Reinforcement Learning

Deep Reinforcement Learning

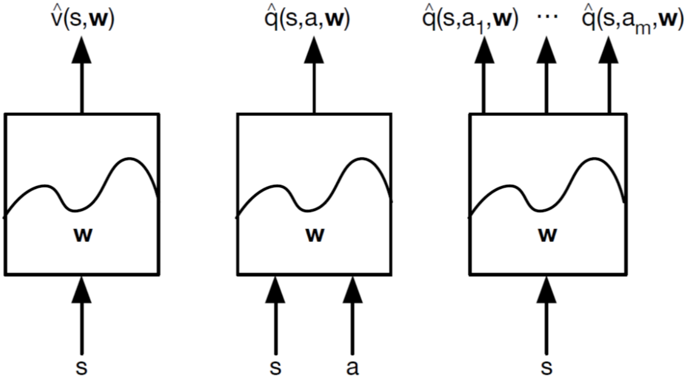
Mario Martin

CS-UPC

April 11, 2018

Deep Neural Networks

Use of Neural Networks for regression



Neural Fitted Q-learning

Fitted Q-learning

Given $\mathcal{D}$ of size T with examples $(s_t, a_t, r_{t+1}, s_{t+1})$, and regression algorithm, set N to zero and $Q_N(s, a) = 0$ for all a and s

repeat

$N \leftarrow N + 1$

Build training set $TS = \{\langle (s_t, a_t), r_{t+1} + \gamma \max_a Q_N(s_{t+1}, a) \rangle\}_{t=1}^{T}$

$Q_{N+1} \leftarrow$ regression algorithm on TS

until $Q_N \approx Q_{N+1}$ or $N >$ limit

return π based on greedy evaluation of Q_N

Neural Fitted Q-learning

Neural Fitted Q-learning: Wrong version. Why?

Initialize weights θ for NN for regression
Collect $\mathcal{D}$ of size T with examples $(s_t, a_t, r_{t+1}, s_{t+1})$
repeat
  Sample $\mathcal{B}$ mini-batch of $\mathcal{D}$
  $\theta \leftarrow \theta - \alpha \sum_{t \in \mathcal{B}} \frac{\partial Q_\theta}{\partial \theta}(s_t, a_t) - (Q_\theta(s_t, a_t) - [r_{t+1} + \gamma \max_{a'} Q_\theta(s_{t+1}, a')])$
until convergence on learning or maximum number of steps
return π based on greedy evaluation of Q_θ

- Does not work well
- It's not a Batch method. Can you see why?

Neural Fitted Q-learning (Riedmiller, 2005)

Neural Fitted Q-learning

Initialize weights θ for NN for regression
Collect $\mathcal{D}$ of size T with examples $(s_t, a_t, r_{t+1}, s_{t+1})$
repeat
  $\theta' \leftarrow \theta$
  repeat
    Sample $\mathcal{B}$ mini-batch of $\mathcal{D}$
    $\theta \leftarrow \theta - \alpha \sum_{t \in \mathcal{B}} \frac{\partial Q_\theta}{\partial \theta}(s_t, a_t) - (Q_\theta(s_t, a_t) - [r_{t+1} + \gamma \max_{a'} Q_{\theta'}(s_{t+1}, a')])$
  until convergence on learning or maximum number of steps
until maximum limit iterations
return π based on greedy evaluation of Q'_θ

- Notice target does not change during supervised regression

Neural Fitted Q-learning: Another version

- That works, however the update of parameters is not smooth
- Alternative version to avoid moving target

Fitted Q-learning avoiding moving target

Initialize weights θ for NN for regression
Collect $\mathcal{D}$ of size T with examples $(s_t, a_t, r_{t+1}, s_{t+1})$
repeat
  Sample $\mathcal{B}$ mini-batch of $\mathcal{D}$
  $\theta \leftarrow \theta - \alpha \sum_{t \in \mathcal{B}} \frac{\partial Q_\theta}{\partial \theta}(s_t, a_t) - (Q_\theta(s_t, a_t) - [r_{t+1} + \gamma \max_{a'} Q_{\theta'}(s_{t+1}, a')])$
  $\theta' \leftarrow \tau\theta' + (1 - \tau)\theta$
until maximum limit iterations
return π based on greedy evaluation of Q'_θ

- Value of τ close to one (f.i. $\tau = 0.999$) reduces the "speed" of the moving target.

