Reinforcement Learning

Function approximation

Mario Martin

Mario Martin - CS-UPC

February 11, 2026

Goal of this lecture

- Methods we have seen so far work well when we have a *tabular* representation for each state, that is, when we represent value function with a lookup table.
- This is not reasonable on most cases:
  - In Large state spaces: There are too many states and/or actions to store in memory (f.i. Backgammon: 10^{20} states, Go 10^{170} states)
  - and in continuous state spaces (f.i. robotic examples)
- In addition, we want to generalize from/to similar states to speed up learning. It is too slow to learn the value of each state individually.

Goal of this lecture

- Methods we have seen so far work well when we have a *tabular* representation for each state, that is, when we represent value function with a lookup table.
- This is not reasonable on most cases:
  - In Large state spaces: There are too many states and/or actions to store in memory (f.i. Backgammon: 10^{20} states, Go 10^{170} states)
  - and in continuous state spaces (f.i. robotic examples)
- In addition, we want to generalize from/to similar states to speed up learning. It is too slow to learn the value of each state individually.

Goal of this lecture

- Methods we have seen so far work well when we have a *tabular* representation for each state, that is, when we represent value function with a lookup table.
- This is not reasonable on most cases:
  - In Large state spaces: There are too many states and/or actions to store in memory (f.i. Backgammon: 10^{20} states, Go 10^{170} states)
  - and in continuous state spaces (f.i. robotic examples)
- In addition, we want to generalize from/to similar states to speed up learning. It is too slow to learn the value of each state individually.

Goal of this lecture

- We'll see now methods to learn policies for large state spaces by using *function approximation to estimate value functions*:

$$\begin{aligned} V_\theta(s) &\approx V^\pi(s) \qquad (1)\\ Q_\theta(s,a) &\approx Q^\pi(s,a) \qquad (2)\end{aligned}$$

- θ is the set of parameters of the function approximation method (with size much lower than $|S|$)
- Function approximation allow to generalize from seen states to unseen states and to save space.
- Now, instead of storing V values, we will update θ parameters using MC or TD learning so they fulfill (1) or (2).

Which Function Approximation?

- There are many function approximators, e.g.
  - Artificial neural network
  - Decision tree
  - Nearest neighbor
  - Fourier/wavelet bases
  - Coarse coding
- In principle, any function approximator can be used. However, the choice may be affected by some properties of RL:
  - Experience is not i.i.d. Agents action affect the subsequent data it receives
  - During control, value function V(s) changes with the policy (non-stationary)

Incremental methods

Which Function Approximation?

- Incremental methods allow to directly apply the control methods of MC, Q-learning and Sarsa, that is, back up is done using *"on-line"* sequence of data of the trial reported by the agent following the policy.
- Most popular method in this setting is **gradient descent**, because it adapts to changes in the data (non-stationary condition)

Gradient Descent

- Let $L(\theta)$ be a differentiable function of parameter vector θ, we want to minimize
- Define the gradient of $L(\theta)$ to be:

$$\nabla_\theta L(\theta) = \begin{bmatrix} \frac{\partial L(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial L(\theta)}{\partial \theta_n} \end{bmatrix}$$

- To find a local minimum of $L(\theta)$, gradient descent method adjust the parameter in the direction of negative gradient:

$$\Delta\theta = -\frac{1}{2}\alpha \nabla_\theta L(\theta)$$

where is a stepsize parameter

Value Function Approx. by SGD

Minimizing *Loss function* of the approximation

Goal: Find parameter vector θ minimizing mean-squared error between approximate value function $V_\theta(s)$ and true value function $V^\pi(s)$

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

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

- Gradient descent finds a *local* minimum:

$$\Delta\theta = \frac{1}{2}\alpha(V^\pi(s) - V_\theta(s))\,\nabla_\theta V_\theta(s)$$

TD prediction for FA and SGD

- In TD(0) we use Q of *next state* to estimate Q on the *current state* using Bellman equations. So, in general,

$$\begin{aligned}\Delta\theta_i &= \alpha(\qquad Q^\pi(s,a) \qquad - Q_\theta(s,a))\nabla_\theta Q_\theta(s,a) \\ &= \alpha(r + \gamma Q_\theta(s', \pi(s')) - Q_\theta(s,a))\nabla_\theta Q_\theta(s,a)\end{aligned}$$

