

# **COMP9444**

## **Neural Networks and Deep Learning**

### **7b. Deep Reinforcement Learning**

# Outline

---

- Policy Learning
  - ▶ Evolution Strategies
  - ▶ Policy Gradients
- Actor-Critic
- History of Reinforcement Learning
- Deep Q-Learning for Atari Games
- Asynchronous Advantage Actor Critic (A3C)

# Hill Climbing (Evolution Strategy)

---

- Initialize “champ” policy  $\theta_{\text{champ}} = 0$
- for each trial, generate “mutant” policy

$$\theta_{\text{mutant}} = \theta_{\text{champ}} + \text{Gaussian noise (fixed } \sigma)$$

- champ and mutant are evaluated on the same task(s)
- if mutant does “better” than champ,

$$\theta_{\text{champ}} \leftarrow (1 - \alpha)\theta_{\text{champ}} + \alpha\theta_{\text{mutant}}$$

- in some cases, the size of the update is scaled according to the difference in fitness (and may be negative)

# Policy Gradients

---

Policy Gradients are an alternative to Evolution Strategy, which use gradient ascent rather than random updates.

Let's first consider episodic games. The agent takes a sequence of actions

$$a_1 \ a_2 \ \dots \ a_t \ \dots \ a_m$$

At the end it receives a reward  $r_{\text{total}}$ . We don't know which actions contributed the most, so we just reward all of them equally. If  $r_{\text{total}}$  is high (low), we change the parameters to make the agent more (less) likely to take the same actions in the same situations. In other words, we want to increase (decrease)

$$\log \prod_{t=1}^m \pi_{\theta}(a_t | s_t) = \sum_{t=1}^m \log \pi_{\theta}(a_t | s_t)$$

# Policy Gradients

---

If  $r_{\text{total}} = +1$  for a win and  $-1$  for a loss, we can simply multiply the log probability by  $r_{\text{total}}$ . Differentials can be calculated using the gradient

$$\nabla_{\theta} r_{\text{total}} \sum_{t=1}^m \log \pi_{\theta}(a_t | s_t) = r_{\text{total}} \sum_{t=1}^m \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

The gradient of the log probability can be calculated nicely using Softmax.

If  $r_{\text{total}}$  takes some other range of values, we can replace it with  $(r_{\text{total}} - b)$  where  $b$  is a fixed value, called the **baseline**.

# REINFORCE Algorithm

---

We then get the following REINFORCE algorithm:

```
for each trial
  run trial and collect states  $s_t$ , actions  $a_t$ , and reward  $r_{\text{total}}$ 
  for  $t = 1$  to length(trial)
     $\theta \leftarrow \theta + \eta(r_{\text{total}} - b) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$ 
  end
end
```

This algorithm has successfully been applied, for example, to learn to play the game of Pong from raw image pixels.

# Policy Gradients

---

We wish to extend the framework of Policy Gradients to non-episodic domains, where rewards are received incrementally throughout the game (e.g. PacMan, Space Invaders).

Every policy  $\pi_\theta$  determines a distribution  $\rho_{\pi_\theta}(s)$  on  $\mathcal{S}$

$$\rho_{\pi_\theta}(s) = \sum_{t \geq 0} \gamma^t \text{prob}_{\pi_\theta, t}(s)$$

where  $\text{prob}_{\pi_\theta, t}(s)$  is the probability that, after starting in state  $s_0$  and performing  $t$  actions, the agent will be in state  $s$ . We can then define the fitness of policy  $\pi$  as

$$\text{fitness}(\pi_\theta) = \sum_s \rho_{\pi_\theta}(s) \sum_a Q^{\pi_\theta}(s, a) \pi_\theta(a|s)$$

# Policy Gradients

---

$$\text{fitness}(\pi_\theta) = \sum_s \rho_{\pi_\theta}(s) \sum_a Q^{\pi_\theta}(s, a) \pi_\theta(a|s)$$

Note: In the case of episodic games, we can take  $\gamma = 1$ , in which case  $Q^{\pi_\theta}(s, a)$  is simply the expected reward at the end of the game.

