

# COMP9444

## Neural Networks and Deep Learning

### 10. Deep Reinforcement Learning

# Outline

---

- History of Reinforcement Learning
- Deep Q-Learning for Atari Games
- Actor-Critic
- Asynchronous Advantage Actor Critic (A3C)
- Evolutionary / Variational methods

# Reinforcement Learning Timeline

---

## ■ model-free methods

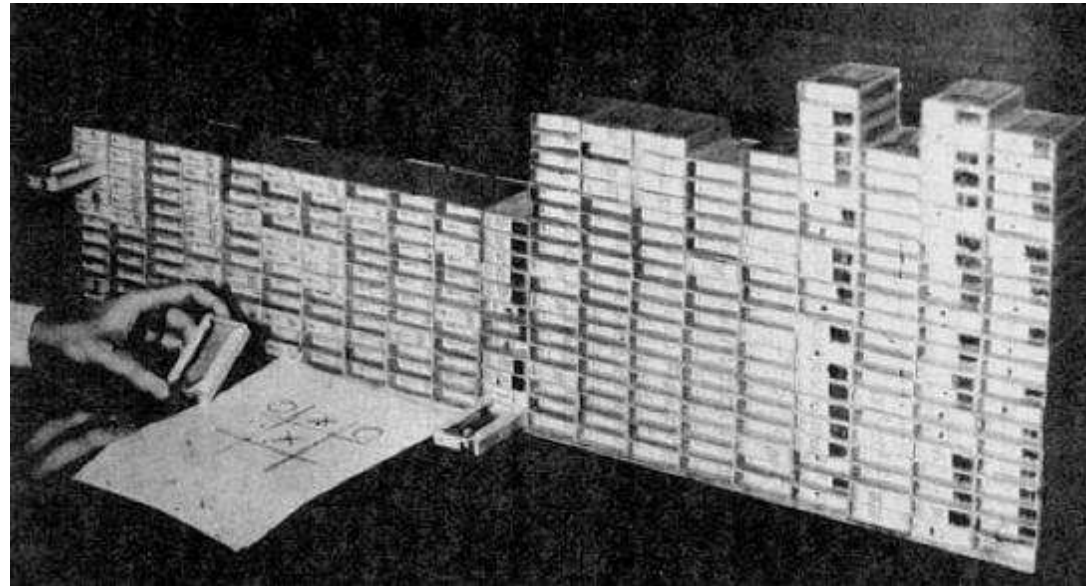
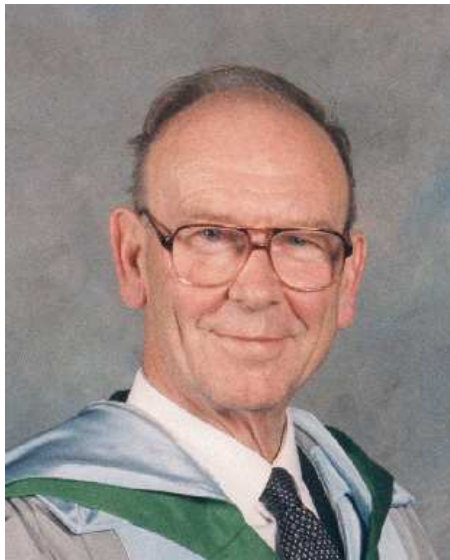
- ▶ 1961 MENACE tic-tac-toe (Donald Michie)
- ▶ 1986 TD( $\lambda$ ) (Rich Sutton)
- ▶ 1989 TD-Gammon (Gerald Tesauro)
- ▶ 2015 Deep Q Learning for Atari Games
- ▶ 2016 A3C (Mnih et al.)
- ▶ 2017 OpenAI Evolution Strategies (Salimans et al.)

