# Autonomous Driving and Reinforcement Learning – an Introduction

Michael Bosello
michael.bosello@studio.unibo.it

Università di Bologna – Department of Computer Science and Engineering, Cesena, Italy

Smart Vehicular Systems A.Y. 2019/2020

# Outline

# Next in Line...

# Self-Driving I

## Goals

- Reach the destination
- Avoid dangerous states
- Provide comfort to passengers (e.g. avoid oscillations, sudden moves)

## Settings

- Multi-agent problem
  - One have to consider others
- Interaction is both
  - Cooperative, all the agents desire safety
  - Competitive, each agent has its own goal
- Environment
  - Non-deterministic
  - Partially observable
  - Dynamic

# Self-Driving II

## Three sub-problems

**Recognition** Identify environment's components

**Prediction** Predict the evolution of the surrounding

**Decision making** Take decisions and act to pursue the goal

## Two approaches

- Single task handling
  - Use human ingenuity to inject knowledge about the domain
- End-to-end
  - Let the algorithm optimize towards the final goal without constrains

# Self-Driving III

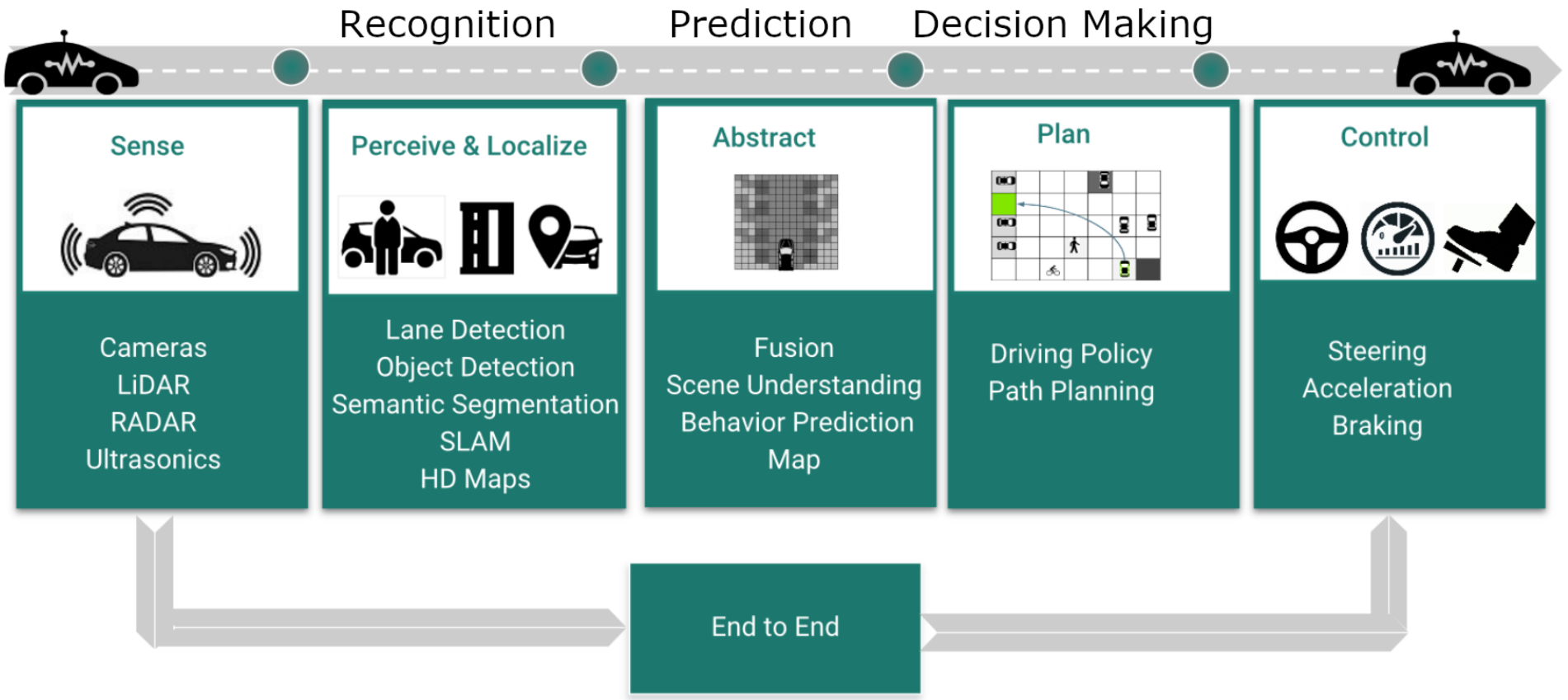



Figure: Modules of autonomous driving systems (source [Talpaert et al., 2019] - modified)

# Sensing & Control

## Sensing

- Which sensors?
- Single/multiple
- Sensor expense
- Operating conditions

## Control

- Continuous – steering, acceleration, braking
- Discrete – forward, right, left, brake

# Recognition

- Convolutional Neural Networks (CNNs)
  - YOLO [Redmon et al., 2015]
- Computer Vision techniques for object detection
  - Feature based
  - Local descriptor based
- Mixed

# Prediction

- Model-based
  - Physics-based
  - Maneuver-based
  - Interaction-aware
