W12: An Introduction to Deep Reinforcement Learning

Complexity-in-Action Research Lab
Macquarie University
Fred Amouzgar
2021S1

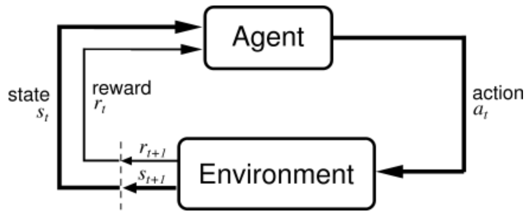
AGENDA

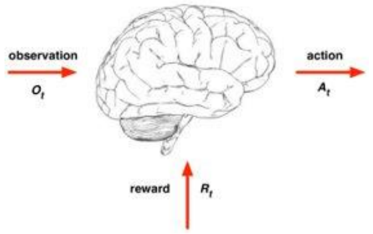
- **Classical Reinforcement Learning: A Review**
- **Introduction to Approximate Reinforcement Learning**
  1. The problem of large and continuous state spaces
  2. Approximate SARSA and Q-learning
- **Deep Reinforcement Learning**
  1. DQN (Deep Q-Network)
  2. Enhanced DQNs
  3. Policy Gradient method and the REINFORCE algorithm
  4. An Application of REINFORCE in text generation
  5. Actor-Critic algorithms
  6. Advanced Deep RL
- **Conclusion**

Classical Reinforcement Learning: A Review

REINFORCEMENT LEARNING

- RL is a computational approach to learning from interaction.
- It provides the formalism for intelligent and adaptive behaviour.
- Challenges:
  1. Actions have long-term consequences.
  2. Delayed Rewards (the credit assignment problem).
  3. It might be better to sacrifice immediate rewards to gain more long-term rewards.
  4. Exploration vs. Exploitation Dilemma
- Three main elements of RL algorithms:
  1. Reward
  2. State
  3. Action
- Agent's goal is to maximize the cumulative reward or the return (G).





TRAJECTORY AND RETURN

- Multi-Armed Bandit: A simple environment to demonstrate exploration-eploitation dilemma, and test a simple, but fundamental idea of learning.

$$\textit{NewEstimate} \leftarrow \textit{OldEstimate} + \textit{StepSize}\Big[\textit{Target} - \textit{OldEstimate}\Big]$$

- Trajectory: The life of an agent can be formally captured by a sequence of states, actions and rewards, $s_0, a_0, r_1, s_1, a_1, \ldots, s_T$ (terminal state).

- Return: The return G_t is the total discounted reward from time-step t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma\left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots\right) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

MARKOV DECISION PROCESS (MDP)

- It introduces assumptions to deal with RL at the mathematical level. Assumptions which we suspend in practice.
  - Markov Property: the future is independent of the past given the present (all you need to know is here, and once the current state is known, the history can be discarded). Thus, a Markov state is defined as:

$$\mathbf{P}[S_{t+1}|S_t] = \mathbf{P}[S_{t+1}|S_1, S_2, ..., S_t]$$

  - Environment is fully observable

- Almost all RL problems can be formalized as MDP.
- A Markov Decision Process is a tuple <S, A, P, R, γ>
  - S is a finite set of states.
  - A is a finite set of actions.
  - P is a state transition probability matrix (the dynamic of the MDP, usually unknown)

$$P[S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a]$$

  - R is a reward function $r(s, a, s') = E[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s']$
  - γ is a discount factor $\gamma \in [0, 1]$

POLICY AND VALUE FUNCTIONS

- **Policy**: A policy (π) is a function that maps the current state onto a set of probabilities for taking each action (also a distribution over *a* given *s*).

$$\pi : s \rightarrow p(a) \text{ also written as } \pi(a|s)$$

$$\sum_{a_t \in A(s_t)} \pi(a_t|s_t) = 1 \ , \ \pi(a|s) \geq 0$$

- The solution to an MDP is a policy that associates a decision with every state that the agent might reach.
- It's evident that the deterministic policy is a special case of a stochastic policy when the probability of one action is 1 and the rest is 0.

- **Value Functions**: If we provide a policy, value functions can predict reward in the future following that policy. Thus, the policy can be derived from them or they can guide the policy. There're two types of value functions:
  1. State-value function
  2. Action-value function

VALUE FUNCTIONS DEFINITIONS

- The state-value function V of an MDP is the expected return starting from state s, and then following policy π.

$$v_\pi(s) = E_\pi[G_t | S_t = s]$$

- The action-value function Q is the expected return starting from state s, taking action a, and then following policy π.

$$Q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a]$$

MACQUARIE University

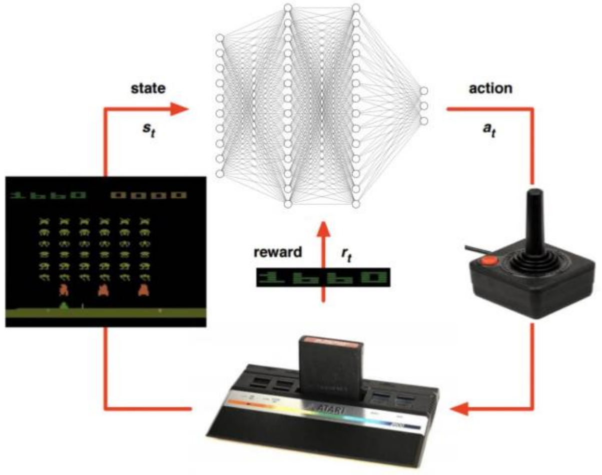
Introduction to Approximate Reinforcemet Learning

THE PROBLEM OF LARGE AND CONTINUOUS STATE SPACES

- Model-free RL algorithms such as SARSA and Q-learning can find optimal policies for arbitrary environments, **IF the value functions and policies can be represented in tables.**
- Most problems are too large and complex (e.g., chess, Go, learning from pixels, etc)
- It's postulated that Go alone has $2.08 * 10^{170}$ legel states. (There are 10^{97} particles in the visible universe!)
- One possible solution is using Function Approximation (ML methods).
- Will RL work with function approximators?

