Outline

Reinforcement Learning: AI = RL

- RL is a general-purpose framework for artificial intelligence
- We seek a single agent which can solve any human-level task
- The essence of an intelligent agent
- Powerful RL requires powerful representations

Outline

Deep Representations

- A deep representation is a composition of many functions

$$x \xrightarrow[w_1]{} h_1 \xrightarrow[w_2]{} \dots \xrightarrow[w_n]{} h_n \xrightarrow[w_{n+1}]{} y$$

- Its gradient can be backpropagated by the chain rule

$$\begin{array}{ccccccc}
\frac{\partial h_1}{\partial x} & \leftarrow & \frac{\partial h_2}{\partial h_1} & \leftarrow \dots \leftarrow & \frac{\partial y}{\partial h_n} & \leftarrow & \frac{\partial}{\partial y} \\
 & & \downarrow & & \downarrow & & \downarrow \\
 & & \frac{\partial h_1}{\partial w_1} & \dots & \frac{\partial h_n}{\partial w_n} & & \frac{\partial y}{\partial w_{n+1}}
\end{array}$$

Deep Neural Network

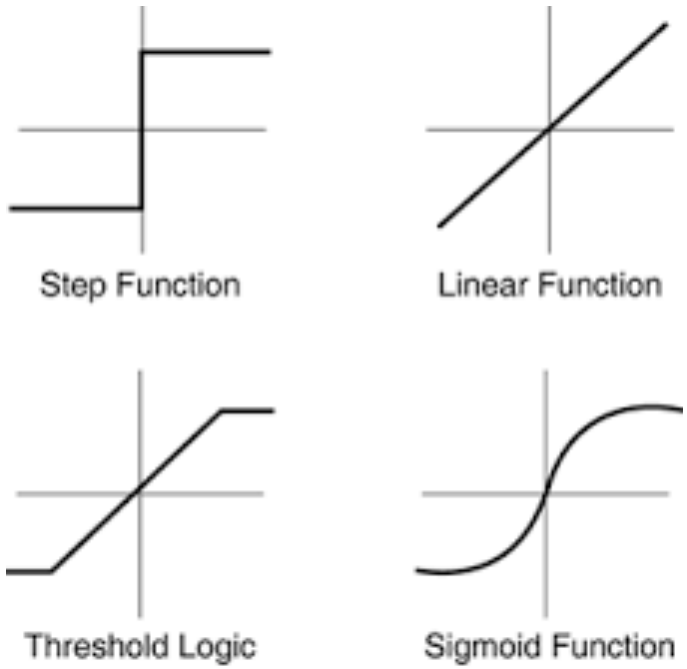
A deep neural network is typically composed of:

- Linear transformations

$$h_{k+1} = Wh_k$$

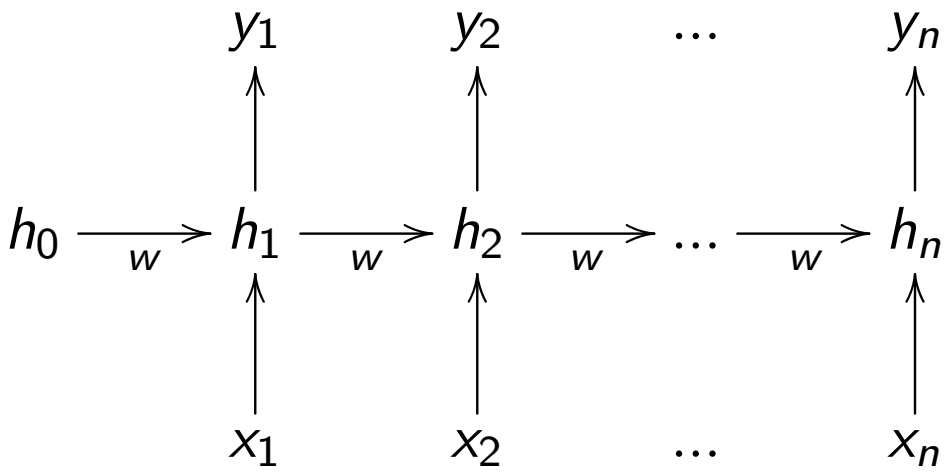
- Non-linear activation functions

$$h_{k+2} = f(h_{k+1})$$

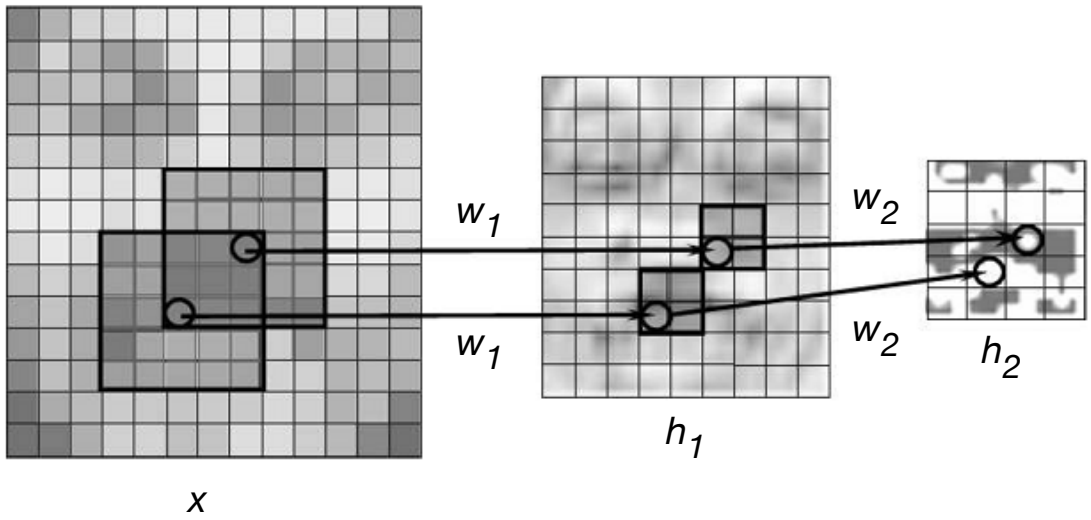


Weight Sharing

Recurrent neural network shares weights between time-steps



Convolutional neural network shares weights between local regions



Loss Function

- A loss function $l(y)$ measures goodness of output y, e.g.
  - Mean-squared error $l(y) = ||y^* - y||^2$
  - Log likelihood $l(y) = \log \mathbb{P}[y^*|x]$
- The loss is appended to the forward computation

$$x \xrightarrow[w_1]{} h_1 \xrightarrow[w_2]{} \dots \xrightarrow[w_n]{} h_n \xrightarrow[w_{n+1}]{} y \longrightarrow l(y)$$

- Gradient of loss is appended to the backward computation

$$\begin{array}{ccccccccc}
\frac{\partial h_1}{\partial x} & \longleftarrow & \frac{\partial h_2}{\partial h_1} & \longleftarrow & \dots & \longleftarrow & \frac{\partial y}{\partial h_n} & \longleftarrow & \frac{\partial l(y)}{\partial y} \\
 & & \downarrow & & & & \downarrow & & \downarrow \\
 & & \frac{\partial h_1}{\partial w_1} & & \dots & & \frac{\partial h_n}{\partial w_n} & & \frac{\partial y}{\partial w_{n+1}}
\end{array}$$

Stochastic Gradient Descent

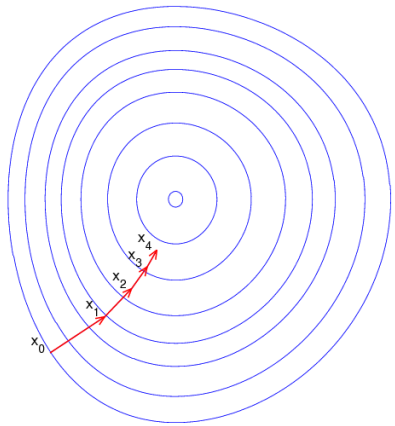
- Minimise expected loss $\mathcal{L}(w) = \mathbb{E}_x [l(y)]$
- Follow the gradient of $\mathcal{L}(w)$

