

## POLITECNICO DI TORINO

**Corso di Laurea Magistrale in Ingegneria Informatica**


Master Thesis

# Deep Reinforcement Learning for TSP

**Problem description, approach, generalization**


Candidate: **Arya Houshmand**

Supervisor: **Tania Cerquitelli (PoliTo)**
**Ernestina Menasalvas (UPM)**
**Álvaro García Sanchéz (UPM)**
**Francisco Espiga Fernandez (UPM)**



Madrid, June 2022

# Acknowledgements

This document was developed during a 10 month Erasmus scholarship program at the *Universidad Politecnica de Madrid*, coming from the *Politecnico di Torino*.

It was an exciting and new reality, and I was able to experience it thanks to the support of my family, who eagerly fostered my fascination for computer science from a young age and deeply believe in the value of education.

I always looked up to my brother during my academic and personal life, he pushed me to strive and influenced me with his dedicated work ethic, which also led me to these circumstances.

I would like to thank my supervising professors for offering me the chance to explore this enthralling topic with their guidance.

# 1 Introduction

## Preface

Machine learning and artificial intelligence are more than ever changing how we perceive the relationship between humans and technology. In the past few years we witnessed a consistent surge in attention by the *STEM* community towards this field, which subsequently led to substantial improvements and new walls being broken.

As much as learning methods based on large pre-existent datasets are effective, have been proven to be reliable and have been applied in many fields, many researchers' attention has been leaning towards Reinforcement Learning techniques because of the potential freedom it enables. Reliance on data can be a critical constraint for certain problems. The information chosen may be warped, influenced by temporary actors, may not be feature rich or may not be of a satisfying size.

Simply put, in **RL** the aim is to model the problem with the appropriate and realistic inputs and outputs, well enough to enable the entity to learn from its (*generally simulated*) surroundings, in order to be able to solve problems in a real setting. Reinforcement Learning techniques endow the designer with the power to build entities capable of solving several problems, without the limit of large amounts of pre-existing data.

Combinatorial problems have recently sparked the interest of the RL community. Problems such as the *Travelling Salesman Problem* (***TSP***) and the *Vehicle Routing Problem* (***VRP***) are not a mistery and have been previously solved, but when the dimensions of the problem explode, current approaches cannot keep up with the enormous number of calculations that need to be performed. For the uninitiated, TSP asks the following :

> *"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"*

Both from an energy and time consumption perspective, it's worth trying to explore this new approach.

## Objectives and methodology

The first objective of this document was to describe and offer a model able to solve the *TSP* given a random instance of the problem, with the support of RL principles, in an acceptable amount of time.

In order to achieve this, the first step was to have a firm grasp of RL basics and the theoretical foundations. After this was achieved, with the support of the professors assigned to this project, the direction of the document was steered towards value iteration methods, ideal for low dimension problems. The following step was to introduce *deep neural networks* as a means to solve single instances.

The next natural objective was to train a model able to generalize the TSP instance and solve it from scratch with a nearly-optimal path solution.

In order to generalize the problem, the first hurdle was deciding a valid representation structure of the nodes to feed the neural network.

With the intention to improve the learning speed and performance of the model, the hypothesis that a higher dimensional representation of the node matrix could be beneficial was pursued.

Embedding techniques such as ***DeepWalk*** and ***Node2Vec*** were considered to extrapolate useful information about the instance structure and node position, but ultimately the focus went on towards ***vanilla autoencoders***.

## Document Structure

### Theoretical foundations

This document starts by touching the very basic principles of *Reinforcement Learning* and *Markovian Decision Processes*, *state and state-action value* functions and *Bellman's equations*, to offer the reader an acceptable understanding of the subject.

As soon as the basic concepts are clear, *Deep Neural Networks* will be briefly described, with the aim of allowing a better understanding of the Actor-Critic method and *Pointing and Attending mechanism* implemented by Bello in *Neural Combinatorial Optimization with Reinforcement Learning* [3].

### Practical attempts

In order to also have practical evidence of the inner basic workings of an RL agent, ***value iteration*** was chosen as the first method. It is widely used and its implementation was taken from [8]. The next technique was a vanilla Deep Q Learning method, whose main goal was to prove it was able to tackle the issues found in *Value Iteration*. It was consciously chosen to keep the first two approaches on a single graph instance, to prove the effectiveness of Reinforcement Learning for combinatorial problems.

The last and final experiment was made with the help of Bello's implementation, and an *Autoencoder* was incorporated to speed up the learning process.

# Contents

# 2 Reinforcement learning : formalisms and basics

## 2.1 Introduction

Reinforcement learning is a subfield of machine learning. To get a first glimpse about RL, it is important to start from the definitions of supervised and unsupervised learning.

- **Supervised Learning** : A *ground truth* is set, for example in the form of a dataset, where a large amounts of inputs and associated outputs offer a basis on which to build the learning abilities of the desired agent. Examples of supervised learning are *image classification and recognition* and *sentiment analysis*, for instance from large amounts of customer reviews.

- **Unsupervised Learning** : The data the model is working with is unlabeled. The main goal and objective is to be able to determine hidden relationships between data items, which otherwise may have been highlighted with a supervised technique. *Clustering* is a typical basic example for unsupervised learning : given a set of data points, the model is required to find similarities and common features and therefore classify and label the points.

Among the numerous machine learning approaches and techniques formalised during the years, (deep) reinforcement learning (**RL**), lies in a slightly separated realm.

The data is not predefined, but fed to the agent as it explores the given environment. The model is not completely blind and random: it has a reward system that over time dictates the learning direction of the agent, according to its past experiences. The specifics of this mechanic will be discussed more in depth later in the document.

## 2.2 Formalisms and definitions

The major elements in a RL problem definition are the following

- ***Agent***: the entity that interacts with the environment. It can perform *actions*, which determine a change in the environment. The agent can make *observations* on the state of the environment and can receive *rewards* from its surroundings based on the action performed at a particular moment.

- ***Environment*** : it is everything outside of the agent.

- ***Actions*** performed by the agent can be

  - **Discrete** : the agent can perform a set of limited, mutually exclusive actions. For example an agent that can only go in 4 directions, such as in a videogame, performs *discrete* actions.

  - **Continuous** : the action is represented with a value. If we were to describe the action of a robotic agent that has to perform a task, the action "put pressure on surface" will probably have a variety of magnitudes, which can have multiple outcomes on the environment.

- The ***reward*** is a scalar value that the environment communicates to the agent. There is not a defined reward frequency, but it is conventionally returned either in a cyclic matter with a defined time interval, or every time that the agent interacts with the environment. It is a way to inform the agent about the quality of the decision(s) taken. It is clear to the

reader that a single reward value is not enough to determine how the agent should behave, but it is necessary to observe the rewards over a certain amount of time. The agent's objective is to achieve the largest possible accrued reward over a sequence of actions.

- ***Observations*** are what the agent is able to retrieve from the environment's state. it is important to differentiate *state* and *observations*. The state includes everything that's not the agent. State and observations can overlap in certain simulated environments, but for most cases it is unlikely and unrealistic for the agent to have total knowledge of its surroundings. If state and observation overlap, it is called a *complete observation*, otherwise it is a *partial observation*.

The interactions between entities in a RL system can be summarised with the following diagram.

