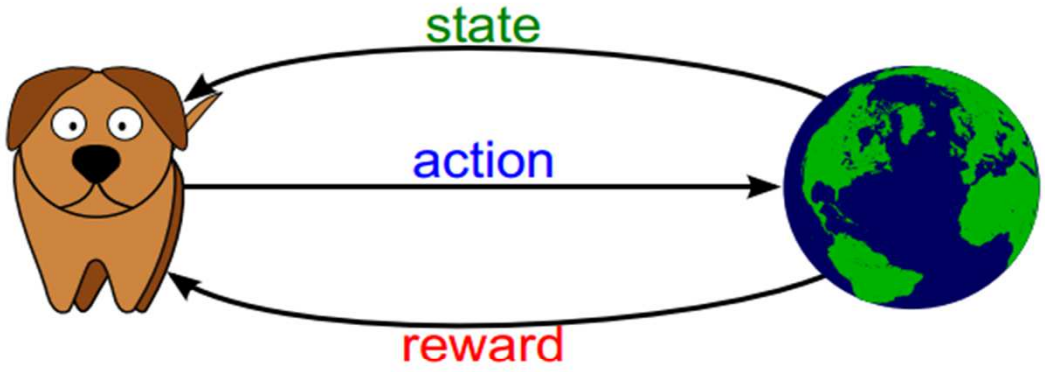
Classification of Learning Techniques

- Supervised Learning needs labeled training samples
- Unsupervised Learning has no information on the correct solution; similar structures are found
- Reinforcement Learning uses a (delayed) feedback of the environment as measure, without stating the correct solution

Example 1 (Reinforcement Learning).

Scenario: An agent hast to learn a board game
Formulation: The agent receives a *reward* if it won the game and a *punishment* (negative reward) if it loses. All other situation result in neutral feedbacks.

Reinforcement Learning process



- Interaction with environment via states and actions
- Reward as feedback for the last action
- Agent discovers usability of actions during learning
- Goal: Find policy, that maximizes discounted returns

Reinforcement Learning in Single Agent Systems

- Reward function offers numerical rewards for state-action pairs

→ Goal: Learn successful policy for any state

→ Maximizing this reward leads to proper behavior

- No labeled examples, i.e. no information on correct behavior
- Agent is not told which action to choose

→ Trial-and-error

- Learning agent has often no knowledge about its environment
- Difficulty: Current actions may influence future rewards
- Formulation as Markov Decision Process

The Markov Property

- Environment's behavior may depend on the complete history:

$$P\left[s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_0, s_0, a_0\right]$$

- If the state signal has the Markov Property, the response at $t+1$ only depends on the state and action at time t:

$$P\left[s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\right]$$

- If the system has the Markov property, both probability distributions are equal!

Markov Decision Process

Definition 1 (Markov Decision Process).

A Markov Decision Process is defined by a tuple (S, A, r, δ) with:

- *Finite set of states S*
- *Finite set of actions A*
- *Reward function r*
- *State transition function δ*
- *The state signal has the Markov property*

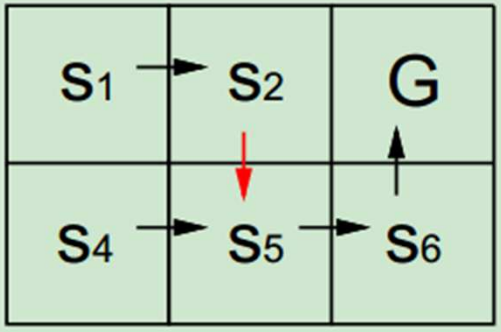
- $s_t \in S$ is perceived in time t and $a_t \in A$ is executed
- Environment responds with $r_t = r(s_t, a_t) \in \mathbb{R}$ and transitions to state $s_{t+1} = \delta(s_t, a_t)$
- δ, r are part of the environment and may be unknown

Policy

- Policy π determines agent's behavior:

$$\pi : S \rightarrow A$$

- $\pi(s_t) = a_t$ decides upon action in state s_t
- But: what is the optimal policy?



$$\pi(s_1) = a_{\text{right}}$$

$$\vdots$$

$$\pi(s_6) = a_{\text{up}}$$

Figure: Illustration of a policy

(State) Value Function

- Goal: Learn a policy that maximizes the sum of discounted rewards
- Cumulated discounted value $V^{\pi}(s_t)$ starting in s_t following arbitrary policy π:

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

- Discount factor $0 \leq \gamma < 1$: value of delayed rewards in relation to immediate rewards
  - Reward received i steps in future are discounted by γ^i
  - $\gamma \to 0$ consider only immediate rewards
  - $\gamma \to 1$ higher influence of distant rewards
- Again: Optimal policy?

