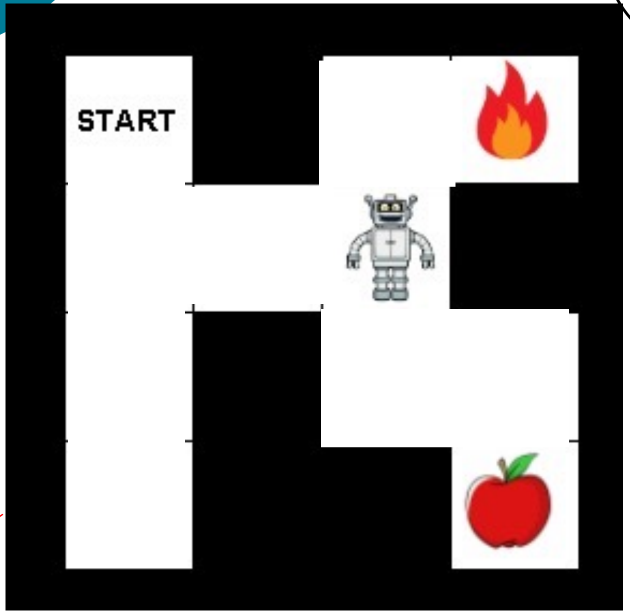
SIT796 Reinforcement Learning

# Deep Reinforcement Learning

Presented by:
Thommen George Karimpanal
School of Information Technology

# Recap: Function approximation



State

Action

$\longrightarrow$

| | $a_1$ | $a_2$ | $a_3$ |
|---|---|---|---|
| $s_1$ | $Q(s_1,a_1)$ | $Q(s_1,a_2)$ | $Q(s_1,a_3)$ |
| $s_2$ | $Q(s_2,a_1)$ | $Q(s_2,a_2)$ | $Q(s_2,a_3)$ |
| $s_3$ | $Q(s_3,a_1)$ | $Q(s_3,a_2)$ | $Q(s_3,a_3)$ |
| $s_4$ | $Q(s_4,a_1)$ | $Q(s_4,a_2)$ | $Q(s_4,a_3)$ |

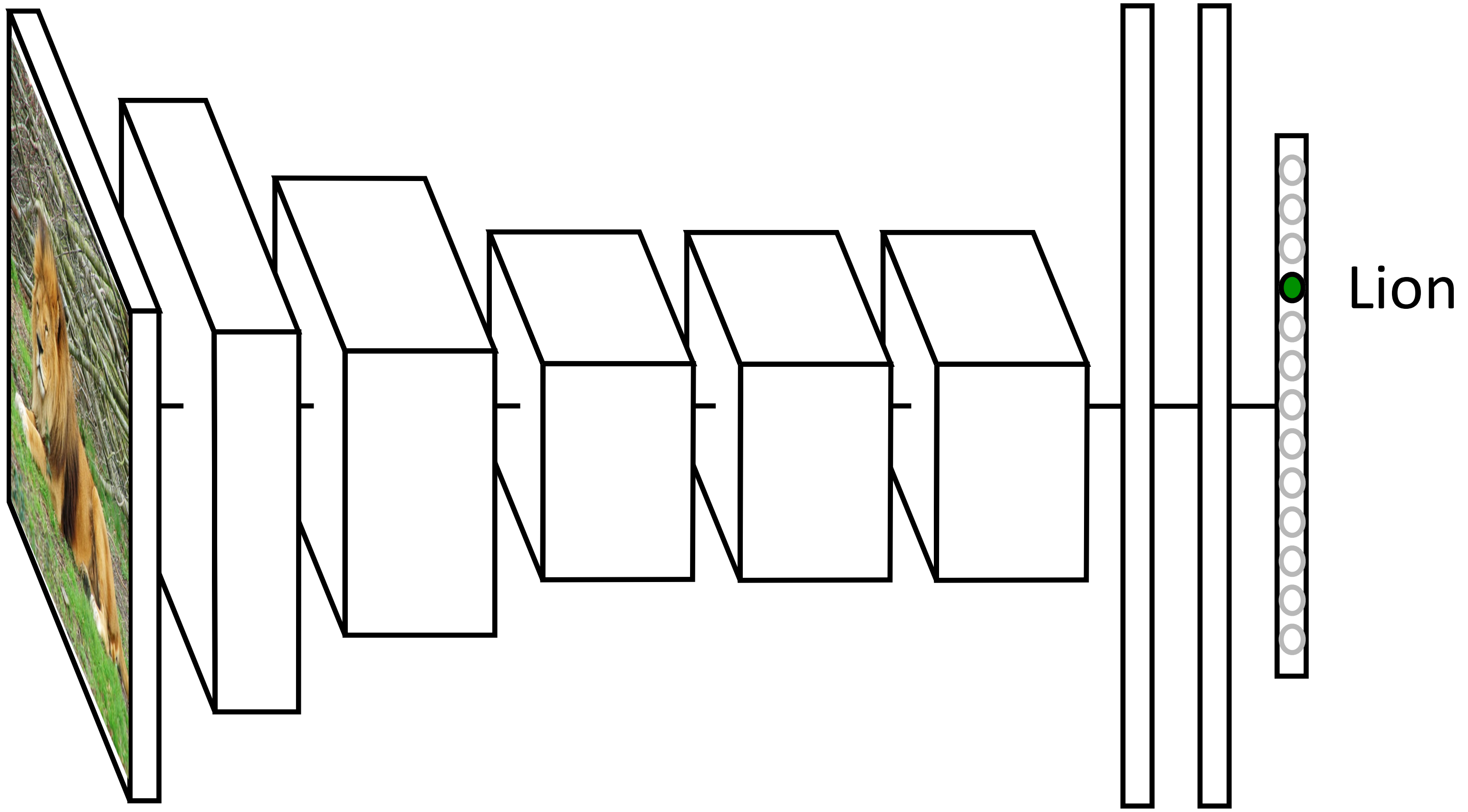
$\longrightarrow \quad \pi^*$

Function approximators
(Linear, RBF, NN)

## Neural Networks as a function approximator

- Rumelhart 1986 – great early success
- Interest subsides in the late 90's as other models are introduced – SVMs, Graphical models, etc.
- Convolutional Neural Nets – LeCun ,1989 ,for Image recognition, speech, etc.
- Deep Belief nets (Hinton) and Stacked auto-encoders (Bengio) in 2006
- Unsupervised pre-training followed by supervised training
  - Good feature extractors.
  - 2012 Initial successes with supervised approaches which overcome vanishing gradient

## What is a Deep Network?



Krizhevsky, Sutskever, Hinton — NIPS 2012

## Intuition…

- First layer learns 1st order features (e.g. edges…)
- Higher order layers learn combinations of features (combinations of edges, etc.)
- Some models learn in an unsupervised mode and discover general features of the input space – serving multiple tasks related to the unsupervised instances (image recognition, etc.)
- Final layer of transformed features are fed into supervised layer(s)
- And entire network is often subsequently tuned using supervised training of the entire net, using the initial weightings learned in the unsupervised phase

## Properties

- Neural networks have been shown to be a universal function approximator (it can approximate any function given enough data and model complexity)

- Convolutional nets – for image inputs, MLP: for other type of inputs, Time series: Recurrent NNs etc.,

- It is after all, a supervised learning technique. So data must be i.i.d. (Independent and identically distributed)