How to get the data?

- So now, we have learning stabilized just any batch method but using NN.
- However, now there is the problem of dependence of dataset $\mathcal{D}$. How we obtain the data?
- Data can be obtained using a random policy, but we want to minimize error on states visited by the policy!

$$L(\theta) = \mathbb{E}_\pi\left[(V^\pi(s) - V_\theta(s))^2\right] = \sum_{s \in \mathcal{S}} \mu^\pi(s) \left[V^\pi(s) - V_\theta(s)\right]^2$$

where $\mu^\pi(s)$ is the time spent in state s while following π

How to get the data?

- Data has to be generated by the policy
- But it also has to be probabilistic (to ensure exploration)
- So, collect data using the policy and add them to $\mathcal{D}$
- Also remove old data from $\mathcal{D}$.
  - Limit the size of the set
  - Remove examples obtained using old policies
- So, collect data using a *buffer* of limited size (we call **replay buffer**).

When to get the data?

Batch Q-learning with replay buffer and target network

Initialize weights θ for NN for regression
Collect $\mathcal{D}$ of size T with examples $(s_t, a_t, r_{t+1}, s_{t+1})$ using random policy
repeat
  $\theta' \leftarrow \theta$
  repeat
    Collect M experiences following ϵ-greedy procedure and add them to **buffer** $\mathcal{D}$
    repeat
      Sample $\mathcal{B}$ mini-batch of $\mathcal{D}$
      $\theta \leftarrow \theta - \alpha \sum_{t \in \mathcal{B}} \frac{\partial Q_\theta}{\partial \theta}(s_t, a_t) - (Q_\theta(s_t, a_t) - [r_{t+1} + \gamma \max_{a'} Q_{\theta'}(s_{t+1}, a')])$
    until maximum number of steps K
  until maximum number of iterations N
until maximum limit iterations
return π based on greedy evaluation of Q'_θ

DQN algorithm (Mnih, et al. 2015)

- **Deep Q-Network** algorithm breakthrough
  - In 2015, Nature published DQN algorithm.
  - It takes profit of "then-recent" *Deep Neural Networks* and, in particular, of *Convolutional NNs* so successful for vision problems
  - Applied to Atari games directly from pixels of the screen (no hand made representation of the problem)
  - Very successful on a difficult task, surpassing in some cases human performance
- It is basically the previous algorithm with $K = 1$, and $M = 1$ that is applied *on the current state.*
- It goes back to incremental learning

DQN algorithm (Mnih, et al. 2015)