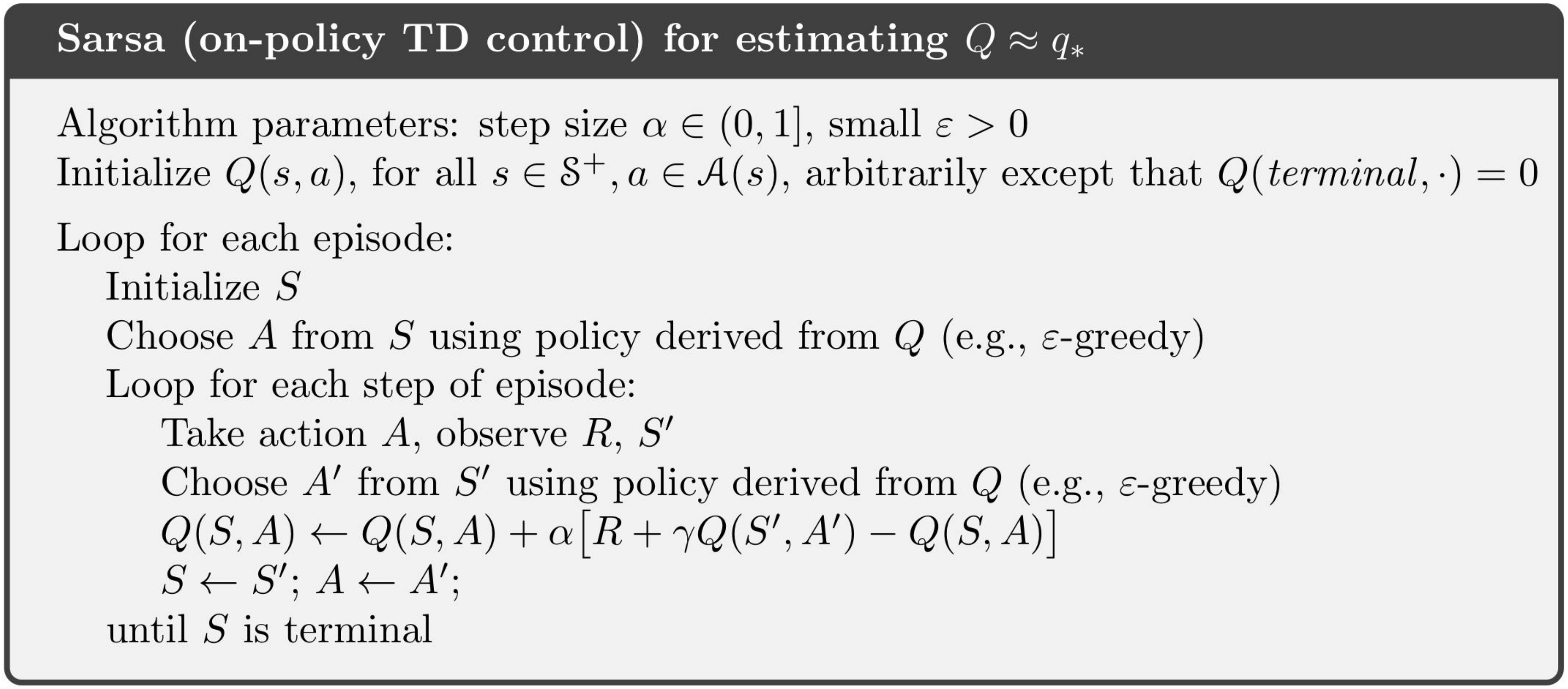
FUNCTION APPROXIMATION

- We can represent the state-value (V) or the action-value (Q) function by a parameterized function approximator with parameter θ.
- The notation changes from Q(s,a) to <u>Q(s,a,θ)</u> and from V(s) to <u>V(s,θ)</u>. θ can be parameterized by a deep neural network or a simple linear weighting of features.
- Function approximation such as neural network addresses two of tabular methods' shortcomings:
  1. <u>Curse of dimensionality</u>: the table grows exponentially with number of states and actions
  2. <u>Lack of generalization</u>: the ability to deal with similar states, similarly
- All the recent success stories in RL comes from the power of approximators.



FUNCTION APPROXIMATION – TABULAR SARSA

- Remember the Tabular SARSA algorithm
- For function approximation we turn the TD-error into the loss function.



Sarsa (on-policy TD control) for estimating $Q \approx q_*$

```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Loop 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
```

FUNCTION APPROXIMATION – TD LOSS

- Consider the following objective function (MSE), based on the bellman equation (the TD error)

$$L(w) = E\left[\left(R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, w) - \hat{q}(S_t, A_t, w)\right)^2\right]$$

- The target here, $R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, w)$, depends on the parameter, but if we ignore that dependance when taking the derivative, then we get semi-gradient SARSA update.

$$\Delta w_t = \alpha\left(R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, w_t) - \hat{q}(S_t, A_t, w_t)\right)\frac{\partial\hat{q}(S_t, A_t, w_t)}{\partial w_t}$$

- We are now ready to turn the tabular SARSA to an approximate method.

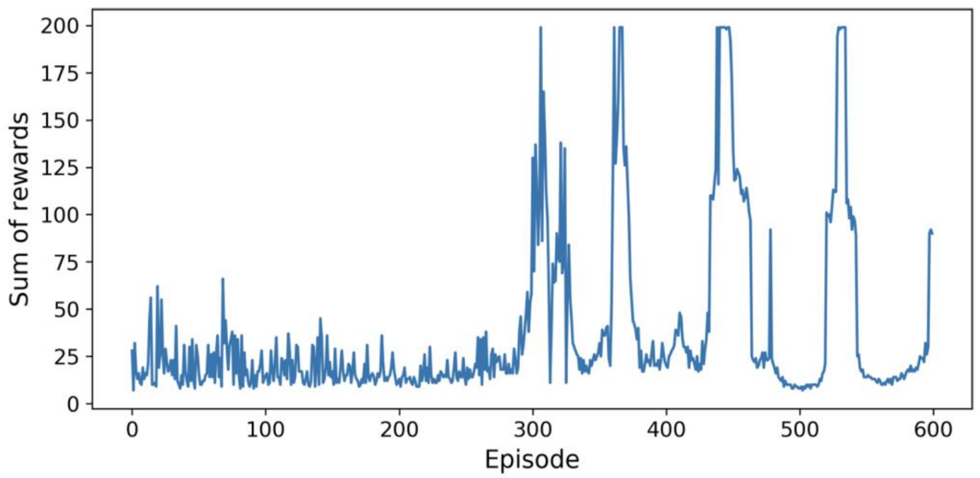
FUNCTION APPROXIMATION – APPROXIMATE SARSA

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

```
Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
    $S, A \leftarrow$ initial state and action of episode (e.g., $\varepsilon$-greedy)
    Loop for each step of episode:
        Take action $A$, observe $R, S'$
        If $S'$ is terminal:
            $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
            Go to next episode
        Choose $A'$ as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., $\varepsilon$-greedy)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
        $S \leftarrow S'$
        $A \leftarrow A'$
```

FUNCTION APPROXIMATION - PROBLEMS

- Function approximation's idea is that: **Neural Net + RL = Intelligence**!
- **Does it really work?** Sometimes, but with deeper neural networks and more complex environments, it may diverge or forget after learning (Catasrophic Forgetting), e.g, NN + Q-learning.
- **The Deadly Triad**: when all these three elements are present, the agent's learning will be unstable:
  1. **Function Approximation**: Neural networks, Linear models.
  2. **Bootstrapping**: Using TD-based methods such as SARSA and Q-learning.
  3. **Off-policy Training**: Off-policy methods in general like Q-learning.