SIT796 Reinforcement Learning

# Deep Nets and Reinforcement Learning

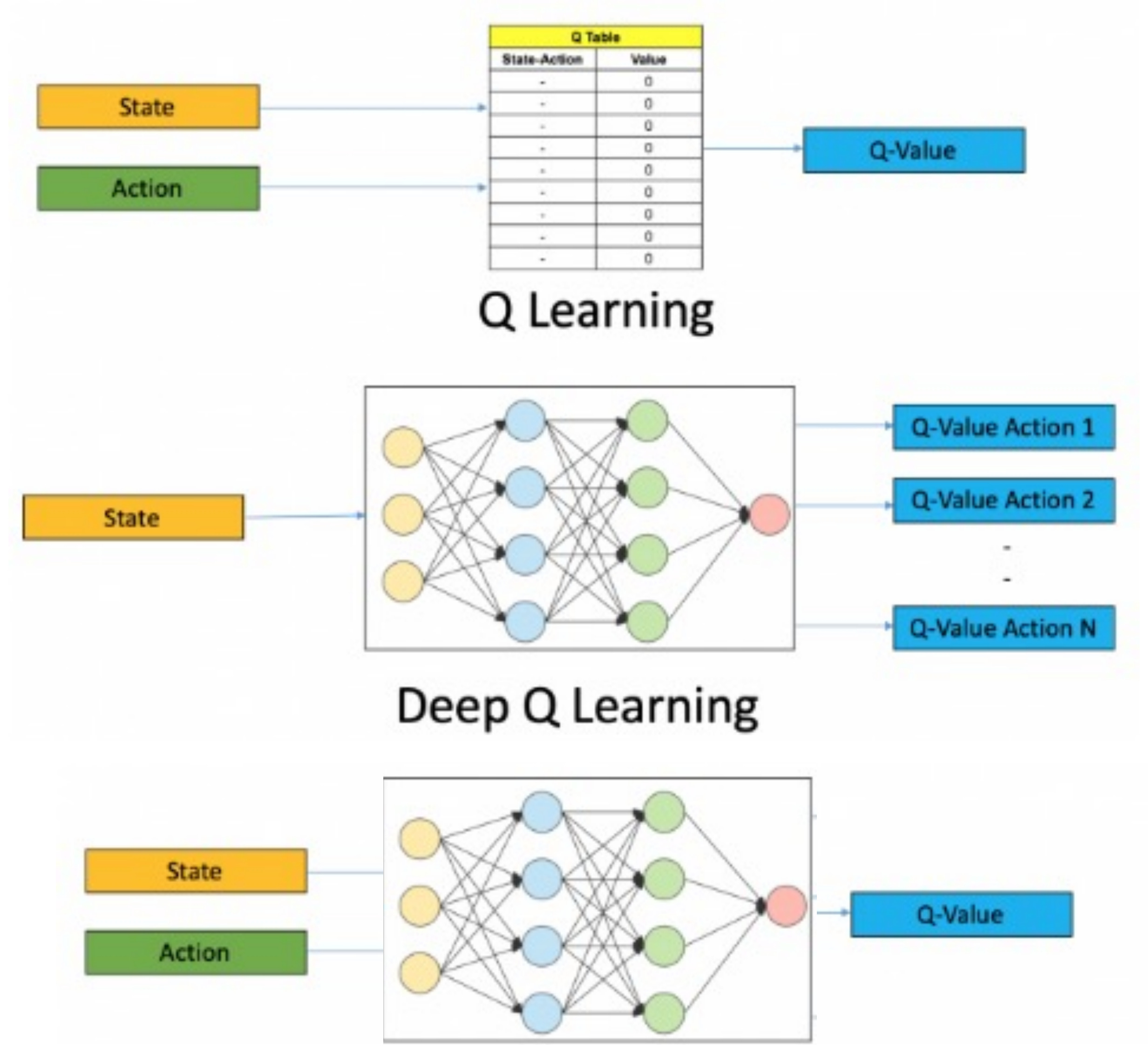
Presented by:
Thommen George Karimpanal
School of Information Technology

## What is Deep Reinforcement Learning?

- Deep reinforcement learning is standard reinforcement learning where a deep neural network is used to approximate either a policy or a value function
- Deep neural networks require lots of real/simulated interaction with the environment to learn
- Lots of trials/interactions is possible in simulated environments
- We can easily parallelise the trials/interaction in simulated environments
- We cannot do this with robotics (no simulations) because action execution takes time, accidents/failures are expensive and there are safety concerns

# Deep Q-Networks (DQN)

- It is common to use a function approximator Q(s, a; θ) to approximate the action-value function in Q-learning
- Deep Q-Networks is Q-learning with a deep neural network function approximator called the Q-network
- Discrete and finite set of actions A
- Example: Breakout has 3 actions – move left, move right, no movement
- Uses epsilon-greedy policy to select actions



## Q-Networks

- Core idea: We want the neural network to learn a non-linear hierarchy of features or feature representation that gives accurate Q-value estimates

- The neural network has a separate output unit for each possible action, which gives the Q-value estimate for that action given the input state (Can also be coded such that each state-action pair produces one Q value as output)

- The neural network is trained using mini-batch stochastic gradient updates and experience replay

  - Go though batches in the dataset

  - These batches make up for an epoch

  - Go through several epochs until convergence

## Experience Replay

- Experience is a sequence of states, actions, rewards and next states $e_t = (s_t, a_t, r_t, s_{t+1})$
- Store the agent's experiences at each time step $e_t = (s_t, a_t, r_t, s_{t+1})$ in a dataset $D = e_1, \ldots, e_n$ pooled over many episodes into a replay memory
- In practice, only store the last N experience tuples in the replay memory and sample when performing updates
- These experiences occur in sequence. So need to randomise them to make them i.i.d.
- It may require a lot of experience to obtain enough samples.