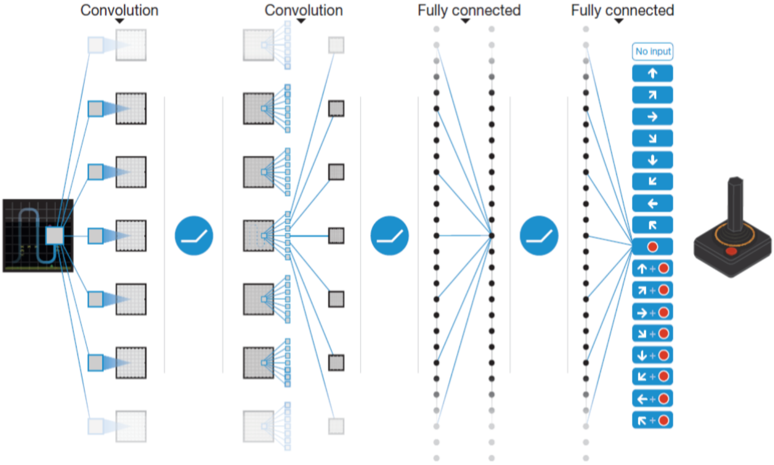
DQN algorithm

Initialize weights θ for NN for regression
Set s to initial state, and k to zero
repeat
  Choose a from s using policy π_θ derived from Q_θ (e.g., ϵ-greedy)
  $k \leftarrow k + 1$
  Execute action a, observe r, s', and add $\langle s, a, r, s' \rangle$ to buffer $\mathcal{D}$
  Sample $\mathcal{B}$ mini-batch of $\mathcal{D}$
  $\theta \leftarrow \theta - \alpha \sum_{t \in \mathcal{B}} \frac{\partial Q_\theta}{\partial \theta}(s_t, a_t) - (Q_\theta(s_t, a_t) - [r_{t+1} + \gamma \max_{a'} Q_{\theta'}(s_{t+1}, a')])$
  if k==N **then**
    $\theta' \leftarrow \theta$
    $k \leftarrow 0$
  end if
until maximum limit iterations
return π based on greedy evaluation of Q'_θ

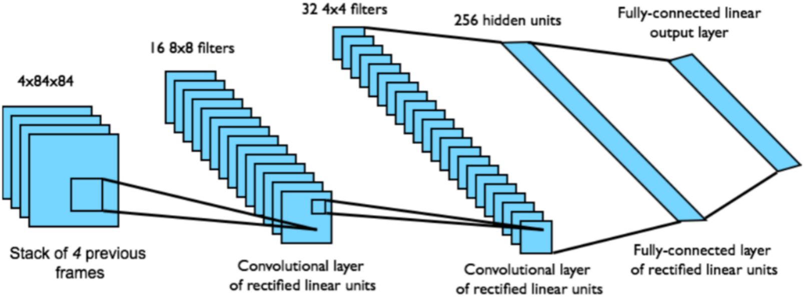
DQN algorithm on Atari

- End-to-end learning of values Q(s; a) from pixels:
  - **State:** Input state s is stack of raw pixels from last 4 frames
  - **Actions:** Output is $Q(s, a)$ value for each of 18 joystick/button positions
  - **Reward:** Reward is direct change in score for that step
- Network architecture and hyper-parameters **fixed across all games**, No tuning!
- Clipping reward -1,0,1 to avoid problem of different magnitudes of score in each game

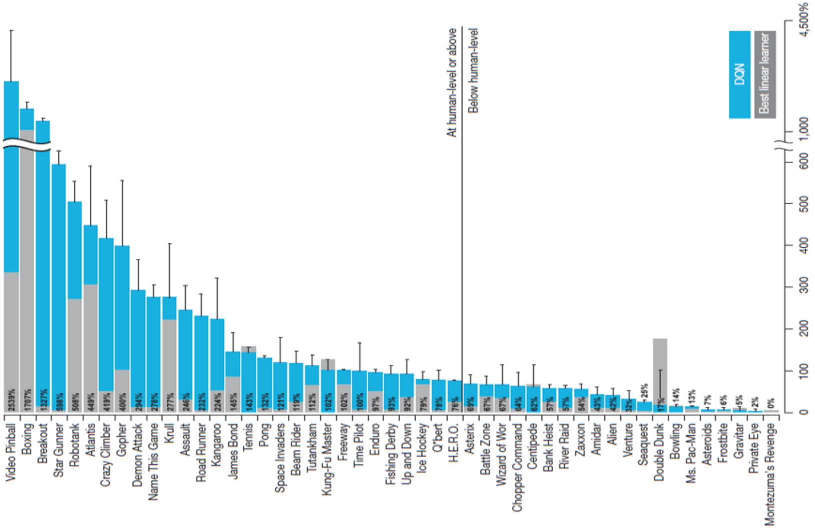
DQN algorithm on Atari



DQN algorithm on Atari



DQN algorithm on Atari



DQN algorithm on Atari

DQN algorithm on Atari

- What is the effect of each trick on Atari games?

