# Reinforcement Learning

CS 59300: RL1

September 11, 2025

Joseph Campbell

Department of Computer Science

## Today's lecture

1. Off-policy learning and Q-learning

2. Deep Q-Networks

*Some content inspired by David Silver's UCL RL course and Katerina Fragkiadaki's CMU 10-403*

## Recap: Temporal Difference policy evaluation

In Monte Carlo learning, our “target” is the actual return

$$Q(s,a) = Q(s,a) + \alpha \left(G_t - Q(s,a)\right)$$

In Temporal Difference learning, our target is the *estimated* return

$$Q(s,a) = Q(s,a) + \alpha \left(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s,a)\right)$$

**Why?** Remember the Bellman expectation equations!

## Recap: Bellman expectation equation

We can decompose value functions into two parts:

- The immediate reward
- The expected future returns

$$\mathbb{E}[G_t | s_t = s] = \mathbb{E}[r_{t+1} + \gamma G_t | s_t = s]$$

State-value: $V_\pi(s) = \mathbb{E}[r_{t+1} + \gamma V_\pi(s_{t+1}) | s_t = s]$

Action-value: $Q_\pi(s, a) = \mathbb{E}[r_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1}) | s_t = s, a_t = a]$

# Recap: SARSA: TD on-policy learning

Look familiar? Same as before…

Two-step iterative algorithm. Randomly initialize policy $\pi$…

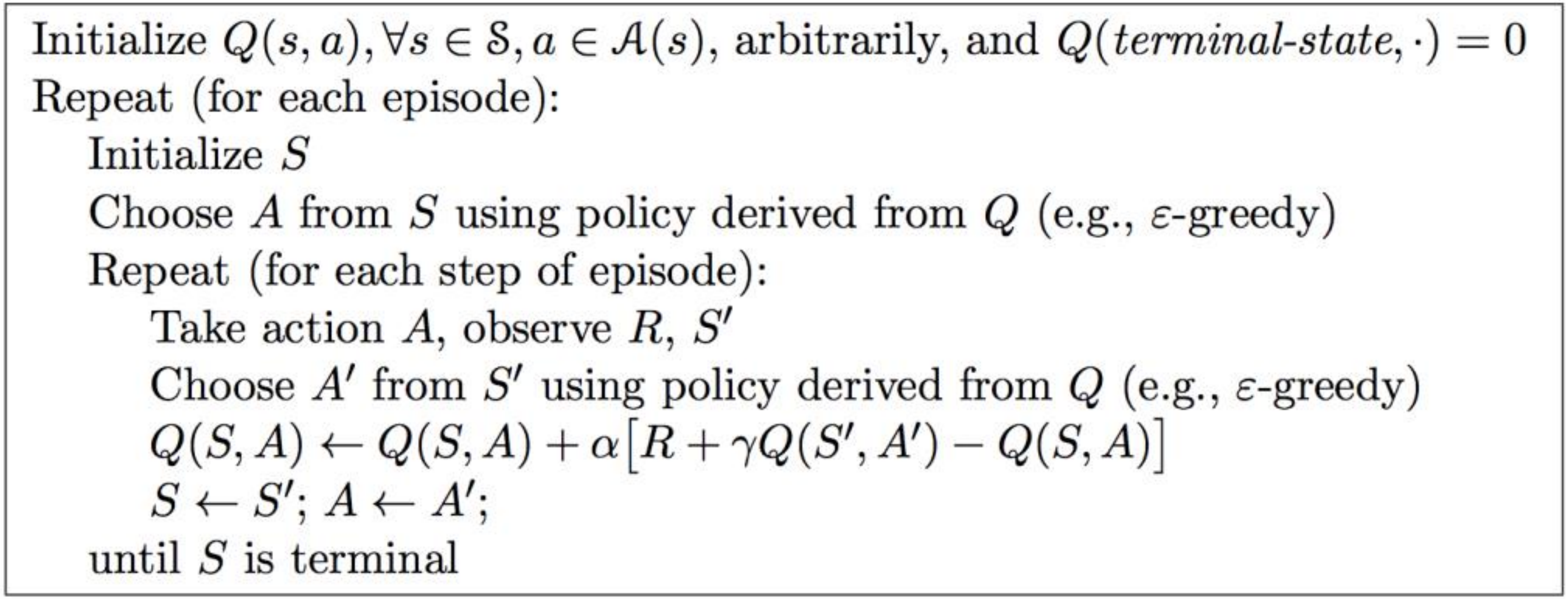
- Evaluate the policy with sampled episodes

$$Q_\pi(s,a) \text{ approximated TD estimate}$$

- Improve the policy by acting $\epsilon$-greedily with respect to $Q_\pi$

$$\pi' = \epsilon{-}\text{greedy } Q(s,a)$$

# Recap: SARSA: TD on-policy learning



```
Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Repeat (for each step of episode):
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until $S$ is terminal
```

*From Sutton and Barto Chapter 6.4, which is why notation is slightly different.*

## Off-policy learning and Q-learning

## What is “on-policy” learning?

On-policy learning:

- “Learn on the job”
- The policy learns from its own experience
- Improve policy $\pi$ from episodes sampled from $\pi$

Off-policy learning:

- “Look over someone’s shoulder”
- The policy learns from another policy’s experience
- Improve policy $\pi$ from episodes sampled from $\beta$

## The problem with on-policy learning

**What makes the previous methods on-policy?**

## The problem with on-policy learning

**What makes the previous methods on-policy?**

- Because value functions are with respect to a policy $\pi$!

If we learn value functions with respect to $\epsilon$-greedy policy and then try to deploy a pure greedy policy, we may be sub-optimal

- The exploratory actions (which we now ignore) influenced value estimates
- The policy during training may have over-fit to these exploratory actions, which means it may now be sub-optimal

## The problem with on-policy learning

What can we do to solve this?

## The problem with on-policy learning

What can we do to solve this?

- Decay $\epsilon$ in $\epsilon$-greedy
- Use $\epsilon$-greedy during deployment

Or…we use off-policy learning

# Off-policy learning

Goal: evaluate target policy $\pi(a|s)$ to compute $V_\pi(s)$ or $Q_\pi(s, a)$ while following behavior policy $\mu(a|s)$

- This means exploration is handled by $\mu$!

$$s_1, a_1, r_2, \dots s_t \sim \mu$$

Benefits:

- Learn from observing humans or other agents
- Re-use experience from old policies
- Learn optimal policy while following sub-optimal exploratory policy

## Off-policy learning

The easiest way is to apply an importance sampling ratio

$$\mathbb{E}_{X\sim P}[f(X)] = \mathbb{E}_{X\sim Q}\left[\frac{P(X)}{Q(X)} f(X)\right]$$

$$\Big\downarrow$$

$$\rho_t = \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}$$

## Simple derivation

Assume discrete actions and $P$ and $Q$ are probability masses

$$\mathbb{E}_{X\sim P}[f(X)] = \sum_x f(x)P(x)$$

## Simple derivation

Assume discrete actions and $P$ and $Q$ are probability masses

$$\mathbb{E}_{X\sim P}[f(X)] = \sum_x f(x)P(x)$$

But we need Q since the expectation is w.r.t. Q

$$\mathbb{E}_{X\sim Q}[f(X)] = \sum_x f(x)P(x)$$

## Simple derivation

Assume discrete actions and $P$ and $Q$ are probability masses

$$\mathbb{E}_{X\sim P}[f(X)] = \sum_x f(x)P(x)$$

Sums to 1 so adding no new information

$$\mathbb{E}_{X\sim Q}[f(X)] = \sum_x f(x)P(x) = \sum_x f(x)P(x)\frac{Q(x)}{Q(x)}$$

