University of Padua

Department of Mathematics
"Tullio Levi-Civita"

Master Degree in Computer Science

Addressing State Representation in Deep Reinforcement Learning

A critical analysis of state-of-the-art methods

Master Thesis

Supervisor
Professor Alessandro Sperduti

Master Student
Fiammetta Cannavò

Academic year
2021-2022

To us, who finally did it.

... since human beings cannot program a computer to be intuitive or creative for the very good reason that we do not know what we ourselves do when we exercise these qualities.

Isaac Asimov

Acknowledgments

THANK YOU to my supervisor, for the help and all advice he gave me in the last few months of work to this project.

Thank you to my friend Laura, for constantly being my source of laugh and emotional support in the last few years.

Thank you to Lorenzo, for always being there, no matter what.

Thank you to my family, for always supporting (and bearing) me, giving me the discipline I needed when I was not able to provide it myself.

And finally, thanks to my cat, for hardly ever being there when I wanted to annoy it between one paragraph of the thesis and another.

Abstract

Reinforcement learning is one of the most relevant research areas in machine learning: it allows to model scenarios where an autonomous agent has to gather information about its environment and to learn the best policy to solve its task, which usually means the one that yelds the largest final reward. A modern development of this methods is deep reinforcement learning, in which the look-up table of rewards is substituted by a neural network that learns those rewards, aiming at the same time to learn efficient state representations of the system. This is particularly desirable when the information the agent receives from the environment is complex and high-dimensional. In this work we aim to survey come of the most recent state representation approaches in deep reinforcement learning, to classify and to analyze them in terms of efficiency and applicability in real scenarios. We finally present some possible future research directions in this field.

Contents

List of Figures

List of Tables

1 Introduction

Reinforcement learning is one of the most interesting fields of machine learning: it allows to model all those scenarios where an agent needs to be trained on a reward and punishment mechanism. The agent is rewarded for correct decisions and punished for the wrong ones, allowing it to learn which moves increase its reward and which reduce it, and to always prefer the one that maximizes the reward. The possible applications are limitless: news recommendation, where the system must decide which articles to show to a user, thus many features have to be taken into account, like their interests, the time they daily dedicate to news reading, the news they like the most; self-driving cars have to learn optimal driving policies taking into account all the obstacles and potential dangers a moving vehicle can encounter, and all the parameters it has to consider whenever it makes a move (adjacent vehicle, pedestrians, position in the lane, distance from destination, obstacles on the lane, navigation direction...); game playing, where the agent has to learn the game rules and the best strategies for beating the opponent; but also translation or question answering in natural language processing, optimization in large-scale production systems, robotics manipulation, query optimization, finance and trading, and more.

However, classic reinforcement learning approaches to store all information about state-action rewards in a lookup table, in which each entry corresponds to a (state, action) pair and contains the reward associated with that combination. It is easy to see that this modality presents many issues: first, this table needs to be accessed or updated every time a reward value is needed or a new one is computed, respectively. Moreover, the table has to be stored in memory, which can be extremely expensive if the size of the table grows. Unfortunately, this happens in many applications: if the environment is complex (as real environments often are), the combination of states and actions can grow exponentially with data size, resulting in a potentially giant table. The other side of the problem is that, to fill a large table, a large number of data samples is needed, and this is often not possible in real scenarios: many interactions with the environment are necessary to ensure that each entry in the table is updated a sufficient number of times to get a good estimate of the correct value, and this can become really expensive. In other words, a simple reinforcement learning setting would probably not be able to properly model any of the applicative scenarios described above.
Integrating deep learning methods into reinforcement learning systems can help solve some of these problems: a deep neural network is trained to learn an approximate policy for the reinforcement learning problems, without the need to store anything in a table. However, the problem of high sample complexity remains also in deep reinforcement learning systems: one way to face this issue is to use efficient input representations, to significantly improve the models' performance.
Efficient data representation is a crucial point in almost all deep learning models: these representations should be low dimensional (that is, significantly smaller than the data dimensionality) and should encode essential information (for a specific task) while discarding the many irrelevant aspects of the original data.

In this survey, numerous deep reinforcement learning systems are studied and analyzed, with a specific focus on their state representation choices, the way they are implemented, and the reasons that have led to those choices.
This work starts with a machine learning background of reinforcement learn-

ing, state representation learning and deep learning; a quick presentation is also given of the most common deep reinforcement learning algorithms used in literature (and, in particular, in the works analyzed here). Then, all the approaches studied for this survey are presented, classified by the shape or the structure of the state representation used; these methods are compared and discussed in their applicability and generalizability. After that, conclusions are drawn from the analysis of the previous chapter, and some future perspectives of research; finally, a rich appendix includes all the theoretical details useful for deeply understanding the methods presented in this survey.

2 Reinforcement Learning background

In this chapter the theoretical basis of this work is presented: first Reinforcement Learning (RL) is introduced, then the problem of state representation (SRL) is addressed and finally Deep Learning (DL) is presented in its main features but, specifically, in its more common usages in RL contexts.

2.1 Reinforcement Learning

Reinforcement learning is a general-purpose framework for decision-making in contexts where an autonomous agent aims to learn the optimal actions it needs to perform to achieve its goal[65]. At each step, the agent performs an action, to which corresponds a reward, a neutral outcome or a punishment, based on the state that is reached (whether it is desirable, or in any way closer to the goal, or not); each action alters the agent's state in some way. The aim of the agent is to learn the best policy, i.e. the state-action function maximizing the expected reward starting from any state. A scheme of a reinforcement learning system is represented in Figure 2.1.
Different settings can vary on the kind of actions' outcomes (deterministic

or nondeterministic), and the absence or presence of prior knowledge of the agent about the environment's responses to its actions. In some applications, for example, not each state corresponds to a reward, and the reward may also be given just at the end of the exploration (or the end of the task in general); however, regardless of when it is provided, the reward is necessary if the agent has to learn the policy, because the agent's criterion for selecting the best states (or sequence of states) is the final reward the chosen trajectory provides: the higher is the cumulative reward associated to a state sequence, the more likely will be the agent to choose the policy that selects that sequence [3].

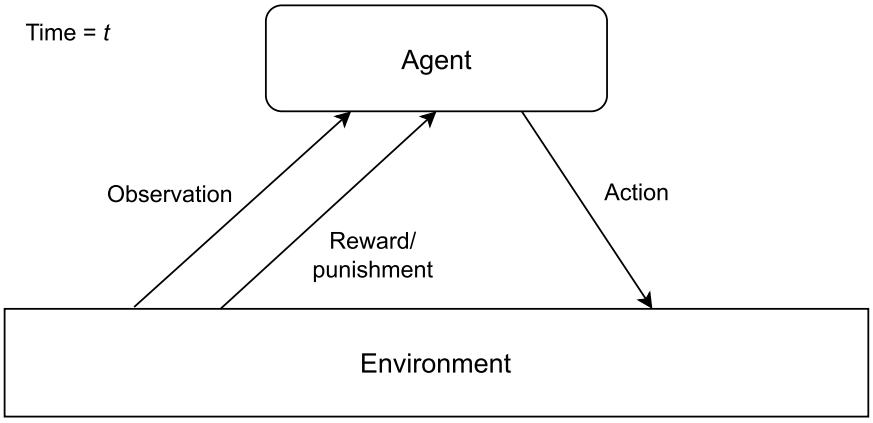

Figure 2.1: An agent interacting with the environment: it gets the observation of the states, performs an action and usually receives a reward or penalty as a result of the action.

2.1.1 What is different in reinforcement learning

The main difference between reinforcement learning and supervised learning is the absence of a "teacher" who tells whether each choice is good or not: RL is also called "learning with a *critic*", where the critic is who gives a (possibly incomplete) feedback only *after* the choice is made, and thus only on past actions. From this partial feedback (the reward), the agent has to understand which particular actions have led to the final result, in order to be able to replicate that result or to re-use the best actions when required. Thus, the agent assigns to each intermediate state an internal value based on how close it is to the goal, so that its task becomes to choose, step by step, the action which

leads to the best neighbouring state (i.e., the one with the most desirable internal value). This is usually defined as a *credit assignment problem*[3].

Differently from other kinds of learners, the RL agent directly influences the data it receives, because it depends on the specific action sequence the agent chooses. Thus, the agent has also to determine the best tradeoff between choosing actions that aim to explore new parts of the environment, in order to gain more information, and already performed actions that have brought a high reward: this is called the exploration-exploitation tradeoff.

Finally, RL can help model real scenarios in which the agent can access only partial information about its state, so it may need to perform actions aiming just to gather more information about the environment (so, to explore), or where the agent has to learn different tasks in the same environment, where past experience of actions and states for old tasks can help reducing sample complexity of new tasks.

2.1.2 Formal definitions

The nomenclature adopted in this work is very close to the one most commonly used in literature.
The agent performs action $a_t \in \mathcal{A}$ in the environment $\mathcal{E}$, where a_t is the action executed at time t and $\mathcal{A}$ is the action set of the agent. From $\mathcal{E}$ the agent may get some data, through its sensors, in the form of observations $o_t \in \mathcal{O}$, where o_t is the observation obtained at time t and $\mathcal{O}$ is the observation space. Performing an action makes the agent transition from state s_t to state s_{t+1}, both ground states in the agent's ground state space $\mathcal{S}$. Each action may correspond a reward r_t, whose value varies based on how close to the goal is the reached state, or more in general how much desirable the state is for the agent's task. Finally, the agent usually creates an efficient representation of the state it is in, which here is denoted as $\tilde{s}_t \in \tilde{\mathcal{S}}$, where $\tilde{s}_t$ is the representation of the state reached at time t and $\tilde{\mathcal{S}}$ is the state representation space.
The nomenclature is summed up in Table 2.1.

Entity	Symbol
Environment	$\mathcal{E}$
Action set	$\mathcal{A}$
Observation space	$\mathcal{O}$
Ground state space	$\mathcal{S}$
State representation space	$\tilde{\mathcal{S}}$
Action at time t	a_t
Observation at time t	o_t
Ground state at time t	s_t
State representation at time t	$\tilde{s}_t$
Reward at time t	r_t

Table 2.1: Nomenclature adopted for reinforcement learning.

2.1.3 Markov Decision Process

The agent's exploration of the environment is usually modeled as a *Markov Decision Process* (MDP).

Definition: A Markov Decision Process *is a discrete-time stochastic control process represented by a tuple* $(\mathcal{S},\ \mathcal{A},\ P,\ R)$*, where:*

- $\mathcal{S},\ \mathcal{A}$ *have the meaning described in Section 2.1.2;*
- P *is a probability density function where* $P(s' \mid s,\ a)$ *is the probability of moving from state* s *to state* s' *performing action* a*;*
- R *is a probability density function where* $R(r \mid s,\ a)$ *is the probability of receiving reward* r *when the agent moves from* s *doing* a*.*

The task of the agent becomes to determine the optimal mapping of a given state to an action, $\pi^*(s)$ (the policy, se Equation 2.1) such that the chosen action results in maximizing the expected discounted sum of the rewards.

It is worth noticing that, in this formulation, both the reward r and the next state s' only depend on the current state s and action a, which is the main characteristic of a Markov process.

2.1.4 Policy

The policy $\pi(s)$ is a mapping $\pi : \mathcal{S} \rightarrow \mathcal{A}$ from states in $\mathcal{E}$ to actions: the agent aims to learn π to choose, at each time step t, the action that will yield the best reward given the current state, that is $\pi(s) = a$.

2.1.4.1 Value function

The most common approach is to learn the policy which generates the largest *cumulative reward*; by defining as $V^{\pi}(s_t)$ the cumulative value obtained by following the policy π starting from a state s_t, the optimal policy π^* would be

$$\pi^* = \underset{\pi}{\operatorname{argmax}} V^{\pi}(s), \ \forall s \ . \tag{2.1}$$

The function V^{π} can be defined in two ways[3]:

- if the agent has to perform just a finite number T of steps (finite horizon), then

$$V^{\pi}(s_t) = r_t + \ r_{t+1} + \ r_{t+2} + ... = \sum_{i=0}^{T} r_{t+i} \ .$$

- if the task is continuing, then there is no limit T to the number of steps (infinite horizon) and future rewards are discounted:

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{i=0}^{\infty} \gamma^i r_{t+1} \ .$$

  In this, case, $0 \leq \ \gamma \ < 1$ determines how much future rewards will be discounted: lower values of γ will give much more relevance to immediate rewards, while larger values of γ will weight more also future rewards; based on the scenario, it can be more reasonable to aim to get all the reward as soon as possible, or, instead, to aim to obtain larger incomes later in the future.

The optimal V^{π} is V^{π^*}, but its usual notation is V^* for brevity.

2.1.4.2 Evaluation function

In many real applications, it is not possible to learn any V^{π}, because it requires knowing for each $s \in \mathcal{S}$ the exact immediate reward and the transition function from one state to another, and this is often not available to the agent. For this reason, another strategy has been developed which uses an evaluation function $Q(s,\ a)$, which works on state-action couples.
The *evaluation function* $Q(s,\ a)$ is defined as the maximum discounted cumulative reward which can be achieved starting from state s and performing action a; thus, the Q-function can be expressed as the sum of the immediate reward for executing a in s and $V^*(s')$, being s' the state reached by the agent after performing a in s.
Denoting as δ the transition function among states and ρ as the reward function, for an action a in a state s, Q can be defined as follows:

$$Q(s,\ a)\ =\ \rho(s,\ a)\ +\ \gamma V^*(\delta(s,\ a))\ .$$

From previous equations, using δ and ρ as defined above, $\pi^*(s)$ can be written as

$$\pi^*(s)\ =\ \underset{a}{\operatorname{argmax}}(\rho(s,\ a) + \gamma V^*(\delta(s,\ a)))\ ,$$

whence derives

$$\pi^*(s)\ =\ \underset{a}{\operatorname{argmax}}\, Q(s,\ a)\ ,$$

which does not need either δ or ρ to be computed.

2.1.5 Model-based and model-free

Depending on how much prior knowledge of the environment the agent has, RL can be *model-based* or *model-free*[3].

- In *model-based* RL all environment's parameters are known (or can be approximated): states, actions, rewards, transition function and probability distributions. In this case, no exploration is needed and the (unique) optimal policy can be directly computed using dynamic programming techniques; the two main approaches for solving these problems are:

  - value iteration: an iterative algorithm which converges to $V^*(s)$: it refines the evaluation function at each iteration, assigns its maximum value to $V(s)$ until this one converges, and then uses it to compute π^*;
  - policy iteration: iterative algorithm which converges to π^*; it directly updates the policy, starting with an initial π_0, then computing $V(s)$ and finally updating π until it reaches the optimum.

- In *model-free* RL, the model is not known and thus the agent has to explore the environment. The exploration strategy needs to both gather information about the environment but also to deal with the exploration-exploitation tradeoff (see Section 2.1.1), meaning that must decide whether and when to give priority to the exploration of new states or exploitation of already seen states which the agent knows to give a good reward. Focusing on the actual learning task, the main algorithm for model-free problems is Q-learning, which can be adapted to both when states and actions are deterministic or not. The algorithm is described in the following paragraph.

2.1.5.1 Q-LEARNING

As proved in Section 2.1.4.2, finding the optimal Q-function corresponds to finding the optimal policy π^*.
Since

$$V^*(s) = \max_{a'} Q(s, a') ,$$

Q can be recursively rewritten as

$$Q(s, a) = \rho(s, a) + \gamma \max_{a'} Q(\delta(s, a), a') .$$

The Q-learning algorithm is based on this definition of Q: it iterates over an approximation of Q, $\hat{Q}$, which is represented by a table whose elements are the $\hat{Q}$ estimated values for each state-action pairs (s, a), that is the agent's estimate of $Q(s, a)$. At each iteration, the agent observes a state s, performs an action a,

observes the reward $r = \rho(s,\ a)$, reaches the next state $s' = \delta(s,\ a)$ and updates $\hat{Q}(s,\ a)$ following the rule

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

This algorithm works for deterministic MDPs, where state-action pairs are assumed to be visited infinitely often.

In a nondeterministic setting, functions γ and ρ have probabilistic outcomes, which can be seen as them first generating a distribution over their outcomes and then drawing one of these results according to those distributions. In other words, the same action performed in the same state can have different outcomes in terms of reward and next state.
The system is a nondeterministic MDP if the two functions depend just on the current state and action, and not on previous ones.
In this setting, all the expressions for policy, value function and evaluation function have to be rewritten taking into consideration the non-determinism:

- the value function expression becomes

$$V_\pi(s_t) = E\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right];$$

- the policy expression then becomes

$$\pi^*\ =\ \operatorname*{argmax}_\pi V^\pi(s)\ =\ \operatorname*{argmax}_\pi \left(E\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right]\right),\ \forall s\ ;$$

- the evaluation function expression turns to

$$Q(s,\ a)\ =\ E[\rho(s,\ a)]\ +\ \gamma \sum_{s'} P(s' \mid s,\ a) V^*(s')\ ,$$

  where $P(s' \mid s,\ a)$ is the probability distribution mentioned in Section 2.1.3; moreover, the recursive expression of Q becomes

$$Q(s,\ a)\ =\ E[\rho(s,\ a)]\ +\ \gamma \sum_{s'} P(s' \mid s,\ a) \max_{a'} Q(s',\ a')\ .$$

Q-learning's update rule changes too from the deterministic case, but the change is not as straightforward as the ones listed above. Because a single pair $(s,\ a)$ can correspond different outcomes, the $\hat{Q}$ value for that pair may be continuously updated to one of those values, never really converging. To face this problem, the update rule of Q-learning for the nondeterministic case becomes

$$\hat{Q}_n(s,\ a) \leftarrow (1 - \alpha_n)\hat{Q}_{n-1}(s,\ a) + \alpha_n(r + \gamma \max_{a'} \hat{Q}_{n-1}(s',\ a'))\ ,$$

where

- $\hat{Q}_n$ is the estimate at iteration n;
- the estimate is the result of the weighted average between the current $\hat{Q}_{n-1}$ value and its revised estimate $r + \gamma \max_{a'} \hat{Q}_{n-1}(s',\ a')$;
- the weight parameter α_n can be defined as

  $$\alpha_n = \frac{1}{1 + v_n(s,\ a)}\ ,$$

  where v_n is the number of visits executed on a pair $(s,\ a)$ up to the n^{th} iteration included; however, this definition of α_n is just one of the many theoretically valid ones which guarantee the convergence of the model.

The value α_n determines the graduality of the estimate updates, averaging the deterministic update rule with the un-revised entry of $\hat{Q}$. This parameter's value decreases as n grows, so later visits to the pair $(s,\ a)$ are weighted less than the early ones.

2.1.6 Temporal Difference Learning

The Q-learning algorithm is an example of *temporal difference learning* approach: it iteratively reduces the distance between Q value estimates on consecutive states. However, this class of approaches may also include algorithms that do not just work on adjacent states (as Q-learning does), but on longer chains of consecutive states.

One of these methods is Sutton's TD(λ): it blends a series of training estimates on states at various temporal distances from each other, resulting in

$$Q^{\lambda}(s_t,\ a_t)\ =\ (1-\lambda)(Q^{(1)}(s_t,\ a_t)\ +\ \lambda Q^{(2)}(s_t,\ a_t)\ +\ \lambda^2 Q^{(3)}(s_t,\ a_t)\ +\ ...)\ ,$$

which can also be recursively written as

$$Q^{\lambda}(s_t,\ a_t)\ =\ \rho(s_t,\ a_t)\ +\ \gamma((1-\lambda)\max_a \hat{Q}(s_t, a)\ +\ \lambda Q^{\lambda}(s_{t+1},\ a_{t+1}))\ .$$

In $Q^{(1)}$, the apex indicates the number of lookahead steps the estimate considers, that is the number of steps (and rewards) the rule considers when updating Q; $Q^{(1)}$ corresponds to the Q-learning update rule.

2.1.7 Partially observable states

In some applications, the agent does not have complete access to information about its state: it may have some sensors which can observe some features of the state, but not knowing it entirely.
These scenarios are similar to Markov processes: the main difference is that, once an action is performed, the arrival state is not completely known, but instead the agent can gather some information about it in the form of *observations* through its sensors; observations are stochastic functions of actions and states. This system is defined as a *Partially Observable Markov Decision Process* (POMDP). In such situations, the problem becomes inferring the state from observations (or, more precisely, a distribution over states) and then choosing actions in order to maximize the cumulative reward.[3]

A relevant problem of this approach is that observations do not comply with the Markov property, because the next observation does not depend only on current observation and action. For this reason, the agent must have a state estimator to update its belief state (which is Markov-compliant). The belief state is a probability distribution over ground states given the initial belief state and past observation-action pairs. It depends only on the previous belief state and the corresponding action, and the policy π decides the next action based on the current belief state.

Differently from fully-observable MDPs, in this case, the agent may need to perform actions only for *collecting more information* about the environment: this means that the reward is not the only criterion of action selection anymore, because information may be as valuable as a high reward in some situations.

2.1.8 Problems

Reinforcement learning models are able to represent many useful real-life applications in which an agent has to interact with the environment, observe it and navigate it to complete its task, however their implementation presents some relevant issues:

- in classic reinforcement learning approaches and algorithms, the policy is represented by a look-up table storing the estimated optimal values for each of *all* the possible state-action pairs; this implies that
  - it is not possible to generalize over unseen states, since all the information the agent has is just about discrete states and actions already seen during training;
  - all states should be stored in a finite table, which is impossible for many real-life scenarios, where the state space is extremely large and/or continuous: this would thus strongly limit the applicability of reinforcement learning approaches, because they would not be able to fully represent complex scenarios;

- as a consequence, real-life RL applications often present serious resource and performance issues, since:
  - when the state space is big, the look-up table may become extremely large, and all the operations of access, fill in and update of its entries would require a large amount of time, causing a loss of efficiency for the algorithms;
  - if the search state is large, the number of episodes necessary to completely fill in the table would significantly increase, again worsening performance or making the model unusable in such contexts.

- in large and complex environments, the information gathered by the agent may be too high-dimensional and raw, making it difficult to work with if not previously simplified and cleaned.

For all these reasons, it seems clear that tabular reinforcement learning is not suited for numerous real-life applications: an approach is needed to solve issues of both tabular value function representation and high complexity of the environment.

2.1.8.1 Dealing with reinforcement learning issues

A way to deal with the problem of efficiently working with complex environments may be to improve the state representation: raw data (in the case of reinforcement learning, raw agent's observations), for example the ones gathered by physical sensors, often include some noise and many information irrelevant to the task. On the other hand, a state representation cleaned from any kind of noise and generated only from significant features (and thus usually smaller in size than the raw observations) can be much more expressive, making the whole learning process faster and more efficient.

The issue of the Q-table's size, instead, is usually dealt with by substituting the table with a deep model which approximates the value function: this way, no more cell indicization, access, and update operation are needed, nor the table has to be stored in memory.

The problem of state representation in reinforcement learning is covered more in detail in the following Section; the issue of Q-table's exploding size is faced in Section 2.3.

2.2 State Representation Learning in Reinforcement Learning

The state in reinforcement learning is a representation of the environment where the agent is in a precise instant. This state is usually observed by the agent, and it includes all relevant information about the environment that the agent needs to know in order to make a decision, and it should not maintain information irrelevant to the task. The state can be represented as a vector, an image or a more complex data structure, such as a sequence of elements (vectors, or images, like video frames) or a graph.

State representation learning has a central role in reinforcement learning problems, as it allows the agent to keep track of its progress and to make decisions based on previous experience: it should efficiently organize information the agent gathers from sensors and observations in a way that facilitates the task. In general, the main idea is to map a complex observation to a simpler, low-dimensional vector representation of the state, which should encode all and only relevant information for target computation.
Raw agent's observations of the environment often include information irrelevant to the task: directly using them as a representation of the state would require a much larger number of samples for finding the optimal target function. On the other hand, being able to compute a compact, small vector of significant features would make the whole computation much more efficient.

2.2.1 Essential elements of a good representation

According to [9], the main features of a good state representation in a reinforcement learning problem should be:

- *compactness* for efficient estimation, meaning it should be low-dimensional compared to the observations;
- *efficient to estimate*, meaning that a relatively small number of samples should be needed to fit the SRL model;
- able to *generalize* over unseen states;
- containing *relevant state information* for the target policy;

- *markovian*, meaning that it should respect the Markov property (see Section 2.1.3), as states in the state space do.

All these features cannot be used as a guideline for learning the state representation: during training it would be hard to define the proper low dimension for efficient representation estimation (a low dimension can be forced or encouraged, but this cannot be related to the degree of efficiency of its estimation); evaluating generalization *during training* would imply to have new samples apart from those used for the training; knowing *a priori* which information is relevant for the task and which is not is usually not feasible; verifying the Markov property on representations *while* generating them may become unfeasible, because it would require to know*in advance* which features are useful for properly selecting an action. Instead, they should be verified *after* the model has been created [56].
It is worth noticing that all these characteristics do not force a unique state representation for a specific problem, thus it is possible to have many different techniques for representing the same state: in this case, the most convenient in terms of available resources or actual applicability should be chosen.

2.2.2 State Representation techniques

State representation is particularly relevant in reinforcement learning: it serves both to approximate the value function with fewer parameters and to generalize to new unseen states. Because of this importance, many different SRL techniques have been designed for reinforcement learning systems; they can be classified into the following categories:

- hand-engineered features;
- fixed representations;
- code tiling;
- basis functions;
- reward-informed representations;
- auxiliary functions and deep models.

All these classes of methods are explained in the following paragraphs.

Hand-engineered features When using this approach, for each singular task, the set of relevant features to use for state representation is "manually" designed; this is the most basic and old approach and even though this may guarantee a complete control over the state representation, it requires the work of some expert, which can be really expensive in terms of resources and time; moreover, it is not flexible to environment's changing and may not include all the necessary information for properly learning the target policy

Fixed representations A mapping function ϕ is designed a priori and then used on each set of observations to produce the state; this approach is still very simple and requires just the design of the mapping ϕ, however this still needs to be designed "by hand", does not adapt to evolving environments and may still leave behind some relevant features useful for action prediction or, more in general, policy learning.

Code tiling As described in [94], code tiling uses multiple overlapping tilings of the state space to produce a feature representation for a final linear mapping, which is then used for the actual policy learning (see Figure 2.2). This system resembles a network with fixed radial basis functions*, except that it is more computationally efficient.

It is worth noticing that the tilings need not be just simple grids. For example, to avoid all the issues deriving from high-dimensional state spaces, a proposed solution is to ignore some dimensions in some tilings by using hyperplanar slices instead of boxes. Another trick is "hashing", i.e. a consistent random collapsing of a large set of tiles into a much smaller set. By using this technique, memory requirements are often significantly reduced, with little loss of performance. This is possible because high resolution is needed in only a small fraction of the state space. Hashing solves high-dimensionality problems in the sense that memory requirements need not be exponential in the number of dimensions.