SIT796 Reinforcement Learning

## Deep Q-Learning

Presented by:
Thommen George Karimpanal
School of Information Technology

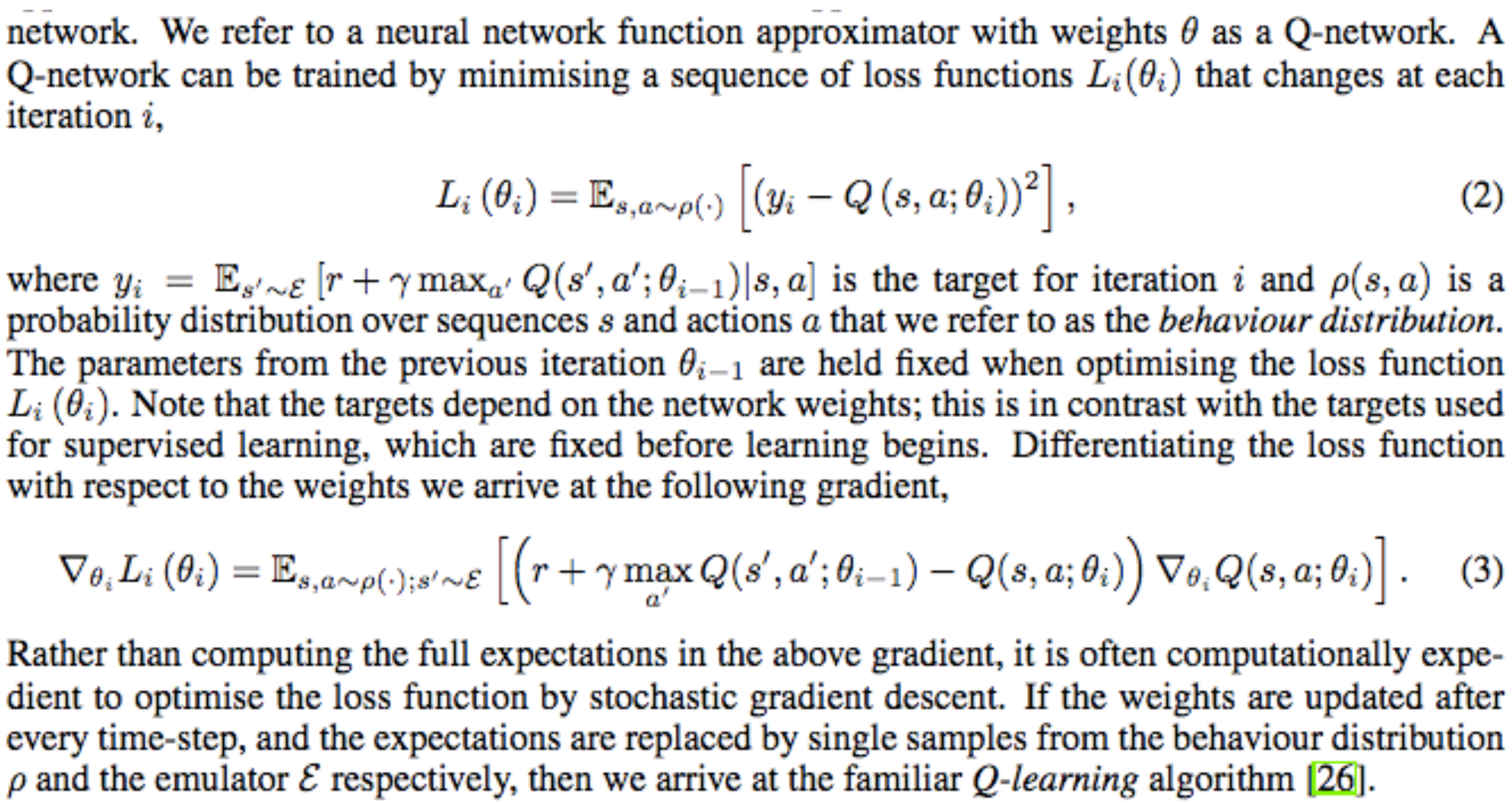
## Q-Network Training

- Sample random mini-batch of experience tuples uniformly at random from D (replay buffer)

- Similar to Q-learning update rule but:
  - Use mini-batch stochastic gradient updates
  - The gradient of the loss function for a given iteration with respect to the parameter $\theta_i$ is the difference between the target value and the actual value is multiplied by the gradient of the Q function approximator $Q(s, a; \theta)$ with respect to that specific parameter

- Use the gradient of the loss function to update the Q function approximator

## Loss Function Gradient Derivation



network. We refer to a neural network function approximator with weights $\theta$ as a Q-network. A Q-network can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration $i$,

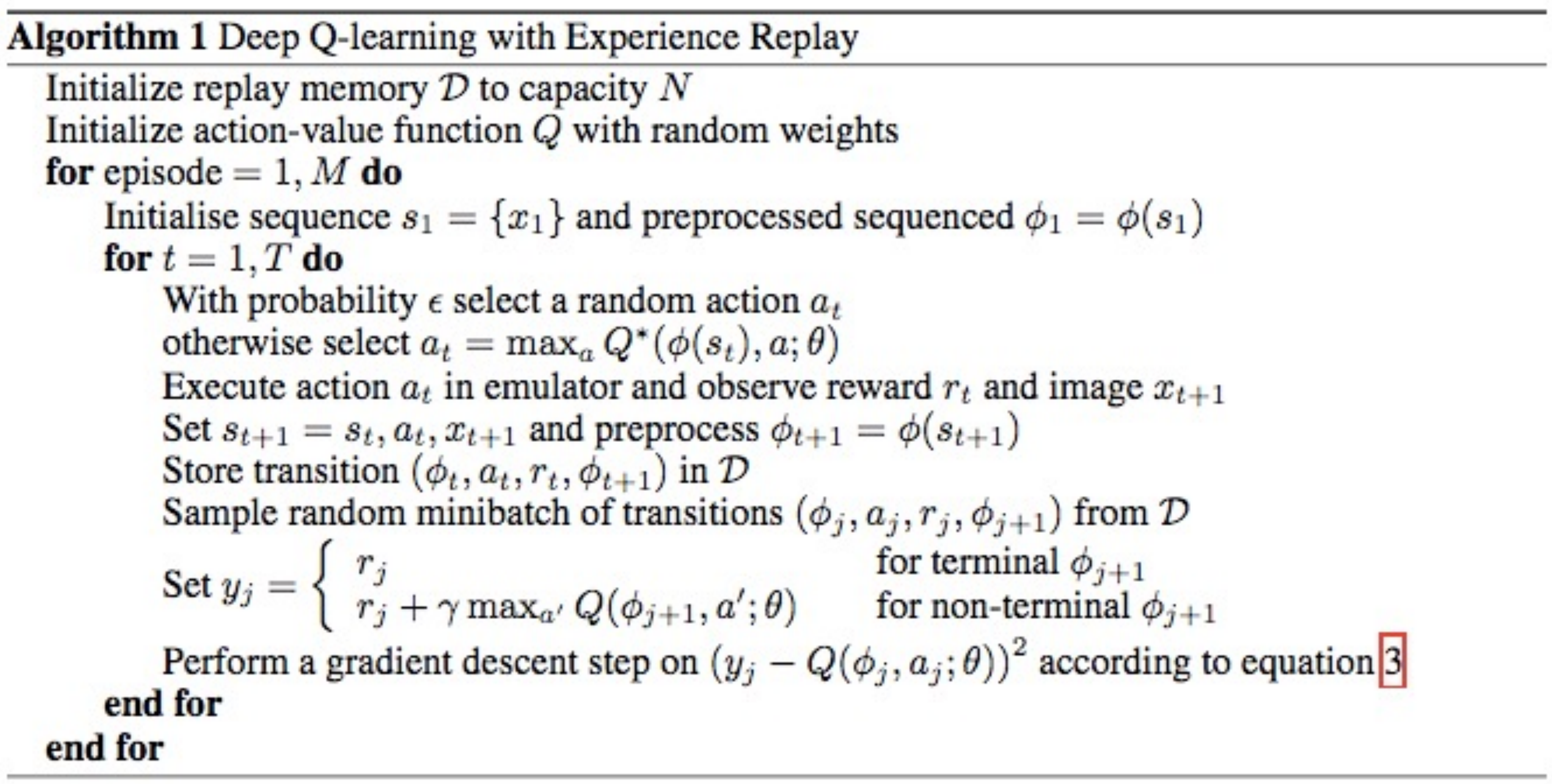
$$L_i\left(\theta_i\right) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[\left(y_i - Q\left(s,a;\theta_i\right)\right)^2\right], \tag{2}$$

where $y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a\right]$ is the target for iteration $i$ and $\rho(s,a)$ is a probability distribution over sequences $s$ and actions $a$ that we refer to as the *behaviour distribution*. The parameters from the previous iteration $\theta_{i-1}$ are held fixed when optimising the loss function $L_i\left(\theta_i\right)$. Note that the targets depend on the network weights; this is in contrast with the targets used for supervised learning, which are fixed before learning begins. Differentiating the loss function with respect to the weights we arrive at the following gradient,

$$\nabla_{\theta_i} L_i\left(\theta_i\right) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]. \tag{3}$$

Rather than computing the full expectations in the above gradient, it is often computationally expedient to optimise the loss function by stochastic gradient descent. If the weights are updated after every time-step, and the expectations are replaced by single samples from the behaviour distribution $\rho$ and the emulator $\mathcal{E}$ respectively, then we arrive at the familiar *Q-learning* algorithm [26].