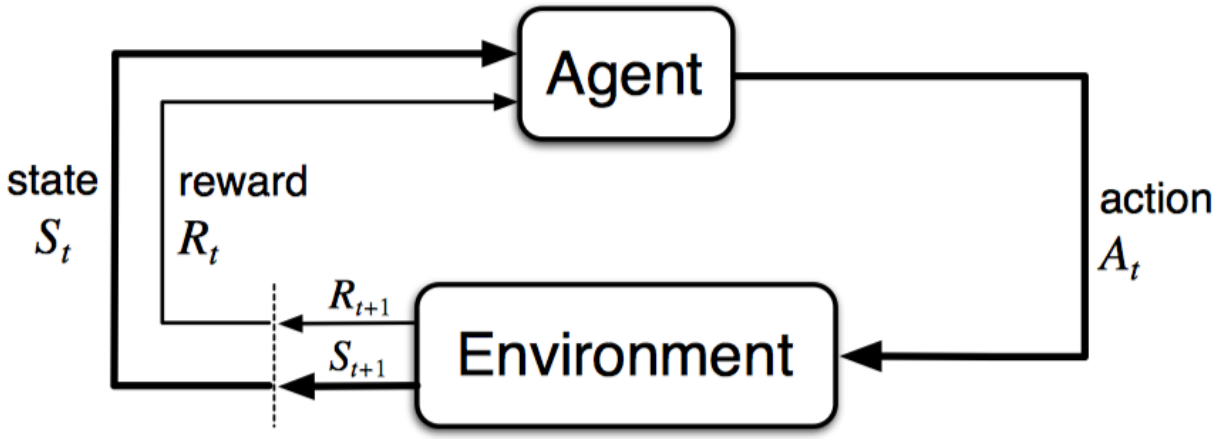



Figure 1: Basic diagram of RL entities and their interactions

### 2.2.1 Action Selection Policies

The action selection policy may be dictated by an $\epsilon$-**greedy policy** similar to the following, which has been proven to be effective in RL literature.

The agent can either perform decisions based on its current policy and knowledge (here represented by max $Q_t(a)$, later explained in 2.3.4) or randomly.

This requires the agent to balance its decision making with a mix of ***exploitation***, where the agent makes a decision based on its current policy and ***exploration***, where the agent takes a random action, in an attempt to discover more convenient actions. The above statement can be summarized as follows.

$$action_t = \begin{cases} maxQ_t(a) & with\ p = (1\text{- }\epsilon) \\ random\ action & with\ p = \epsilon \end{cases} \tag{1}$$

### 2.2.2 Reinforcement Learning algorithms

Reinforcement Learning literature offers a wide range of techniques, depending on the task and the environment specifics. It is the designer's responsibility to choose the most appropriate algorithm depending on the circumstances.

The most known algorithms are ***Monte Carlo***, ***SARSA***, ***Q-Learning***, ***Actor-Critic***. In this document most of the attention will be on *Q-Learning* and *Actor-Critic* based models, both *Temporal Difference (**TD**) Learning* algorithms. As defined by Sutton and Barto in [13],

> *TD is an approach to learning how to predict a quantity that depends on future values of a given signal. The name TD derives from its use of changes, or differences, in predictions over successive time steps to drive the learning process. The prediction at any given time step is updated to bring it closer to the prediction of the same quantity at the next time step. It is a supervised learning process in which the training signal for a prediction is a future prediction. TD algorithms are often used in reinforcement learning to predict a measure of the total amount of reward expected over the future, but they can be used to predict other quantities as well.*

There are advantages and disadvantages to such a choice.

- ***Advantages***
  1. TD updates its weights and parameters incrementally, and is not bound to the end of the episode. Therefore TD can be appropriate with environments where there might be incomplete sequences, without terminal states.
  2. TD takes advantage of the *Markov Property* (later examined in 2.3.1), hence it is more effective in *Markovian Decision Processes (MDPs)* .
- ***Disadvantages***
  1. Initial values may sway the learning process in the incorrect direction.
  2. The fact that target values are based on the model itself makes the model inherently biased.

When updating its weights and comparing target and prediction values, the *temporal difference error* is computed in the following way, as described by Barto in [13]

$$\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t) \tag{2}$$

where

- $R_{t+1}$ is the *return* gained from the state transition $S_t \rightarrow S_{t+1}$
- $V_t(S_t)$ is the current estimate for the state value
- $\gamma V_t(S_{t+1})$ is the discounted value of the next state in the transition.
- $\gamma$ is the discount factor

This definition will become clearer once the reader has gone through the concepts described in 2.3.

The *temporal difference error* represents the difference between the *current estimate* ($V_t(S_t)$) and the *target estimate* $y_{t+1}$ ( $R_{t+1} + \gamma V_t(S_{t+1})$ ). It is clear that the TD error can only be calculated after the step has been taken, when the necessary values will be available to the agent.

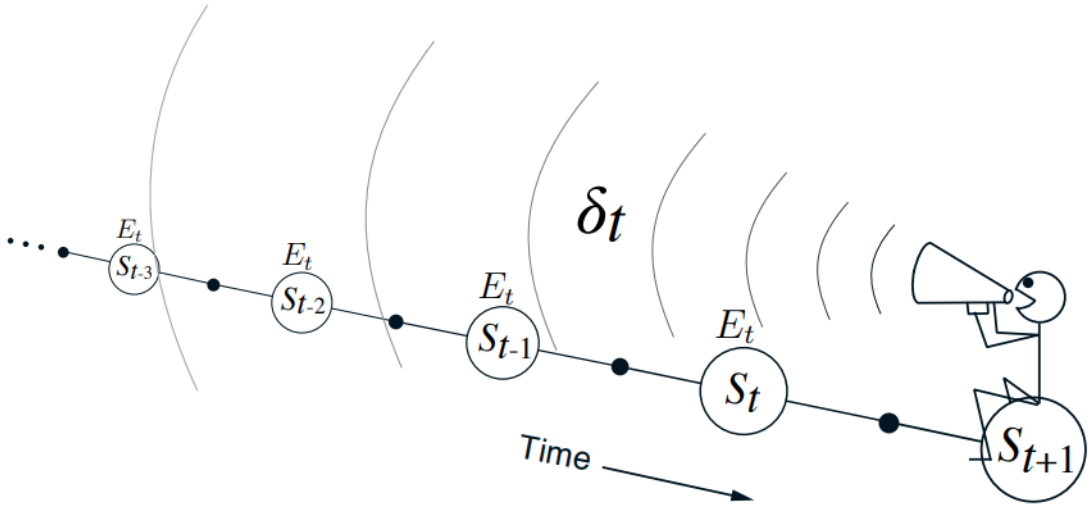



Figure 2: A more practical visual representation of the TD error, from [13]

By paraphrasing Barto, it is possible to assert that each update depends on the current TD error value $\delta_t$ and traces of past events.

## 2.3 Markovian Decision Processes (MDP)

### 2.3.1 Markov Property

Before the subject of *state values* and *state-action values*, it is necessary to make a rapid digression about MDPs. MDPs are relevant in RL literature because they enable to model *decision making* into a discrete, stochastic sequence of actions in the environment. *MDPs* need to satisfy the Markov property.