*A radial basis function (RBF) is a function the value of which depends only on the distance from the origin value; the Gaussian function is the most common example of RBF.

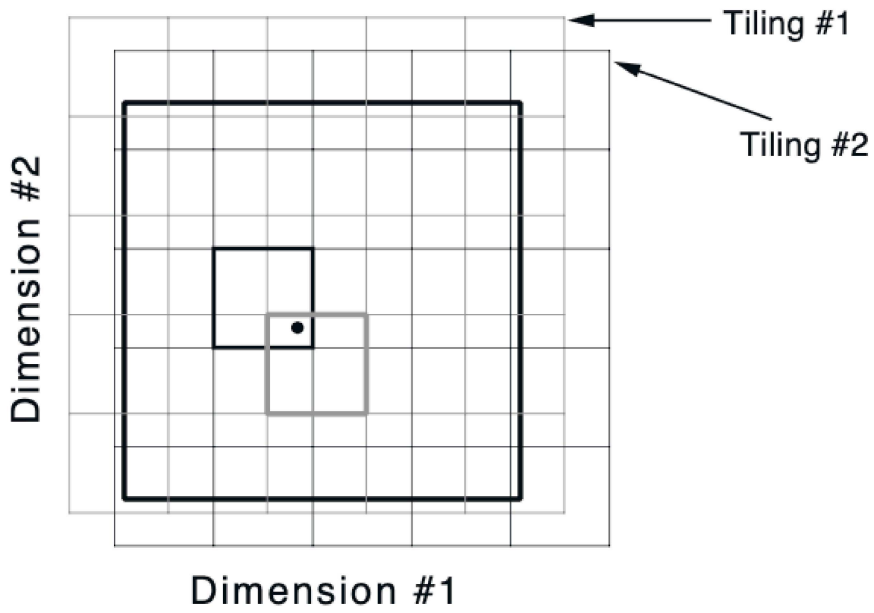

Figure 2.2: Overlapping tiles from the system in [94]: there are two $5 * 5$ regular tilings over a 2D, continuous, state space. A state is on exactly one tile of each tiling, and the state's tiles (from each tiling) are used to represent it.[Image source: [94]]

BASIS FUNCTIONS Basis function methods define a set of functions ϕ_is which are used as mappings for feature variables: this way, each function can encode a different characteristic to take into account for obtaining the state representation.
These methods can be classified into[†]:

- *polynomial basis function*: each variable itself is used as a basis function, generalized to a polynomial setting:

$$\phi_i(\boldsymbol{x}) = \prod_{j=1}^{d} x_j^{c_{i,j}} ,$$

  where $c_{i,j}$ is an integer between 0 and n, $\boldsymbol{x}$ is the feature vector of size d; the basis is called "order n polynomial". This method is quite simple, which makes it easy to implement it, but also potentially less expressive than others.

[†]Where not differently indicated, the classification and its details are taken from [47].

- *radial basis function (RBF)*: each basis function is a Gaussian

$$\phi_i(\boldsymbol{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-||c_i - x_i||^2}{2\sigma^2}},$$

where c_is are a collection of given centers, usually evenly distributed among each dimension, and σ^2 is the Gaussian's variance; these functions generalize well local changes (i.e. that do not affect the entire state space);

- *proto-value function (PVF)*: based on the agent's experience, PVFs are obtained from a diffusion model[‡] over the graph identified by the MDP of the reinforcement learning system. From the adjacency matrix A and the nodes' degrees matrix D of a graph, the Laplacian matrix is defined as $L = D - A$; the eigenvectors of L are used as an orthogonal basis for discrete state spaces. In [61], basis functions are obtained diagonalizing L, finding its "smoothest" eigenvectors that correspond to the smallest eigenvalues. Eigenvectors capture large-scale temporal properties of a transition process, thus they can be seen as similar to value functions, which reflect the accumulation of rewards over the long run; this similarity sometimes can eventually lead to a highly compact encoding. In other words, the approach in [61] constructs new basis functions which reflect the geometric properties of the environment: it does not explicitly take into account rewards, but, as said before, eigenvectors can be a trustworthy guideline for learning a performing spectral basis. This family of basis functions is called "proto value" because each eigenvector of L induces a real-valued mapping over the entire state space. The approach presented in [61] is also called "successor representation", because it is built from a graph structure, obtained by a diffusion model which explores the state space moving randomly from one state to its successors; successor representation can be traced back to [20], where the representation of a feature vector $\boldsymbol{x}$ defined as

$$\phi_j(\boldsymbol{x_i}) = I_{ij} + Q_{ij} + Q_{ij}^2 + \ldots = (I - Q)_{ij}^{-1},$$

where Q is the MDP transition matrix and I is the identity matrix.
As shown in Figure 2.3, however, a RL system adopting this method first needs to collect enough state samples to build the underlying graph. Useful representations are obtained by using the top eigenvectors[61],

[‡]Inspired by non-equilibrium thermodynamics, diffusion models aim to learn the latent structure of a set of variables by modeling how information diffuses through the latent space.

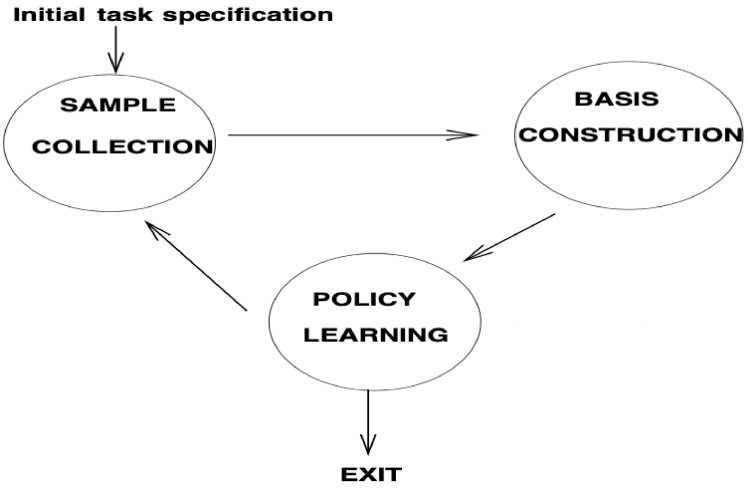

Figure 2.3: Flowchart of the system introduced in [61]: first, samples need to be collected, and for this purpose, task specification is needed; then, the basis functions are generated, which are finally used in learning the RL policy. The process can then be iterated.[Image source: adapted from [61]]

the singular eigenvectors[7] of L or its top k singular eigenvectors[92].

- *Fourier basis function*: the Fourier series is used for approximating periodic functions; in [47], a n^{th} order Fourier expansion is used, optimized by dropping the sin term of the original Fourier expansion, resulting in:

$$\phi_i(\boldsymbol{x}) \;=\; cos(\pi \boldsymbol{c}_i \;\cdot\; \boldsymbol{x}) \;.$$

  Each basis function has a vector $\boldsymbol{c}_i$, corresponding to the i^{th} basis functionϕ_i, that multiplies by an integer coefficient (in $[0,\; ...,\; n]$) each each element in $\boldsymbol{x}$; the basis set is obtained by varying these coefficients;

- *local basis function*: this class of models selects a set of basis functions based on a measure of similarity (closeness, locality) between states. For example, in [79] the similarity between a feature vector $\boldsymbol{x}$ and a location $\boldsymbol{h}$ in the state space is:

$$\mu(\boldsymbol{x},\; \boldsymbol{h}) \;=\; \begin{cases} 1 \;-\; \frac{|x_i \;-\; h_i|}{\beta_i} & if\; |x_i \;-\; h_i| \;\leq\; b_i \\ 0 & otherwise\;. \end{cases}$$

  Then, the mapping $\phi(\boldsymbol{x})$ is defined as

$$\phi(\boldsymbol{x}) \;=\; \frac{\sum_{k \in x} \mu_k w_k}{\sum_{k \in x} \mu_k}\;,$$

where $\mu_k = \mu(\boldsymbol{h}_k, \boldsymbol{x})$ and w_k is the value stored at the location of $\boldsymbol{h}_k$. The normalization in $\phi(\boldsymbol{x})$ smoothes the function and makes it a linear combination of local features.

Reward informed methods If the reward structure of the problem is known *a priori*, a representation can focus its capacity on modeling future rewards and how they diffuse through the environment. For example, [78] suggests the Krylov basis generated by the transition function associated to the MDP of the reinforcement learning setting; this is a spectral method, as the PVF of [61], from which it is derived, however it exploits the reward in defining the basis. Another difference from PVF is that [78], instead of building the adjacency matrix and then the Laplacian, directly works on the transition matrix of the MDP.

Auxiliary tasks and deep models Most current deep reinforcement learning methods augment their main objective with additional losses called auxiliary tasks, usually to facilitate and regularize the representation learning process [8]. It has been empirically seen that the design of auxiliary tasks can also help in learning better state representations: in [8], the author proposes to seek a state representation that best approximates the value function of a stationary policy in a MDP, which is measured by the error in the value approximation, and then refine their set of auxiliary tasks to adversarial value functions, which either maximize or minimize the expected return at each state; in [19], the series of discarded policies produced during reinforcement learning (which, step by step, attempt to approximate the optimum policy) is exploited to obtain efficient state representations.

In the remaining of this work the focus will be on state representation learning techniques used in deep reinforcement learning settings: in these systems, in the majority of the cases, deep models are used for state representation, but there are also some works that use embedding methods for the same purpose: both these categories are exhaustively analyzed in the next chapter. Before diving into the analysis, the next section introduces deep reinforcement learning.

2.3 Deep Reinforcement Learning

As the name suggests, *Deep Reinforcement Learning* (DRL) exploits deep learning techniques in reinforcement learning systems. To be precise, in DRL systems the Q-table (and the values memorized therein) is substituted by a deep neural network, which aims to approximate the target Q-function.
This becomes relevant when the reinforcement learning state space is very wide, for example because it is complex and articulated in many variables, implying:

- a larger Q-table, which requires more memory space and in turn makes all the operation of indicization, access and update of values slower;
- a lager number of episodes to fill in all the entries of the Q-table.

Unfortunately, this scenario is typical of many real-life applications of RL systems, as it has been further explained in Section 2.1.8.

Instead, when using deep networks there is no need to store the Q-value for each state in a table, because they are computed by the network. Moreover, deep networks can generalize over unseen states, which makes overall the system more expressive and capable of dealing with new scenarios.
However, while DRL approaches mainly build their success on the ability of the network to generate useful internal representations, they suffer from a high sample complexity, as all deep learning systems do. One way to overcome this problem is to start with an efficient input representation, using state representation learning techniques (see Section 2.2), this way significantly improving the learning performance.

2.3.1 Most used Deep Neural Networks in DRL

Deep models come in numerous different forms; in the rest of this work, the focus will be on the following types:

- *feedforward neural networks* (FNNs): the most simple type of artificial neural network, where information flows only forward and there are no cycles; for further details on FNNs see Section A.2.1;

- *convolutional neural networks* (CNNs): a kind of feedforward neural network specific for image input data, or more in general for grid data with some spatial consistency properties; for further details on CNNs see Section A.2.2;
- *recurrent neural networks* (RNNs): artificial neural networks with one or more circular connections between nodes; these networks are able to model temporal dynamics and sequentiality; for further details on RNNs see Section A.2.3;
- *graph neural networks* (GNNs): artificial neural networks specific for graph input data, meaning that they take into account relational dynamics expressed by graph structures; for further details on GNNs see Section A.2.4.

2.3.2 Main methods of Deep Reinforcement Learning

In this section, the most common DRL algorithms for policy learning are presented, as many of the works mentioned in the following sections adopt one of these approaches. First, some more generic DRL approaches are presented, and then the main algorithms are described.

2.3.2.1 Model-based or model-free

Based on the information the agent can get on the environment, deep reinforcement learning can be *model-based* or *model-free*:

- *model based*: all information about the environment is known to the agent (because it is either pre-learned or just available), thus no exploration is needed;
- *model free*: the model is not known in advance, so the agent has to gather information through exploration.

Further details about model-based and model-free, for more general reinforcement learning settings, have been given in Section 2.1.5.

2.3.2.2 Off-policy

Based on the policy followed for choosing actions, DRL algorithms can also be distinguished between *off-policy* and *on-policy*:

- *off-policy*: in this setting, the agent can exploit data and information obtained by any of its (past) experience, also older, less trained policies. More precisely, off-policy algorithms distinguish between the current policy, which is the current approximation of the optimal policy, and the behavioural policy, which corresponds to the actual actions of the agent in the environment, that are collected into a replay buffer. in the form of tuples $(s,\ a,\ r,\ s')$, where the symbols have the usual meaning. Thus, the current policy, which is updated step-by-step, and the behavioural policy, which is the one actually followed in choosing actions, are distinguished. This way, the replay buffer becomes a sort of model of the environmental dynamics and can be used to update the current estimate of the optimal policy [52]. Q-learning is an off-policy algorithm because it updates its Q-values using the Q-value of the next state s' and the *greedy* action a', even though the current policy is not greedy in the choice of actions.
- *on-policy*: these algorithms work with a single policy, which is the one followed by the agent in choosing actions *and* the one updated afterward; thus, the agent does always follow the current best approximation of the optimal policy. Updating the (unique) policy deprecates the old transition samples, making it necessary to collect new samples at each update: on-policy methods are therefore less sample-efficient than off-policy ones [52].

2.3.2.3 Actor-critic methods

Actor-critic methods are temporal difference methods (see Section 2.1.6 for further details about temporal difference learning) that have a separate network to represent the policy independently from the value function. The policy structure is called *actor*, because it is used to select actions, and the estimated value function is the *critic*, because it criticizes the actions made by the actor.
In actor-critic algorithms, learning is always on-policy: the critic must learn

and critique the policy that is currently being followed by the actor. The critique is communicated to the actor in the form of TD error: this value is the sole output of the critic and drives all learning in both actor and critic [55]. Figure 2.4 represents a simple actor-critic system.

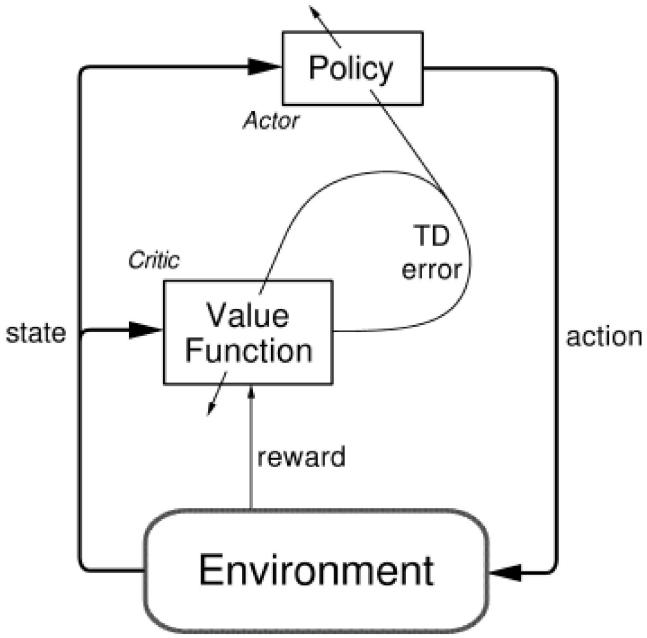

Figure 2.4: A scheme of an actor-critic RL system: after each action selection the critic evaluates the next state to determine based on the TD error obtained. The error is used to evaluate the last executed action, and to encourage or discourage its choice in the future.[Image source: [55]]

2.3.2.4 Policy gradient methods

Policy gradient methods rely on optimizing parametrized policies with respect to the expected return (long-term cumulative reward) by gradient descent. Modern policy gradients use an Advantage estimator A that evaluates how good an action is compared to the average action in that state [52]: the Advantage is the difference between the expected future reward and the value estimate: $A(s,\ a)\ =\ Q(s,\ a)\ -\ V(s)$.

2.3.2.5 Most common methods

In this section, the main DRL algorithms exploited by the works analyzed in this work are introduced.

Deep Q-Learning Deep Q-learning (DQN) [66] [67] is a model-free, off-policy, deep learning model which learns control policy directly from high-dimensional input (images). The deep model is a convolutional neural network taking an image as input and generating the value function for future rewards as output. It has been introduced first in [66] and then re-proposed in [67], with an updated version of the convolutional network; the latter version is the one widely adopted in DRL problems.
The training is performed on a variant of Q-learning with stochastic gradient descent. One of the main issues that this method faces is DRL algorithms instability, mainly due to high sample correlation and also to the correlation between action values and target values. These problems are addressed by introducing: a replay buffer, where past experience is stored in the form of tuples, which are decorrelated and then randomly sampled; an iterative update, that adjusts the action values towards target values that are only periodically updated, thus reducing correlations with the target.

(Asynchronous) Advantage Actor-Critic Asynchronous Advantage Actor-Critic (A3C) [68] is an on-policy, model-free, actor-critic, policy gradient method. It faces the issue of sample correlation in a different way than off-policy methods do: instead of storing and decorrelating experience samples into a replay buffer, the authors propose to asynchronously run in parallel many agents on the same number of instances of the same environment. This solves the memory issues of replay buffers (which use more memory and computational power per interaction) and removes the constraint of using an off-policy algorithm, obtaining the same decorrelation result for on-policy methods, because parallel explorations produce uncorrelated samples. Moreover, the implementation of this algorithm requires much less computational power than previous DRL methods.
In [68] the asynchronous version of different DRL methods are proposed

(among which there is DQN), but the best one is Asynchronous Advantage Actor-Critic, which is there tried also in a soft form, with the addition of a policy entropy term to the objective function.
Advantage Actor-Critic (A2C) is the synchronous version of A3C: instead of running the multiple agents in parallel, the agents run sequentially, so one after the other, and the policy update is obtained averaging over all the agents' results.

Proximal Policy Optimization Proximal Policy Optimization algorithm (PPO)[87] is a model-free, on-policy, actor-critic algorithm, is based on TRPO (Trust Region Policy Optimization [86]) but simpler and faster. Like TRPO, PPO is a policy gradient algorithm: it optimizes a policy improvement objective, meaning that the objective is to maximize the expected value of the discounted Advantage of the old policy under the trajectory distribution of the new policy. This problem is intractable: TRPO faces this issue using a surrogate objective with importance sampling, proving that the expectation computation can be avoided if the updated policy remains similar to the current policy; PPO takes the importance sampling surrogate objective from TRPO but then just clips it, this way avoiding the policy update to diverge. The algorithm uses many epochs of stochastic gradient ascent to perform each policy update.

Soft Actor-Critic Soft Actor-Critic (SAC) [35] is a model-free, off-policy, stochastic actor-critic algorithm in which the actor aims to maximize expected reward while also maximizing entropy, that is, executing the desired task while acting as randomly as possible. The algorithm adds an entropy-maximization term to the original RL objective function, which encourages maximum randomness while seeking the optimum objective.; by doing this, the method keeps exploring the environment up to the end of training, while usually RL algorithms stop exploring sooner, converging to sub-optimal policies. The loss thus balances between maximum reward and maximum entropy, with a hyperparameter (called *temperature*) regulating the importance of the entropy term [52]. This algorithm provides sample efficiency and stability also

with complex, high-dimensional tasks: together with DQN, this is probably one of the most widely used algorithms in DRL problems.

Deep Deterministic Policy Gradient The Deep Deterministic Policy Gradient algorithm (DDPG) [57] combines Q-learning, deterministic policy gradient algorithms [89] and deep Q-networks (DQN) [67] and creates a model-free, off-policy, actor-critic RL algorithm that applies deep neural networks' universal approximation to control problems with high-dimensional, continuous action spaces, starting from low-dimensional observations, often learning end-to-end policies directly from raw observations.
The algorithm uses a fixed size, last-in-first-out, replay buffer, to avoid sample correlation, which is typical of consecutive sequences of samples resulting from exploration; being DDPG an off-policy algorithm, the replay buffer can be large, allowing the algorithm to benefit from learning across a wide set of uncorrelated transitions.
Being actor-critic, the model has two sets of parameters: the actor deterministically maps the state to an action; the critic evaluates the state-action pair. The target network "interpolates" between the actor and the critic, reducing this way the model's instability. The target network is similar to the one of DQN, but modified for actor-critic and using "soft" target updates, rather than directly copying the weights. Target values are constrained to change slowly, greatly improving the stability of learning. A copy of actor's and critic's weights are created and used for calculating target values. This modification moves the unstable problem of learning the action-value function closer to supervised learning, a problem for which robust solutions exist. This slows the learning process, but the advantage in terms of stability outweighs that drawback.

3

State representation in Deep Reinforcement Learning

In this chapter the focus will be on state-of-the-art techniques of state representation in deep reinforcement learning. First, the criterium of classification is explained, then the methods are presented according to the criterium. The approaches are also compared and analyzed in terms of performance and application fields, when possible.

3.1 Criterion of presentation

In this work, the different state representation techniques are organized by the type of input data they are built for:

- "flat" data: they can be *vectorial features*, like euclidean coordinates in a grid-like environment, and *images*, for example pictures collected by sensors;
- *sequences*, like in the case of frames of a video, or incremental states resulting from the processing of many features (e.g., profiling a user based on his likes and dislikes);

- structured data, particularly *graphs*, like networks or robotic skeletons.

Almost all models presented in this section use deep learning architectures for computing state representations:

- convolutional neural networks, for images;
- recurrent neural networks, for sequences;
- graph neural networks, for graphs.

In some cases, deep neural networks are used to compute state embeddings, i.e. mappings from the large and complex state space to a smaller one, encoding just a defined set of state space's features: this is the case of bisimulation-based state representation methods (see Section 3.2.3.3) or graph-based representations (see Section 3.2.1.3). These embeddings are then used as state representations in models.

3.2 State representation techniques

The next sections are organized as follows: first, methods for vectorial features and images ("flat" state) are introduced; then, the approaches of state representation for sequential data are described, and afterward those based on graph state are presented.

3.2.1 "Flat" representation

In this section approaches made for vectorial or image input state are presented: those that use raw features are first introduced, followed by some techniques which exploit embeddings; lastly, methods for image state are described.

3.2.1.1 Background

This kind of "flat" input, with no internal structure (in the case of vectorial features) or in a grid-like form (for images), is usually processed with two

kinds of deep models: feedforward neural networks, for the former, and convolutional neural networks, for the latter.

Feedforward Neural Networks A feedforward neural network (FNN) is the simplest kind of neural network. In FNNs, information between nodes flows in a single direction (forward), usually from an input layer of nodes to the output one, with no loops. This kind of network may have one or more hidden layers, meaning that they are made of nodes that are neither input nor output ones; as for all other neural networks, a FNN with many hidden layers is called deep. For further details on FNNs, see Appendix A.2.1.

Convolutional Neural Networks The convolutional neural network (CNN) is the most used model for fitting image input datasets. A CNN can capture spatial dependencies in an image by applying a convolution operation with specific filters to input images. This kind of architecture performs efficiently on images mainly because it shares the filters' parameters among the whole image, thus drastically reducing the number of parameters to store in memory, which could not be possible if the image would be considered a long rectified vector of pixels processed by a FNN. For further details on CNNs and the convolution operation, see Appendix A.2.2.

3.2.1.2 Raw features state representation

The most simple way of representing the state in a deep reinforcement learning setting is by using a feature vector input to a feedforward network, without any significant preprocessing of the agent's observation: this is the case of [57] [25] [34] [11] [105] [22] [117].
What usually happens in these contexts of application is that the state can be directly represented as a vector, either because it is, for example, a pair of coordinates, like in [34] [11] or, more in general, low-dimensional observations, like also joint angles [57] [25] or application-specific topological features[105] [22] [117].

This type of representation is usually chosen every time the context of the

application allows for it, because it requires little computational effort to be generated. In the case of [57] [25], it is only necessary to extract the coordinates of the state or some geometrical features of it (angles), and concatenate these features; a similar situation is in [34], where the state vector for a robotic application is the concatenation of geometrical features and their time derivatives; in [11], only a pair of coordinates is used as state vector, which is then fed to a module for computing a bisimilarity measure between states. In the case in [105], a traffic application of DRL, a multilayer relational grid represents the relation of a vehicle with others: each layer corresponds to a feature, thus each column corresponds to the features of a specific vehicle. In this case, the entire grid is fed to the network, but, because all the features are already available in the grid, a multilayer fully-connected FNN is enough and no convolutional layer is needed. A similar scenario is in [22], where each $2D$ layer of the grid represents a feature of the vehicle and the pedestrians around it; the grid state of [22] is represented in Figure 3.1.

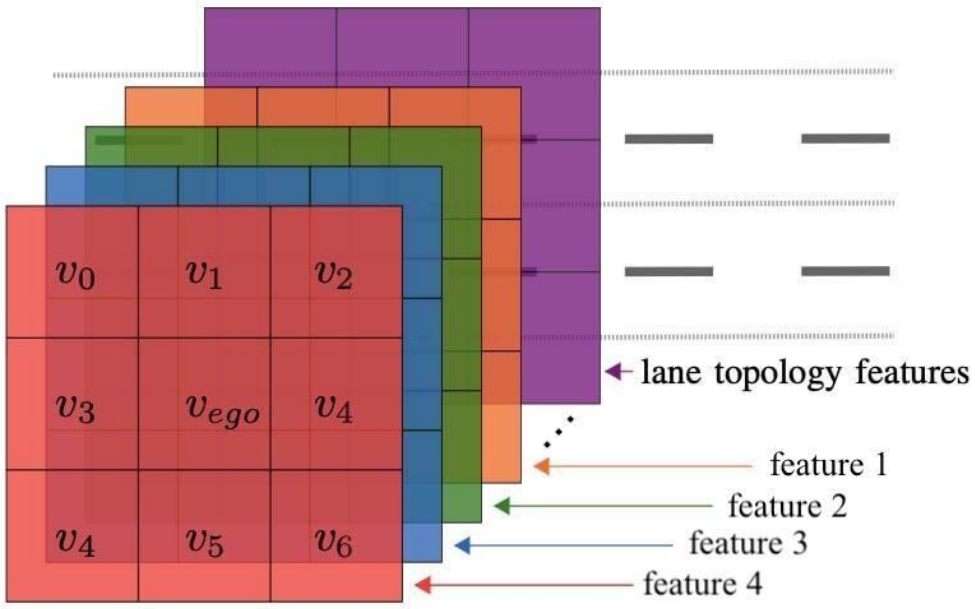

Figure 3.1: The $3D$ feature tensor of [22]: each layer of the tensor corresponds to a feature.; v_is are the vehicles' feature vectors.[Image source: adapted from [22]]

Table 3.1 summarizes the kind of vectorial features used by the approaches described in this section, and the connectivity of the neural network layers

used to process them.