## Simple derivation

Assume discrete actions and $P$ and $Q$ are probability masses

$$\mathbb{E}_{X\sim P}[f(X)] = \sum_x f(x)P(x)$$

$$\mathbb{E}_{X\sim Q}[f(X)] = \sum_x f(x)P(x) = \sum_x f(x)P(x)\frac{Q(x)}{Q(x)} = \sum_x f(x)\frac{P(x)}{Q(x)}Q(x)$$

## Importance sampling intuition

$$\rho_t = \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}$$

**Idea:** the ratio of the probability of taking action $a_t$ under $\pi$ to the probability of taking the same action under $\mu$

- Corrects for the discrepancy between the likelihood of the observed action under the target policy and the behavior policy
- Reweights the data so that, on average, the adjusted data looks like what *would* have been generated by the target policy

## Importance sampling intuition

$$\rho_t = \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}$$

Off-policy learning is a fundamental aspect of reinforcement learning. Importance ratios are everywhere!

**Idea:** t the
probab

- Corrects for the discrepancy between the likelihood of the observed action under the target policy and the behavior policy
- Reweights the data so that, on average, the adjusted data looks like what *would* have been generated by the target policy

## TD off-policy learning

Weight TD target by the importance sampling ratio

$$Q(s,a) = Q(s,a) + \alpha\left(\frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}(r_{t+1}+\gamma Q(s_{t+1},a_{t+1})) - Q(s,a)\right)$$

However, if target and behavior policies are different, leads to high variance and unstable learning…

# Another way for off-policy learning

**Can we do it without importance sampling?** *Hint: value iteration*

# Another way for off-policy learning

**Can we do it without importance sampling?** *Hint: value iteration*

Yes! By trying to estimate the optimal value function!

- Rather than estimating values corresponding to the current policy
- Instead estimate values corresponding to the optimal greedy policy

## Q-learning: off-policy TD learning

In Sarsa…

$$Q(s,a) = Q(s,a) + \alpha\,(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s,a))$$

In Q-learning…

$$Q(s,a) = Q(s,a) + \alpha\,(r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s,a))$$

## Q-learning: off-policy TD learning

In Sarsa…

$$Q(s,a) = Q(s,a) + \alpha\left(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s,a)\right)$$

In Q-learning…

$$Q(s,a) = Q(s,a) + \alpha\left(r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s,a)\right)$$

Remember Bellman optimality equations?

$$Q^*(s,a) = \sum_{s',r} p(s', r | s, a)[r + \gamma \max_{a \in \mathcal{A}} Q^*(s', a')]$$

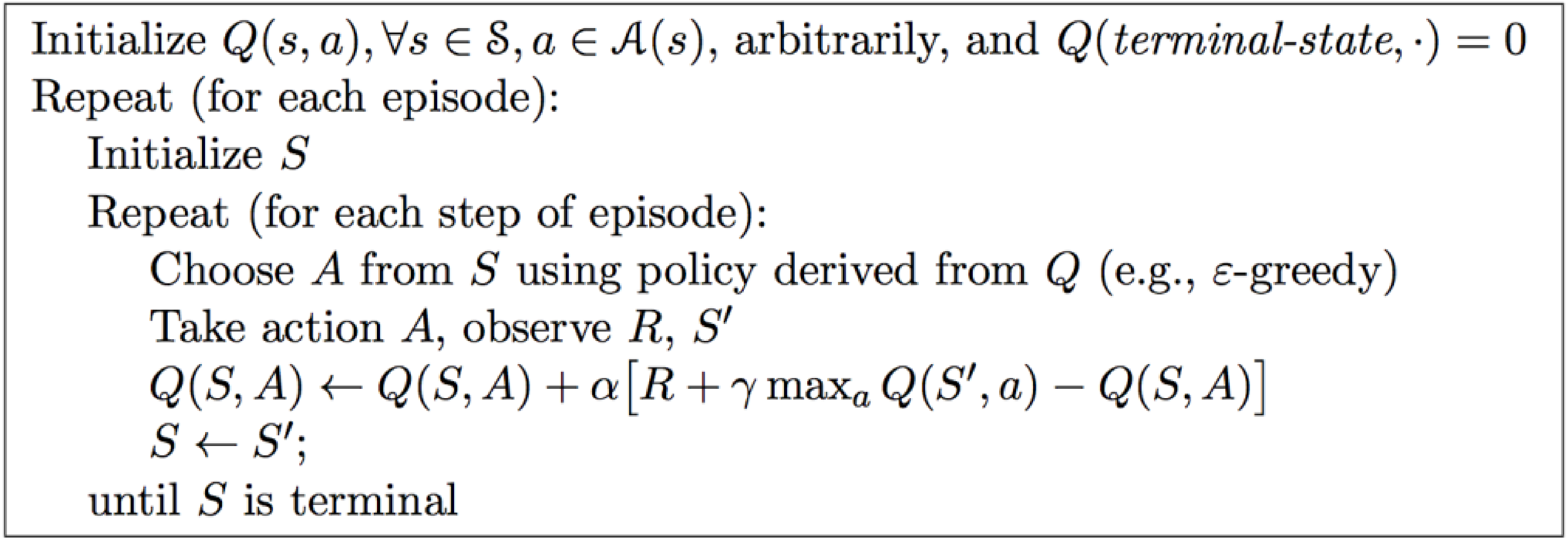
# Q-learning intuition

**Idea:** Q-learning updates the action-value estimates to be closer to the optimal value function. No need for correction!

Why does this work? Because the target policy isn't changing. It's always the greedy policy!

- "Pretend" that the optimal action is taken next timestep when calculating the TD target
- Can learn with any behavior policy and exploratory actions

## Q-learning: off-policy TD learning



```
Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Repeat (for each step of episode):
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        $S \leftarrow S'$;
    until $S$ is terminal
```

Q-learning

Chooses next greedy action **irrespective of policy.**

```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Repeat (for each step of episode):
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \underline{\gamma \max_a Q(S',a) - Q(S,A)}]$
        $S \leftarrow S'$;
    until $S$ is terminal
```

SARSA

Chooses next action **from current policy**.

```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Repeat (for each step of episode):
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \underline{\gamma Q(S',A') - Q(S,A)}]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until $S$ is terminal
```

## Deep Q-Networks

## How do we solve large-scale problems?

So far we have talked about value functions in terms of *lookup tables*
- Every state or state-action pair has an entry

But this doesn't work with large state spaces!
- Backgammon = $10^{20}$ states
- Go = $10^{170}$ states
- Robotics = continuous (usually)

Can you make a lookup table big enough?

## Value function approximation

Instead, we want to *approximate* the value function with a model

- $\hat{V}_\pi(s) \approx V_\pi(s)$ and $\hat{Q}_\pi(s,a) \approx Q_\pi(s,a)$

What do I mean by model?

