AMCO
ARTIFICIAL INTELLIGENCE, MACHINE LEARNING AND CONTROL RESEARCH GROUP

Reinforcement Learning
2025/2026

RL2
Reinforcement Learning Research Lab @ UniPD

Lecture #20
Batch Methods & Intro to DeepRL

Gian Antonio Susto

Recap: Actor-Critic with advantage function -> Advantage Actor Critic (A2C)

- Consider the TD error

$$\delta_{\pi_\theta} = r + \gamma\, v_{\pi_\theta}(s') - v_{\pi_\theta}(s)$$

- The TD error is an unbiased estimate of the advantage function

$$\begin{aligned}\mathbb{E}\left[\delta_{\pi_\theta} | s, a\right] &= \mathbb{E}\left[r + \gamma\, v_{\pi_\theta}(s') | s, a\right] - v_{\pi_\theta}(s) \\ &= q_{\pi_\theta}(s, a) - v_{\pi_\theta}(s) \\ &= A_{\pi_\theta}(s, a)\end{aligned}$$

- Let's exploit this:

$$\nabla J(\theta) = \mathbb{E}_\pi\left[\nabla \ln \pi(a|s, \theta)\, \cancel{A_{\pi_\theta}(s, a)}\right]$$

$$\downarrow$$

$$\nabla J(\theta) = \mathbb{E}_\pi\left[\nabla_\theta \ln \pi(a|s, \theta)\, \delta_{\pi_\theta}\right]$$

In practice we can use an approximator:

$$\delta_{\mathbf{v}} = r + \gamma \hat{v}(s', \mathbf{v}) - \hat{v}(s, \mathbf{v})$$

Recap: Actor-Critic with advantage function -> Advantage Actor Critic (A2C)

Initialize policy parameter $\theta, \boxed{v}$

Algorithm parameter : step-size $\alpha, \boxed{\beta} > 0$

Repeat (for each episode) :

Initialize s and take an action $a \sim \pi(a|s,\theta)$

Loop for each step of the episode $t = 0, \ldots, T-1$

Observe reward r and new state s'

$$\delta_{\mathbf{v}} = r + \gamma \hat{v}(s', \mathbf{v}) - \hat{v}(s, \mathbf{v})$$

$$\theta = \theta + \alpha\, \delta_{\mathbf{v}}\, \nabla_\theta \ln \pi(a|s,\theta)$$

$$\boxed{\mathbf{v} = \mathbf{v} + \beta \delta_{\mathbf{v}} \nabla \hat{v}(s, \mathbf{v})}$$

$$s \leftarrow s'$$

Now the critic is simplified! We get only 1 ste of parameters

Batch methods

Recap: Incremental vs. Batch approaches

- Up until now we have been dealing with incremental approaches: data and experience is collected ‘on-line’ in an incremental fashion
- Batch approaches instead store data during experience to achieve some benefits
- Batch approaches were a breakthrough in ATARI [1]
- Notes:
  - No dedicated chapter for batch approaches (some references in chapter 16)
  - We briefly talked about ‘batch’ in the past…

[1] Mnih et al., “Human-level control through deep reinforcement learning” (Nature, 2015)

Prediction: (batch) MC and TD – AB Example

Two states, no discounting, 8 episodes of experience:

A, 0, B, 0
B, 1
B, 1
B, 1
B, 1
B, 1
B, 1
B, 0

What is $V(A)$ and what is $V(B)$?

$V(B)$ = 3/4
$V(A)$ = 0 is the Monte Carlo solution
-> MC converges to solution with minimun mean-squared error
$V(A)$ = 3/4 is the TD(0) solution
-> TD(0) (implicitely) converges to solution of max likelihood Markov model

Recap: Batch methods in Reinforcement Learning

- Gradient descent is simple and appealing
- But it is not sample efficient (one of the reason we introduced off-policy)
- Batch methods seek to find the best fitting value function given the agent's experience (training data)