Paper	Coord.	Other topologic. features	Domain-specific features	Fully connected FNN layer(s)
[11]	X			X
[22]			X	X
[25]	X	X		
[34]		X	X	
[57]	X	X		
[105]			X	X
[117]			X	

Table 3.1: Main characteristics of feature vector state representation approaches: euclidean coordinates, other topological features (angles, velocities, etc...), domain-specific features; the last column indicates if the FNN is fully connected or not, a feature which impacts the complexity of the network and the number of its parameters..

Analysis The feature vector state representation is clearly limited in its applicability and expressive power: not all applications allow for such a representation, especially the most complex one; for example, in scenarios where the agent must take into account (or interact with) many elements of different kinds (for example, traffic or transportation problems), all these information cannot be concatenated in a single vector, because it would become too long, and it would be complex to deal with them separately, because they would likely have different sizes. A solution like a multilayer feature tensor (adopted in [105] [22]) can be used, but the relational grid may dangerously grow in size if the features number increases or if the environment becomes too complex. Moreover, the concatenation of features, both topological or context-specific, may not capture some structural information of the state space (for example, similarity measures): this lack of expressivity could make the representation less effective, losing some information about the state that would be relevant for the task. Finally, it happens that the FNN taking as

input these vectors is fully connected [11] [105] [22], a configuration which may be needed for producing expressive hidden representations: this means that the number of parameters of the model can become really large.

In conclusion, this state representation method can be beneficial in applications where the main traits of the state space can be effectively represented by a small number of features, or when the advantage of not having to perform representation learning on the input brings better performance in terms of execution time. A different scenario may be having some preprocessed features of the state space, possibly obtained from other tasks performed on the same state space.

3.2.1.3 Bisimulation-based embeddings

In this section some embedding methods based on bisimulation measures are introduced and compared by analyzing their target functions.

Bisimulation among states Bisimulation is a form of recursive "behavioural equivalence", meaning that it measures how similar two elements are based on their evolutional behaviour. In Markov decision processes (MDPs), this corresponds to a form of equivariance, for which two states are considered equivalent if they share equivalent distributions over the next equivalent states and they yield the same immediate reward[51]; as a consequence, states belonging to the same bisimulation class belong to the same abstract group of "similar" states. This definition has been modified and updated by different works: in [12] two new definitions have been proposed, considering only actions with positive outcomes, and then other authors tried to improve them by adding some stricter assumptions [29] [113].

In [112], the authors try a different approach: they try to reduce the method's complexity by designing a more flexible and easily applicable distance measurement by avoiding some expensive operations (i.e., expectations). They design a stochastic-approximation-based method that can learn a mapping function (encoder) from image observations to latent representation space.

Their bisimulation measure $B(\phi(s),\ \phi(t)$, where $\phi(s)$ is the embedding, measures the cosine distance among the agent's observations to compute their behavioural difference: the mapped latent space becomes a unit sphere, where the difference between representations depends on the cosine distance of the encoded observations. $B(\phi_\psi(s),\ \phi_\psi(t)$ is defined as:

$$B(\phi(s),\ \phi(t)) \ = \ 1 \ - \ cos(\phi(s),\ \phi(t)) \ = \ 1 \ - \ \frac{\phi(s)^T \ \cdot \ \phi(t)}{||\phi(s)|| \ \cdot \ ||\phi(t)||} \ .$$

The loss for the state encoder is:

$$\mathcal{L}_\phi(\psi) \ = \ \mathbb{E}_{(s,\ r(s,\ a),\ a,\ s'),\ (t,\ r(t,\ a),\ a,\ t') \ \sim \ \mathcal{R}} (B(\phi_\psi(s),\ \phi_\psi(t)) \ - \ \hat{B})^2 \ ,$$

where

- ϕ is the mapping from the image observation to the latent space, which is parametrized by ψ;
- $s,\ t,\ s',\ t'$ are states;
- $\mathcal{R}$ is the replay buffer, where past experience is stored in tuples of the form $(s,\ r(s,\ a),\ a,\ s')$;
- B is the bisimulation metric in the latent space and $\hat{B}$ is the bisimulation approximate measure, based on reward values.

For further technical details, see [112].
This approach avoids computing expectations by following two approaches applying sample-based computation. This requires sampling the next state representations., which can be done either by sampling the next observations and encoding them, or encoding the current observations into the latent space and learning the latent transition model explicitly to compute the next state representations; the authors implement both this options in [112].

The work in [113] explicitly aims to find a representation method that only considers state information that is relevant to the task, while also being invariant to irrelevant information. Based on the work of [44], they use the reward signal to help determine task relevance: the intuition behind this choice

is that, being cumulative rewards the objective, state elements are relevant if they influence both the current reward and future state elements that will influence future rewards. Such recursive relationship can be expressed as a recursive concept of state abstraction: the representation must be predictive of rewards, but also predictive of itself in the future; the recursive form of this representation corresponds to the one of bisimulation.
The approach proposed in [113] is a gradient-based method for directly learning a representation space with the properties of bisimulation metrics; their system is represented in Figure 3.2.

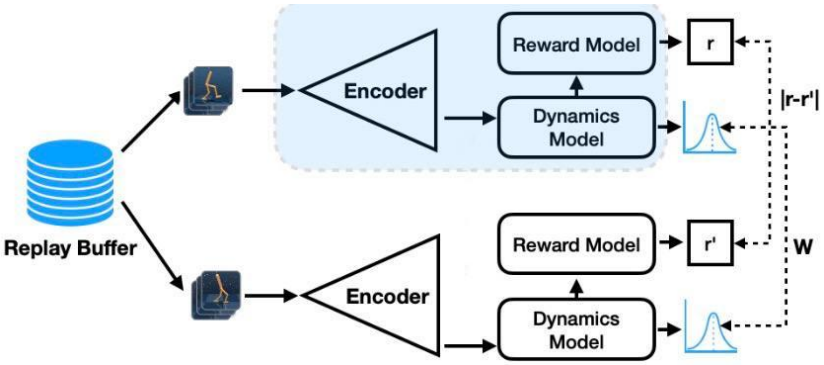

Figure 3.2: The deep bisimulation for control model, introduced by [113]: the model simultaneously trains the encoder, and the RL policy; W is a distance measure between state distributions (for further technical details, see [113]).[Image source: adapted from [113]]

The model aims to learn representations under which distances correspond to bisimulation metrics[23], and use these representations for improving RL. Thus, the goal is to learn encodings of states that are suitable for their specific task, while discarding any irrelevant information. They base their definition of bisimulation on that of [32], which defines as bisimulation metric the equivalence between states according to which, $\forall\, s,\ t \in \mathcal{S}$,

$$R(s,\ a)\ =\ R(t,\ a)\, \forall\, a\ \in \mathcal{A}\ ,$$

$$P(\mathcal{C}\mid s,\ a)\ =\ P(\mathcal{C}\mid t,\ a)\, \forall\, a\ \in \mathcal{A},\ \forall\, \mathcal{C}\ class\ of\ equivalence\ ,$$

where $P(\mathcal{C}\mid s,\ a)\ =\ \sum_{s'\ \in\ \mathcal{C}} P(s'\mid t,\ a)$. This definition implies that two states are bisimilar if executing the same action from each of them brings the same

instant reward, and if the state distributions after executing that action on both states are equal. The loss for the state encoder in [113] is:

$$\mathcal{L}(\phi) = \mathbb{E}_{(s,\, r(s,\, a),\, a,\, s'),\, (t,\, r(t,\, b),\, b,\, t') \sim \mathcal{R}'}(||\phi(s) - \phi(t)||_1 - |r(s,\, a) - r(t,\, b)| - \gamma W_2)^2\,,$$

where

- $\mathcal{R}'$ is a batch of samples from the replay buffer;
- ϕ is the mapping from the image observation to the latent space;
- $s,\ t$ are states;
- $a,\ b$ are actions;
- W_2 is a metric for distance among state distributions;
- γ is a discount factor.

For further technical details, see [113].

It is worth mentioning, however, that the approach introduced by [113] is based on a quite strong assumption: each observation gathered by the agent uniquely determines its originating state. This assumption does not apply to all those domains where two states can generate the same observation for the agent, which is not such a rare scenario.

Analysis As it can be seen, the losses of the two methods described above are very similar: both minimize the difference between a bisimulation metric and a measure of the specific feature they want to prioritize in their embedding, the reward in this case.
Both methods presented in this Section aim to build a state representation only based on features relevant to the task, so that no computational effort is put into learning features that do not significantly contribute to learning the target policy, which usually cannot be avoided when using representations obtained as byproducts of deep learning models. Both are first conceived for image input, but their definition makes them quite versatile to other kinds of input observations.

3.2.1.4 Convolutional neural networks for state representation

The most common way to represent the state of the environment in a DRL problem is with images (or video frames), which are then processed with convolutional neural networks (see Section A.2.2).
Probably, among the causes of such a wide adoption of convolutional networks in DRL problems is their success in computer vision, as well as the reduced number of parameters typically used due to the sharing of parameters. Moreover, also when the state is not an image, but a grid, or a matrix, convolutional networks are the type of neural network which is most likely to produce a compact and efficient representation of this data type. Moreover, cameras may be a convenient and inexpensive way to acquire state information, especially in complex and unstructured environments.
For these reasons, some of the most popular deep reinforcement learning algorithms have been first implemented for image input, like DDPG,[57] DQN, [66] [67] A2C and A3C, [68] and TRPO[86] (see Section 2.3.2 for details about this algorithms).

Most popular convolutional networks In years, many different CNN models have been proposed in DRL systems, for various tasks and scenarios: robotic applications, traffic navigation and prediction, vehicle motion, data compression, and more.
Based on the literature studied for this work, there are two principal CNNs, which many other authors refer to or draw from for building their convolutional network in DRL:

- *DQN convolutional network*[66] [67]: DQN (Deep Q-Learning) is a deep learning model which learns a control policy directly from high-dimensional input (see Section 2.3.2). The CNN takes as input the agent's state observation (in the form of an image) and produces the value function for reward prediction. This model does not have a separate module for a state representation task: the input image is processed by the CNN, which is also used for approximating the Q-function. Differently from models that take in input the state observation with the action, this approach takes only the observation and outputs a value for each possible

action, allowing to compute the Q-value associated with every action with a single forward pass through the network.The first version of this architecture [66] was made of:

1. the input layer, taking as input pictures of shape $84 * 84 * 4$; they are usually the result of some preprocessing on raw images, which are larger in size and thus more computationally demanding; here this preprocessing is applied to the last 4 frames of the history, which are then stacked to produce the input to the Q-function;
2. a hidden convolutional layer with 16 filters of size $8 * 8$, stride 4 and a rectified non-linearity[71];
3. a hidden convolutional layer with 32 filters of size $4 * 4$, stride 2 and a rectified non-linearity;
4. a final, fully connected hidden layer of 256 rectified non-linear units;
5. the fully connected output layer, with a reward value for each possible action.

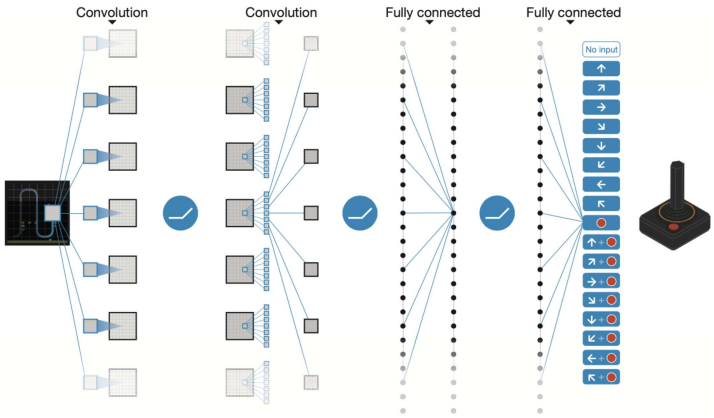

Figure 3.3: The second version of the DQN network.[Image source: [67]]

The second version of the architecture[67] (showed in Figure 3.3), introduced two years later, changed the network as follows:

1. the input layer remained the same;
2. the first hidden convolutional layer had 32 filters of size $8 * 8$, with same stride and activation function as before;
3. the second hidden layer had 64 filters of size $4 * 4$, with same stride and activation function as before;

4. a third hidden convolutional layer was added, with 64 filters of size $3 * 3$ and stride 1, always with the same nonlinear rectifier;
5. the final hidden layer has 512 rectifier units;
6. the output layer remained the same.

It can be seen that the second version of the architecture was more powerful and expressive than the first one, as the increased number of filters, convolutional layers and units in the last hidden layer proves; later works which adopted this architecture almost always used its second version.

- *SAC+AE convolutional network*[108]: SAC+AE (Soft Actor-Critic + Autoencoder) is a model-free, off-policy* RL algorithm for pixel observations with a reconstruction loss as auxiliary task, meaning that the authors designed a specific reconstruction loss for state representation learning, separated from that of actual RL. It is a model-free approach that trains state representation jointly with the DRL policy: it takes SAC model [35] (see Section 2.3.2) and augments it with a regularized autoencoder (see Section A.2.5). The complete network scheme is represented in Figure 3.4.

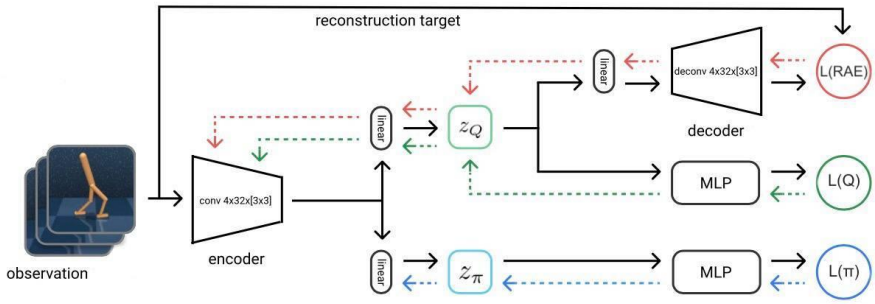

Figure 3.4: The SAC+AE model: full arrows correspond to the forward flow, while dotted arrows are the different gradients. $\mathcal{L}(RAE)$ is the loss of the regularized autoencoder;$\mathcal{L}(Q)$ and $\mathcal{L}(\pi)$ are two losses associated with the SAC approach to RL (see [35]. for further details). [Image source: [108]]

The convolutional encoder of this model extends the one used in [95], which in turn took the D4PG approach from [6] and extended it for raw-pixel input. They use the regularized AE loss function from [31], which imposes a L_2 penalty on the learned representation z_t and weight-decay on the decoder parameters θ. The loss is as follows:

$$\mathcal{L}(RAE) \;=\; \mathbb{E}_{o_t \sim \mathcal{R}}[logp_\theta(o_t \mid z_t) \;+\; \lambda_z||\, z_t||^2 \;+\; \lambda_\theta||\, \theta||^2]\,,$$

*see Section 2.3.2 for the definitions od model-free and off-policy methods

where:

- o_t is the agent's observation at time t;
- $\mathcal{R}$ is the replay buffer;
- p is the sampling distribution of states;
- z_t is the representation learned by the regularized AE (RAE) of the observation at time t;
- λs are hyperparameters.

This loss aims to obtain an optimized representation while imposing a bias both on the obtained representation and on the decoder parameters θ (with the L_2 penalty).

The SAC+AE convolutional network is thus built as follows:

1. the input layer, taking as input images of shape $84 * 84 * 3$;
2. the first hidden convolutional layer has 32 filters of size $3 * 3$ with stride 2 and $ReLU$ activation[71] (in [95], the activation was ELU [17]);
3. the other three hidden convolutional layers have 32 filters of size $3*3$ with stride 1 and $ReLU$ activation function (in [95] there where only two convolutional layers instead of four);
4. the output layer is a single fully-connected layer with 50 units, with layer normalization[5] and $tanh()$ activation.

Analogously, the decoder has a perfectly symmetric structure, with a fully-connected input layer, four deconvolutional layers with 32 filters of size $3 * 3$ and stride 1 for the first three layers, 2 for the last one.

As can be seen, the two approaches described above are quite different in their objective: the first one, DQN, is in itself the actual deep network for the DRL system, while the second one, SAC+AE, is just a part of a more complex system, and it just deals with the state representation for the other components of the system. Thus, in both cases the convolutional networks take image input and generate a representation of it for the RL problem, but they do not have the exact same role in their respective systems.
In Table 3.2 some approaches are listed that use one among DQN and SAC+AE convolutional network, and whether they made some modification to the original model or not.

Paper	DQN	SAC+AE	Modified
[13]		X	
[18]	X		
[21]	X		X
[26]	X		X
[38]	X		
[48]		X	
[54]		X	X
[68]	X		X
[73]	X		
[96]	X		
[101]	X		X
[109]		X	
[112]		X	
[114]		X	

Table 3.2: Some of the most recent approaches using DQN or SAC+AE convolutional networks.

Convolutional networks in attention methods *Attention mechanisms* in machine learning try to mimic what cognitive attention in humans does: when looking at a picture, a person does focus on some features of the image more than on others, because this makes the subject recognition cognitive task faster: these features usually are color intensity, shapes, edges and their inclination, movement [40]. This idea was taken by computer vision: by looking at different parts of an input image it could be possible to learn to accumulate information about a feature and accordingly classify the image [50]; this was then extended to other fields of machine learning, like learning from sequences. Attention can be thus considered as a particular type of memory[1], where the focus of the model is concentrated on specific features/parts of the input, depending on the task.

An interesting insight on attention is its difference with respect to convolutions [50]: the colors of the attention connections in Figure 3.5 indicate

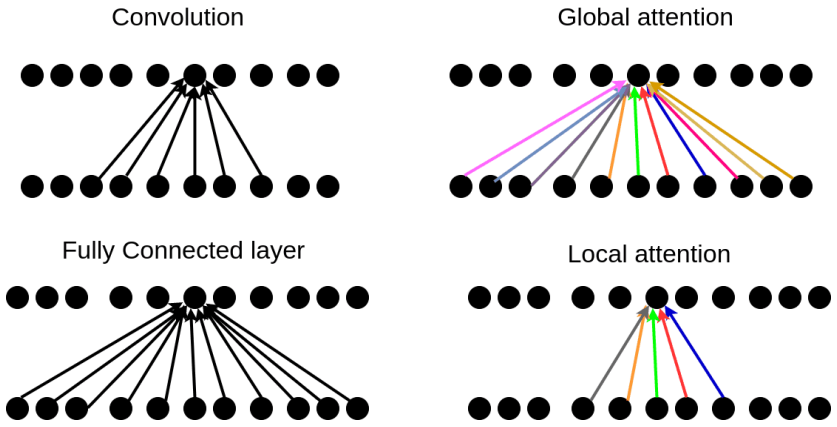

Figure 3.5: Comparison between a fully-connected layer, a convolutional layer, a local attention layer and a global attention layer. [Image source: [1]]

that the weights associated to those links change constantly, because their value depends on the context, while in convolutional or fully connected layers, weights are just slowly updated with gradient descent and then are fixed.

However, here the focus will not be on a comparison between convolution and attention (which nevertheless helps in understanding the idea behind attention on images), but rather on presenting DRL approaches based on attention but first processing their input with a convolutional network.
An attention function can be defined as a mapping of a query and a set of key-value pairs to an output, where query (which usually are more than one, packed in a matrix Q), keys (K), values (V) and output are all vectors. The output is usually the weighted sum of the values, where the weight assigned to each value is given by a compatibility function of the query with the corresponding key [97].
Attention modules usually use a $softmax$ function over the matrices Q, K, V for computing the output, taking the following form:

$$Attention(Q,\ K,\ V)\ =\ softmax\left(\frac{QK^T}{\sqrt{d_K}}\right)V\ ,$$

where QK^T is the dot-product between Q and K and d_K is the size of the keys in K.
Multi-head attention performs in parallel many attention modules with Qs,

Ks, Vs of different sizes; the outputs of each of these attentions are then concatenated.

In [88][111][14], the image input is first processed by a convolutional neural network, whose output is then given to the attention module of the system, which is usually a multi-head attention module [88][111] (see Figure 3.6).

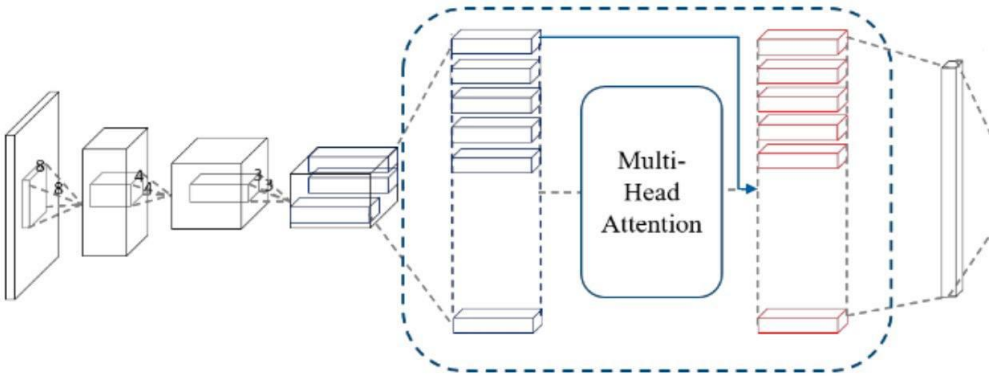

Figure 3.6: The architecture of the model in [88]; this structure is analogous to the ones in [111][14]: the input image is first processed by a convolutional network, and then fed to a (possibly articulated) attention module, before passing through the rest of the network, which can take different forms (in [14], for example, the convolutional encoder and attention module are parts of a RNN). [Image source: adapted from [88]]

The main reason for using a convolutional network in these systems is that attention modules cannot work directly on raw images: they need a compact representation of their input, in order to be able to access the feature vectors they work onto.

In Table 3.3 all the approaches analyzed in this work are listed which use a CNN for processing their input observation; some structural information of the models is reported, when available. A dash (-) in the table means either that the information required is missing, or that the field does not apply to that specific entry (e.g.: in the FILTER SIZE column, when the layer is not convolutional there cannot be any filter associated to it).

Paper	Input size	Hidden layers	Num. channels / num. units	Filter size	Nonlinear activation for hidden layers
[13]	$84 * 84$	Conv	32	$3 * 3$	*ReLU*
		Conv	32	$3 * 3$	*ReLU*
		Conv	32	$3 * 3$	*ReLU*
		Conv	32	$3 * 3$	*ReLU*
[14]	224*224	Conv	96	$11 * 11$	-
		Conv	256	$5 * 5$	
		Conv	384	$3 * 3$	
		Conv	384	$3 * 3$	
		Conv	256	$3 * 3$	
[18]	$84 * 84$	Conv	32	$8 * 8$	*ReLU*
		Conv	64	$4 * 4$	*ReLU*
		Conv	64	$3 * 3$	*ReLU*
		FC	512	-	*ReLU*
[21]	$64 * 64$	Conv	32	$8 * 8$	*ReLU*
		Conv	64	$4 * 4$	*ReLU*
		Conv	64	$3 * 3$	*ReLU*
[26]	$70 * 70$	FC Conv	64	$3 * 3$	*ReLU*
		FC Conv	64	$3 * 3$	*ReLU*
		FC Conv	64	$3 * 3$	*ReLU*
		FC Conv	64	$3 * 3$	*ReLU*
[38]	$84 * 84$	Conv	32	$8 * 8$	*ReLU*
		Conv	64	$4 * 4$	*ReLU*
		Conv	64	$3 * 3$	*ReLU*
		FC	512	-	*ReLU*
[48]	$84 * 84$	Conv	32	$3 * 3$	*ReLU*
		Conv	32	$3 * 3$	*ReLU*
		Conv	32	$3 * 3$	*ReLU*
		Conv	32	$3 * 3$	*ReLU*

Paper	Input size	Hidden layers	Num. channels / num. units	Filter size	Nonlinear activation for hidden layers
[53]	$32 * 32$	Conv Conv FC Conv FC	6 16 120 84	$28 * 28$ $10 * 10$ $1 * 1$ -	*sigmoid* - - -
[54]	$84 * 84$	Conv Conv Conv Conv FC	32 32 32 32 50	$3 * 3$ $3 * 3$ $3 * 3$ $3 * 3$ -	TLU[†] TLU TLU -
[57]	$64 * 64$	Conv Conv Conv FC	32 32 32 200	-	-
[66]	$84 * 84$	Conv Conv FC	16 32 256	$8 * 8$ $4 * 4$ -	$ReLU$ $ReLU$ $ReLU$
[67]	$84 * 84$	Conv Conv Conv FC	32 64 64 512	$8 * 8$ $4 * 4$ $3 * 3$ -	$ReLU$ $ReLU$ $ReLU$ $ReLU$
[68]	$84 * 84$	Conv Conv FC	16 32 64 256	$8 * 8$ $4 * 4$ -	$ReLU$ $ReLU$ $ReLU$
[73]	$84 * 84$	Conv Conv Conv FC	32 64 64 512	$8 * 8$ $4 * 4$ $3 * 3$ -	$ReLU$ $ReLU$ $ReLU$ $ReLU$

[†]Thresholded Linear Unit activation funtion.

Paper	Input size	Hidden layers	Num. channels / num. units	Filter size	Nonlinear activation for hidden layers
[85]	$32 * 32$	Conv Conv FC FC	32 64 200 10	$3 * 3$ $2 * 2$ - -	*ReLU* *ReLU* *ReLU*
[86]	$84 * 84$	Conv Conv FC	16 16 20	$4 * 4$ $4 * 4$ -	-
[88]	-	Conv Conv Conv	32 64 64	$8 * 8$ $4 * 4$ $3 * 3$	*ReLU* *ReLU* *ReLU*
[95]	$84 * 84$	Conv Conv FC	32 32 50	$3 * 3$ $3 * 3$ -	*ELU* *ELU* *tanh*
[96]	$84 * 84$	Conv Conv Conv FC	32 64 64 512	$8 * 8$ $4 * 4$ $3 * 3$ -	*ReLU* *ReLU* *ReLU* *ReLU*
[101]	$84 * 84$	Conv Conv Conv FC FC	32 64 64 512 512	$8 * 8$ $4 * 4$ $3 * 3$ - -	*ReLU* *ReLU* *ReLU* *ReLU* *ReLU*
[108]	$84 * 84$	Conv Conv Conv Conv	32 32 32 32	$3 * 3$ $3 * 3$ $3 * 3$ $3 * 3$	*ReLU* *ReLU* *ReLU* *ReLU*

Paper	Input size	Hidden layers	Num. channels / num. units	Filter size	Nonlinear activation for hidden layers
[109]	$84 * 84$	Conv	32	$3 * 3$	$ReLU$
		Conv	32	$3 * 3$	$ReLU$
		Conv	32	$3 * 3$	$ReLU$
		Conv	32	$3 * 3$	$ReLU$
[111]	$12 * 12$	Conv	-	$4 * 4$	$ReLU$
		Conv		$3 * 3$	$ReLU$
[112]	$84 * 84$	Conv	32	$3 * 3$	$ReLU$
		Conv	32	$3 * 3$	$ReLU$
		Conv	32	$3 * 3$	$ReLU$
		Conv	32	$3 * 3$	$ReLU$
[114]	$84 * 84$	Conv	32	$3 * 3$	$ReLU$
		Conv	32	$3 * 3$	$ReLU$
		Conv	32	$3 * 3$	$ReLU$
		Conv	32	$3 * 3$	$ReLU$
[115]	$70 * 70$	FC Conv	64	$3 * 3$	$ReLU$
		FC Conv	64	$3 * 3$	$ReLU$
		FC Conv	64	$3 * 3$	$ReLU$
		FC Conv	64	$3 * 3$	$ReLU$

Table 3.3: Implementation details of DRL models with a convolutional network for image input processing; "Conv" indicates and hidden convolutional layer (possibily with pooling), "FC" indicates a fully-connected layer.

