Lesson 12

Deep Q-Learning (DQN)

The problem of Q-learning

Q-learning exploits Bellman's equation

$$Q^*(s,a) = \mathbb{E}_{s'}[r_0 + \gamma max_{a'} Q^*(s',a')]$$

We compute Q^* via iterative updates:

$$\underbrace{Q^{i+1}(s,a)}_{\text{next estimation}} \leftarrow \underbrace{Q^i(s,a)}_{\text{current estimation}} + \alpha(\underbrace{r_0 + \gamma max_{a'} Q^i(s',a') - Q^i(s,a)}_{\text{recursive update}})$$

$Q^i \to Q^*$ when $i \to \infty$.

Not scalable!

Must compute $Q(s,a)$ for every state-action pair.

The problem of Q-learning

Q-learning exploits Bellman's equation

$$Q^*(s,a) = \mathbb{E}_{s'}[r_0 + \gamma max_{a'} Q^*(s',a')]$$

We compute Q^* via iterative updates:

$$\underbrace{Q^{i+1}(s,a)}_{\substack{next\\estimation}} \leftarrow \underbrace{Q^i(s,a)}_{\substack{current\\estimation}} + \alpha(\underbrace{r_0 + \gamma max_{a'} Q^i(s',a') - Q^i(s,a)}_{recursive\ update})$$

$Q^i \to Q^*$ when $i \to \infty$.

Not scalable!

Must compute $Q(s,a)$ for every state-action pair.

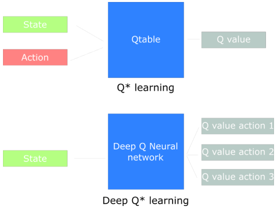
Deep Q-learning

Deep Q-learning: use a function approximator (a Neural Network!) to estimate the optimal action-value function

$$Q(s, a, \theta) \approx Q^*(s, a)$$

θ are the function parameters to be learned.

Instead of taking a as input, it is customary to return a value for each possible action a (the two functions are isomorphic)



The loss function

Given (s, a) the current Q-value estimate of the network

$$Q(s, a, \theta)$$

The expected value, given by the Bellman equation is

$$\mathbb{E}_{s'}[r_0 + \gamma max_{a'} Q(s', a', \theta)]$$

We try to minimize their distance, to get the fixpoint

$$L(\theta) = (\mathbb{E}_{s'}[r_0 + \gamma max_{a'} Q(s', a')] - Q(s, a, \theta))^2$$

Experience replay

For learning, we just need the loss function and its gradient

$$L(\theta) = (\mathbb{E}_{s'}[r + \gamma max_{a'} Q(s', a')] - Q(s, a, \theta))^2$$

To compute it, we need transitions

$$(s_i, a_i, r_i, T_i, s_{i+1})$$

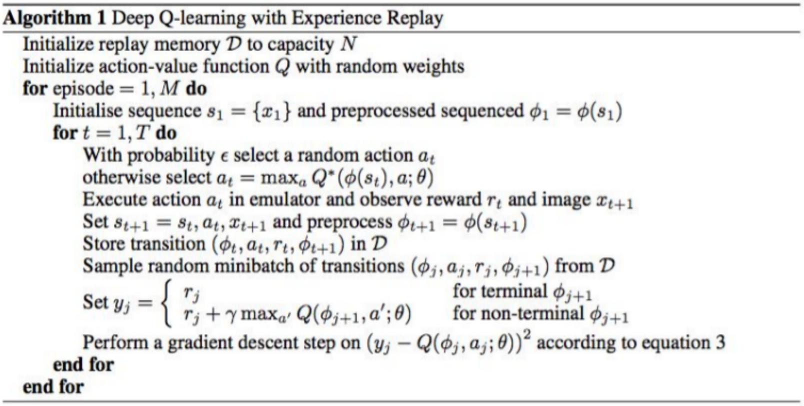
We can store these transitions in an experience memory, and replay them at leisure during training (**experience replay**).

