DQN

Deep Q-Networks

Angela Carnevale

MathLearn - Galway, June 2026

0. Brief Recap

Q-table : $Q(s, a)$

state · action

state-action function

governed by Bellman equation (together with $V(s)$)

Use TD-target & TD-control to update Q-values

That is: Q-learning learns a table

0. Brief Recap

Q-table : $Q(s,a)$ update according to

learning rate

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

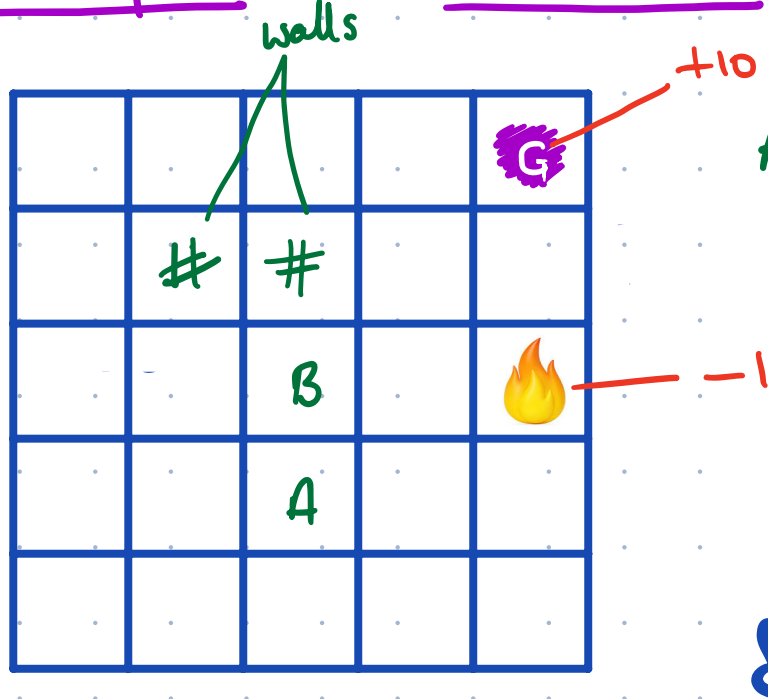
discount factor

TD-target

TD-error

So each update improves one number in our $|\hat{S}| \times |\hat{A}|$ table...

Example GridWorld



Actions: {↑, ↓, ←, →}

Reward -1 for each move

So our Q table is a

(5.5) x 4 = 25x4 table ✓

Example Matrix World

$$\begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix} \in \mathrm{Mat}_{3\times 3}(\mathbb{F}_2)$$

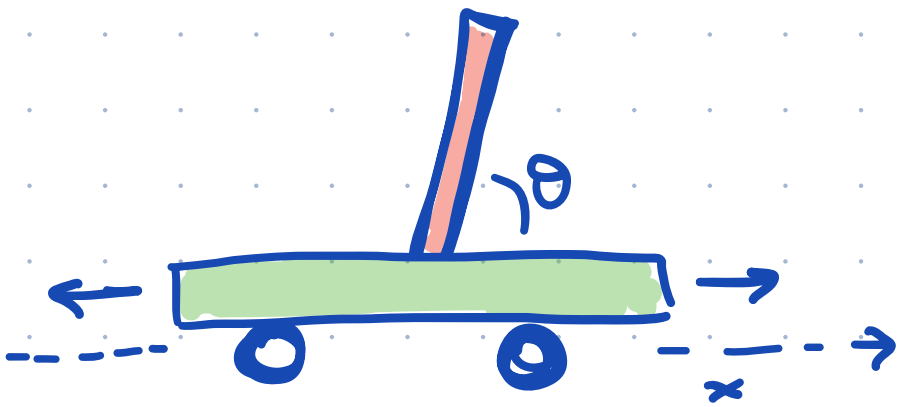
Goal: turn into upper triangular form using

Actions := {row operations}

$$|Q\text{-table}| = \underbrace{2^{3^2}}_{|S|} \cdot \underbrace{\binom{3}{2} \cdot 2 \cdot \binom{3}{2}}_{|A|}$$

becomes huge very quickly!