Analysis As it has been extensively shown convolutional networks are the most common way to process the state of a DRL problem where the agent's observations consist of images: this does not surprise, since CNNs are still the best deep learning model for applications with images as input.
Clearly CNNs cannot be applied to contexts where the state is not representable with an image or a tensor, which does not allow to consider them as a universal state representation method in DRL; moreover, almost always im-

ages need a preprocessing step before they can be fed into the convolutional network. In [67] and all its applications, input images are first reduced in size with downsampling (from $210 * 160$ to $160 * 84$), then converted from RGB to grey-scale and finally cropped to $84 * 84$ by selecting the area of interest in the image. All this prior elaboration is needed because directly using the original image would be too expensive from a computational point of view; the preprocessing solves this issue but may add considerable overhead to the whole system.

3.2.2 Complex state: sequences

In this section, works representing the system's state as a sequence of features or observations are presented. These approaches consider a more complex view of the state, because they take into account some form of internal structure of it, here related to the sequentiality of its elements.

3.2.2.1 Background

Recurrent neural networks (RNNs) are a type of artificial neural network which uses sequential data and are commonly used for ordinal or temporal problems. They are distinguished from other neural networks by their "memory": they take information from prior inputs to influence the current input and output. While traditional deep networks assume independence between inputs and outputs, the output of recurrent neural networks depends on the prior elements of the sequence. In cases where also future elements would affect the output, bidirectional RNN can take this influence into account by adding backward relations. For further details about RNNs, see Section A.2.3. Together with backward links, another distinguishing feature of RNNs concerning classic FNNs are shared weights among layers. However, these parameters are still adjusted through backpropagation and gradient descent to facilitate reinforcement learning (for details about learning on RNNs, see Section A.2.3).

3.2.2.2 Recurrent neural networks for state representation

Some studies have proved that recurrent neural networks can provide efficient state representations for deep reinforcement learning problems.

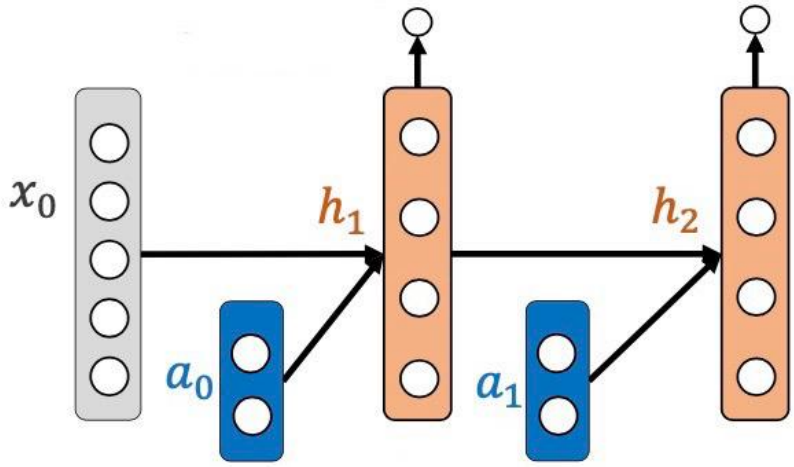

Figure 3.7: The computational graph of the model in [72]. The system starts with a representation of the entire database (x_0), and when given an action (selected using RL), the model transitions into a new state, having now built a larger subquery. Each action represents a query operation and each state captures a representation of the subquery's intermediate result.[Image source: [72]]

In [72], for solving a problem of database query optimization, a model is proposed that incrementally learns relevant properties of the data to more accurately infer query cardinality: the model learns subqueries' representations to then derive cardinality and representation of more complex queries. More precisely, the approach described in this work incrementally generates a compact representation of each subquery's intermediate results, taking as input a subquery and a new operation to predict the resulting subquery's representation. A visual scheme of the model can be seen in Figure 3.7. The model aims to learn a hidden representation that captures not only enough information to predict the cardinality of that subquery but also of other subqueries built by extending it.

In [45] a recurrent layer is added to the main DRL network architecture, after a CNN stack for processing image input (the network is the one of [101]), to perform efficient recurrent state encoding for state sequences replay. State replay can be extremely useful when the environment is partially observable,

because it can compensate for the lack of information with past experienced states. To achieve good performance in a partially observable environment, the agent needs a state representation that encodes information about its state-action trajectory, together with its current observation: a way to achieve this is by using an RNN, typically an LSTM, as part of the agent's state encoding. To train an RNN from replay and enable it to learn meaningful long-term dependencies, state-action trajectories need to be stored in replay and then used for training the network. The authors of [45] adopt two approaches, which they use both alone and combined:

- storing the recurrent state in replay and using it to initialize the network at training time: this solves some issues of the zero-initialization of the start state, of each sample sequence, which can lead to unnatural behaviours of the network during computation (since it is not true that each sequence starts with a 0 state value), but can also suffer from problems of 'representational drift' or 'state staleness', which are two relevant issues of recurrent state implementations.
- allowing the network a 'burn in' period by using a portion of the replay sequence only for unrolling the network and producing a start state, and update the network only on the remaining part of the sequence, which should help the network recover from an old, "stale" state, starting the new replay with a more updated one.

The target for the Q-function is, with both of the above approaches,

$$y_t \;=\; h\left(\sum_{k=0}^{n-1} r_{t+k}\gamma^k \;+\; \gamma^n h^{-1}(Q(s_{t+n},\, a^*;\, \theta^-))\right),$$

where

- $h()$ is the function of the hidden state of the LSTM;
- n is the number of steps, $a^* \;=\; \text{argmax}_a\, Q(s_{t+n},\, a; \theta)$;
- r_k is the reward at step k;
- γ is an hyperparameter;
- θ^- are the parameters taken from the online network's parameters θ every fixed number of steps.

The authors have compared the effects of both these methods alone and combined by using a measure called *Q-value discrepacy*, which compares the effects on the hidden state representation without these techniques and with one (or a combination) of them, normalizing their difference by the maximum Q-value to guarantee comparability of different environments and training stages. The results show that both methods bring improvements compared to the absence of them, but the best results are given by their combination. Further details of this analysis can be found in [45].

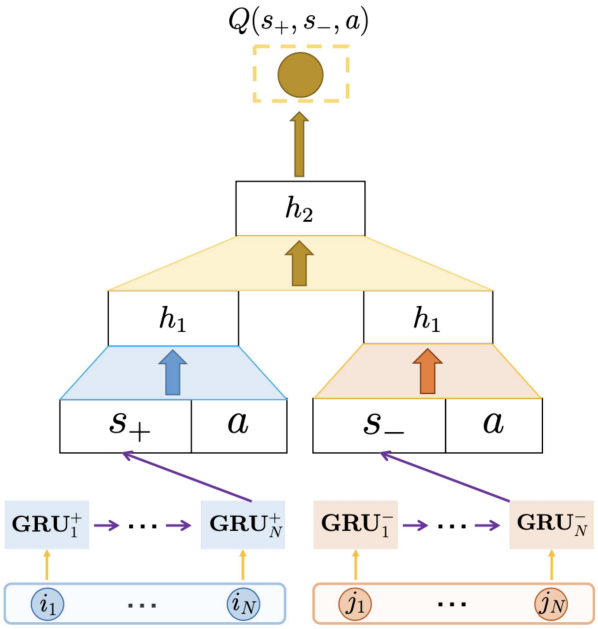

Figure 3.8: The computational graph of the model in [116]: s_+ and s_- are obtained from the processing of sequences of items with positive and negative feedback respectively in a GRU layer. The resulting s_+ and s_- are both concatenated with an item a (that is, the action of selecting that item) and then first processed separately and finally concatenated and processed together. This design choice helps capture the two distinct contributions (positive and negative) of s_+ and s_- to the recommendation.[Image source: [116]]

The system in [116] exploits reinforcement learning for recommendation systems, which are e-commerce applications assisting users by suggesting items (products, services, information) that best fit their preferences. Deep reinforcement learning is applied here to try to learn optimal strategies via recommending trial-and-error items and receiving reinforcements of these items from users' feedback; this approach, in particular, tries to use not only positive feedbacks, but also negative ones, which are often more numerous. Deep

reinforcement learning has great potential in predicting recommendations because it can try to model the dynamicity of users' preferences, which could be harder to model with a "static" recommendation system.
In this setting, the state corresponds to the output of a GRU which is given as input the sequence of (positive or negative) preferences of the user, while the action set corresponds to the items, which can be selected or not by the system. At each new user's (positive or negative) preference, the item is added to the sequence of the positive or negative state respectively.
As shown in Figure 3.8, feedbacks are used for state representation: $i_1, ..., i_N$ are the items with positive feedbacks, $j_1, ..., j_N$ are those with negative feedbacks. However, instead of just concatenating items to obtain a vector state, the sequence of items is given as input to GRU, a kind of recurrent neural network (see Section A.2.3.6). The GRU uses the update gate to generate a new state and the reset gate to control the input of the former state h_{n-1}. The inputs of GRU are the embeddings of items with positive feedbcks and the output is the final hidden state h_N, which is then used as the representation of the positive state: $s+ = h_N$. The negative state (made from items with negative feedbacks) is computed analogously. The state representations obtained via GRU are then concatenated with an action and given in input to a DQN network[67].
The loss function of the whole model is:

$$\mathcal{L}_\theta = \mathbb{E}_{s,\, a,\, r,\, s'} \left[\left(y - Q(s_+,\, s_-,\, a) \right)^2 - \alpha \left(Q(s_+,\, s_-,\, a) - Q(s_+,\, s_-,\, a^C) \right)^2 \right],$$

where $y = \mathbb{E}_{s'}[r + \gamma max_{a'} Q(s'_+,\, s'_-, a') | s_+,\, s_-,\, a]$ is the iteration target, i.e. the best item for the next state given the current one.
The term tuned by the α hyperparameter aims to maximize the difference between an item a and its competitor a^C at a particular state $(s_+,\, s_-)$; a competitor item a^C is one of the same category of item a to which the user gives an opposite feedback than a in a very close instant of time.

Analysis As the above examples show, recurrent neural networks can be perfect candidates for representing states whose structure or nature can be

considered incremental: this is the case of the queries in [72], which are seen as incrementally made by their subqueries, and also of the user-based states in [116], where the preferences of a user are processed recurrently as a sequence, to get a representation that organically takes into account all the interests of the user. This approach could probably be extended to all the applicative contexts where the state can be described through many features of the same kind (like do feedbacks of favorite items in a recommendation system), thus many scenarios of user profiling, but also a history of signals, or physical measurements.
A different approach is tried in [45]: the sequence processed by the RNN is the state-action trajectory made of consecutive agent's steps, to obtain a representation of the current state that is informed of the past history. Interestingly, this methods does not depend on the particular state structure, but can potentially be applied to any general sequence of replay buffer samples, making it relevant for a wide range of DRL systems.
However, choosing to represent the state of the system with a RNN should also consider the computational implications of this choice: as described in Section A.2.3, the time and memory complexity of training RNNs via gradient descent strongly depends on the length of the input sequence and on the type of training needed (online or offline).

3.2.3 Complex state: graphs

In this section, works representing the system's state as a graph are presented. These approaches consider a more complex view of the state, because they take into account some form of internal structure of it, here related to the relational structure of its components (for example, problems of network routing, traffic modeling, knowledge bases' analysis...).
Graph states can be first processed by some embedding methods, which generate a vectorial representation of some of its relevant features, or they can be used as direct input to the deep network of the DRL system, which must be a graph neural network. Thus, the approaches described in this section are:

- graph-based embedding representations, where embeddings provide a representation with the advantages of a compact vector state, but also

encode some domain-relevant information about data distribution.

- GNN-based deep reinforcement learning, where the state is considered as a graph and the network is designed to directly take it as an input.

3.2.3.1 Background

Graph neural networks (GNNs) are a class of deep learning models designed to perform inference on data in the form of graphs: they can be directly applied to graphs and provide an easy way to do node-level, edge-level and graph-level prediction tasks.
The reason why a specific type of neural network is needed for graph input is that "classic" deep models like FNNs or CNNs are not able to properly deal with this complex structure. A feed-forward network would need a vector encoding (embedding) of the graph; a convolutional network would not work because the convolution operation, in short, takes a little sub-patch of the image (a rectangular part of it), applies a function to it and produces a new part (a new pixel): so it aggregates information from a pixel's neighbours (and the pixel itself) in the corresponding output pixel. It is thus very difficult to perform convolution on graphs because of the arbitrary size of graphs, their complex topology (which usually presents no spatial locality) and the fact that there's no fixed node ordering for a graph, but the same result is needed for each ordering, meaning that the function applied should be invariant to it. For further details on graph neural networks, see Section A.2.4.

3.2.3.2 Graph neural networks for state representation

Recently, also graph neural networks have been exploited for state representation in DRL problems. The most intuitive applicative scenarios are those with a graph-like input, such as networks. In problems of network optimization (e.g., routing), DRL approaches based on classic deep learning models (like feedforward networks or convolutional networks) fail to generalize over new network configurations. This mainly happens because networks are essentially graphs, and classic deep learning models like FNNs or CNNs do not

perform well on graph input because they cannot properly encode their relational nature; moreover, graph neural networks have proved to well model networks in optimization tasks [2].

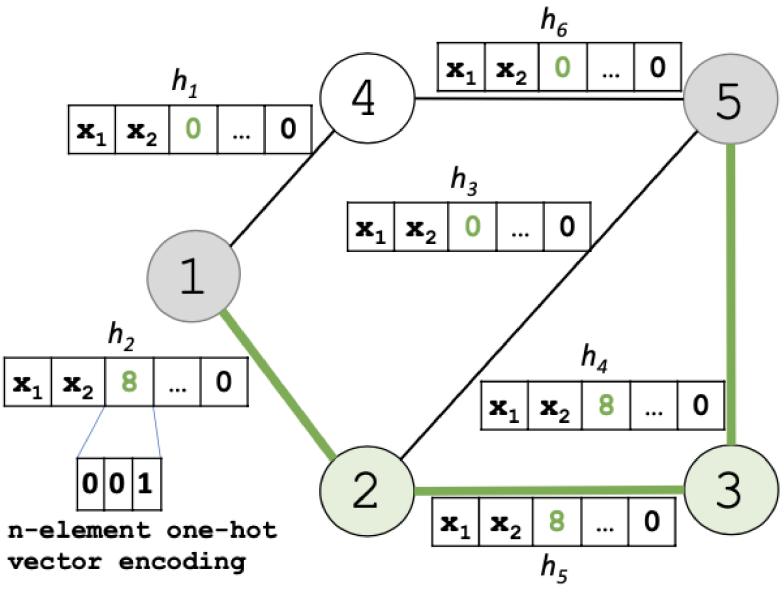

Figure 3.9: Input features for link hidden states in the network routing setting of [2]. The first two features are state features: link available capacity (amount of capacity available on the link) and link betweenness (a measure of centrality on the network); the third one is the action feature, that is the bandwidth allocated over the link after applying the routing action; the rest is zero-padding.[Image source: [2]]

For example, in the work presented in [2], the problem consists in finding the optimal routing policy for each incoming source-destination traffic demand: the DRL agent has to learn to allocate traffic demands as they arrive, maximizing the traffic volume routed through the network. At each time step, the agent receives a graph-structured network state observation and a traffic demand as inputs.
In the graph representation built by the GNN, the graph entities are the links of the topology, and link hidden states are initialized considering the input link-level features and the routing action to evaluate: the network state corresponds to the topology with some link-level features; actions are introduced as link-level feature, too (see Figure 3.9 for an example): this choice aims to improve the generalization capability of GNN, because it makes action representation invariant to node and edge permutation.
An iterative message passing process runs between the link hidden states ac-

cording to the graph structure: for each link, messages (i.e., state features) from all neighbours are propagated between all links, then messages for the same node with its neighbours are aggregated with an element-wise sum and finally the hidden states are updated. At the end of the message passing phase, which iterates the process above many times, the GNN outputs an instant reward estimate, that is evaluated over a limited set of actions; among these actions, the DRL agent selects the action by using an ϵ-greedy strategy., meaning that a random action is executed with probability ϵ, while the action with higher q-value is selected with probability $(1 - \epsilon)$.

However, not only graph-like environments can be modeled as graphs: a graph can represent any situation in which different elements communicate or interact in some ways, and this is much more general than just networks. This is the case of the systems implemented in the other works analyzed in this section ([37] [110] [70] [99]).

- In [37], the problem is behaviour generation for traffic modeling. Most reinforcement learning approaches used in behaviour generation have vectorial input, thus requiring the network to have a predefined input size, which means assuming the maximum number of vehicles. Moreover, vectorial representations are not invariant to the order and number of vehicles, while graph neural networks are invariant to those measures as they directly operate on graphs. Additionally, GNNs encode explicit relational information, which does not have to be inferred from previously collected experiences. Figure 3.10 shows a small example of the graph structure used in [37].
  All these reasons make GNNs good potential candidates for modeling decision-making problems in autonomous driving.
  In [37], the modeled graph is directed, and also in this case both node and edges contain additional information (the vehicle's state and relational information respectively); the graph is built by a GraphObserver, which observes the environment from the ego-vehicle perspective and connects the n nearest vehicles with each other; see Figure 3.10 for an example of the adopted representation.
  In the GNN, each layer has three computation steps:

  1. the next edge values are computed using the current edge values and those of the two nodes connected by that edge. These values

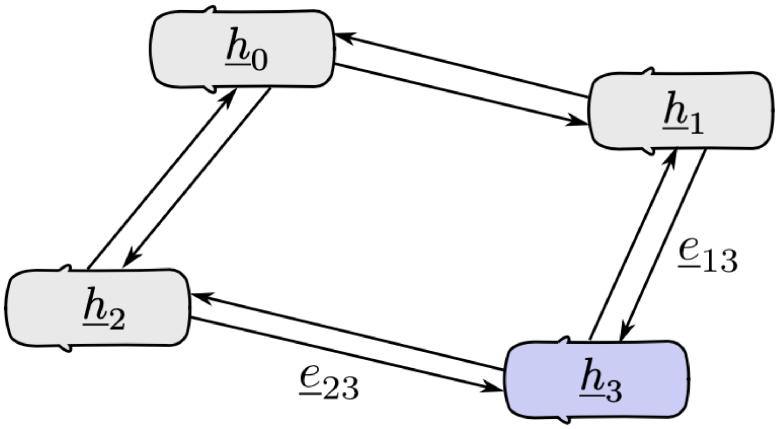

Figure 3.10: Directed graph given as input to the GNN of [37], where vehicles are connected to their two nearest neighbours. Each node n_i and edge e_{ij} have vectorial information; h_i is the value of node n_i.[Image source: [37]]

are concatenated and passed into a dense neural network layer. If e_{ij} is the edge from node n_i to node n_j, t is the current instant and f is the function associated to a single dense network layer, then

$$e_{ij}^{t+1} \;=\; f([h_i^t,\; e_{ij}^t,\; h_j^t])\;;$$

2. all incoming edge values to the node are aggregated with the sum operation; if n_{in} is the number of incoming edges into node n_j, then:

$$e_{agg,j}^{t+1} \;=\; \sum_{i=0}^{n_{in}} e_{ij}^{t+1}\;;$$

3. the next node values are computed using a dense neural network layer; if g is the function associated to a single dense network layer, then

$$h_j^{t+1} \;=\; g([e_{agg,j}^{t+1}, h_j^t])\;.$$

These three steps are performed in every layer, with each layer having dense network layers f and g. The model uses PPO[87] for performing policy learning of DRL (see Section 2.3.2 for details about PPO).

- The problem modeled in [110] is a job dispatching problem in edge computing: access points receive tasks submitted by end-users and must dispatch them to a server in a remote cloud or edge for execution. Decisions are based on many factors, such as resource and deadline requests of jobs, resource limitations of edge servers, network bandwidth restric-

tion of edge nodes and remote cloud.

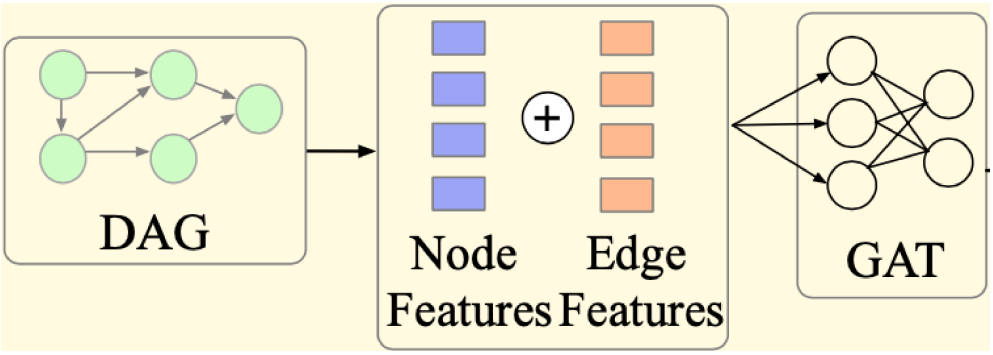

Figure 3.11: Input processing in [110]: via attention, the input graph's neighbour nodes' features are aggregated into higher-dimensional features, which are fed to a graph attention network to produce the features for the DRL module. More precisely, the figure represents the module that generates the state, which is then fed to the DRL module.[Image source: adapted from [110]]

Many deep learning algorithms have been proposed for such tasks, but usually they can't adapt to the characteristics of jobs and need to be constantly adjusted in actual applications, thus lacking flexibility and generalization. Moreover, these methods assume that data points are independent, but usually the submitted jobs contain multiple dependent tasks, composing in fact a directed acyclic graph. Thus, graph neural networks can be applied directly to graph data and aggregate the features of neighbour nodes to generate new feature representations.
Each job is fed to the GNN in the form of a directed acyclic graph: attributes of nodes and edges in the DAG are extracted and converted into abstract higher-dimensional features. Then, the module concatenates the properties of nodes, edges, and the status of edge servers to form the state. After that, the state is given as input to the DRL agent, who makes the dispatching decision based on the state. Figure 3.11 represents the job processing module of [110]. The state is then composed of the vector features output by the graph attention network (GAT, for further details see Section A.2.4.1) and the server's features, which are given as input to a DQN[67] agent (see Section 2.3.2 for details about DQN). In the GAT, the features h_i of each node v_i are computed as:

$$h_i' = \sigma\left(\frac{1}{Z}\sum_{z=1}^{Z}\sum_{j\in neighbours(v_i)} a_{ij}^z W^z h_j\right),$$

where

– σ is the sigmoid function;

- Z are the attentions;
- W is the weight matrix for nodes' features;
- h_js are the neighbours of v_i;
- a_{ij}^z are the attention weights, which are the weights of the feature of nieghbour v_j for node v_i: they are computed as

$$softmax(\alpha(Wh_i,\ Wh_j))\ ,$$

where α is a Leaky ReLU function and the softmax is applied for computational reasons.

In this way, the value of each node is computed taking into account the contribution of all its neighbours, but weighted based on the different attentions.

- The approach presented in [70] is inspired by the human process of decision-making: when confronting complicated tasks that involve expertise from multiple domains, humans typically first identify which domain of expert they should consult and then search for specific knowledge to solve the problem. This search method is much simpler than directly looking for the exact solution in all domains.

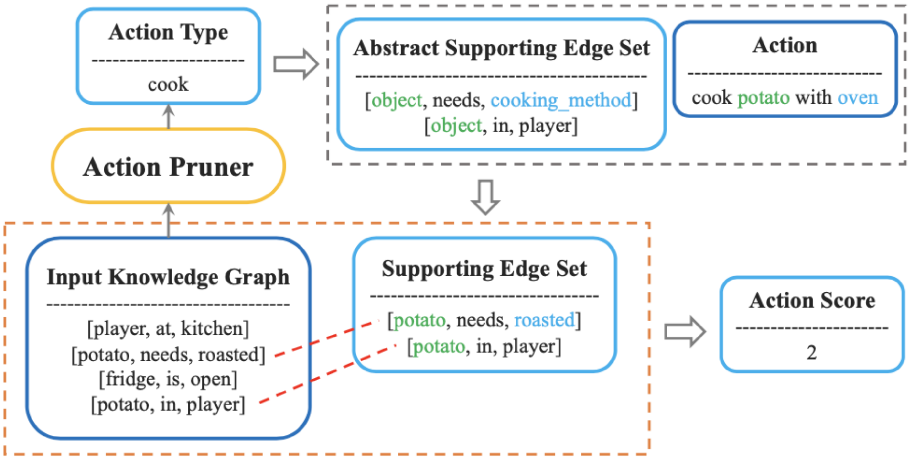

Figure 3.12: An example of the DRL framework of GNN the of [70]. Based on the knowledge graph the action pruner first predicts the action type that needs to be taken at the current state, and instantiates the abstract supporting edge set to a concrete supporting edge set. Comparing the input knowledge graph with the supporting edge set, it computes action scores for each action and selects the action with the highest score. [Image source:[70]]

Analogously, in [70] policy learning on a complex knowledge base is disentangled into a classification for problem type selection and a rule

miner. The classification finds a mapping from graph input to an action type: this module handles high-order logical interactions among node and edge representations with a graph neural network; the rule miner then conducts one-hop reasoning over the graph and provides selective explanations by mining several decisive edges. This two-step decision-making approach ensures interpretability, generalization and robustness.
The two steps are articulated as follows: the agent will receive a state at each time step, when it first calls the action pruner, implemented as a GNN, to select the action type. Then the rule-based action selector takes as inputs the current state and the action type given by action pruner to select the specific action to be executed in that step.
In order to be given in input to the GNN action pruner, the knowledge graph, drawn from the knowledge base, has to be preprocessed. Relations and nodes are initially represented in text, in the dataset: these texts are processed to obtain compact embeddings via a pretrained embedding method (fastText [64]); these embeddings are pre-computed and fixed during training, but for each relation and node they are appended to a trainable embedding vector. The final embeddings of nodes and relations are then passed through GNNs to get the latent embedding of the entire input knowledge graph. An example of the framework of [70] can be seen in Figure 3.12.