## ■ 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

---



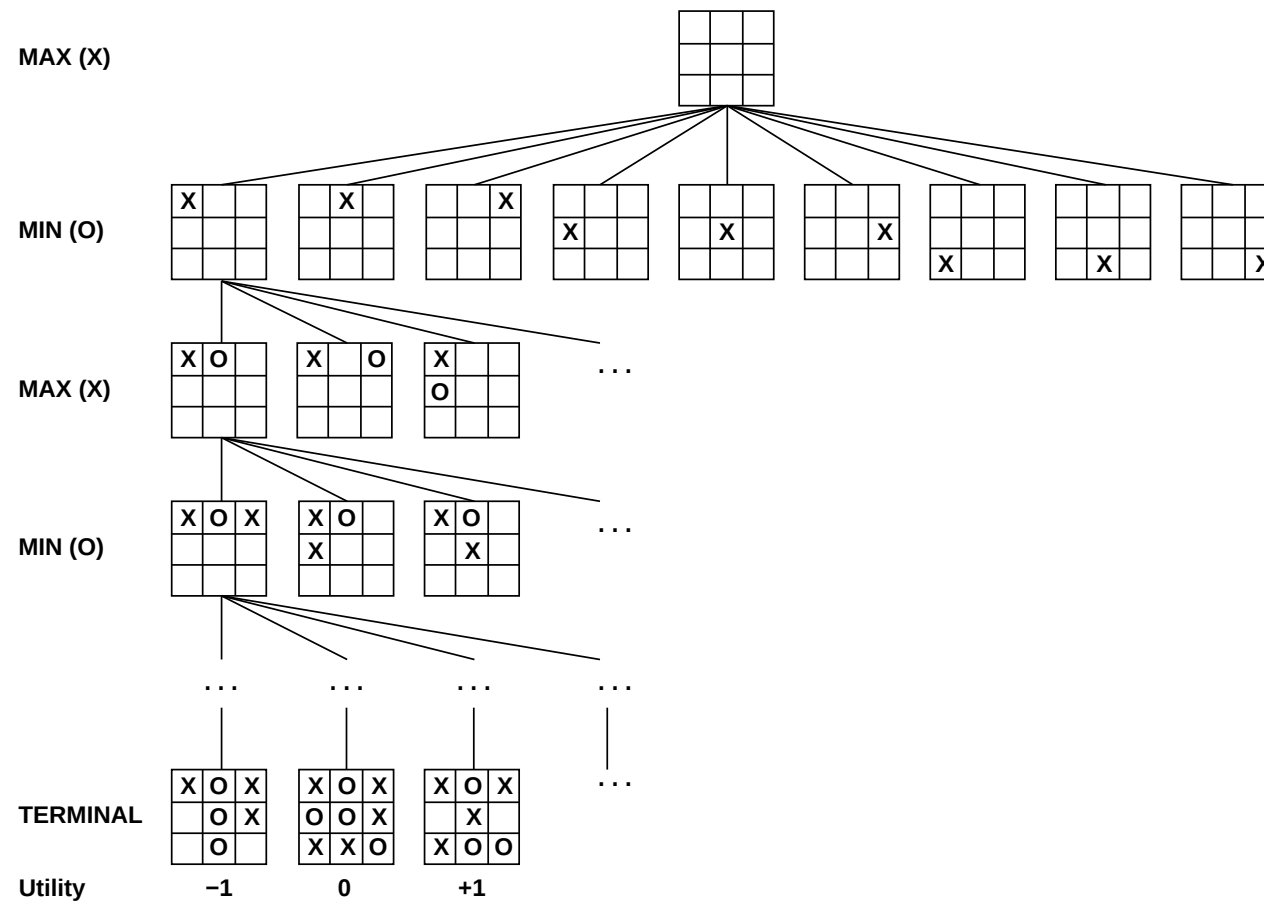
Machine Educable Noughts And Crosses Engine  
Donald Michie, 1961

