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.)

Outline

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

- Diagram of the deep Q-network structure.

Q-Learning

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

- 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
  $$\left[ r_t + \gamma \max_b Q_w(s_{t+1}, b) - Q_w(s_t, a_t) \right]^2$$
  - 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
  \[ r_t + \gamma \max_b Q_w(s_{t+1}, b) - Q_w(s_t, a_t) \]
- removes temporal correlations by sampling from variety of game situations in random order
- makes it easier to parallelize the algorithm on multiple GPUs

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)

DQN Improvements

- Prioritised Replay
  - weight experience according to surprise
- Double Q-Learning
  - current Q-network $w$ is used to select actions
  - older Q-network $\overline{w}$ is used to evaluate actions
- Advantage Function
  - action-independent value function $V_w(s)$
  - action-dependent advantage function $A_w(s, a)$
  \[ Q(s, a) = V_w(s) + A_w(s, a) \]
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 \( \overline{w} \), 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 \( \overline{w} \) for evaluation; we therefore minimize
  \[
  [r_t + \gamma Q_{\overline{\pi}}(s_{t+1}, \text{argmax}_b Q_w(s_{t+1}, b)) - Q_w(s_t, a_t)]^2
  \]
- a new version of \( \overline{w} \) is periodically calculated from the distributed values of \( w \), and this \( \overline{w} \) is broadcast to all processors.

Policy Gradients and Actor-Critic

Recall:

\[
\nabla_\theta \text{fitness}(\pi_\theta) = E_{a_0} [ Q^{\pi_\theta}(s, a) \nabla_\theta \log \pi_\theta(a | s) ]
\]

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.

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:

\[
Q(s, a) = V^\pi(s) + A^\pi(s, a)
\]

Note that actions can be selected just using \( A^\pi(s, a) \), because

\[
\text{argmax}_b Q(s_{t+1}, b) = \text{argmax}_b A_w(s_{t+1}, b)
\]

Actor Critic Algorithm

for each trial

sample \( a_0 \) from \( \pi(a | s_0) \)

for each timestep \( t \) do

sample reward \( r_t \) from \( R(r_t | s_t, a_t) \)

sample next state \( s_{t+1} \) from \( \delta(s_t | s_t, a_t) \)

sample action \( a_{t+1} \) from \( \pi(a | s_{t+1}) \)

\[
\frac{dE}{dQ} = -[r_t + \gamma Q_w(s_{t+1}, a_{t+1}) - Q_w(s_t, a_t)]
\]

\( \theta \leftarrow \theta + \eta \nabla_\theta Q_w(s_t, a_t) \log \pi_\theta(a_t | s_t) \)

\( w \leftarrow w - \eta_w \frac{dE}{dQ} \nabla_w Q_w(s_t, a_t) \)

end

end
### Advantage Actor Critic

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

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

In the actor-critic framework, $r_{total}$ 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 $V_\upsilon(s)$, in which case the $Q$ function is replaced by the advantage function

$$A_\upsilon(s, a) = Q(s, a) - V_\upsilon(s)$$

### Asynchronous Advantage Actor Critic

- use policy network to choose actions
- learn a parameterized Value function $V_\upsilon(s)$ by TD-Learning
- estimate Q-value by n-step sample
  $$Q(s_t, a_t) = r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{n-1} r_{t+n} + \gamma^n V_\upsilon(s_{t+n})$$
- update policy by
  $$\theta \leftarrow \theta + \eta_\theta [Q(s_t, a_t) - V_\upsilon(s_t)] \nabla \theta \log \pi_\theta(a_t | s_t)$$
- update Value function by minimizing
  $$[Q(s_t, a_t) - V_\upsilon(s_t)]^2$$

### 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