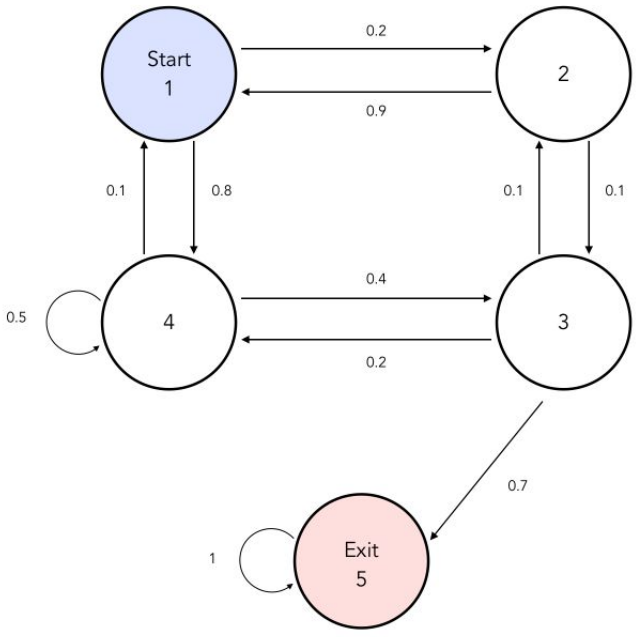
The ***Markov property*** requires that *"the future be independent of the past given the present"*. This means that the future in a MDP solely depends on the present history and does not depend on the past. The state of the process at a certain time $t$, $S_t$ has captured all relevant information from past history, and therefore all previous states $S_1, S_2, ..., S_{t-1}$ can be discarded when elaborating the ***transition probability*** to the next state $S_{t+1}$.

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, ..., S_t]$$

Given the current state $S$, the *state transition probability* is the probability that the next state $S'$ takes place.

### 2.3.2 Markov Process / Chain

It is the most simple MDP. it is a sequence of random states $S_1, S_2, ..., S_n$ that satisfy the Markov Property. A basic example of a Markov Process (without rewards) is the following.




It can be defined with a tuple: Markov Chain tuple ⟨S,P⟩

- S is a finite set of states
- P is the state transition probability matrix

To better understand the diagram, let's analyze one single node. If the agent chooses to go back to $S_1$, there is a 0.9 chance this will happen, and a 0.1 chance it will move to $S_2$. It is immediate that the sum of probabilities from each state must have a total sum of 1.

### 2.3.3 Markov Decision Process / Chain

Markov Decision Processes include rewards and active decision making in the operation. Defined as a tuple:

Markov Decision Process tuple $\langle$ S, A, P, R, $\gamma$ $\rangle$, where

- S is a finite set of states
- A is a finite set of actions
- P is a state transition probability matrix
- R is a reward function $R_s = [R_{t+1}|S_t = s]$
- $\gamma$ is a discount factor $\gamma \in [0, 1]$. it is an arbitrary value chosen by the designer.
  $\gamma \to 0$ determines a more short-sighted decision making operation, one that values immediate reward higher.
  $\gamma \to 1$ instead encourages more far-sighted decisions.

In a MDP, the goal is to maximise the accumulated ***return*** $G_t$. $G_t$ is the total discounted reward, starting from time step $t$.

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3}$$

In MDPs, it is fundamental to also define the following:

- ***Policy*** : informally it is the strategy chosen by the agent to reach its objective and (ideally) maximise the return $G_t$. More formally, it is the distribution over action given states.
  $$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s] \tag{4}$$
  In short, policies determine a mapping between one state and another. Given an agent at state $s_x$, from that particular state it will map the probability of taking a certain action.
- ***State Value Function v(s)*** : is a measure of the long-term value of state s.
  $$v_\pi(s) = \mathbb{E}\left[G_t|S_t = s\right] \tag{5}$$
  States with a higher value are preferred because they generally imply a better total reward.
- ***Action-Value function*** $q_\pi(s, a)$: it is the expected return for an agent that starts in state $s$, performs action $a$ and follows policy $\pi$.
  $$q_\pi(s, a) = \mathbb{E}\left[G_t|S_t = s, A_t = a\right] \tag{6}$$
  it is the measure of how convenient is to choose a certain action starting from a particular state.

### 2.3.4 Bellman Expectation and Optimality equations

Bellman wrote the aforementioned variables in another and easier to grasp format.

- ***State-value function***

$$\begin{aligned} v_\pi(s) &= \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s\right] \\ &= \sum_{a \in A} \pi(a|s) q_\pi(s,a) \\ &= \sum_{a \in A} \pi(a|s)(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s')) \end{aligned} \tag{7}$$

  it is evident it is a sum, whose components are the following

  - A ***policy*** $\pi$ that dictates the action to be taken at state s
  - An action-value function that will return the action-value of the current *(state, action)* tuple. This element is then decomposed as the $R_s^a$, the return from that specific *(state, action)* combination and the sum of the *discounted* future state-values for that given policy, also taking into consideration the state transition probability $P_{ss'}^a$

- ***State-Action value Function***

$$\begin{aligned} q_\pi(s) &= \mathbb{E}\left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a\right] \\ &= R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s') \\ &= R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a \in A} \pi(a'|s') q_\pi(s', a') \end{aligned} \tag{8}$$

  It describes the *expected* return when the agent is in state $s$ and performs action $a$ at time step $t$.

- ***Optimal State-value and State-Action value Functions*** In order to garner the maximum possible reward, the agent must pick out the *optimal policy*, the one that maximises the total return. The solution of an MDP lies within the optimal policy.

  The *optimal state-value function* $v_*(s)$ is the maximum value function, between all policies. It will compute the maximum possible return that can be generated from the system. It can be more formally written as

$$v_*(s) = max_\pi v_\pi(s) \tag{9}$$

  The *optimal state-action value function* $q_*(s,a)$ corresponds to the maximum action-value that can be extracted over all the policies, when the system starts in state $s$ and takes action $a$.

$$q_*(s) = max_\pi q_\pi(s,a) \tag{10}$$

# 3 Value Iteration algorithm

The following method can only be used in RL environments where the *transition probabilities* and the *rewards* are known to the designer. The main idea at the center of this algorithm is to gradually improve the policy by iteratively increasing the quality of the $V(s)$ estimation. The basic pseudo code is the following

```
V(s) is initialized with arbitrary values
Loop until V(s) converges
    For every state
        For every action
            compute action-value Q(s,a)
        V(s) = max Q(s,a)
```

Despite the power of these concepts, the ***Value Iteration*** method has several constraint.

- In order to be of practical use, the value-iteration algorithm must work with discrete and small state spaces, to ensure that looping through the states is computationally feasible.
- It was previously said that *state transition probabilities* and *rewards* must be known to the agent. When that is not achievable a priori, it is advisable to perform random exploration through the environment to discover these necessary values.

At the end of the value-iteration algorithm, the agent will have filled out an exhaustive mapping structure that will enable the agent to perform actions based on a (ideally) optimal policy.

With the (state,action) $\rightarrow Q(s, a)$ mappings, at every step the agent will have a value to compare different actions when performing a decision.

The second section of this document will have a practical implementation demonstration of a value-iteration algorithm for the *Travelling Salesman Problem*.

# 4 Deep Q Learning

Value Iteration is a powerful tool, but in order to fully exploit the RL principles discussed previously, the research community has adopted *Artificial Neural Networks* in order to make improvements in the field.

## 4.1 Deep Learning principles, reasoning

As defined by Haykin in [6]

> *A neural network is a massively parallel distributed processor made up of simple processing units that has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:*
>
> 1. *Knowledge is acquired by the network from its environment through a learning process.*
> 2. *Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge*

Layers of weighted neurons can mimic *human* neural interconnections and improve their results by implementing *learning algorithms*.

Deep Neural Networks can tame the exploding dimensions issue present in the value iteration example previously analyzed. Such a statement has been proven by Lapan in [8] and will be confirmed by the practical implementation available in the second part of this document.