However, the above equation holds in the non-episodic case as well.

The gradient of  $\rho_{\pi_\theta}(s)$  and  $Q^{\pi_\theta}(s, a)$  are extremely hard to determine, so we ignore them and instead compute the gradient only for the last term  $\pi_\theta(a|s)$ .

$$\nabla_\theta \text{fitness}(\pi_\theta) = \sum_s \rho_{\pi_\theta}(s) \sum_a Q^{\pi_\theta}(s, a) \nabla_\theta \pi_\theta(a|s)$$



# The Log Trick

---

$$\begin{aligned}\sum_a Q^{\pi_\theta}(s, a) \nabla_\theta \pi_\theta(a|s) &= \sum_a Q^{\pi_\theta}(s, a) \pi_\theta(a|s) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \\ &= \sum_a Q^{\pi_\theta}(s, a) \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s)\end{aligned}$$

So

$$\begin{aligned}\nabla_\theta \text{fitness}(\pi_\theta) &= \sum_s \rho_{\pi_\theta}(s) \sum_a Q^{\pi_\theta}(s, a) \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s) \\ &= \mathbf{E}_{\pi_\theta} [Q^{\pi_\theta}(s, a) \nabla_\theta \log \pi_\theta(a|s)]\end{aligned}$$

The reason for the last equality is this:

$\rho_{\pi_\theta}(s)$  is the number of times (discounted by  $\gamma^t$ ) that we expect to visit state  $s$  when using policy  $\pi_\theta$ . Whenever state  $s$  is visited, action  $a$  will be chosen with probability  $\pi_\theta(a|s)$ .

# Actor-Critic

---

Recall:

$$\nabla_{\theta} \text{fitness}(\pi_{\theta}) = \mathbf{E}_{\pi_{\theta}}[Q^{\pi_{\theta}}(s, a) \nabla_{\theta} \log \pi_{\theta}(a|s)]$$

For non-episodic games, we cannot easily find a good estimate for  $Q^{\pi_{\theta}}(s, a)$ . One approach is to consider a family of Q-Functions  $Q_w$  determined by parameters  $w$  (different from  $\theta$ ) and learn  $w$  so that  $Q_w$  approximates  $Q^{\pi_{\theta}}$ , at the same time that the policy  $\pi_{\theta}$  itself is also being learned.

This is known as an **Actor-Critic** approach because the policy determines the action, while the Q-Function estimates how good the current policy is, and thereby plays the role of a critic.

# Actor Critic Algorithm

---

```
for each trial
  sample  $a_0$  from  $\pi(a|s_0)$ 
  for each timestep  $t$  do
    sample reward  $r_t$  from  $\mathcal{R}(r|s_t, a_t)$ 
    sample next state  $s_{t+1}$  from  $\delta(s|s_t, a_t)$ 
    sample action  $a_{t+1}$  from  $\pi(a|s_{t+1})$ 
     $\frac{dE}{dQ} = -[r_t + \gamma Q_w(s_{t+1}, a_{t+1}) - Q_w(s_t, a_t)]$ 
     $\theta \leftarrow \theta + \eta_\theta Q_w(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t | s_t)$ 
     $w \leftarrow w - \eta_w \frac{dE}{dQ} \nabla_w Q_w(s_t, a_t)$ 
  end
end
```