- So. it seems direct to apply FA to Q-learning

Incremental Q-learning with FA

Q-learning with FA

```
initialize parameters $\theta$ arbitrarily (e.g. $\theta = 0$)
for each episode do
    Choose initial state $s$
    repeat
        Choose $a$ from $s$ using policy $\pi_\theta$ derived from $Q_\theta$ (e.g., $\epsilon$-greedy)
        Execute action $a$, observe $r$, $s'$
        $\theta \leftarrow \theta + \alpha\,(r + \gamma\,\max_{a'} Q(s', a')) - Q_\theta(s, a))\nabla_\theta Q_\theta(s, a))$
        $s \leftarrow s'$
    until $s$ is terminal
end for
```

FA and Q-learning

- Sounds good ... doesn't work
- Do not blame Neural Networks and local minimum. Even with linear function approximation, it does not work
- Causes?

FA and Q-learning

- Sounds good ... doesn't work
- Do not blame Neural Networks and local minimum. Even with linear function approximation, it does not work
- Causes?

FA and Q-learning

- Sounds good ... doesn't work
- Do not blame Neural Networks and local minimum. Even with linear function approximation, it does not work
- Causes?

Problems with incremental Q-learning with FA

Caution!

- Notice that TD targets are not independent of parameters. In TD(0):

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

  depends of θ

- Bootstrapping methods are not true gradient descent[a]: they take into account the effect of changing θ on the estimate, **but ignore its effect on the target**. This produces the effect of the **moving target**.

[a]They include only a part of the gradient and, accordingly, we call them *semi-gradient methods.*

Problems with incremental Q-learning with FA

Essence of off-policy learning.

```
repeat
    Choose $a$, execute it and observe $r$ and $s'$ $(s, a, r, s')$ using any probabilistic policy
    $\theta \leftarrow \theta + \alpha\,(r + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s, a))\nabla_\theta Q_\theta(s, a))$
    $s \leftarrow s'$
until $s$ is terminal
```

- Several problems with incremental off-policy TD learning
  1. SGD does not converge because gradient does not follow true gradient. Target value is always changing and SGD does not converge
  2. Data is not even close to iid (it is strongly correlated) so another problem for SGD convergence
- How to solve these problems?

Batch methods

Batch Reinforcement Learning

- Gradient descent is simple and appealing
  - It is computationally efficient (one update per sample)
  - ... But it is not sample efficient (does not take all profit from samples)
- We can do better at the cost of more computational time
- **Batch methods** seek to find the *best fitting value function* of given agents experience (training data) in a supervised way.

Batch Reinforcement Learning

- Gradient descent is simple and appealing
  - It is computationally efficient (one update per sample)
  - ... But it is not sample efficient (does not take all profit from samples)
- We can do better at the cost of more computational time
- **Batch methods** seek to find the *best fitting value function* of given agents experience (training data) in a supervised way.

Fitted Q-learning

Generalizarion of off-policy learning

Let's generalize the method:

Generalizarion of off-policy learning.

```
Get $\mathcal{D} = \{\langle s, a, r, s' \rangle\}$ using any probabilistic policy
repeat
    Set $\mathcal{SD}$ to $N$ samples randomly taken from $\mathcal{D}$
    for each sample $i$ in $\mathcal{SD}$ do
        $y_i \leftarrow r + \gamma \max_{a'} Q_\theta(s'_i, a')$
    end for
    $\theta \leftarrow \arg\min_\theta \sum (Q_\theta(s_i, a_i) - y_i)^2$    // Any ML regression method
until convergence
```

Generalizarion of off-policy learning

- Notice several differences:
  1. Randomly sample a set of N examples instead of only 1
  2. Don't use 1-step of gradient descent but compute exact solution (regression problem)
- Each change improves convergence
  1. Samples obtained randomly reduce correlation between them and stabilize Q value function for the regression learner
  2. Computation of exact solution avoid the true gradient problem

Generalizarion of off-policy learning

- Notice several differences:
  1. Randomly sample a set of N examples instead of only 1
  2. Don't use 1-step of gradient descent but compute exact solution (regression problem)

- Each change improves convergence
  1. Samples obtained randomly reduce correlation between them and stabilize Q value function for the regression learner
  2. Computation of exact solution avoid the true gradient problem