DQN

	Q-learning	Q-learning + Target Q	Q-learning + Replay	Q-learning + Replay + Target Q
Breakout	3	10	241	**317**
Enduro	29	142	831	**1006**
River Raid	1453	2868	4103	**7447**
Seaquest	276	1003	823	**2894**
Space Invaders	302	373	826	**1089**

Overestimates: Double Q-learning

Double Q-learning (Hasselt, et al. 2015)

- **Problem of overestimation of Q values**.
- We use "max" operator to compute the target in the minimization of:

$$L(s,a) = (Q(s,a) - (r - \gamma \max_{a'} Q(s',a')))^2$$

- Surprisingly here is a problem.
  1. Suppose $Q(s',a')$ is 0 for all actions, so $Q(s,a)$ should be r.
  2. But $\gamma \max_{a'} Q(s',a') \geq 0$ because stochasticity and use of the max operator.
  3. So estimation $Q(s,a) \geq r$, overestimating true value
  4. All this because for max operator:

$$\mathbb{E}[\max_{a'} Q(s',a')] \geq \max_{a'} \mathbb{E}[Q(s',a')]$$

- This overestimation is propagated.

Double Q-learning

- Solution (Hasselt, 2010): Train 2 action-value functions: Q_A and Q_B, and compute argmax with the other network
- Do Q-learning on both, but
  - never on the same time steps (Q_A and Q_B are independent)
  - *pick Q_A or Q_B at random to be updated on each step*
- Notice that:

$$r + \gamma \arg\max_{a'} Q(s, a') = r + \gamma Q(s, \arg\max_{a'} Q(s', a'))$$

- When updating one network, use the values of the other network:

$$Q_A(s, a) \leftarrow r + \gamma Q_B(s, \arg\max_{a'} Q_A(s', a'))$$

$$Q_B(s, a) \leftarrow r + \gamma Q_A(s, \arg\max_{a'} Q_B(s', a'))$$

- Idea is that they should compensate mistakes of each other because they will be independent. When one network overestimate, probably, the other no, so they mutually cancel overestimation

Double DQN (Hasselt, et al. 2015)

- In DQN, in fact, we have 2 value functions: Q_θ and $Q_{\theta'}$
- so, no new to add another one:
  - Current Q-network θ is used to select actions
  - Older Q-network θ' is used to evaluate actions
- Update in Double-DQN (Hasselt, et al. 2015):

$$Q_\theta(s,a) \leftarrow r + \gamma \overbrace{Q_{\theta'}(s, \underbrace{\arg\max_{a'} Q_\theta(s',a')}_{\text{Action Selection}})}^{\text{Action Evaluation}}$$

- Works well in practice.

Prioritized Experience Replay

Prioritized Experience Replay (Schaul, et al. 2016)

- Idea: sample transitions from replay buffer more cleverly
- Those states with poorer estimation in buffer will be selected with preference for update
- We will set probability for every transition. Let's use the absolute value of TD-error of transition as a probability!

$$p_i = |\text{TD-error}_i| = |Q_{\theta'}(s_i, a_i) - (r_i + \gamma Q_{\theta'}(s_i, \arg\max_{a'} Q_\theta(s_{i+1}, a'))|$$

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

where *alpha* is parameter (*alpha*=0 is uniform probability)

Prioritized Experience Replay (Schaul, et al. 2016)

- Do you see any problem?
- Now transitions are no i.i.d. and therefore we introduce a bias.
- Solution: we can correct the bias by using **importance-sampling** weights

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)} \right)^{\beta}$$

- For numerical reasons, we also normalize weights by $\max_i w_i$
- When we put transition into experience replay, we set it to maximal priority $p_t = \max_{i<t} p_i$

Prioritized Experience Replay (Schaul, et al. 2016)