- A parameterized statistical or machine learning model

Benefits:

- Generalize to unseen states (continuous state spaces anyone?)
- Update parameters of model using MC or TD learning

## Many types of models

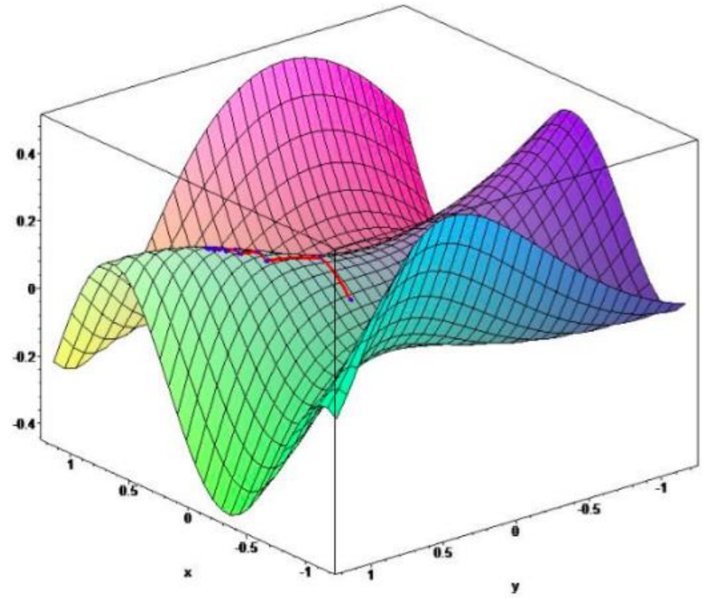
- Linear/non-linear regression
- **Neural networks**
- Decision tree
- Fourier bases
- …

We only care about neural networks in this class

## Gradient descent

Let $J(\mathbf{w})$ be a differentiable function of parameter vector $\mathbf{w}$

- Define the gradient as
- $$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_n} \end{pmatrix}$$
- To find local minimum, adjust $\mathbf{w}$ in the direction of negative gradient



## Value function approximation in RL

Goal: find the parameters $\mathbf{w}$ which minimize the mean-squared error between the true value function and our approximation

$$J(\mathbf{w}) = \mathbb{E}_\pi[\left(Q_\pi(s,a) - \hat{Q}(s,a,\mathbf{w})\right)^2]$$

## Value function approximation in RL

Goal: find the parameters $\mathbf{w}$ which minimize the mean-squared error between the true value function and our approximation

$$J(\mathbf{w}) = \mathbb{E}_\pi[\left(Q_\pi(s,a) - \hat{Q}(s,a,\mathbf{w})\right)^2]$$

This is standard gradient descent as used in supervised learning

- Except in RL our supervised value is either $G_t$ or the TD target

## So…let's throw a DNN at Q-learning?

Simply replace our original estimate of $Q$ with our approximation

$$\hat{Q}(s,a,\mathbf{w}) = \hat{Q}(s,a,\mathbf{w}) + \alpha\left(r_{t+1} + \gamma \max_{a' \in \mathcal{A}} \hat{Q}\left(s_{t+1}, a', \mathbf{w}\right) - \hat{Q}(s,a,\mathbf{w})\right)$$

Our weight update looks something like

$$\Delta\mathbf{w} = \alpha\left(r_{t+1} + \gamma \max_{a' \in \mathcal{A}} \hat{Q}\left(s_{t+1}, a', \mathbf{w}\right) - \hat{Q}(s,a,\mathbf{w})\right) \nabla_{\mathbf{w}} \hat{Q}(s,a,\mathbf{w})$$

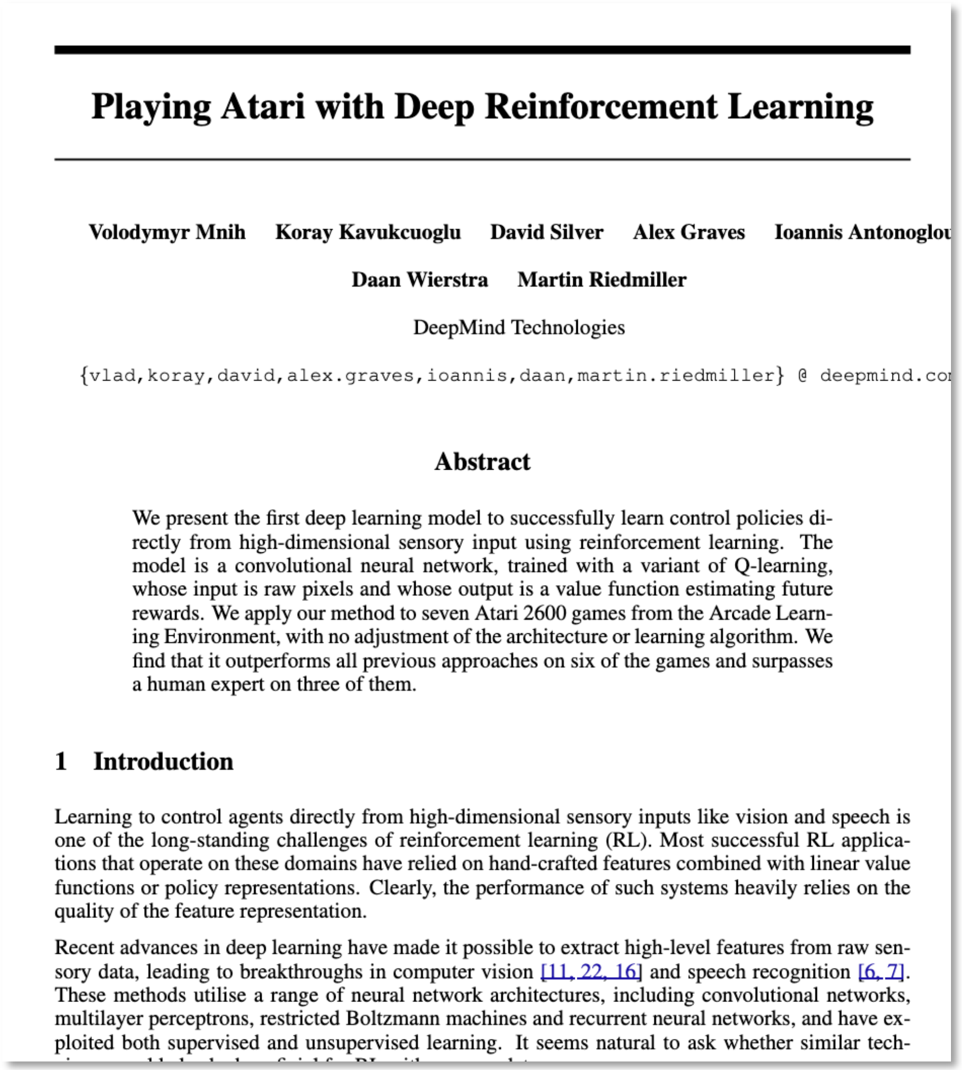
## So…let’s throw a DNN at Q-learning?

