Deep Reinforcement Learning

Seminar: Introduction to Deep Learning

Erik A. Daxberger

February 3, 2017

Motivation

Ultimate goal of ML/AI research: **general purpose intelligence**, i.e. general intelligent agent, excelling at a wide variety of human-level tasks

Real world is insanely complex $\rightarrow$ use **video games** as a testbed!

Is it possible to design a single AI agent, being able to play a wide variety of different games, **end-to-end**, at a human level?

Motivation

Recently, **DeepMind Technologies** published a paper succeeding at that task.

DeepMind

Shortly after, DeepMind was bought by Google for **$500M** and published a Nature cover paper

Motivation

(Oh and by the way, they recently managed to beat a human pro at Go)

So how did they do that?! $\rightarrow$ deep reinforcement learning!

Goal of this talk

1. good understanding of DeepMind's Atari paper
2. ability to read other state-of-the-art papers

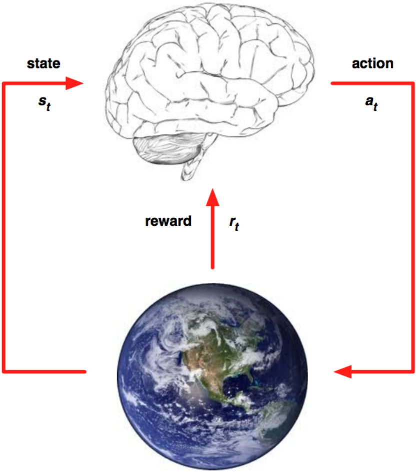
Contents

Contents

Machine Learning Paradigms

1. **Supervised learning**
   - in: example input/output pairs (by a "teacher")
   - out: general mapping rule
   - example: learn to detect dogs in images → classification
2. **Unsupervised learning**
   - in: inputs without outputs
   - out: hidden structure in the inputs
   - example: learn groups of different animals → clustering
3. **Reinforcement learning**
   - in: observations and feedback (rewards or punishments) from interacting with an environment
   - out: achieve some goal
   - example: learn to play Atari games from scratch

The RL Setting



RL is a **general-purpose framework** for AI

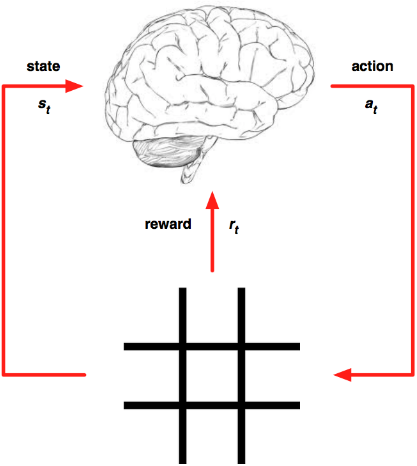
At each timestep t an **agent** interacts with an **environment** $\mathcal{E}$ and

- observes **state** $s_t \in \mathbb{R}^d$
- receives **reward** $r_t \in \mathbb{R}$
- executes **action** $a_t \in A$

Reward signals may be sparse, noisy and delayed.

$\rightarrow$ **Agent-environment-loop**

Example: TicTacToe



$$s_t \in \left\{ \begin{array}{c|c|c} O & & X \\ \hline & X & \\ \hline X & & O \end{array}, \begin{array}{c|c|c} X & & X \\ \hline & X & \\ \hline O & O & O \end{array}, \begin{array}{c|c|c} & O & \\ \hline & & \\ \hline X & & O \end{array}, \ldots \right\}$$

$$r_t = \begin{cases} +1 & \text{if we won} \\ -1 & \text{if we lost} \\ 0 & \text{otherwise} \end{cases}$$

$$a_t \in \left\{ \begin{array}{c|c|c} X & & \\ \hline & & \\ \hline & & \end{array}, \begin{array}{c|c|c} & X & \\ \hline & & \\ \hline & & \end{array}, \begin{array}{c|c|c} & & X \\ \hline & & \\ \hline & & \end{array}, \ldots \right\}$$

Reinforcement Learning vs. Deep Learning

Why don't we just use **Deep Learning**?!