## 4.2 Deep Q Learning

### 4.2.1 Deep Q Networks (DQN)

Deep Neural networks substitute the tabular mappings defined by the *Value Iteration algorithm*, enabling a less memory and computationally heavy operation. The neural network will perform a mapping between *states* and *(action, Q-value)* pairs.

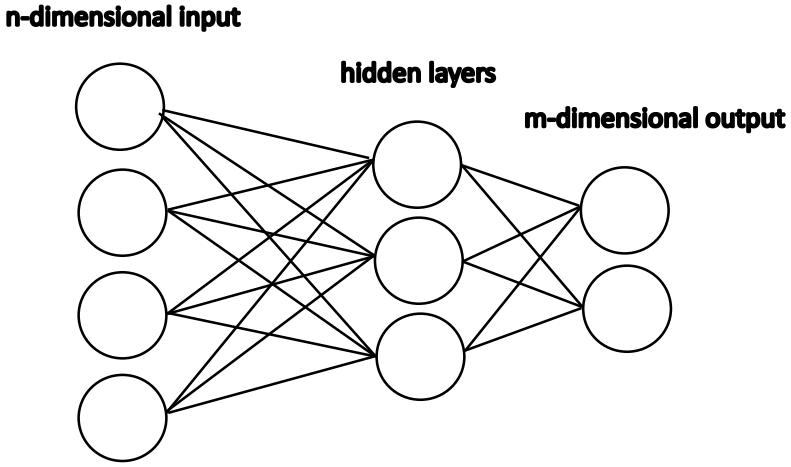



Figure 3: Intuition of Deep Neural Network structure

The input can be a n-dimensional state describing the current circumstances in a vectorial or matrix form. The output has the same dimensions as the action space. Given $m$ possible actions that can be performed by the agent, the network will assess a Q-value associated with the index of the action to determine the quality of the decision.

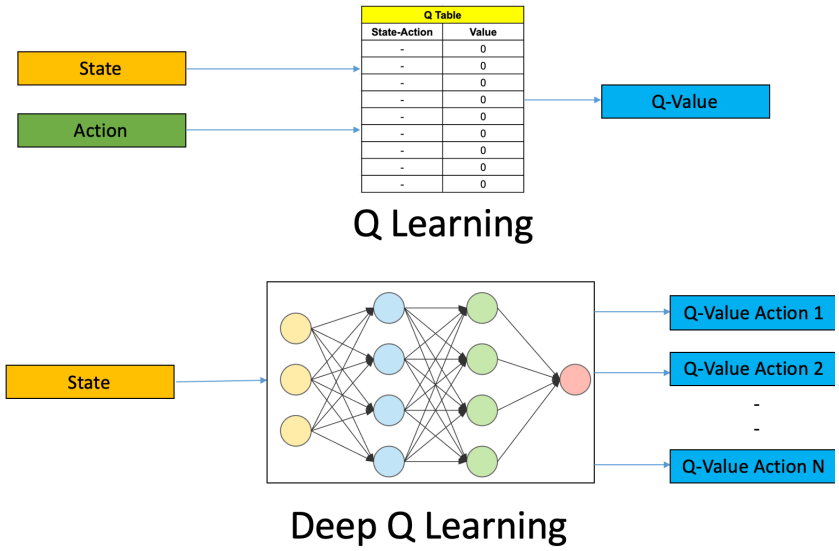



Figure 4: Comparison in approach visualized

### 4.2.2 Experience Replay

Instead of performing learning operations for every step of the episode simulations, the agent will gather *experiences* from the interaction with the environment and use this information to build its knowledge base. The experience can be represented as a tuple $(state, action, nextstate, reward)$. From the pool of recorded experiences, the agent will extract a batch and learn from this portion. This choice ensures low correlation between experiences and should avoid *overfitting* and encourage *generalization*.

### 4.2.3 Learning Process

The learning process in *Deep Q Learning* requires 2 neural networks, a main and a target network. The reasoning behind this choice can be made clear with a comparison with regular *deep learning*. Conventional deep learning has fixed target values, since the ground truth is extracted from a pre-existing dataset. In Reinforcement Learning, the target values change continuously as the agent explores the environment. A single network cannot efficiently compute both the target and the prediction values. In fact, these two values may diverge and the estimation may become unfeasible, hence the need for two networks. Once every N steps, the weights of the main network are copied to the target one. This choice ensures more stability in the learning process, as confirmed in [10]. A system with only one neural network could incur in ***catastrophic interference***, as described in [9]. This phenomenon happens when marginal updates may have negative effects on the quality of decisions by the agent. If the designer were to only use one network, the learning process will probably make improvements for a period, then it will start taking disadvantageous choices.

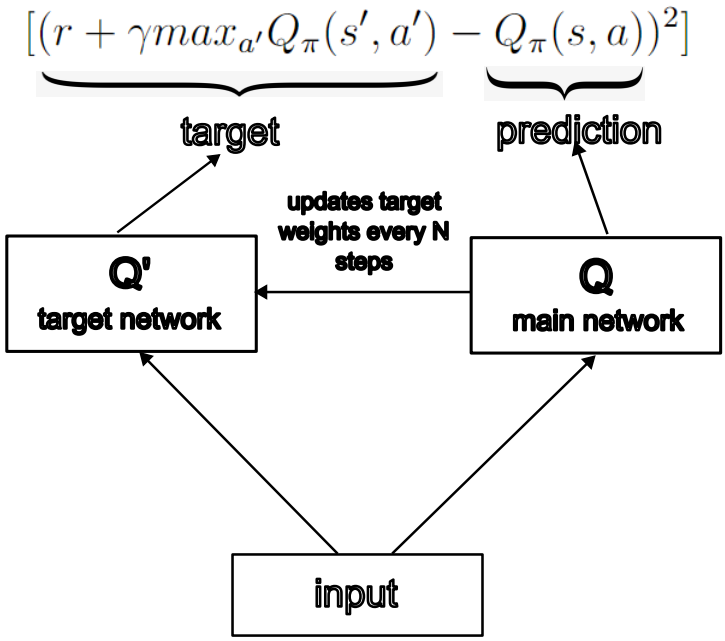



Figure 5: Inner workings

The Deep Q Learning process can then be summarised in the following steps

1. The agent's state $s$ will be fed to the *Deep Q Network* ***DQN***, which will output the Q-values corresponding to the possible actions.
2. Select the action depending on the chosen *action selection policy*
3. Perform the chosen action. The agent will move to the next state $s'$ and will be given a reward.
4. Save the experience $(s, a, r, s')$ tuple in the *experience replay buffer*
5. Extract a batch of experience tuples from the buffer, and determine the ***loss*** value.
6. Perform gradient descent to adjust network parameters in order to minimize the *loss*.
7. Every N iterations copy the *main network* weights to the *target network*.
8. Repeat for the number of episodes required.

In the second part of this document there will be a practical demonstration of a DQN implementation for the *Travelling Salesman Problem*.

# 5 Actor-Critic Model

### 5.0.1 Introduction

In Reinforcement Learning, as mentioned by Konda in [7], the majority of RL methods fall under two main categories.

- ***Actor-only methods*** : the model's goal is to learn the optimal policy. Policy-gradient algorithms have been proven to be effective in Actor-only methods. Such methods are less prone to oscillations during the learning process, but they lack the past-awareness of *Critic methods*. Therefore, as the policy is updated, the gradient estimation happens independently from past experiences.
- ***Critic-only methods*** : based on the *value function*, it performs an evaluation of the action. As mentioned in 2, the action with the highest value will be chosen as the most convenient, according to the current policy. Such methods are reliable to approximate the *Bellman equation*, but do not always ensure near-optimality in the policy. An example of critic-only method is *Q-Learning*.