# Reinforcement Learning Timeline

---

## ■ model-free methods

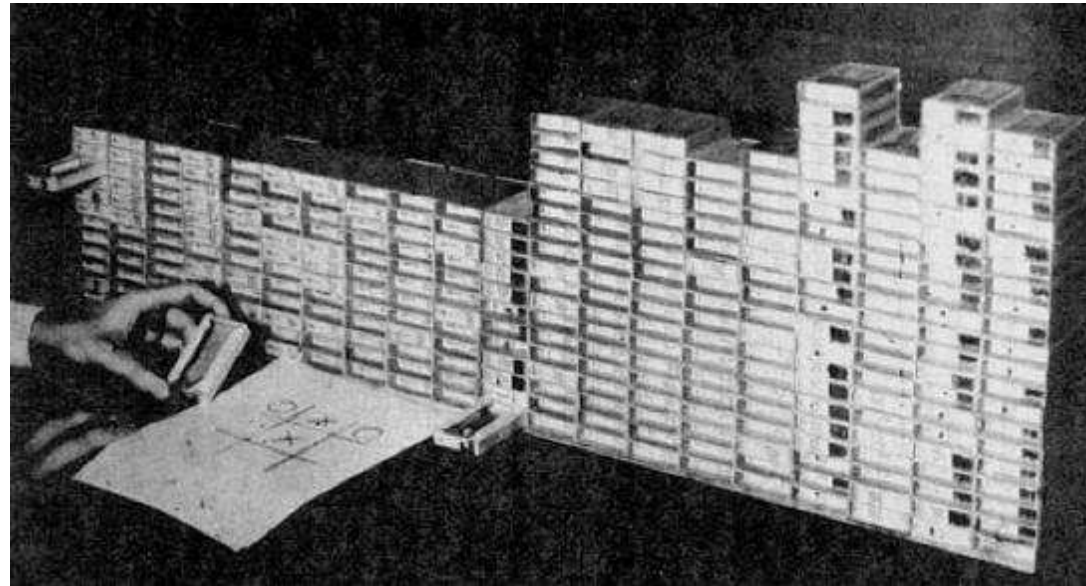
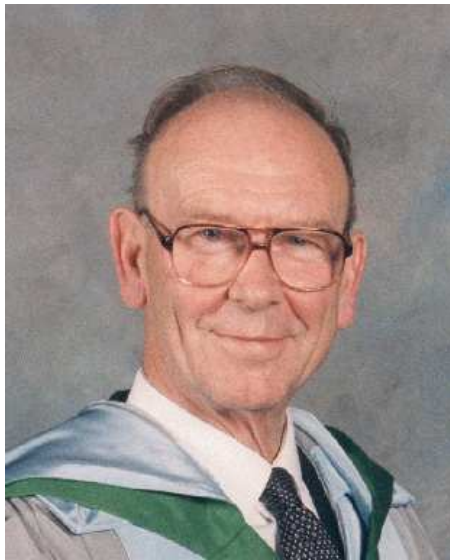
- ▶ 1961 MENACE tic-tac-toe (Donald Michie)
- ▶ 1986 TD( $\lambda$ ) (Rich Sutton)
- ▶ 1989 TD-Gammon (Gerald Tesauro)
- ▶ 2015 Deep Q Learning for Atari Games
- ▶ 2016 A3C (Mnih et al.)
- ▶ 2017 OpenAI Evolution Strategies (Salimans et al.)

## ■ methods relying on a world model

- ▶ 1959 Checkers (Arthur Samuel)
- ▶ 1997 TD-leaf (Baxter et al.)
- ▶ 2009 TreeStrap (Veness et al.)
- ▶ 2016 Alpha Go (Silver et al.)