# MENACE

---

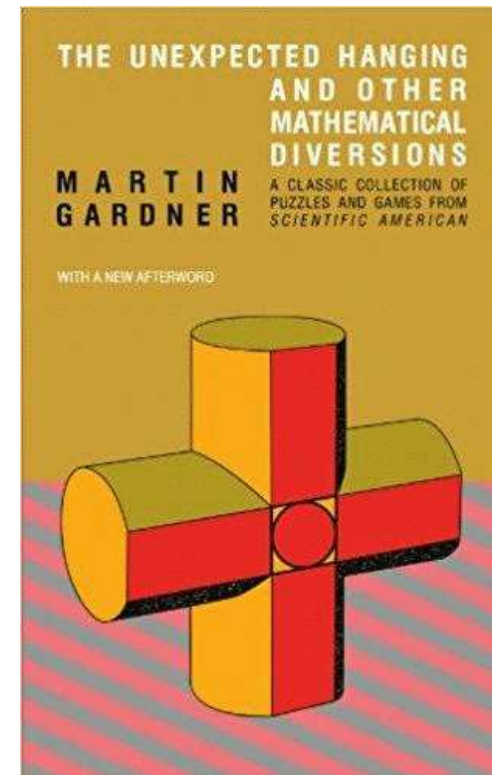


# Game Tree (2-player, deterministic)



# Martin Gardner and HALO

---



# Hexapawn Boxes

---



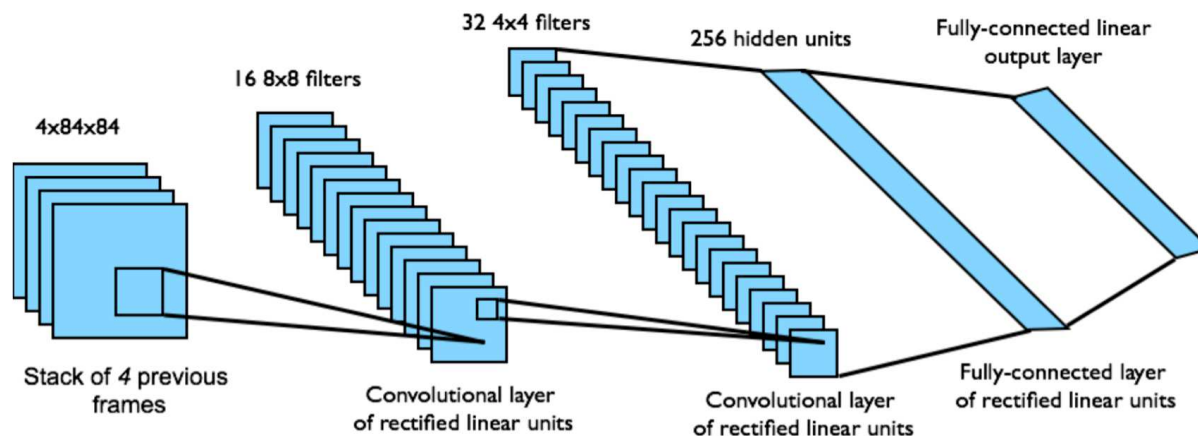
# Reinforcement Learning with BOXES

---

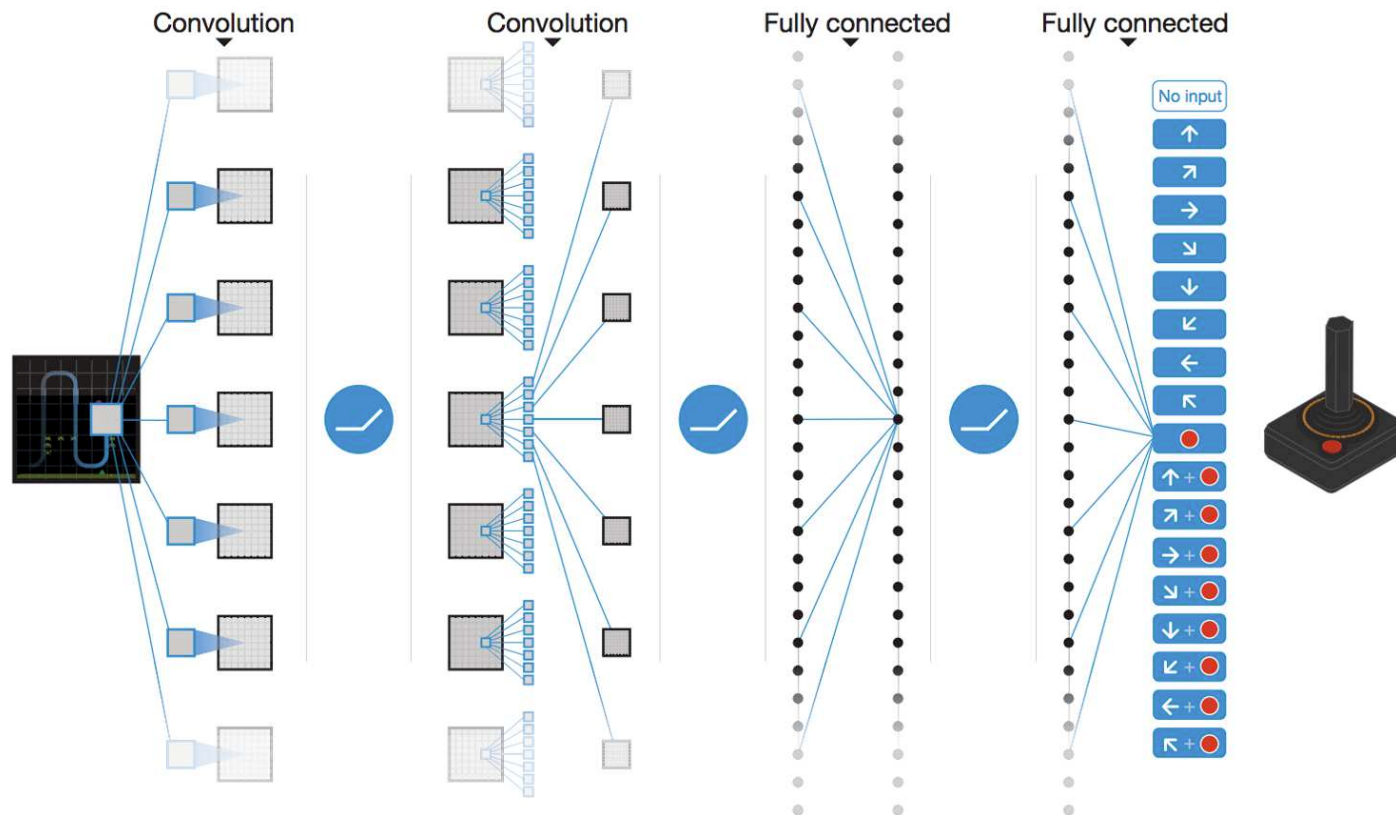
- this BOXES algorithm was later adapted to learn more general tasks such as Pole Balancing, and helped lay the foundation for the modern field of Reinforcement Learning
- for various reasons, interest in Reinforcement Learning faded in the late 70's and early 80's, but was revived in the late 1980's, 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