Optimal Policy

- Agent has to learn policy π that maximizes $V^{\pi}(s)$ for all states s
- Optimal Policy $\pi^{\star}$:

$$\pi^{\star}: V^{\pi^{\star}}(s) \geq V^{\pi}(s) \; \forall s \in S, \forall \pi$$

- $V^{\pi^{\star}}(s)$ is the sum of discounted rewards for an optimal policy starting in s
- $V^{\star}(s)$ is short for $V^{\pi^{\star}}(s)$

$\Rightarrow$ The agent's goal is to learn an optimal policy $\pi^{\star}$

Learning an Optimal Policy

- For $V^\star$ it holds:

$$V^\star(s_1) > V^\star(s_2) \Leftrightarrow \text{ agent prefers } s_1 \text{ over } s_2$$

- Bellman optimality:

$$V^\star(s) = \max_a r(s,a) + \gamma V^\star(\delta(s,a))$$

- But: $V^\star$ values states and not actions!
- Optimal action in state s is action a, that maximizes the sum of reward $r(s,a)$ and $V^\star$-value of the successor state:

$$\pi^\star(s) = \operatorname*{argmax}_a \left(r(s,a) + \gamma V^\star(\delta(s,a))\right)$$

(State-Action) Value Function

- $Q(s, a)$ is the maximal discounted cumulated reward that can be received after executing action a in state s

$$Q(s, a) = r(s, a) + \gamma V^\star(\delta(s, a))$$

- Assumption: agent follows an optimal policy after performing action a
- Till now, $\pi^\star(s)$ requires δ and r to be known:

$$\pi^\star(s) = \underset{a}{\operatorname{argmax}} \left(r(s, a) + \gamma V^\star(\delta(s, a))\right)$$

- $\pi^\star(s)$ in terms of $Q(s, a)$:

$$\pi^\star(s) = \underset{a}{\operatorname{argmax}}\, Q(s, a)$$

- Sufficient to learn $Q(s, a)$
  $\rightarrow$ No knowledge of δ, r needed

Approaches for Reinforcement Learning

- Policy Iteration:
  - Approximates V-function and computes π thereof
  - Needs knowledge on the environment's model (δ and r as well as the probability distributions in the non-deterministic setting)
- Monte Carlo:
  - Approximates $V(s)$ by averaging rewards received after visiting state s
  - Quality of approximation increases over time
- Temporal Difference:
  - Iterative reduction of differences between estimates for any state-action pair at different points of time
  - Example: Q-Learning

Q-Learning : Off-policy TD Control

- How to learn from delayed rewards?
  $\rightarrow$ Iterative approximation
- Close relation between $V^\star(s)$ and $Q(s,a)$:

$$V^\star(s) = \max_{a'} Q(s,a')$$

- Recursive formulation of $Q(s,a)$:

$$\begin{aligned} Q(s,a) &= r(s,a) + \gamma V^\star(\delta(s,a)) \\ &= r(s,a) + \gamma \max_{a'} Q(\delta(s,a),a') \end{aligned}$$

Q-Learning Algorithm

- $\widehat{Q}$ is the agent's estimation of Q
- Agent stores estimated Q-values for each state-action pair
- The agent performs action a in state s and observes the reward r and the successor state s'
- Update of the Q-estimation after each step

$$\widehat{Q}(s,a) = r + \gamma \max_{a'} \widehat{Q}(s',a')$$

- Update only requires $\widehat{Q}$
- r and s' are known to the agent because the update is performed **after** the environment's reaction

$\Rightarrow$ No knowledge of reward function or state transition function needed

Q-Learning Algorithm

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
    Initialize s
    Repeat (for each step of episode):
        Choose a from s using policy derived from Q (e.g., ε-greedy)
        Take action a, observe r, s'
        $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$
        $s \leftarrow s'$;
    until s is terminal

Q-Learning in a Grid World

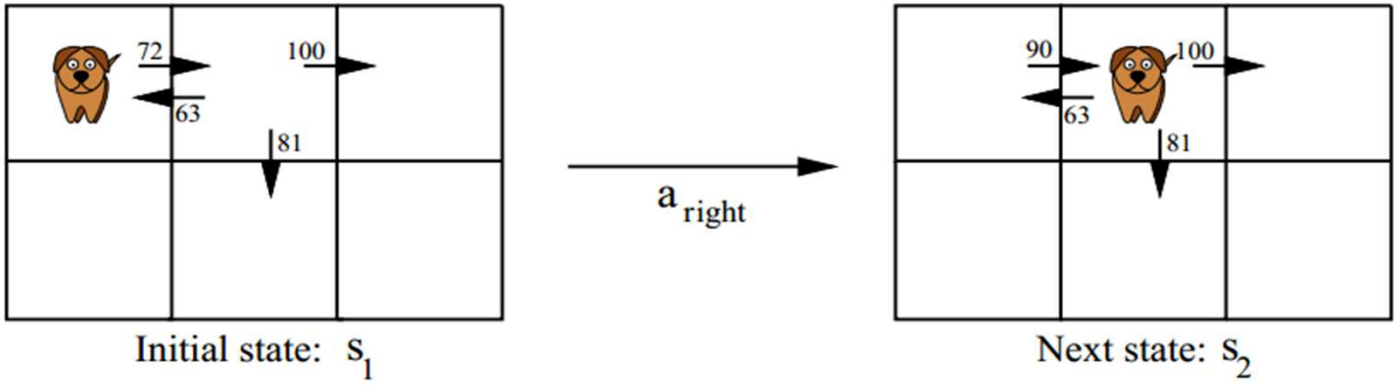


Figure: Grid World
[Mitchell, Machine Learning]

$$\begin{aligned}\widehat{Q}(s_1, a_{\text{right}}) &= r + \gamma \max_{a'} \widehat{Q}(s_2, a') \\ &= 0 + 0.9 \cdot \max\{63, 81, 100\} \\ &= 90\end{aligned}$$