Recap: Batch methods in Reinforcement Learning

- Gradient descent is simple and appealing
- But it is not sample efficient (one of the reason we introduced off-policy)
- Batch methods seek to find the best fitting value function given the agent's experience (training data)

Intuition behind batch methods: *in general one of the requirement for SGD optimization is that the training data is indipendent and identically distributed and when an Agent interacts with the environment, the sequence of experience tuples can be highly correlated.*

Learning algorithms that learn from each of these experiences tuples in sequential order runs the risk of getting swayed by the effects of this correlation

Recap: Experience Replay

- We can improve this situation using a large buffer of past experience and sample training data from it (a random subset), instead of using latest experience
- This technique is called **replay buffer or experience buffer**. The replay buffer contains a collection of tuples (S, A, R, S'). The tuples are gradually added to the buffer as we are interacting with the environment*
- The act of sampling a small batch of tuples from the replay buffer in order to learn is known as **experience replay**
- In addition to breaking harmful correlations, experience replay allows us to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience

*The simplest implementation is a buffer of fixed size, with new data added tothe end of the buffer so that it pushes the oldest experience out of it

Batch approaches

VFA – TLDR:

- You collect a dataset D={(s,a,r,s')}
- You repeatedly update V(s) or Q(s,a) by training on all samples in the dataset

$$V_\theta(s) \approx V^\pi(s)$$
$$Q_\theta(s,a) \approx Q^\pi(s,a)$$

Policy Gradient – TLDR:

- You collect a batch of trajectories
- Then compute

$$\mathcal{D} = \{\tau_1, \tau_2, \ldots, \tau_N\}$$

$$\nabla J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t$$

Batch approaches

VFA – TLDR:

- You collect a dataset D={(s,a,r,s')}
- You repeatedly update V(s) or Q(s,a) by training on all samples in the dataset

$$V_\theta(s) \approx V^\pi(s)$$
$$Q_\theta(s, a) \approx Q^\pi(s, a)$$

In the following slides we concentrate on this...

Incremental Methods: short recap

1. Forget about Policy-based approaches

2. With function approximation we have seen semi-gradient SARSA:

   a. We approximate Q.

   b. We use TD(0) targets.

   c. We update Q with SGD

   d. We update the policy with epsilon-greedy.

Incremental Methods: short recap

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

Replay buffer

1. With incremental methods, we sample (s,a,s’,r), we do one update and we do not use that sample anymore.

Replay buffer

1. With incremental methods, we sample (s,a,s’,r), we do one update and we do not use that sample anymore.

2. This is not “sample efficient”.

Replay buffer

1. With incremental methods, we sample (s,a,s’,r), we do one update and we do not use that sample anymore.

2. This is not “sample efficient”.

3. Instead, we save each sample in a replay buffer D.

Replay buffer

1. With incremental methods, we sample (s,a,s’,r), we do one update and we do not use that sample anymore.

2. This is not “sample efficient”.

3. Instead, we save each sample in a replay buffer D.

4. We can use SGD with one sample sampled from D.

Replay buffer

1. Given the usual mean squared error over D:

$$
\begin{aligned}
LS(\mathbf{w}) &= \sum_{t=1}^{\boxed{T}} \left(v_t^\pi - \hat{v}(s_t, \mathbf{w})\right)^2 \\
&= \mathbb{E}_{\mathcal{D}}\left[\left(v^\pi - \hat{v}(s, \mathbf{w})\right)^2\right]
\end{aligned}
$$

t and T are just related to the index and cardinality of the batch

Replay buffer

2. Given experience $\mathcal{D}$ consisting of $\langle state, value \rangle$ pairs

$$\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, \ldots, \langle s_T, v_T^\pi \rangle\}$$

Repeat :

1. Sample state, value from experience

$$\langle s, v^\pi \rangle \sim \mathcal{D}$$

2. Apply stochastic gradient descent update

$$\Delta \mathbf{w} = \alpha \left(v^\pi - \hat{v}(s, \mathbf{w})\right) \nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w})$$

$$\mathbf{w} \leftarrow \mathbf{w} + \Delta \mathbf{w}$$

Replay buffer

2. Given experience $\mathcal{D}$ consisting of $\langle state, value \rangle$ pairs

$$\mathcal{D} = \{\langle s_1, \boxed{v_1^\pi} \rangle, \langle s_2, \boxed{v_2^\pi} \rangle, \ldots, \langle s_T, \boxed{v_T^\pi} \rangle\}$$