Fitted Q-learning

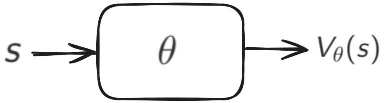
Fitted Q-learning

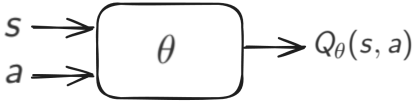
```
Given $\mathcal{D}$ of size $T$ with examples $(s_t, a_t, r_{t+1}, s_{t+1})$, and regression algorithm, set $N$ to zero and $Q_N(s,a) = 0$ for all $a$ and $s$
repeat
    $N \leftarrow N + 1$
    Build training set $TS = \{\langle (s_t, a_t), r_{t+1} + \gamma \max_a Q_N(s_{t+1}, a) \rangle\}_{t=1}^{T}$
    $Q_{N+1} \leftarrow$ regression algorithm on TS
until $Q_N \approx Q_{N+1}$ or $N >$ limit
return $\pi$ based on greedy evaluation of $Q_N$
```

- Works specially well for forward Neural Networks as regressors (Neural Fitted Q-learning)

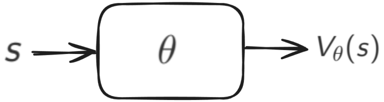
Deep Neural Networks: DQN

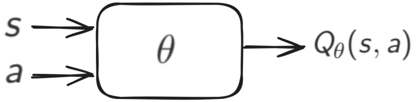
Use of Neural Networks for regression

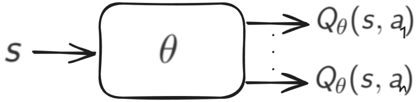




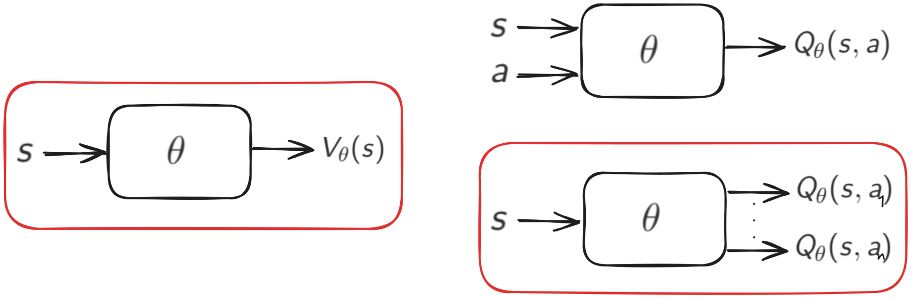
Use of Neural Networks for regression







Use of Neural Networks for regression



Recap of FA solutions

Two possible approaches for function approximation:

1. Incremental:
   - Pro: Learning on-line
   - Cons: No convergence due to (a) Data not i.i.d., that can lead to *catastrophic forgetting*, and (b) Moving target problem
2. Batch Learning:
   - Cons: Learn from collected dataset (not own experience)
   - Pro: Better convergence

Recap of FA solutions

Two possible approaches for function approximation:

1. Incremental:
   - Pro: Learning on-line
   - Cons: No convergence due to (a) Data not i.i.d., that can lead to *catastrophic forgetting*, and (b) Moving target problem
2. Batch Learning:
   - Cons: Learn from collected dataset (not own experience)
   - Pro: Better convergence

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 $\pi$ based on greedy evaluation of $Q_N$
```

Neural Fitted Q-learning

Neural Fitted Q-learning: Wrong version. Why?

```
Initialize weights $\theta$ 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 $\pi$ based on greedy evaluation of $Q_\theta$
```

- 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 $\theta$ 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 $\mathcal{D}$
        $\theta \leftarrow \theta - \alpha \sum_{t \in \mathcal{B}} \frac{\partial Q_\theta}{\partial \theta}(s_t, a_t)\left(Q_\theta(s_t, a_t) - [r_{t+1} + \gamma \max_{a'} Q_{\theta'}(s_{t+1}, a')]\right)$
    until convergence on learning or maximum number of steps
until maximum limit iterations
return $\pi$ based on greedy evaluation of $Q'_\theta$
```