Reinforcement Learning	**vs.**	**Deep Learning**
sparse and noisy rewards	vs.	hand-labeled training data
delay between actions and rewards ($\triangleq$ *credit assignment problem*)	vs.	direct association between inputs and outputs
highly correlated inputs from a non-stationary data distribution	vs.	i.i.d. data samples

Fundamental RL Concepts

Policy π tells the agent which actions to choose, given states

$$a = \pi(s)$$

Goal: Select actions to **maximize future reward** $\triangleq$ **return**

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$$

Value function $Q^\pi(s, a)$ defines the expected total reward from state s and action a under policy π ("How good is action a in state s?")

$$Q^\pi(s, a) = \mathbb{E}\left[R_t | s, a\right] = \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \middle| s, a\right]$$

Optimal value function

$$Q^*(s, a) = \max_\pi Q^\pi(s, a)$$

Example: TicTacToe

$$\pi(s) = \begin{cases} \begin{array}{c|c|c} & & X \\ \hline & & \\ \hline & & \end{array} & \text{if } s = \begin{array}{c|c|c} X & X & \\ \hline & O & \\ \hline O & & \end{array} \\ \begin{array}{c|c|c} & & \\ \hline & X & \\ \hline & & \end{array} & \text{if } s = \begin{array}{c|c|c} & & \\ \hline & & \\ \hline & & \end{array} \\ \dots \end{cases}$$

Let $\gamma = 0.9$

$$\begin{aligned} R_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \\ &= 0 + 0.9 * 0 + \dots + 0.9^5 * 1 \\ &\approx 0.59 \end{aligned}$$

$$Q^\pi(s, a) \quad = \quad \begin{array}{c|cccc} & \begin{array}{c|c|c} X & & X \\ \hline & O & \\ \hline & & \end{array} & \begin{array}{c|c|c} O & X & X \\ \hline & & \\ \hline & O & \end{array} & \dots & \begin{array}{c|c|c} & O & \\ \hline O & X & \\ \hline & & X \end{array} \\ \hline \begin{array}{c|c|c} X & & \\ \hline & & \\ \hline & & \end{array} & 0.3 & 0.5 & \dots & 0.9 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \begin{array}{c|c|c} & & \\ \hline & & \\ \hline & & X \end{array} & 0.6 & 0.1 & \dots & 0.8 \end{array}$$

Bellman Equation and Value Iteration

Optimal value function can be unrolled recursively $\rightarrow$ **Bellman equation**

$$\begin{aligned} Q^*(s,a) &= \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \middle| s,a\right] \\ &= \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') \middle| s,a\right] \end{aligned}$$

Value iteration algorithms estimate Q^* using iterative **Bellman updates**

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a') \middle| s,a\right]$$

This procedure is **guaranteed to converge**, i.e.

$$Q_i \rightarrow Q^* \quad \text{as} \quad i \rightarrow \infty$$

Function Approximators

These tabular methods are **highly impractical**

1. all Q-values are stored seperately $\rightarrow$ **technical challenge**
   Example: Chess
   - 10^{47} states
   - 35 possible moves/state
   - $\rightarrow 10^{47} * 35 * 1\text{Byte} \approx 10^{27}\text{Zettabytes}$

   (The Internet is 10 Zettabytes)

2. no generalization over unseen states $\rightarrow$ **"dumb" approach**

$\Longrightarrow$ use a **function approximator** to estimate Q!

Contents

DQN Idea

Approximate Q by an **ANN** with weights $\theta \rightarrow$ **deep Q-network (DQN)**