- Data-driven
  - Recurrent NN (like Long-Short Term Memory LSTM)

# Decision Making

- Planning/Tree Search/Optimization
- Agent-oriented programming (like the BDI model)
- Deep Learning (DL)
- Reinforcement Learning (RL)

# End-to-end

## Supervised Learning

- Based on imitation
- Current approach by major car manufacturer
- **Robotic drivers are expected to be perfect**

### Drawbacks

**Training data** Huge amounts of labeled data or human effort

**Covering all possible driving scenarios** is very hard

## Unsupervised Learning

- Learns behaviors by trial-and-error
  - like a young human driving student

- Does not require explicit supervision from humans.

- Mainly used on simulated environment
  - to avoid consequences in real-life

# Simulations

## Keep in mind

- An agent trained in a virtual environment will not perform well in the real setting
  - For cameras: different visual appearance, especially for textures
  - You need some kind of transfer learning or conversion of images/data
- Always enable noises of sensors/actuators
- Some “grand truth” sensors present in some simulators aren’t available in real life
  - For example, in TORCS (a circuit simulator popular in the ML field) you can have exact distances from other cars and from the borders. Too easy...

# Next in Line...

# Agent-environment interaction

- In RL, an agent learns how to fulfill a task by interacting with its environment
- The interaction between the agent and the environment can be reduced to:

| | |
|---|---|
| State | Every information about the environment useful to predict the future |
| Action | What the agent do |
| Reward | A real number that Indicates how well the agent is doing |