Action Selection

- Agent has to perform an action $a \in A$ in each step
- Always choosing action $a = \underset{a'}{\text{argmax}}\, Q(s, a')$
  - Exploits gained knowledge
  - **But:** Prefers state-action pairs with high values in the beginning
  - Important: visit unknown state-action pairs (s, a) to gain new information (exploration)

→ Exploration/exploitation Trade-off

ε-greedy: Choose a random action with probability ε and with probability $(1 - \varepsilon)$ an action with highest Q-value

Boltzmann: Probability to select action a in state s:

$$P(a \mid s) = \frac{\exp(Q(s,a)/\tau)}{\sum_{a' \in A} \exp(Q(s,a')/\tau)}$$

Convergence

Q-Learning with a tabular representation of the knowledge converges to the real Q-values under following assumptions:

1. The system is a deterministic MDP
2. The rewards are bound:

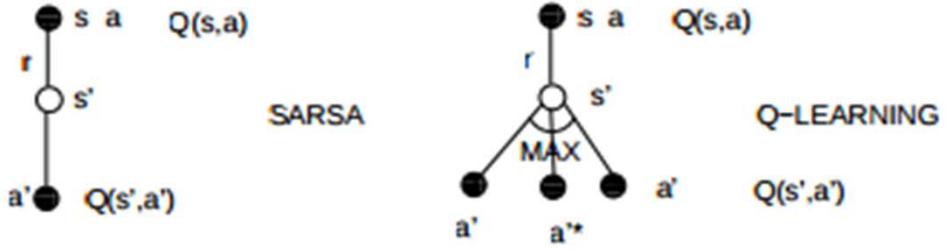
$$\forall s \forall a : \ |r(s,a)| \leq c$$

3. All state-action pairs (s, a) are visited infinitely often

SARSA : On-Policy TD Control

Q-Learning: $Q^*(s,a) = E_{s'}[r + \gamma \max_{a'} Q^*(s',a') \mid s,a]$

SARSA: $Q^*(s,a) = E_{s'}[r + \gamma Q^*(s',a') \mid s,a]$



SARSA backs up using the action a' actually chosen by the behaviour policy.

Q-LEARNING backs up using the Q-value of the action a'^* that is the *best* next action, i.e. the one with the highest Q value, $Q(s', a'^*)$. The action actually chosen by the behaviour policy *and followed* is not necessarily a'^*

SARSA Algorithm

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
    Initialize s
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Repeat (for each step of episode):
        Take action a, observe r, s'
        Choose a' from s' using policy derived from Q (e.g., ε-greedy)
        $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$
        $s \leftarrow s'$; $a \leftarrow a'$;
    until s is terminal

Deep Reinforcement Learning

- Can we apply deep learning to RL?
- Use deep network to represent value function / policy / model
- Optimise value function / policy /model end-to-end
- Using stochastic gradient descent

Bellman Equation

- Value function can be unrolled recursively

$$\begin{aligned} Q^{\pi}(s,a) &= \mathbb{E}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \mid s, a\right] \\ &= \mathbb{E}_{s'}\left[r + \gamma Q^{\pi}(s', a') \mid s, a\right] \end{aligned}$$

- Optimal value function $Q^*(s,a)$ can be unrolled recursively

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

- Value iteration algorithms solve the Bellman equation

$$Q_{i+1}(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q_i(s', a') \mid s, a\right]$$

Deep Q-Learning

- Represent value function by deep Q-network with weights w

$$Q(s, a, w) \approx Q^{\pi}(s, a)$$

- Define objective function by mean-squared error in Q-values

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

- Leading to the following Q-learning gradient

$$\frac{\partial \mathcal{L}(w)}{\partial w} = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a', w) - Q(s, a, w)\right) \frac{\partial Q(s, a, w)}{\partial w}\right]$$

- Optimise objective end-to-end by SGD, using $\frac{\partial L(w)}{\partial w}$

Stability Issues with Deep RL

Naive Q-learning oscillates or diverges with neural nets

1. Data is sequential
   - Successive samples are correlated, non-iid
2. Policy changes rapidly with slight changes to Q-values
   - Policy may oscillate
   - Distribution of data can swing from one extreme to another
3. Scale of rewards and Q-values is unknown
   - Naive Q-learning gradients can be large
     unstable when backpropagated

Deep Q-Networks

DQN provides a stable solution to deep value-based RL

1. Use experience replay
   - Break correlations in data, bring us back to iid setting
   - Learn from all past policies
2. Freeze target Q-network
   - Avoid oscillations
   - Break correlations between Q-network and target
3. Clip rewards or normalize network adaptively to sensible range
   - Robust gradients

Stable Deep RL (1): Experience Replay

To remove correlations, build data-set from agent's own experience

- Take action a_t according to ϵ-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathcal{D}$
- Sample random mini-batch of transitions (s, a, r, s') from $\mathcal{D}$
- Optimise MSE between Q-network and Q-learning targets, e.g.