$$Q(s, a; \theta) \approx Q^{\pi}(s, a)$$

Loss function = MSE in Q-Values

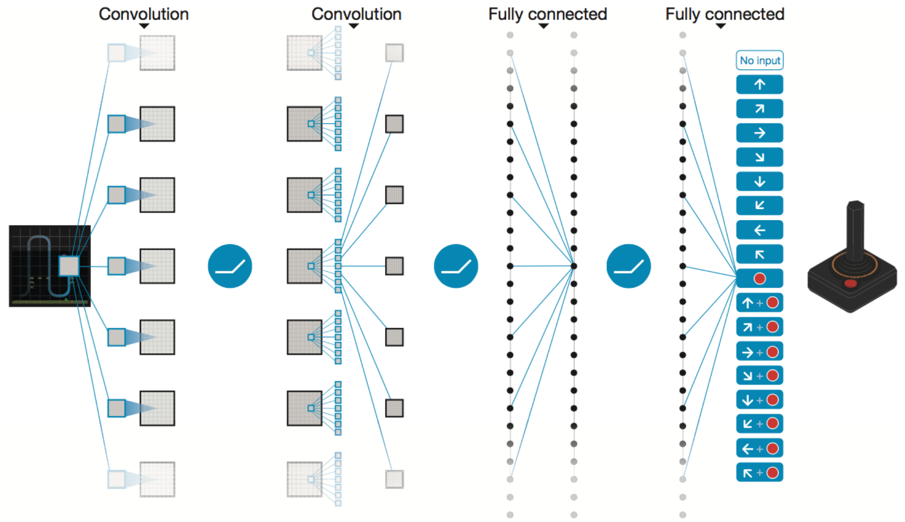
$$\mathcal{L}(\theta) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s', a'; \theta)}_{\text{target}} - Q(s, a; \theta)\right)^2\right]$$

Q-learning gradient

$$\nabla_\theta \mathcal{L}(\theta) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a; \theta)\right) \nabla_\theta Q(s, a; \theta)\right]$$

$\rightarrow$ Optimize $\mathcal{L}(\theta)$ with $\nabla_\theta \mathcal{L}(\theta)$ using **stochastic gradient descent**

DQN Architecture



Stability Issues & Solutions

Standard Q-learning **oscillates** or **diverges** when using ANNs

1. Sequential nature of data $\rightarrow$ non-i.i.d., as successive samples are (highly) correlated
2. Policy changes rapidly with slight changes to Q-values $\rightarrow$ policy may oscillate and data distribution may swing between extremes

Tweaks DQN uses to **stabilize training**

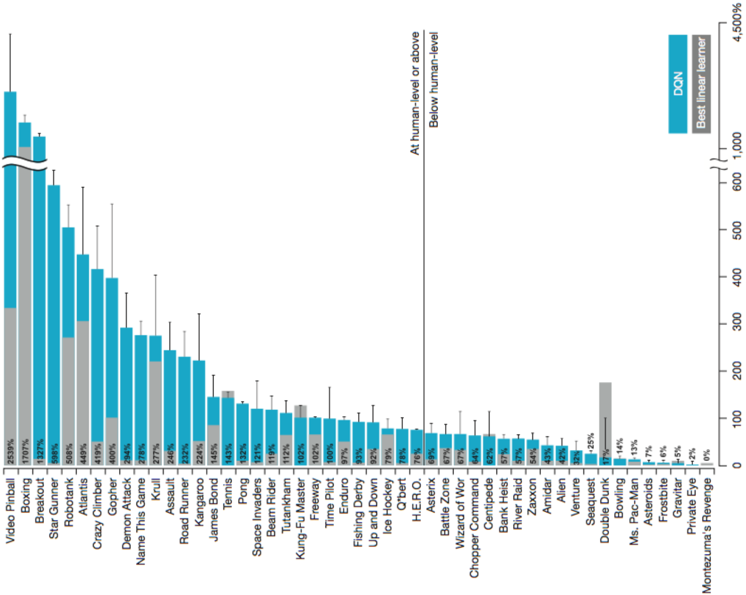
1. **Experience replay**: store agent's experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ in a data set $\mathcal{D} = \{e_1, \ldots, e_N\}$, into a *replay memory* $\rightarrow$ apply minibatch updates to samples $e \sim \mathcal{D}$
2. **Frozen target network**: hold previous parameters θ^- fixed in the Q-learning target when optimizing $\mathcal{L}_i(\theta)$, and update them only periodically, i.e. $\theta^- \leftarrow \theta$

DQN Pseudocode

Algorithm: Deep Q-learning

Initialize Q-function with random weights
for $episode = 1, \ldots, M$ **do**
  for $t = 1, \ldots, T$ **do**
    Choose $a_t = \begin{cases} \text{sampled randomly} & \text{with probability } \epsilon \\ \max_a Q^*(s_t, a; \theta) & \text{otherwise} \end{cases}$
    Execute a_t in emulator and observe reward r_t and image s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } s_{t+1} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta) & \text{for non-terminal } s_{t+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$
  end
end