Better than learning from batches of consecutive samples because:

- samples tend to be correlated $\Rightarrow$ inefficient learning
- great risk of introducing biases during learning

Q-learning algorithm with experience replay

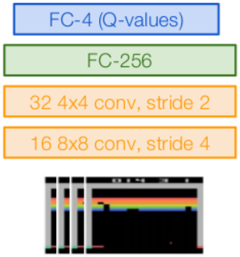
Playing Atari with DRL, Mnih et al. Nature 2015.



```
Algorithm 1 Deep Q-learning with Experience Replay
  Initialize replay memory $\mathcal{D}$ to capacity $N$
  Initialize action-value function $Q$ with random weights
  for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
      With probability $\epsilon$ select a random action $a_t$
      otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
      Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
      Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
      Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
      Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
      Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
      Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
  end for
```

The Atari Q-learning architecture

A single feed-forward pass to compute Q-values for all actions from the current state



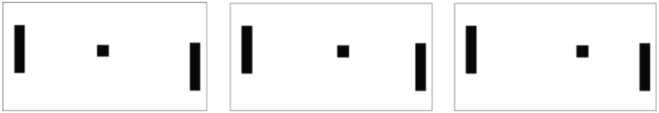
Last layer has an output for each action a, returning Q(s,a).

Number of actions between 4-18 depending on Atari game

state: 84x84x4 stack of last 4 frames
(RGB$\rightarrow$grayscale conversion, rescaling)

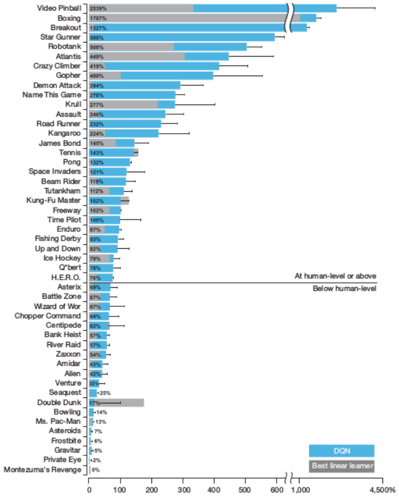
Why stacking frames?

We need a sequence of frames to capture movement



As an alternative, you can use a LSTM layer (after processing the state, and before computing Qvalues).

Atari Games and Q-learning



The **same network architecture** was applied to all games

Input are screen frames

Works well for reactive games, not for planning

Recently extended to Augmented Imagination (2017)

video

A note on rewards

Rewards are very different from game to game.

How can we use a same architecture?

The solution adopted in Playing Atari with DRL was to **crop all positive rewards to 1, and all negative rewards to -1**.

The lesson to be learned seems to be that the **entity of the reward does not really matter**: what matters is to distinguish between positive, negative, and neutral moves.

An alternative: RAM instead of screenshots

The RAM for the Atari 2600 console consists of 128 bytes!

We can directly exploit it as state observations, instead of video screenshots: see Learning from the memory of Atari 2600

Next Arguments

Deep Q-learning improvements:

- **Fixed Q-targets**
- **Double Q-learning**
- **Prioritized Experience Replay**
- **Dueling**
- Noisy Networks
- Distributional RL
- Rainbow

Next Arguments

Fixed Q-targets

Fixed Q-targets

With Q-learning, we try to approximate the optimal target Q-function Q^*, through progressive updates:

$$Q^*(s,a) - Q(s,a)$$

Moreover, we approximate the computation of $Q^*(s,a)$ as $r_o + max_{a'} Q(s',a')$, giving us the q-learning update

$$\underbrace{r_0 + \gamma max_{a'} Q(s',a')}_{\substack{\text{approximated} \\ \text{target } Q^*}} - \underbrace{Q(s,a)}_{\substack{\text{current} \\ \text{estimation}}}$$

The Q function is computed by a neural network $Q(s,a,w)$.

shared weights

In the loss function, the *same newtork* is used to provide two different estimations of the Q-function:

$$\underbrace{r_0 + \gamma max_{a'} Q(s', a', w)}_{\substack{\text{approximated} \\ \text{target } Q^*}} - \underbrace{Q(s, a, w)}_{\substack{\text{current} \\ \text{estimation}}}$$

At every step of training, the Q value shifts but also the “target value” shifts.

We are getting closer to our target but the target is also moving, that may lead to big oscillations in training.

Fixed Q-targets

Use a separate network $\overline{Q}$ with **fixed parameters** for estimating the TD target.

$$\underbrace{r_0 + \gamma max_{a'} \overline{Q(s', a')}}_{\text{approximated target } Q^*} - \underbrace{Q(s, a)}_{\text{current estimation}}$$

Periodically, copy the parameters from Q to $\overline{Q}$, to update the target network.