Simply replace our original estimate of $Q$ with our approximation

$$\hat{Q}(s,a,\mathbf{w}) = \hat{Q}(s,a,\mathbf{w}) + \alpha\left(r_{t+1} + \gamma \max_{a' \in \mathcal{A}} \hat{Q}\,(s_{t+1},a',\mathbf{w}) - \hat{Q}(s,a,\mathbf{w})\right)$$

$$\Delta\mathbf{w} = \overset{\text{Step size}}{\alpha}\left(\overbrace{r_{t+1} + \gamma \max_{a' \in \mathcal{A}} \hat{Q}\,(s_{t+1},a',\mathbf{w}) - \hat{Q}(s,a,\mathbf{w})}^{\text{Magnitude of error}}\right)\overbrace{\nabla_{\mathbf{w}}\,\hat{Q}(s,a,\mathbf{w})}^{\text{Direction of error}}$$

# Deep Q-Networks



## Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih  Koray Kavukcuoglu  David Silver  Alex Graves  Ioannis Antonoglou
Daan Wierstra  Martin Riedmiller
DeepMind Technologies
{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.co

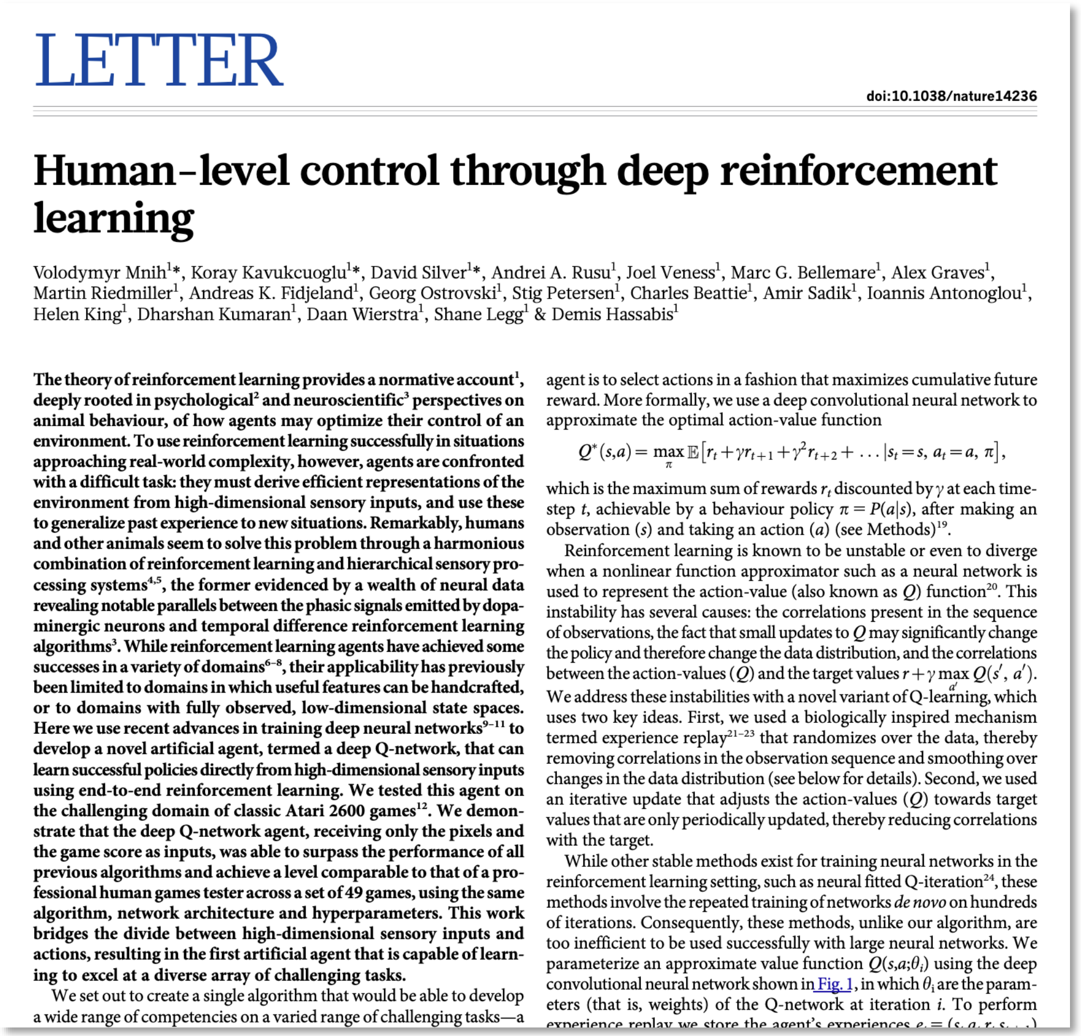
### Abstract

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

### 1 Introduction

Learning to control agents directly from high-dimensional sensory inputs like vision and speech is one of the long-standing challenges of reinforcement learning (RL). Most successful RL applications that operate on these domains have relied on hand-crafted features combined with linear value functions or policy representations. Clearly, the performance of such systems heavily relies on the quality of the feature representation.

Recent advances in deep learning have made it possible to extract high-level features from raw sensory data, leading to breakthroughs in computer vision [11, 22, 16] and speech recognition [6, 7]. These methods utilise a range of neural network architectures, including convolutional networks, multilayer perceptrons, restricted Boltzmann machines and recurrent neural networks, and have exploited both supervised and unsupervised learning. It seems natural to ask whether similar tech-



LETTER

doi:10.1038/nature14236

## Human-level control through deep reinforcement learning

Volodymyr Mnih[1]*, Koray Kavukcuoglu[1]*, David Silver[1]*, Andrei A. Rusu[1], Joel Veness[1], Marc G. Bellemare[1], Alex Graves[1], Martin Riedmiller[1], Andreas K. Fidjeland[1], Georg Ostrovski[1], Stig Petersen[1], Charles Beattie[1], Amir Sadik[1], Ioannis Antonoglou[1], Helen King[1], Dharshan Kumaran[1], Daan Wierstra[1], Shane Legg[1] & Demis Hassabis[1]

**The theory of reinforcement learning provides a normative account[1], deeply rooted in psychological[2] and neuroscientific[3] perspectives on animal behaviour, of how agents may optimize their control of an environment. To use reinforcement learning successfully in situations approaching real-world complexity, however, agents are confronted with a difficult task: they must derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalize past experience to new situations. Remarkably, humans and other animals seem to solve this problem through a harmonious combination of reinforcement learning and hierarchical sensory processing systems[4,5], the former evidenced by a wealth of neural data revealing notable parallels between the phasic signals emitted by dopaminergic neurons and temporal difference reinforcement learning algorithms[3]. While reinforcement learning agents have achieved some successes in a variety of domains[6–8], their applicability has previously been limited to domains in which useful features can be handcrafted, or to domains with fully observed, low-dimensional state spaces. Here we use recent advances in training deep neural networks[9–11] to develop a novel artificial agent, termed a deep Q-network, that can learn successful policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning. We tested this agent on the challenging domain of classic Atari 2600 games[12]. We demonstrate that the deep Q-network agent, receiving only the pixels and the game score as inputs, was able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human games tester across a set of 49 games, using the same algorithm, network architecture and hyperparameters. This work bridges the divide between high-dimensional sensory inputs and actions, resulting in the first artificial agent that is capable of learning to excel at a diverse array of challenging tasks.**

We set out to create a single algorithm that would be able to develop a wide range of competencies on a varied range of challenging tasks—a agent is to select actions in a fashion that maximizes cumulative future reward. More formally, we use a deep convolutional neural network to approximate the optimal action-value function