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

Stable Deep RL (2): Fixed Target Q-Network

To avoid oscillations, fix parameters used in Q-learning target

- Compute Q-learning targets w.r.t. old, fixed parameters w^-

$$r + \gamma \max_{a'} Q(s', a', w^-)$$

- Optimise MSE between Q-network and Q-learning targets

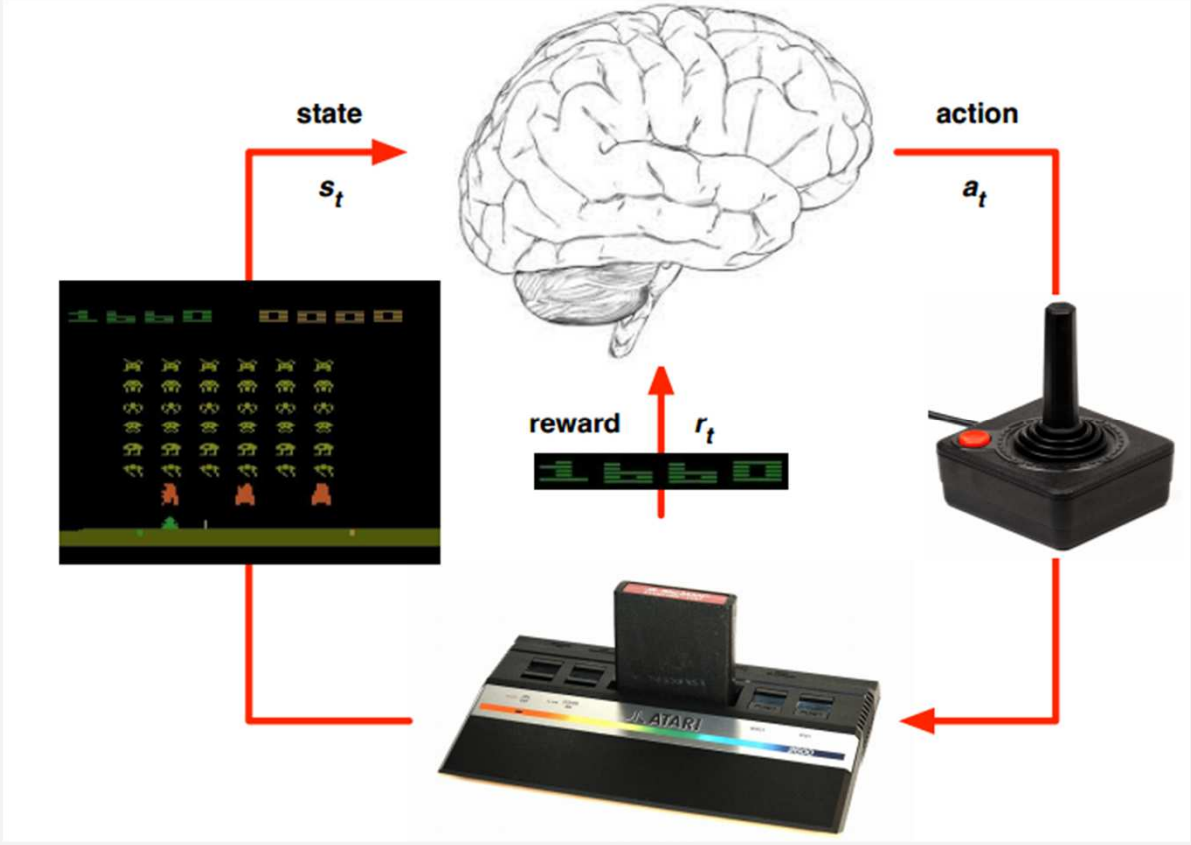
$$\mathcal{L}(w) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}} \left[\left(r + \gamma \max_{a'} Q(s', a', w^-) - Q(s, a, w) \right)^2 \right]$$