## DQN Algorithm



**Algorithm 1** Deep Q-learning with Experience Replay

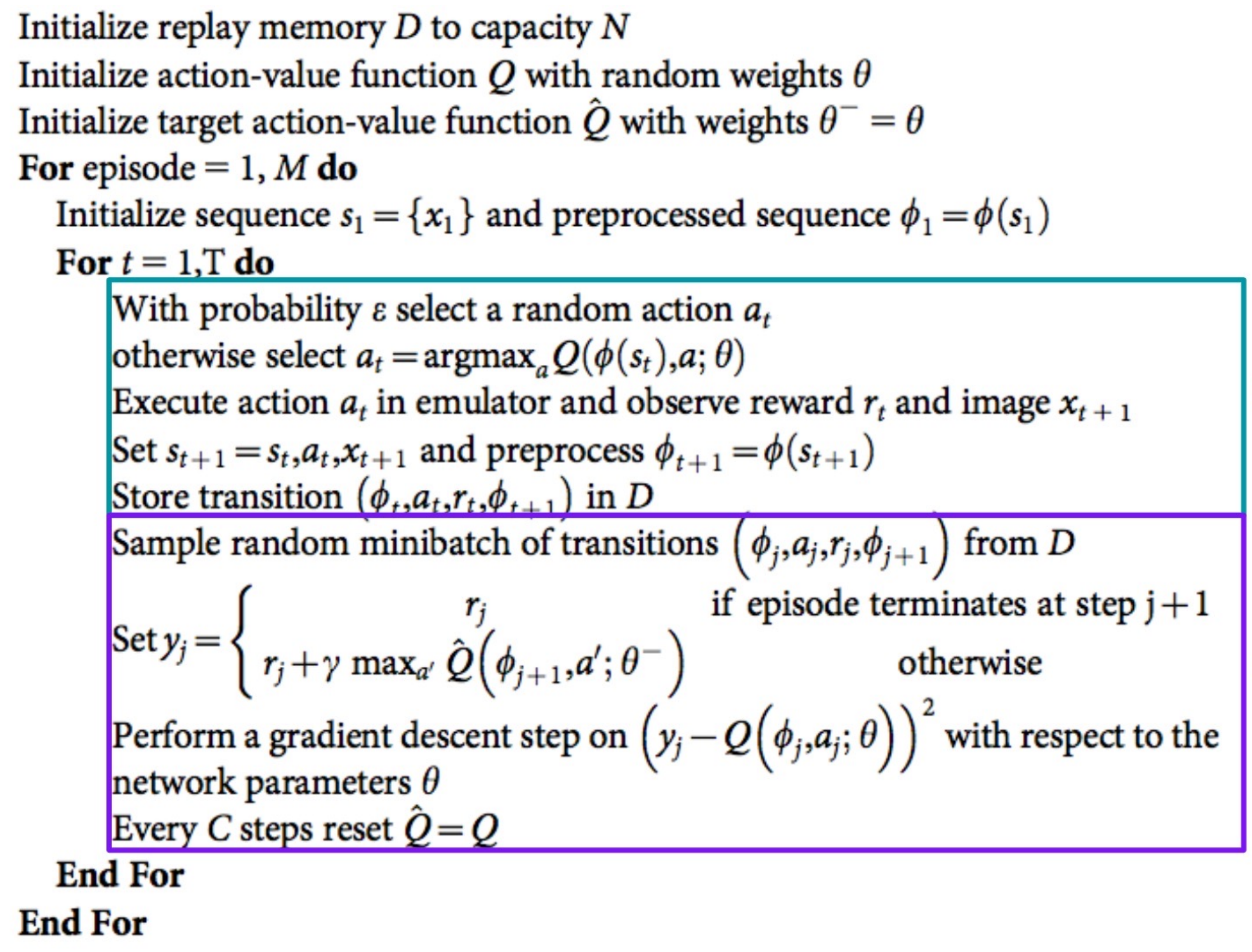
```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation $\boxed{3}$
    end for
end for
```

Regular Q learning update: $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

# DQN in Practice: Trick 1 – Experience Replay

- It was previously thought that the combination of simple online reinforcement learning algorithms with deep neural networks was fundamentally unstable
- The sequence of observed data (states) encountered by an online reinforcement learning agent is non-stationary and online updates are strongly correlated
- The technique of DQN is stable because it stores  the agent’s data in experience replay memory so that it can be randomly sampled from different time-steps
- Aggregating over memory reduces non-stationarity and decorrelates updates but limits methods to off-policy reinforcement learning algorithms
- Experience replay updates use more memory and computation per real interaction than online updates, and require off-policy learning algorithms that can update from data generated by an older policy

# DQN in Practice: Trick 2



```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode $= 1, M$ do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For $t = 1$,T do
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
```

Sampling

Training

Use two networks: a policy network and target network

Freeze the target network and update it only after C steps

Tends to stabilize learning

Source: https://huggingface.co/learn/deep-rl-course/unit3/deep-q-algorithm

## DQN in Practice: Trick 3

Clip rewards to some fixed range [-1,1]

Not so important, but it helps

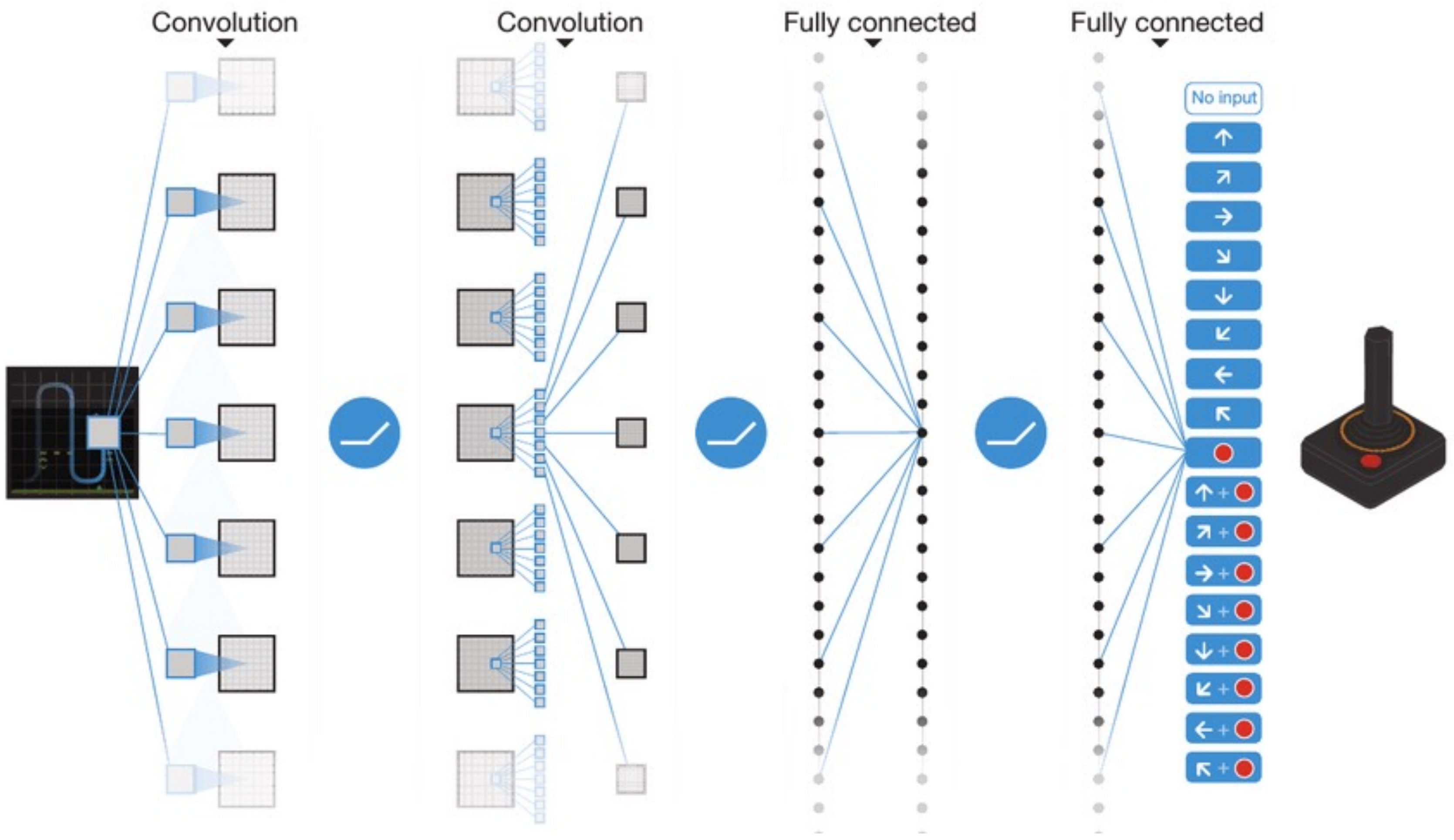
Using these 3 “tricks”, DQN training became stable