FUNCTION APPROXIMATION – APPROXIMATE Q-LEARNING

- We saw that Q-learning had a lot of strengths in its tabular form.
- Unfortunately, when we apply approximation methods on Q-learning, it diverges.
- Reasons for its divergance:
  1. The correlations in the sequence of observations
  2. Small updates to Q may significantly change the policy and therefore change the data distribution
  3. The correlations between the action value (Q) and the target values $(r + \gamma \max Q(s',a'))$
- The DQN (Deep Q-Network) method solved these problems by augmenting approximate Q-learning with two ideas:
  1. Experience Replay Buffer
  2. Target network

MACQUARIE University

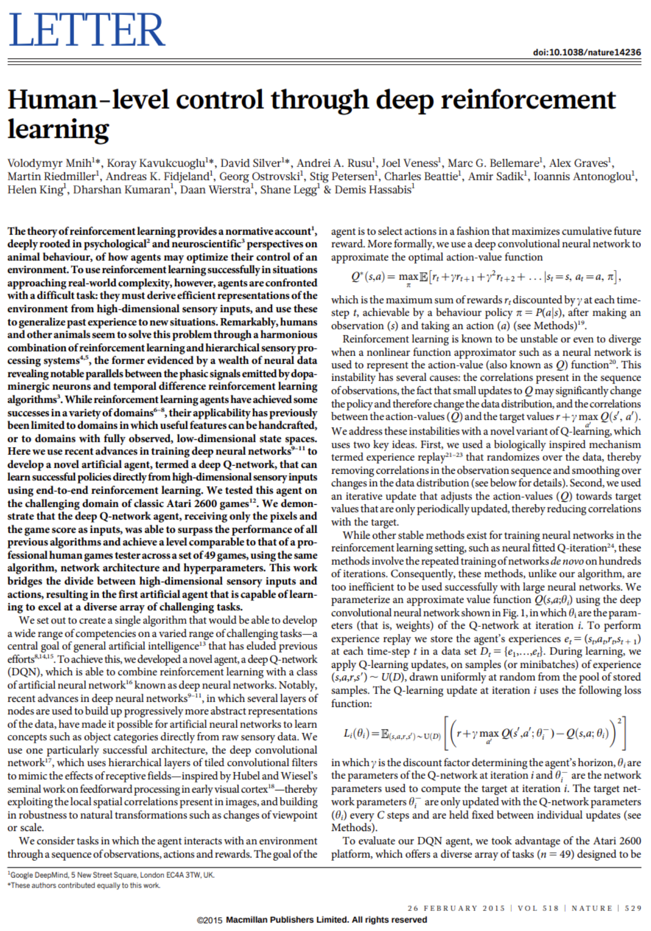
Deep Reinforcement Learning

MACQUARIE
University

Deep Reinforcement Learning Events that Shook the World!

DEEP RL – DQN (2013-2015)

We can say that the deep RL era started somewhere between 2013-2015.



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 central goal of general artificial intelligence[13] that has eluded previous efforts[8,14,15]. To achieve this, we developed a novel agent, a deep Q-network (DQN), which is able to combine reinforcement learning with a class of artificial neural network[16] known as deep neural networks. Notably, recent advances in deep neural networks[9–11], in which several layers of nodes are used to build up progressively more abstract representations of the data, have made it possible for artificial neural networks to learn concepts such as object categories directly from raw sensory data. We use one particularly successful architecture, the deep convolutional network[17], which uses hierarchical layers of tiled convolutional filters to mimic the effects of receptive fields—inspired by Hubel and Wiesel's seminal work on feedforward processing in early visual cortex[18]—thereby exploiting the local spatial correlations present in images, and building in robustness to natural transformations such as changes of viewpoint or scale.

We consider tasks in which the agent interacts with an environment through a sequence of observations, actions and rewards. The goal of the 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

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

which is the maximum sum of rewards r_t discounted by γ 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.

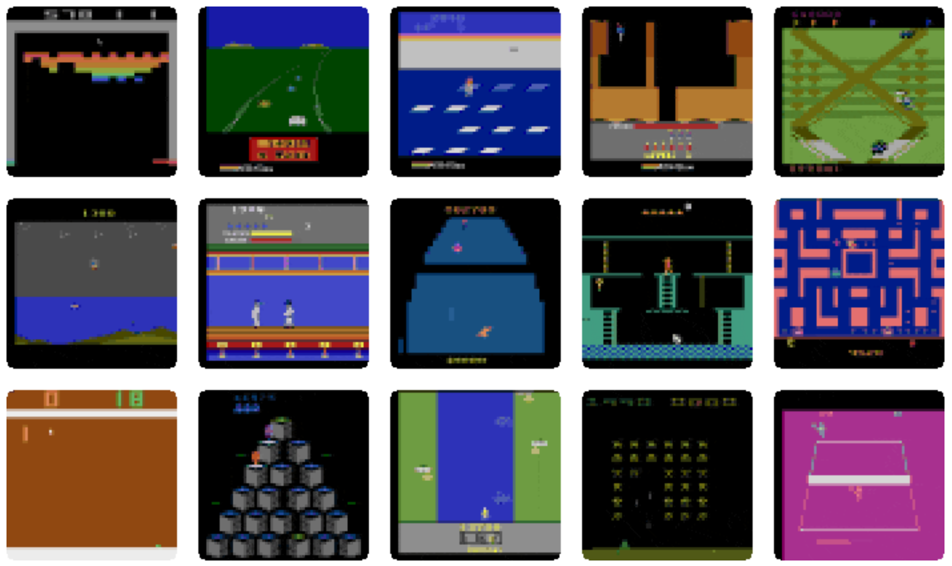
While other stable methods exist for training neural networks in the reinforcement learning setting, such as neural fitted Q-iteration[24], these methods involve the repeated training of networks *de novo* on hundreds of iterations. Consequently, these methods, unlike our algorithm, are too inefficient 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 θ_i are the parameters (that is, weights) of the Q-network at iteration i. To perform experience replay we store the agent's experiences $e_t = (s_t,a_t,r_t,s_{t+1})$ at each time-step t in a data set $D_t = \{e_1,\ldots,e_t\}$. During learning, we apply Q-learning updates, on samples (or minibatches) of experience $(s,a,r,s') \sim U(D)$, drawn uniformly at random from the pool of stored samples. The Q-learning update at iteration i uses the following loss function:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim \mathrm{U}(D)}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]$$

in which γ is the discount factor determining the agent's horizon, θ_i are the parameters of the Q-network at iteration i and θ_i^- are the network parameters used to compute the target at iteration i. The target network parameters θ_i^- are only updated with the Q-network parameters (θ_i) every C steps and are held fixed between individual updates (see Methods).

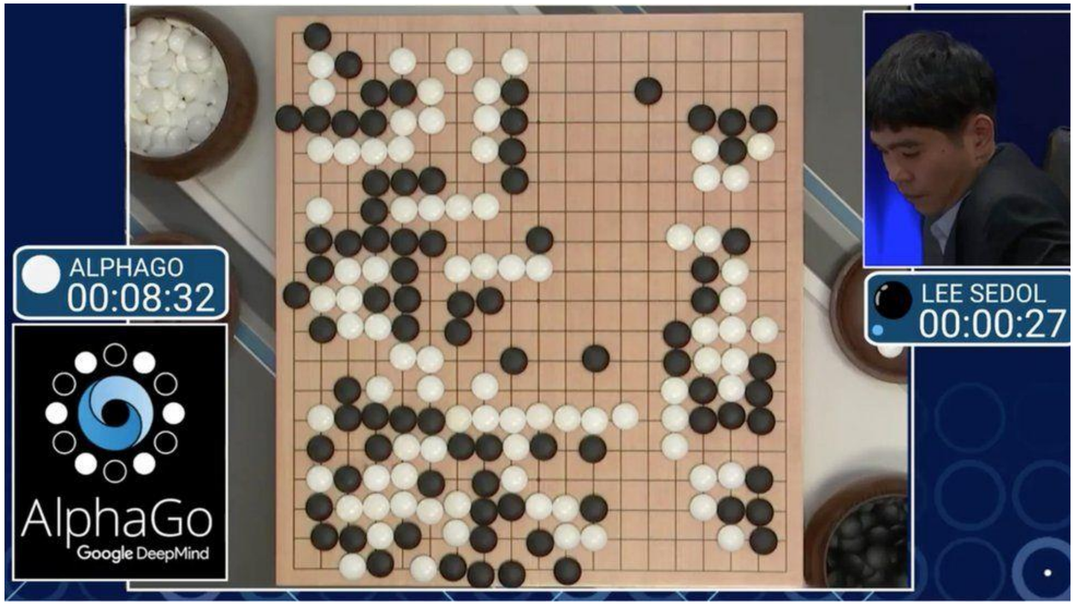
To evaluate our DQN agent, we took advantage of the Atari 2600 platform, which offers a diverse array of tasks ($n = 49$) designed to be

[1]Google DeepMind, 5 New Street Square, London EC4A 3TW, UK.
*These authors contributed equally to this work.