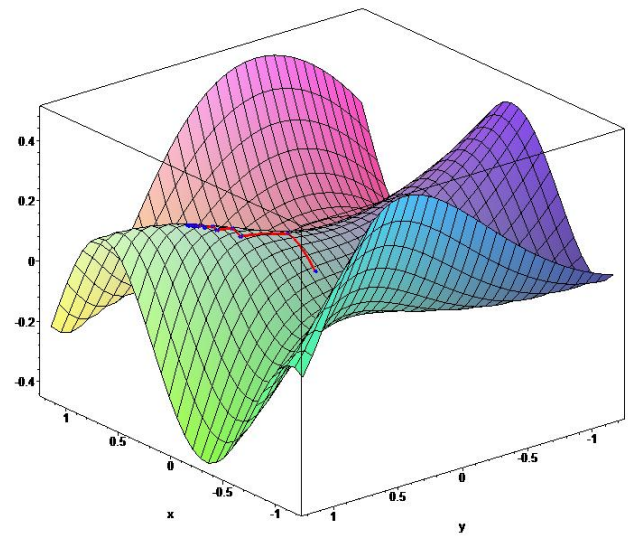
$$\frac{\partial \mathcal{L}(w)}{\partial w} = \mathbb{E}_x \left[\frac{\partial l(y)}{\partial w} \right] = \mathbb{E}_x \begin{pmatrix} \frac{\partial l(y)}{\partial w^{(1)}} \\ \vdots \\ \frac{\partial l(y)}{\partial w^{(k)}} \end{pmatrix}$$

- Adjust w in direction of -ve gradient

$$\Delta w = -\frac{\alpha}{2} \alpha \frac{\partial l(y)}{\partial w}$$

where α is a step-size parameter





Deep Supervised Learning

- Deep neural networks have achieved remarkable success
- Simple ingredients solve supervised learning problems
  - Use deep network as a function approximator
  - Define loss function
  - Optimise parameters end-to-end by SGD
- Scales well with memory/data/computation
- Solves the representation learning problem
- State-of-the-art for images, audio, language, ...

Deep Supervised Learning

- Deep neural networks have achieved remarkable success
- Simple ingredients solve supervised learning problems
  - Use deep network as a function approximator
  - Define loss function
  - Optimise parameters end-to-end by SGD
- Scales well with memory/data/computation
- Solves the representation learning problem
- State-of-the-art for images, audio, language, ...
- Can we follow the same recipe for RL?

Outline

Policies and Value Functions

- Policy π is a behaviour function selecting actions given states

$$a = \pi(s)$$

- Value function $Q^\pi(s, a)$ is expected total reward from state s and action a under policy π

$$Q^\pi(s, a) = \mathbb{E}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \mid s, a\right]$$

"How good is action a in state s?"

Approaches To Reinforcement Learning

Policy-based RL

- Search directly for the optimal policy π^*
- This is the policy achieving maximum future reward

Value-based RL

- Estimate the optimal value function $Q^*(s, a)$
- This is the maximum value achievable under any policy

Approaches To Reinforcement Learning

Policy-based RL

- Search directly for the optimal policy π^*
- This is the policy achieving maximum future reward

Value-based RL

- Estimate the optimal value function $Q^*(s, a)$
- This is the maximum value achievable under any policy

Model-based RL

- Build a transition model of the environment
- Plan (e.g. by lookahead) using model

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

Outline

Bellman Equation

- Bellman expectation equation unrolls value function Q^{π}

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

Bellman Equation

- Bellman expectation equation unrolls value function Q^{π}

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

- Bellman optimality equation unrolls optimal value function Q^{*}

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

Bellman Equation

- Bellman expectation equation unrolls value function Q^{π}

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

- Bellman optimality equation unrolls optimal value function Q^{*}

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

- Policy iteration algorithms solve Bellman expectation equation

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

- Value iteration algorithms solve Bellman optimality equation

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

Policy Iteration with Non-Linear Sarsa

- Represent value function by Q-network with weights w

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

Policy Iteration with Non-Linear Sarsa

- Represent value function by Q-network with weights w

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

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

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

Policy Iteration with Non-Linear Sarsa

- Represent value function by Q-network with weights w

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

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

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

- Leading to the following Sarsa gradient

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

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

Value Iteration with Non-Linear 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

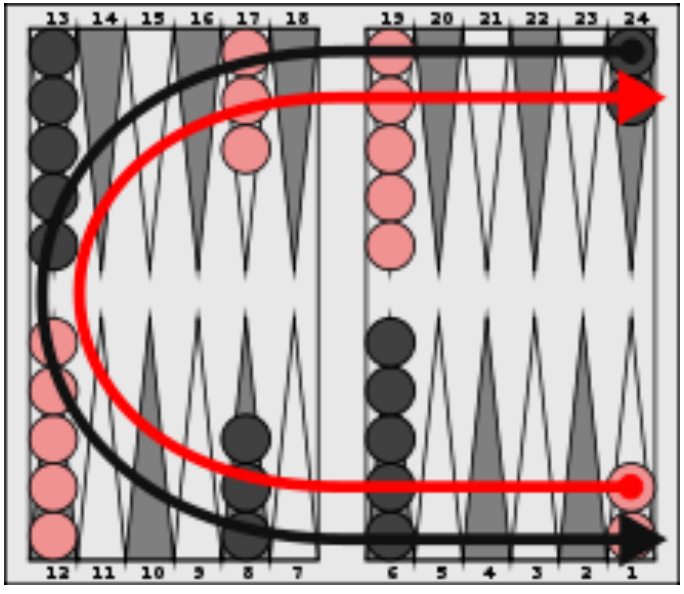
$$\mathcal{L}(w) = \mathbb{E}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s', a', w)}_{\text{target}} - Q(s, a, w)\right)^2\right]$$

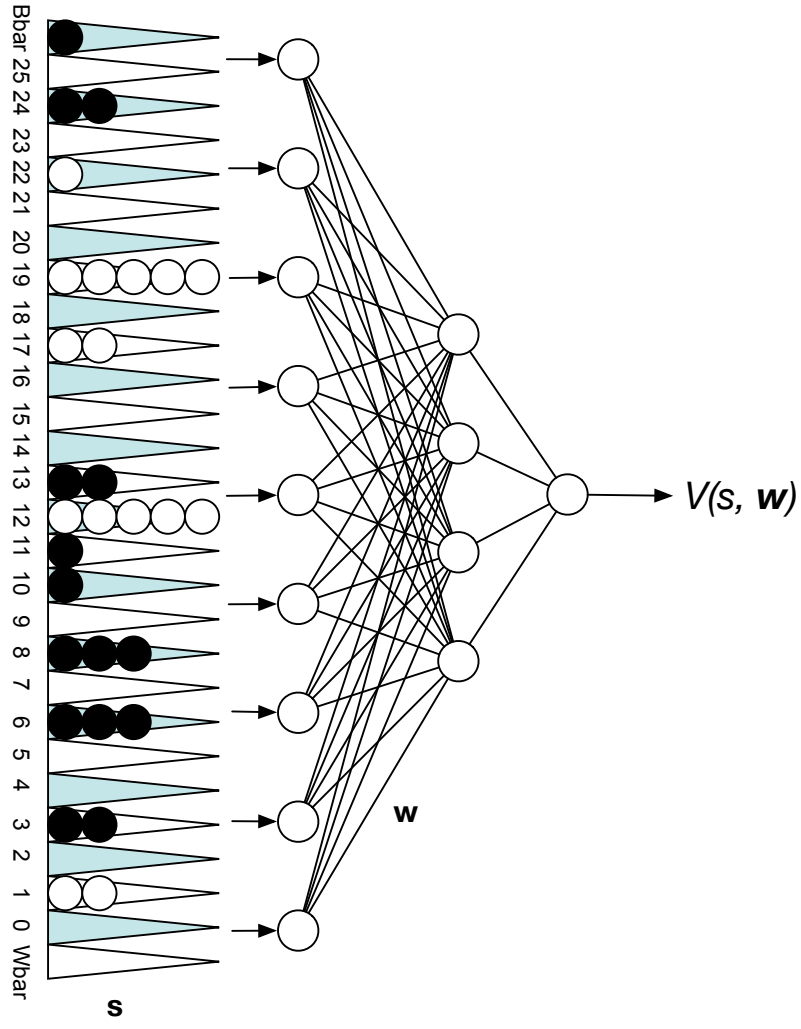
- Leading to the following Q-learning gradient