Algorithm 1 Double DQN with proportional prioritization

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, budget T.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ **to** T **do**
  Observe S_t, R_t, γ_t
  Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
  if $t \equiv 0 \mod K$ **then**
    for $j = 1$ **to** k **do**
      Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
      Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
      Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
      Update transition priority $p_j \leftarrow |\delta_j|$
      Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
    end for
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
    From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
  end if
  Choose action $A_t \sim \pi_\theta(S_t)$
end for

Dueling Network Architectures

Dueling Network Architectures (Wang, et al. 2016)

- Until now, use of generic NN for regression of Q-value function
- Now, present new Deep Architecture specific for RL
- First lets introduce concept of *Advantage function:*

$$A(s,a) = Q(s,a) - V(s)$$

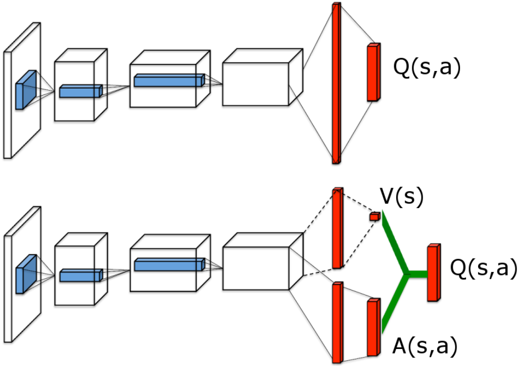
- So,

$$Q(s,a) = A(s,a) + V(s)$$

- Intuitively, Advantage function is relative measure of importance of each action

Dueling Network Architectures (Wang, et al. 2016)

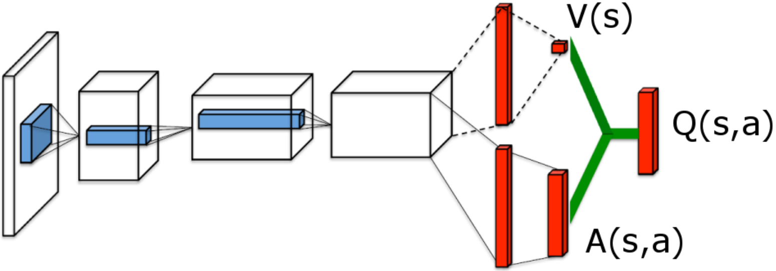
- Dueling network:



- Intuitive idea is that now we don't learn $Q(s,a)$ independently but share part that is $V(s)$ that improves generalization across actions

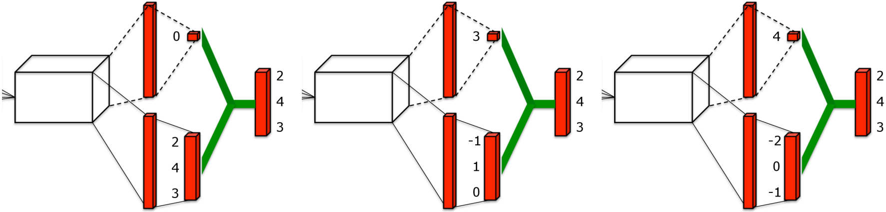
Dueling Network Architectures (Wang, et al. 2016)

- We have now 3 sets of parameters:
  - θ: Usual weights of NN until red section
  - β: Weights to compute $V(s)$
  - α: Weights to compute $A(s,a)$
- Green part computes $A(s,a) + V(s)$



Dueling Network Architectures (Wang, et al. 2016)

- However, there is a problem: one extra freedom degree!
- Example:



Dueling Network Architectures (Wang, et al. 2016)

- Which is the correct one? Notice that:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} \; Q^*(s,a)$$

and that,

$$V^*(s) = \max_{a \in \mathcal{A}} Q^*(s,a)$$

- So,

$$\begin{aligned}
\max_{a \in \mathcal{A}} A(s,a) &= \max_{a \in \mathcal{A}} (Q(s,a) - V(s)) \\
&= \max_{a \in \mathcal{A}} Q(s,a) - V(s) \\
&= 0
\end{aligned}$$