# MENACE

---



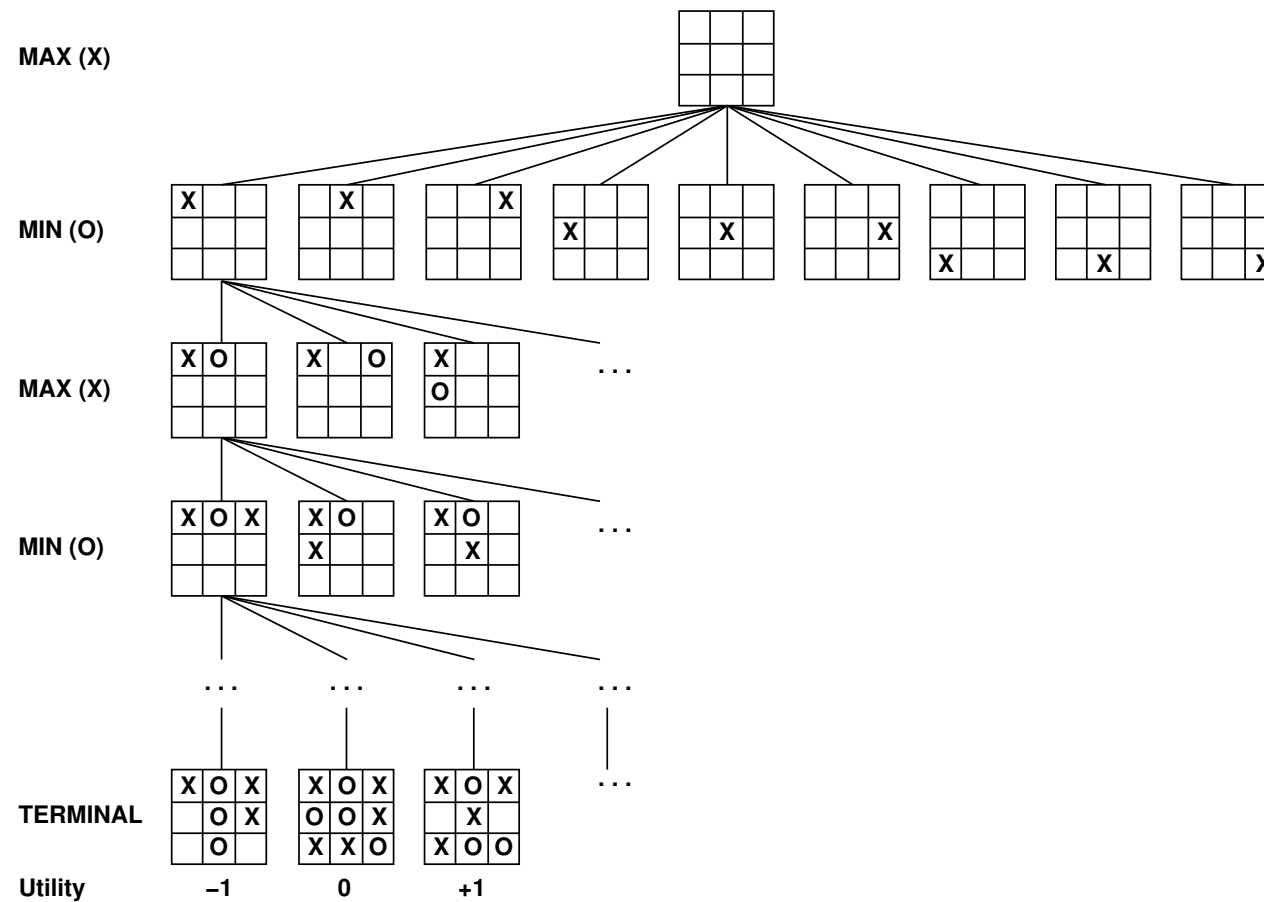
Machine Educable Noughts And Crosses Engine  
Donald Michie, 1961

# MENACE

---



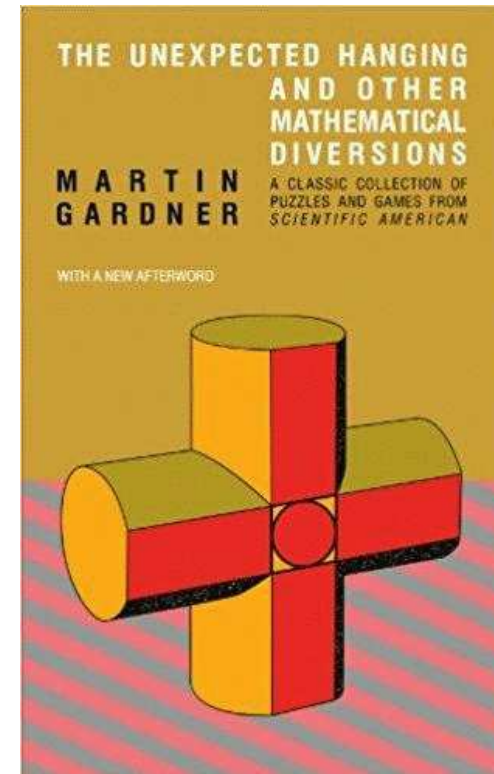
# Game Tree (2-player, deterministic)