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

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

Example: TD Gammon





Self-Play Non-Linear Sarsa

- Initialised with random weights
- Trained by games of self-play
- Using non-linear Sarsa with afterstate value function

$$Q(s, a, w) = \mathbb{E}\left[V(s', w)\right]$$

- Greedy policy improvement (no exploration)
- Algorithm converged in practice (not true for other games)

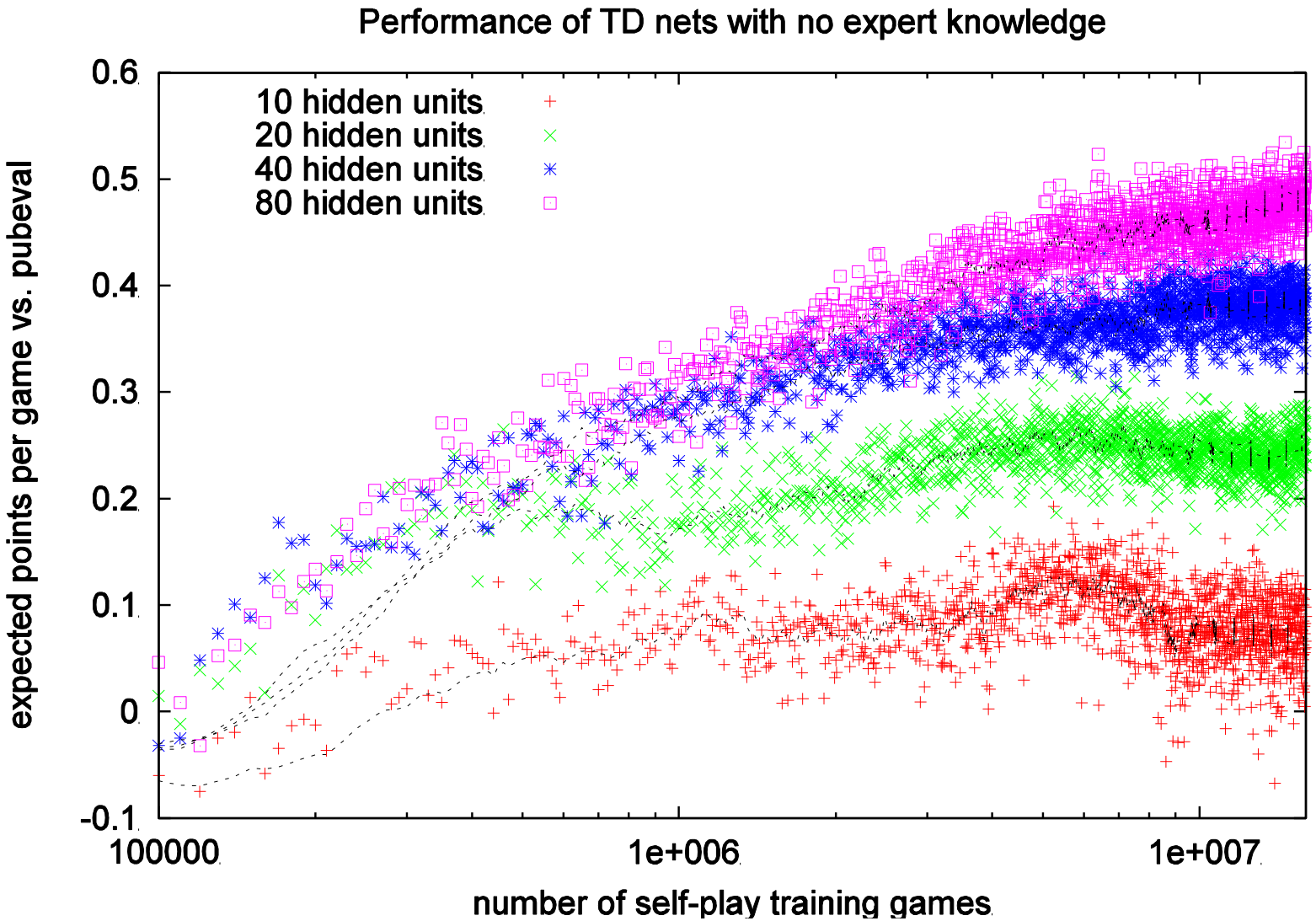
Self-Play Non-Linear Sarsa

- Initialised with random weights
- Trained by games of self-play
- Using non-linear Sarsa with afterstate value function

$$Q(s, a, w) = \mathbb{E}\left[V(s', w)\right]$$

- Greedy policy improvement (no exploration)
- Algorithm converged in practice (not true for other games)
- TD Gammon defeated world champion Luigi Villa 7-1 (Tesauro, 1992)

New TD-Gammon Results



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
   - Using off-policy Q-learning
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^-