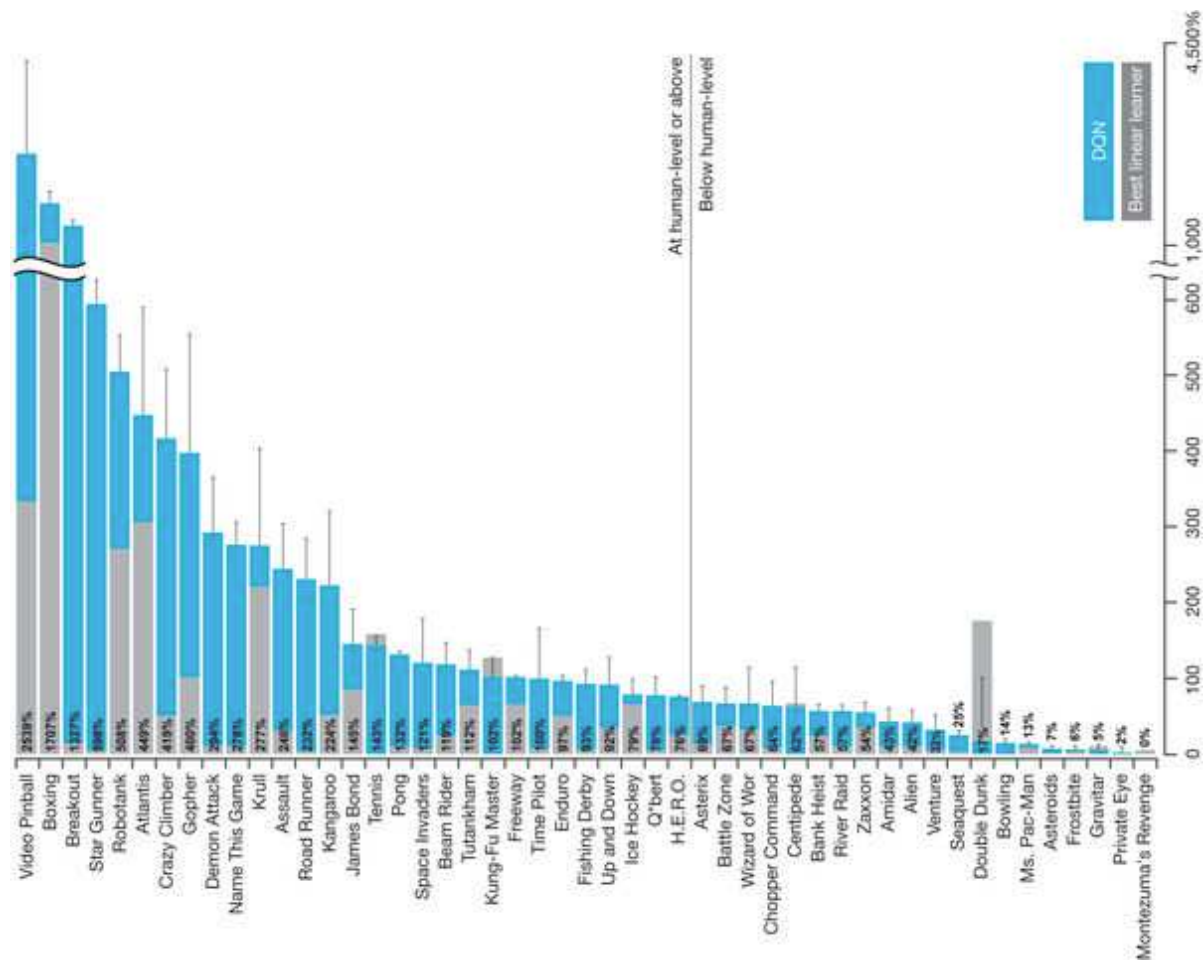
---

- 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

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

# Policy Gradients and 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
```

# 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

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

# Hill Climbing

---

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

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

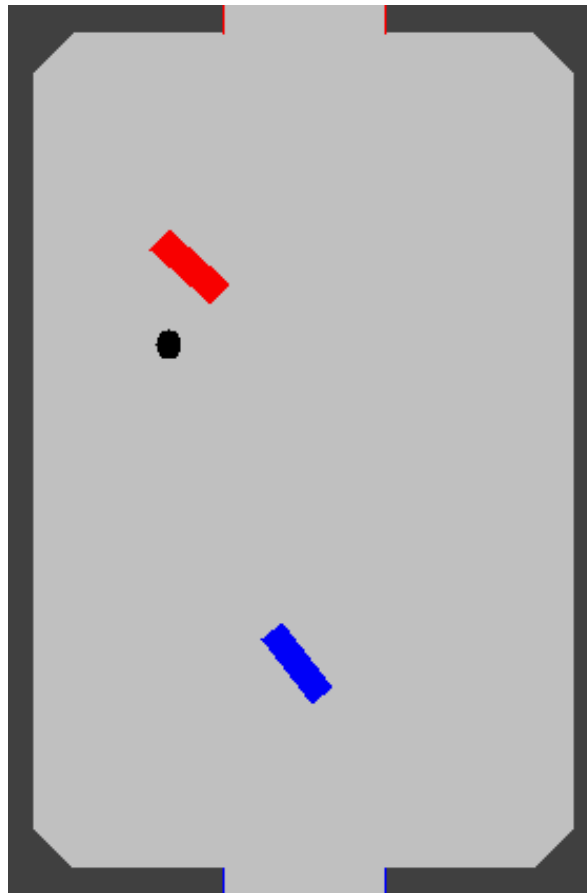
- champ and mutant play up to  $n$  games, with same game initial conditions (i.e. same seed for generating dice rolls)
- if mutant does “better” than champ,

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

- “better” means the mutant must score higher than the champ in the first game, and at least as high as the champ in each subsequent game.

# Simulated Hockey

---



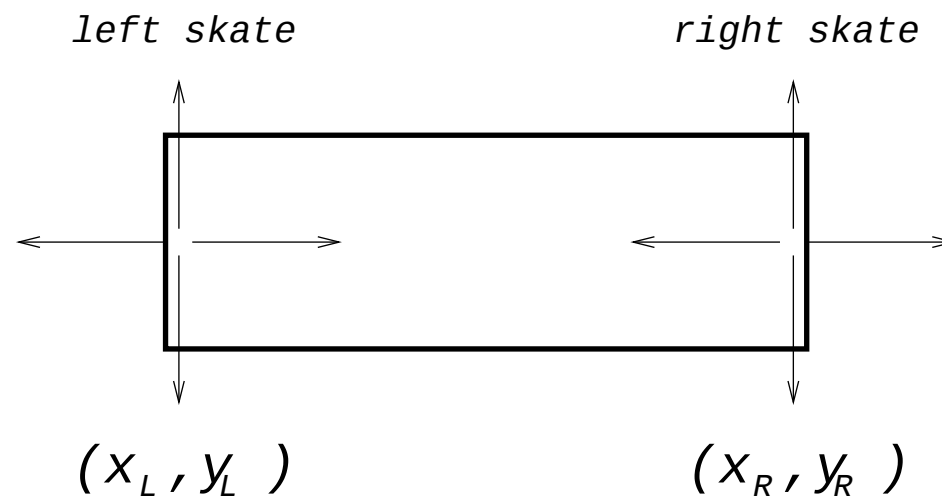
# Shock Physics

---

- rectangular rink with rounded corners
- near-frictionless playing surface
- “spring” method of collision handling
- frictionless puck (never acquires any spin)

# Shock Actuators

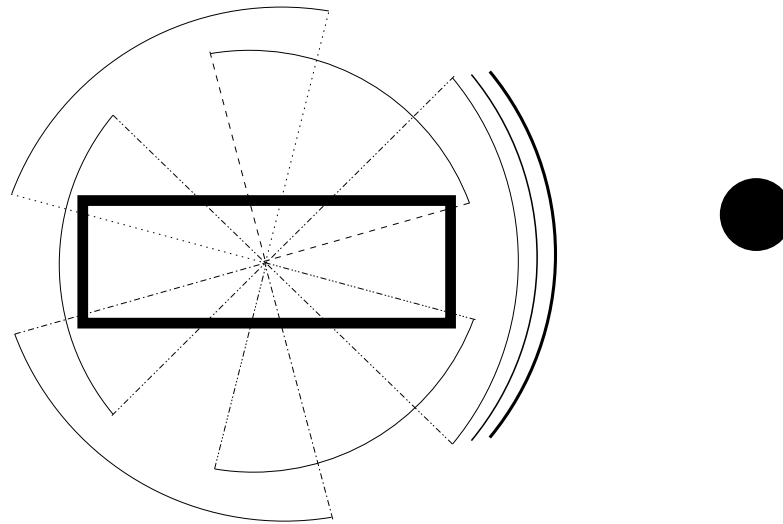