By combining the advantages of both method types, it is possible to design a better performing agent. The *critic*, by simulating experiences, will be able to approximate the Bellman Value function, which will be used by the *actor* to change its parameters in the direction of better performance.

It should be highlighted that learning is *on-policy*: the critic must follow the actor's direction, while performing its *critiquing* role. The *critique* manifests itself as the *TD error*, whose value is the driving force of the learning process.

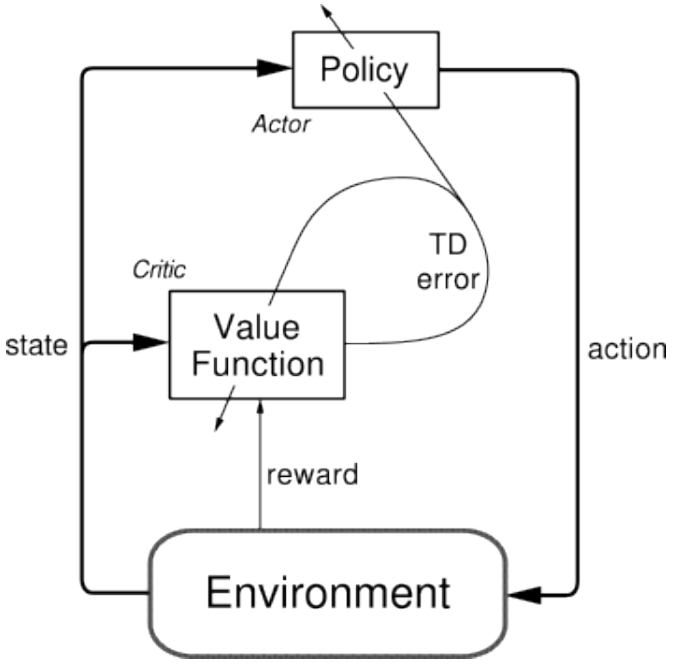



Figure 6: Actor-Critic visual representation, from [13]

### 5.0.2 Basic learning process

It has been previously stated that TD can be expressed as 2. The TD error can be used to determine the quality of the action selected by the actor.

If the $\delta_t$ is positive, the action $a_t$ selected while in state $s_t$, the weights will be adjusted accordingly to strenghten this decision making pattern. If otherwise $\delta_t$ is negative, the tendency to perform such an action should be weakened.

Informally, it is possible to summarise the process

1. The *agent* will feed its current state $s_t$ to the *actor* who, depending on its current policy, will return a certain action as the most convenient.

2. The selected action is performed and applied to the environment, which will update the state, now $s_{t+1}$, and return the reward $R_t$

3. The successive states and the associated information, the tuple $(s_t, a_t, R_t, s_{t+1})$ will be used to compute $\delta_t$ by the critic. As previously said, this value will determine the direction of the learning process.

4. The value of the TD error will determine the direction of the updates in parameters and weights, aimed at minimizing the loss value. Arbitrarily selected $\alpha$ learning rate values will be chosen by the designer as hyperparameters to determine the scale of the step to be taken.

Or also, in pseudocode form, as presented by Sutton and Barto in [13], implementing the ***REINFORCE*** algorithm to support the actor.

Actor-Critic algorithm

Input : a differentiable policy parametrization $\pi(a|s, \boldsymbol{\theta})$
Input : a differentiable state-value parametrization $\tilde{v}(s, \boldsymbol{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0, \alpha^{\boldsymbol{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $w \in \mathbb{R}^d$
Repeat forever
    Initialize S (first state of episode)
    $I \longleftarrow 1$
    While S is not terminal
        $A \sim \pi(\cdot|S, \boldsymbol{\theta})$
        Take action A, observe S', R
        $\delta \longleftarrow R + \gamma \hat{v}(s', \mathbf{w}) - \hat{v}(s, \mathbf{w})$  (if s' is terminal, $\hat{v}(s', \mathbf{w}) = 0$ )
        $\mathbf{w} \longleftarrow \mathbf{w} + \alpha^{\boldsymbol{w}} \delta \nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w})$
        $\boldsymbol{\theta} \longleftarrow \boldsymbol{\theta} + \alpha^{\theta} \delta \nabla_{\boldsymbol{\theta}} ln \pi(A|s, \boldsymbol{\theta})$
        $I \longleftarrow \gamma I$
        $s \longleftarrow s'$

A common misconception about the Actor-Critic model architecture is the following:

*Why is not the critic-only approach the most effective? If the model is able to determine the best action by approximating the action-value function $q(s, a)$, why would a policy-directing network be necessary?*

The question is legitimate, but it takes for granted the fact that the Critic's approximation of the value-function is near-optimal, which is hardly the case, especially in systems where the experiences and states explode in number.

The Critic's policy is often a greedy and simple $argmax_a q(s, a)$, which can often cause the learning process to be unstable. If for example the model suddenly determines a certain maximising action, this event could determine a substantial change in the policy, which equates to a less smooth learning process.

## 5.1 Actor-Critic methods for TSP

Deep Q Networks and vanilla Q-Learning are a valid choice to approach and solve the *Travelling Salesman Problem*, but as for Critic-only learning methods, there is a high chance the agent will settle for a local minimum when minimizing the loss value. As discussed above, combining the main advantages of both methods can prove to be effective when solving a problem such as TSP. A recent implementation was published by Bello in [3].

### 5.1.1 Neural Network Architecture

Bello's architecture is aimed at solving a 2D Euclidean TSP. The *state* fed to the network consists of the node coordinates of the environment to explore.

The networks proposed by Bello are composed of two *recurrent neural network* (**RNN**) modules, an *encoder* and a *decoder*. Both these networks are *Long Short-Term Memory (**LSTM**)* cells. The reasoning behind such a choice can be found for example in [2]. Bakker asserts that

> *RNNs, such as LSTM, can be applied to RL tasks in various ways. One way is to let the RNN learn a model of the environment, which learns to predict observations and rewards, and in this way learns to infer the environmental state at each point. LSTM's architecture would allow the predictions to depend on information from long ago. The model-based system could then learn the mapping from (inferred) environmental states to actions as in the Markovian case, using standard techniques such as Q-learning, or by backpropagating through the frozen model to the controller*

LSTMs are generally aimed at problems that require learning long-term dependencies in timeseries data. This directly contradicts the Markovian principle, but the assumption is generally soft, so it is possible, as demonstrated by Bello, to take advantage of RNNs in Reinforcement Learning. The decision-making process is guided by a Pointer Network, first introduced by Vinyals in [15]. This approach deserves a small excursus, in order to better understand how the model behaves and learns.

### 5.1.2 Attention Mechanism

To better grasp the concepts, it is necessary to go a bit more in depth. The attention model was first introduced in *neural machine translation*, where a ***seq2seq*** model was largely used beforehand. This model's architecture is composed of an encoder-decoder structure.

The ***seq2seq*** mechanism works in the following way.