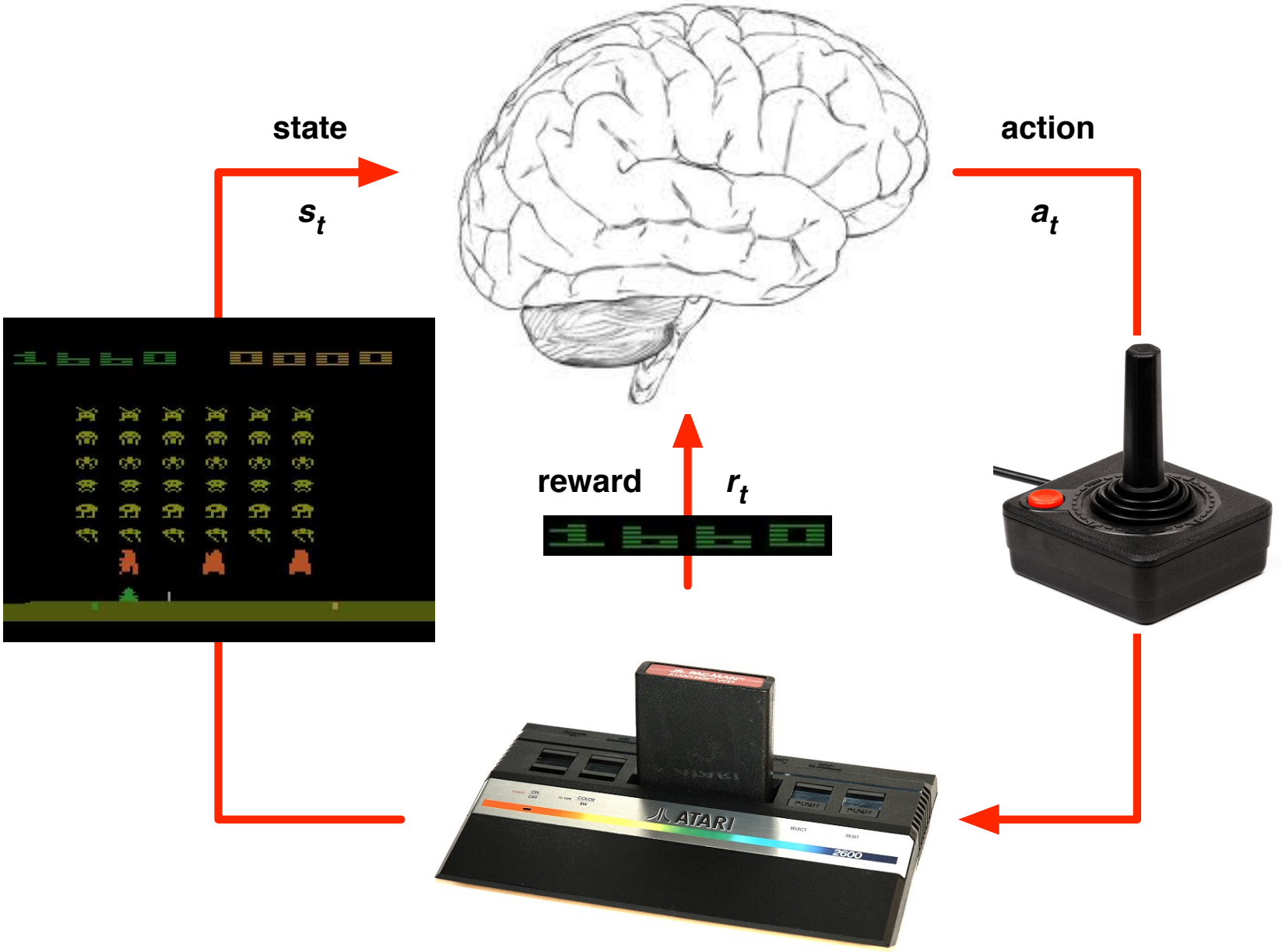
$$r + \gamma \max_{a'} Q(s', a', w^-)$$

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

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

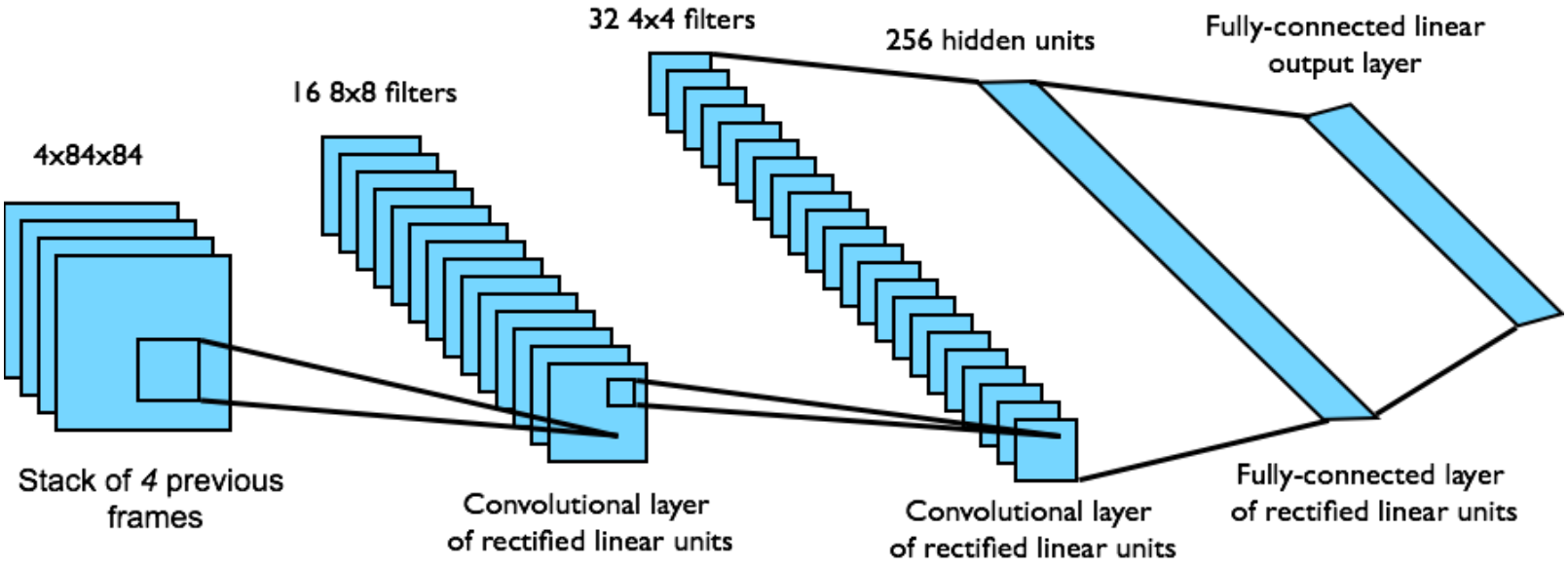
- Periodically update fixed parameters $w^- \leftarrow w$

Reinforcement Learning in Atari



DQN in Atari

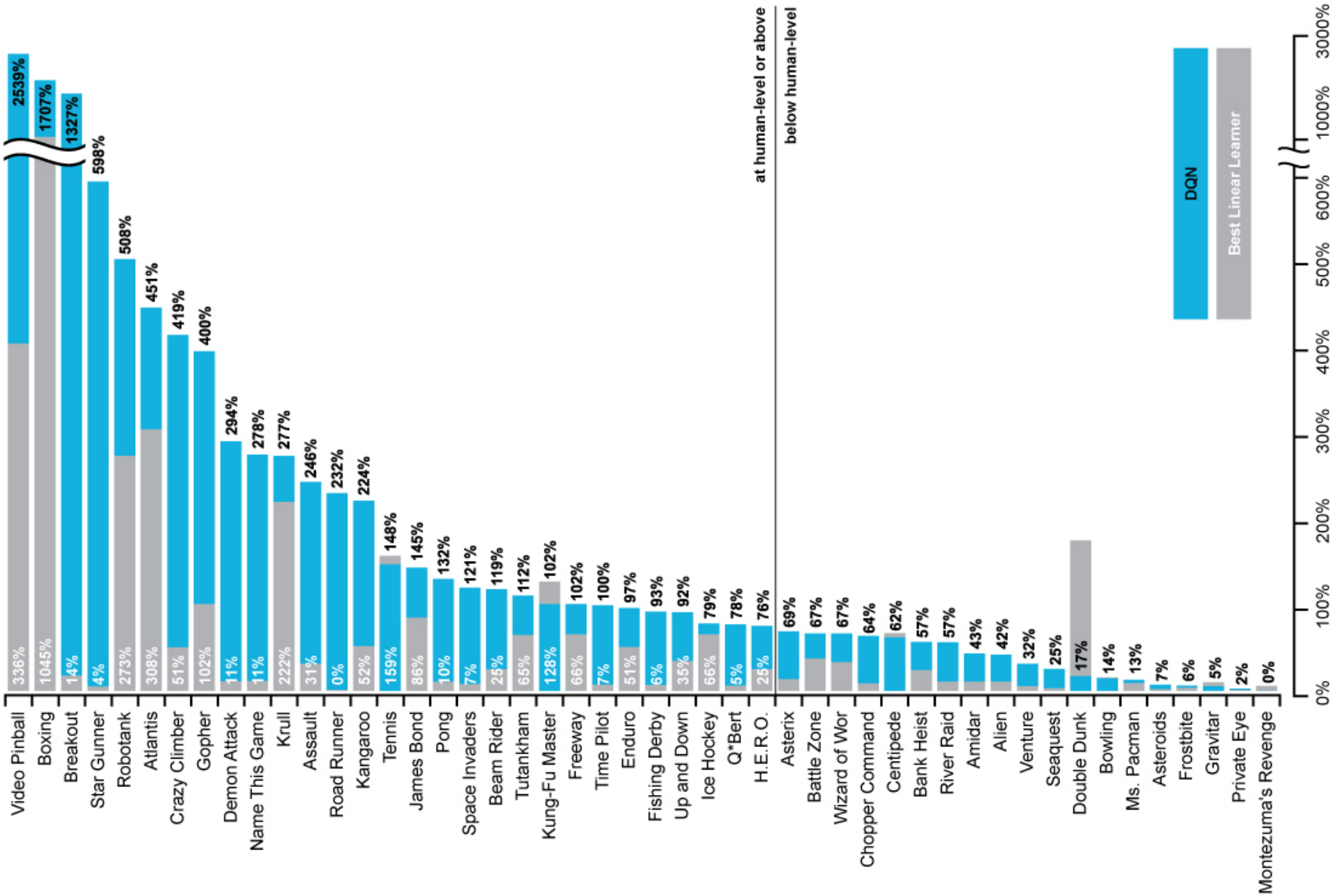
- End-to-end learning of values $Q(s, a)$ from pixels s
- Input state s is stack of raw pixels from last 4 frames
- Output is $Q(s, a)$ for 18 joystick/button positions
- Reward is change in score for that step



Network architecture and hyperparameters fixed across all games

[Mnih et al.]