---



- a skate at each end of the vehicle with which it can push on the rink in two independent directions

# Shock Sensors

---



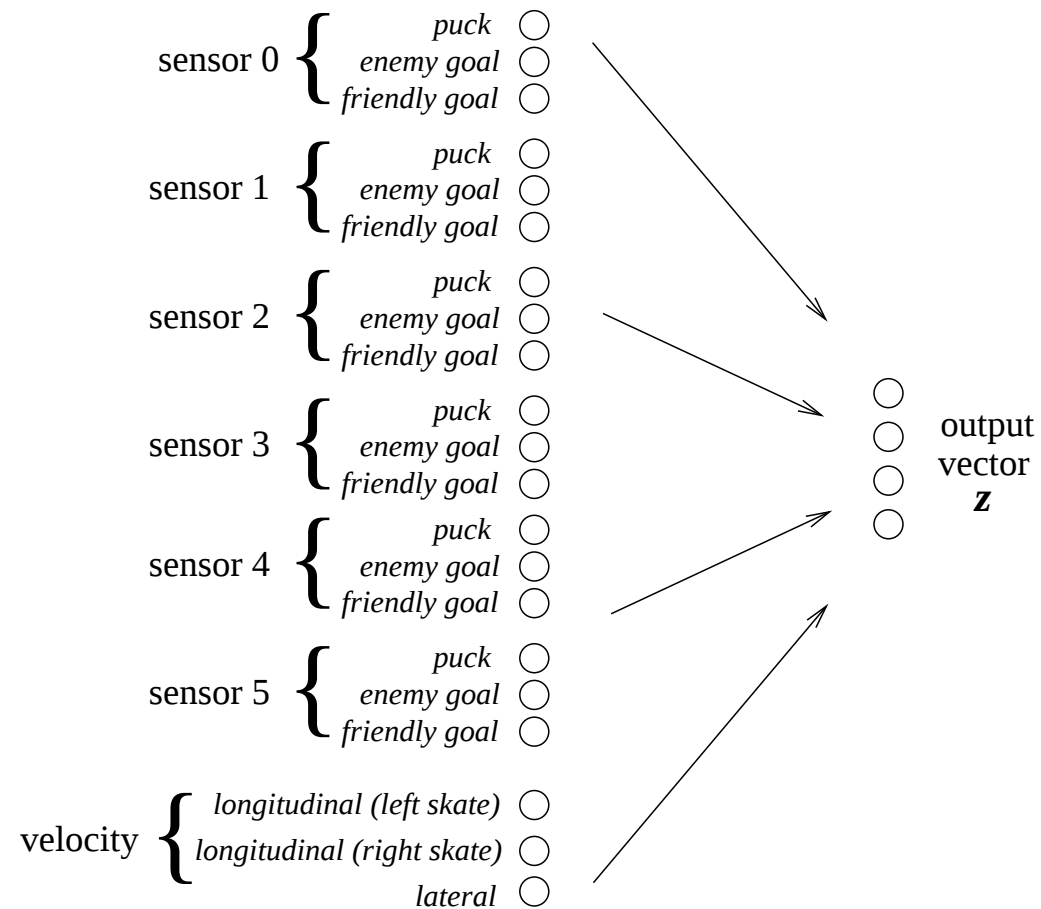
- 6 Braitenberg-style sensors equally spaced around the vehicle
- each sensor has an angular range of  $90^\circ$  with an overlap of  $30^\circ$  between neighbouring sensors

# Shock Inputs

---

- each of the 6 sensors responds to three different stimuli
  - ▶ ball / puck
  - ▶ own goal
  - ▶ opponent goal
- 3 additional inputs specify the current velocity of the vehicle
- total of  $3 \times 6 + 3 = 21$  inputs

# Shock Agent



# Shock Agent

---

- Perceptron with 21 inputs and 4 outputs
- total of  $4 \times (21 + 1) = 88$  parameters
- mutation = add Gaussian random noise to each parameter, with standard deviation 0.05
- $\alpha = 0.1$

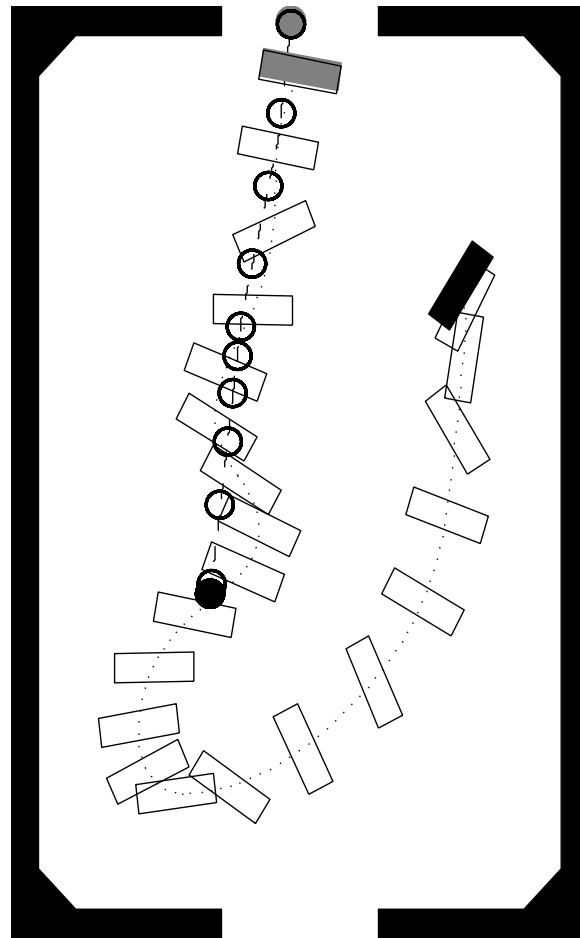
# Shock Task

---

- each game begins with a random “game initial condition”
  - ▶ random position for puck
  - ▶ random position and orientation for player
- each game ends with
  - ▶ +1 if puck  $\rightarrow$  enemy goal
  - ▶ -1 if puck  $\rightarrow$  own goal
  - ▶ 0 if time limit expires

# Evolved Behavior

---