- Periodically update fixed parameters $w^- \leftarrow w$

Stable Deep RL (3): Reward/Value Range

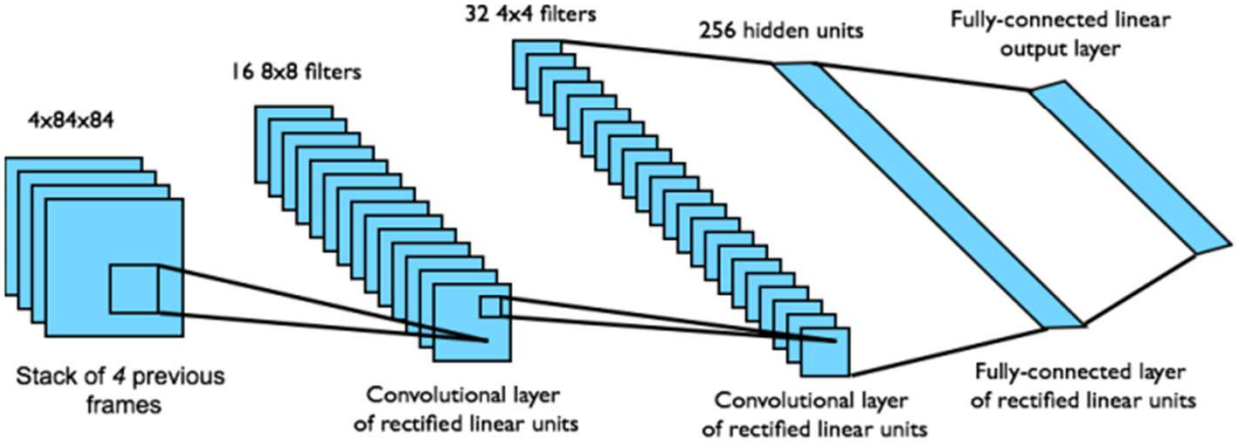
- DQN clips the rewards to $[-1, +1]$
- This prevents Q-values from becoming too large
- Ensures gradients are well-conditioned
- Can't tell difference between small and large rewards

Reinforcement Learning in Atari



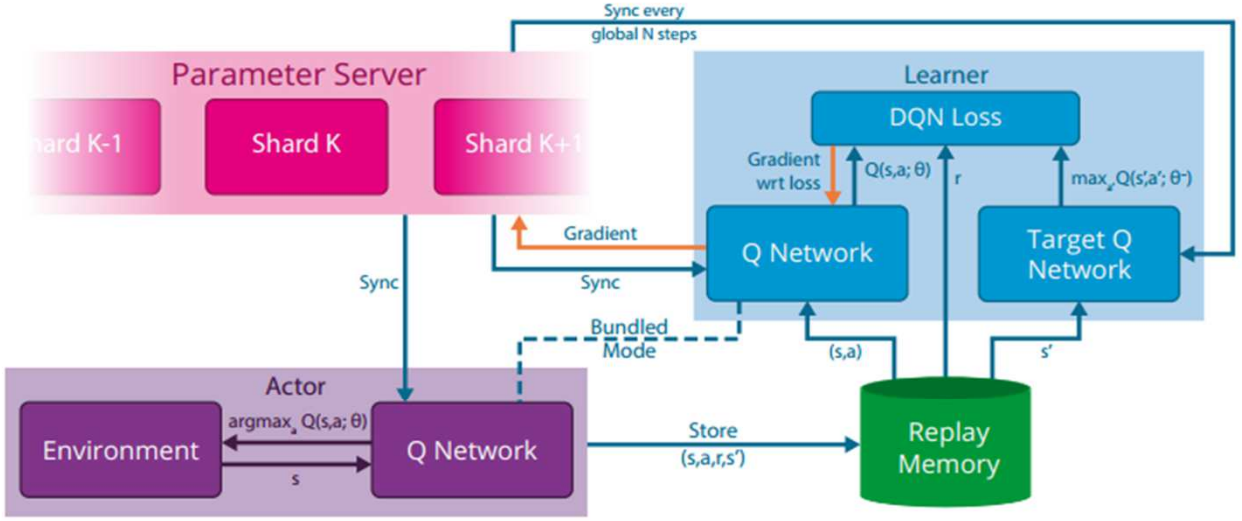
DQN in Atari

- End-to-end learning of values $Q(s, a)$ from pixels s
- Input state s is stack of raw pixels from last 4 frames
- Output is $Q(s, a)$ for 18 joystick/button positions
- Reward is change in score for that step



Network architecture and hyperparameters fixed across all games

Google Reinforcement Learning Architecture



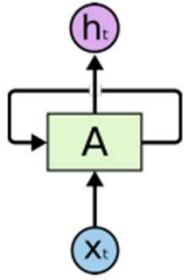
- Parallel acting: generate new interactions
- Distributed replay memory: save interactions
- Parallel learning: compute gradients from replayed interactions
- Distributed neural network: update network from gradients

Recurrent Neural Network

Humans don't start their thinking from scratch every second. As you read this essay, you understand each word based on your understanding of previous words. You don't throw everything away and start thinking from scratch again. Your thoughts have persistence.

Traditional neural networks can't do this, and it seems like a major shortcoming. For example, imagine you want to classify what kind of event is happening at every point in a movie. It's unclear how a traditional neural network could use its reasoning about previous events in the film to inform later ones.

Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist.