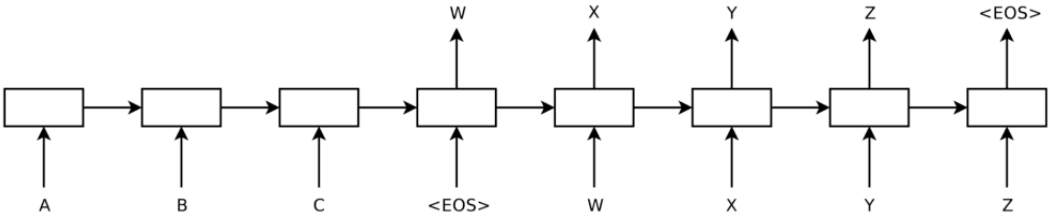



Figure 7: Basic example of *seq2seq* mechanism at work

In a *machine translation environment*, the input sequence $I = A, B, C, < EOS >$ will generate an output $O = W, X, Y, Z, < EOS >$, where the output length may be different from the input. The *encoder* will read the sentence word by word, and as a consequence its internal state will be updated accordingly. When it reaches a *EOS* (end of sentence) the procedure will stop. At this point the encoder is supposed to have a sufficiently accurate vectorial representation ***of fixed length*** of the input sequence, which will be used by the *decoder* to generate the output (translated) sequence. The output is generated one word at a time, by feeding the decoder with the previous decoder state of the previous time step. The issue with such an architecture is that input sequences of variable length are represented by a *fixed-length vector*, likely resulting in the vector lacking relevant information about the input sequence.

The issue is solved by implementing an *attention mechanism*, first introduced in [14] by Vaswani. It exploits attention as a *pointer* to select an element on the input sequence as the output. The *attention mechanism* was proposed to address this particular issue. The main idea is not to discard intermediate states generated by the encoder, but instead make use of all states to build an appropriate context vector to be later fed to the decoder.

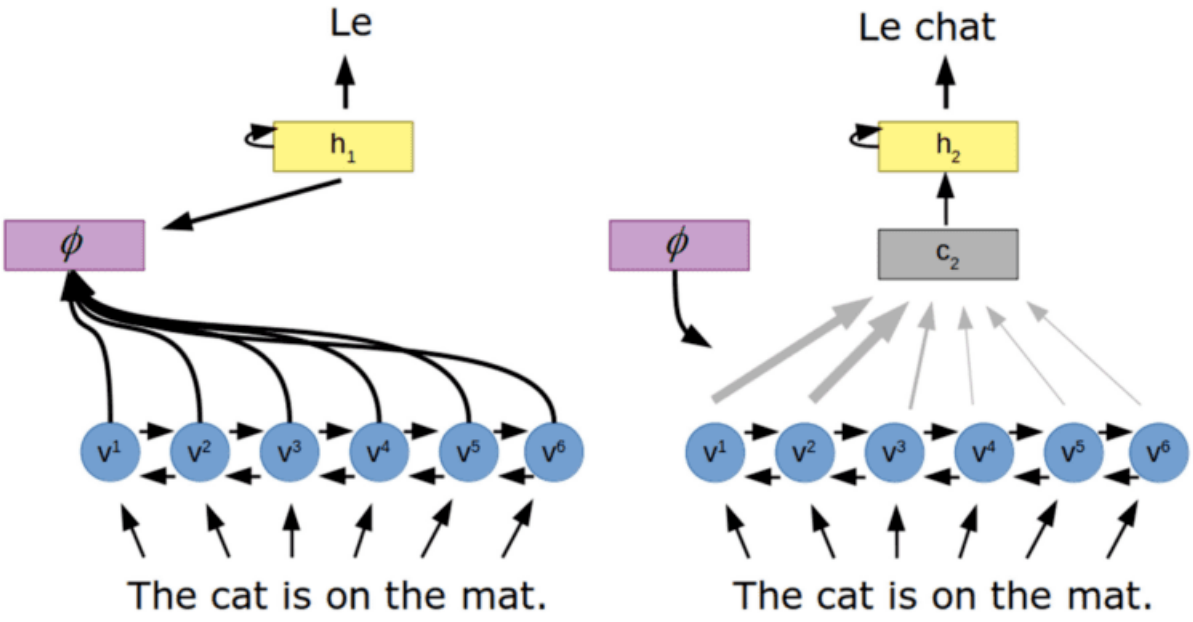



Figure 8: Representation of attention mechanism's inner working, from [1]

The attention mechanism, here denoted as $\boldsymbol{\theta}$, will adjust its weights to observe and acquire the relationships between the encoded input vectors ($v^x$) and the hidden state of the decoder ($h$). The attention mechanism's weights will then be employed to perform a weighted sum of the hidden states of the encoder in order to generate the *context vector c*. The attention mechanism will have now captured the most relevant portions of the input sequence, which will result in a more accurate and efficient translation.

As described by Vaswani in [14], it is possible to describe the Attention mechanism in the following manner.

> *An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.*

.

It can be also described as *a generalized pooling method with bias alignment over inputs.* Here follows a dissection of the mechanism, specifically for Bello's model. In [3]

- $W_{ref}$ = learnable weight matrix, refers to the *keys*
- $W_q$ = learnable weight matrix, refers to the *queries*
- $v^T$ = attention vector
- $u_i$ = attention scores vector
- $A(ref, q; W_{ref}, W_q, v\ )$ = attention mechanism

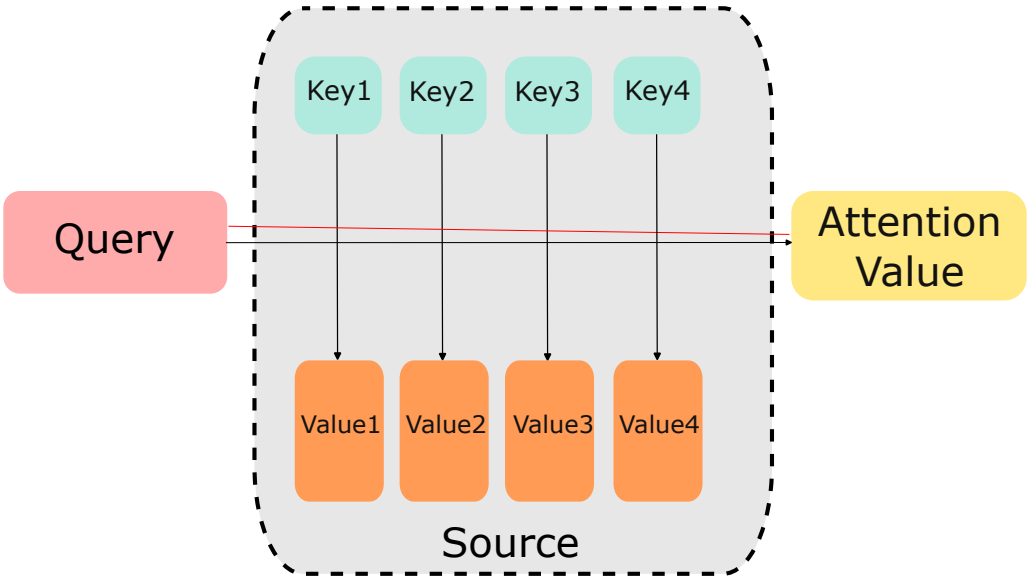



Figure 9: Attention Mechanism Query, Key, Value representation

- ***INPUT :*** query: it corresponds to the hidden decoder state at the selected time step.
- ***OUTPUT :*** It is based on the model's memory, which is a set of *key-value* pairs. The output is a measure of the similarity between the query and the keys. In a TSP environment, the higher this attention score, the more convenient is the node to take at that specific decoder step. This output, the attention score, is determined by the *score function* $\alpha$, which can be chosen by the designer depending on the model's desired behaviour. The score is determined by the learnable weight matrixes $W_{ref}$,$W_q$, $v^T$.