- Of course, for actions $a \neq a^* \; A(s,a) \leq 0$

Dueling Network Architectures (Wang, et al. 2016)

- **Solution**: require $\max_a A(s,a)$ to be equal to zero!
- So the Q-function computes as:

$$Q_{\theta,\alpha,\beta}(s,a) = V_{\theta,\beta}(s) + \left(A_{\theta,\alpha}(s,a) - \max_{a' \in \mathcal{A}} A_{\theta,\alpha}(s,a') \right)$$

- In practice, the authors propose to implement

$$Q_{\theta,\alpha,\beta}(s,a) = V_{\theta,\beta}(s) + \left(A_{\theta,\alpha}(s,a) - \frac{1}{|A|} \sum_{a' \in \mathcal{A}} A_{\theta,\alpha}(s,a') \right)$$

- This variant increases stability of the optimization because now depends on softer measure (*average* instead of *max*)
- Now Q-values loses original semantics, but it not important. The important thing is a *reference* between actions

Multi-step learning

Multi-step learning

- Idea: instead of using TD(0), use n-steps estimators like we described in lecture 2
- In buffer we should store experiences:

$$\left\langle s_t, a_t, r_t, \sum_{i=0}^{n} \gamma^i r_{t+1} + \gamma^n \max_{a'} Q_{\theta'}(s_{t+n}, a') \right\rangle$$

Multi-step learning

- Idea: instead of using TD(0), use n-steps estimators like we described in lecture 2
- In buffer we should store experiences:

$$\left\langle s_t, a_t, r_t, \sum_{i=0}^{n} \gamma^i r_{t+1} + \gamma^n \max_{a'} Q_{\theta'}(s_{t+n}, a') \right\rangle$$

- Again, there is a problem!
- Only correct when learning on-policy! (not an issue when $n = 1$)
- How to fix that?
  - Ignore the problem (often works well)
  - Dynamically choose n to get only on-policy data (Store data until not policy action taken)
  - Use importance sampling (Munos et al, 2016)

Rainbow: Combining Improvements in Deep Reinforcement Learning

Rainbow (Hessel et al. 2017)

- Idea: Let's try to investigate how each of the different improvements over DQN help to improve performance on the Atari games
- Over DQN, they added the following modifications:
  - Double Q-learning
  - Prioritized replay
  - Dueling networks
  - Multi-step learning
  - Distributional RL
  - Noisy Nets
- They perform an ablation study where over the complete set of improvement, they disable one an measure the performance

Rainbow (Hessel et al. 2017)

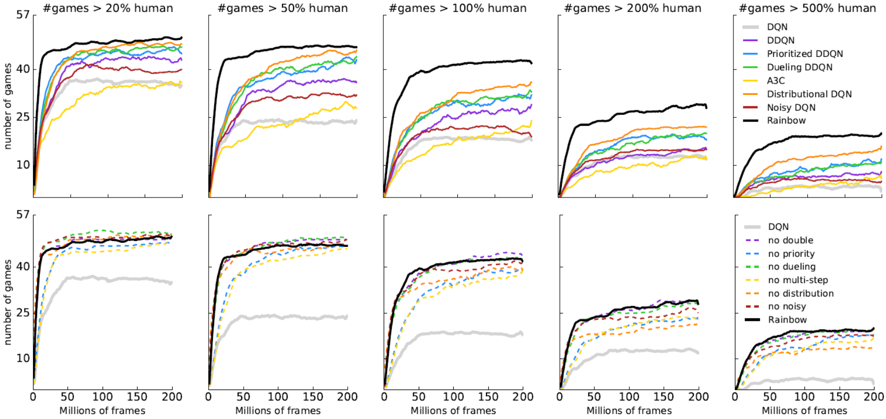
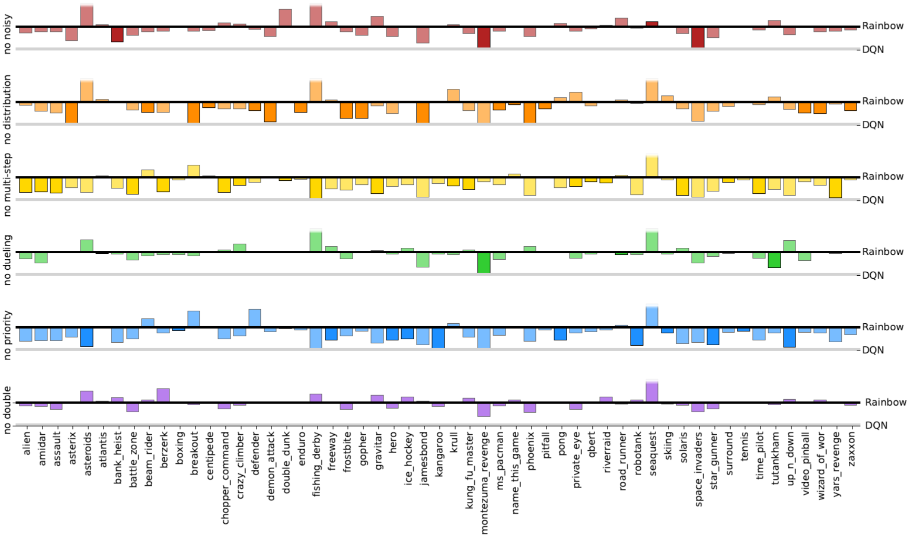