- 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 $\theta$ 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 $\pi$ based on greedy evaluation of $Q'_\theta$
```

- 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 should 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 $\theta$ 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 $\epsilon$-greedy procedure and add them to **buffer** $\mathcal{D}$
        **repeat**
            Sample $\mathcal{B}$ mini-batch of $\mathcal{D}$
            $\theta \leftarrow \theta - \alpha \sum_{t \in \mathcal{B}} \frac{\partial Q_\theta}{\partial \theta}(s_t, a_t) \left( Q_\theta(s_t, a_t) - [r_{t+1} + \gamma \max_{a'} Q_{\theta'}(s_{t+1}, a')] \right)$
        **until** maximum number of steps $K$
    **until** maximum number of iterations $N$
**until** maximum limit iterations
**return** $\pi$ based on greedy evaluation of $Q'_\theta$
```

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 $\theta$ for NN for regression
Set $s$ to initial state, and $k$ to zero
**repeat**
    Choose $a$ from $s$ using policy $\pi_\theta$ derived from $Q_\theta$ (e.g., $\epsilon$-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 $\mathcal{D}$
    $\theta \leftarrow \theta - \alpha \sum_{t \in \mathcal{B}} \frac{\partial Q_\theta}{\partial \theta}(s_t, a_t) \left(Q_\theta(s_t, a_t) - [r_{t+1} + \gamma \max_{a'} Q_{\theta'}(s_{t+1}, a')]\right)$
    **if** k==N **then**
        $\theta' \leftarrow \theta$
        $k \leftarrow 0$
    **end if**
**until** maximum limit iterations
**return** $\pi$ based on greedy evaluation of $Q'_\theta$
```

DQN algorithm on Atari

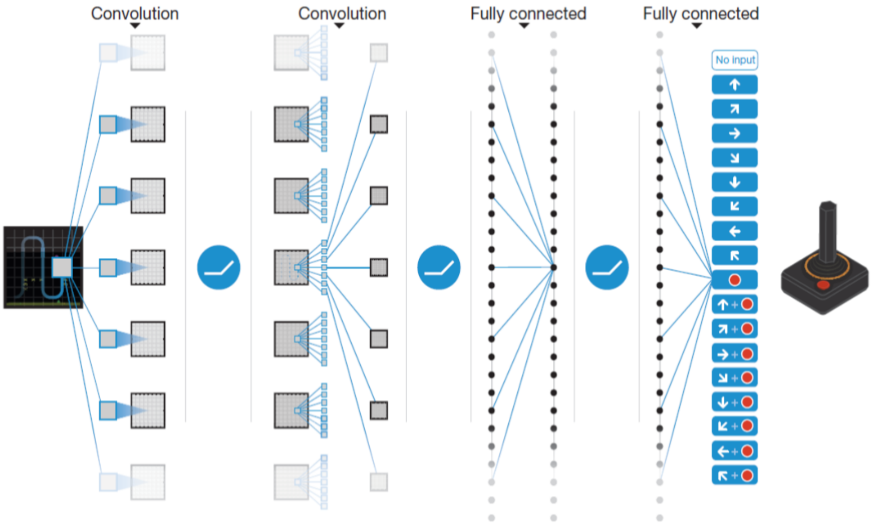
- Atari games in the gym
- End-to-end learning of values $Q(s, a)$ from pixels:

  State: Input state s is stack of **raw pixels from last 4 frames**

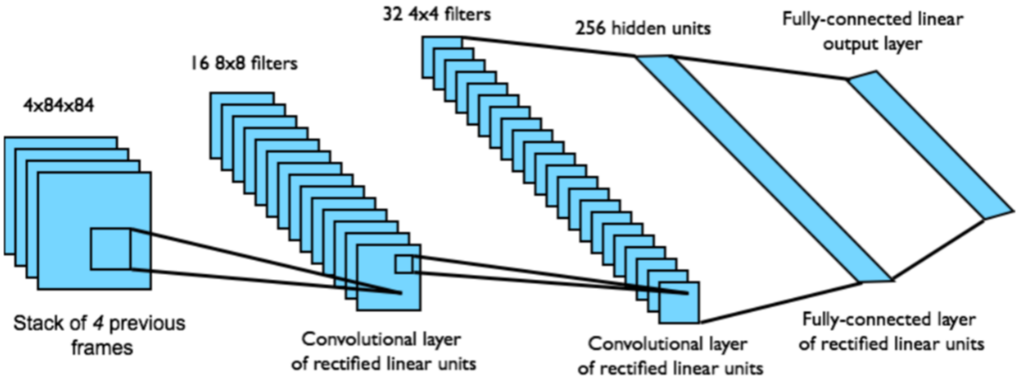
  Actions: Output is $Q(s, a)$ value for each of 18 joystick/button positions

  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

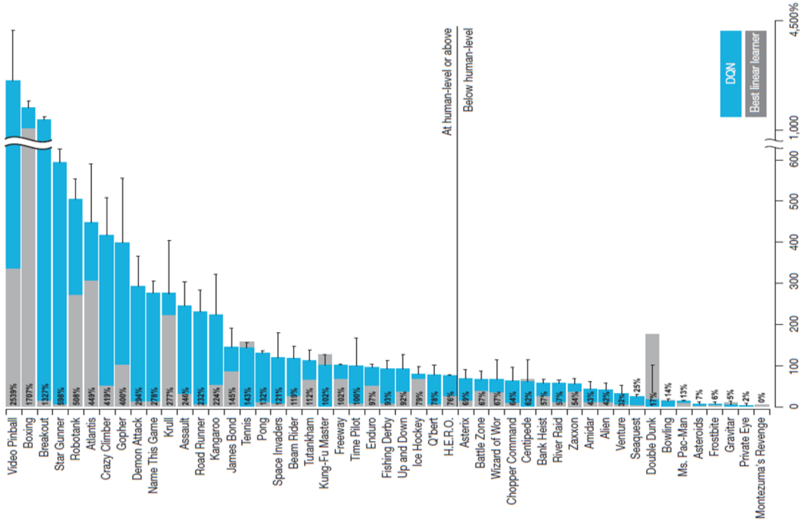
DQN algorithm on Atari



DQN algorithm on Atari


32 4x4 filters
256 hidden units
Fully-connected linear output layer
16 8x8 filters
4x84x84
Stack of 4 previous frames
Convolutional layer of rectified linear units
Convolutional layer of rectified linear units
Fully-connected layer of rectified linear units

DQN algorithm on Atari



DQN algorithm on Atari

Google Deepmind DQN playing
Atari Breakout

Setup:
NVIDIA GTX 690
i7-3770K - 16 GB RAM
Ubuntu 16.04 LTS
Google Deepmind DQN

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**

Improvements over basic DQN

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 random initialization 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 to other states.

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:

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

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

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

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

Double DQN (Hasselt, et al. 2015)

- In DQN, in fact, we have 2 value functions: Q_θ and $Q_{\theta'}$
- so, no need 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. Lets use the absolute value of TD-error of transition as a probability!

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

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

where $P(i)$ is probability of selecting sample i for the mini-batch, and $\alpha \geq 0$ is a new parameter ($\alpha = 0$ implies 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)

- 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