No convergence guarantees!

Converges to a local optimum that is not far from the global optimum

# DQN Example: Playing Atari Games

- The input is the 8x8 image region about the current position of the snake.
- Q-network with 3 convolutional layers of size
  - 32x8x8;stride 4
  - 64x4x4;stride 4
  - 64x3x3;stride 2
- The final two layers are fully connected layers with 512



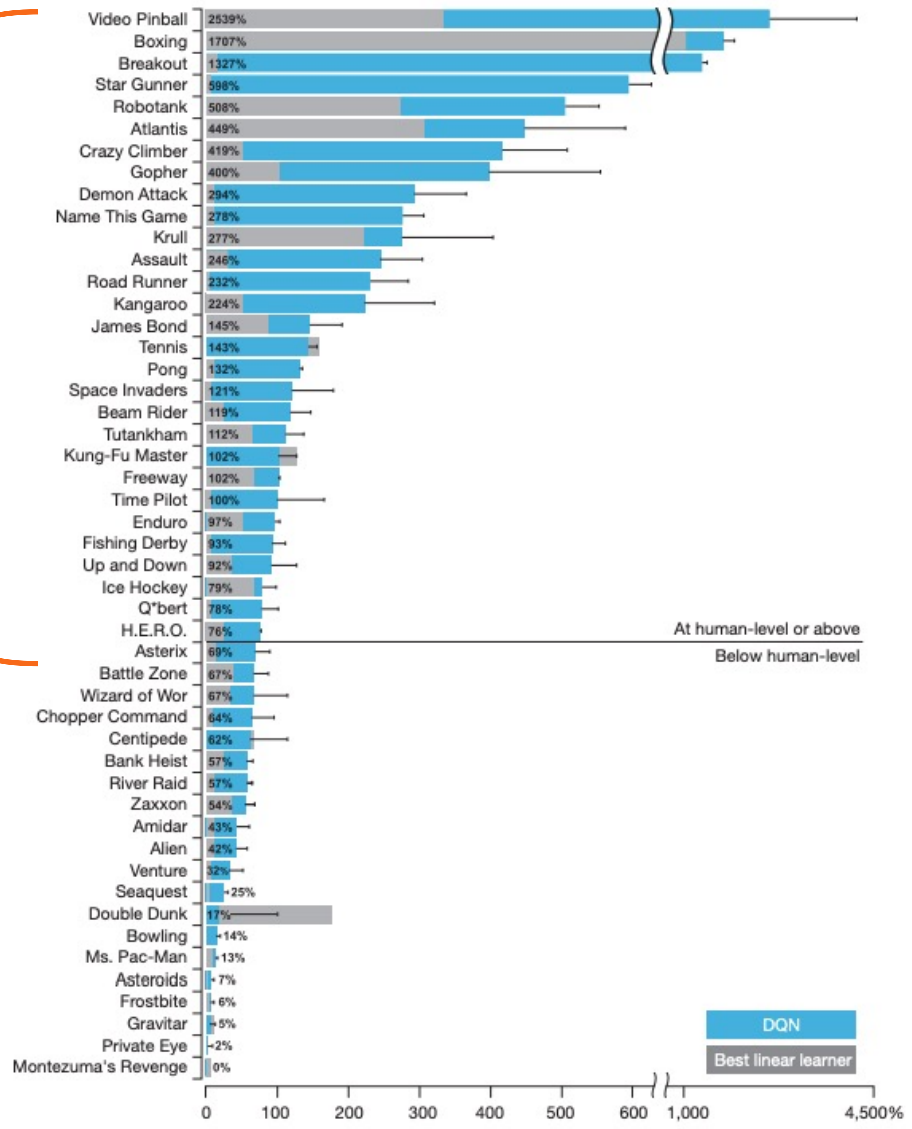
Mnih et al. (2015). Human-level control through deep reinforcement learning

# DQN Example: Playing Atari Games

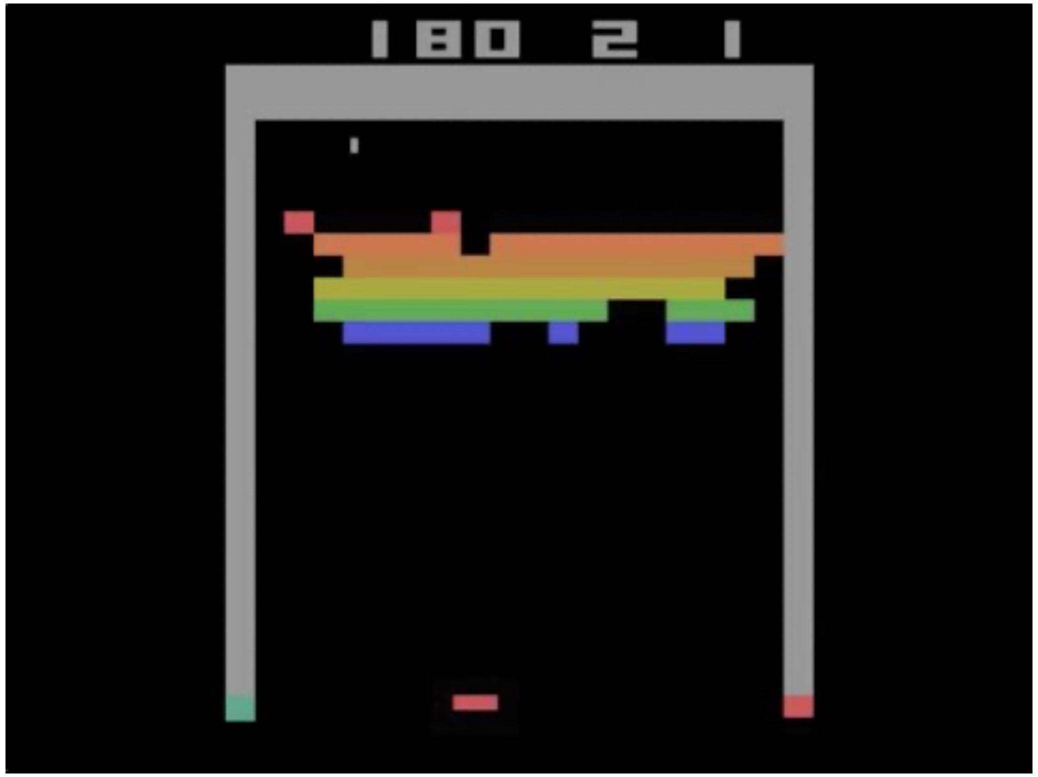
>= Humans

Poor sample efficiency

Each game learned from scratch



## DQN Example: Playing Atari Games



## DQN Example: Playing Atari Games

Better than human performance

May need excessive data ~ $10^9$ samples (if 1s per sample, then >31 years!)