At every step $j$ of the decoder, the attention function will determine the *alignment score* in the following way, where

- $q$ = *query*
- $r_i$ =*reference*$_i$, the key, (the node index in a TSP environment)
- $\pi$ is the current sequence of visited nodes.

1. Score function $\alpha$ is applied *for* $i = 1, 2, .., k$

$$u_i = \begin{cases} v^T \cdot tanh(W_{ref} \cdot r_i + W_q \cdot q) & \text{if } i \neq \pi(j) \text{ for all } j < i \\ -\infty & \text{otherwise} \end{cases} \tag{11}$$

The score will be computed only for nodes that have not been visited yet in the current path exploration instance. Nodes that have already been visited are masked with a -$\infty$, to exclude them a priori from consideration.

2. The ***softmax*** function is applied to $u$, to generate a probability distribution over the input elements.

$$A(ref, q; W_{ref}, W_q, v^T) := softmax(u) \tag{12}$$

The operation can be summarised with the following definition, taken from [3] *"Our pointer network, at decoder step j, assigns the probability of visiting the next point of the tour as follows"*

$$p(\pi(j)|\pi(< j), s) := A(enc_{1:n}, dec_j) \tag{13}$$

### 5.1.3 Pointer Network

Vinyals, in [15], summarises the Pointer Network architecture as

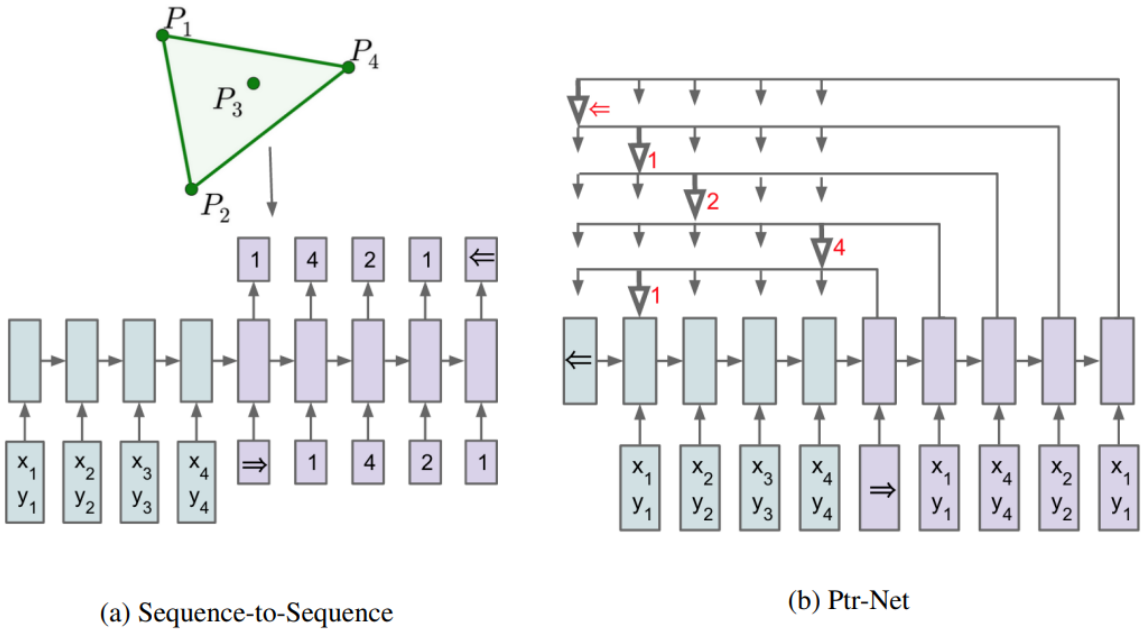
> *(...) a new architecture that allows us to learn a conditional probability of one sequence $C^P$ given another sequence $P$, where $C^P$ is a sequence of discrete tokens corresponding to positions in $P$.*

As pointed out before, the main issue with *seq2seq* models is that the *softmax* operation is executed on a fixed size output dictionary. This is not compatible with problems where the output dictionary size depends on the input sequence length. The pointer network can be informally introduced as an extension of the Attention Mechanism. The *Ptr-Net* introduced by Vinyals attempts to solve the problem in the following way.

The attention mechanism described in 5.1.2 is here implemented in the following way.

$$u_j^i = v^T tanh(W_1 e_j + W_2 d_i) j \in (1, ..., n)$$
$$p(C_i|C_1, ..., C_{i-1}, P) = softmax(u^i) \quad (14)$$

- $v^T, W_1, W_2$ are learnable parameters of the output model
- $e_i$ and $d_i$ are the encoder and decoder hidden states
- The *softmax* function is applied to $u_j^i$ to generate a distribution over the input sequence elements.
- As Vinyals asserts in [15], *"(...) We also note that our approach specifically targets problems whose outputs are discrete and correspond to positions in the input"* which is an ideal method for problems such as TSP.




(a) Sequence-to-Sequence (b) Ptr-Net

Figure 10: Graphic comparison of Seq2Seq and Ptr-Net, from [15]

At this point, with a better comprehension of the *Attention Mechanism* and the *Pointer Network* architecture, it is easier to understand the reasoning behind Bello's design decisions for its model. The ***Attention Mechanism*** and the ***Pointer Network*** were chosen to optimize

a problem where past memory is fundamental for its decision making process during an episode. In a TSP instance, the agent is given information about a set of nodes, and at each step must perform a memory-conscious choice between them, considering the past. The reader can immediately recognise the parallel between the language translation process described before and the TSP solving process.

## 5.2 Bello's implementation for TSP

In [3], Bello implements the Pointer Network introduced by Vinyals to design a model optimized specifically for the Travelling Salesman problem.

### 5.2.1 Model training and optimization

Exactly like the previously Ptr-Net architecture described before, Bello's model is composed by two *Recurrent Neural Networks*, or ***RNNs***, both of which *Long Short-Term Memory* cells, or ***LSTMs***. The ***encoder*** network will read the input sequence one node at a time, and convert it into a set of *latent memory states* $\{enc_i\}_{i=1}^{n}$

The ***decoder*** network also has its own *latent memory states* $\{dec_i\}_{i=1}^{n}$ At every step $i$, the ***pointing mechanism*** will "*(...) produce a distribution over the next city to visit in the tour*".

The learning algorithm used is ***Actor-Critic***, presented in a slightly different version from what seen in 5.0.2.

In order to better understand the algorithm proposed by Bello, here follows a small introduction.

- The objective is to reduce the cost function of the expected tour length, here denoted as

$$J(\theta|s) = \mathbb{E}_{\pi \sim p_\theta(\cdot|s)} L(\pi|s)$$

  Paraphrasing, given parameters $\theta$ of the *policy network (actor)* it is the expected tour length $L$, according to policy $\pi$, on graph $s$.
- Bello concentrates on the fact that the use of a heuristic TSP solver to determine target values was avoided, because
  1. The learning progress, and the policy will be influenced by the quality of the labeled data available
  2. The generation of quality and reliable labeled data may become unfeasible as the dimensions of the problem increase.
  3. It is more ideal to build a model able to learn from its experience and become competitive, and not on replicating the results of a different algorithm.