```
1: Input: minibatch $k$, step-size $\eta$, replay period $K$ and size $N$, exponents $\alpha$ and $\beta$, budget $T$.
2: Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
3: Observe $S_0$ and choose $A_0 \sim \pi_\theta(S_0)$
4: for $t = 1$ to $T$ do
5:     Observe $S_t, R_t, \gamma_t$
6:     Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
7:     if $t \equiv 0 \mod K$ then
8:         for $j = 1$ to $k$ do
9:             Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
10:            Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
11:            Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
12:            Update transition priority $p_j \leftarrow |\delta_j|$
13:            Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
14:        end for
15:        Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
16:        From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
17:    end if
18:    Choose action $A_t \sim \pi_\theta(S_t)$
19: 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, specific Deep Architecture specific for RL
- *Advantage function* definition:

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

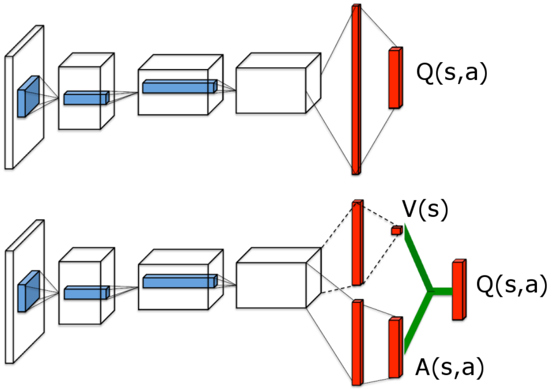
- So,

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

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

Dueling Network Architectures (Wang, et al. 2016)