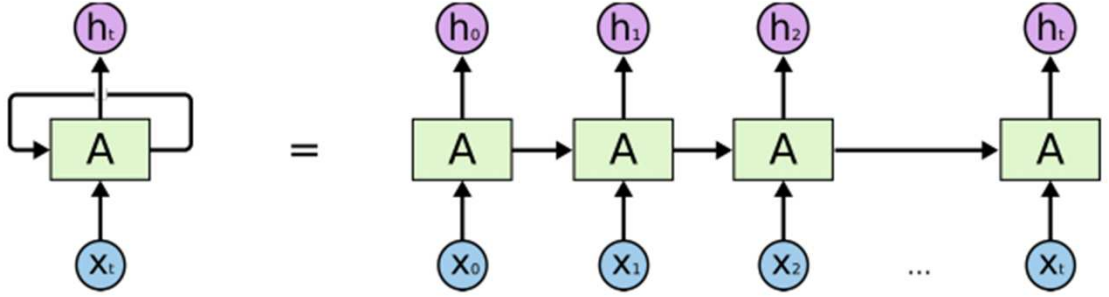


Recurrent Neural Networks have loops.

Recurrent Neural Networks

In the above diagram, a chunk of neural network, A, looks at some input x_i and outputs a value h_i. A loop allows information to be passed from one step of the network to the next.

These loops make recurrent neural networks seem kind of mysterious. However, if you think a bit more, it turns out that they aren't all that different than a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:

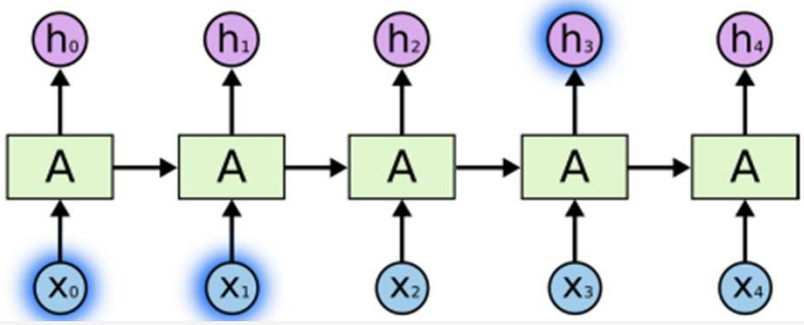


An unrolled recurrent neural network.

The Problem of Long-Term Dependencies

One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame.

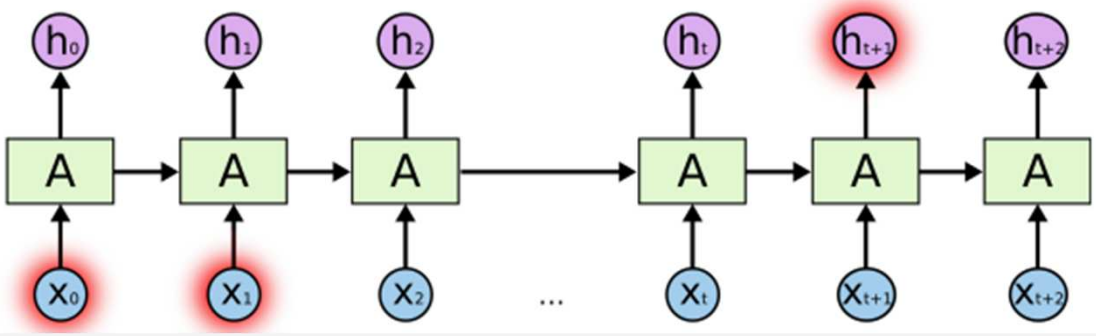
Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in "the clouds are in the *sky*," we don't need any further context – it's pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.



The Problem of Long-Term Dependencies

But there are also cases where we need more context. Consider trying to predict the last word in the text "I grew up in France... I speak fluent *French*." Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large.

Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.

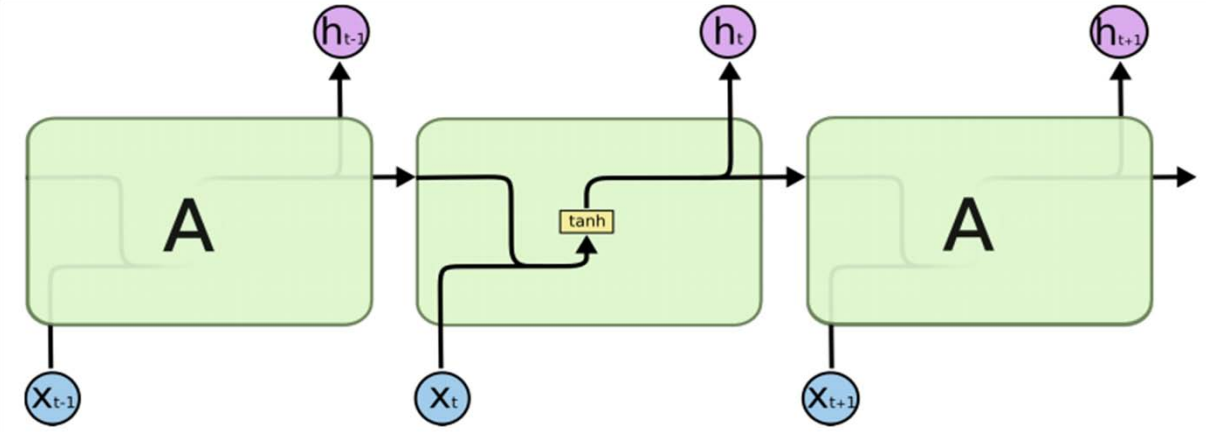


LSTM Networks (Long Short Term Memory)

Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies.

LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

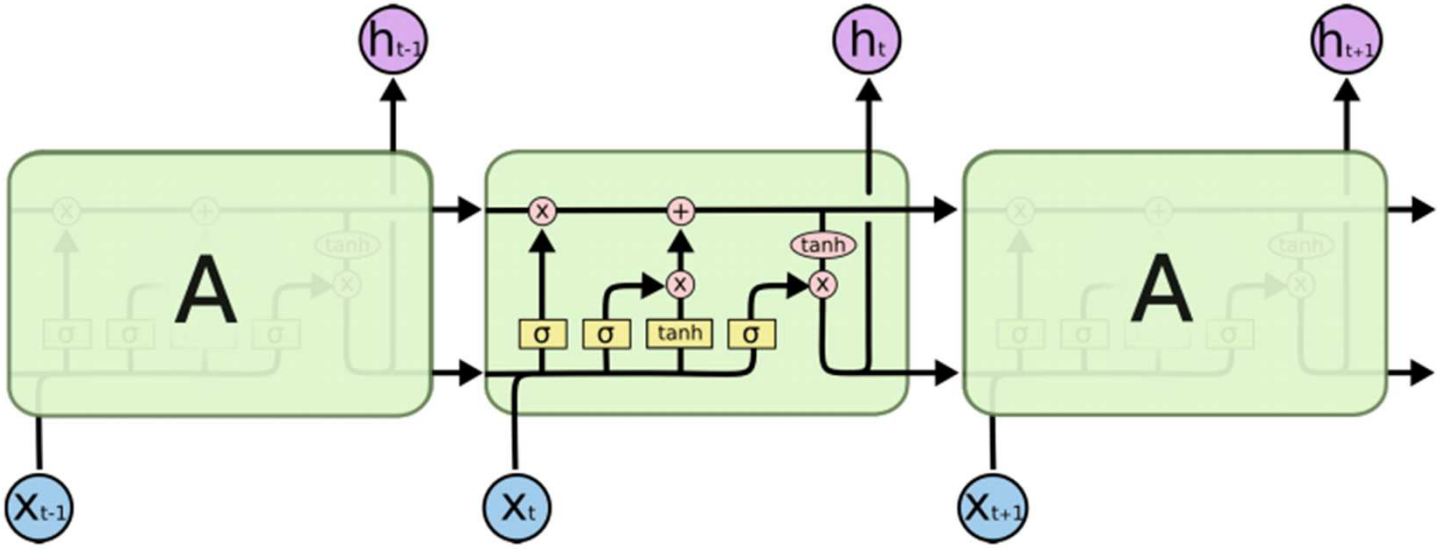
All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.



The repeating module in a standard RNN contains a single layer.

LSTM Networks (Long Short Term Memory)

LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.

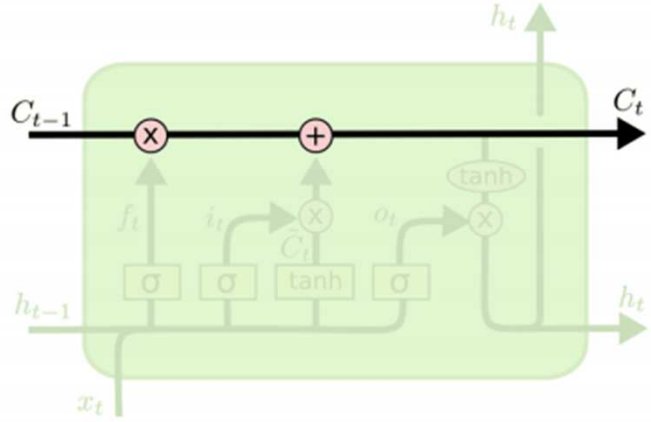


The repeating module in an LSTM contains four interacting layers.

The Core Idea Behind LSTMs

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.

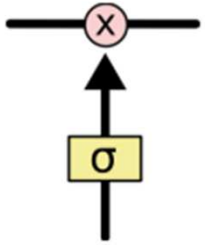
The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.



The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

The Core Idea Behind LSTMs

Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.



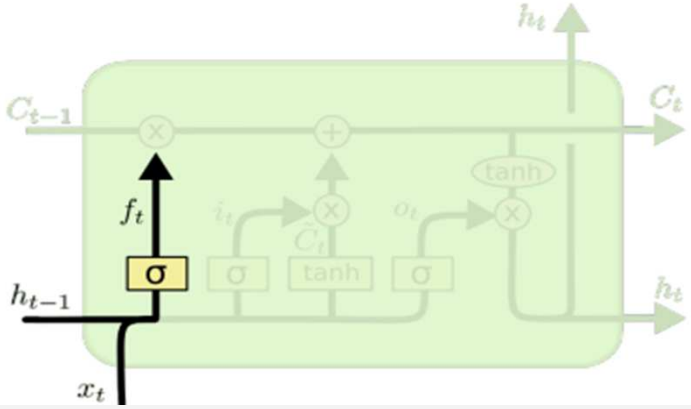
The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through," while a value of one means "let everything through!"