Example CartPole



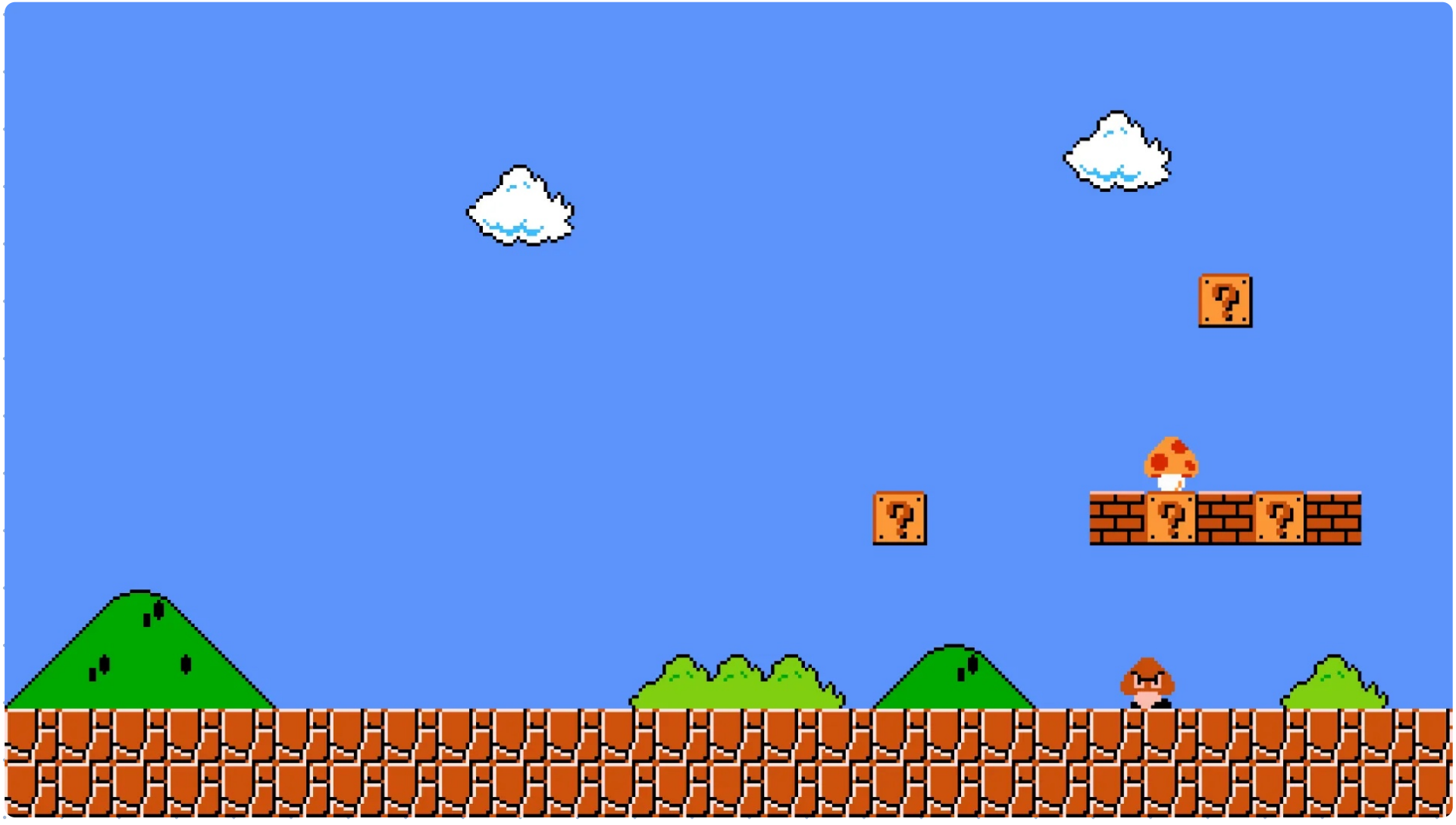
State: $(x, \dot{x}, \theta, \dot{\theta})$

Actions: $\{\leftarrow, \rightarrow\}$

Only 2 columns but continuous states

Example Mario!