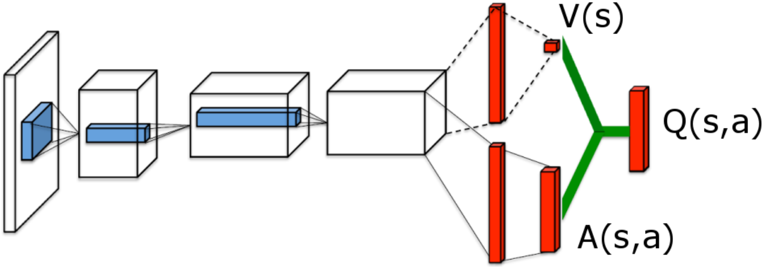
- Dueling network:



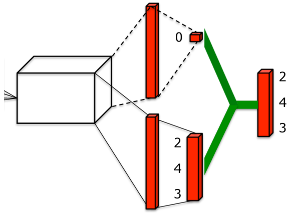
- Intuitive idea is that now we don't learn $Q(s,a)$ independently but share part that is $V(s)$ that improves generalization across actions

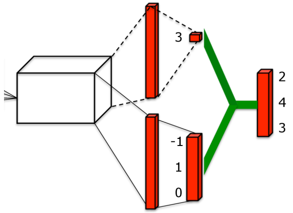
Dueling Network Architectures (Wang, et al. 2016)

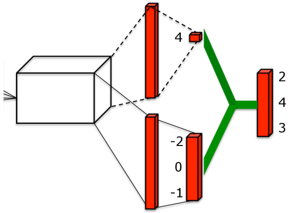
- We have now 3 sets of parameters:
  - θ: Usual weights of NN until red section
  - β: Weights to compute $V(s)$
  - α: Weights to compute $A(s, a)$
- Green part computes $A(s, a) + V(s)$



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







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:

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

- 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)

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:

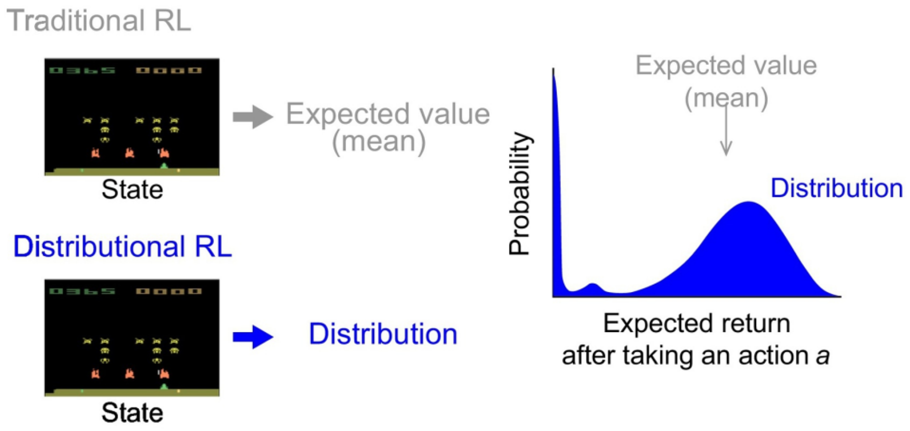
$$\left\langle s_t, a_t, r_t, \sum_{i=0}^{n} \gamma^{i-1} r_{t+1} + \gamma^n \max_{a'} Q_{\theta'}(s_{t+n}, a') \right\rangle$$

- 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)

Distributional RL

Distributional RL (Dabney et al. 2017)

- Instead of working with Expectation of Long-term-reward, work with Distributions of Long-term-reward



Distributional RL (Dabney et al. 2017)

- Apply Bellman equation on the distribution (some theory behind necessary)