## Evolutionary/Variational Methods

---

- initialize mean  $\mu = \{\mu_i\}_{1 \leq i \leq m}$  and standard deviation  $\sigma = \{\sigma_i\}_{1 \leq i \leq m}$
- for each trial, collect  $k$  samples from a Gaussian distribution

$$\theta_i = \mu_i + \eta_i \sigma_i \quad \text{where} \quad \eta_i \sim \mathcal{N}(0, 1)$$

- sometimes include “mirrored” samples  $\bar{\theta}_i = \mu_i - \eta_i \sigma_i$
- evaluate each sample  $\theta$  to compute score or “fitness”  $F(\theta)$
- update mean  $\mu$  by

$$\mu \leftarrow \mu + \alpha(F(\theta) - \bar{F})(\theta - \mu)$$

▶  $\alpha$  = learning rate,  $\bar{F}$  = baseline

- sometimes,  $\sigma$  is updated as well

# OpenAI Evolution Strategies

---

- Evolutionary Strategy with fixed  $\sigma$
- since only  $\mu$  is updated, computation can be distributed across many processors
- applied to Atari Pong, MuJoCo humanoid walking
- competitive with Deep Q-Learning on these tasks

# Methods for Updating Sigma

---

- Evolutionary Strategy

- ▶ select top 20% of samples and fit a new Gaussian distribution

- Variational Inference

- ▶ minimize Reverse KL-Divergence