DQN Results in Atari



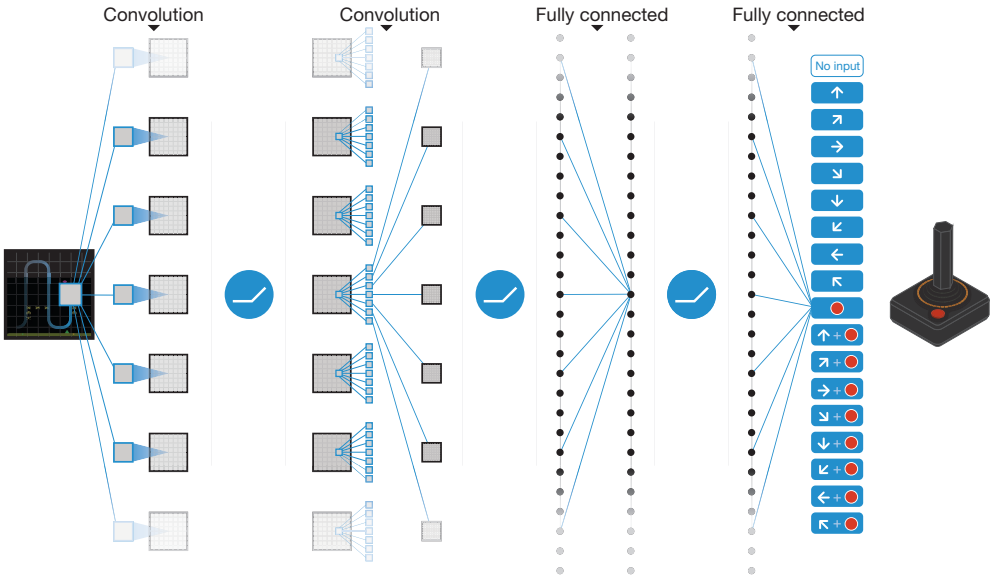


Figure 1 | Schematic illustration of the convolutional neural network. The details of the architecture are explained in the Methods. The input to the neural network consists of an $84 \times 84 \times 4$ image produced by the preprocessing map ϕ, followed by three convolutional layers (note: snaking 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, $\max(0,x)$).

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 (210×160 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[12,15]. 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;

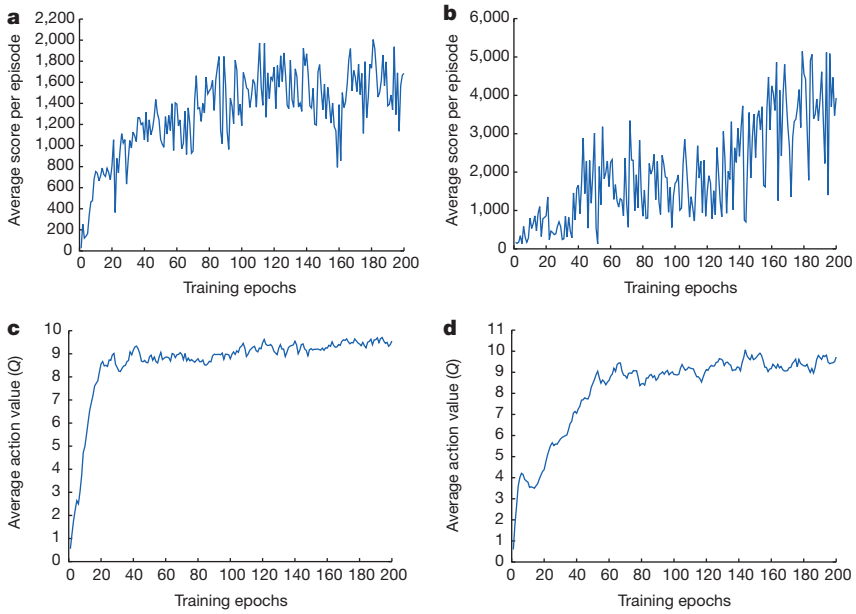


Figure 2 | Training curves tracking the agent's average score and average predicted action-value. a, Each point is the average score achieved per episode after the agent is run with ε-greedy policy ($\varepsilon = 0.05$) for 520 k frames on Space Invaders. **b**, Average score achieved per episode for 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). **d**, Average predicted action-value on Seaquest. See Supplementary Discussion for details.

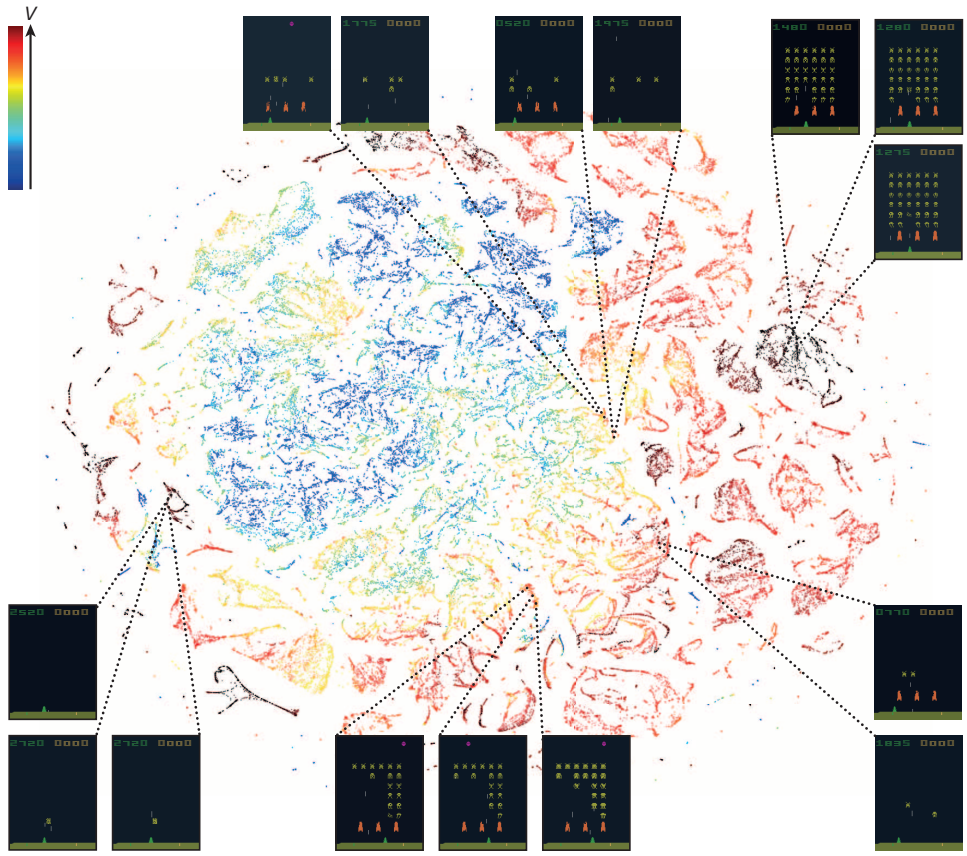


Figure 4 | Two-dimensional t-SNE embedding of the representations in the last hidden layer assigned by DQN to game states experienced while playing Space Invaders. The plot was generated by letting the DQN agent play for 2 h of real game time and running the t-SNE algorithm[25] on the last hidden layer representations assigned by DQN to each experienced game state. The points are coloured according to the state values (*V*, maximum expected reward of a state) predicted by DQN for the corresponding game states (ranging from dark red (highest *V*) to dark blue (lowest *V*)). The screenshots corresponding to a selected number of points are shown. The DQN agent predicts high state values for both full (top right screenshots) and nearly complete screens (bottom left screenshots) because it has learned that completing a screen leads to a new screen full of enemy ships. Partially completed screens (bottom screenshots) are assigned lower state values because less immediate reward is available. The screens shown on the bottom right and top left and middle are less perceptually similar than the other examples but are still mapped to nearby representations and similar values because the orange bunkers do not carry great significance near the end of a level. With permission from Square Enix Limited.