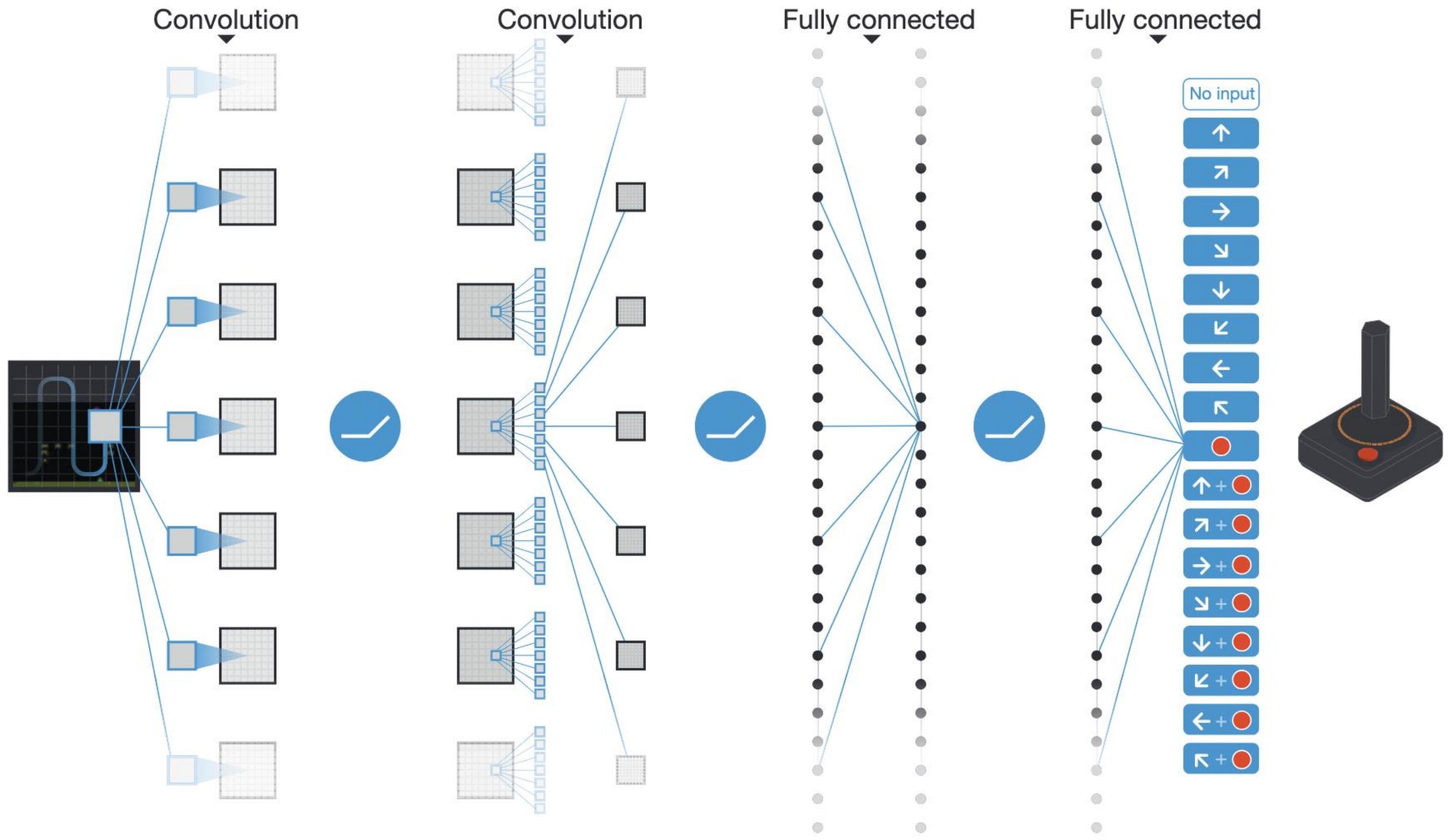
$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = s,\, a_t = a,\, \pi\right],$$

which is the maximum sum of rewards $r_t$ discounted by $\gamma$ at each time-step $t$, achievable by a behaviour policy $\pi = P(a|s)$, after making an observation ($s$) and taking an action ($a$) (see Methods)[19].

Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as $Q$) function[20]. This instability has several causes: the correlations present in the sequence of observations, the fact that small updates to $Q$ may significantly change the policy and therefore change the data distribution, and the correlations between the action-values ($Q$) and the target values $r + \gamma \max_{a'} Q(s', a')$. We address these instabilities with a novel variant of Q-learning, which uses two key ideas. First, we used a biologically inspired mechanism termed experience replay[21–23] that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution (see below for details). Second, we used an iterative update that adjusts the action-values ($Q$) towards target values that are only periodically updated, thereby reducing correlations with the target.

While other stable methods exist for training neural networks in the reinforcement learning setting, such as neural fitted Q-iteration[24], these methods involve the repeated training of networks *de novo* on hundreds of iterations. Consequently, these methods, unlike our algorithm, are too inefficient to be used successfully with large neural networks. We parameterize an approximate value function $Q(s,a;\theta_i)$ using the deep convolutional neural network shown in Fig. 1, in which $\theta_i$ are the parameters (that is, weights) of the Q-network at iteration $i$. To perform experience replay we store the agent's experiences $e_t = (s_t, a_t, r_t, s_{t+1})$

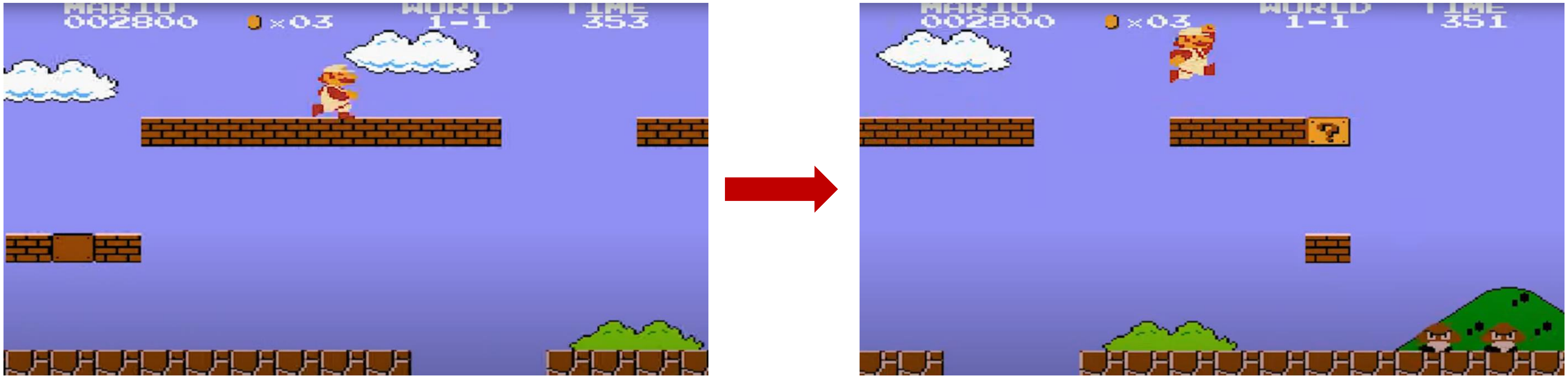
180 2 1



Mnih et al. Human-level Control Through Deep Reinforcement Learning. 2015.