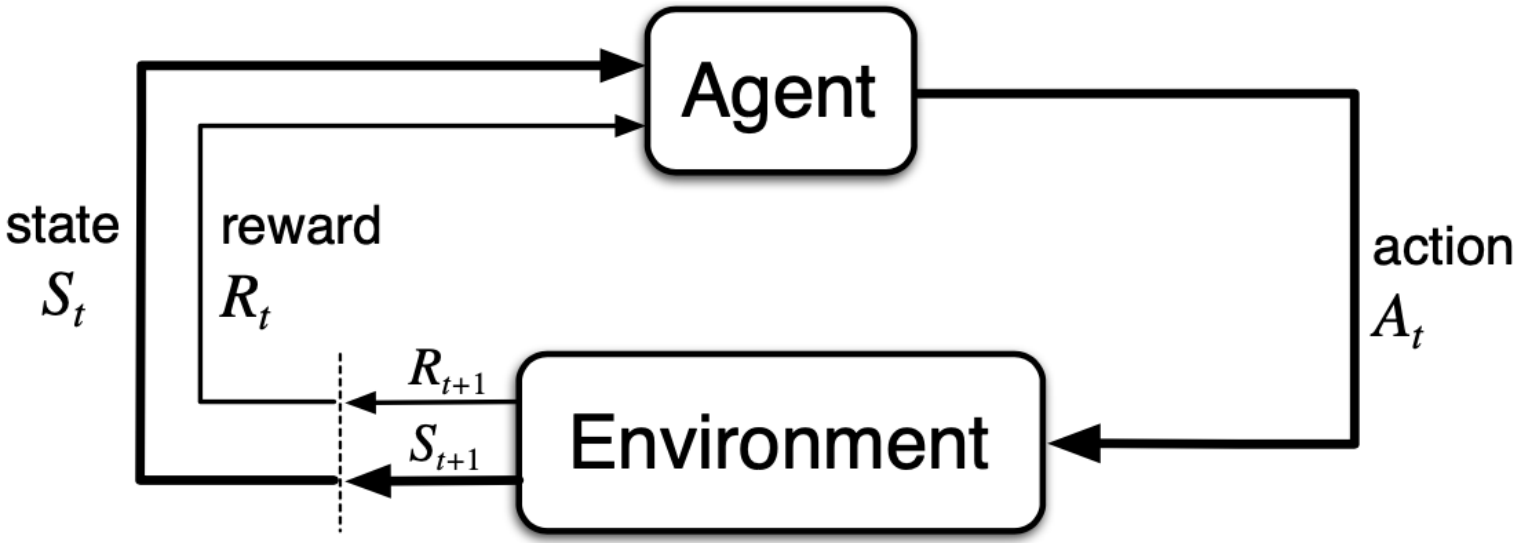



Figure: At each ***time step*** $t$ the agent perform an action $A_t$ and receives the couple $S_{t+1}, R_{t+1}$ (source [Sutton and Barto, 2018])

# Terminology

## Episode

- A complete run from a starting state to a final state

**Episodic task** A task that has an end

**Continuing task** A task that goes on without limit

- We can split it e.g. by setting a maximum number of steps

# Markov Decision Process (MDP)

A RL problem can be formalized as a MDP. The 4-tuple:

- $S$ set of states
- $A_s$ set of actions available in state $s$
- $P(s'|s,a)$ state-transition probabilities (probability to reach $s'$ given $s$ and $a$)
- $R(s,s')$ reward distribution associate to state transitions

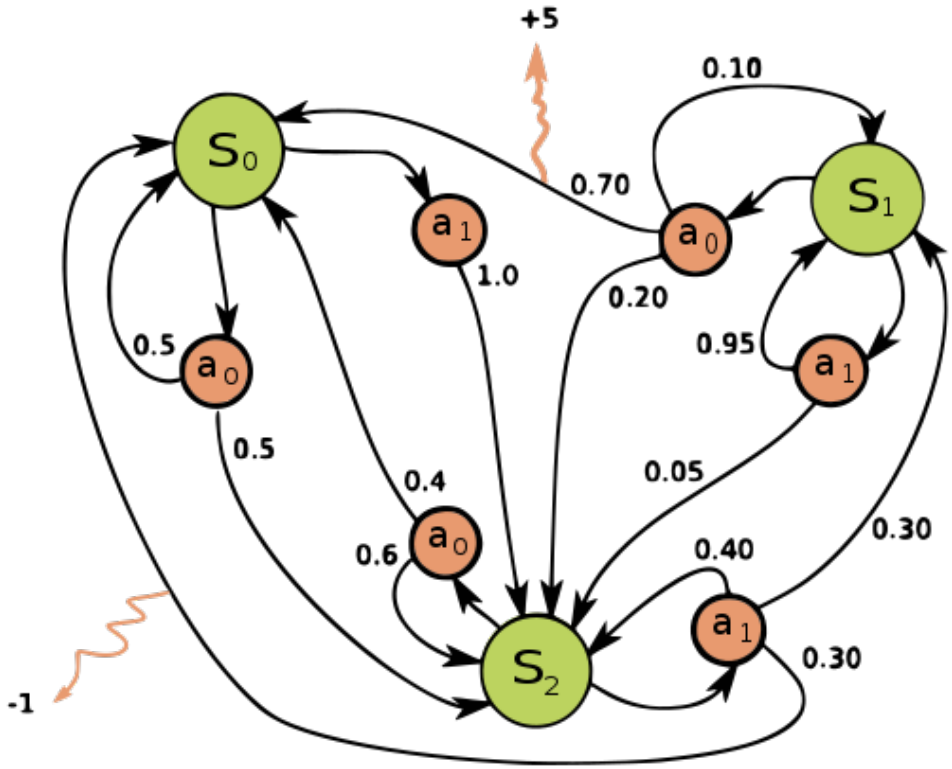



Figure: Example of a simple MDP (source Wikipedia)