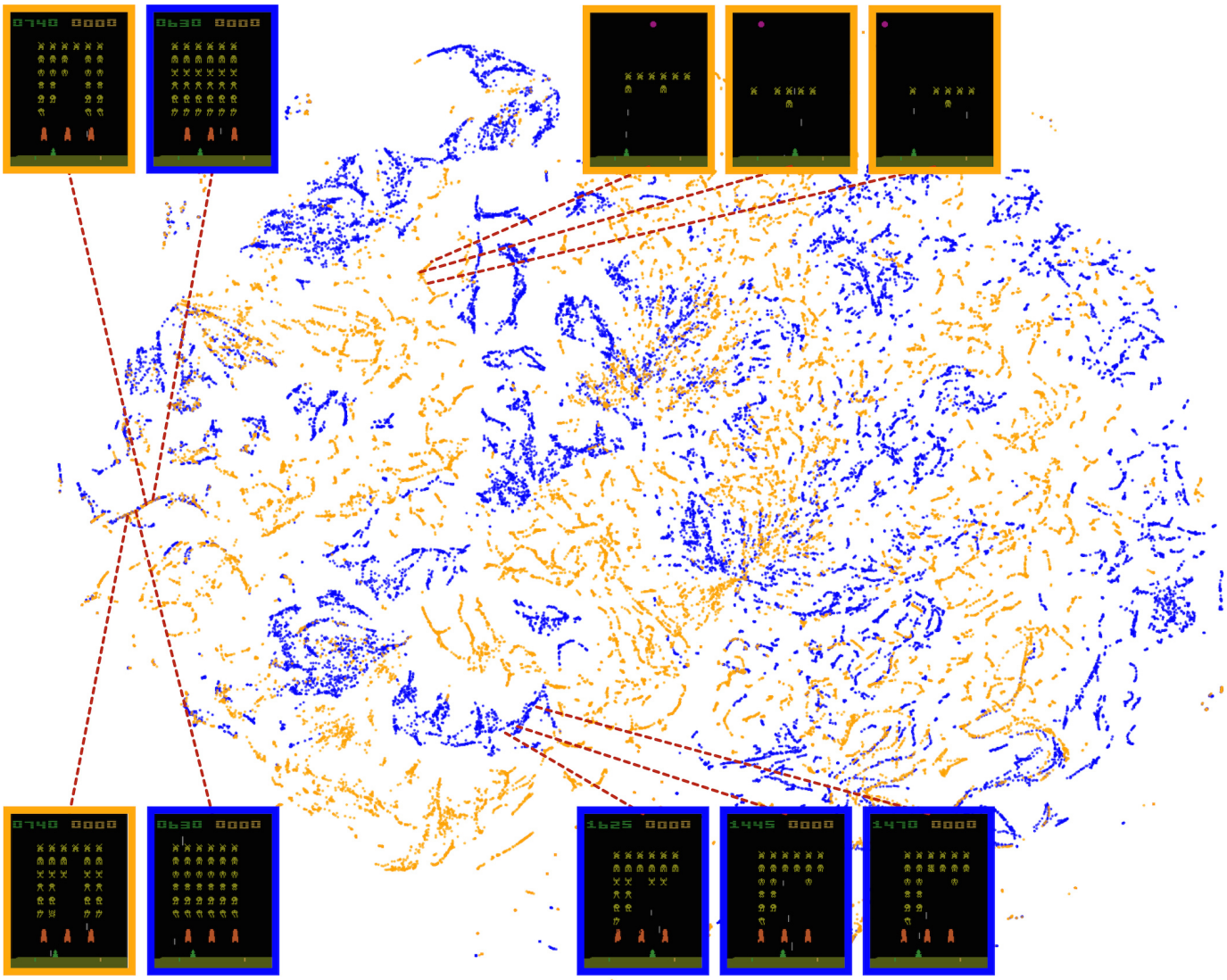
Indeed, in certain games DQN is able to discover a relatively long-term strategy (for example, Breakout: the agent learns the optimal strategy, which is to first dig a tunnel around the side of the wall allowing the ball to be sent around the back to destroy a large number of blocks; see Supplementary Video 2 for illustration of development of DQN's performance over the course of training). Nevertheless, games demanding more temporally extended planning strategies still constitute a major challenge for all existing agents including DQN (for example, Montezuma's Revenge).

In this work, we demonstrate that a single architecture can successfully learn control policies in a range of different environments with only very minimal prior knowledge, receiving only the pixels and the game score as inputs, and using the same algorithm, network architecture and hyperparameters on each game, privy only to the inputs a human player would have. In contrast to previous work[24,26], our approach incorporates 'end-to-end' reinforcement learning that uses reward to continuously shape representations within the convolutional network towards salient features of the environment that facilitate value estimation. This principle draws on neurobiological evidence that reward signals during perceptual learning may influence the characteristics of representations within primate visual cortex[27,28]. Notably, the successful integration of reinforcement learning with deep network architectures was critically dependent on our incorporation of a replay algorithm[21–23] involving the storage and representation of recently experienced transitions. Convergent evidence suggests that the hippocampus may support the physical realization of such a process in the mammalian brain, with the time-compressed reactivation of recently experienced trajectories during offline periods[21,22] (for example, waking rest) providing a putative mechanism by which value functions may be efficiently updated through interactions with the basal ganglia[22]. In the future, it will be important to explore the potential use of biasing the content of experience replay towards salient events, a phenomenon that characterizes empirically observed hippocampal replay[29], and relates to the notion of 'prioritized sweeping'[30] in reinforcement learning. Taken together, our work illustrates the power of harnessing state-of-the-art machine learning techniques with biologically inspired mechanisms to create agents that are capable of learning to master a diverse array of challenging tasks.

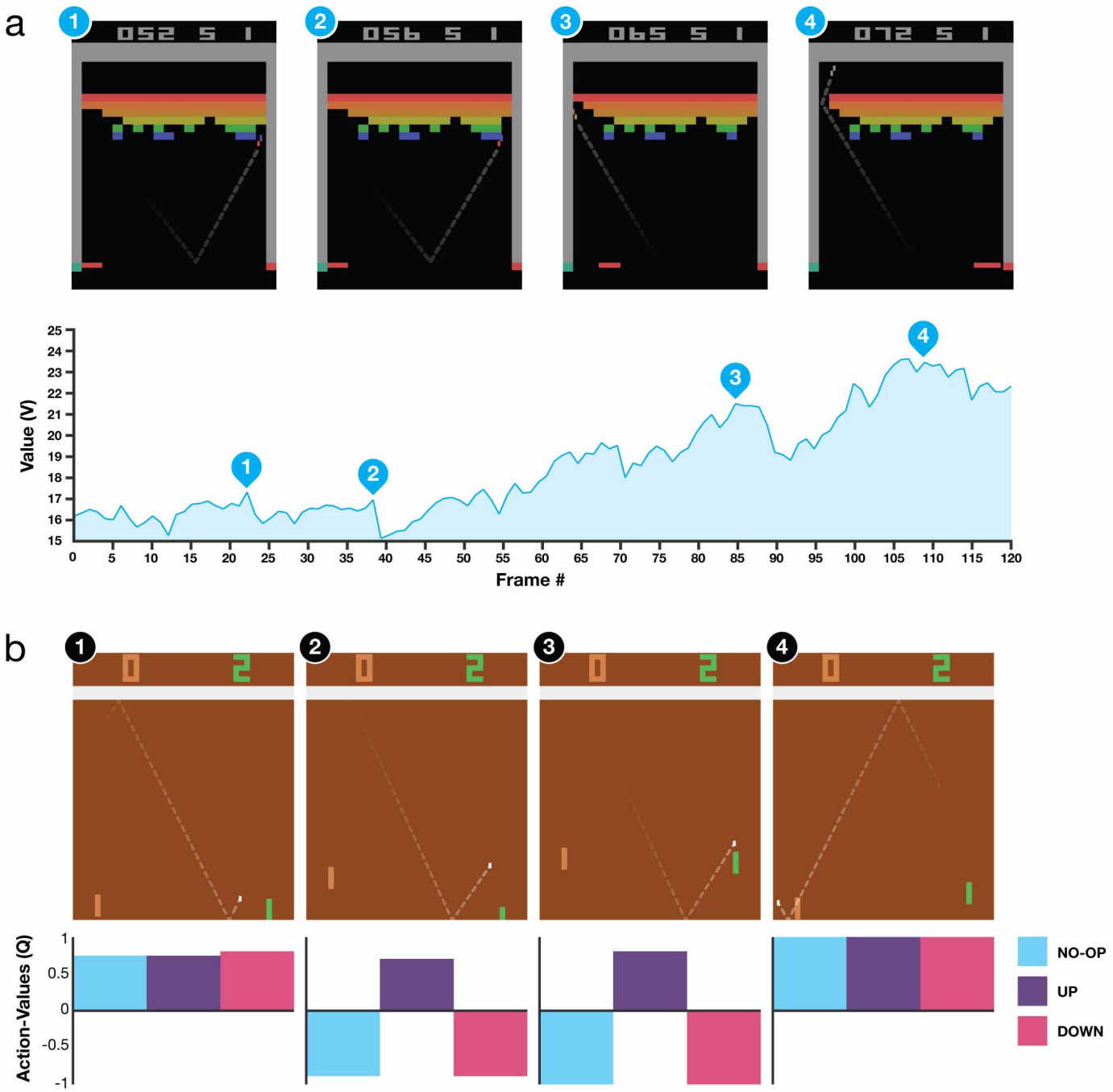
Online Content Methods, along with any additional Extended Data display items and Source Data, are available in the online version of the paper; references unique to these sections appear only in the online paper.