Implementation

Implementing fixed q-targets is straightforward:

- create two (identical) networks: DQNetwork, TargetNetwork
- define a function to transfer parameters from DQNetwork to TargetNetwork
- during training, compute the TD target using our target network. Update the target network with the DQNetwork every tau steps (tau is a user-defined hyper-parameter).

Next Arguments

Double Q-learning

Action values overestimation

The approximation of target action value is computed using a maximum over actions:

$$\underbrace{r_0}_{\text{local reward for taking action a}} + \gamma \cdot \underbrace{max_{a'} Q(s', a')}_{\text{max Q-value over all possible actions}}$$

Since the approximation is noisy, it is possible to prove that this will eventually result in a positive bias, finally resulting in an overestimation of the correct value.

Decoupling action choice and its estimation

The Double Q-learning approach consists in decoupling the choice of the action from its estimation, using two networks Q^A and Q^B.

```
1.  Initialize $Q^A, Q^B, s$
2.  repeat
3.      choose $a$ using $\epsilon, Q^A, Q^B$; observe $r, s'$
4.      choose (e.g. random) between UPDATE-A and UPDATE-B
5.      if UPDATE-A: # use $Q^A$ to choose action, $Q^B$ to estimate it
6.          $a^* = arg\ max_a Q^A(s', a)$
7.          $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(r + Q^B(s,a^*) - Q^A(s,a))$
8.      else if UPDATE-B: # use $Q^B$ to choose action, $Q^A$ to estimate it
9.          $a^* = arg\ max_a Q^B(s', a)$
10.         $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(r + Q^A(s,a^*) - Q^B(s,a))$
11.     $s \leftarrow s'$
12. until end
```

Decoupling action choice and its estimation

If we use Q^A to select best action $a^* = arg\ max_a Q^A(s', a)$, the value of $Q^A(s', a^*)$ could be biased by the choice (we know it will be large: it was the maximum!).

Since Q^B was updated on the same problem, but with a different set of experience samples, $Q^B(s', a^*)$ provides a better, unbiased estimate for the value of action a^*.

Next Arguments

Prioritized Experience Replay

Prioritized Experience Replay

Prioritized Experience Replay (PER) exploits the idea is that some experiences may be more important than others, and thus should be replayed more frequently.

We want to give higher priority to transitions for which there is a large difference between our prediction and the expected target.

For a transition $t = (s, a, r, F, s')$, its update is

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

and we set its priority to

$$p_t = |\delta_t|$$

stochastic prioritization

The probability of being chosen for a replay is computed accoding to the following rule:

$$P_t = \frac{p_t^\alpha}{\sum_t p_t^\alpha}$$

If $\alpha = 0$, all transistions have same probability; if α is large, it priveleges transitions with high priority p_t.

However, we are introducing a bias toward high-priority samples (more chances to be selected), with a risk of over-fitting over the small portion of experiences that we **presume** to be interesting.

importance sampling weight

We can correct the bias with **importance sampling weights**.
If N is the dimension of the replay buffer, then

$$w_t = (N \cdot P_t)^{-\beta}$$

that compensate the non-uniform probabilities P_t when $\beta = 1$ (if some transition has a high probability, we reduce its weight).

These weights are folded into the Q-learning update by using $w_t \delta_t$ instead of δ_t.

For stability reasons, weights are normalized by $1/(max_t w_t)$ so that they only scale the update δ_t downwards.

β is initialized to 1 and goes to 0 during training.

Next Arguments

Dueling

Advantage

Each Q-value $Q(s, a)$ estimates how good it is to take action a in state s: it depends both on the action a and the value of the given state s.

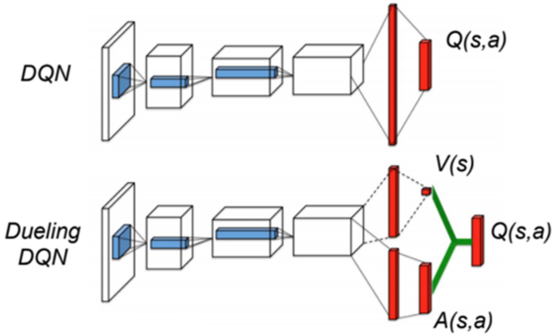
We can decompose $Q(s, a)$ as the sum of:

- V(s): the value of being at state s
- A(s,a): the advantage of taking action a in state s, measuring how much better is to take action a versus all other possible actions at that state.

$$Q(s, a) = \underbrace{V(s)}_{value} + \underbrace{A(s, a)}_{advantage}$$

Dueling DQN

Dueling Network Architectures (DDQN) split the computation of $V(s)$ and $A(s,a)$ in two different streams:



Dueling DQN

Intuitively, the dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state.