Still needs actions to be discrete – not feasible in many cases

SIT796 Reinforcement Learning

## Dealing with continuous actions

Presented by:
Thommen George Karimpanal
School of Information Technology

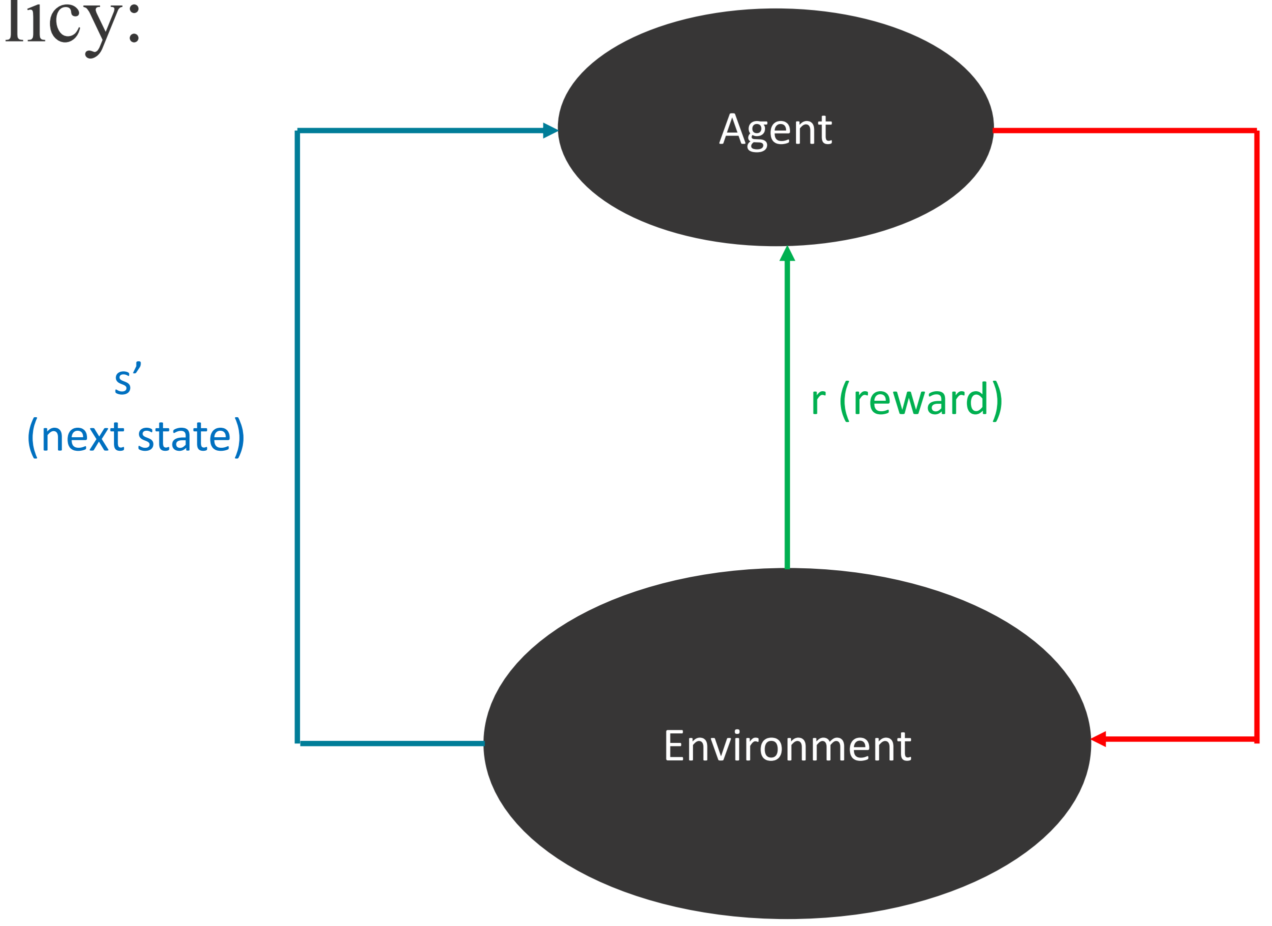
# RL algorithm types

Three approaches to find RL policy:

1. Value-based methods (everything covered so far, incl. DQN)

2. Directly obtain policy (policy gradient methods)

3. Actor-critic methods (combination of 1. and 2.)



# Policy Gradient Methods

- Several kinds:
  - Finite Difference Policy Gradient
  - Monte Carlo Policy Gradient
  - Actor-Critic Policy Gradient

- Directly parameterize and learn the policy

$$\pi(a|s) = f_\theta(\boxed{\phi(s,a)})$$

Feature vector of state-action pair

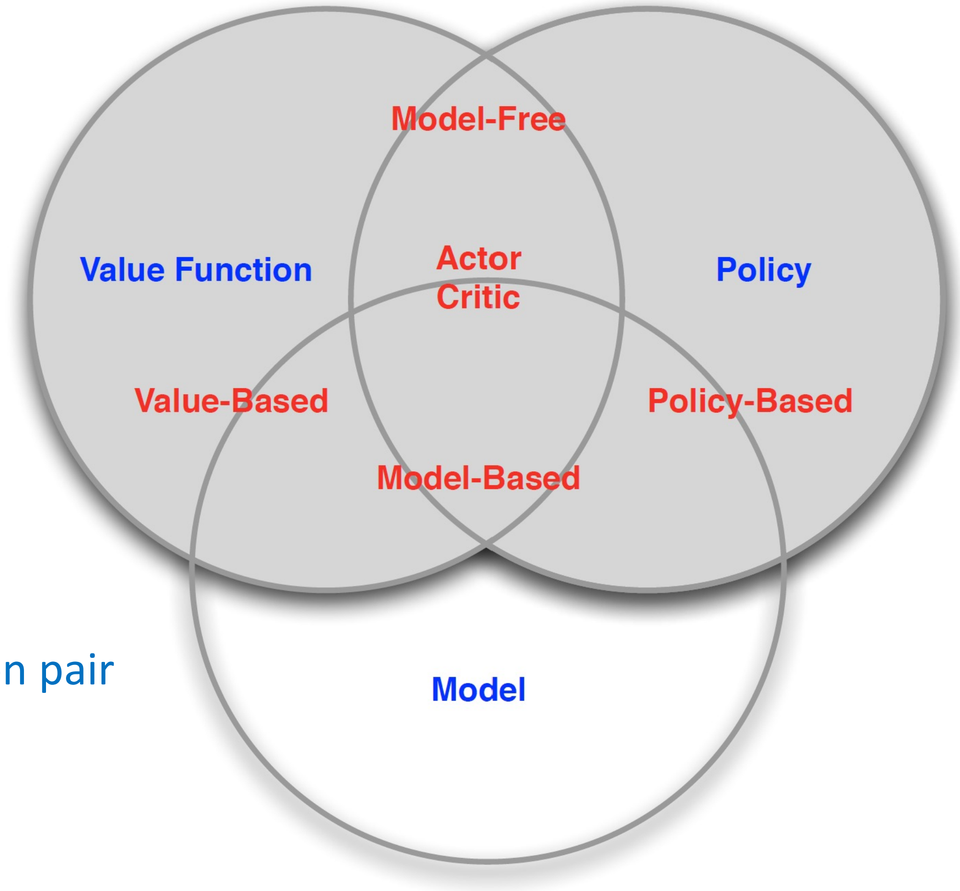
- Can have several forms such as:

$$\pi(a|s) \propto \exp(\underline{\theta^\top \phi(s,a)})$$

No need to be related to the value function!