Example Mario!



We need to:

- be able to deal with large size
- make more efficient updates
- be able to generalise across states

Idea: bring in a neural network ...

("learn the table's pattern")

Q-table → $Q_\theta(s,a)$ → DQN

... and then make it work

Towards $Q_\theta(s,a)$

1. Features and weights

We can view our table lookup for the entry $Q(s,a)$ in

$Q(s,a)$	a_1	a_2	...
s_1			
s_2			
⋮			

as: $Q(s,a) = w^T \phi(s,a)$

weights

feature vector $\phi(s,a)$

$$\begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

but we can generalise the "state feature" to a set of features describing the state

1. Features and weights

Example GridWorld: from $(s,a) = ((x,y), a)$ to

$$\phi(s,a) = \begin{bmatrix} \text{distance to goal} \\ \text{is near wall} \\ \text{is high reward nearby} \\ \text{is near lava } \text{😱} \\ \text{action is down} \\ \vdots \end{bmatrix}$$

and use this to approximate

$$Q_w(s,a) = w^T \phi(s,a)$$

1. Features and weights

From here:

→ can use same Q-learning target,

TD error etc

→ update weights

already improved our update: now it's not a single entry but all weights.

However, hand-designed features are not yet enough to deal with more complex states.

2. Introducing $Q_\theta(s,a)$

NN parameters → θ

Introducing a neural network $Q_\theta(s,a)$

we move from hand-designed features to learned features

For discrete actions:

$$Q_\theta(s,\cdot) = \begin{bmatrix} Q_\theta(s,a_1) \\ \vdots \\ Q_\theta(s,a_k) \end{bmatrix}$$

with action choice

$$a = \underset{a'}{\text{argmax}}\, Q(s,a')$$

2. Introducing $Q_\theta(s,a)$

First attempt: we want to predict $Q_\theta(s,a)$

Target: $y = r + \gamma \max_{a'} Q_\theta(s',a')$

Loss: $L(\theta) = (y - Q_\theta(s,a))^2$

Looks like supervised learning! But it is not, and this this creates problems.

Problems & fixes:

DQN

- highly correlated data
- chasing moving targets
- (large function approximation)

} "deadly triad"

3. Data correlation: the problem

In RL, we get highly correlated, serial data:

$$(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$$

and then

$$(S_{t+1}, A_{t+1}, R_{t+2}, S_{t+2}, A_{t+2})$$

and so on.

Example Walking on a stretch of very similar images in a Mario game, or along a corridor in GridWorld.

⚠ NNs become unstable when training on highly correlated data (standard assumption is iid)