# Martin Gardner and HALO

---





# Hexapawn Boxes

---



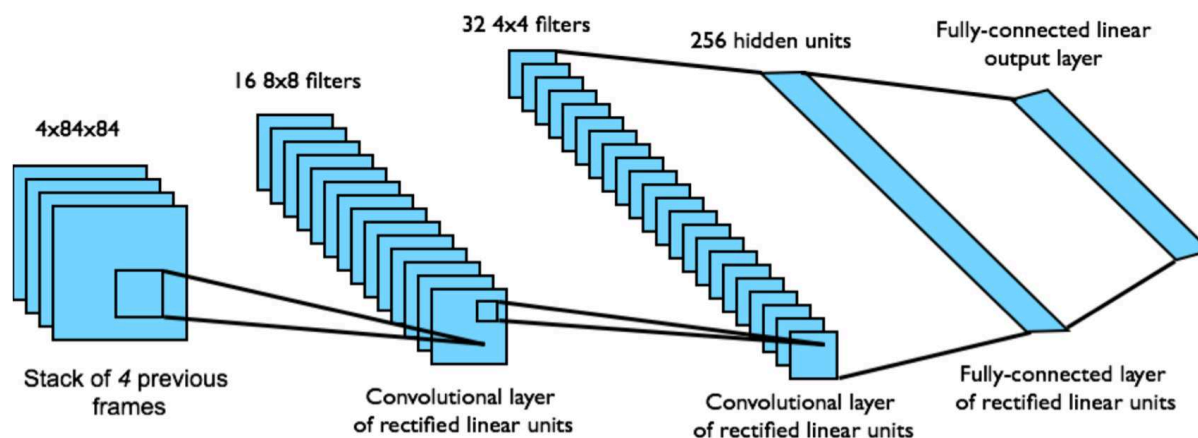
# Reinforcement Learning with BOXES

---

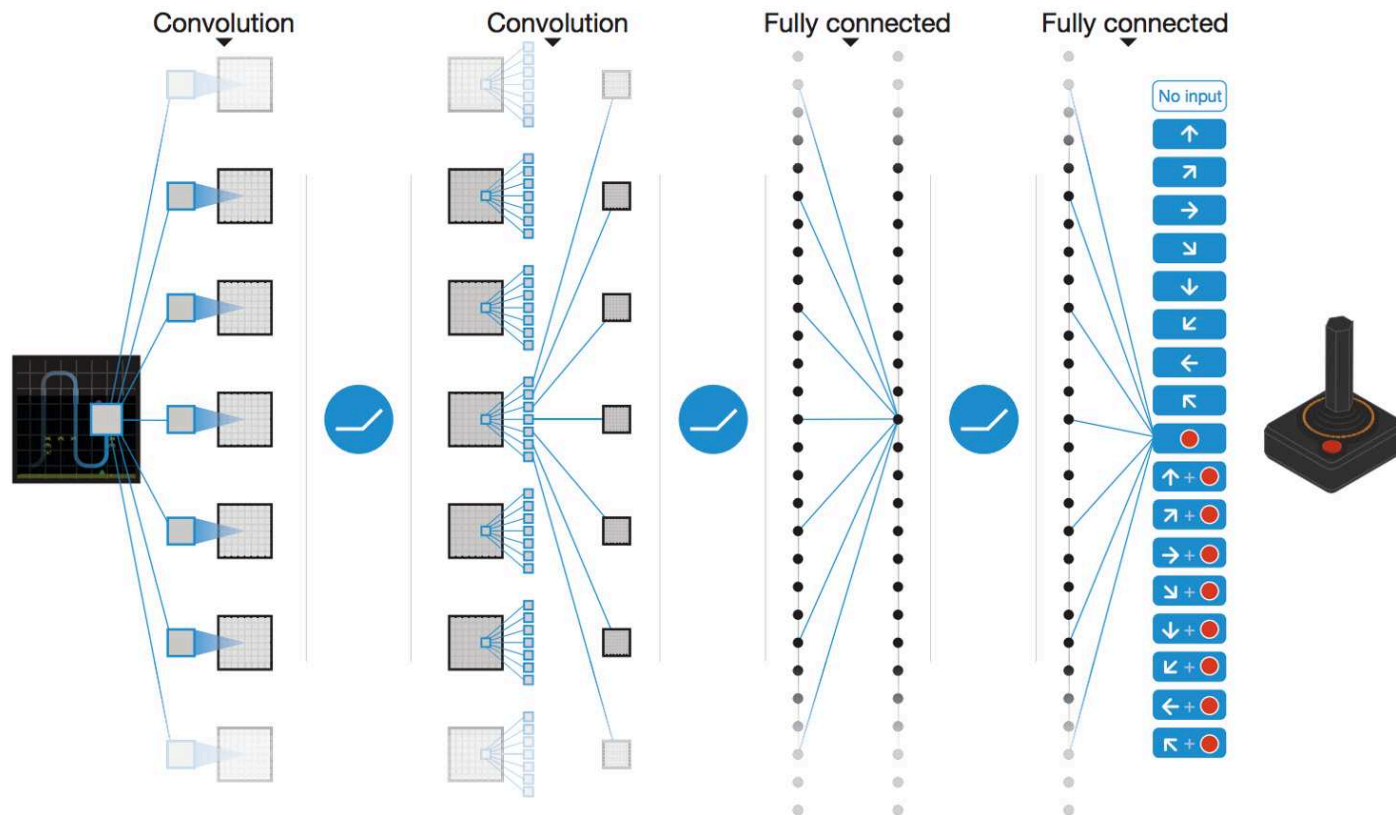
- this BOXES algorithm was later adapted to learn more general tasks such as Pole Balancing, and helped lay the foundation for the modern field of Reinforcement Learning
- for various reasons, interest in Reinforcement Learning faded in the late 70's and early 80's, but was revived in the late 1980's, largely through the work of Richard Sutton
- Gerald Tesauro applied Sutton's TD-Learning algorithm to the game of Backgammon in 1989