Figure 2: Each plot shows, for several agents, the number of games where they have achieved at least a given fraction of human performance, as a function of time. From left to right we consider the 20%, 50%, 100%, 200% and 500% thresholds. On the first row we compare Rainbow to the baselines. On the second row we compare Rainbow to its ablations.

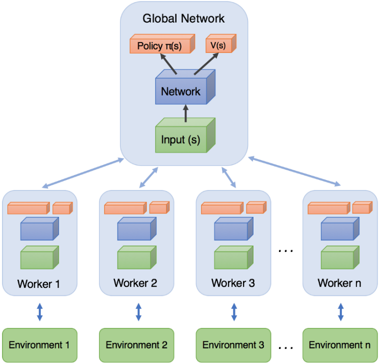
Rainbow (Hessel et al. 2017)



Asynchronous Q-learning

Asynchronous Q-learning (Mnih et al. 2016)

- Idea: Parallelize learning with several workers



- After some time steps, the worker passes gradients to the global network

Asynchronous Q-learning (Mnih et al. 2016)

Algorithm 1 Asynchronous one-step Q-learning - pseudocode for each actor-learner thread.

```
// Assume global shared θ, θ⁻, and counter T = 0.
Initialize thread step counter t ← 0
Initialize target network weights θ⁻ ← θ
Initialize network gradients dθ ← 0
Get initial state s
repeat
    Take action a with ε-greedy policy based on Q(s, a; θ)
    Receive new state s' and reward r
    y = r                              for terminal s'
        r + γ max_a' Q(s', a'; θ⁻)     for non-terminal s'
    Accumulate gradients wrt θ: dθ ← dθ + ∂(y − Q(s, a; θ))² / ∂θ
    s = s'
    T ← T + 1 and t ← t + 1
    if T mod I_target == 0 then
        Update the target network θ⁻ ← θ
    end if
    if t mod I_AsyncUpdate == 0 or s is terminal then
        Perform asynchronous update of θ using dθ.
        Clear gradients dθ ← 0.
    end if
until T > T_max
```

where

$$y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s' \end{cases}$$

$$d\theta \leftarrow d\theta + \frac{\partial (y - Q(s,a;\theta))^2}{\partial \theta}$$

Faster Deep RL by optimality tightening

Faster Deep RL by optimality tightening (He, et al. 2016)