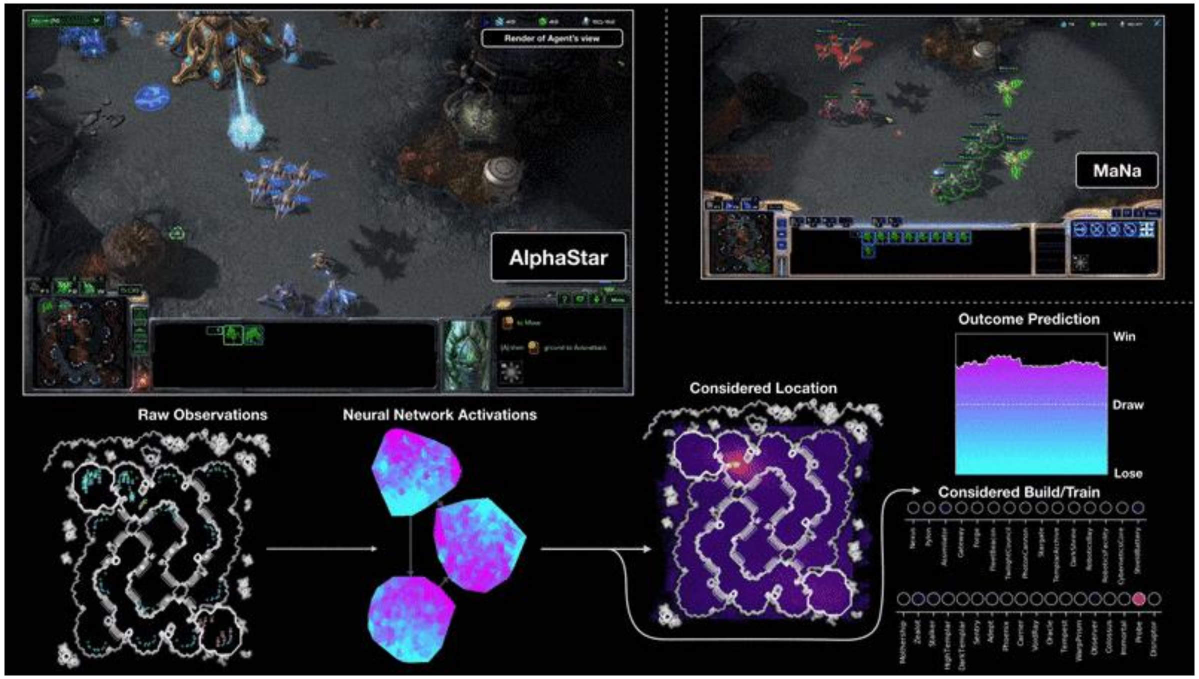
DEEP RL – ALPHA GO & ALPHA GO ZERO (2016-2017)

- The Alpha Go full documentary.
- For years it was believed that the sheer state space size of the game of Go makes it inaccessible to machines and it requires human "intuition".
- They underestimated neural networks and reinforcement learning!
- Alpha Go Zero = Deep Learning + RL (in MCTS + Self-play)



DEEP RL – ALPHA GO & ALPHA GO ZERO (2019)

- The Alpha Star documentary.
- The magic was only in a very deep neural network, an advanced actor-critic algorithm (IMAPALA), and astronomical amounts of computation power!



MACQUARIE University

DQN

DQN – ADDITIONS

What is the Experience Replay Buffer?

- Experience Replay Buffer is like a dataset in supervised learning.
- The difference is that the agent collects this as it explores the environment.
- In every step the agent adds its "experience", the tuple of (state, action, new_state, reward, done), to this dataset.
- In every D timesteps (e.g., 4), we sample a batch from it and train the Q function's neural network with it, like a normal regression.

What is a target network?

- A target network is a copy of the main network with delay.
- It is mainly used in the calculation of the target.
- The delay helps to reduce the decorrelation problem
- In every C steps (e.g., 10000), we replace its weights with the main network.

DQN – ALGORITHM [12]

Algorithm 1: deep Q-learning with experience replay.