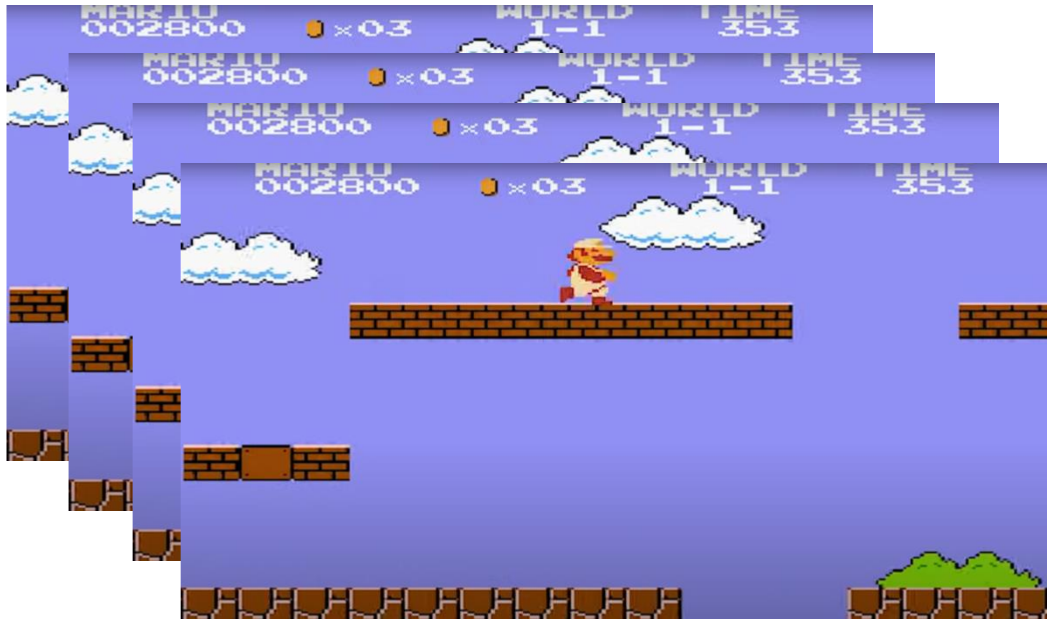
## How do we account for temporal information?

Remember the Markov assumption?



How do we know an object’s velocity from a single image?

# Frame stacking



Stack the last N frames together

Commonly, N=4

## How does a CNN process 4 images at a time?

Pre-process each image such that:

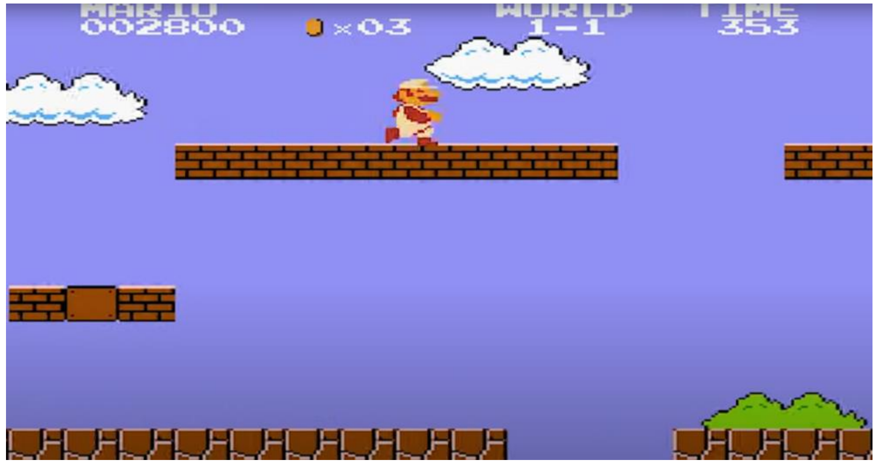
- Down-sampled and cropped to 84x84 pixels
- 1-channel grayscale

Each image is thus 84x84x1. After frame stacking we have 84x84xN

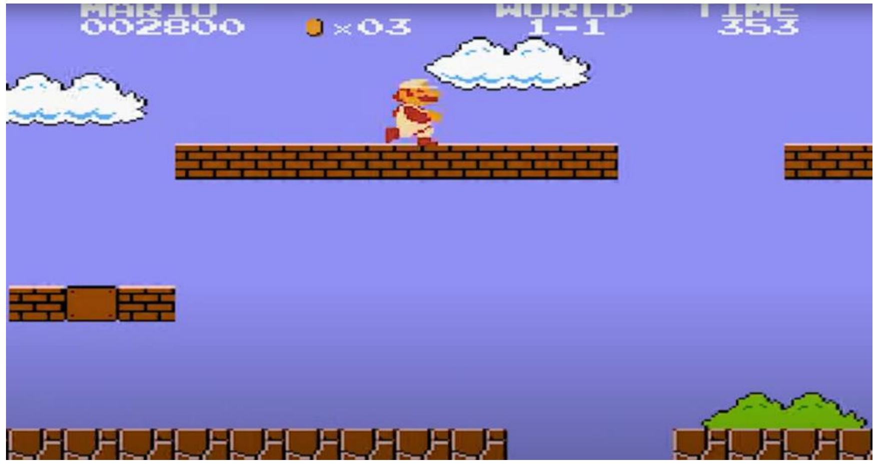
**Solution:** treat each image as a separate “channel”

- Convolutions are over 2D space but filters have depth N

# Action frequency



$t = 1$



$t = 2$

Suppose the game runs at 24 frames per second

- The difference between these two frames is 1/24 of a second

Not enough time has elapsed to tell if the action was meaningful!

## How can we solve this?

# How can we solve this?

We repeat actions!

States that we collect and train over

$$s_1, a_1 \to s_2, a_1 \to s_3, a_1 \to s_4, a_1 \to s_5, a_2 \to s_6, a_2 \ldots$$

Repeated actions

Repeat an action K times and skip the intermediate frames

- Here K=4

## There are some caveats to make this work

While $Q^*$ might converge using a table lookup representation, this does not hold true for deep neural networks!

Diverges due to

- Correlations between samples
- Non-stationary TD targets

## Dealing with correlations

Consecutive samples are strongly correlated

- **What does this mean?**

## Dealing with correlations

Consecutive samples are strongly correlated

- An agent interacts with the environment in a consecutive manner
- The experiences in a single episode are highly correlated, e.g. the state it visits at time $t$ and time $t + 1$ are very similar

# Dealing with correlations

Consecutive samples are strongly correlated

- An agent interacts with the environment in a consecutive manner
- The experiences in a single episode are highly correlated, e.g. the state it visits at time $t$ and time $t + 1$ are very similar

**Why is this a problem?**

## Dealing with correlations

Consecutive samples are strongly correlated

- An agent interacts with the environment in a consecutive manner
- The experiences in a single episode are highly correlated, e.g. the state it visits at time $t$ and time $t + 1$ are very similar

**Why is this a problem?**

- Formally: it violates the i.i.d. assumption of supervised learning
- Informally: because it leads to biased updates. The model might fit to particular patterns not representative of the overall environment

## Dealing with correlations

The effect of this:

- May cause value updates to oscillate and diverge
- Potential feedback errors, small errors get amplified over time

**Any ideas on how to solve this?**

*Hint: DQN is off-policy, so can we store data…?*

## Experience replay

Instead of using consecutive values, store all samples in a buffer

During training, we randomly sample experiences from the buffer

- Helps approximate i.i.d. assumption (any 2 samples are unlikely to be strongly correlated)
- Breaks temporal correlations and stabilizes training

```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode = 1...M do
    Initialize sequence $s_1 = \{x_1\}$ and pre. seq. $\phi_1 = \phi(s_1)$
    for t=1...T do
        With probability $\epsilon$ select a random action $a_t$
            otherwise $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ and observe $r_t$ and $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and pre. $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch from $D$
        Set $y_j = r_j$ if episode ends else $r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta)$
        Perform a gradient step on $(y_j - Q(\phi_j, a_j; \theta))^2$
```

## Non-stationary TD targets

Current $Q$ values are used to determine the TD targets.