# Environment features

- MDPs can deal with non-deterministic environments (because they have probability distributions)
- But we need to cope with observability

## Markov Property – Full observability

- Classical RL algorithms rely on the Markov property
- A state satisfy the property if it includes information about all aspects of the (past) agent-environment interaction that make a difference for the future

## From states to *observations* – Partial observability

- Sometimes, an agent has only a partial view of the world state
- The agent gets information (observations) about some aspects of the environment
- Abusing the language, observations/history are called "state" and symbolized $S_t$

## How to deal with partial observability?

- Approximation methods (e.g. NN) doesn't rely on the Markov property
- Sometimes we can reconstruct a Markov state using a history
  - Stacking the last n states
  - Using a RNN (e.g. LSTM)

# Learning a Policy I

We want the agent to learn a strategy i.e. a policy

## Policy $\pi$

A map from states to actions used by the agent to choose the next action

In particular, we are searching for the optimal policy
$\rightarrow$ The policy that maximize the cumulative reward

## Cumulative reward

The discounted sum of rewards *over time*

$$R = \sum_{t=0}^{n} \gamma^t r_{t+1} \mid 0 \leq \gamma \leq 1$$

For larger values of $\gamma$, the agent focuses more about the long term rewards
For smaller values of $\gamma$, the agent focuses more about the immediate rewards

## What does this mean?

- The actions of the agent affect its possibilities later in time
- It could be better to choose an action that led to a better state than an action that gives you a greater reward but led to a worse state

# Learning a Policy II

## Problem

we don't know the MDP model (state transition probabilities and reward distribution)

## Solution

Monte Carlo approach: let's make estimations based on experience

- A way to produce a policy is to estimate the value (or action-value) function
  - given the function, the agent needs only to perform the action with the greatest value

## Expected return $\mathbb{E}[G_t]$

The expectation of cumulative reward i.e. our current estimation

## Value function $V_\pi(s) = \mathbb{E}[G_t|S_t = s]$

- (state) $\rightarrow$ expected return when starting in $s$ and following $\pi$ thereafter

## Action-value function $Q_\pi(s, a) = \mathbb{E}[G_t|S_t = s, A_t = a]$

- (state, action) $\rightarrow$ expected return performing $a$ and following $\pi$ thereafter

# Learning a Policy III

## SARSA and Q-learning

- They are algorithms that approximate the action-value function
- At every iteration the estimation is refined thanks to the new experience.
- Every time the evaluation becomes more precise, the policy gets closer to the optimal one.

## Generalized Policy Iteration (GPI)

- Almost all RL methods could be described as GPI
  - They all have identifiable policies and value functions
- Learning consist of two interacting processes

  Policy evaluation Compute the value function
  Policy improvement Make the policy greedy

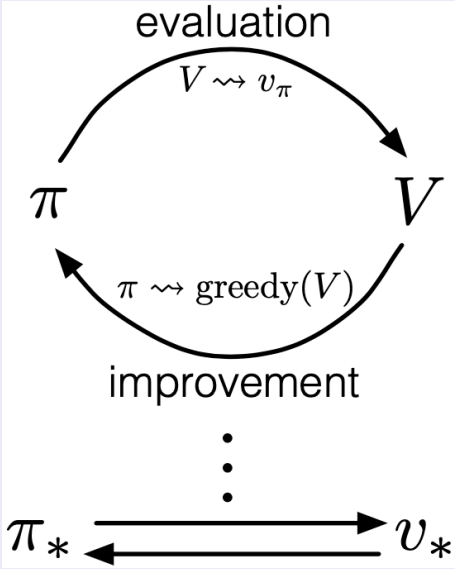



Figure: [Sutton and Barto, 2018]

# Learning a policy IV

## The exploration/exploitation dilemma

- The agent seeks to learn action values of optimal behavior, but it needs to behave non-optimally in order to explore all actions (to find the optimal actions)
- $\varepsilon$-greedy policy: the agent behave greedily but there is a (small) $\varepsilon$ probability to select a random action.