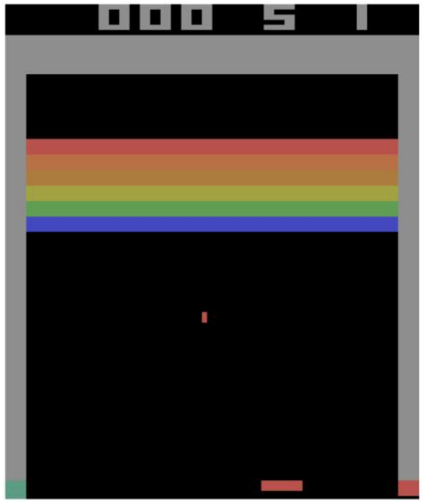
```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T do
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
    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 $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the
    network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

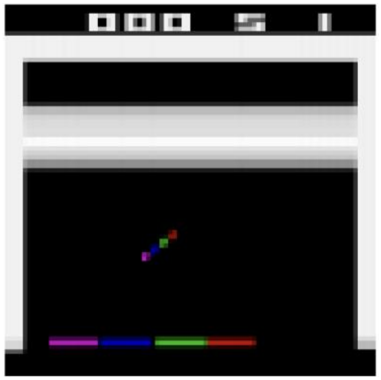
DQN – THE Φ PREPROCESSING FUNCTION

The main target of DQN was learning from pixels. This requires some image processing before giving the image to the Convultional Neural Network.

What do we do in the preprocessing function?

- Cut the redundant borders.
- When possible, make the image grayscale.
- Make it smaller and square.
- Do Frame-Stacking (making the state Markovian).

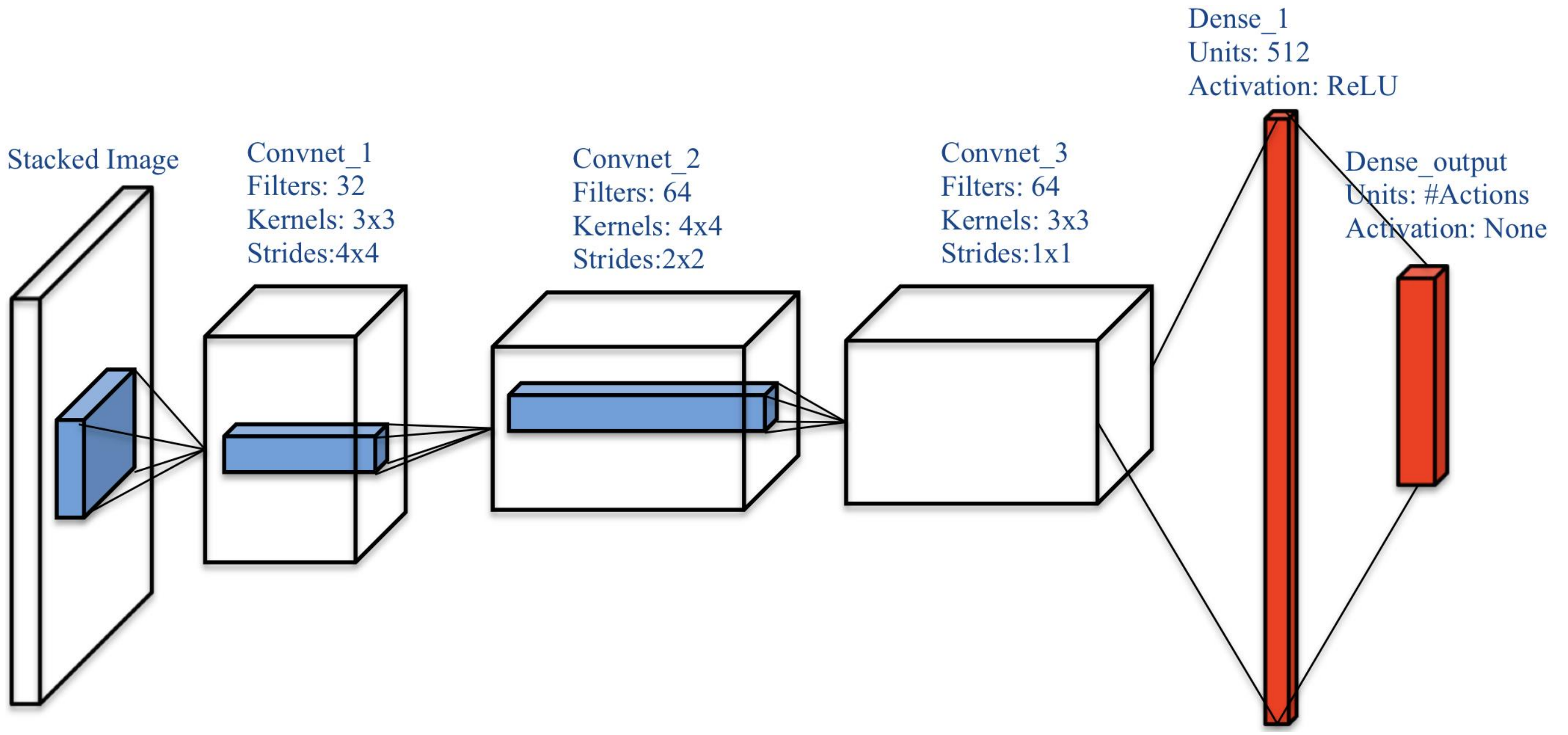




Note: In the practical, we use a classic control environment (e.g., the MountainCar-v0). The state in these environments is a small vector. We don't need to do these preprocessings on them.

DQN – NEURAL NETWORK ARCHITECTURE

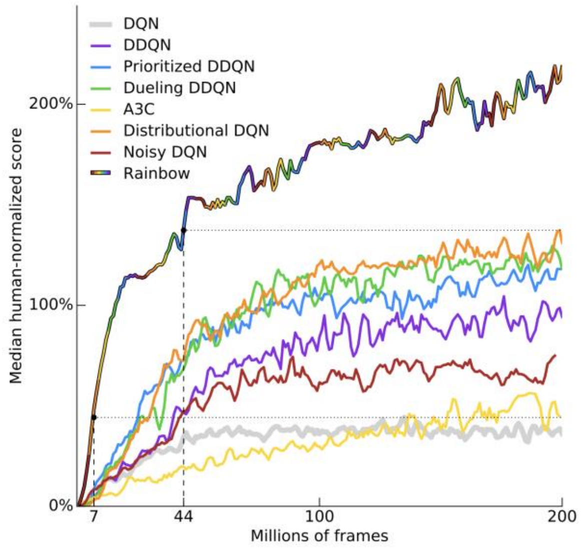
- When learning from pixels we should follow a CNN architecture like the figure below.
- For vector environments, a simple multi-layer perceptron (couple of Dense layers) is enough.



ENHANCED DQN

Can we make it even better?

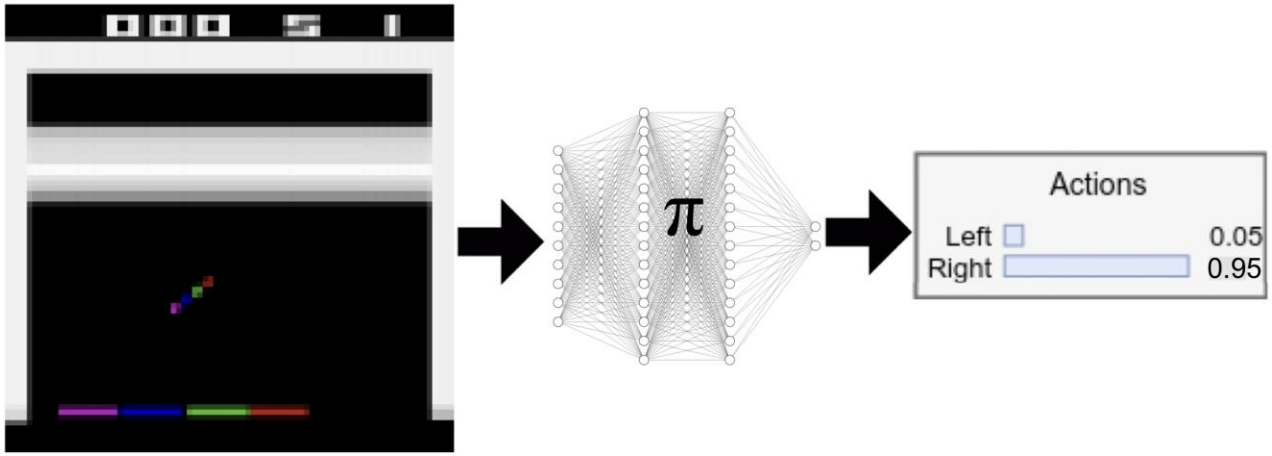
1. **Double DQN**: Solving the over-optimism problem of DQN (2015).
2. **Dueling DQN**: Making DQN more robust by breaking down the Q function at the neural network level (2016).
3. **Priority Experience Buffer (PER)**: all experiences are not equally important. By giving more weight to more important experiences, we can speed up training (2016).
4. **Rainbow**: Putting all of these and more together. Creating a super-DQN algorithm! (2017)



Policy Gradient Methods

POLICY GRADIENT METHODS

- So far, we have focused on value-based methods (e.g., SARSA, DQN).
- The decision-making process in approximate value-based methods:
  1. The state is processed by a neural net representing the Q function.
  2. The Q-values for each action and an action-policy such as ε-greedy will generate an action in each step.
  3. We update the Q function, but the goal is to DO better (a better policy, π).
- What if we directly parameterized the policy (π)?



POLICY GRADIENT METHODS - POLICY

- What is a policy? A (stochastic) policy π is a function, mapping states to action probabilities which is used to sample an action a ~ π(s).
- A good policy: like all RL method, the objective is to maximize the cumulative discounted reward (G).

$$G_t(\tau) = \sum_{t'=t}^{T} \gamma^{t'-t} r_t$$

- The objective is the expected return over the entire trajectory generated by the agent.

$$J(\pi_\theta) = E_{\tau\sim\pi_\theta}[G_t(\tau)] = E_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t r_t\right]$$

- Formally we say a policy gradient method solves the following problem:

$$\max_\theta J(\pi_\theta) = E_{\tau\sim\pi_\theta}[G_t(\tau)]$$

- To maximize the objective, we perform gradient ascent on the policy parameters:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta)$$

- There's one problem here, G(τ) is not differentiable with respect to θ, as rewards are generated by an unknown reward function independent of policy parameters.
- The solution is to transform the objective function to a differentiable format.

POLICY GRADIENT METHODS - POLICY

Given f(x) is a parameterized probability distribution p(x | θ), in this way we can transform its expectation.

$$
\begin{aligned}
&\nabla_\theta \mathbb{E}_{x\sim p(x|\theta)}[f(x)] \\
&\quad = \nabla_\theta \int dx\; f(x)p(x \mid \theta) \\
&\quad = \int dx\; \nabla_\theta \big(p(x \mid \theta) f(x)\big) \\
&\quad = \int dx\; \big(f(x)\nabla_\theta p(x \mid \theta) + p(x \mid \theta)\nabla_\theta f(x)\big) \\
&\quad = \int dx\; f(x)\nabla_\theta p(x \mid \theta) \\
&\quad = \int dx\; f(x)p(x \mid \theta)\frac{\nabla_\theta p(x \mid \theta)}{p(x \mid \theta)} \\
&\quad = \int dx\; f(x)p(x \mid \theta)\nabla_\theta \log p(x \mid \theta) \\
&\quad = \mathbb{E}_x[f(x)\nabla_\theta \log p(x \mid \theta)]
\end{aligned}
$$

By applying the derivative trick, we can rewrite the cost function:

$$\nabla_\theta \log p(x \mid \theta) = \frac{\nabla_\theta p(x \mid \theta)}{p(x \mid \theta)}$$

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)\nabla_\theta \log p(\tau \mid \theta)]$$

POLICY GRADIENT METHODS - POLICY

We will apply this result to the notion of the probability of a trajectory:

$$p(\tau \mid \theta) = \prod_{t \geq 0} p(s_{t+1} \mid s_t, a_t) \pi_\theta(a_t \mid s_t)$$

$$\log p(\tau \mid \theta) = \log \prod_{t \geq 0} p(s_{t+1} \mid s_t, a_t) \pi_\theta(a_t \mid s_t)$$

$$\log p(\tau \mid \theta) = \sum_{t \geq 0} \left(\log p(s_{t+1} \mid s_t, a_t) + \log \pi_\theta(a_t \mid s_t) \right)$$

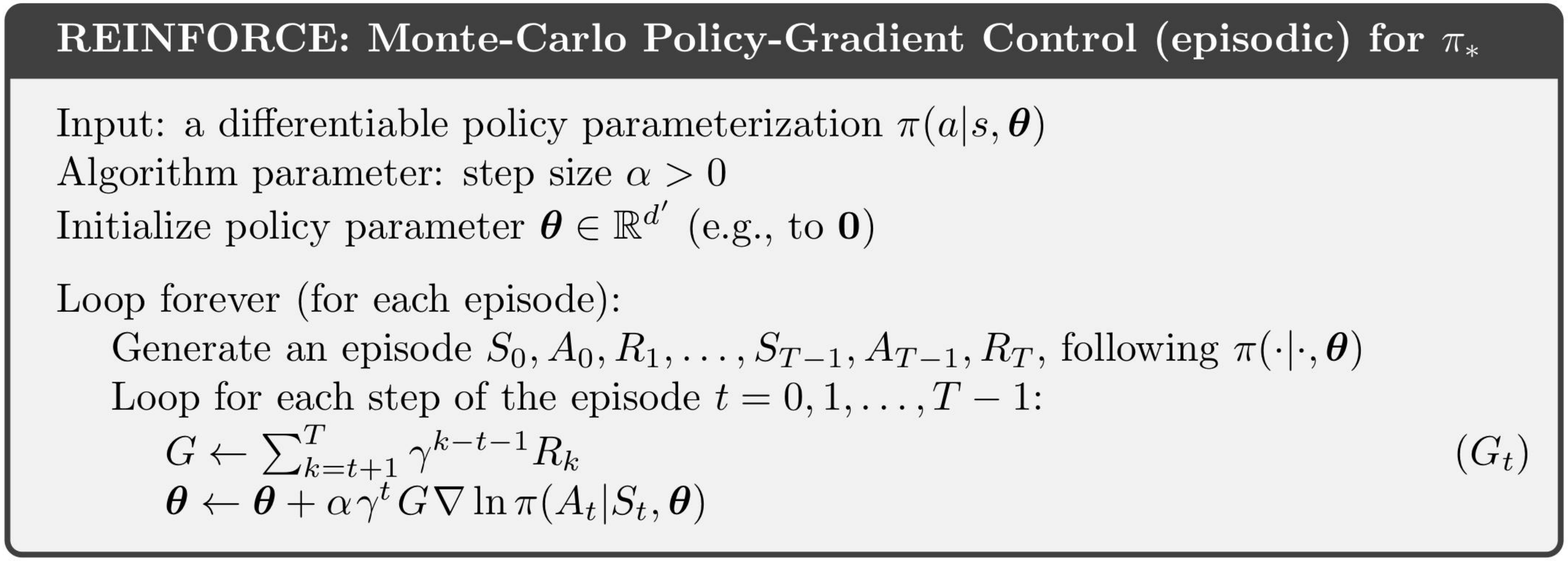
$$\nabla_\theta \log p(\tau \mid \theta) = \nabla_\theta \sum_{t \geq 0} \left(\log p(s_{t+1} \mid s_t, a_t) + \log \pi_\theta(a_t \mid s_t) \right)$$

$$\nabla_\theta \log p(\tau \mid \theta) = \nabla_\theta \sum_{t \geq 0} \log \pi_\theta(a_t \mid s_t)$$

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} R_t(\tau) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right] \checkmark$$

POLICY GRADIENT METHODS - REINFORCE

- We can turn that update formula to the simplest policy gradient algorithm known as Vanilla Policy Gradient or REINFORCE.
- REINFORCE is a Monte-Carlo (MC) algorithm. Like all MC algorithms, we generate an entire episode then use the returns in reverse to update our policy.



REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