**Why is this a problem?**

- Hint: feedback loop

## Non-stationary TD targets

Current $Q$ values are used to determine the TD targets.

**Why is this a problem?**

- As the network updates its weights to reduce the error between predicted value and target value, the target itself changes!
- Can cause the network to chase a moving target
- Leads to oscillation and instability

## Non-stationary TD targets

Current $Q$ values are used to determine the TD targets.

**Why is this a problem?**

- As the network updates its weights to reduce the error between predicted value and target value, the target itself changes!
- Can cause the network to chase a moving target
- Leads to oscillation and instability

# Use a “target network”

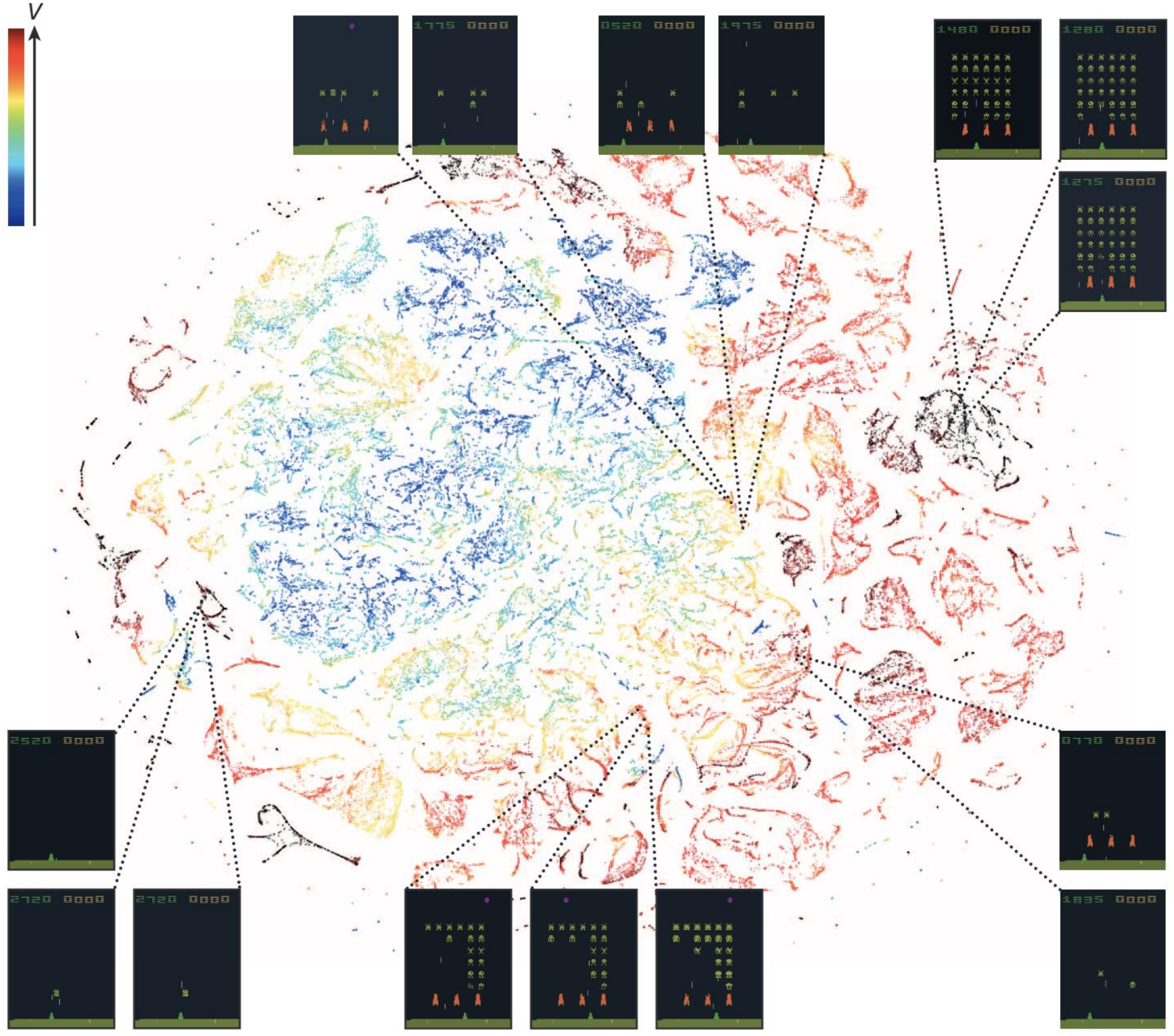
Make a copy of the Q-network that is used to calculate targets

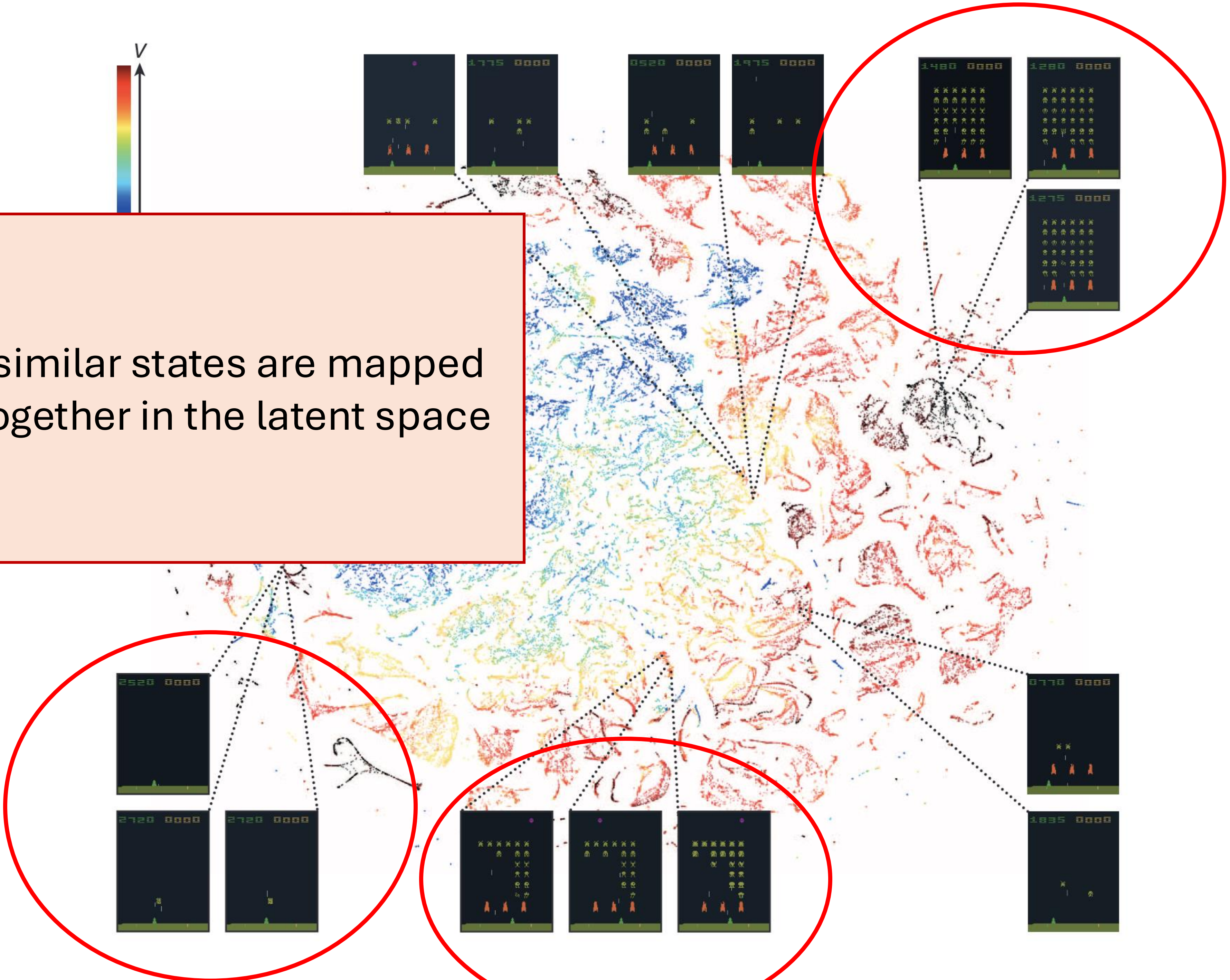
- This target network is updated less frequently
- Every few thousand steps