An LSTM has three of these gates, to protect and control the cell state.

Step-by-Step LSTM Walk Through

The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer." It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. A 1 represents "completely keep this" while a 0 represents "completely get rid of this."

Let's go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.

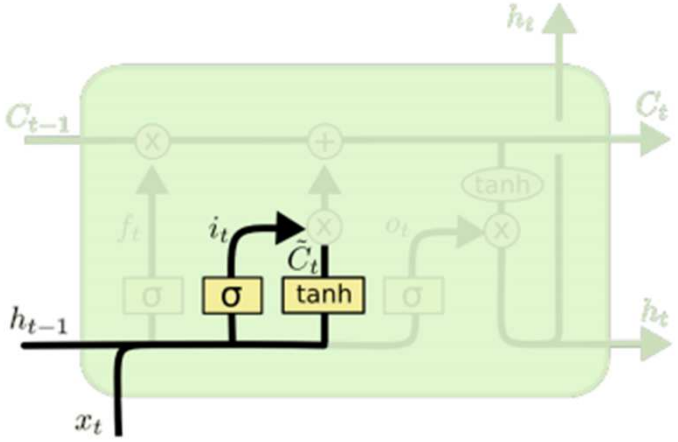


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Step-by-Step LSTM Walk Through

The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, $\tilde{C}_t$, that could be added to the state. In the next step, we'll combine these two to create an update to the state.

In the example of our language model, we'd want to add the gender of the new subject to the cell state, to replace the old one we're forgetting.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

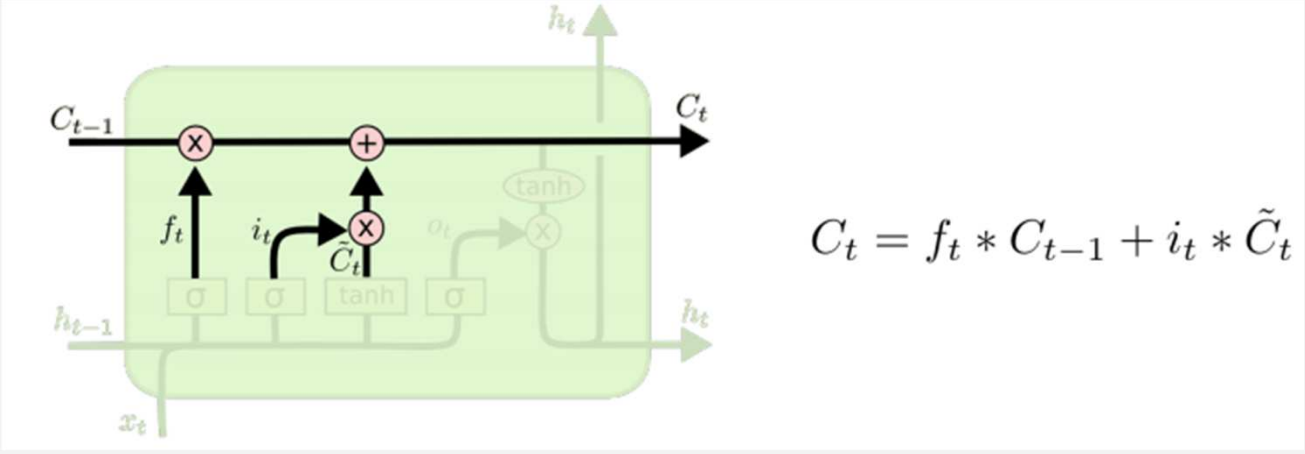
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Step-by-Step LSTM Walk Through

It's now time to update the old cell state, C_{t-1}, into the new cell state C_t. The previous steps already decided what to do, we just need to actually do it.

We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $i_t * \tilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value.

In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in the previous steps.

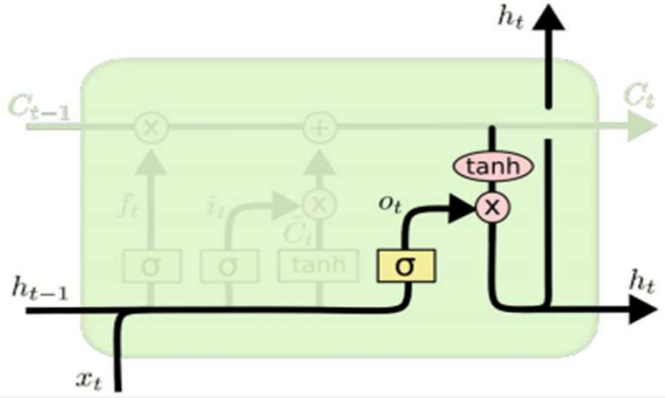


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Step-by-Step LSTM Walk Through

Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through **tanh** (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

Step-by-Step LSTM Walk Through