Why not directly define $\pi(a|s) \propto \exp(\boxed{\lambda} v(s,a))$?

Temperature of soft-max



## What are Policy Gradient Methods?

- Before: Learn the values of actions and then select actions based on their estimated action-values. The policy was generated directly from the value function
- We want to learn a parameterised policy that can select actions without consulting a value function. The parameters of the policy are called policy weights
- A value function may be used to learn the policy weights but this is not required for action selection
- Policy gradient methods are methods for learning the policy weights using the gradient of some performance measure with respect to the policy weights
- Policy gradient methods seek to maximise performance and so the policy weights are updated using gradient ascent

## Policy-based Reinforcement Learning

- Search directly for the optimal policy $\pi^*$
- Can use any parametric supervised machine learning model to learn policies $\pi(a \mid s; \boldsymbol{\theta})$ where $\boldsymbol{\theta}$ represents the learned parameters
- Recall that the optimal policy is the policy that achieves maximum future return

SIT796 Reinforcement Learning

## Policy Approximation

Presented by:
Thommen George Karimpanal
School of Information Technology

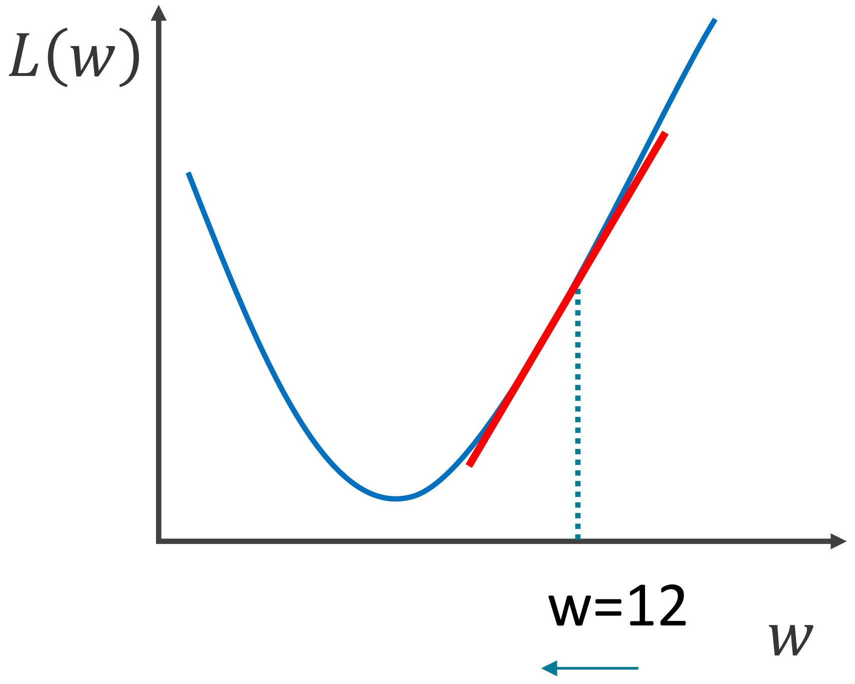
# Gradient Descent

- Optimizer for functions.
- Guaranteed to find optimum for convex functions.
    - Non-convex = find *local* optimum.

- Works for multi-variate functions.
    - Need to compute matrix of *partial derivatives (“Jacobian”)*

1. Start with a random value of w (e.g. w = 12)
2. Compute the gradient (derivative) of L(w) at point w = 12. (e.g. dL/dw = 6)
3. Recompute w as:

$$w = w - \lambda(dL(w) / dw)$$



## Policy Gradient: General Idea

Directly learn policy from  objective function:

$$J(\theta) = \mathbb{E}_{\pi}[r(\tau)]$$

τ: trajectory of (s,a,r) pairs

Directly maximize $J(\theta)$ : obtain the update equation:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t)$$

Policy gradient theorem: It can be shown that:

$$\nabla J(\theta_t) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta_t)$$

Refer to textbook for derivation. It forms the basis for the policy gradient family of methods

## Policy Approximation

- Basic assumption: policy is differentiable w.r.t. θ . Eg:

$$\pi(a|s,\boldsymbol{\theta}) \doteq \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}}$$

soft-max in action preferences

- Action preferences could also be linear: $h(s,a,\boldsymbol{\theta}) = \boldsymbol{\theta}^\top \mathbf{x}(s,a)$

- For some problems, it is simpler to learn the policy directly rather than learning the value functions, and extracting the policy from it later.

## Notation

- The policy is $\pi(a \mid s, \boldsymbol{\theta})$, which represents the probability that action a is taken in state s with policy weight vector $\boldsymbol{\theta}$

- If using learned value functions, the value function's weight vector is $\mathbf{w}$

SIT796 Reinforcement Learning

## REINFORCE

Presented by:
Thommen George Karimpanal
School of Information Technology

DEAKIN
UNIVERSITY

## REINFORCE

Using the policy gradient theorem, the update rule can be modified to be:

$$\theta_{t+1} = \theta_t + \alpha\gamma^t G_t \nabla ln\pi(a_t|s_t, \theta_t)$$

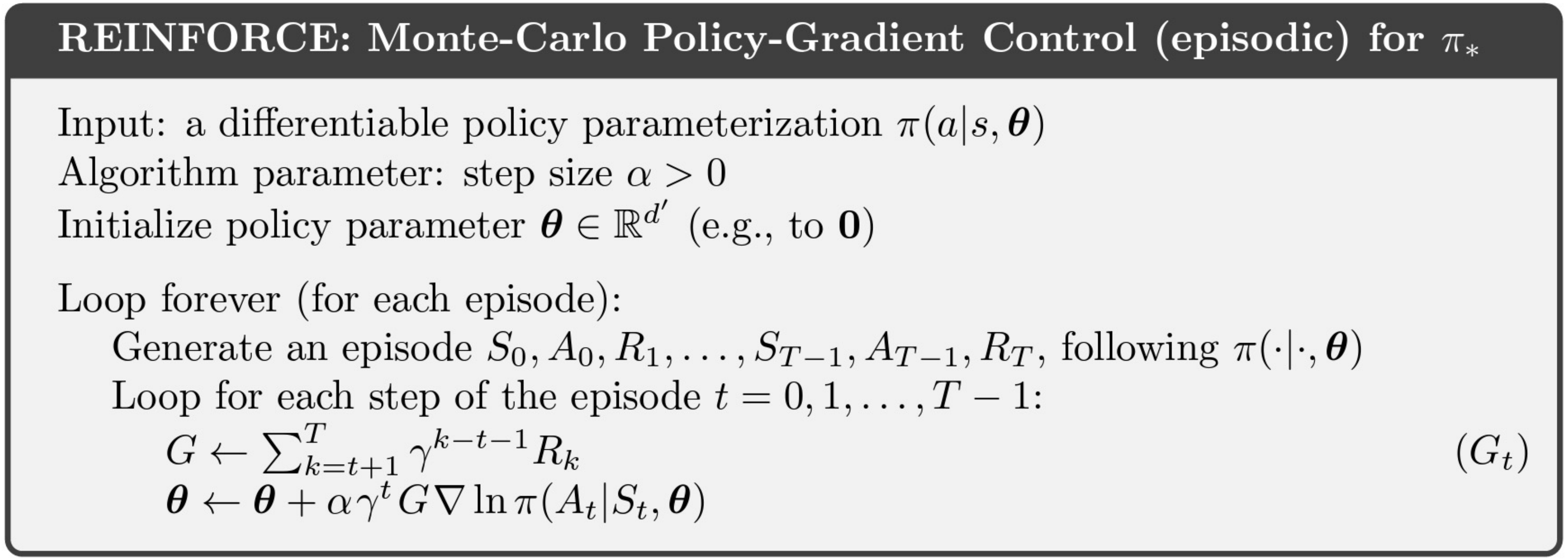
where

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

Iteratively updating $\theta$ with the above update rule will lead to the optimal policy $\pi^*$