Repeat :

1. Sample state, value from experience

$$\langle s, v^\pi \rangle \sim \mathcal{D}$$

2. Apply stochastic gradient descent update

$$\Delta \mathbf{w} = \alpha \left(\boxed{v^\pi} - \hat{v}(s, \mathbf{w}) \right) \nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w})$$

$$\mathbf{w} \leftarrow \mathbf{w} + \Delta \mathbf{w}$$

We are assuming the availability of the oracle...

Replay buffer

3.
   - We do not know true values v_t^π
   - In practice, our *training data* must use noisy or biased samples of v_t^π

Linear Least Squares Prediction Algorithms

LSMC Least Squares Monte-Carlo uses return

$$v_t^\pi \approx G_t$$

LSTD Least Squares Temporal-Difference uses TD target

$$v_t^\pi \approx R_{t+1} + \gamma\, \hat{v}(S_{t+1}, \mathbf{w})$$

LSTD(λ) Least Squares TD(λ) uses λ-return

$$v_t^\pi \approx G_t^\lambda$$

Least Squares Prediction with Linear models

1. Recall what we have with linear models: $\hat{v}(s, \mathbf{w}) = \mathbf{x}(s)^T \mathbf{w}$

2. We can solve least squares directly.

Least Squares Prediction with Linear models (optional)

- At minimum of LS($\mathbf{w}$), the expected update must be zero

$$\boxed{\mathbb{E}_{\mathcal{D}}\left[\Delta \mathbf{w}\right] = 0}$$

$$\boxed{\Delta \mathbf{w} = \alpha(v^{\pi} - \hat{v}(s, \mathbf{w}))\nabla_{\mathbf{w}}\hat{v}(s, \mathbf{w})}$$

$$\alpha \sum_{t=1}^{T} \mathbf{x}(s_t)\left(v_t^{\pi} - \mathbf{x}(s_t)^T \mathbf{w}\right) = 0$$

This is the standard thing we do in Least Squares: putting the gradient equal to 0

Least Squares Prediction with Linear models (optional)

- At minimum of LS($\mathbf{w}$), the expected update must be zero

$$\mathbb{E}_{\mathcal{D}}\left[\Delta \mathbf{w}\right] = 0$$

$$\boxed{\Delta \mathbf{w} = \alpha(v^{\pi} - \hat{v}(s, \mathbf{w}))\nabla_{\mathbf{w}}\hat{v}(s, \mathbf{w})}$$

$$\alpha \sum_{t=1}^{T} \mathbf{x}(s_t)\left(v_t^{\pi} - \mathbf{x}(s_t)^T \mathbf{w}\right) = 0$$

$$\sum_{t=1}^{T} \mathbf{x}(s_t) v_t^{\pi} = \sum_{t=1}^{T} \mathbf{x}(s_t)\mathbf{x}(s_t)^T \mathbf{w}$$

$$\mathbf{w} = \left(\sum_{t=1}^{T} \mathbf{x}(s_t)\mathbf{x}(s_t)^T\right)^{-1} \sum_{t=1}^{T} \mathbf{x}(s_t) v_t^{\pi}$$

Least Squares Prediction with Linear models (optional)