```
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 $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$                    $(G_t)$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$
```

POLICY GRADIENT METHODS – REINFORCE WITH BASELINE

- As a stochastic method REINFORCE has good theoretical convergence properties, but as a Monte-Carlo method may be of high variance and lead to slow learning.

- REINFORCE can be generalized to include a comparison of the action value to an arbitrary baseline. The baseline has variance reduction properties.

- The baseline can be any function, even a random variable, as long as it does not vary with the action (a). This gaurantees that its gradient remains unchanged.

$$(G - b(S_t))\nabla \log \pi(A_t|S_t, \theta) = (G - b(S_t))\frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)}$$

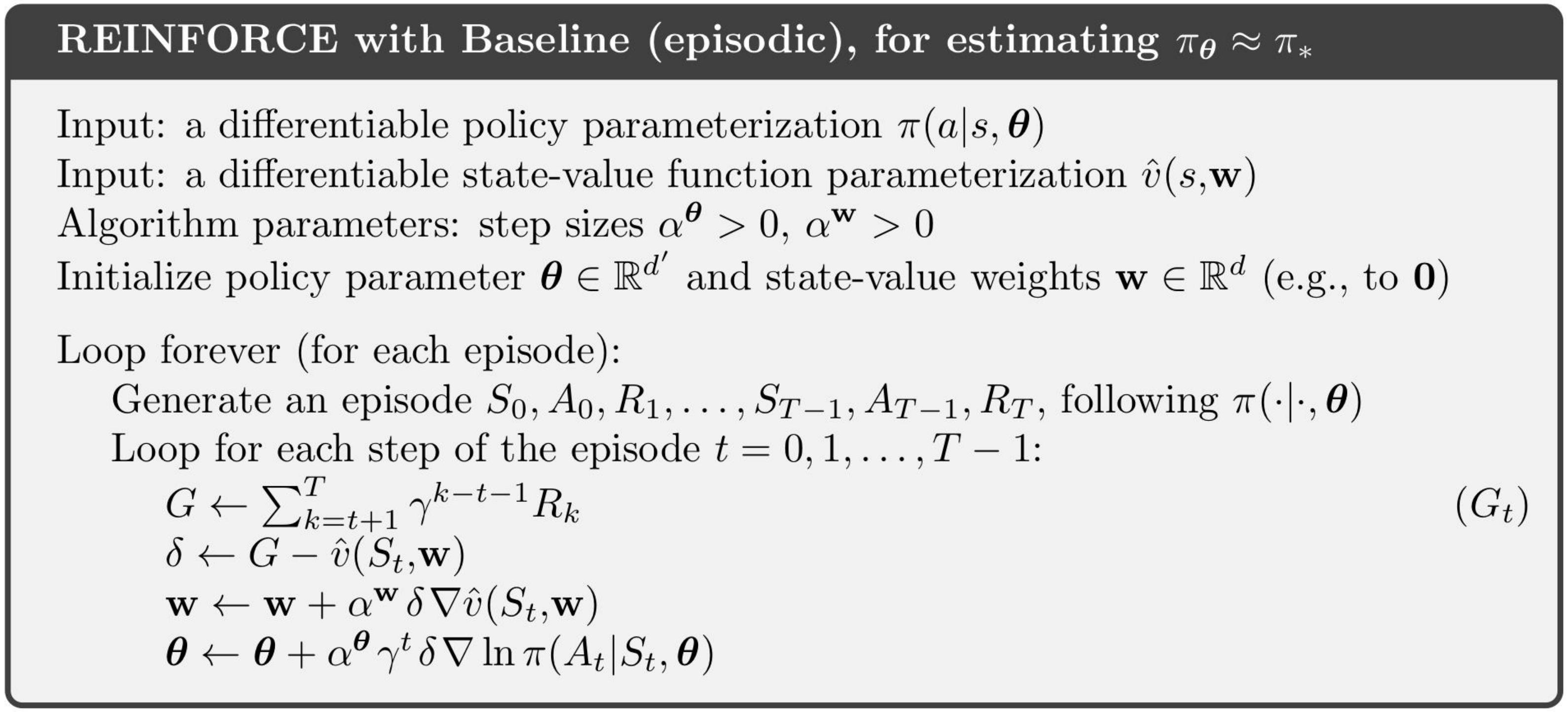
$$\sum_a b(s)\nabla \pi(a|s, \theta) = b(s)\sum_a \nabla \pi(a|s, \theta) = b(s) \times \nabla 1 = 0$$

- This can be the basis of a new update rule:

$$\theta_{t+1} \leftarrow \theta_t + \alpha\left(G_t - b(S_t)\right)\nabla \log \pi(A_t|S_t, \theta_t)$$

POLICY GRADIENT METHODS – REINFORCE WITH BASELINE

- One natural choice for the baseline is an estimate of the state value, v(S, w).
- So, we use MC to learn two functions the policy and the state value function.
- Notice that they're parameterized by two different functions and have different learning rates.

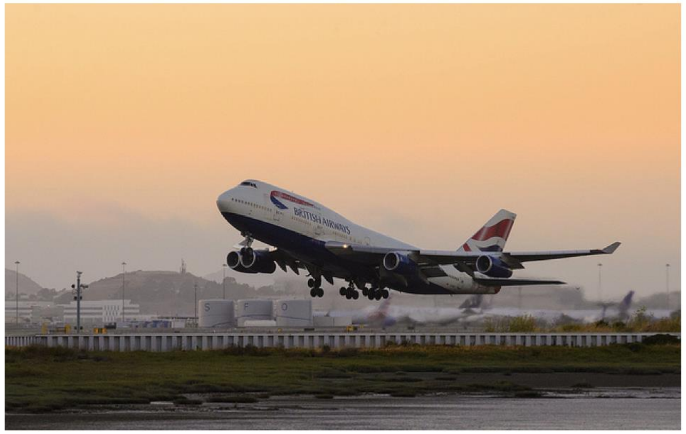


REINFORCE with Baseline (episodic), for estimating $\pi_\theta \approx \pi_*$

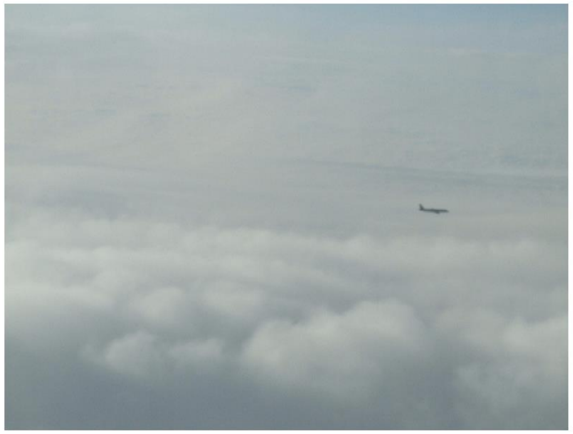
```
Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
    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 $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$                    $(G_t)$
        $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$
```

POLICY GRADIENT METHODS – AN APPLICATION OF THE REINFORCE IN TEXT GENERATION (MIXER, SCST) [5, 6, 7]

- RL can act as a tool for text generation.
- GANs, RL, Attention, and Transformers are the state-of-the-art methods in this field. More on this in week 13.



Human: a large jetliner taking off from an airport runway
ATTN+CIDER: a large airplane is flying in the sky
Ours: a large airplane taking off from runway



Human: a jet airplane flying above the clouds in the distance
ATTN+CIDER: a large airplane is flying in the sky
Ours: a plane flying in the sky with a cloudy sky

POLICY GRADIENT METHODS – AN APPLICATION OF THE REINFORCE IN TEXT GENERATION (MIXER) [5]

- The Cross-Entropy (XENT) method is conventionally used to maximize the probability of the observed sequence according to the model.
- The loss function that we are trying to minimize with Cross-Entropy:

$$L = -\log p(w_1, \ldots, w_T) = -\log \prod_{t=1}^{T} p(w_t | w_1, \ldots, w_{t-1}) = -\sum_{t=1}^{T} \log p(w_t | w_1, \ldots, w_{t-1})$$

- This loss function trains the model to be good at greedily predicting the next word at each step without considering the whole sentence.
- This process clearly has limitations, we can assist it by augmenting it with RL.
- First step is to formulate it as an MDP problem (S, A, P, R):
  1. The RNN generative model. (model => agent)
  2. The agent does actions in the environment. (actions => generated words, env => context vector)
  3. The reward is generated by NLP metrics (e.g., BLEU, ROUGE-2)
- REINFORCE struggles to bootstrap the model, so we start the training with Cross-Entropy, then refine the model with REINFORCE.

POLICY GRADIENT METHODS – AN APPLICATION OF THE REINFORCE IN TEXT GENERATION (MIXER, SCST) [5, 6, 7]

- This technique and its other variations such as Self-Critical Sequence Training (SCST) have been used in Machine Translation and Image Captioning.