- The work presented in [99] is extremely relevant for GNN application in robotics settings.
  The authors exploit the body structure of an agent, and physical dependencies that naturally exist in such agents, relying upon the fact that the bodies of most robots have a discrete graph structure: the joints could correspond to nodes, and their relations to the edges.

  The graph in [99] has two kinds of nodes: body nodes, which are abstract nodes used to construct the kinematic graph via nesting, and joint nodes, which represents the degrees of freedom of motion between the two body nodes. Finally, a root node is added, from which other informations are observed; thus the graph is a tree. In Figure 3.13 an example of this graph-construction technique is shown.
  The model, NerveNet, propagates information between different parts of the body based on the underlying graph structure before outputting the action for each part. By doing so, NerveNet can leverage the structural information encoded by the agent's body which is advantageous in learning the correct inductive bias, and thus is less prone to overfitting.
  The propagation model follows this flow:

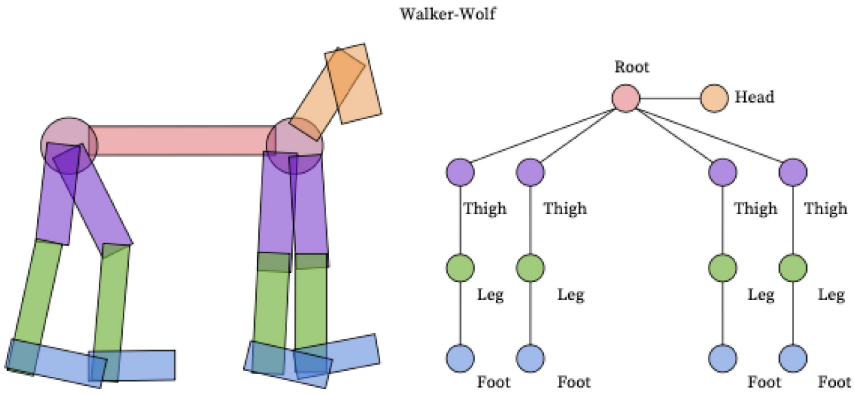

Figure 3.13: An example of the graph structure in the system presented in [99].[Image source: [99]]

1. For each time step, each node gathers a vector observation, which is first passed through an input network (e.g., a multilayer perceptron) to obtain a fixed-size state vector, operation that may involve same zero-padding, in case different nodes get observations of different sizes:
$$h_i^0 \;=\; F_{in}(x_i^0)\;,$$
where h_i^0 is the fixed state vector for node i at time 0 and x_i^0 is the observation vector of the same node. at time 0.
2. Then, the message must be created and propagated: at time t, for each out edge $(i,\ j)$ from node v_i, the message is computed as
$$m_{(i,j)}^t \;=\; M_{(i,j)}(h_i^t)\;,$$
where $M_{(i,j)}$ is the message function, which can be a MLP.
3. After this, messages for each node from all their neighbours are aggregated :
$$m_i^t \;=\; A(h_j^t \mid j\ incoming(i))\;,$$
where A is an aggregation function (that can be summation, average or max-pooling) and $incoming(i)$ is the set of nodes from which starts an edge to node i.
4. Finally, every node's state is updated based on the aggregated message and its current state vector:
$$h_i^{t+1} \;=\; U_i(h_i^t,\ m_i^t)\;,$$
where U_i is the update function, which can be implemented as a

RNN (like a GRU or a LSTM unit) or a simpler MLP.

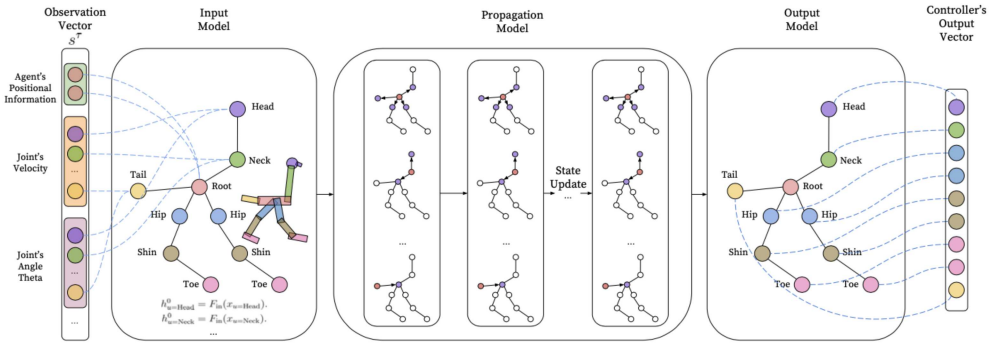

Figure 3.14: An example of NerveNet [99]. In the input model, for each node, NerveNet fetches the corresponding elements from the observations, then computes the messages between neighbours in the graph and updates the hidden state of each node. This is repeated for each propagation step. In the output model, the policy is produced by collecting the output from each controller.[Image source: [99]]

This process is recurrently applied for a fixed number of time steps to get the final state vector of all nodes, which is then fed to a controller model that implements the policy. Figure 3.14 shows an instantiation of the whole NerveNet pipeline.

Analysis As it has been shown in this section, graph neural networks offer a strong potential for properly representing the state in DRL problems: they allow to generalize over different topologies, or, more in general, different input configurations, requiring less effort of "manual" adaptation of existing models to new contexts or tasks. This is the main common trait of all the approaches analyzed in this section: working on a graph input (thus, using a graph neural network) improves models' generalization and flexibility, which is also extensively proved by all the works mentioned in their experiments.
It is worth mentioning that graph representation seems to perform well not only on systems with an "evident" graph-like structure, such as networks[2], but also in all those situations where some elements are connected (physically or not) and interact with each other: traffic modeling, [37] job dispatching, [110] inference in databases[70] and even skeletons of robots [99]. Moreover, in all these situations, the graph representation effortlessly encodes the relational bias among the nodes (i.e., the communicating elements of the system),

without the need to manually encode it or learn it as an extra task. In other words, like many structured data representations, graphs encode much more information than other simpler forms of data, which is usually a desirable feature, because more information about data means more knowledge about the problem to exploit for improving efficiency and performance.

3.2.3.3 Graph-based embeddings

Representation learning approaches generating embeddings have been introduced also for DRL problems with a graph-like state form. Also in this case, graph representation learning methods are applied both on graph input [106] and on generic input [107] [102].

In [106] the problem is multi-hop reasoning over knowledge graphs, that is learning explicit inference formulas. The task is formulated in the form of a sequential decision-making problem that can be solved by a RL agent. To encode the semantic information of the knowledge graph's entities and relations, translation-based embeddings are used (TransE [10] and TransH [100]), that map the symbols to a low-dimensional vector space. The state vector is represented by a tuple

$$\boldsymbol{s}_t = (\boldsymbol{e}_t,\ \boldsymbol{e}_{target} - \boldsymbol{e}_t)\ ,$$

where $\boldsymbol{e}_t$, is the embedding of the current state at time t and $\boldsymbol{e}_{target}$ is the embedding of the target state.

For the two embedding methods TransE and TransH, let us represent an edge in a knowledge graph (KG)[‡] by a triplet $(head\ entity,\ tail\ entity,\ relation\ entity)$, shortly $(h,\ r,\ t)$, and denote the embedding function as $\phi(\boldsymbol{h},\ \boldsymbol{t})$; then, the two ϕs for TransE and TransH are:

- $\phi_{TransE}(\boldsymbol{h},\ \boldsymbol{t}) = ||\boldsymbol{h} + \boldsymbol{r} - \boldsymbol{t}||_{\ell_{\frac{1}{2}}}$ where $\boldsymbol{r} \in \mathbb{R}^k$ with k size of the embedding space, and $\ell_{\frac{1}{2}}$ is a norm L^p with $p = \frac{1}{2}$;

[‡]A knowledge graph, or semantic network, represents a network of real-world entities (objects, events, situations, or concepts) and illustrates the relationship between them; this information is usually stored in a graph database and visualized as a graph structure, hence the name knowledge graph.

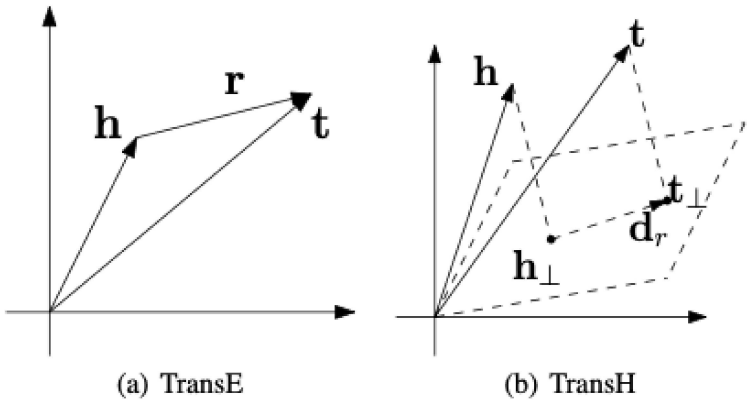

Figure 3.15: A graphic representation of the two embedding methods TransE and TransH. TransErepresents a relation by a translation vector $\boldsymbol{r}$ so that the pair of embedded entities in a triplet $(h,\ r,\ t)$ can be connected by $\boldsymbol{r}$ with low error; TransH starts from TransE and, for the relation $\boldsymbol{r}$, positions the relation-specific translation vector $\boldsymbol{d}_r$ in the relation-specific hyperplane $\boldsymbol{w}_r$ (the perpendicular vector) rather than in the same space of entity embeddings $\boldsymbol{h}$ and $\boldsymbol{t}$.$\boldsymbol{h}_\perp = \boldsymbol{h} - \boldsymbol{w}^T\boldsymbol{h}\boldsymbol{w}$ and $\boldsymbol{t}_\perp = \boldsymbol{t} - \boldsymbol{w}^T\boldsymbol{t}\boldsymbol{w}$.[Image source: [100]]

- $\phi_{TransH}(\boldsymbol{h},\ \boldsymbol{t}) = ||(\boldsymbol{h} - \boldsymbol{w}^T\boldsymbol{h}\boldsymbol{w}) + \boldsymbol{d} - (\boldsymbol{t} - \boldsymbol{w}^T\boldsymbol{t}\boldsymbol{w})||_2^2$ where $\boldsymbol{w},\ \boldsymbol{d} \in \mathbb{R}^k$ with k size of the embedding space.

Figure 3.15 shows a graphic representation of the two methods.
TransE is really effective for state-of-the-art predictions but has flaws in dealing with reflexive/one-to-many/many-to-one/many-to-many relations;
TransH starts from TransE and tries to solve its issues: it enables an entity to have distributed representations when involved in many relations.

In [107], on the other hand, a more general graph-based representation learning method is studied. In this case, like also in [102], the graphs are not the input data of the model, but rather the actual model itself: Markov decision processes correspond to a graph structure themselves, and for this reason graph representation techniques can be applied to them.
The authors here aim to solve the problem of the explosion of state space by finding an abstract MDP (AMDP) with a smaller abstract state space, generated not manually, but by defining a proper state representation method for encoding states, which, aggregated via K-means[58], generate the AMDP.
The state representation approach presented in [107] introduces is based on the idea that if two states are in the same abstract state they should be topolog-

ically close but also their long-term expected discounted reward should also be similar. Thus, their embeddings encode similarity among states in terms of both topology and reward. Topological proximity here is intended as the reachability of two nodes in a small number of steps; the reward similarity, on the other hand, is needed because, in the AMDP, similar nodes are attributed similar rewards, thus there must be this correspondence also between ground states in the original MDP.
The authors introduce a reward exploration strategy, which generates experiences with a bias towards making states sharing similar reachability and ground value functions co-occur more frequently. The exploration phase is needed because the graph is not assumed to be known in advance. Experience gathered during exploration is then used to learn state representations: the sequences of nodes are passed through a SkipGram model[63] to generate the embeddings. These representations are then clustered into abstract states for constructing the AMDP. The target function to maximize for state representation is

$$\sum_{s \in V} \sum_{s' \in \mathcal{N}(s)} \left(log(\sigma(\Phi(s) \cdot \Theta(s'))) + \sum_{k=1}^{\kappa} \mathbb{E}_{s'' \sim P} \, log(\sigma(-\Phi(s) \cdot \Theta(s''))) \right),$$

which is the SkipGram cross-entropy loss function, where:

- Φ, Θ are the representations for states and contexts;
- V is the set of nodes in the graph G corresponding to the MDP of the system;
- $\mathcal{N}(s)$ is the set of co-occurring states of the state s;
- κ is a constant for the number of negative samples (of SkipGram)[§];
- P is a random distribution over the observed node set;
- $\sigma(x) = 1/(1 + e^{-x})$ is the sigmoid function.

[§]Negative sampling is a technique applied in SkipGram models to solve a computation problem: it avoids the computation of a factor over the entire state set, maximizing the similarity of the states in the same neighbourhoods and minimizing it when they occur in different neighbourhoods. Instead of doing the minimization for all the states except for the neighbourhood, it randomly selects k states and uses them to optimize the objective.

A work similar to the one in [107], but more comparative, is presented in [102]: they perform a comparative analysis of many graph-based state representation approaches in grid-world navigation tasks, which are representative of a large class of RL problems. From their experiments, it results that all embedding methods outperform the more commonly used matrix representation of grid-world environments in all of the studied cases.
More precisely, they show results for six methods:

- DeepWalk[77]: a random-walk-based[¶] method which exploits advanced natural language processing techniques; it generates node sequences from graphs using short random walks and trains the SkipGram model[63] using hierarchical softmax like in word2vec-based training procedure[63]. Embedding methods attempt to find node embeddings such that the likelihood of observing a node given its neighbouring vertices within a specified number of random walk steps is maximized. Given v_{i-m}, ..., v_{i-1}, v_{i+1}, ..., v_{i+m} set of neighbours of a node v_i and ϕ the mapping for the representation, the target function is:

$$\min_{\phi} -logP(v_{i-m}, \ ..., \ v_{i-1}, \ v_{i+1}, \ ..., \ v_{i+m} \mid \phi(v_i)).$$

- NERD[46] (Node Embeddings Respecting Directionality): exploits the fact that, in directed graphs, the neighbourhood of a node differs based on its role as a source or a target. To model node similarity while preserving its role, the authors introduce an alternating random walk strategy that alternates between source and target nodes; two embeddings are learned for each node, such that the probability of observing the sampled neighbours (in their respective roles) from an alternating random walk is maximized. Given r_1, r_2 roles, ϕ_{r_1}, ϕ_{r_2} the respective representations, κ the constant of negative samples and $P_{r_i}(v')$ the in/outdegree noise distribution (depending on the target role), the target function is:

$$\mathcal{L}(u, \ v) \ = \ log\sigma(\phi_{r_1}(u) \ \cdot \ \phi_{r_2}(v)) \ + \ \kappa\mathbb{E}_{v' \sim P_{r_2}(v')} \, log\sigma(-\phi_{r_1}(u) \ \cdot \ \phi_{r_2}(v')).$$

- HOPE[75] (High-Order Proximity preserved Embeddings): embedding framework designed for directed graphs; it aims to find source and target node embeddings while optimizing for various high-order proxim-

[¶]A random walk is a process by which randomly-moving objects wander away from where they started; on graphs, random walks are sequences of nodes generated from a start vertex by selecting an edge, traversing the edge to a new vertex, and repeating the process.

ity measures. If $\boldsymbol{S}$ is the proximity matrix, $\boldsymbol{U}^{r_1}$, $\boldsymbol{U}^{r_2}$ the representation matrices for roles r_1, r_2, the target function in matrix form is:

$$\mathcal{L} = \min_{\boldsymbol{U}^{r_1}, \boldsymbol{U}^{r_2}} ||\boldsymbol{S} - \boldsymbol{U}^{r_1} \cdot (\boldsymbol{U}^{r_2})^T||^2.$$

- APP[118] (Asymmetric Proximity Preserving): conceived for both undirected and directed graphs, this method uses an approximate version of Rooted PageRank in which many paths are sampled from the starting node using a restart probability. Each node of the graph is given two representations, source and target, such that the likelihood of a training of vertices in their respective source and target roles is maximized. Given representations ϕ_{r_1}, ϕ_{r_2} for roles r_1, r_2, number of negative samples κ and node distribution P, also for this model, the loss function takes from SkipGram and has the form:

$$\mathcal{L}(u, v) = log(\phi_{r_1}(u) \cdot \phi_{r_2}(v)) + \kappa \mathbb{E}_{t_n \sim P(n)} log(\phi_{r_1}(u) \cdot \phi_{r_2}(n)).$$

- GraphSAGE[36]: it is a graph convolutional network (GCN) model, meaning that it incorporates a neighbourhood aggregation mechanism in the learning algorithm to generate node representations. A graph-based loss is used which pulls nearby nodes to have similar representations and far apart nodes are to distinct representations. Given representation ϕ which can be obtained through the different aggregation mechanisms, number of negative samples κ and node distribution P, the loss function is:

$$L(u, v) = -(log(\sigma(\phi(u)^T \phi(v))) + \kappa \mathbb{E}_{v_n \sim P(v)} log(\sigma(-\phi(u)^T \phi(v_n)))).$$

- GLAE[82]: this approach belongs to a family of models aiming to map each node to a vector in a lower dimension, from which then reconstruct the node and its neighbourhood. A linear model is used because, according to the authors of [102], this was the best performing among many other types of graph autoencoders, both in their shallow and deep forms. The embeddings are obtained by simple matrix multiplication:

$$\Phi = \tilde{A}W\,,$$

where $\tilde{A}$ is the normalized adjacency matrix of the graph and W is a weight matrix, tuned by gradient descent; if there are node-level features x, they are encoded as $\Phi = \tilde{A}XW$.

Results of these experiments by [102] show that the best-performing GRL model on undirected graphs is DeepWalk, which surprisingly works better than many other approaches also on directed graphs, even though it does not consider edge directionality.
This may allow us to think that pure random-walk-based embedding approaches are the best among the ones considered in [102]. However, when analyzing these experiments it must be considered that, during their executions, the embeddings and/or graph representations were pretrained and precomputed, so they were not contextually learned during DRL training.

Analysis As it can be seen, almost all embedding methods presented in this section adopt a similar loss function structure, which is taken from natural language processing techniques: a cross-entropy loss with negative sampling, which produces similar representations for co-occurrent/close nodes and more dissimilar ones for other nodes. It is interesting how some of the approaches described here succeed in applying graph techniques for state representation also to contexts where the system is not actually a graph, by exploiting the fact that a Markov decision process, which is the mathematical model of reinforcement learning problems, actually identifies a graph.
However, the nature itself of embeddings risks losing some of the information those graphs can provide: by trying to encode nodes into vector representations and then working with these vectors, some relevant information about the relational structure of the graph can be lost, for example the different densities that neighbourhoods can have. Embeddings can encode some features of a graph and its nodes (like co-occurrence, as many methods presented here do), but in fact they cannot be as expressive as the original graph, with its nodes and edges, can be. This design choice can bring some advantages in terms of the complexity of the model, because working on vector embeddings is usually simpler than directly dealing with complex structures like graphs, but this potential lack of expressivity may be damaging in some scenarios.

4

Conclusions and future perspectives

4.1 Conclusions

State representation is one of the main issues in deep learning models, and thus also in deep reinforcement learning systems: deep neural networks often require an extremely large amount of data samples to be properly trained, and this gives rise to two main problems: having access to all these data and spending time to train the model on all of them. A way to overcome this problem is to feed the deep neural network with efficient input representations (state representations, in the case of deep reinforcement learning), which already encode in themselves much information about the data and their distribution, this way reducing the number of samples needed to properly train the model. Thus, the focus now switches to finding the most effective state representation techniques for different deep learning models, which, in this case, are deep reinforcement learning ones.

A wide survey of recent state representation approaches in deep reinforcement learning systems has been presented in this work. These techniques have been categorized based on the specific shape their state representation takes and on the machine learning method used to process them. The two main families of approaches are those using a deep neural network to produce

a state representation, like feedforward networks, convolutional networks, recurrent networks or graph networks, and those using embedding methods based on topological features or the model structure: bisimulation-based and graph-based embedding methods.

The most popular deep learning model used for state representation in deep reinforcement learning seems to be convolutional neural networks, as shown by the large number of works that utilize them for processing their input: this is probably because many deep reinforcement learning applicative domains have to process observations in the form of images or grids (for example, obtained from visual sensors), for which convolutional networks are notoriously the best deep learning model, while other types of deep networks are not able to properly encode their bias.

Surprisingly, recurrent neural networks may have a relevant potential as state representation models in those cases where the state or its composing features have a recursive or incremental nature, like for user profiling in recommending systems, where an incremental state can be built feeding a recurrent network with the history of the user's preferences.

Finally, the deep model which seems to be the most universal above all are graph neural networks: state-of-the-art works have implemented many graph neural network models in which the graph representation is given for an input network (which has, in the end, an evident graph structure) or knowledge graphs, or for those environments in which elements interact with each other, forming a sort of relational structure that can be represented as a graph.
Nevertheless, as some embedding-based approaches have addressed, the real potential of graphs in deep reinforcement learning lies in the fact that the Markov decision process, according to which almost all deep reinforcement learning problems are modeled, defines *itself* a graph: thus, potentially the state of every deep reinforcement learning evolving system can be represented as the node of a graph. It would be worth trying to explore this direction, trying to model as graphs generic Markov decision processes and to train graph neural network-based representations on them, while keeping the model re-

alistic and applicable in real-life scenarios (that is, taking into consideration training time, degree of knowledge of the environment, assumptions on the systems, etc.).

Moreover, it would be worth exploring the direction of using a structured state representation in deep reinforcement learning systems whenever the context allows for it. This kind of reasoning holds for both graphs and sequence: exploiting the structure of the state, instead of mapping its features into a low-dimensional vector, allows a more expressive representation, which directly encodes relational and temporal dependencies, without requiring further processing.
It is worth mentioning that using a sequence or graph representation does not necessarily imply that the whole deep reinforcement learning system must use a recurrent or a graph neural network, respectively: approaches like the ones presented in [116] and in [110], which represent their state as a sequence and a graph, respectively, but then both use the DQN network[67] for policy learning.

4.2 Future perspectives

Some efforts have been made in the direction of exploiting the graph underlying a Markov decision process for state representation by some works on embedding techinques[102]; however, embeddings are usually not able to encode the relational bias among nodes like a graph structure can; additionally, a graph neural network can have higher generalization and flexibility than a fixed-size embedding method, which usually builds its representations based on a unique, specific criterion (or a small number of them), encoding just a particular notion of relation between nodes, that may not be suited for different kinds of scenarios. It may be then worth investing energies in trying to understand whether graphs can really be a "universal" kind of state representation for deep reinforcement learning problems, finding new ways of designing the graph corresponding to the MDP and exploiting deep graph models for processing it.

Additionally, it would be interesting to compare different state representation techniques on the same environment and task, choosing a realistic, complex enough setting to also allow for structured state representations, and then trying to exploit the most recent advancements in deep models to obtain a state representation; from these results, it would maybe become possible to classify deep reinforcement learning problems based on the complexity of their state representation.

A

Deep learning generals

Deep learning is a machine learning approach based on multi-layer perceptron networks (MLP). It is known that a single hidden layer MLP is a universal function approximator, but the number of hidden units needed (thus, of hidden parameters) and of training data it would need to learn the policy could become extremely large. Moreover, it has been empirically proved that deep neural networks (DNNs) with more layers, each one with a small number of hidden units, have better generalization, meaning that they perform well also on unseen data (i.e., which were not in the training set).

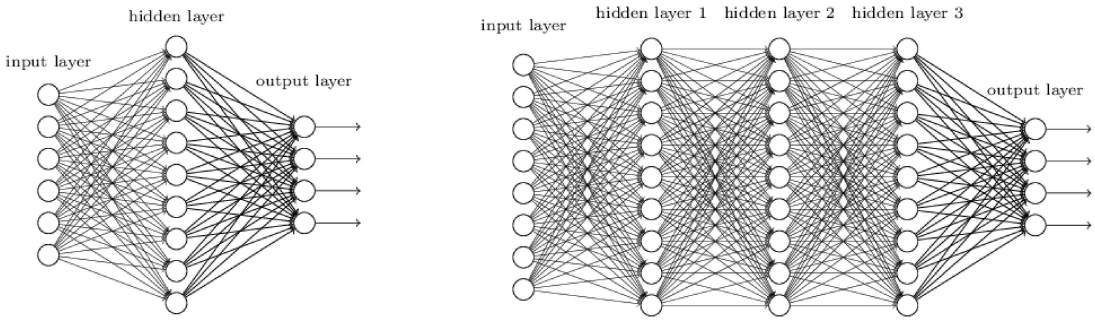

Figure A.1: A comparision between a single-layer MLP (*left*) and a multiple-layer MLP (*right*). A larger number of hidden layers allows to have much more hidden units without increasing the size of each layer: this results in higher expressivity of the model together with better generalization.
Image source: adapted from http://neuralnetworksanddeeplearning.com/chap5.html].

The general idea behind building a DNN with many hidden layers is that each layer combines its input values and learns abstract representations of it; thus, the final layer gives as output an abstract representation of the initial input, which should encode as many structural information about raw data as possible. This is particularly useful when data are complex or highly dimensional: in this case, it would be difficult and expensive identify and/or design proper features for the data by hand; this would make feature design expensive and inefficient. Instead, deep learning methods can create accurate predictive models from large quantities of unlabeled and unstructured data.

To achieve an acceptable level of accuracy, deep learning models need to access to large amounts of training data and processing resources, which have become available to programmers just in recent years, but still in many applicative domains are hard to find.

A.1 Training techniques

Deep neural networks represent nonlinear functions, and this often causes loss functions to be nonconvex, meaning that convergence is not always guaranteed (as it is, instead, with convex models).

A.1.1 Gradient descent

The most common techniques for DNNs optimization are based on *gradient descent*:[33] the gradient of the cost function is usually computed using the *backpropagation* algorithm, and then the actual learning is performed by specific optimization algorithms, which update the different parameters of the model.

A.1.1.1 Backpropagation

Backpropagation computes the gradient on the functions of each network layer, from output to hidden ones, indicating how each layer's contribution should change in order to reduce the final error. The gradient is computed on the parameters of each layer, and then it is used by the learning algorithm to

actually update parameters; this can be done in different fashions, according to the specific optimization algorithm chosen (for example, the update can be applied as soon as it is computed, or it can be delayed after other operations).