- At minimum of LS($\mathbf{w}$), the expected update must be zero

$$\mathbb{E}_{\mathcal{D}}\left[\Delta \mathbf{w}\right] = 0$$

$$\boxed{\Delta \mathbf{w} = \alpha(v^{\pi} - \hat{v}(s, \mathbf{w}))\nabla_{\mathbf{w}}\hat{v}(s, \mathbf{w})}$$

$$\alpha \sum_{t=1}^{T} \mathbf{x}(s_t)\left(v_t^{\pi} - \mathbf{x}(s_t)^T \mathbf{w}\right) = 0$$

$$\sum_{t=1}^{T} \mathbf{x}(s_t) v_t^{\pi} = \sum_{t=1}^{T} \mathbf{x}(s_t)\mathbf{x}(s_t)^T \mathbf{w}$$

$$\mathbf{w} = \left(\sum_{t=1}^{T} \mathbf{x}(s_t)\mathbf{x}(s_t)^T\right)^{-1} \sum_{t=1}^{T} \mathbf{x}(s_t) v_t^{\pi}$$

- For N features, direct solution time is $O(N^3)$
- Incremental solution time is $O(N^2)$ using Sherman-Morrison

Least Squares for Control with Linear models (optional)

We can do the same for control:

1. Approximate Q, for example using LSTDQ (Least Squares TD Q-Learning).
2. Policy improvement.

Least Squares for Control with Linear models (optional)

- The following pseudocode uses LSTDQ for policy evaluation
- It repeatedly re-evaluates experience $\mathcal{D}$ with different policies

ALGORITHM LSPI-TD$(\mathcal{D}, \pi_0)$

$\pi' \leftarrow \pi_0$
repeat
  $\pi \leftarrow \pi'$
  $Q \leftarrow$ **LSTDQ**$(\pi, \mathcal{D})$
  for all $s \in \mathcal{S}$ **do**
    $\pi'(s) \leftarrow \underset{a \in \mathcal{A}}{\text{argmax}}\ Q(s, a)$
  end for
until $(\pi \approx \pi')$
return π

LSPI-TD: Least Squares Policy Iteration with TD

Least Squares for Control with Linear models (optional)

- The following pseudocode uses LSTDQ for policy evaluation
- It repeatedly re-evaluates experience $\mathcal{D}$ with different policies

ALGORITHM LSPI-TD$(\mathcal{D}, \pi_0)$

$\pi' \leftarrow \pi_0$
repeat
  $\pi \leftarrow \pi'$
  $Q \leftarrow$ **LSTDQ**$(\pi, \mathcal{D})$
  for all $s \in \mathcal{S}$ **do**
    $\pi'(s) \leftarrow \underset{a \in \mathcal{A}}{\text{argmax}}\ Q(s, a)$
  end for
until $(\pi \approx \pi')$
return π

You can do this also with Deep Learning (more on this later)

Batch methods for the exam

- Concepts
- Adapt the algorithms and pseudocodes you have seen so far to the batch settings

Content for the second partial (and next theoretic exams) end here!

What to expect until the end of the course

10 & 11 December

- Deep RL (really useful for programming part + real world applications)

17 & 18 December

- Research topics and applications* (with guest lecturers)

1. RL in Robotics
2. RL in Telecommunications (with F. Chiariotti)
3. RL in Manufacturing (with A. Immordino)
4. RL in Smart Mobilty and Autonomous Driving (with M. Cederle)
5. Advancements in RL algorithms (with R. De Monte)

* With RL thesis topics proposals

Towards Deep Reinforcement Learning (DRL)

Towards Deep Reinforcement Learning (DRL)

Up until now, the approximation could have been done with any approach... now we are restricting our considerations to the case of DL architectures

Possible approaches

1. Value-based algorithms:
   a. SARSA
   b. Q-Learning
2. Policy Gradient approaches:
   a. Reinforce
   b. Actor-Critic

$$V_\theta(s) \approx V^\pi(s)$$
$$Q_\theta(s, a) \approx Q^\pi(s, a)$$

$$\pi_\theta(s, a) = \mathbb{P}\left[a \mid s, \theta\right]$$

Possible approaches

1. Value-based algorithms:
   a. SARSA
   b. Q-Learning
2. Policy Gradient approaches:
   a. Reinforce
   b. Actor-Critic

$$V_\theta(s) \approx V^\pi(s)$$
$$Q_\theta(s, a) \approx Q^\pi(s, a)$$

Let's focus on this!

$$\pi_\theta(s, a) = \mathbb{P}\left[a \mid s, \theta\right]$$

Use NNs instead of Linear models

1. Either for control or prediction, linear model might be not enough:
    a. We have to design proper features: hard in many cases!
    b. or we assume that actual Q/V is linear: quite not realistic.