This fixes the problem because the target Q-values are more stable

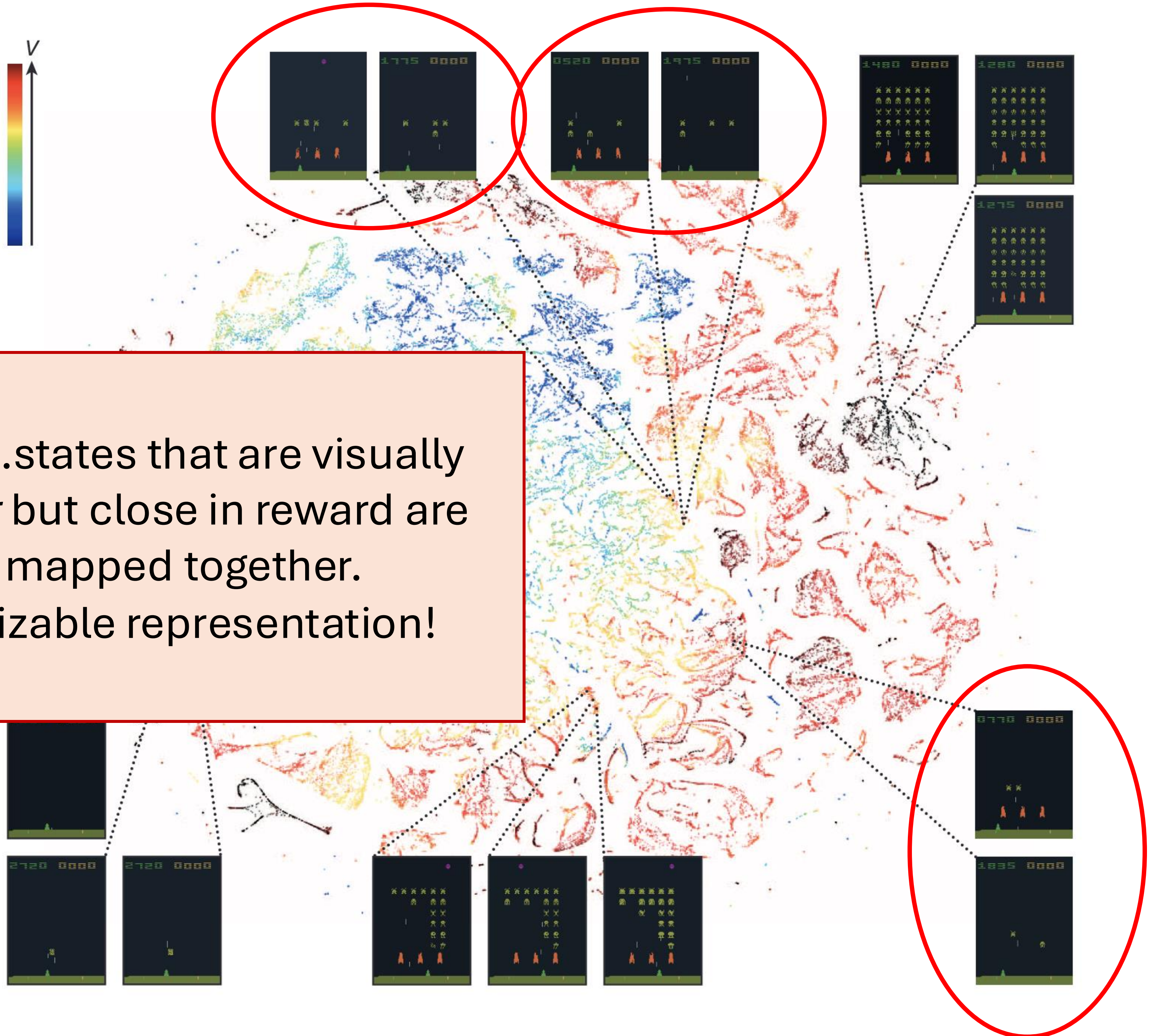
- Targets determined by a set of weights that don’t change as much
- Slows down updates to the Q-values, no large erratic changes
- Reduces oscillation and feedback

```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$ and $\hat{Q}$ with $\theta_2 = \theta$
for episode = 1...M do
    Initialize sequence $s_1 = \{x_1\}$ and pre. seq. $\phi_1 = \phi(s_1)$
    for t=1...T do
        With probability $\epsilon$ select a random action $a_t$
            otherwise $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ and observe $r_t$ and $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and pre. $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch from $D$
        Set $y_j = r_j$ if episode ends else $r_j + \gamma \max_{a'} \hat{Q}\{ (\phi_{j+1}, a'; \theta_2)$
        Perform a gradient step on $(y_j - Q(\phi_j, a_j; \theta))^2$
        Every C steps set $\hat{Q} = Q$
```


V
1775 0000
0520 0000
1975 0000
1480 0000
1280 0000
1275 0000
2520 0000
2720 0000
2720 0000
0770 0000
1835 0000



Visually similar states are mapped closely together in the latent space



But also…states that are visually *dissimilar* but close in reward are also mapped together. Generalizable representation!

## But it’s not perfect

Fails at games that require planning and foresight

- E.g. Ms. Pacman

Doesn’t transfer any knowledge between games

- Each game has a separate Q network

Only works for games with discrete action spaces

The system is trained on about
1000 visually and physically diverse objects

https://youtu.be/W4joe3zzglU?si=9milzRXNGDnbhz3m