```
**Data**: a set of sequences with their corresponding context.
**Result**: RNN optimized for generation.
Initialize RNN at random and set $N^{XENT}$, $N^{XE+R}$ and $\Delta$;
for $s = T, 1, -\Delta$ do
    if $s == T$ then
        train RNN for $N^{XENT}$ epochs using XENT only;
    else
        train RNN for $N^{XE+R}$ epochs. Use XENT loss in the first $s$ steps, and REINFORCE (sampling from
        the model) in the remaining $T - s$ steps;
    end
end
```

Human: a large jetliner taking off from an airport runway
ATTN+CIDER: a large airplane is flying in the sky
Ours: a large airplane taking off from runway

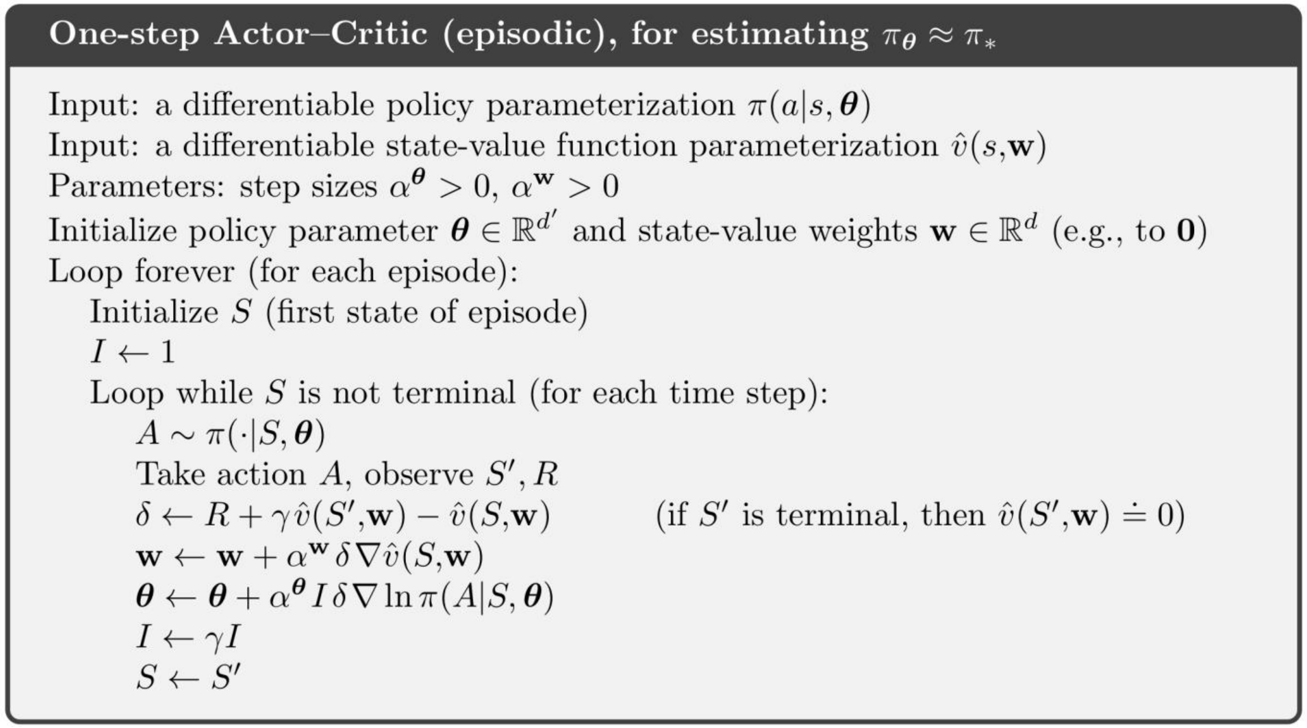
Human: a jet airplane flying above the clouds in the distance
ATTN+CIDER: a large airplane is flying in the sky
Ours: a plane flying in the sky with a cloudy sky

MACQUARIE
University

Actor-Critic Algorithms

ACTOR-CRITIC ALGORITHMS

- The REINFORCE algorithm with a baseline sets the stage for actor-critic approaches.
- Although REINFORCE with baseline can learn both policy and the state-value function, the state value function is still a baseline not a critic.
- A critic usually uses bootstrapping like a TD method and directly influences policies learning.
- Like the 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 $\hat{v}(S',\mathbf{w}) \doteq 0$)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\nabla\hat{v}(S,\mathbf{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I\delta\nabla\ln\pi(A|S,\boldsymbol{\theta})$
        $I \leftarrow \gamma I$
        $S \leftarrow S'$
```