Use NNs instead of Linear models

1. Either for control or prediction, linear model might be not enough:
    a. We have to design proper features: hard in many cases!
    b. or we assume that actual Q/V is linear: quite not realistic.
2. What about Neural Networks?

Use NN instead of Linear models

Neural Networks, Vol. 2, pp. 359–366, 1989
Printed in the USA. All rights reserved.

0893-6080/89 $3.00 + .00
Copyright © 1989 Pergamon Press plc

ORIGINAL CONTRIBUTION

Multilayer Feedforward Networks are Universal Approximators

KUR' HORNIK

Technische Universität Wien

MAXWELL STINCHCOMBE AND HALBEK WHITE

University of California, San Diego

(*Received* 16 *September* 1988; *revised and accepted* 9 *March* 1989)

Abstract—*This paper rigorously establishes that standard multilayer feedforward networks with as few as one hidden layer using arbitrary squashing functions are capable of approximating any Borel measurable function from one finite dimensional space to another to any desired degree of accuracy, provided sufficiently many hidden units are available. In this sense, multilayer feedforward networks are a class of universal approximators.*

Keywords—Feedforward networks, Universal approximation, Mapping networks, Network representation capability, Stone-Weierstrass Theorem, Squashing functions, Sigma-Pi networks, Back-propagation networks.

A big problem in RL (bigger in the DRL case)

A big problem in RL (bigger in the DRL case)

A big problem in RL (bigger in the DRL case)

A big problem in RL (bigger in the DRL case)

- While with linear approaches we have closed form solutions, in DL we need stochastic gradient descent (SGD)
- Convergence in SGS would dramatically benefit from IID data!

Big problem: RL does not provide I.I.D samples

1. If we perform one update per time step (s, a, s’, r), we use SGD on correlated samples.

Big problem: RL does not provide I.I.D samples

1. If we perform one update per time step (s, a, s', r), we use SGD on correlated samples.

2. No stationarity: doing control, we change policy -> state distribution changes over time $\mu_\pi(s)$

Deep Q-Network (DQN): two papers to play Atari Games :)

1. First original paper: “Playing Atari with Deep Reinforcement Learning”, 2013

2. Improved version: “Human-level control through deep reinforcement learning”, 2015, Nature

Deep Q-Network (DQN): two papers to play Atari Games :)

We take such papers as a running example to discuss ‘complexity’ in using DL architectures in VFA. In ATARI, they are using Q-Learning for control:

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

DQN

We start by just applying Neural Network to Q-learning. We define a Q Network for Q:

1. The input to the Neural Network is the state S.

DQN

We start by just applying Neural Network to Q-learning. We define a Q Network for Q:

1. The input to the Neural Network is the state S.
2. We have one output per action: one output neuron (linear unit) per action

DQN

We start by just applying Neural Network to Q-learning. We define a Q Network for Q:

1. The input to the Neural Network is the state S.
2. We have one output per action: one output neuron (linear unit) per action

NOTE: DQN has been used for the first time for Atari games: the input is a stack of different frames -> CNN are used.

DQN: how to update the estimation of Q

1. We approximate Q with a NN as in the previous slide.

DQN: how to update the estimation of Q

1. We approximate Q with a NN as in the previous slide.
2. To train it, we can use the squared loss with Q-learning target.

DQN: how to update the estimation of Q

1. We approximate Q with a NN as in the previous slide.
2. To train it, we can use the squared loss with Q-learning target.
3. We explore as in Q-learning with an epsilon-greedy policy.

DQN: how to update the estimation of Q

1. We approximate Q with a NN as in the previous slide.
2. To train it, we can use the squared loss with Q-learning target.
3. We explore as in Q-learning with an epsilon-greedy policy.
4. Contrary to Q-learning (or SARSA semi-gradient), while the agent explores, the sample $(s_t, a_t, s_{t+1}, r_{t+1})$ is just saved in the **Replay Buffer D**. So, we do not perform any update with the current sample.