- ▶ backpropagate differentials through network, or differentiate with respect to  $\mu_i, \sigma_i$

# Variational Inference

---

- let  $q(\theta)$  be the Gaussian distribution determined by  $\mu, \sigma$
- we want  $q(\theta)$  to be concentrated in regions where  $F(\theta)$  is high
- score function  $F(\theta)$  determines a Boltzmann (softmax) distribution

$$p_T(\theta) = \frac{e^{-\frac{1}{T}F(\theta)}}{Z}$$

- ▶  $T$  = temperature,  $Z$  = normalizing constant
- we can try to minimize the reverse Kullback-Leibler (KL) Divergence between  $q(\theta)$  and  $p_T(\theta)$

$$D_{\text{KL}}(q \| p_T) = \int_{\theta} q(\theta) (\log q(\theta) - \log p_T(\theta)) d\theta$$

# Variational Inference

---

$$\begin{aligned} D_{\text{KL}}(q || p_T) &= \int_{\theta} q(\theta) (\log q(\theta) - \log p_T(\theta)) d\theta \\ &= \frac{1}{T} \int_{\theta} q(\theta) (F(\theta) + T \log q(\theta) + T \log Z) d\theta \end{aligned}$$

- the last term  $T \log Z$  is constant, so its value is not important (in fact, an arbitrary baseline  $\bar{F}$  can be subtracted from  $F(\theta)$ )
- $T \log q(\theta)$  can be seen as a regularizing term which maintains some variation and prevents  $q(\theta)$  from collapsing to a single point