ACTOR-CRITIC ALGORITHMS – ADVANCED METHODS

MACQUARIE University

- There are methods that can make the one-step Actor-Critic better.
  1. Making it multi-process, so differnet copies of the same agent play the same game and push their updates to a central neural network. This also solves the correlation problem. This was used in A2C, A3C, IMPALA and many other Asynchronous algorithms.
  2. Limiting the policy update. This helps the agent's learning process, and it doesn't allow it to deviate from the path to the optimal. This idea was first tested in TRPO and later in PPO. PPO was incrediably successful.
  3. Reduce the bais in the critic by using n-step (4-step or 5-step) boostrapping or a more advanced one called GAE.

$$\delta = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

$$\delta = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \gamma^4 V(S_{t+4}) - V(S_t)$$

  4. Using deterministic policies with Q functions as the critic and Experience Replay Buffer like DQN. This led to the appearance of DDPG, TD3, and D4PG which were very successful in continuous action space environments.

CONCLUSION

- Deep Reinforcement Learning has achieved tremendous success in video games and other areas such as robotics by combining the generalization power of neural networks and learning from experience from RL.
- This is just the beginning! There are still so many open questions.
- In our RL journey, you may have noticed one thing that experience is the key to have a more "human-like" behaviour.

REFERENCES

1. R. Sutton, "Reinforcement Learning: An introduction (2e)", 2018.
2. Graesser, Loon Keng, "Foundations of Deep Reinforcement Learning", 2019.
3. R. Sutton, "Introduction to RL Lecture", NeurIPS 2015.
4. Mnih et al, Human-level control through deep reinforcement learning, 2015.
5. M. Ranzato et al, Sequence Level Training with Recurrent Neural Networks, 2016.
6. R. Luo et al, Discriminability objective for training descriptive captions, 2018.
7. S. Rennie et al, Self-critical Sequence Training for Image Captioning, 2017.