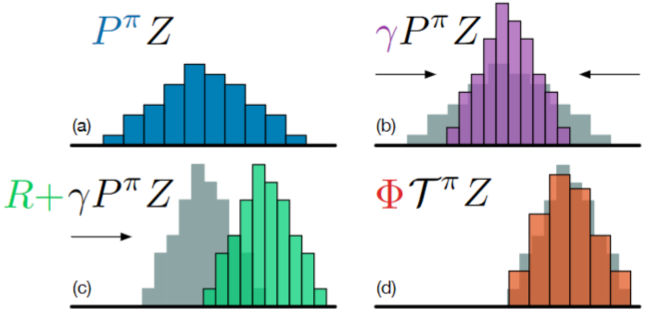
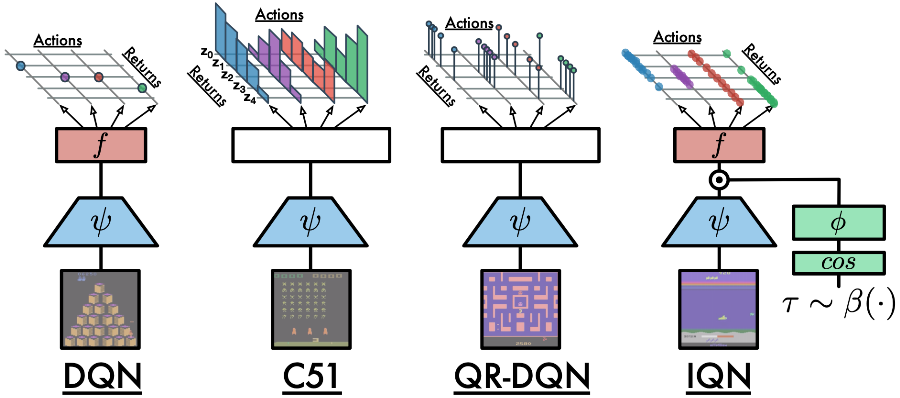


Figure 1. A distributional Bellman operator with a deterministic reward function: (a) Next state distribution under policy π, (b) Discounting shrinks the distribution towards 0, (c) The reward shifts it, and (d) Projection step (Section 4).

Distributional RL (Dabney et al. 2017)

- Several implementations of the same idea: C51, QR-DQN, IQN, and FQF



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)

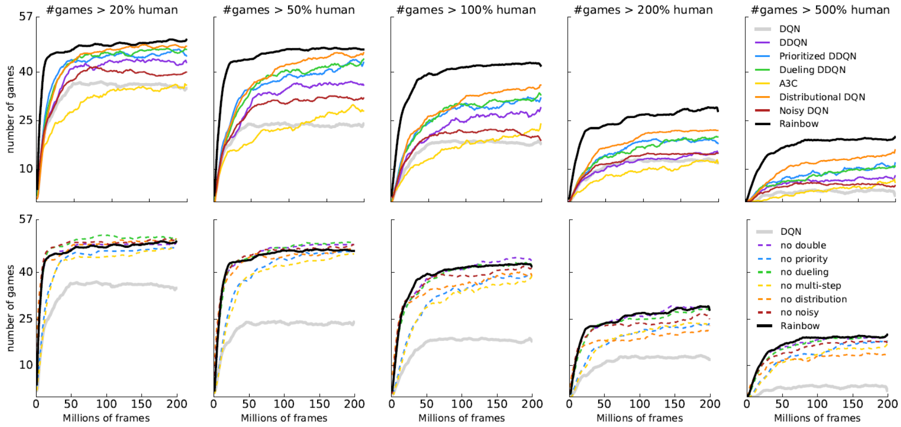
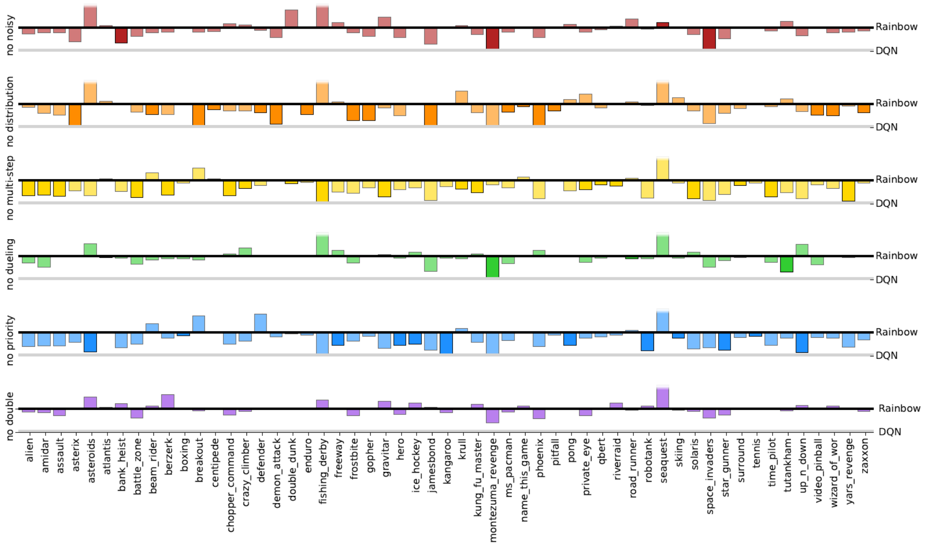