## On-policy learning

- The agent learns action values of the policy that produces its own behavior ($q_\pi$)
- It learns about a near-optimal policy ($\varepsilon$-greedy)
- Target policy and behavior policy correspond

## Off-policy learning

- The agent learns action values of the optimal policy ($q_*$)
- There are two policies: target and behavior
- Every $n$ steps the target policy is copied into the behavior one
  - If $n = 1$ there is only one action-value function
  - But it's not the same of on-policy (the target and behavior policy still not correspond)

# How to estimate the value function?

- A table that associate state (or state-action pairs) to a value
- Randomly initialized

## Temporal Difference (TD) update

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

## TD error

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

- $R_{t+1} + \gamma V(S_{t+1})$ is a better estimation of $V(S_t)$
  - The actual reward obtained plus the expected rewards for the next state
- By subtracting the old estimate we get the error made at time $t$

## Sarsa (on-policy)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

## Q-learning (off-policy)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

# Next in Line...

# Deep Q-Network

## DQN

- When the number of states increases, it becomes impossible to use a table to store the Q-function (alias for action-value function)
- A neural network can approximate the Q-function
- We also get better generalization and the ability to deal with non-Markovian envs

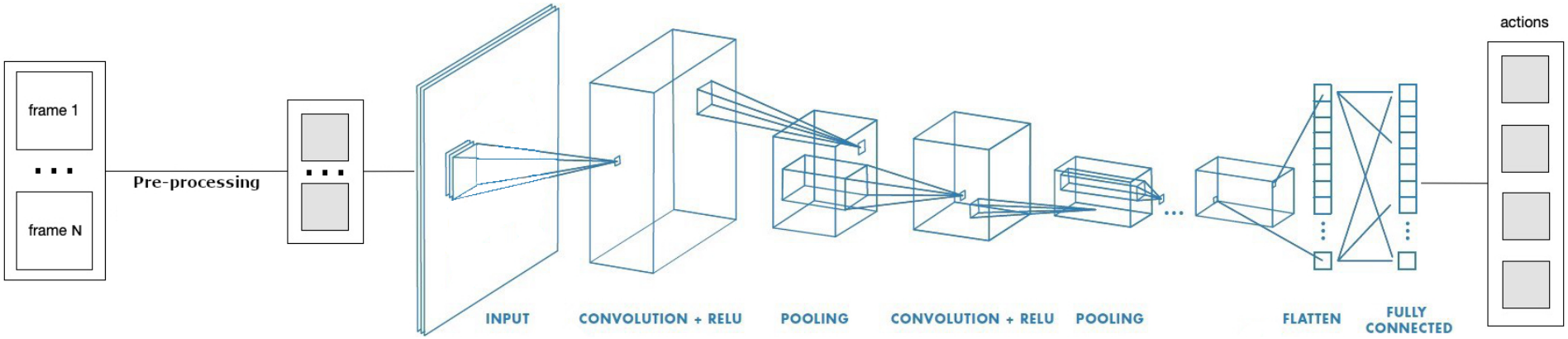
## How to represent the action-value function in a NN?