A.1.1.2 Forward propagation

Even though backpropagation is the most widely used algorithm for gradient computation, there are some applications where it is not convenient (or possible) to keep track and store a significant history of data, or to wait until the data batch is complete to perform parameter updates. This is the case, for example, of recurrent neural networks, a particular kind of deep network with cycling internal connections between layers, which can deal with sequential data (see Section A.2.3 for further details). When the data sequence's length becomes really large, or learning has to be performed in an online fashion, classic backpropagation (which is usually performed in the backpropagation through time algorithm, which has been independently developed in different studies [69] [80] [103]) cannot be applied: to deal with these situations, *real time recurrent learning* algorithm has been introduced, [104] [81] which allows training of arbitrary long sequences and fully online parameter updates using forward propagation of the error's gradient.

A.2 Neural networks in deep reinforcement learning

Deep reinforcement learning models can use different kinds of deep neural networks, depending on the data type the models have to work with (vectors, images, graphs), or to specific features of the application; the most common in literature are:*

- deep feedforward neural networks, possibly fully connected (FNNs), A.2.1;
- convolutional neural networks (CNNs), (see Section A.2.2);
- recurrent neural networks (RNNs), (see Section A.2.3);
- graph neural networks (GNNs), (see Section A.2.4);

*Where not further specified, all contents of this section are taken from [33]

- autoencoders (AEs), (see Section A.2.5).

A.2.1 Feedforward networks

Deep feedforward networks (FNNs) are MLPs: the goal of these models is essentially to learn a target function, but this is often unfeasible (for either computational or numerical reasons), so usually an FNN aims to learn an approximation of the target function. More precisely, it generally computes a mapping

$$\boldsymbol{y} = f(\boldsymbol{x}, \boldsymbol{\theta})$$

and learns the best paramter vector $\boldsymbol{\theta}$.
Information flows just forward in the network (hence the name of this class of models): it starts with input data, goes through all hidden layers and then reaches output units, where the final value of the function is computed. Some graphic examples of FNN can be seen in Figure A.1.

A.2.1.1 Semantic

A FNN can be thought as a composition of functions: each layer of the network corresponds to a function in the composition and the output of one layer is the input of the following one, as happens in function composition. Since the only output value known at training time is the one of the last layer, all other layers (between the first and the last one) are called "hidden" and the output of each hidden unit is called "activation value".

A.2.1.2 Learning an approximation

FNNs are not able to exactly learn the target function because they just have access to data points in the training set, which are necessarily finite, while a function may be continuous and, more importantly, there usually are some data points for which the network must be able to predict the output but which are not present in the training set. The ability of a network to correctly predict the output on unseen data is called *generalization*.

Even if the network is not able to perfectly mimic the objective function, it can try to learn the internal structure and features of training data, which, combined with some supervised information (if it is the case), allows the network to guess the output value of unseen data with an acceptable degree of precision, thus to approximate the optimal function.

FNNs are at the basis of many other deep learning models, like convolutional networks, for image recognition, and recurrent networks, for learning, which are described in following sections.

A.2.2 Convolutional networks

Convolutional neural networks are models specialized in processing data with a grid-like structure, thus mainly images.

A.2.2.1 Convolution

This family of models is named after a mathematical operation, the *convolution*. A convolution is an operation on two functions and its general form is

$$c(x) = (f * g)x \,,$$

where f and g are continuous functions and $c(x)$ is the convolution operation. In deep learning on images, the operation usually takes the form

$$C(i,\, j) = (I * K)(i,\, j) = \sum_m \sum_n I(m,\, n)K(i-m,\, j-n) \,.$$

In this case, the convolution is applied on two axes, I is a grid corresponding to an image and K is the one corresponding to the kernel; usually, the I is called input and the result of the convolution is called convoluted feature or feature map.
A convolutional network is a FFN where standard matrix multiplication is substituted with convolution between matrices in at least one layer. Usually, when talking about convolution in deep learning it is intended as a multiple application of convolutions with different kernels, because a single kernel is

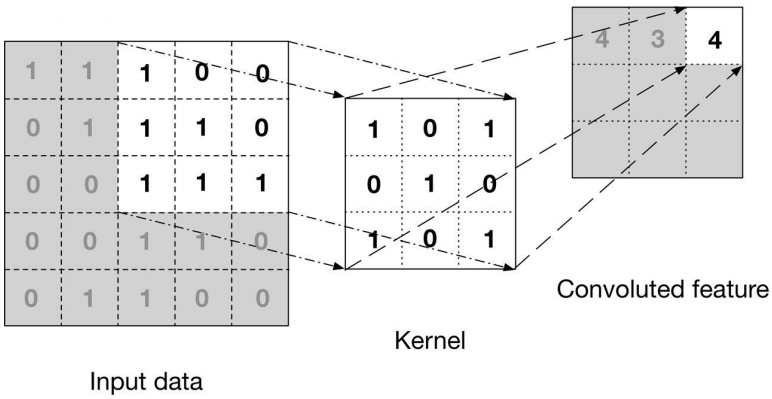

Figure A.2: A convolution application example: the image is a grid of size 5×5, the kernel has size 3×3 and the feature map 3×3.
[Image source: adapted from https://www.oreilly.com/library/view/deep-learning/9781491924570/ch04.html].

just able to detect a single feature, while it is desirable to work on many features simultaneously.

A.2.2.2 Pooling

A convolutional layer is usually made of three components:

1. a convolution stage, where possibly many convolutions are performed in parallel;
2. a detector stage, where all liner outputs of the previous stage are run through a nonlinear activation function;
3. a pooling stage, where a summary statistic is applied to the nonlinear activations.

Pooling functions are applied on a fixed-size neighbourhood, iteratively until covering all activations. The most popular pooling functions are *max*, *average*, *weighted average* and L^2.
Pooling makes the result of a convolution invariant to local translation, i.e. pooled values of the same neighbourhood are mostly equal, so small traslations do not change them; this is a desirable feature, for example, in image processing, where learning to recognize a feature in the gird is more important than its position in it.

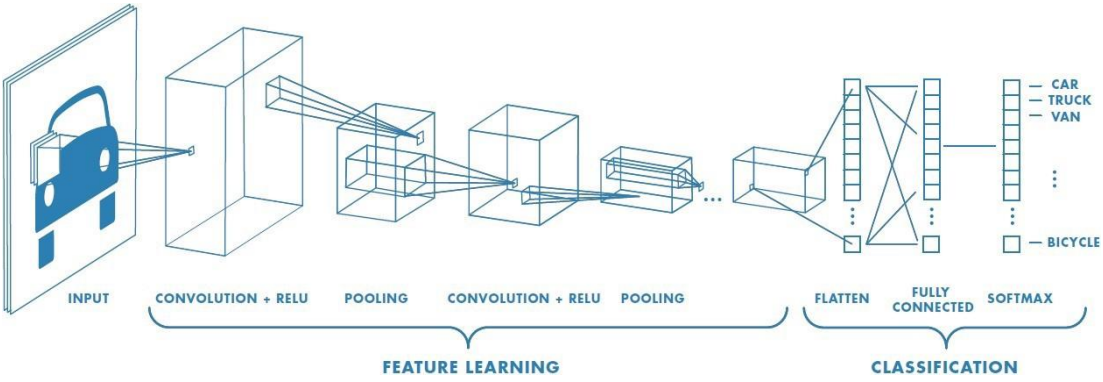

Figure A.3: A typical CNN architecture for image classification: each convolutional layer has a convolution stage, a detector stage (the ReLU function) and a pooling stage; the output of the last convolutional layer is then flattened and used as input in a fully-connected MLP, which finally performs classification with a $softmax$ output function. [Image source: adapted from https://python.plainenglish.io/convolution-neural-network-cnn-in-deep-learning-77f5ab457166].

A.2.2.3 Convolution and pooling as prior

The search for the best approximation function can be often guided imposing a prior distribution over model's parameters, encoding some kind of prior belief which can help in reducing the search space. The convolution operation, together with pooling, can be seen as an infinitely strong prior over the parameters of the network.[33] If we think of the CNN as a fully-connected FNN, an infinitely strong prior over parameters pulls the network to have similar weights in all hidden units shifted of a fixed amount, and that weights must be zero except for those in the same neighbourhood (where the neighbourhood size grows cascading from one layer to another). In other words, this prior would enforce equivariance to traslation in general and invariance to small trasaltions, which are exactly the effects of convolution and pooling.

A.2.2.4 Convolution issues and variants

The standard convolution operation presents some problems, which have been addressed and solved in some variants of this technique:

- Input data of CNNs are usually images, which are multichannel data (a channel for each value of green, red and blue), so also convolution must be *multichannel* and kernels too.

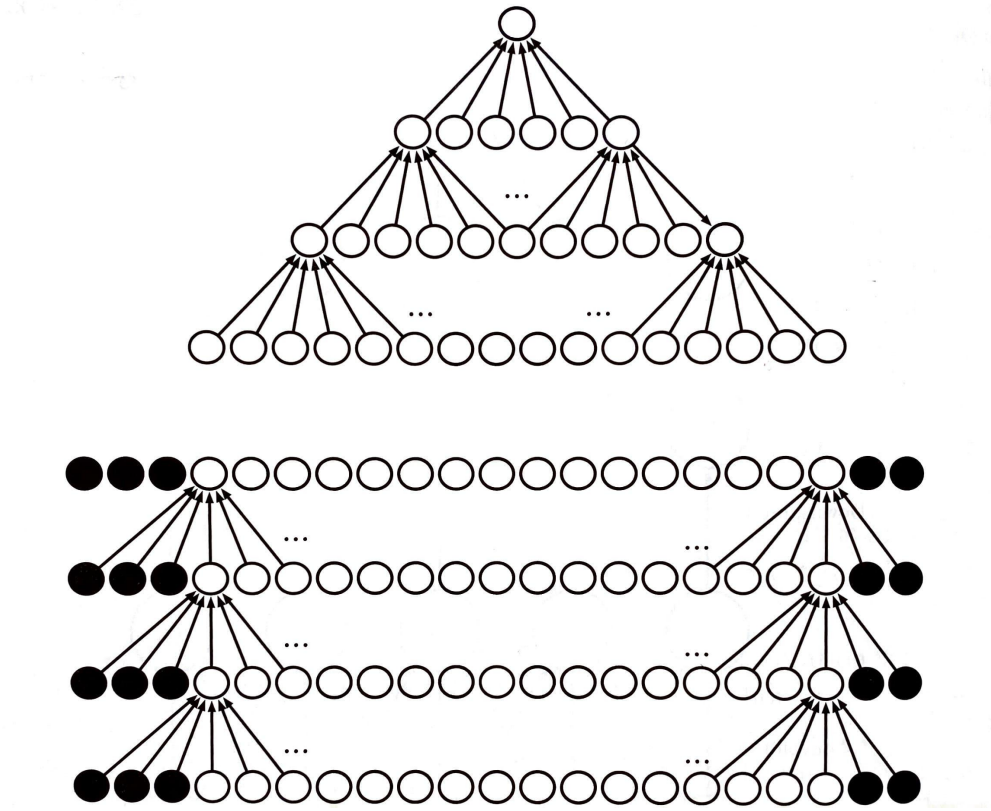

Figure A.4: *(Top)* A CNN with no 0-padding: at each convolution, intermediate representations shrink of one unit less than the kernel size (here, the kernel size is 6 so the shrinkage is 5; *(bottom)* applying a same padding to each hidden layer of the CNN keeps the representation dimension equal after every convolution.[Image source: [33].

- As shown in Figure A.4, a mere application of convolution on an image and a kernel causes the output dimension to diminish of almost the size of the kernel at each application; this would drastically reduce output size in a few convolutions, and using smaller kernels would just slightly slow down the process. In any case, this problem would provoke a limitation to the network's expressive power. The solution to this problem is *zero-padding*, which consists in increasing the input size by adding 0 elements to the perimeter of the grid: increasing input's dimensions at each convolution avoids its drastic reduction. There are three main kinds of zero padding:
  - *valid*: no padding is applied, and the kernel is allowed to visit only those areas where it is completely contained in the image; this is the scenario of input shrinkage, which consequently limits the size of the network to a maximum number of hidden layers such that the

input cannot enter another convolution without disappearing;

  - *same*: enough 0 pixels are added to just guarantee that the size of input remains the same after the convolution; this solves the problem of shrinkage but changes the influence of single pixels on the convolution output: in valid padding, all pixels of the output are function of the same number of pixels of the input, while in the case of same padding external pixels of the input image are "underrepresented" in the output;
  - *full*: enough 0 pixels are added so that every pixel of the original image is visited the same number of times in each direction; this solves the problem of same padding but now output pixels near borders are function of fewer input pixels than those near the output center.

- In applications where it is necessary to detect some feature in a specific area of the image, but not to look for it in the whole space, applying the same kernel on the entire image may not be useful; in these cases, where local connectivity is needed but not weight sharing, *unshared convolution* can be used: it consists of implementing local connectivity in specific areas of the network, but each one with its specific weights.
- Tiled convolution uses a set of kernels rotating them over the representation: in this way, adjacent areas will have different sets of weights, but they will not be all different over the entire layer; in this way, an effect similar to the one of unshared convolution can be achieved, but with much less need of memory, because the only extra space required is for the set of kernels, not for all the different weights of a layer.

A.2.3 Recurrent networks

Recurrent neural networks (RNN) are neural networks for processing sequential data, able to scale to input sequences of variable length. Analogously to CNNs, also in RNNs parameters are shared across different components of the model, increasing the generalization of this kind of network.
RNNs usually operate on sequences in time, meaning that a sequence corresponds to a vector and each element is associated to a time step t, even though

RNNs often work on batches of these sequences, and can also be applied to bidimensional input, such as images.
As a naive example, the recurrent equation

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})$$

corresponds to the RNN in Figure A.5.

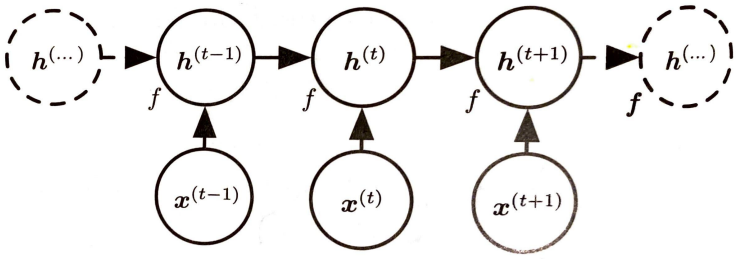

Figure A.5: A simple RNN with no output: the network just process information, incorporates it in the hidden unit and propagates it through time. [Image source: [33]].

There is actually a similarity between CNNs and RNNs: with a one-dimension convolution, a CNN can operate on a temporal sequence; however, convolutions are shallow operations, which enforce parameter sharing only in a small neighrbourhood and whose outputs thus depend only on that neighbourhood. On the other hand, in RNNs each output depends on all previous elements in the sequence and of the output. In CNNs, parameters are shared in the kernel, in RNNs they are shared in the update rule, which is the same recurrent formula for all outputs, which is, for a RNN like the one in Figiure A.6 (not considering, for the moment, the loss $L^{(t)}$ and the supervision $\boldsymbol{y}^{(t)}$),

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + Ux^{(t)}) ,$$

$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} ,$$

where $f()$ is a nonlinear function and $\boldsymbol{h}^{(0)} = 0$.

A.2.3.1 Recurrence

The recurrence in this kind of network lies in the expressions of the elements of the model.

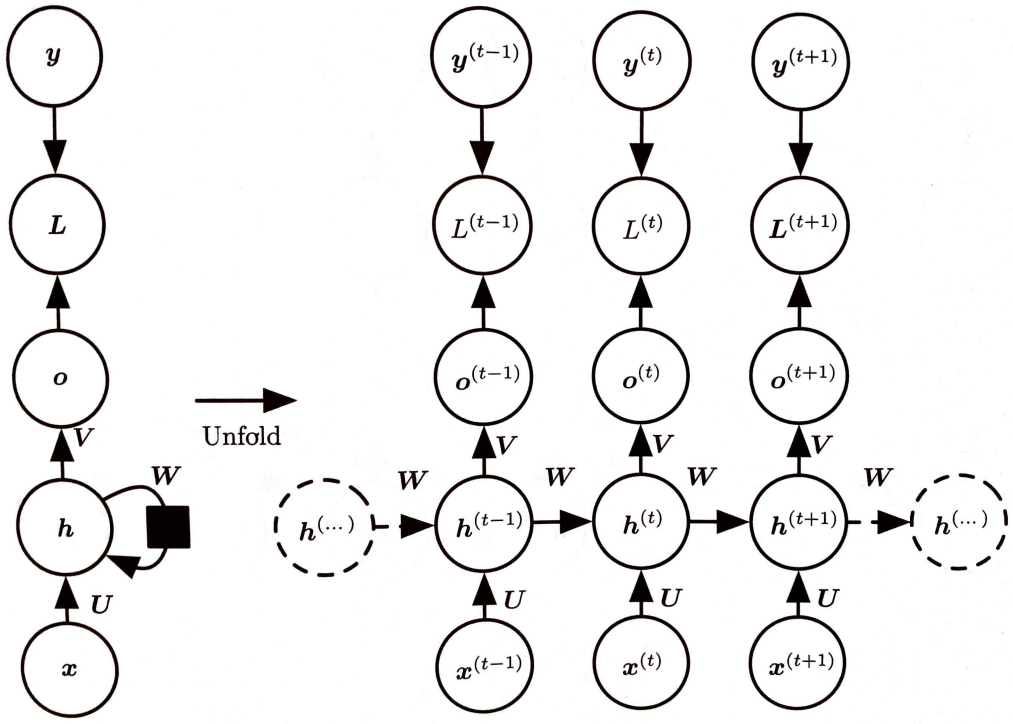

Figure A.6: A RNN architecture: *(left)* the computational graph, where the recurrent link is a loop is a loop on the hidden node; *(right)* the unfolded graph, where the time component is represented in the form of a sequence of copies of the computational graph connected by the recurrent link from the $\boldsymbol{h}^{(t)}$ node to the $\boldsymbol{h}^{(t+1)}$ node.[Image source: [33].

Let's take the RNN in Figure A.6 as an example: in this network, the recurrent connection is represented by the loop connection of the hidden node, parametrized by the weight matrix W.
Recurrence often starts from hidden units because, when the objective is to predict the future from past sequence values, $\boldsymbol{h}^{(t)}$ is used as a sort of summary of all information coming from previous input, like a short-term memory of what happened before; $\boldsymbol{h}^{(t)}$ can necessarily represent just a part of past information, because it is a finite vector: some approaches have been conceived for selecting the most relevant information for the specific task (further details about these models are described later in this section).

The RNN in figure A.6 is an example of network mapping a sequence to another of the same length: $\boldsymbol{x}^{(t)}$ nodes are the elements of the input sequence; $\boldsymbol{h}^{(t)}$ nodes are the hidden representations; $\boldsymbol{o}^{(t)}$ nodes are the output of the net-

work at time t; $\boldsymbol{L}^{(t)}$ is the loss at time t and $\boldsymbol{y}^{(t)}$ is the supervision value at time t. U is the matrix of weights between the input and the hidden unit, W is the matrix of weights between hidden units at different time steps, V is the matrix of weights between hidden units and output units.
In this class of problems, the total loss is the sum of the losses at each time step, that is

$$L(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}, \boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(T)}) = -\sum_{t} log\, p_{model}(y^{(t)} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)},) .$$

The gradient of this function is very expensive to compute: first, a forward propagation step is needed, in "time" direction, and then the backward step has to be done, in the opposite direction, and these computations cannot be parallelized because each time step can be executed only after the previous one. This means that all states computed in the forward phase need to be stored in memory for being then used in the backward phase. Both runtime and memory costs are thus $O(T)$.

A.2.3.2 Training with gradient in RNNs

Gradient descent on RRNs is quite straightforward: it is standard backpropagation applied to the unfolded computational graph. This approach is called *backpropagation through time* (BPTT), and has been introduced independently by different authors. [69] [80] [103]. This approach handles the problem of backpropagation for recursive functions: it needs to consider, for each gradient value timestep t, all previous (actually, subsequent, meaning from time $t+1$ and later) gradients and the different components by which each gradient is influenced. For each timestep, the gradient has to be computed for the loss L on weight matrices U, V, W and possible bias vectors. In other words, the unfolded RNN can be trained as a normal FNN with backpropagation, the only difference is that, when updating the weights, it must be taken into consideration that all copies of the same link must stay identical; thus, as in CNNs, all gradient updates are averaged and the result is applied as unique change to all the copies of the same connection.

Another approach for gradient learning of RNNs is *real time recurrent learning* (RTRL): as described in Section A.1.1.2, RTRL is an algorithm of forward propagation for online applications, where it is not necessary to wait until the input sequence has been completely processed to start computing gradients. Instead, RTRL computes weight updates at each timestep and propagates the gradient forward in the network.
The gradient computed by BPTT and RTRL is the same: they compute the same gradient, but their learning trajectories differ in the fact that RTRL updates the weights as soon as it encounters a target and not at the end of the sequence. Thus, the two algorithms are equivalent in the final result, but they differ in resource consumption: being n the number of units and T the sequence's length, the two algorithms' complexities are as in Table A.1.

Algorithm	Memory	Time
BPTT	$O(nT)$	$O(n^2T)$
RTRL	$O(n^3)$	$O(n^4)$

Table A.1: Comparison of time and memory complexity of BPTT and RTRL.

It can be seen that, on equal terms, BPTT is more efficient than RTRL until $T \leq n^2$, that is with shorter sequences.

A.2.3.3 Deep recurrent neural networks

A recurrent neural network in which one or more of the functions defining the system is implemented as deep neural network is called a deep RNN, which means the state function (corresponding to the connection between the input and the hidden state,), the transition function (corresponding to the loop connection on the hidden unit) and the output function (between hidden and output state). This design choice can increase the expressivity of the RNN, but can also worsen the optimization complexity, for example extending the shortest path between different elements of the input sequence. For this reason, sometimes skip connections are added: this way the expressive power is maintained, and also shortest paths are not lengthened.

A.2.3.4 Problems

One of the main problems of RNNs' training is gradient vanishing or exploding, which is the large increase or decrease of the gradient value, usually due to many multiplications, as happens in RNN training. This becomes a problem when the input sequence presents some long-term dependencies among its elements, meaning that an element at time t depends on an element at time $t-k$, with large k. Thus, the weights among hidden units, $\boldsymbol{W}$, need to be multiplied many times, and this can cause their value to either explode or vanish (the latter is the most common one).
More precisely, if the spectral radius[†] of $\boldsymbol{W}$, $\rho(\boldsymbol{W})$, is smaller than 1, then

$$\lim_{n \to \infty} \boldsymbol{W}^n = 0 ,$$

meaning that also

$$\lim_{t-k \to \infty} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-k)}} = 0 ,$$

where k is the distance between the two related elements of the sequence. With such a small gradient value, it is almost impossible to perform meaningful gradient descent.
On the other hand, if $\rho(\boldsymbol{W}) > 1$ then

$$\lim_{n \to \infty} ||\boldsymbol{W}^n|| = \infty$$

and thus

$$\lim_{t-k \to \infty} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-k)}} = \infty,$$

making it hard to decide the appropriate descent "step" size (a too large step would risk to skip entire areas of the gradient surface, possibly missing some relevant areas of it).

For these reasons, it has been necessary to develop some variants of RNNs to handle the problem of vanishing/exploding gradients while finding a solution for modeling long-term dependencies.

[†]The spectral radius is the eigenvalue with the maximum module of a matrix.

A.2.3.5 Long short-term memory

Long short-term memory (LSTM) are RNNs specifically designed to learn short-term dependencies for a large number of steps, enforcing a constant (thus, not exploding nor vanishing) error flow, given that, at a specific point, the gradient computation is not memorized anymore.[39]. A scheme of an LSTM unit is in Figure A.7.

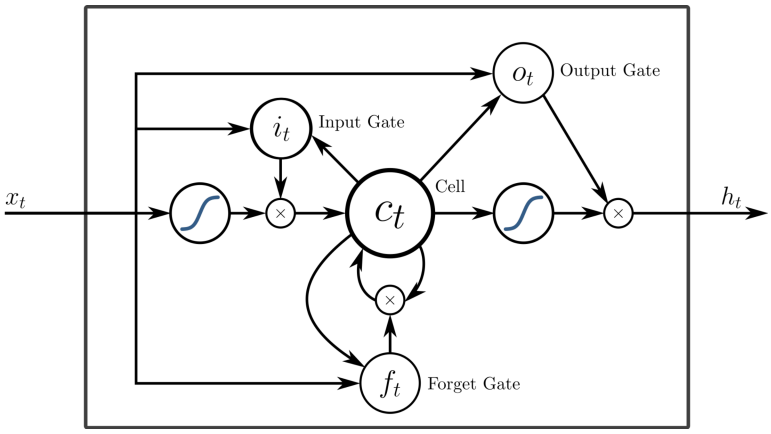

Figure A.7: A LSTM cell with peephole connections (i.e. the direct links between the cell c_t and each gate i_t, f_t and o_t).
[Image source: https://commons.wikimedia.org/wiki/File:Peephole_Long_Short-Term_Memory.svg.

In LSTM, instead of a unique hidden unit $\boldsymbol{h}$, a more articulated network handles the memory of dependencies. This structure is characterized by three gates, often implemented as sigmoid functions σ:

- input gate: determines whether the input can enter the memory cell or not: the new input goes through the sigmoid activation and can enter the memory cell only if the input gate is open;
- forget gate: determines whether the current memory content can be reset to 0 or kept: the cell content is canceled only if the forget gate is open;
- output gate: determines whether the current memory value should be read as output or not: the cell content is given as output only in the output gate is open.

All gate units have a sigmoid nonlinear activation. The linear cell can keep "in memory" information about the past for a long time, and its linearity pre-

vents vanishing gradients (even though there still may be the problem of exploding gradient).

A.2.3.6 Gated recurrent units

Gated recurrent units[15] are a simplification of LSTM models; a GRU is presented in Figure A.8.

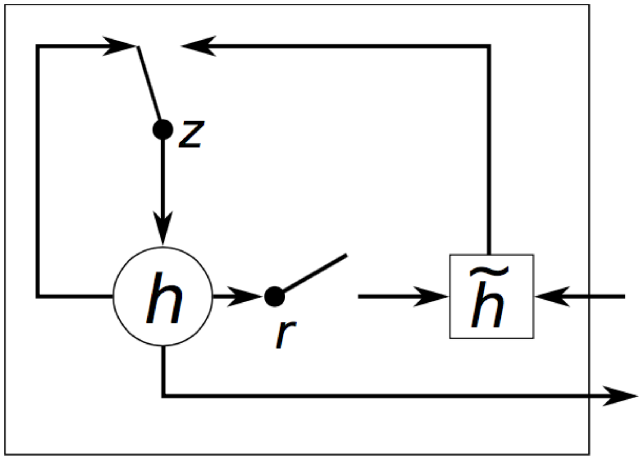

Figure A.8: A GRU cell.
[Image source: https://arxiv.org/pdf/1412.3555v1.pdf.