## REINFORCE Properties and Algorithm

- On-policy method based on SGD
- Uses the complete return from time t, which includes all future rewards until the end of the episode
- REINFORCE is thus a Monte Carlo algorithm and is only well-defined for the episodic case with all updates made in retrospect after the episode is completed



**REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for $\pi_*$**

```
Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
    Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$                    $(G_t)$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$
```

# Actor-Critic Methods

Actor-critic methods are a fusion of policy gradient and value-based methods

Actor: deals with the policy (policy gradient-based) $a_t \sim \pi_\theta$

Critic: evaluates the actor's action (value-based) $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$

$\theta_t$ is updated to minimise $\delta_t$ with the update rule: $\theta_t \leftarrow \theta_t + \beta \delta_t$

where $\beta$ is a positive step size hyperparameter

Both actor and critic are updated using $\delta_t$ to determine the optimal policy $\pi^*$

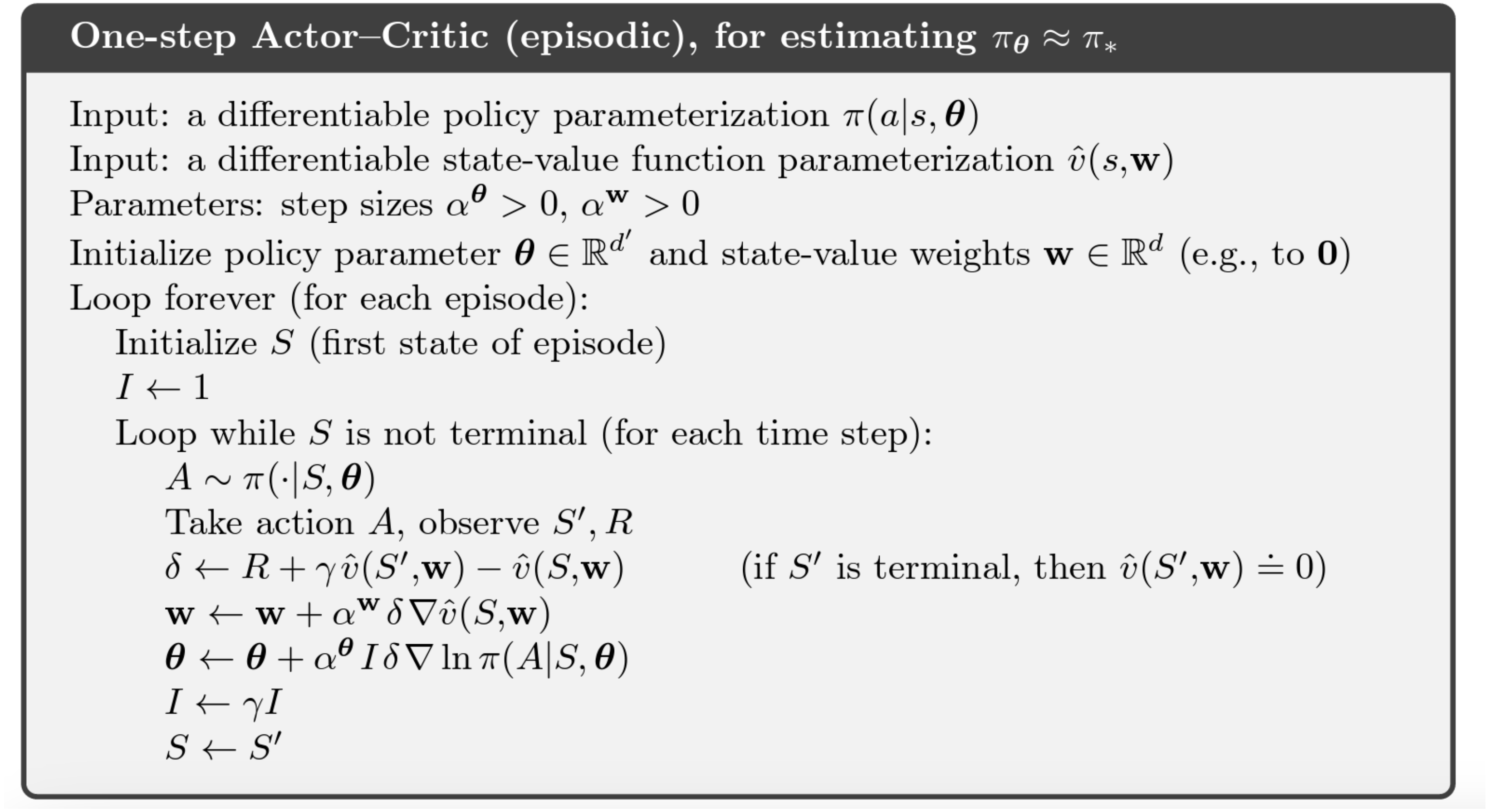
Modern approaches include several variants of the actor-critic algorithm.

# One-step Actor-Critic Update Rules

- On-policy method
- The state-value function update rule is the TD(0) update rule
- The policy function update rule is shown below.

$$\begin{aligned}
\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha\Big(G_{t:t+1} - \hat{v}(S_t,\mathbf{w})\Big)\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}\\
&= \boldsymbol{\theta}_t + \alpha\Big(R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})\Big)\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}\\
&= \boldsymbol{\theta}_t + \alpha\delta_t\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}.
\end{aligned}$$

# One-step Actor-Critic Algorithm



**One-step Actor–Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$**

```
Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
    Initialize $S$ (first state of episode)
    $I \leftarrow 1$
    Loop while $S$ is not terminal (for each time step):
        $A \sim \pi(\cdot|S, \boldsymbol{\theta})$
        Take action $A$, observe $S'$, $R$
        $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$        (if $S'$ is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S,\mathbf{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S, \boldsymbol{\theta})$
        $I \leftarrow \gamma I$
        $S \leftarrow S'$
```

## Modern Algorithms

- Trust Region Policy Optimisation
- Proximal Policy Optimisation (Simplification of TRPO)
- Soft Actor-Critic (SAC)
- DDPG (Deep deterministic policy gradient)
- and many more!

## Readings

This lecture focused on introducing Deep Learning for RL.

- Future topics will expand on this topic by looking at particular methods in Deep RL.
- Ensure you understand what was discussed here before doing the following topics

For more detailed information see:

- https://www.ics.uci.edu/~dechter/courses/ics-295/fall-2019/texts/An_Introduction_to_Deep_Reinforcement_Learning.pdf
- https://rail.eecs.berkeley.edu/deeprlcourse/

- Other Readings:
  - Playing Atari with Deep Reinforcement Learning (https://arxiv.org/abs/1312.5602)