Received 10 July 2014; accepted 16 January 2015.

1. Sutton, R. & Barto, A. *Reinforcement Learning: An Introduction* (MIT Press, 1998).
2. Thorndike, E. L. *Animal Intelligence: Experimental studies* (Macmillan, 1911).
3. Schultz, W., Dayan, P. & Montague, P. R. A neural substrate of prediction and reward. *Science* **275,** 1593–1599 (1997).
4. Serre, T., Wolf, L. & Poggio, T. Object recognition with features inspired by visual cortex. *Proc. IEEE. Comput. Soc. Conf. Comput. Vis. Pattern. Recognit.* 994–1000 (2005).
5. Fukushima, K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biol. Cybern.* **36,** 193–202 (1980).



Extended Data Figure 1 | Two-dimensional t-SNE embedding of the representations in the last hidden layer assigned by DQN to game states experienced during a combination of human and agent play in Space Invaders. The plot was generated by running the t-SNE algorithm[25] on the last hidden layer representation assigned by DQN to game states experienced during a combination of human (30 min) and agent (2 h) play. The fact that there is similar structure in the two-dimensional embeddings corresponding to the DQN representation of states experienced during human play (orange points) and DQN play (blue points) suggests that the representations learned by DQN do indeed generalize to data generated from policies other than its own. The presence in the t-SNE embedding of overlapping clusters of points corresponding to the network representation of states experienced during human and agent play shows that the DQN agent also follows sequences of states similar to those found in human play. Screenshots corresponding to selected states are shown (human: orange border; DQN: blue border).



Extended Data Figure 2 | Visualization of learned value functions on two games, Breakout and Pong. a, A visualization of the learned value function on the game Breakout. At time points 1 and 2, the state value is predicted to be ~17 and the agent is clearing the bricks at the lowest level. Each of the peaks in the value function curve corresponds to a reward obtained by clearing a brick. At time point 3, the agent is about to break through to the top level of bricks and the value increases to ~21 in anticipation of breaking out and clearing a large set of bricks. At point 4, the value is above 23 and the agent has broken through. After this point, the ball will bounce at the upper part of the bricks clearing many of them by itself. **b**, A visualization of the learned action-value function on the game Pong. At time point 1, the ball is moving towards the paddle controlled by the agent on the right side of the screen and the values of all actions are around 0.7, reflecting the expected value of this state based on previous experience. At time point 2, the agent starts moving the paddle towards the ball and the value of the 'up' action stays high while the value of the 'down' action falls to −0.9. This reflects the fact that pressing 'down' would lead to the agent losing the ball and incurring a reward of −1. At time point 3, the agent hits the ball by pressing 'up' and the expected reward keeps increasing until time point 4, when the ball reaches the left edge of the screen and the value of all actions reflects that the agent is about to receive a reward of 1. Note, the dashed line shows the past trajectory of the ball purely for illustrative purposes (that is, not shown during the game). With permission from Atari Interactive, Inc.

How much does DQN help?

DQN

	Q-learning	Q-learning + Target Q	Q-learning + Replay	Q-learning + Replay + Target Q
Breakout	3	10	241	**317**
Enduro	29	142	831	**1006**
River Raid	1453	2868	4103	**7447**
Seaquest	276	1003	823	**2894**
Space Invaders	302	373	826	**1089**

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

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
- Better approach: normalise network output
- e.g. via batch normalisation

Demo: Normalized DQN in PacMan

Outline

Policy Gradient for Continuous Actions

- Represent policy by deep network $a = \pi(s, u)$ with weights u
- Define objective function as total discounted reward

$$J(u) = \mathbb{E}\left[r_1 + \gamma r_2 + \gamma^2 r_3 + ...\right]$$

- Optimise objective end-to-end by SGD
- i.e. Adjust policy parameters u to achieve more reward

Deterministic Policy Gradient

The gradient of the policy is given by

$$\begin{aligned}\frac{\partial J(u)}{\partial u} &= \mathbb{E}_s\left[\frac{\partial Q^\pi(s,a)}{\partial u}\right] \\ &= \mathbb{E}_s\left[\frac{\partial Q^\pi(s,a)}{\partial a}\frac{\partial \pi(s,u)}{\partial u}\right]\end{aligned}$$

Policy gradient is the direction that most improves Q

Deterministic Actor-Critic

Use two networks

- Actor is a policy $\pi(s, u)$ with parameters u

$$s \xrightarrow[u_1]{} \ldots \xrightarrow[u_n]{} a$$

- Critic is value function $Q(s, a, w)$ with parameters w

$$s, a \xrightarrow[w_1]{} \ldots \xrightarrow[w_n]{} Q$$

- Critic provides loss function for actor

$$s \xrightarrow[u_1]{} \ldots \xrightarrow[u_n]{} a \xrightarrow[w_1]{} \ldots \xrightarrow[w_n]{} Q$$

- Gradient backpropagates from critic into actor

$$\frac{\partial a}{\partial u} \longleftarrow \ldots \longleftarrow \frac{\partial Q}{\partial a} \longleftarrow \ldots \longleftarrow$$

Deterministic Actor-Critic: Learning Rule

- Critic estimates value of current policy by Q-learning

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

- Actor updates policy in direction that improves Q

$$\frac{\partial J(u)}{\partial u} = \mathbb{E}_s\left[\frac{\partial Q(s, a, w)}{\partial a}\frac{\partial \pi(s, u)}{\partial u}\right]$$

Deterministic Deep Policy Gradient (DDPG)

- Naive actor-critic oscillates or diverges with neural nets
- DDPG provides a stable solution

Deterministic Deep Policy Gradient (DDPG)

- Naive actor-critic oscillates or diverges with neural nets
- DDPG provides a stable solution