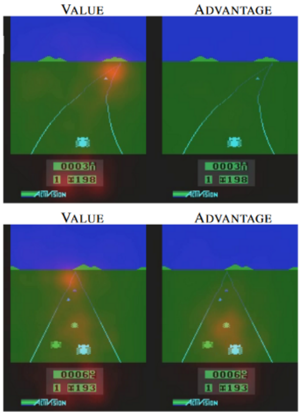
This is particularly useful in states where actions do not affect the environment in any relevant way.

Conversely, in states where the action is relevant, it can focus on the advantage without caring for the current evaluation of the state.

Saliency maps on Enduro

For estimating Value, the network focus on:

- the horizon
- the score



no car close by: action is not relevant

pay attention to the car in front: action is crucial

Saliency maps (the orange blurs) are computed with Jacobians (partial derivatives) on input images. See: Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps

Naif Aggregation

Naif aggregation:

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

Problem: given $Q(s,a)$, it is impossible to recover $V(s)$ and $A(s,a)$ and hence it is difficult to distribute the error during backpropagation (identifiability).

Aggregating Value and Advantage

$$\cancel{Q(s,a) = V(s) + A(s,a)}$$

Force A to have **zero advantage** for the best action a^*:

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

If $a^* = argmax_a A(s,a)$, $Q(s,a^*) = V(s)$ and $A(s,a^*) = 0$.

Alternative: mean instead of max

Experimentally, replacing max with mean seem to work better:

$$Q(s,a) = V(S) + A(s,a) - mean_a A(s,a)$$

Subtracting the mean value helps to improve stability during training.

Next Arguments

Noisy Networks

Noisy Networks

Noisy Networks are characterized by **noisy dense layers**, combining a deterministic and a noisy stream:

$$y = \underbrace{b + Wx}_{usual\ layer} + \underbrace{(b_{noisy} \odot \epsilon^b + (W_{noisy} \odot \epsilon^w)x)}_{noisy\ stream}$$

- $W, b, W_{noisy}, b_{noisy}$ are learned parameters
- ϵ^b, ϵ^w are randomly generated

This layer is used in substitution of any standard dense layer (doubling the number of parameters).

Random exploration

The purpose of the noise is to augment the randomicity in the choice of actions.

Since noisy-weights are learned, and the resulting noise is state dependent, this allows the network to randomly explore the environment at different rates in different parts of the state space.

Noisy networks replace and seem to work better than the ϵ-greedy strategy.

Next Arguments

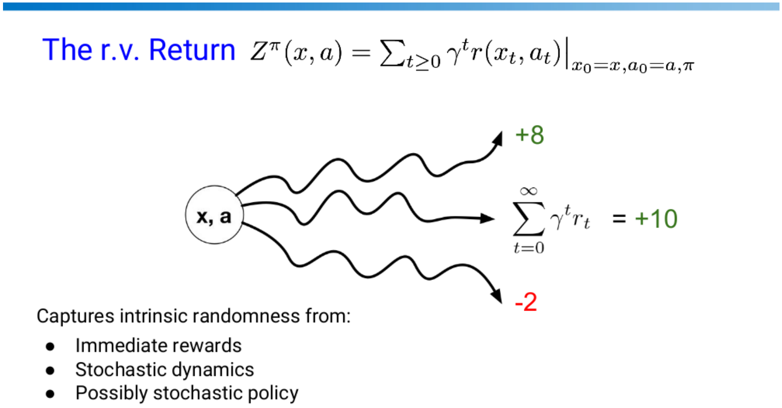
Distributional RL

Distributional RL

Distributional RL try to learn the probability distribution of the future cumulative reward, instead of the traditional approach of modeling the expectation of this return.

Specifically, DRL addresses the random return Z whose expectation is the value Q.