# Deep Q-Learning for Atari Games

- end-to-end learning of values  $Q(s, a)$  from pixels  $s$
- input state  $s$  is stack of raw pixels from last 4 frames
  - ▶ 8-bit RGB images,  $210 \times 160$  pixels
- output is  $Q(s, a)$  for 18 joystick/button positions
- reward is change in score for that timestep



# Deep Q-Network



# Q-Learning

---

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta [r_t + \gamma \max_b Q(s_{t+1}, b) - Q(s_t, a_t)]$$

- with lookup table, Q-learning is guaranteed to eventually converge
- for serious tasks, there are too many states for a lookup table
- instead,  $Q_w(s, a)$  is parametrized by weights  $w$ , which get updated so as to minimize

$$[r_t + \gamma \max_b Q_w(s_{t+1}, b) - Q_w(s_t, a_t)]^2$$

- ▶ note: gradient is applied only to  $Q_w(s_t, a_t)$ , not to  $Q_w(s_{t+1}, b)$
- this works well for some tasks, but is challenging for Atari games, partly due to temporal correlations between samples (i.e. large number of similar situations occurring one after the other)

# Deep Q-Learning with Experience Replay

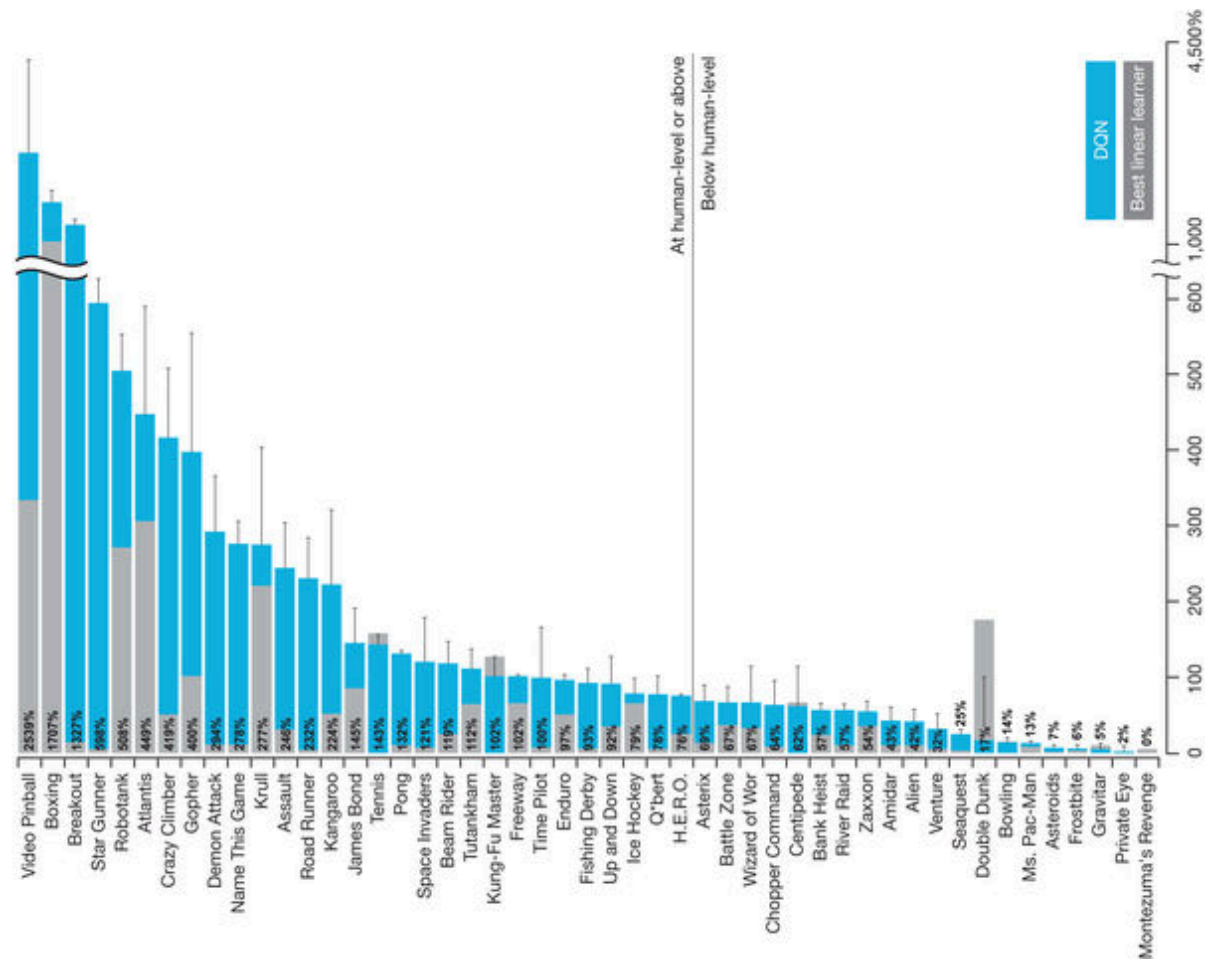
---

- choose actions using current Q function ( $\epsilon$ -greedy)
- build a database of experiences  $(s_t, a_t, r_t, s_{t+1})$
- sample asynchronously from database and apply update, to minimize

$$[r_t + \gamma \max_b Q_w(s_{t+1}, b) - Q_w(s_t, a_t)]^2$$

- removes temporal correlations by sampling from variety of game situations in random order
- makes it easier to parallelize the algorithm on multiple GPUs

# DQN Results for Atari Games



# DQN Improvements

---

## ■ Prioritised Replay

- ▶ weight experience according to surprise

## ■ Double Q-Learning

- ▶ current Q-network  $w$  is used to **select** actions
- ▶ older Q-network  $\bar{w}$  is used to **evaluate** actions

## ■ Advantage Function

- ▶ **action-independent** value function  $V_u(s)$
- ▶ **action-dependent** advantage function  $A_w(s, a)$

$$Q(s, a) = V_u(s) + A_w(s, a)$$



# Prioritised Replay

---

- instead of sampling experiences uniformly, store them in a priority queue according to the DQN error

$$|r_t + \gamma \max_b Q_w(s_{t+1}, b) - Q_w(s_t, a_t)|$$

- this ensures the system will concentrate more effort on situations where the Q value was “surprising” (in the sense of being far away from what was predicted)