DQN: how to update the es

- Using a replay buffer in DQN makes the training data *approximately* IID because it **breaks the strong temporal correlations** present in sequential samples.
- We randomly mixing transitions from different times and trajectories, breaking the temporal correlations inherent in online RL.

1. We approximate Q with a NN as in the p
2. To train it, we can use the squared loss
3. We explore as in Q-learning with an epsilon-greedy policy.
4. Contrary to Q-learning (or SARSA semi-gradient), while the agent explores, the sample $(s_t, a_t, s_{t+1}, r_{t+1})$ is just saved in the **Replay Buffer D**. So, we do not perform any update with the current sample.
5. Instead, we sample a batch of samples $(s_j, a_j, s_{j+1}, r_{j+1}) \sim D$ and we perform a mini-batch SGD update.

DQN

Formally, the update is:

$$y_j = \begin{cases} r_j & s_{j+1} \text{ terminal} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta_i) & s_{j+1} \text{ not terminal} \end{cases}$$

$$\theta_{i+1} = \theta_i + \alpha \frac{1}{M} \sum_{j=1}^{M} (y_j - Q(s_j, a_j; \theta_i)) \nabla_\theta Q(s_j, a_j; \theta_i)$$

The M tuples $(s_j, a_j, s_{j+1}, r_{j+1}) \sim D$ might be obtained from different episodes and therefore we have no correlation.

DQN Nature: how to “reduce” non-stationarity of targets

1. Changing the Q network at one step, we change the targets (this due to bootstrapping).

- Nothing new here: we have been dealing with changing targets in bootstrapping before (even before introducing VFA)
- With VFA approximation this is more relevant as one change of parameters affect the whole estimation of Q (for all states and actions pairs)
- With DRL is even worse: we want IID samples and so we want stationarity!

DQN Nature: how to “reduce” non-stationarity of targets

1. Changing the Q network at one step, we change the targets (this due to bootstrapping).

2. They propose to define a so-called target network: they consider a copy of the same network initialized with the same weights:

$$\hat{Q}(s, a; \bar{\theta}); \quad \boxed{\bar{\theta} = \theta_0}$$

For the initialization

DQN Nature: how to “reduce” non-stationarity of targets

1. Changing the Q network at one step, we change the targets (this due to bootstrapping).

2. They propose to define a so-called target network: they consider a copy of the same network initialized with the same weights:

$$\hat{Q}(s, a; \bar{\theta}); \quad \bar{\theta} = \theta_0$$

3. This network is update every C steps with C > 1, by copying the weights of the other network.

4. The Q-learning target now is: $y_j = r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \bar{\theta})$

DQN pseudo code (in the ATARI case)

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode $= 1, M$ **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For $t = 1$,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    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 D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

A peculiar thing of the application: changing the images to black and white, the stack of different frames.... Think of this as features of the states

DQN pseudo code (in the ATARI case)

Replay buffer stuffs

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode $= 1, M$ **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For $t = 1$,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    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 D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

DQN pseudo code (in the ATARI case)

Replay buffer stuffs

Target net stuffs

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function Q with weights $\theta^{-} = \theta$

For episode $= 1, M$ **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For $t = 1$,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$

    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 D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

Other improvements/versions

1. Instead of periodic reset for the target network, we can use a **slow update** using the so called polyak averaging. It is just an exponential moving average:

$$\hat{\theta}_{i+1} = \tau\theta_{i+1} + (1-\tau)\bar{\theta}, \qquad \tau \in (0,1)$$

2. Double Q-Learning (initially proposed for Q-learning but also used for DQN): “Deep Reinforcement Learning with Double Q-learning”.

3. Distributional-based RL: C51, in A “Distributional Perspective on Reinforcement Learning”.

4. And many others

Thank you!
Questions?

Lecture #20:
Batch Methods & Intro to DeepRL

Gian Antonio Susto