  - ▶ if we only update  $\mu$  and not  $\sigma$ , this term is not needed

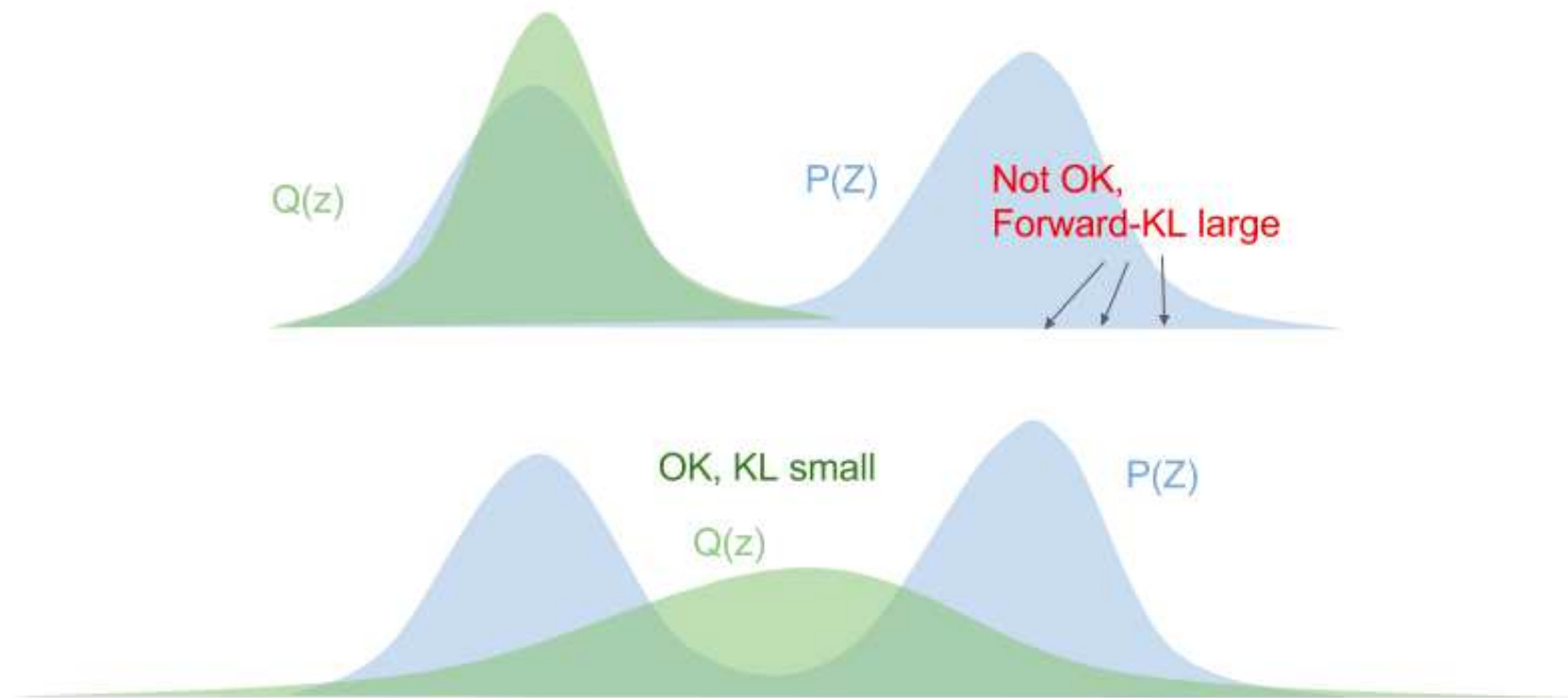
## KL-Divergence and Entropy

---

- the **entropy** of a distribution  $q()$  is  $H(q) = \int_{\theta} q(\theta)(-\log q(\theta))d\theta$
- in Information Theory,  $H(q)$  is the amount of information (bits) required to transmit a random sample from distribution  $q()$
- for a Gaussian distribution,  $H(q) = \sum_i \log \sigma_i$
- KL-Divergence  $D_{\text{KL}}(q \| p) = \int_{\theta} q(\theta)(\log q(\theta) - \log p(\theta))d\theta$
- $D_{\text{KL}}(q \| p)$  is the number of **extra** bits we need to transmit if we designed a code for  $p()$  but then the samples are drawn from  $q()$  instead.

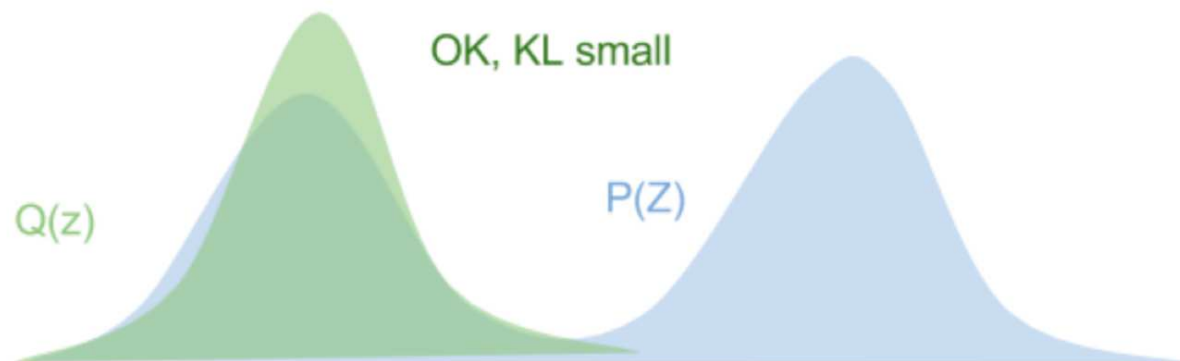
# Forward KL-Divergence

---



# Reverse KL-Divergence

---



# KL-Divergence

---

- KL-Divergence is used in some policy-based deep reinforcement learning algorithms such as Trust Region Policy Optimization (TPRO) (but we will not cover these in detail).
- KL-Divergence is also important in other areas of Deep Learning, such as Variational Autoencoders.

# 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>