3. Data correlation: The Fix

Replay Memory

Store transitions in a **Replay Buffer**

$$\mathcal{D} = \{(s, a, r, s', \text{done})\}$$

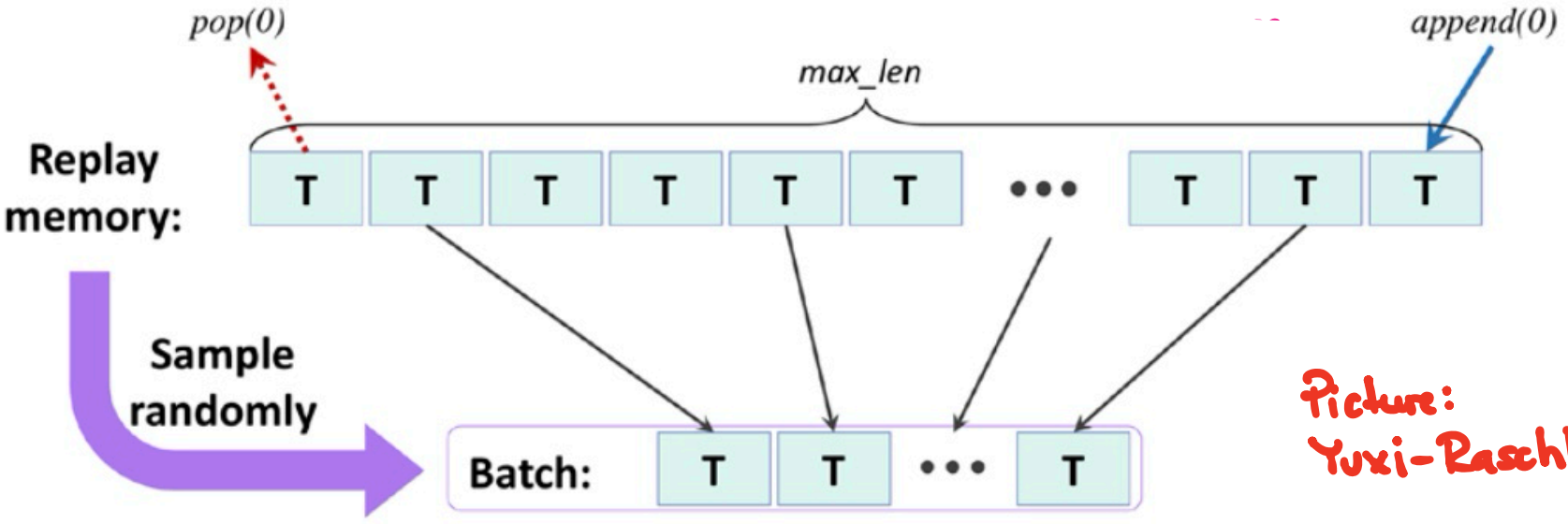
Then sample random mini-batches and train on those

Good! → breaks correlation
→ reuses experience

RL ends up "mini-batch supervised learning".

3. Data correlation: The Fix

Replay Memory (Flashcards!)



Picture: Yuxi-Raschka

Good! → breaks correlation
→ reuses experience

RL ≈ "mini-batch supervised learning".

4. Moving targets: the problem

Assuming we introduced a NN $Q_\theta(s,a)$, we have

prediction: $Q_\theta(s,a)$

target: $y = r + \gamma \max_{a'} Q_\theta(s',a')$

That is: the agent is chasing a moving target

4. Moving targets: the fix

DQN fixes this issue by introducing an auxiliary network. So it uses

online network Q_θ ← trained every update

target network Q_{θ^-} ← used to build targets

$$y = r + \gamma \max_{a'} Q_{\theta^-}(s', a')$$

Every C steps, we copy the online weights into target:

$$\theta^- \leftarrow \theta$$

4. Moving targets: the fix

Recall this means that DQN is off-policy.

Behaviour policy ≠ target policy

we also act according to an ε-greedy policy

(note this works nicely with experience replay!)

5. Putting the pieces together

Full DQN algorithm

1. Initialise online network Q_θ
2. Initialise target network $Q_{\theta^-} \leftarrow Q_\theta$
3. Initialise replay buffer D
4. Step:
   - → choose action (ε-greedy) — decay ε over time
   - → observe r, s', done
   - → store $(s, a, r, s', \text{done})$ in D
   - → sample mini-batch from D
   - → compute target

$$y = \begin{cases} r & \text{if done} \\ r + \gamma \max_{a'} Q_{\theta^-}(s', a') & \text{o/w} \end{cases}$$