The cumulative reward random variable Z



The r.v. Return $Z^{\pi}(x,a) = \sum_{t\geq 0} \gamma^t r(x_t, a_t)\Big|_{x_0=x, a_0=a, \pi}$

x, a
+8
$\sum_{t=0}^{\infty} \gamma^t r_t$ = +10
-2

Captures intrinsic randomness from:

- Immediate rewards
- Stochastic dynamics
- Possibly stochastic policy

Slide from Distributional Reinforcement Learning by R.Munos.

Distributional Bellman

Let $a*$ be the best possible action in s'.
The Bellman equation is

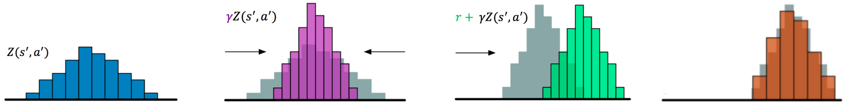
$$Q(x,a) = R(x,a) + \gamma Q(x', a^*)$$

Similarly,

$$Z(x,a) = R(x,a) + \gamma Z(s', a^*)$$

discretization

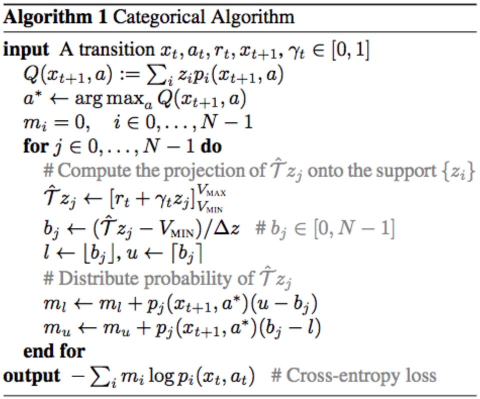
The distribution is discretized over a support in a given range between V_{min} and V_{max} using a fixed number of bins (e.g. 51).



(1) Z at s' (2) discounted Z (3) shift by r (4) project

The loss between the two distribution can be computed with KL-divergence.

C51-pseudocode



Algorithm 1 Categorical Algorithm

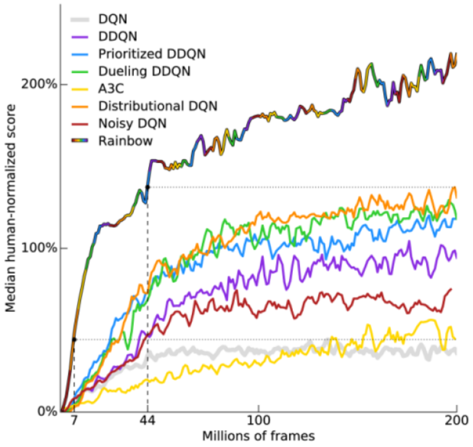
```
input A transition $x_t, a_t, r_t, x_{t+1}, \gamma_t \in [0,1]$
  $Q(x_{t+1}, a) := \sum_i z_i p_i(x_{t+1}, a)$
  $a^* \leftarrow \arg\max_a Q(x_{t+1}, a)$
  $m_i = 0, \quad i \in 0, \ldots, N-1$
  for $j \in 0, \ldots, N-1$ do
    # Compute the projection of $\hat{\mathcal{T}} z_j$ onto the support $\{z_i\}$
    $\hat{\mathcal{T}} z_j \leftarrow [r_t + \gamma_t z_j]_{V_{\text{MIN}}}^{V_{\text{MAX}}}$
    $b_j \leftarrow (\hat{\mathcal{T}} z_j - V_{\text{MIN}})/\Delta z$   # $b_j \in [0, N-1]$
    $l \leftarrow \lfloor b_j \rfloor, u \leftarrow \lceil b_j \rceil$
    # Distribute probability of $\hat{\mathcal{T}} z_j$
    $m_l \leftarrow m_l + p_j(x_{t+1}, a^*)(u - b_j)$
    $m_u \leftarrow m_u + p_j(x_{t+1}, a^*)(b_j - l)$
  end for
output $-\sum_i m_i \log p_i(x_t, a_t)$   # Cross-entropy loss
```

Suggested reading: Distributional Bellman and the C51 Algorithm

Next argument

Rainbow

Rainbow


DQN
DDQN
Prioritized DDQN
Dueling DDQN
A3C
Distributional DQN
Noisy DQN
Rainbow
200%
100%
0%
Median human-normalized score
7
44
100
200
Millions of frames