DQN Atari Results



Contents

Framework

Goal: Agent that learns how to play Pong, end-to-end

- **input**: raw image frame in form of a $210 \times 160 \times 3$ byte array, i.e.

$$s_t \in \{0, \ldots, 255\}^{100.800}$$

- **output**: distribution over actions $\rho(a_t)$ ($\rightarrow$ *stochastic* policy, i.e., we sample $a_t \sim \rho(a_t)$ in every t), with

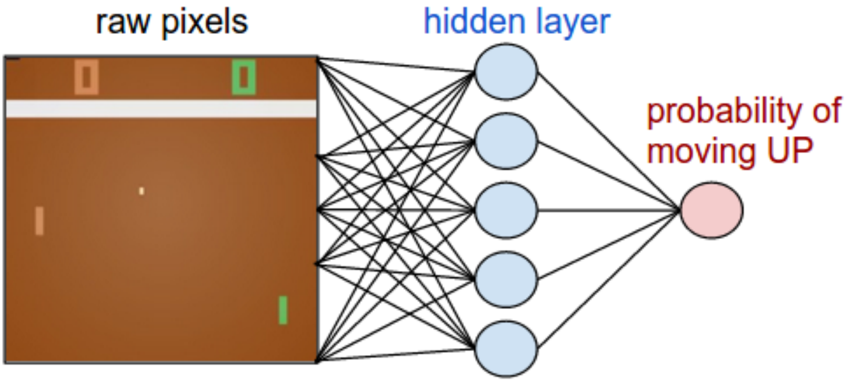
$$a_t \in \{\mathrm{UP}, \mathrm{DOWN}\}$$

- $\rightarrow$ game emulator executes a_t and emits s_{t+1} and a **reward**, i.e.,

$$r_t = \begin{cases} +1 & \text{if we score a point} \\ -1 & \text{if our opponent scores a point} \\ 0 & \text{otherwise} \end{cases}$$

Architecture

Policy network: 2-layer fully-connected ANN with 200 hidden nodes



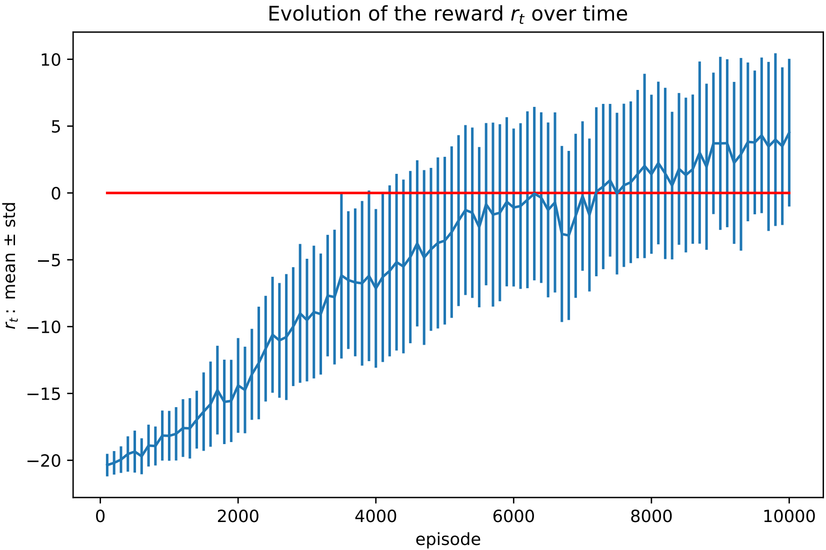
→ **Goal**: find optimal weights θ of the policy network