Figure 2: Each plot shows, for several agents, the number of games where they have achieved at least a given fraction of human performance, as a function of time. From left to right we consider the 20%, 50%, 100%, 200% and 500% thresholds. On the first row we compare Rainbow to the baselines. On the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

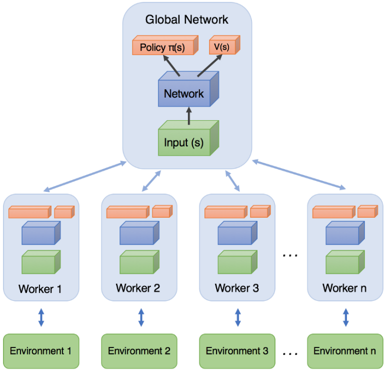
Rainbow (Hessel et al. 2017)



Asynchronous Q-learning

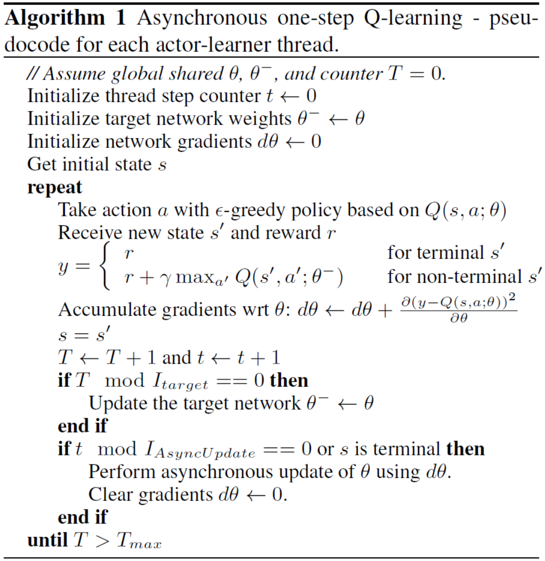
Asynchronous Q-learning (Mnih et al. 2016)

- 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 $\theta$, $\theta^-$, and counter $T = 0$.
Initialize thread step counter $t \leftarrow 0$
Initialize target network weights $\theta^- \leftarrow \theta$
Initialize network gradients $d\theta \leftarrow 0$
Get initial state $s$
repeat
    Take action $a$ with $\epsilon$-greedy policy based on $Q(s, a; \theta)$
    Receive new state $s'$ and reward $r$
    $y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s' \end{cases}$
    Accumulate gradients wrt $\theta$: $d\theta \leftarrow d\theta + \frac{\partial (y - Q(s,a;\theta))^2}{\partial \theta}$
    $s = s'$
    $T \leftarrow T + 1$ and $t \leftarrow t + 1$
    if $T \mod I_{target} == 0$ then
        Update the target network $\theta^- \leftarrow \theta$
    end if
    if $t \mod I_{AsyncUpdate} == 0$ or $s$ is terminal then
        Perform asynchronous update of $\theta$ using $d\theta$.
        Clear gradients $d\theta \leftarrow 0$.
    end if
until $T > T_{max}$
```

Practical tricks

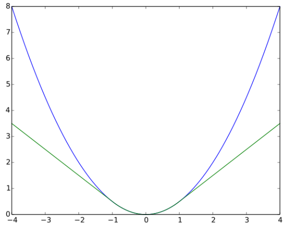
Practical tricks

- Patience. Training takes time (roughly hours to day on GPU training to see improvement)
- Learning rate scheduling is beneficial. Try high learning rates in initial exploration period.
- ϵ annealing f.i. from 1 to .1 is beneficial too
- *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

Partial Observability

- In a lot of cases the agent has not complete information of the *true* state and uses its perception as state.
- The problem is not anymore an MDP.
- How to solve these case?
  1. Formalize as a POMDP: MDP extended with set of observations O and probability of each observation given the true state. Agent work with a *belief vector* of probabilities of being in each state. Solve with dedicated algorithms
  2. Works with memory as a way to disambiguate the true state. Simple approaches like window of last n perceptions (DQN), or more interesting ones using LSTM