1. Input: the state (as a vector) $\rightarrow$ Output: the values of each action
2. Input: the state and an action $\rightarrow$ Output: the value for that action

## Design the driver agent

- Environment: states and actions, reward function
- Neural network
- Training cycle

# Neural Network



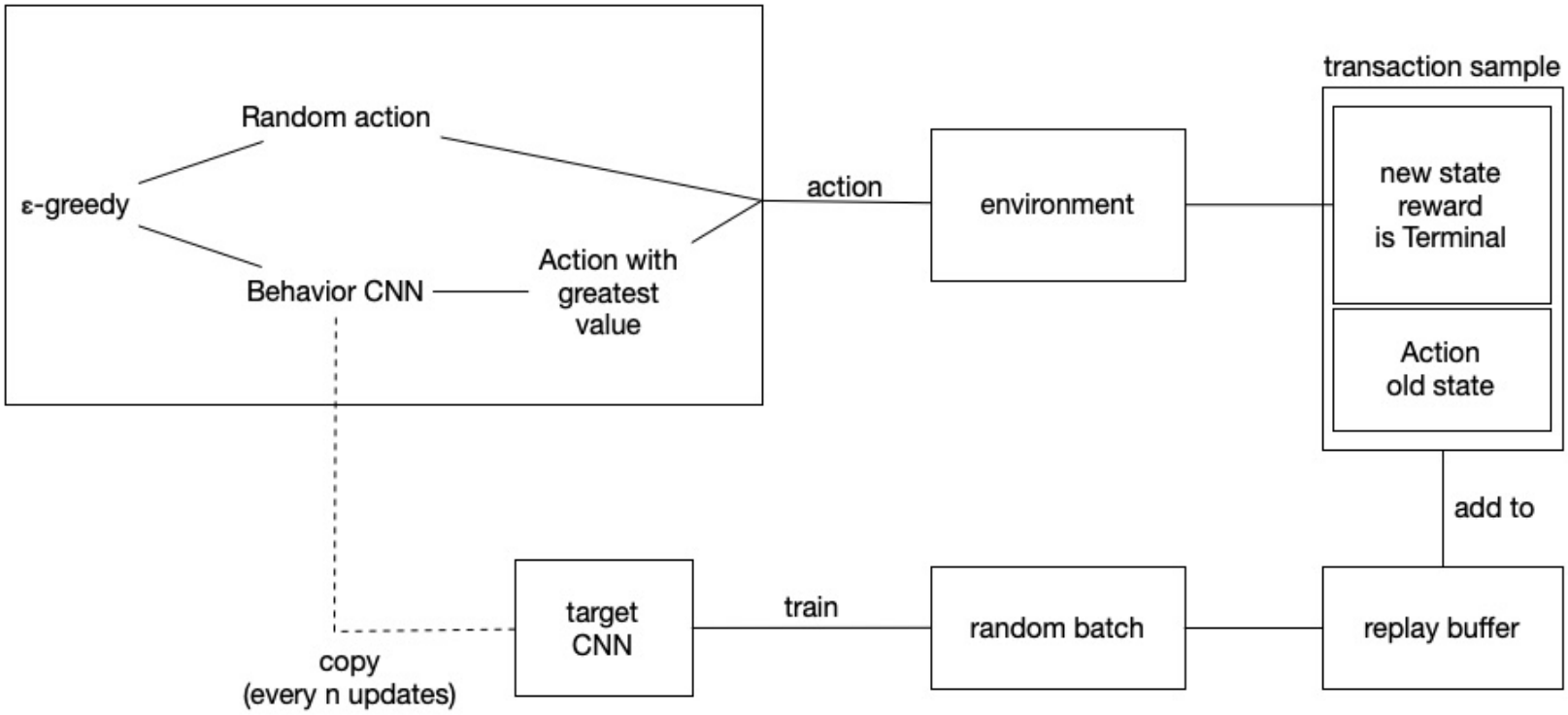

## Pre-processing

- It's important how we present the data to the network
- Let's take images as an example
  - Resize to small images (e.g. 84x84)
  - If colors are not discriminatory, use grayscale
  - How many frames to stack?
  - Remember: the larger the input vector, the larger the searching space

## Convolutional Neural Networks

- A good idea for structured/spatially local data
  - e.g. images, lidar data ...