## Double Q-Learning

---

- if the same weights  $w$  are used to select actions and evaluate actions, this can lead to a kind of confirmation bias
- could maintain two sets of weights  $w$  and  $\bar{w}$ , with one used for selection and the other for evaluation (then swap their roles)
- in the context of Deep Q-Learning, a simpler approach is to use the current “online” version of  $w$  for selection, and an older “target” version  $\bar{w}$  for evaluation; we therefore minimize

$$[r_t + \gamma Q_{\bar{w}}(s_{t+1}, \operatorname{argmax}_b Q_w(s_{t+1}, b)) - Q_w(s_t, a_t)]^2$$

- a new version of  $\bar{w}$  is periodically calculated from the distributed values of  $w$ , and this  $\bar{w}$  is broadcast to all processors.

# Advantage Function

---

The Q Function  $Q^\pi(s, a)$  can be written as a sum of the value function  $V^\pi(s)$  plus an **advantage function**  $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$

$A^\pi(s, a)$  represents the advantage (or disadvantage) of taking action  $a$  in state  $s$ , compared to taking the action preferred by the current policy  $\pi$ . We can learn approximations for these two components separately:

$$Q(s, a) = V_u(s) + A_w(s, a)$$

Note that actions can be selected just using  $A_w(s, a)$ , because

$$\operatorname{argmax}_b Q(s_{t+1}, b) = \operatorname{argmax}_b A_w(s_{t+1}, b)$$

# Advantage Actor Critic

---

Recall that in the REINFORCE algorithm, a baseline  $b$  could be subtracted from  $r_{\text{total}}$

$$\theta \leftarrow \theta + \eta(r_{\text{total}} - b) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

In the actor-critic framework,  $r_{\text{total}}$  is replaced by  $Q(s_t, a_t)$

$$\theta \leftarrow \theta + \eta_{\theta} Q(s_t, a_t) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

We can also subtract a baseline from  $Q(s_t, a_t)$ . This baseline must be independent of the action  $a_t$ , but it could be dependent on the state  $s_t$ . A good choice of baseline is the value function  $V_u(s)$ , in which case the Q function is replaced by the advantage function

$$A_w(s, a) = Q(s, a) - V_u(s)$$

# Asynchronous Advantage Actor Critic

---

- use policy network to choose actions
- learn a parameterized Value function  $V_u(s)$  by TD-Learning
- estimate Q-value by n-step sample

$$Q(s_t, a_t) = r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{n-1} r_{t+n} + \gamma^n V_u(s_{t+n})$$

- update policy by

$$\theta \leftarrow \theta + \eta_\theta [Q(s_t, a_t) - V_u(s_t)] \nabla_\theta \log \pi_\theta(a_t | s_t)$$

- update Value function by minimizing

$$[Q(s_t, a_t) - V_u(s_t)]^2$$

# Latest Research in Deep RL

---

- augment A3C with unsupervised auxiliary tasks
- encourage exploration, increased entropy
- encourage actions for which the rewards are less predictable
- concentrate on state features from which the preceding action is more predictable
- transfer learning (between tasks)
- inverse reinforcement learning (infer rewards from policy)
- hierarchical RL
- multi-agent RL

# References

---

- David Silver, Deep Reinforcement Learning Tutorial,  
[http://icml.cc/2016/tutorials/deep\\_rl\\_tutorial.pdf](http://icml.cc/2016/tutorials/deep_rl_tutorial.pdf)
- A Brief Survey of Deep Reinforcement Learning,  
<https://arxiv.org/abs/1708.05866>
- Asynchronous Methods for Deep Reinforcement Learning,  
<https://arxiv.org/abs/1602.01783>
- Evolution Strategies as a Scalable Alternative to Reinforcement Learning, <https://arxiv.org/abs/1703.03864>
- Eric Jang, Beginner's Guide to Variational Methods,  
<http://blog.evjang.com/2016/08/variational-bayes.html>