In GRUs there are two gates: the first one (z in Figure A.8) determines whether the current memory should be kept as it is or updated with $\tilde{h}$, the second one, r, determines whether $\tilde{h}$ will contain just the new input (which, in Figure A.8, arrives from outside the GRU cell) or also what has been stored in memory up to that moment, i.e. h.

A.2.3.7 Reservoir computing

Reservoir computing is an approach of RNN training and the main models are *echo state networks* and *liquid state machines*.
The common element of this class of methods is that they are made of a fixed-weight RNN (without output units) and another model, possibly linear, which performs the specific task of the whole model. The fixed-weights

module is called *reservoir* and it has fixed input-hidden and hidden-hidden connections, which are randomly initialized: they cannot thus be trained; the second part is called *readout* and is usually a dense classifier or linear regressor, taking in input the activations computed in the reservoir and is trainable.

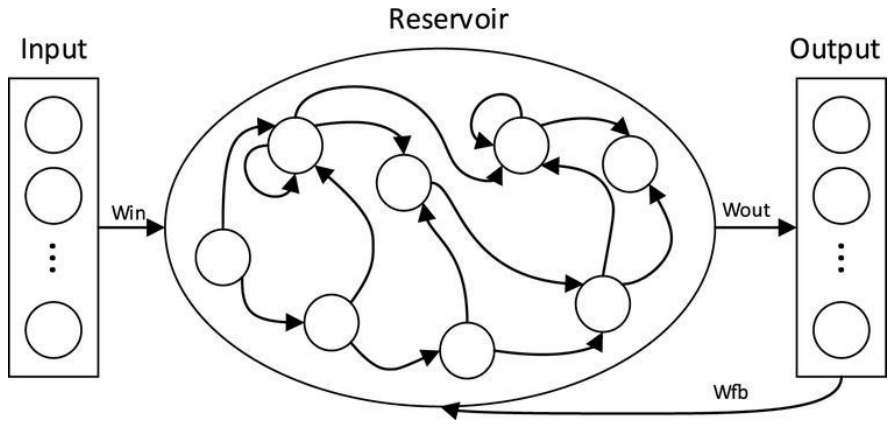

Figure A.9: A reservoir network scheme.
[Image source: https://content.iospress.com/articles/journal-of-intelligent-and-fuzzy-systems/ifs169552.

The main trait of these approaches is to have a sort of intrinsic memory effect, because the random weight initialization in the reservoir generates loops on the same unit or including many units, which end up forming a memory cell.

A.2.3.8 The *reservoir*

The first part of these models is called "reservoir" because it is closed, since it has no output connections: it receives the input and its hidden states interactions maintain a nonlinear history of the input.
To express a large set of dynamics, the reservoir should:

- be rich, meaning that it should be made of many units, to increase expressivity;
- be randomly and sparsely connected, to achieve high nonlinearity but avoid chaotic models;
- satisfy the *echo state property*, for which $\rho(\boldsymbol{W}) < 1$, meaning that the signal in the reservoir should slowly vanish with time.

Some of the most successful techniques of reservoir computing are:

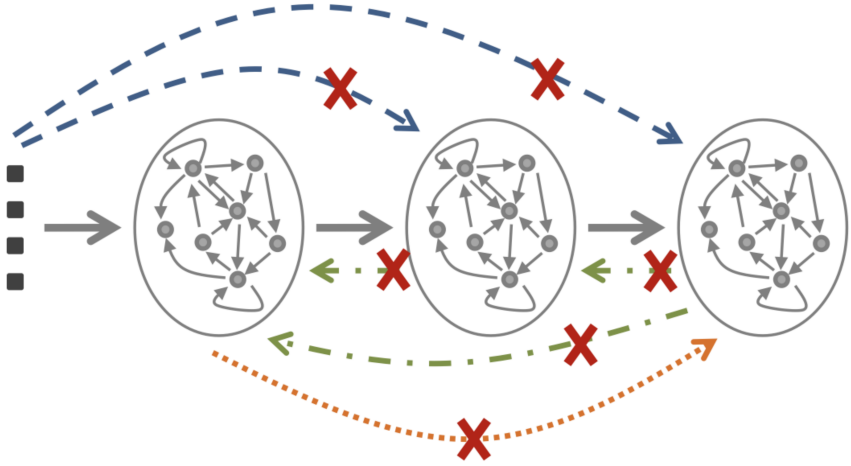

Figure A.10: A deep reservoir seen as a constrained shallow one. [Image source: [27].

- *Echo State Networks* (ESN), introduced by [43], [42], which present standard recurrent nodes in the reservoir, with leaky integrator units[‡]. These models are called "echo" networks because information propagates in the reservoir like echo does in air. Deep ESNs[28] have proved to be a valid deep model for increasing even more the set of dynamics and for modeling different time scales. In these models, the reservoir is deep, meaning that it is organized in different layers, in which the output of a layer is the input of the following one. What can be noticed is that a Deep ESN can also be seen as a constrained shallow ESN, with the same number of hidden units, just organized in an ordered layered pipeline[27], meaning that backward connections between layers are removed, only connections from input to first layer are kept and only connections between adjacent layers are allowed.

- *Liquid State Machines* (LSM)[60] use spiking neural network[59], a specific kind of neural network that more closely mimics the behaviour of neurons in human brain, with dynamic synaptic connection models. The word "liquid" derives from an analogy to dropping a stone into a pond: the falling stone will generate ripples in the liquid, just like the input of LSMs are converted into a spatio-temporal pattern of information displacement in the reservoir.

[‡]A leaky integrator is a hidden unit averaging between its value current value and its value at time $t-1$ with a parameter α indicating the weight of the two parts in the average operation. A toy example for a leaky unit is $\mu^{(t)} \leftarrow \alpha\mu^{(t-1)} + (1-\alpha)v^{(t)}$, where μ is a running average, v is a generic value and α is the parameter measuring the dependece of $\mu^{(t)}$ from $\mu^{(t-1)}$.

A.2.4 Graph networks

Graph neural networks (GNN) are deep learning models whose inputs are graphs. A graph is a structure composed of a set of elements (vertices/nodes) in which some pairs of the elements are in some sense "related"; connections between nodes are called edges, which can have different weights, in which case the graph is said "weighted". Graphs are usually defined as a tuple $G = (V, E)$ where V is the set of vertices and E is the set of edges. To a graph is associated a square adjacency matrix A of size $|V| * |V|$ whose entry $A(x, y)$ is equal to $A(y, x)$ and is set to 1 if there is an edge between nodes x and y, otherwise it is 0; if the graph is weighted, the entry contains the weight of the edge, and if the graph is oriented (meaning that edges are arrows with a source node and a target node), $A(x, y)$ and $A(y, x)$ are not necessarily equal. Graphs are almost everywhere in our world:[83] not only networks (physical or relational), but also molecules in chemistry,which have a graph-like structure; images, where adjacent pixels can be seen as nodes; texts, where words are nodes and their sequence can define a basic directed graph, but also their semantics relation can define some edges; road systems, where streets are edges and destinations are nodes; and many more. In all these examples, the graph structure encodes some relevant relational information about the data, and thus it would be useful to keep this information when processing graphs-like data, which means maintaining the complex structure of the graph, with nodes and edges.

Neural networks for structured data have their roots in [91] [90], in late '90s, but first proposals for graph neural networks arrive almost ten years later, with [62] [84]. Graph neural networks (GNN) model and operate over graphs to achieve relational reasoning and combinatorial generalization: GNNs facilitate learning relations between graph elements and the rules for composing them, and have in fact shown unprecedented generalization capabilities in the field of networking modeling and optimization[2].
The core idea of GNNs is to learn a representation of nodes in the graph conditioned on the representations of neighbouring nodes.[76] Obtaining such representation is functional to the design of predictors in two main settings:

prediction over single nodes in a (large) graph or oover the whole graph; these are the two main tasks of GNNs, together with prediction over edges.
The main issues when building a GNN are related to invariance to representation and relevance of the encoded information:[76] the former refers to the fact that the same graph can be visually represented in many different ways, to which correspond different nodes and edges orderings, but two graphs of this kind (i.e., isomorphic) should induce the same representation in the GNN; the latter requires that only structural information that is relevant to the task is learned.

A GNN is usually composed of three modules[119]:

- *propagation module*: used to propagate information between nodes so that the aggregated information is allowed to capture both feature and topological information. Usually, convolution operators and recurrent operators are used to aggregate information, while skip connections are used to gather information from historical representations of nodes and mitigate the over-smoothing problem.;
- *sampling module*: usually combined with the previous module, is needed for propagation in very large graphs;
- *pooling module*: when general or high-level representations of graphs or subgraphs are needed, pooling is used for extracting information from nodes.

In Figure A.11 is represented a general scheme for a GNN architecture.
The GNN taxonomy presented in the following sections is mainly taken from [119].

A.2.4.1 Propagation module

Propagation modules are usually distincted in *convolutional*, *recurrent* and with *skip connections*:

- *propagation via convolution*: the objective in convolutional approaches on graphs is to find a definition of convolution operation which can work on structured data; classic convolutional networks (the ones which work well on images) are not suited for graphs, due to the arbitrary size

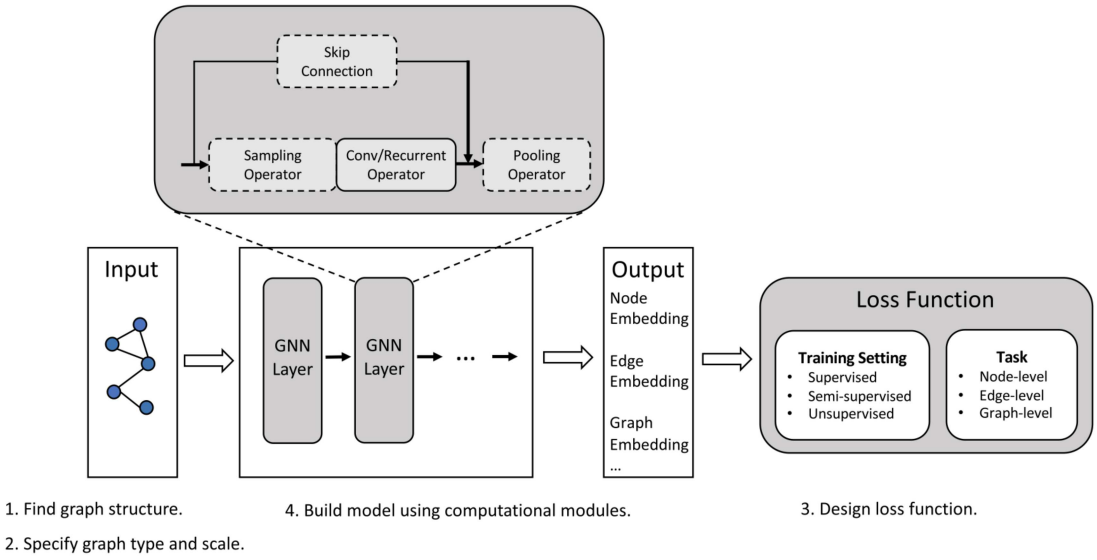

Figure A.11: A typical GNN architecture: the graph is given as input to the GNN. [Image source: [119]].

of this kind of data and their complex topology; moreover, graphs are invariant to node ordering, so the convolution operation on graph should produce the same result regardless node ordering. Approaches in this category are classified as *spectral* and *spatial*:

– *spectral convolution*: spectral approaches work on a spectral representation of the graphs and define the convolution operator in the spectral domain. The graph signal is first transformed into the spectral domain by the graph Fourier transform $\mathcal{F}$, then the convolution operation is applied. After the convolution, the resulted signal is transformed back using the inverse graph Fourier transform $\mathcal{F}^{-1}$:

$$\mathcal{F}(\boldsymbol{x}) = \boldsymbol{U}^T\boldsymbol{x}\,,\ \mathcal{F}^{-1}(\boldsymbol{x}) = \boldsymbol{U}\boldsymbol{x}\,,$$

where $\boldsymbol{U}$ is the eigenmatrix of the normalized Laplacian matrix[16] $\boldsymbol{L}$. Applying the convolution theorem,[93], convolution on the spectral domain is defined as

$$\boldsymbol{g} * \boldsymbol{x} = \mathcal{F}^{-1}(\mathcal{F}(\boldsymbol{g}) \odot \mathcal{F}(\boldsymbol{x}) = \boldsymbol{U}(\boldsymbol{U}^T\boldsymbol{g} \odot \boldsymbol{U}^T\boldsymbol{x})\,,$$

where $\boldsymbol{U}^T\boldsymbol{g}$ is the filter in the spectral domain. The filter's expression can be simplified by learning a diagonal matrix $\boldsymbol{g}_w$, obtaining

the basic expression of spectral convolutional methods

$$\boldsymbol{g} * \boldsymbol{x} = \boldsymbol{U}\boldsymbol{g}_w\boldsymbol{U}^T\boldsymbol{x} .$$

This filter expression has been implemented in many different ways, which define the different convolutional methods of this class.

– *spatial convolution*: these approaches define convolutions directly on the graph, based on its topology; the main challenge of spatial approaches is defining the convolution on neighbourhoods of different sizes and maintaining the local invariance of CNNs. Neighbourhood size is can be identified in various ways:[119] using different weight matrices for nodes with different degrees; using transition matrices; normalizing the neighbourhood; executing max-pooling on neighbours to extract the most significant features of it; by neighbours sampling and aggregation; using attention methods. Message passing is a spatial propagation technique: the first phase, aggregates messages from the neighbours of each node, then these are combined with the current state of the node to update its status:

$$m_i^{t+1} = \sum_{j \in neighbours(i)} M_t(H_j^t,\ h_i^t,\ e_{ij}),\ h_i^{t+1} = U_t(h_i^t,\ m_i^{t+1})$$

where e_{ij} are the features of the edge between nodes i ad j and M, U are the message and update functions, which can be instantiated in many different ways. Figure A.12 shows an example of message passing on a small graph.

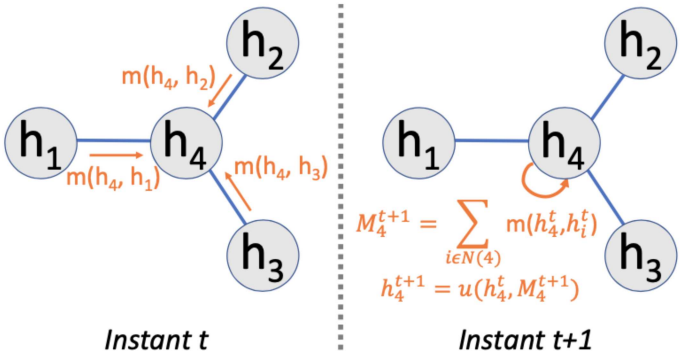

Figure A.12: Message passing example for node 4 in two time steps.: m and u are the message and update functions applied to the single nodes. [Image source: [2]].

– *attention-based spatial approaches*: differently from previous approaches, attention operators assign different weights to neighbours.For example, Graph Attention Network (GAT) utilizes a multihead attention to stabilize learning: Z different attention heads are applied separately to compute hidden states and then their features are concatenated. The feature vector of each node can be expressed as:

$$h_i^{t+1} = \sigma\left(\frac{1}{Z}\sum_{z=1}^{Z}\sum_{j\in neighbours(i)} a_{ij}^z W_z h_i^t\right),$$

where a_{ij}^z is the normalized attention coefficient of the z^{th} attention head. This approach is efficient because neighbours' individuation is parallelizable and it can be applied to graphs with variable topologies (e.g., with different node degrees).

- *propagation via recurrence*: recurrent methods in GNNs have been just recently explored (even though they were already mentioned in [90][24]), and still present some issues. The main difference between recurrent operators and convolution operators remains, as in other recurrent models, that layers with convolution use different weights while layers with recurrence share same weights. A general definition of a recurrent convergence-based GNN can be

$$\boldsymbol{H} = F(\boldsymbol{H}, \boldsymbol{X}), \boldsymbol{O} = G(\boldsymbol{H}, \boldsymbol{X}_N),$$

where $\boldsymbol{H}$, $\boldsymbol{O}$, $\boldsymbol{X}$ and $\boldsymbol{X}_N$ are matrices of stacked state representations h_v, outputs o_v, features x_v and node features x_{nv}, and F, G are the global transition and output functions, resulting from stacking the transition and output functions of all graph's nodes. There are also several works that use gate mechanism like GRU and LSTM in the propagation step to solve some computational limitations and improve long-term propagation; however, they are not guaranteed to converge but run for a fixed number of training steps.

- *propagation with skip connections*: many experiments have proved that, differently from what could be expected, too deep GNN models do not perform better than more shallow ones: in fact, they seem to work worse. This probably happens because many layers also propagate noisy information from an exponentially increasing number of neighbours for each node; additionally, higher depth also causes over smoothing,because

nodes tend to have similar representations after the aggregation, when models get deeper. For these reasons, many methods have tried to add skip connections between layers to make GNN models deeper. without actually adding too many layers.

A.2.4.2 Sampling module

At each layer, nodes in GNNs aggregate messages from their neighbours in the previous layer; this may cause the size of the supporting neighbourhood to explode with depth. Moreover, when the graph is large, sometimes it is not possible to store and process all information coming from neighbours. To deal with this problem, some sampling techniques have been introduced: *node sampling*, *layer sampling* and *subgraph sampling*.

- *node sampling*: selects a subset of nodes from a node's neighbourhood, to reduce its size;
- *layer sampling*: selects a small set of nodes (with their whole neighbourhood) for aggregation in each layer to limit the size explosion; it can be parametrized so that the number of nodes to retain is learned by the model;
- *subgraph sampling*: working on a larger scale, it samples a small number of subgraphs of the main graph and restricts the neighbourhood search only to these portions of the graph.

A.2.4.3 Pooling module

Like in CNN a convolutional layer is usually followed by a pooling layer to obtain more general features, in GNNs this happens too, especially on large graphs. Pooling operations have also to deal with possible hierarchical graph structures, which are relevant in case of classification tasks.
Thus, pooling modules can be:

- direct: learn graph-level representations directly from nodes, using different node selection strategies: maximum, mean, sum, attention, ordering; they are also called readout functions. These methods directly learn graph representations from nodes, without investigating the hierarchical property of the graph structure;

- hierarchical: learn graph representations by layers, keeping the hierarchical structure of the graph.

A.2.5 Autoencoders

Autoencoders are (usually) deep feedforward neural networks which aim to copy the input into the output, and the central hidden layer $\mathbf{h}$ describes a representation of the input, which is the basis for reconstructing it into the output.
The structure of an autoencoder is symmetrical with respect to the central hidden layer: the first half of the architecture, from the input layer to the central hidden one, is called "encoder"; the second half, from the central hidden layer to the output one, the "decoder". The encoder can be seen as a deep network learning the function f such that $\mathbf{h} = f(\mathbf{x})$, while the decoder learns the function g such that $\mathbf{o} = g(\mathbf{h})$.
A deep autoencoder architecture whit 3 hidden layers is shown in Figure A.13.

However, autoencoders should not perfectly learn to copy the input into the output for each data point (that would mean learning f, g such that $g(f(\mathbf{x}) = \mathbf{x}$) because this would be quite useless; instead, the main objective when using autoencoders is to obtain a (compact) representation of the input, thus the network should be able also to generalize over unseen data. For this reason, autoencoders are designed such that they do not perfectly learn to copy the input, but to copy the input only approximatively or with some restrictions: in this way, the autoencoder will focus on the most relevant properties of the input data, generating an expressive internal representation $\mathbf{h}$.

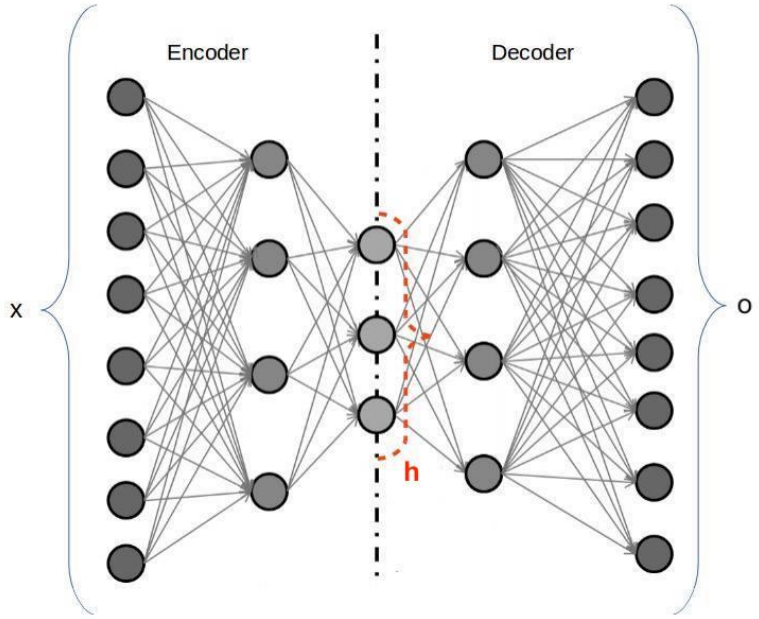

Figure A.13: A deep autoencoder architecture: the input $\mathbf{x}$ is used to produce the hidden representation $\mathbf{h}$, from which the output $\mathbf{o}$ is produced as a reconstruction of $\mathbf{x}$.
[Image source: https://www.researchgate.net/publication/317559243_Deep_Autoencoder_Based_Speech_Features_for_Improved_Dysarthric_Speech_Recognition].

A.2.5.1 Hidden layers size

Based on the size of their hidden layers, and thus also of $\mathbf{x}$, AEs can be defined *overcomplete* or *undercomplete*:

- *overcomplete* AEs have hidden layers larger than the input (and output) one. This kind of architecture allows to learn a great number of features, but there is the potential risk that it learns the identity function, thus becoming useless; to prevent this, *regularized autoencoders* use a loss function that encourages the network to prevent from just copying the input into its output;

- *undercompelte* AEs have hidden layers smaller that the input (and output) one. Having to compress the input in a smaller representation forces network to capture only the most important features.

In Figure A.14 deep overcomplete and undercomplete autoencoders are represented.

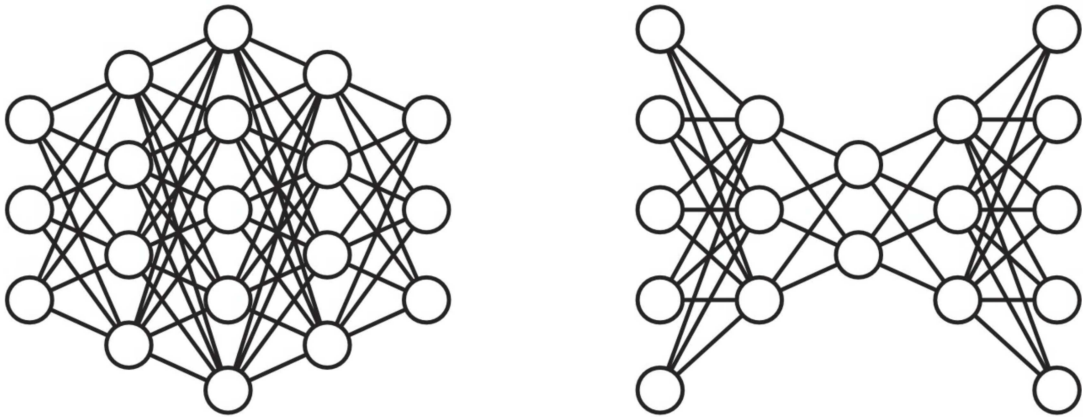

Figure A.14: On the *left*, a deep overcomplete autoencoder; on the *right*, a deep undercomplete autoencoder.

A.2.5.2 Stochastic autoencoders

Modern autoencoders have generalized beyond deterministic functions to stochastic mappings $p_{encoder}(\mathbf{h}|\mathbf{x})$ and $p_{decoder}(\mathbf{x}|\mathbf{h})$ (see Figure A.15).

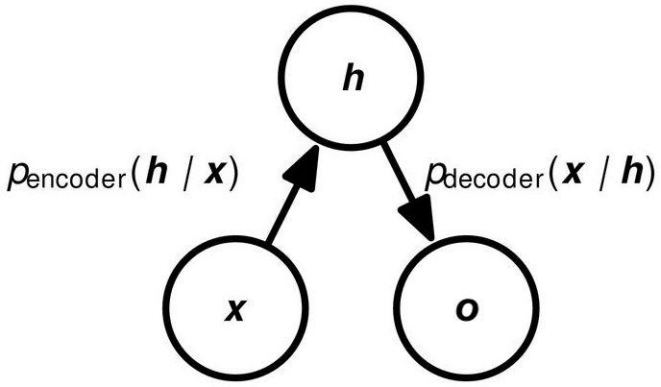

Figure A.15: In a stochastic autoencoder, both the encoder and the decoder have output samples from two distributions, $p_{encoder}(\mathbf{h}|\mathbf{x})$ and $p_{decoder}(\mathbf{x}|\mathbf{h})$ respectively [Image source: [33]].

As in other feedforward networks, the loss function for the autoencoder is the negative log-likelihood, in this case $-\ logp_{decoder}(\mathbf{x}|\mathbf{h})$, because the input $\mathbf{x}$ is also the output of the autoencoder.

A.2.5.3 Other types of autoencoders

Regularized autoencoders Both in the case of overcomplete and undercomplete AEs the model can be given enough capacity to learn the identity function on training data, which is not the objective of autoencoders. One way to deal with this issue is to add regularizing elements to the loss function of the model, in fact limiting its capacity and moving the focus also on other properties the representation should have. This way, the model is limited by software means, not "physically".

Sparse autoencoders In a sparse autoencoder a sparsity penalty $\Omega(\mathbf{h})$ on the internal representation $\mathbf{h}$ is added to the main loss:

$$L(\mathbf{x}, g(f(\mathbf{x}))) + \Omega(\mathbf{h}) .$$

The sparsity penalty is generally used for encouraging the model to learn additional features of data specific of other tasks.

Denoising autoencoders A denoising autoencoder receives corrupted input data and should predict the original, uncorrupted data point.