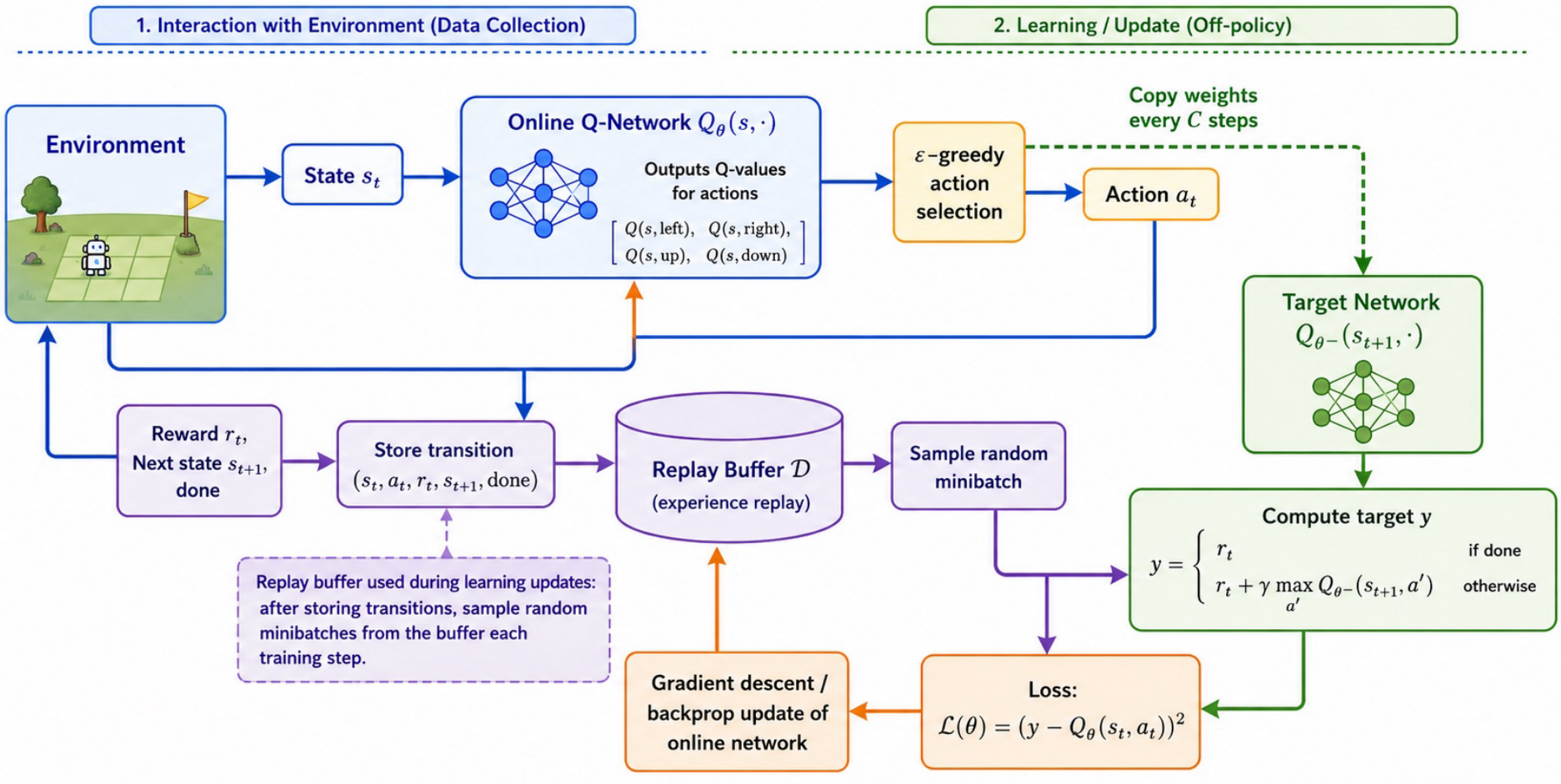
5. Putting the pieces together

Full DQN algorithm

5. Minimise $(y - Q_\theta(s,a))^2$

6. [every C steps] Update $Q_{\theta^-} \leftarrow Q_\theta$

Deep Q-Network (DQN) Workflow



Picture: GPT 5.5

6. What DQN is (not) good for

DQN can handle discrete action spaces, learns Q_θ and the policy is derived from that by taking argmax

In contrast, it cannot handle continuous / stochastic action spaces. For that (and other limitations) it might work to learn the policy (by optimising expected returns)

↝ POLICY GRADIENT METHODS & PPO