# Training Cycle




- Transaction samples are used to compute the target update
- Training use a gradient form of Q-learning ([Mnih et al., 2013] and [Sutton and Barto, 2018] for details)

# DQN Issues and Solutions I

Each of these features significantly improve performance

## Experience Replay

- We put samples in a buffer and we take random batches from it for training
- When full, oldest samples are discarded
- Why? Experience replay is needed to break temporal relation
  - In supervised learning, inputs should be independent and identically distributed (i.i.d.) i.e. the generative process have no memory of past generated samples
- Other advantages:
  - More efficient use of experience (samples are used several times)
  - Avoid catastrophic forgetting (by presenting again old samples)

## Double DQN

- We have two networks, one behavior NN and one target NN
- We train the target NN
- Every N steps we copy the target NN into the behavior one
- Why? To reduce target instability
  - Supervised learning to perform well requires that for the same input a label does not change over time
  - In RL, the target change constantly (as we refine the estimations)

# DQN Issues and Solutions II

## Rewards scaling

Scaling rewards in the range $[-1; 1]$ dramatically improves efficiency [Mnih et al., 2013]

## Frame skipping

- Consequent frames are very similar $\rightarrow$ we can skip frames without loosing much information and speed up training by N times
- The selected action is repeated for N frames

## Initial sampling

It could be useful to save K samples in the replay buffer before start training

## So many parameters...

- [Mnih et al., 2015] is a good place to start
  - The table of hyperparameters used in their work is in the next slide
- More recent works and experiments could provide better suggestions

# Hyperparameters used in [Mnih et al., 2015]

**Extended Data Table 1 | List of hyperparameters and their values**

| Hyperparameter | Value | Description |
|---|---|---|
| minibatch size | 32 | Number of training cases over which each stochastic gradient descent (SGD) update is computed. |
| replay memory size | 1000000 | SGD updates are sampled from this number of most recent frames. |
| agent history length | 4 | The number of most recent frames experienced by the agent that are given as input to the Q network. |
| target network update frequency | 10000 | The frequency (measured in the number of parameter updates) with which the target network is updated (this corresponds to the parameter *C* from Algorithm 1). |
| discount factor | 0.99 | Discount factor gamma used in the Q-learning update. |
| action repeat | 4 | Repeat each action selected by the agent this many times. Using a value of 4 results in the agent seeing only every 4th input frame. |
| update frequency | 4 | The number of actions selected by the agent between successive SGD updates. Using a value of 4 results in the agent selecting 4 actions between each pair of successive updates. |
| learning rate | 0.00025 | The learning rate used by RMSProp. |
| gradient momentum | 0.95 | Gradient momentum used by RMSProp. |
| squared gradient momentum | 0.95 | Squared gradient (denominator) momentum used by RMSProp. |
| min squared gradient | 0.01 | Constant added to the squared gradient in the denominator of the RMSProp update. |
| initial exploration | 1 | Initial value of $\varepsilon$ in $\varepsilon$-greedy exploration. |
| final exploration | 0.1 | Final value of $\varepsilon$ in $\varepsilon$-greedy exploration. |
| final exploration frame | 1000000 | The number of frames over which the initial value of $\varepsilon$ is linearly annealed to its final value. |
| replay start size | 50000 | A uniform random policy is run for this number of frames before learning starts and the resulting experience is used to populate the replay memory. |
| no-op max | 30 | Maximum number of "do nothing" actions to be performed by the agent at the start of an episode. |

The values of all the hyperparameters were selected by performing an informal search on the games Pong, Breakout, Seaquest, Space Invaders and Beam Rider. We did not perform a systematic grid search owing to the high computational cost, although it is conceivable that even better results could be obtained by systematically tuning the hyperparameter values.

# Last things

## Reward function

- Designing a reward function is not an easy task
- It could severely affect learning
- In the case of a driving agent (like in RL for robotics), the rewards are internal and doesn't come from the environment
- You should give rewards also for intermediate steps to direct the agent (i.e. not only $+1$ when successful and $-1$ when it fails)
- Some naive hints
  - Reach target $+1$
  - Crash $-1$
  - Car goes forward $+0.4$ (if too big it will inhibit crash penalty)
  - Car turns $+0.1$ (if too big it will keep circle around)
  - $-0.2$ instead of the positive reward if the agent turns to a direction and just after it turns to the opposite one (i.e. right-left or left-right) to reduce car oscillations