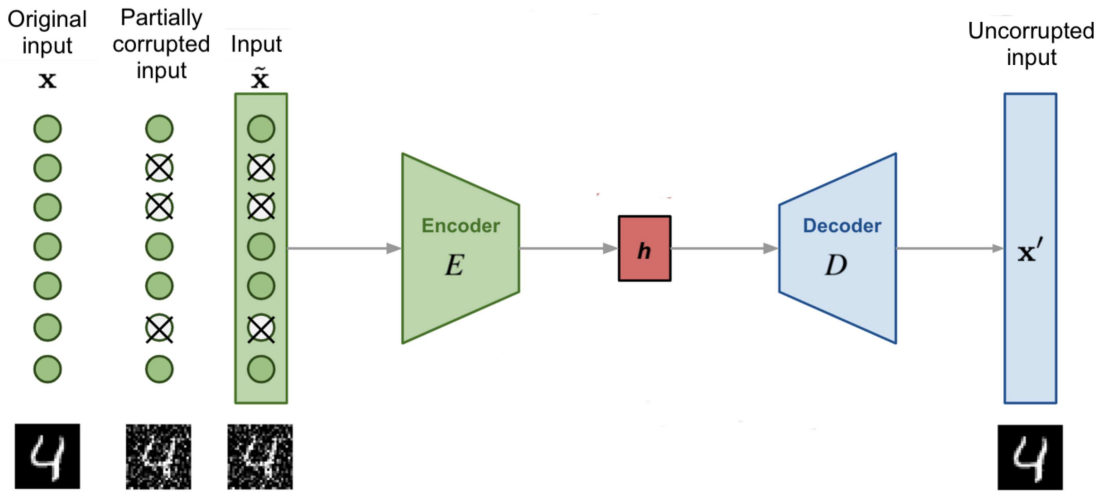

Figure A.16: A denoising autoencoder on image input: the original input is corrupted and then given to the encoder as input; the decoder should be able to generate as output an image very close to the uncorrupted original input.[Image source: https://towardsdatascience.com/autoencoders-and-the-denoising-feature-from-theory-to-practice-db7f7ad8fc78.

What happens on the lefthand part of Figure A.16 is called corruption process, $C(\tilde{\mathbf{x}}|\mathbf{x})$, which is a conditional distribution of corrupted samples $\tilde{\mathbf{x}}$ over original data $\mathbf{x}$. The autoencoder now learns a reconstruction distribution on pairs $(\mathbf{x},\ \tilde{\mathbf{x}})$:

1. sample a training data point $\mathbf{x}$;
2. sample the corresponding corrupted $\tilde{\mathbf{x}}$;
3. use $(\mathbf{x},\ \tilde{\mathbf{x}})$ as training example for the autoencoder for estimating

$$p_{reconstruction}(\mathbf{x},\ \tilde{\mathbf{x}}) \ = \ p_{decoder}(\mathbf{x},\ \mathbf{h}) \ ,$$

   with $f(\tilde{\mathbf{x}}) \ = \ \mathbf{h}$.

With denoising autoencoders, the model is forced to be able to deal with corrupted versions of the same kind of data: this makes the model more resistant to noisy data, which are really common in real-life scenarios, besides avoiding the issue of learning the identity function.

CONTRACTIVE AUTOENCODERS Contractive autoencoders are a kind of regularized AEs where the regularizing element is a sparse penalty, like in sparse AEs, but applied to the derivative of $\mathbf{h}$. The general form of the loss thus is:

$$L(\mathbf{x},\ g(f(\mathbf{h}))\ +\ \Omega(\mathbf{h},\mathbf{x})\ =\ L(\mathbf{x},\ g(f(\mathbf{h}))\ +\lambda\left\|\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}\right\|^2_F \ ,$$

where F is the Frobenius norm[§] of the Jacobian matrix of partial derivatives associated to the encoder function f. This norm forces the derivatives of f to be as small as possible, meaning that f should not change much when $\mathbf{x}$ changes slightly.
The name *contractive* comes from the effect of this penalty on the neighbourhood: it "contracts" a larger neighbourhood into a smaller one, thus locally avoiding great changes of value in the same neighbourhood.

[§]i.e. sum of squared elements, also known as Hilbert-Schmidt norm.

One of the main issues of this technique lies in the fact that it works on the Jacobian matrix (matrix of partial derivatives), which can dangerously grow when the autoencoder is deeper than a single-layer autoencoder. This problem can be faced by separately training a contractive autoencoder for each layer, and then composing them: the result would be contractive as well, even though it will not be the same as training a contractive autoencoder on the whole architecture.

Bibliography

[1] N. Adaloglou and S. Karagiannakos. How attention works in deep learning: understanding the attention mechanism in sequence models. *https://theaisummer.com/*, 2020. URL https://theaisummer.com/attention/.

[2] P. Almasan, J. Suárez-Varela, A. Badia-Sampera, K. Rusek, P. Barlet-Ros, and A. Cabellos-Aparicio. Deep reinforcement learning meets graph neural networks: An optical network routing use case. *CoRR*, abs/1910.07421, 2019. URL http://arxiv.org/abs/1910.07421.

[3] E. Alpaydin. *Introduction to Machine Learning*. The MIT Press, fourth edition, 2020.

[4] P. P. Angelov. *Handbook on Computer Learning and Intelligence*. World Scientific, 2022. doi: 10.1142/12498. URL https://www.worldscientific.com/doi/abs/10.1142/12498.

[5] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization, 2016. URL https://arxiv.org/abs/1607.06450.

[6] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. TB, A. Muldal, N. Heess, and T. P. Lillicrap. Distributed distributional deterministic policy gradients. *CoRR*, abs/1804.08617, 2018. URL http://arxiv.org/abs/1804.08617.

[7] B. Behzadian and M. Petrik. Low-rank feature selection for reinforcement learning. In *ISAIM*, 2018.

[8] M. G. Bellemare, W. Dabney, R. Dadashi, A. A. Taiga, P. S. Castro, N. L. Roux, D. Schuurmans, T. Lattimore, and C. Lyle. A geometric perspective on optimal representations for reinforcement learning, 2019. URL https://arxiv.org/abs/1901.11530.

[9] W. Böhmer, J. T. Springenberg, J. Boedecker, M. Riedmiller, and K. Obermayer. Autonomous learning of state representations for control: An emerging field aims to autonomously learn state representations for reinforcement learning agents from their real-world sensor observations. *KI - Künstliche Intelligenz*, 29(4):353–362, 2015. doi: 10.1007/s13218-015-0356-1. URL https://doi.org/10.1007/s13218-015-0356-1.

[10] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko. Translating embeddings for modeling multi-relational data. In C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013. URL https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf.

[11] P. S. Castro. Scalable methods for computing state similarity in deterministic markov decision processes. *CoRR*, abs/1911.09291, 2019. URL http://arxiv.org/abs/1911.09291.

[12] P. S. Castro. Scalable methods for computing state similarity in deterministic markov decision processes. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(06):10069–10076, Apr. 2020. doi: 10.1609/aaai.v34i06.6564. URL https://ojs.aaai.org/index.php/AAAI/article/view/6564.

[13] P. S. Castro, T. Kastner, P. Panangaden, and M. Rowland. Mico: Learning improved representations via sampling-based state similarity for markov decision processes. *CoRR*, abs/2106.08229, 2021. URL https://arxiv.org/abs/2106.08229.

[14] Y. Chen, C. Dong, P. Palanisamy, P. Mudalige, K. Muelling, and J. M. Dolan. Attention-based hierarchical deep reinforcement learning for lane change behaviors in autonomous driving. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 1326–1334, 2019. doi: 10.1109/CVPRW.2019.00172.

[15] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, abs/1409.1259, 2014. URL http://arxiv.org/abs/1409.1259.

[16] F. Chung and C. B. of the Mathematical Sciences. *Spectral Graph Theory*. Conference Board of Mathematical Sciences. American Mathematical Society, 1997. ISBN 9780821803158. URL https://books.google.it/books?id=4IK8DgAAQBAJ.

[17] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (elus), 2015. URL https://arxiv.org/abs/1511.07289.

[18] W. Dabney, G. Ostrovski, D. Silver, and R. Munos. Implicit quantile networks for distributional reinforcement learning. *CoRR*, abs/1806.06923, 2018. URL http://arxiv.org/abs/1806.06923.

[19] W. Dabney, A. Barreto, M. Rowland, R. Dadashi, J. Quan, M. G. Bellemare, and D. Silver. The value-improvement path: Towards better representations for reinforcement learning, 2020. URL https://arxiv.org/abs/2006.02243.

[20] P. Dayan. Improving generalization for temporal difference learning: The successor representation. 5(4):613–624, 1993. doi: 10.1162/neco.1993.5.4.613.

[21] T. de Bruin, J. Kober, K. Tuyls, and R. Babuška. Integrating state representation learning into deep reinforcement learning. *IEEE Robotics and Automation Letters*, 3(3):1394–1401, 2018. doi: 10.1109/LRA.2018.2800101.

[22] N. Deshpande and A. Spalanzani. Deep reinforcement learning based vehicle navigation amongst pedestrians using a grid-based state representation. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 2081–2086, 2019. doi: 10.1109/ITSC.2019.8917299.

[23] N. Ferns, P. Panangaden, and D. Precup. Bisimulation metrics for continuous markov decision processes. *SIAM Journal on Computing*, 40(6): 1662–1714, 2011. doi: 10.1137/10080484X. URL https://doi.org/10.1137/10080484X.

[24] P. Frasconi, M. Gori, and A. Sperduti. A general framework for adaptive processing of data structures. *IEEE Transactions on Neural Networks*, 9(5):768–786, 1998. doi: 10.1109/72.712151.

[25] S. Fujimoto, H. van Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018. URL http://arxiv.org/abs/1802.09477.

[26] R. Furuta, N. Inoue, and T. Yamasaki. Fully convolutional network with multi-step reinforcement learning for image processing. *CoRR*, abs/1811.04323, 2018. URL http://arxiv.org/abs/1811.04323.

[27] C. Gallicchio and A. Micheli. Deep echo state network (deepesn): A brief survey. *CoRR*, abs/1712.04323, 2017. URL http://arxiv.org/abs/1712.04323.

[28] C. Gallicchio, A. Micheli, and L. Pedrelli. Deep reservoir computing: A critical experimental analysis. *Neurocomputing*, 268:87–99, 2017. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2016.12.089. URL https://www.sciencedirect.com/science/article/pii/S0925231217307567. Advances in artificial neural networks, machine learning and computational intelligence.

[29] C. Gelada, S. Kumar, J. Buckman, O. Nachum, and M. G. Bellemare. DeepMDP: Learning continuous latent space models for representation learning. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2170–2179. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/gelada19a.html.

[30] D. Ghosh and M. G. Bellemare. Representations for stable off-policy reinforcement learning, 2020. URL https://arxiv.org/abs/2007.05520.

[31] P. Ghosh, M. S. M. Sajjadi, A. Vergari, M. J. Black, and B. Schölkopf. From variational to deterministic autoencoders. *CoRR*, abs/1903.12436, 2019. URL http://arxiv.org/abs/1903.12436.

[32] R. Givan, T. Dean, and M. Greig. Equivalence notions and model minimization in markov decision processes. *Artificial Intelligence*, 147(1):163–223, 2003. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(02)00376-4. URL https://www.sciencedirect.com/science/article/pii/S0004370202003764. Planning with Uncertainty and Incomplete Information.

[33] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. The MIT Press, 2016.

[34] S. Gu, E. Holly, T. P. Lillicrap, and S. Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. *CoRR*, abs/1610.00633, 2016. URL http://arxiv.org/abs/1610.00633.

[35] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018. URL https://arxiv.org/abs/1801.01290.

[36] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. *CoRR*, abs/1706.02216, 2017. URL http://arxiv.org/abs/1706.02216.

[37] P. Hart and A. C. Knoll. Graph neural networks and reinforcement learning for behavior generation in semantic environments. *CoRR*, abs/2006.12576, 2020. URL https://arxiv.org/abs/2006.12576.

[38] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver. Rainbow:

Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017. URL http://arxiv.org/abs/1710.02298.

[39] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735.

[40] L. Itti and C. Koch. Computational modelling of visual attention. *Nature Reviews Neuroscience*, 2(3):194–203, 2001. doi: 10.1038/35058500. URL https://doi.org/10.1038/35058500.

[41] S. Ivanov and A. D'yakonov. Modern deep reinforcement learning algorithms. *CoRR*, abs/1906.10025, 2019. URL http://arxiv.org/abs/1906.10025.

[42] H. Jaeger. Echo state network. *Scholarpedia*, 2(9):2330, 2007. doi: 10.4249/scholarpedia.2330. revision #196567.

[43] H. Jaeger and H. Haas. Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication. *Science*, 304 (5667):78–80, Apr. 2004. doi: 10.1126/science.1091277.

[44] R. Jonschkowski and O. Brock. Learning state representations with robotic priors. *Autonomous Robots*, 39:407–428, 10 2015. doi: 10.1007/s10514-015-9459-7.

[45] S. Kapturowski, G. Ostrovski, W. Dabney, J. Quan, and R. Munos. Recurrent experience replay in distributed reinforcement learning. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=r1lyTjAqYX.

[46] M. Khosla, J. Leonhardt, W. Nejdl, and A. Anand. Node representation learning for directed graphs. *CoRR*, abs/1810.09176, 2018. URL http://arxiv.org/abs/1810.09176.

[47] G. Konidaris, S. Osentoski, and P. Thomas. Value function approximation in reinforcement learning using the fourier basis. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, AAAI'11, page 380–385. AAAI Press, 2011.

[48] I. Kostrikov, D. Yarats, and R. Fergus. Image augmentation is all you need: Regularizing deep reinforcement learning from pixels. *CoRR*, abs/2004.13649, 2020. URL https://arxiv.org/abs/2004.13649.

[49] C. L. Lan, S. Tu, A. Oberman, R. Agarwal, and M. G. Bellemare. On the generalization of representations in reinforcement learning, 2022. URL https://arxiv.org/abs/2203.00543.

[50] H. Larochelle and G. E. Hinton. Learning to combine foveal glimpses with a third-order boltzmann machine. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010. URL https://proceedings.neurips.cc/paper/2010/file/677e09724f0e2df9b6c000b75b5da10d-Paper.pdf.

[51] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing (preliminary report). In *POPL '89*, 1989.

[52] T. N. Larsen, H. □. Teigen, T. Laache, D. Varagnolo, and A. Rasheed. Comparing deep reinforcement learning algorithms' ability to safely navigate challenging waters. *Frontiers in Robotics and AI*, 8, 2021. ISSN 2296-9144. doi: 10.3389/frobt.2021.738113. URL https://www.frontiersin.org/articles/10.3389/frobt.2021.738113.

[53] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

[54] K. Lee, I. Fischer, A. Z. Liu, Y. Guo, H. Lee, J. F. Canny, and S. Guadarrama. Predictive information accelerates learning in RL. *CoRR*, abs/2007.12401, 2020. URL https://arxiv.org/abs/2007.12401.

[55] M. Lee. Tactor-critic methods, 2005. URL http://www.incompleteideas.net/book/ebook/node66.html.

[56] T. Lesort, N. Díaz-Rodríguez, J.-F. Goudou, and D. Filliat. State representation learning for control: An overview. *Neural Networks*, 108: 379–392, dec 2018. doi: 10.1016/j.neunet.2018.07.006. URL https://doi.org/10.1016%2Fj.neunet.2018.07.006.

[57] T. Lillicrap, J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *CoRR*, 2015. URL https://arxiv.org/abs/1509.02971v5.

[58] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982. doi: 10.1109/TIT.1982.1056489.

[59] W. Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659–1671, 1997. ISSN 0893-6080. doi: https://doi.org/10.1016/S0893-6080(97)00011-7. URL https://www.sciencedirect.com/science/article/pii/S0893608097000117.

[60] W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, 14(11):2531–2560, 2002.

[61] S. Mahadevan and M. Maggioni. Proto-value functions: A laplacian framework for learning representation and control in markov decision processes. *Journal of Machine Learning Research*, 8(74):2169–2231, 2007. URL http://jmlr.org/papers/v8/mahadevan07a.html.

[62] A. Micheli. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, 2009. doi: 10.1109/TNN.2008.2010350.

[63] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space, 2013. URL https://arxiv.org/abs/1301.3781.

[64] T. Mikolov, E. Grave, P. Bojanowski, C. Puhrsch, and A. Joulin. Advances in pre-training distributed word representations. *CoRR*, abs/1712.09405, 2017. URL http://arxiv.org/abs/1712.09405.

[65] T. M. Mitchell. *Machine Learning*. McGraw Hill education international edition, 1997.

[66] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, 2013. URL http://arxiv.org/abs/1312.5602.

[67] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015. doi: 10.1038/nature14236.

[68] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, 2016. URL http://arxiv.org/abs/1602.01783.

[69] M. Mozer. A focused backpropagation algorithm for temporal pattern recognition. *Complex Systems*, 3, 01 1995.

[70] T. Mu, K. Lin, F. Niu, and G. Thattai. Learning two-step hybrid policy for graph-based interpretable reinforcement learning. *CoRR*, abs/2201.08520, 2022. URL https://arxiv.org/abs/2201.08520.

[71] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA, 2010. Omnipress. ISBN 9781605589077.

[72] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning state representations for query optimization with deep reinforcement learning. *CoRR*, abs/1803.08604, 2018. URL http://arxiv.org/abs/1803.08604.

[73] I. Osband, C. Blundell, A. Pritzel, and B. V. Roy. Deep exploration via bootstrapped DQN. *CoRR*, abs/1602.04621, 2016. URL http://arxiv.org/abs/1602.04621.

[74] K. Ota, T. Oiki, D. K. Jha, T. Mariyama, and D. Nikovski. Can increasing input dimensionality improve deep reinforcement learning? *CoRR*, abs/2003.01629, 2020. URL https://arxiv.org/abs/2003.01629.

[75] M. Ou, P. Cui, J. Pei, Z. Zhang, and W. Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 1105–1114, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342322. doi: 10.1145/2939672.2939751. URL https://doi.org/10.1145/2939672.2939751.

[76] L. Pasa, N. Navarin, and A. Sperduti. *Deep Learning for Graph-Structured Data*, chapter Chapter 14, pages 585–617. doi: 10.1142/9789811247323_0014. URL https://www.worldscientific.com/doi/abs/10.1142/9789811247323_0014.

[77] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. *CoRR*, abs/1403.6652, 2014. URL http://arxiv.org/abs/1403.6652.

[78] M. Petrik. An analysis of laplacian methods for value function approximation in mdps. pages 2574–2579, 01 2007.

[79] B. Ratitch and D. Precup. Sparse distributed memories for on-line value-based reinforcement learning. In J.-F. Boulicaut, F. Esposito, F. Giannotti, and D. Pedreschi, editors, *Machine Learning: ECML 2004*, pages 347–358, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-30115-8.

[80] A. J. Robinson and F. Fallside. The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Engineering Department, Cambridge University, Cambridge, UK, 1987.

[81] A. J. Robinson and F. Fallside. The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Cambridge Univ., 1987.

[82] G. Salha, R. Hennequin, and M. Vazirgiannis. Keep it simple: Graph autoencoders without graph convolutional networks. *CoRR*, abs/1910.00942, 2019. URL http://arxiv.org/abs/1910.00942.

[83] B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. B. Wiltschko. A gentle introduction to graph neural networks. *Distill*, 2021. doi: 10.23915/distill.00033. https://distill.pub/2021/gnn-intro.

[84] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605.

[85] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay, 2015. URL https://arxiv.org/abs/1511.05952.

[86] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust region policy optimization. *CoRR*, 2015. URL http://arxiv.org/abs/1502.05477.

[87] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, 2017. URL http://arxiv.org/abs/1707.06347.

[88] X. Shen, C. Yin, and X. Hou. Self-attention for deep reinforcement learning. page 71–75, 2019. doi: 10.1145/3325730.3325743. URL https://doi.org/10.1145/3325730.3325743.

[89] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page I–387–I–395. JMLR.org, 2014.

[90] A. Sperduti and A. Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8(3):714–735, 1997. doi: 10.1109/72.572108.

[91] A. Sperduti, D. Majidi, and A. Starita. Extended cascade-correlation for syntactic and structural pattern recognition. In P. Perner, P. Wang, and A. Rosenfeld, editors, *Advances in Structural and Syntactical Pattern Recognition*, pages 90–99, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-70631-1.

[92] K. L. Stachenfeld, M. Botvinick, and S. J. Gershman. Design principles of the hippocampal cognitive map. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014. URL https://proceedings.neurips.cc/paper/2014/file/dfd7468ac613286cdbb40872c8ef3b06-Paper.pdf.

[93] M. Stéphane. Chapter 2 - the fourier kingdom. In M. Stéphane, editor, *A Wavelet Tour of Signal Processing (Third Edition)*, pages 33–57. Academic Press, Boston, third edition edition, 2009. ISBN 978-0-12-374370-1. doi: https://doi.org/10.1016/B978-0-12-374370-1.00006-9. URL https://www.sciencedirect.com/science/article/pii/B9780123743701000069.

[94] R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In D. Touretzky, M. Mozer, and M. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8. MIT Press, 1995. URL https://proceedings.neurips.cc/paper/1995/file/8f1d43620bc6bb580df6e80b0dc05c48-Paper.pdf.

[95] Y. Tassa, Y. Doron, A. Muldal, T. Erez, Y. Li, D. de Las Casas, D. Budden, A. Abdolmaleki, J. Merel, A. Lefrancq, T. P. Lillicrap, and M. A. Riedmiller. Deepmind control suite. *CoRR*, abs/1801.00690, 2018. URL http://arxiv.org/abs/1801.00690.

[96] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL http://arxiv.org/abs/1509.06461.

[97] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

[98] Q. Wang, Y. Hao, and J. Cao. Adrl: An attention-based deep reinforcement learning framework for knowledge graph reasoning. *Knowledge-Based Systems*, 197:105910, 2020. ISSN 0950-7051. doi: https://doi.org/10.1016/j.knosys.2020.105910. URL https://www.sciencedirect.com/science/article/pii/S0950705120302525.

[99] T. Wang, R. Liao, J. Ba, and S. Fidler. Nervenet: Learning structured policy with graph neural networks. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=S1sqHMZCb.

[100] Z. Wang, J. Zhang, J. Feng, and Z. Chen. Knowledge graph embedding by translating on hyperplanes. *Proceedings of the AAAI Conference on Artificial Intelligence*, 28(1), Jun. 2014. doi: 10.1609/aaai.v28i1.8870. URL https://ojs.aaai.org/index.php/AAAI/article/view/8870.

[101] Z. Wang, N. de Freitas, and M. Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015. URL http://arxiv.org/abs/1511.06581.

[102] V. Waradpande, D. Kudenko, and M. Khosla. Deep reinforcement learning with graph-based state representations. *CoRR*, abs/2004.13965, 2020. URL https://arxiv.org/abs/2004.13965.

[103] P. J. Werbos. Generalization of backpropagation with application to a recurrent gas market model, Jan. 1988. URL https://doi.org/10.1016/0893-6080(88)90007-x.

[104] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989. doi: 10.1162/neco.1989.1.2.270.

[105] P. Wolf, K. Kurzer, T. Wingert, F. Kuhnt, and J. M. Zöllner. Adaptive behavior generation for autonomous driving using deep reinforcement learning with compact semantic states. *CoRR*, abs/1809.03214, 2018. URL http://arxiv.org/abs/1809.03214.

[106] W. Xiong, T. Hoang, and W. Y. Wang. Deeppath: A reinforcement learning method for knowledge graph reasoning. *CoRR*, abs/1707.06690, 2017. URL http://arxiv.org/abs/1707.06690.

[107] Y. Xue, D. Kudenko, and M. Khosla. Graph learning based generation of abstractions for reinforcement learning.

[108] D. Yarats, A. Zhang, I. Kostrikov, B. Amos, J. Pineau, and R. Fergus. Improving sample efficiency in model-free reinforcement learning from images. *CoRR*, abs/1910.01741, 2019. URL http://arxiv.org/abs/1910.01741.

[109] D. Yarats, R. Fergus, A. Lazaric, and L. Pinto. Reinforcement learning with prototypical representations. *CoRR*, abs/2102.11271, 2021. URL https://arxiv.org/abs/2102.11271.

[110] Z. Yu, W. Liu, X. Liu, and G. Wang. Drag-jdec: A deep reinforcement learning and graph neural network-based job dispatching model in edge computing. In *29th IEEE/ACM International Symposium on Quality of Service, IWQOS 2021, Tokyo, Japan, June 25-28, 2021*, pages 1–10. IEEE, 2021. ISBN 978-1-6654-1494-4. doi: 10.1109/IWQOS52092.2021.9521327. URL https://doi.org/10.1109/IWQOS52092.2021.9521327.

[111] V. Zambaldi, D. Raposo, A. Santoro, V. Bapst, Y. Li, I. Babuschkin, K. Tuyls, D. Reichert, T. Lillicrap, E. Lockhart, M. Shanahan, V. Langston, R. Pascanu, M. Botvinick, O. Vinyals, and P. Battaglia. Deep reinforcement learning with relational inductive biases. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=HkxaFoC9KQ.

[112] H. Zang, X. Li, and M. Wang. Simsr: Simple distance-based state representation for deep reinforcement learning. *CoRR*, abs/2112.15303, 2021. URL https://arxiv.org/abs/2112.15303.

[113] A. Zhang, R. McAllister, R. Calandra, Y. Gal, and S. Levine. Learning invariant representations for reinforcement learning without reconstruction. *CoRR*, abs/2006.10742, 2020. URL https://arxiv.org/abs/2006.10742.

[114] A. Zhang, R. McAllister, R. Calandra, Y. Gal, and S. Levine. Learning invariant representations for reinforcement learning without reconstruction. *CoRR*, abs/2006.10742, 2020. URL https://arxiv.org/abs/2006.10742.

[115] R. Zhang, J. Zhu, Z. Zha, J. Dauwels, and B. Wen. R3l: Connecting deep reinforcement learning to recurrent neural networks for image denoising via residual recovery, 2021. URL https://arxiv.org/abs/2107.05318.

[116] X. Zhao, L. Zhang, Z. Ding, L. Xia, J. Tang, and D. Yin. Recommendations with negative feedback via pairwise deep reinforcement learning. *CoRR*, abs/1802.06501, 2018. URL http://arxiv.org/abs/1802.06501.

[117] G. Zheng, F. Zhang, Z. Zheng, Y. Xiang, N. J. Yuan, X. Xie, and Z. Li. Drn: A deep reinforcement learning framework for news recommendation. Republic and Canton of Geneva, CHE, 2018. International World Wide Web Conferences Steering Committee. ISBN 9781450356398. doi: 10.1145/3178876.3185994. URL https://doi.org/10.1145/3178876.3185994.

[118] C. Zhou, Y. Liu, X. Liu, Z. Liu, and J. Gao. Scalable graph embedding for asymmetric proximity. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1), Feb. 2017. doi: 10.1609/aaai.v31i1.10878. URL https://ojs.aaai.org/index.php/AAAI/article/view/10878.

[119] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun. Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434, 2018. URL http://arxiv.org/abs/1812.08434.