- From Bellman equation we will obtain a bound for a given Q-value:

$$
\begin{aligned}
Q(s_t, a_t) &= \mathbb{E}\left[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a')\right] \\
&\geq \mathbb{E}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^k \max_{a'} Q(s_{t+k+1}, a')\right] \\
&\geq \mathbb{E}\left[\underbrace{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^k Q(s_{t+k+1}, a_{t+k+1})}_{\text{Lower bound} \quad L^{max}}\right]
\end{aligned}
$$

Faster Deep RL by optimality tightening (He, et al. 2016)

- Also we can do this backwards in time. Notice that we had

$$Q(s_t, a_t) \quad \geq \quad r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^k Q(s_{t+k+1}, a_{t+k+1})$$

- Changing indexes

$$Q(s_{t-k-1}, a_{t-k-1}) \quad \geq \quad r_{t-k} + \gamma r_{t-k+1} + \gamma^2 r_{t-k+2} + \ldots + \gamma^k Q(s_t, a_t)$$

- So,

$$\begin{aligned} Q(s_{t-k-1}, a_{t-k-1}) - r_{t-k} - \gamma r_{t-k+1} - \gamma^2 r_{t-k+2} - \ldots \quad &\geq \quad \gamma^k Q(s_t, a_t) \\ \gamma^{-k}\left[Q(s_{t-k-1}, a_{t-k-1}) - r_{t-k} - \gamma r_{t-k+1} - \gamma^2 r_{t-k+2} - \ldots\right] \quad &\geq \quad Q(s_t, a_t) \end{aligned}$$

- So, finally we have an upper bound:

$$Q(s_t, a_t) \quad \leq \quad \underbrace{\gamma^{-k} Q(s_{t-k}, a_{t-k}) - \gamma^{-k} r_{t-k} - \gamma^{-(k-1)} r_{t-(k-1)} - \ldots - \gamma r_{t-1}}_{\text{Upper bound } U^{min}}$$

Faster Deep RL by optimality tightening (He, et al. 2016)

- And now we can modify our loss function using these bounds:

$$
\begin{aligned}
y &= r + \gamma Q_{\theta'}(s', \arg\max_a Q_\theta(s', a)) \\
L(\theta) &= \mathbb{E}\left[(Q_\theta(s,a) - y)^2 + \lambda(L^{max} - Q_\theta(s,a))_+^2 + \lambda(U^{min} - Q_\theta(s,a))_+^2\right]
\end{aligned}
$$

  where λ is a penalization parameter like C in SVMs
- Eq, minimize the original Bellman error. but also *penalizes breaking the bounds*
- Accelerates over an order of magnitude with respect original DQN in number of experiences needed for learning

Practical tricks

Practical tricks

- DQN is more reliable on some tasks than others. Test your implementation on reliable tasks like Pong and Breakout: if it doesn't achieve good scores, something is wrong.
- Large replay buffers improve robustness of DQN, and memory efficiency is key.
- SGD can be slow .. rely on RMSprop (or any new optimizer)
- Convolutional models are more ecient then MLPs
- DQN uses action repeat set to 4 (because fps too high - speeds training time)
- DQN receives 4 frames of the game at a time (grayscale)
- ϵ is aneallled from 1 to .1

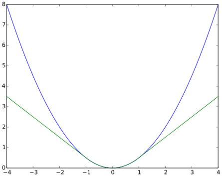
Practical tricks

- Patience. Training takes time (roughly hours to day on GPU training to see improvement)
- Always use Double DQN (3 lines of difference from DQN)
- Learning rate scheduling is benecial. Try high learning rates in initial exploration period.
- *Exploration* is key: Try non-standard exploration schedules.
- Always run at least two different seeds when experimenting

Practical tricks

- Bellman errors can be big. Clip gradients or use Huber loss on Bellman error

$$L_\delta(y, f(x)) = \begin{cases} \frac{(y-f(x))^2}{2}, & \text{when } |y - f(x)| \leq \delta \\ \delta|y - f(x)| - \frac{\delta^2}{2}, & \text{otherwise} \end{cases}$$



- Very large γ or set it to 1 to avoid myopic reward (very large sequences before reward)
- n-steps return **helps** but careful