## Exploration/exploitation

- Start with a big $\varepsilon$
- Gradually reduce it with an $\varepsilon$-decay (e.g. 0.999)
- $\varepsilon_{t+1} = \varepsilon_t * decay$

# Appendix: Reading Suggestions

# Reading Suggestions: Access to Resources

- Books and papers are available through the university proxy
- Search engine: `https://sba-unibo-it.ezproxy.unibo.it/AlmaStart`
- If you want to access an article by link
  converts dots in scores and add .ezproxy.unibo.it at the end of the url e.g:
  `https://www.nature.com/articles/nature14236`
  ↓
  `https://www-nature-com.ezproxy.unibo.it/articles/nature14236`

# Reading Suggestions: Driving Problem

- Extended review of autonomous driving approaches [Leon and Gavrilescu, 2019]
- Overview of RL for autonomous driving and its challenges [Talpaert et al., 2019]
- Tutorial of RL with CARLA simulator `https://pythonprogramming.net/introduction-self-driving-autonomous-cars-carla-python`

## End-to-end approach examples

- CNN to steering on a real car [Bojarski et al., 2016]
- DQN car control [Yu et al., 2016]

## Single task approach example

- NN for recognition + RL for control [Li et al., 2018]

## Converting virtual images to real one

- [Pan et al., 2017]

# Reading Suggestions: RL & DL

## In depth (books)

- [Sutton and Barto, 2018] is the bible of RL – the second version (2018) contains also recent breakthroughs
- [Goodfellow et al., 2016] for Deep Learning

## Compact introductions

- Briefly explanation of RL basics and DQN: `https://rubenfiszel.github.io/posts/rl4j/2016-08-24-Reinforcement-Learning-and-DQN`
- CNN: `https://cs231n.github.io/convolutional-networks/`
  - Even more compact: `https://pathmind.com/wiki/convolutional-network`

## DQN

- [Mnih et al., 2015] Revised version (better)
- [Mnih et al., 2013] Original version
- [van Hasselt et al., 2015] Double Q-learning

# References I



Bojarski, M., Testa, D. D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., and Zieba, K. (2016).
End to end learning for self-driving cars.


Goodfellow, I., Bengio, Y., and Courville, A. (2016).
*Deep Learning*.
MIT Press.
http://www.deeplearningbook.org.


Leon, F. and Gavrilescu, M. (2019).
A review of tracking, prediction and decision making methods for autonomous driving.


Li, D., Zhao, D., Zhang, Q., and Chen, Y. (2018).
Reinforcement learning and deep learning based lateral control for autonomous driving.


Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013).
Playing atari with deep reinforcement learning.

# References II



Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015).
Human-level control through deep reinforcement learning.
*Nature*, 518(7540):529–533.


Pan, X., You, Y., Wang, Z., and Lu, C. (2017).
Virtual to real reinforcement learning for autonomous driving.


Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2015).
You only look once: Unified, real-time object detection.


Sallab, A. E., Abdou, M., Perot, E., and Yogamani, S. (2017).
Deep reinforcement learning framework for autonomous driving.
*Electronic Imaging*, 2017(19):70–76.


Sutton, R. S. and Barto, A. G. (2018).
*Reinforcement learning : an introduction*.
The MIT Press.

# References III


Talpaert, V., Sobh, I., Kiran, B. R., Mannion, P., Yogamani, S., El-Sallab, A., and Perez, P. (2019).
Exploring applications of deep reinforcement learning for real-world autonomous driving systems.

van Hasselt, H., Guez, A., and Silver, D. (2015).
Deep reinforcement learning with double q-learning.

Yu, A., Palefsky-Smith, R., and Bedi, R. (2016).
Deep reinforcement learning for simulated autonomous vehicle control.
*Course Project Reports: Winter*, pages 1–7.

# Autonomous Driving and Reinforcement Learning – an Introduction

Michael Bosello
michael.bosello@studio.unibo.it

Università di Bologna – Department of Computer Science and Engineering, Cesena, Italy

Smart Vehicular Systems A.Y. 2019/2020