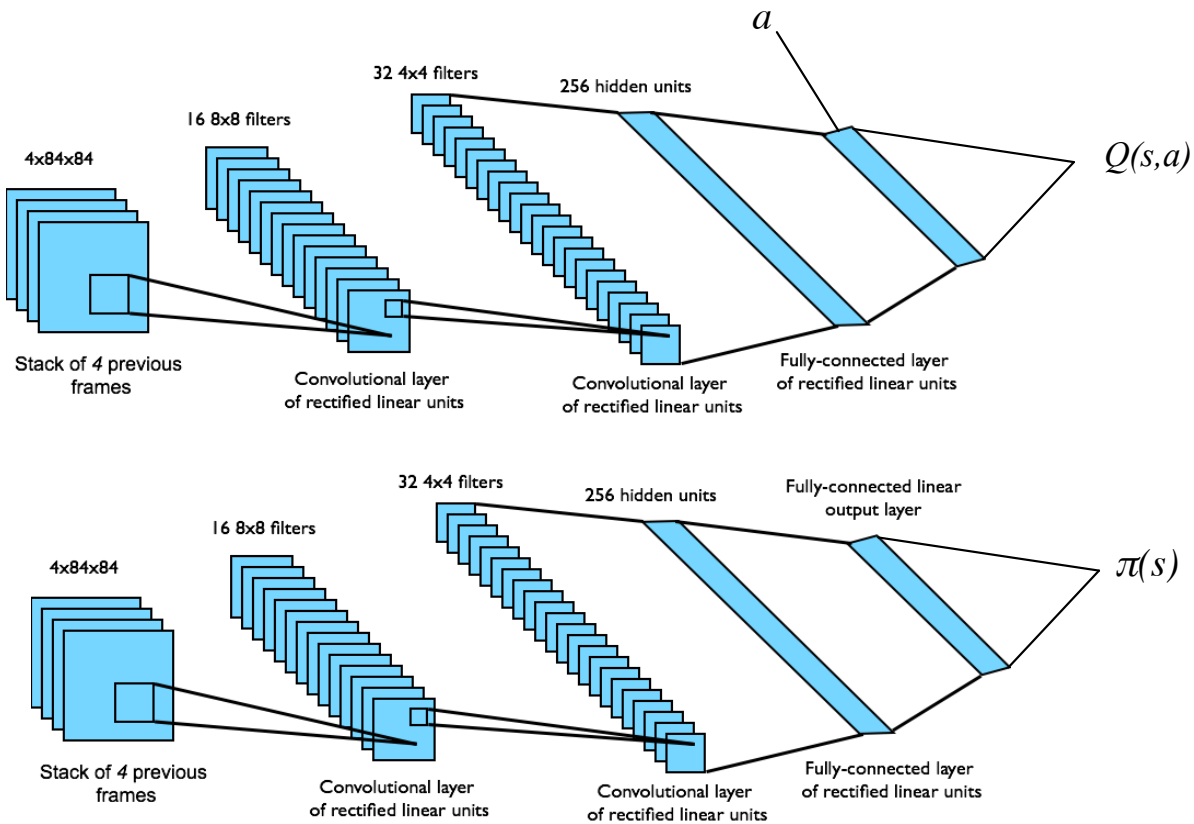
1. Use experience replay for both actor and critic
2. Freeze target network to avoid oscillations

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

$$\frac{\partial J(u)}{\partial u} = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}} \left[\frac{\partial Q(s, a, w)}{\partial a} \frac{\partial \pi(s, u)}{\partial u} \right]$$

DDPG for Continuous Control

- End-to-end learning of control policy from raw pixels s
- Input state s is stack of raw pixels from last 4 frames
- Two separate convnets are used for Q and π
- Physics are simulated in MuJoCo



[Lillicrap et al.]

DDPG Demo

Outline

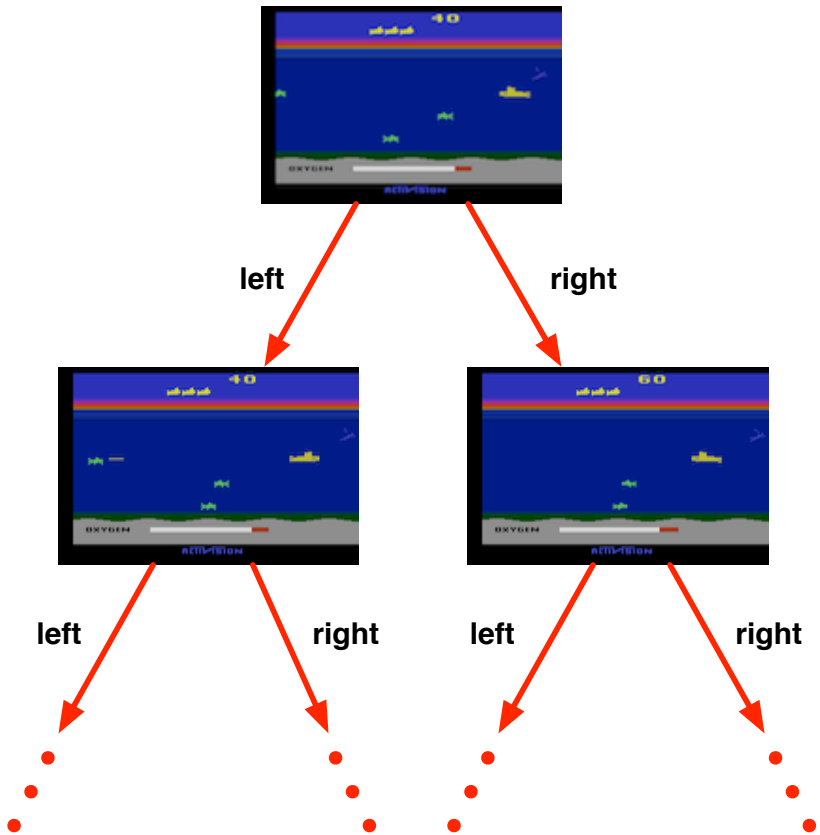
Model-Based RL

Learn a transition model of the environment

$$p(r, s' \mid s, a)$$

Plan using the transition model

- e.g. Lookahead using transition model to find optimal actions



Deep Models

- Represent transition model $p(r, s' \mid s, a)$ by deep network
- Define objective function measuring goodness of model
- e.g. number of bits to reconstruct next state (Gregor et al.)
- Optimise objective by SGD

DARN Demo

Challenges of Model-Based RL

Compounding errors

- Errors in the transition model compound over the trajectory
- By the end of a long trajectory, rewards can be totally wrong
- Model-based RL has failed (so far) in Atari

Challenges of Model-Based RL

Compounding errors

- Errors in the transition model compound over the trajectory
- By the end of a long trajectory, rewards can be totally wrong
- Model-based RL has failed (so far) in Atari

Deep networks of value/policy can "plan" implicitly

- Each layer of network performs arbitrary computational step
- n-layer network can "lookahead" n steps
- Are transition models required at all?

Deep Learning in Go

Monte-Carlo search

- Monte-Carlo search (MCTS) simulates future trajectories
- Builds large lookahead search tree with millions of positions
- State-of-the-art 19×19 Go programs use MCTS
- e.g. First strong Go program *MoGo*

(Gelly et al.)

Deep Learning in Go

Monte-Carlo search

- Monte-Carlo search (MCTS) simulates future trajectories
- Builds large lookahead search tree with millions of positions
- State-of-the-art 19×19 Go programs use MCTS
- e.g. First strong Go program *MoGo*

(Gelly et al.)

Convolutional Networks

- 12-layer convnet trained to predict expert moves
- Raw convnet (looking at 1 position, no search at all)
- Equals performance of MoGo with 10^5 position search tree

(Maddison et al.)

Program	Accuracy
Human 6-dan	$\sim$ 52%
12-Layer ConvNet	55%
8-Layer ConvNet*	44%
Prior state-of-the-art	31-39%

Program	Winning rate
GnuGo	97%
MoGo (100k)	46%
Pachi (10k)	47%
Pachi (100k)	11%

*Clarke & Storkey

Conclusion

- RL provides a general-purpose framework for AI
- RL problems can be solved by end-to-end deep learning
- A single agent can now solve many challenging tasks
- Reinforcement learning + deep learning = AI

Questions?

"The only stupid question is the one you never asked" -*Rich Sutton*