Implementation

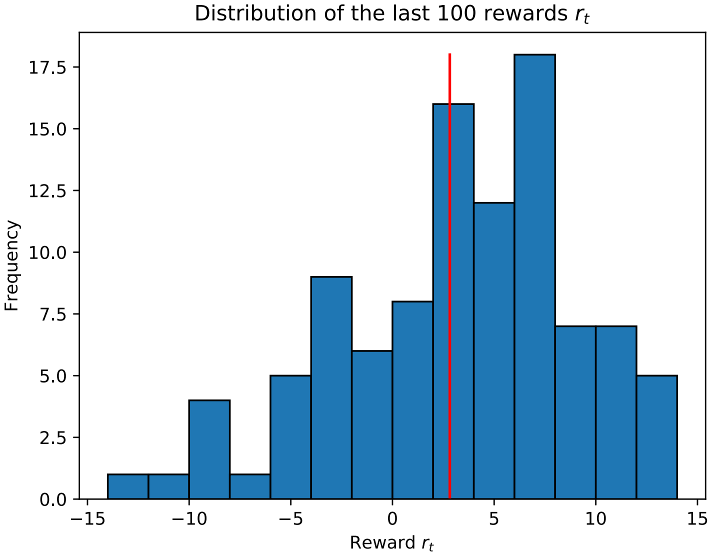
Implemented in Python, using only numpy and **OpenAI Gym**


```
import gym

env = gym.make("Pong-v0")
s_0 = env.reset()

for _ in range(1000):
    env.render()

    a_t = env.action_space.
   sample()

    s_t1, r_t1, done, info =
   env.step(a_t)
```

Results


Evolution of the reward r_t over time
10
5
0
−5
−10
−15
−20
r_t : mean ± std
0
2000
4000
6000
8000
10000
episode

Results


Distribution of the last 100 rewards r_t
Frequency
Reward r_t
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
−15
−10
−5
0
5
10
15

Contents

Summary

- RL is a **general-purpose AI framework**, modelled through the **agent-environment-loop** of actions, states and rewards
- it can solve problems that are difficult for supervised learning, i.e. involving **sparse, noisy and delayed rewards** and **correlated inputs**
- the goal of an RL agent is to find the **policy maximizing its return**
- this can be done by finding the **optimal value function**, using iterative **Bellman updates**, which is guaranteed to converge
- as storing Q in a table can be demanding for large problems and disables generalization, **function approximators** are commonly used
- for example, an **ANN** can be used to approximate Q, allowing us to optimize Q using standard **deep learning** techniques
- as this process may **oscillate/converge**, DQN uses **experience replay** and a **frozen target network** to stabilize training

Open Problems and Current Research

Despite the empirical successes, there are still many fundamental **unsolved problems**.

For example, current algorithms

- are **bad at long-term planning** (i.e. act myopically)
- need a **long training time** (i.e., have a low data efficiency)
- are **unable to understand abstract concepts** (i.e. lack a model)

Some **state-of-the-art** papers (Deep RL Workshop @ NIPS 2016)

- Duan, Yan, et al. *RL^2: Fast Reinforcement Learning via Slow Reinforcement Learning*
- Jaderberg, Max, et al. *Reinforcement Learning with Unsupervised Auxiliary Tasks*
- He, Frank S., et al. *Learning to Play in a Day: Faster Deep Reinforcement Learning by Optimality Tightening*

Now what about AlphaGo?!

Chess	vs.	Go
10^{47} states	vs.	10^{170} states
rule-based	vs.	intuition-based

So how does AlphaGo work?

Deep reinforcement learning

+

Monte Carlo tree search

→ Silver, David, et al. *Mastering the game of Go with deep neural networks and tree search.* Nature 529.7587 (2016): 484-489.

Thank you for your attention!

References

- Mnih, Volodymyr, et al. *Playing atari with deep reinforcement learning.* arXiv preprint arXiv:1312.5602 (2013).
- Mnih, Volodymyr, et al. *Human-level control through deep reinforcement learning.* Nature 518.7540 (2015): 529-533.
- Sutton, Richard S., and Andrew G. Barto. *Reinforcement learning: An introduction.* Vol. 1. No. 1. Cambridge: MIT press, 1998.
- Karpathy, Andrej. *Deep Reinforcement Learning: Pong from Pixels.* `http://karpathy.github.io/2016/05/31/rl/` (21.01.2017).
- Brockman, Greg, et al. *OpenAI gym.* arXiv preprint arXiv:1606.01540 (2016) and `https://gym.openai.com/` (21.01.2017).
- Silver, David. *Deep Reinforcement Learning.* Tutorial at ICLR (2015), `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Resources_files/deep_rl.pdf` (21.01.2017).