- *Policy gradient* and *Stochastic Gradient Descent (**SGD**)* are used to optimize the parameters of the networks. The gradient of the cost function described above is formulated as

$$\nabla_\theta J(\theta|s) = \mathbb{E}_{\pi\sim p_\theta(\cdot|s)}\Big[(L(\pi|s) - b(s))\nabla_\theta \log\, p_\theta(\pi|s)\Big]$$

  using REINFORCE, introduce by *Williams* in [16]. In the previous equation, $b(s)$ is a baseline that does not depend on the selected tour, but it is " *an exponential moving average of the rewards obtained by the network over time*". This baseline however is not capable of discriminating between different input graphs, but was introduced to reduce variance. In order to improve the learning process and also consider the input graphs for the parameter update mechanism, the *critic* will be aided in *SGD* by the *mean squared error (**MSE**)* between the prediction $b_{\theta_v}(s)$ and the actual tour length sampled from the current policy $\theta$. This further objective value is formulated in the following way

$$\mathcal{L}(\theta_v) = \frac{1}{B}\sum_{i=1}^{B} \parallel b_{\theta_v}(s_i) L(\pi_i|s_i) \parallel_2^2$$

Actor-Critic training

**Procedure** *Train* (training set S, T steps, batch size B)
  Initialize pointer network parameter $\theta$ *actor network*
  Initialize critic network parameter $\theta_v$ *critic network*
  **for** $t = 1$ to $T$ **do**
    $s_i \sim$ *SampleInput (S)* for $i \in \{1, ..., B\}$
    $\pi_i \sim$ *SampleSolution* $(p_\theta(\cdot|s_i))$ for $i \in \{1, ..., B\}$
    $b_i \longrightarrow b_{\theta_v}(s_i)$ for $i \in \{1, ..., B\}$
    $g_\theta \longleftarrow \frac{1}{B}\sum_{i=1}^{B}(L(\pi_i|s_i) - b_i)\nabla_\theta \log p_\theta(\pi_i|s_i)$
    $\mathcal{L}_v \longleftarrow \frac{1}{B}\sum_{i=1}^{B} \parallel b_i - L(\pi_i) \parallel_2^2$
    $\theta \longleftarrow Adam(\theta, g_\theta)$
    $\theta_v \longleftarrow Adam(\theta_v, \nabla_{\theta_v}\mathcal{L}_v)$
  **end for**
  **return** $\theta$
**end procedure**

### 5.2.2 Further development and experiments

This paper was chosen because of its novel approach to the problem, and with the input of the supervising professors, an improvement path was suggested to enhance the current working model proposed by Bello.

As written in 5.1.1, the input to the model published by Bello consists simply of the node coordinates. We believe that the quality of the representation given to the network is vital in the learning process, and more useful information can be given about the environment to ensure a better decision making agent and a faster learner.

# 6 Autoencoders

## 6.1 Preface

While choosing between a method to fit our purpose, there were several tests made, between Visual Transformers, Graph Embedding techniques such as ***Node2Vec*** ([5]) and ***DeepWalk*** ([12]), and (Graph) Autoencoders. However, especially for Graph Embeddings, the learning process was painstakingly slow due to the model having to fit each graph fed to it, in order to extract a information-rich representation. Autoencoders instead seemed to fit perfectly, by removing the processing bottleneck and training a model to encode on the fly any graph fed to the network. Repositories and blog posts about effective usage of node embedding techniques in Deep Learning are available online, but the resources constraints pushed the document in the Autoencoder direction.

## 6.2 Introduction

Encoding techniques have always been a central component in computer science and technology. The encoding process translates information from one representation to another, while keeping the underlying information identical.

*Encoding* and *decoding* techniques have proven effective in a number of fields where data and information have to be transformed into another format, either for memory compression, to tailor data for a specific hardware component for more efficient computation or for analog-to-digital conversion.

In this particular task, the aim is to generate a higher quality representation of the nodes, which should ideally contain hidden information that will aid the model in the learning process and/or produce better results and predictions.

***Autoencoders*** are a particular subset of *neural networks* where input and output coincide. The objective of the model is to

1. *encode* the input into a smaller or higher dimensional space
2. *decode* the intermediate output into the initial input.

It is however important to highlight three main aspects of autoencoders.

1. The *reconstructed* output will not be identical to the input, there will still be a marginal discrepancy depending on the quality of the model. It is not a *lossless* technique.
2. Autoencoders can accurately encode data only if it is similar to the chosen training set. They are not expected to be versatile components and able to generalize out of their training scope.
3. The learning process is unsupervised, since the training data is already labeled by definition. It can be said that the process is *self-supervised*.

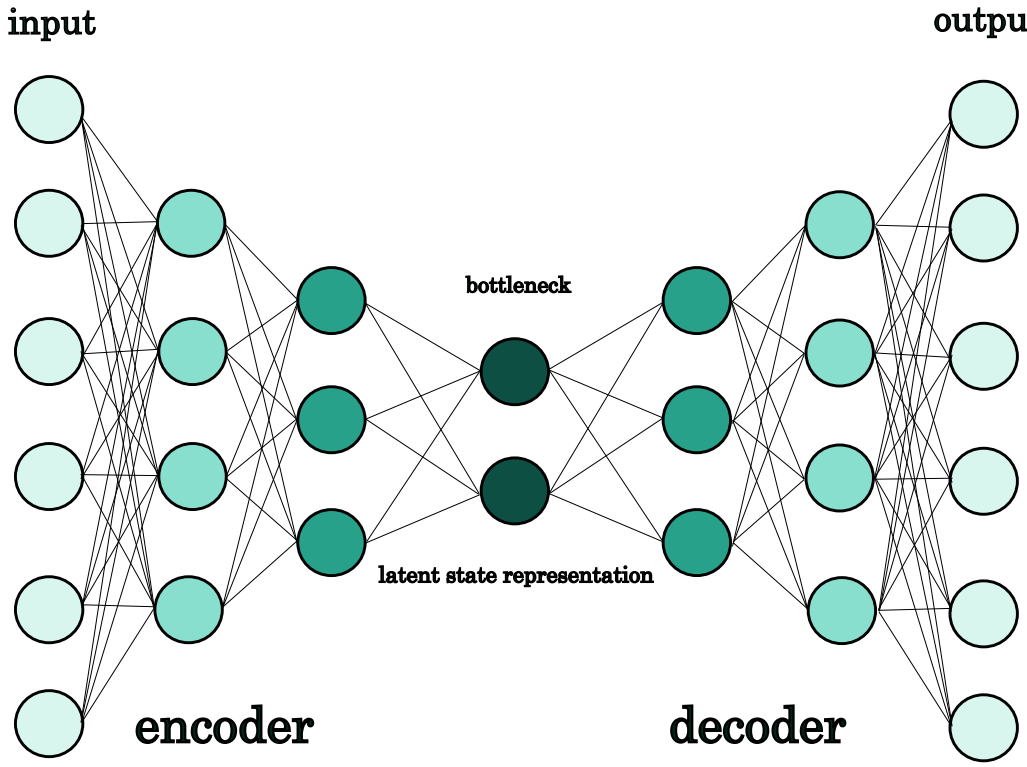



Figure 11: Basic example of autoencoder structure

## 6.3 Graph Autoencoders

The classic version of the *Travelling Salesman Problem* has just 2 main features, between edges and nodes. The nodes have *(x,y)* 2D information, and the edge attribute is the distance between nodes. Other environments have hundreds or thousands of features per node (and/or edge), for example the *Cora Citeseer* dataset is extremely rich in features, therefore a compact version of the information is more useful and may make the learning process easier for the model.

In this TSP approach it was decided to opt for an explosion in dimensionality, from 16 to 128 features per node, starting from the initial 2 *(x,y)*.

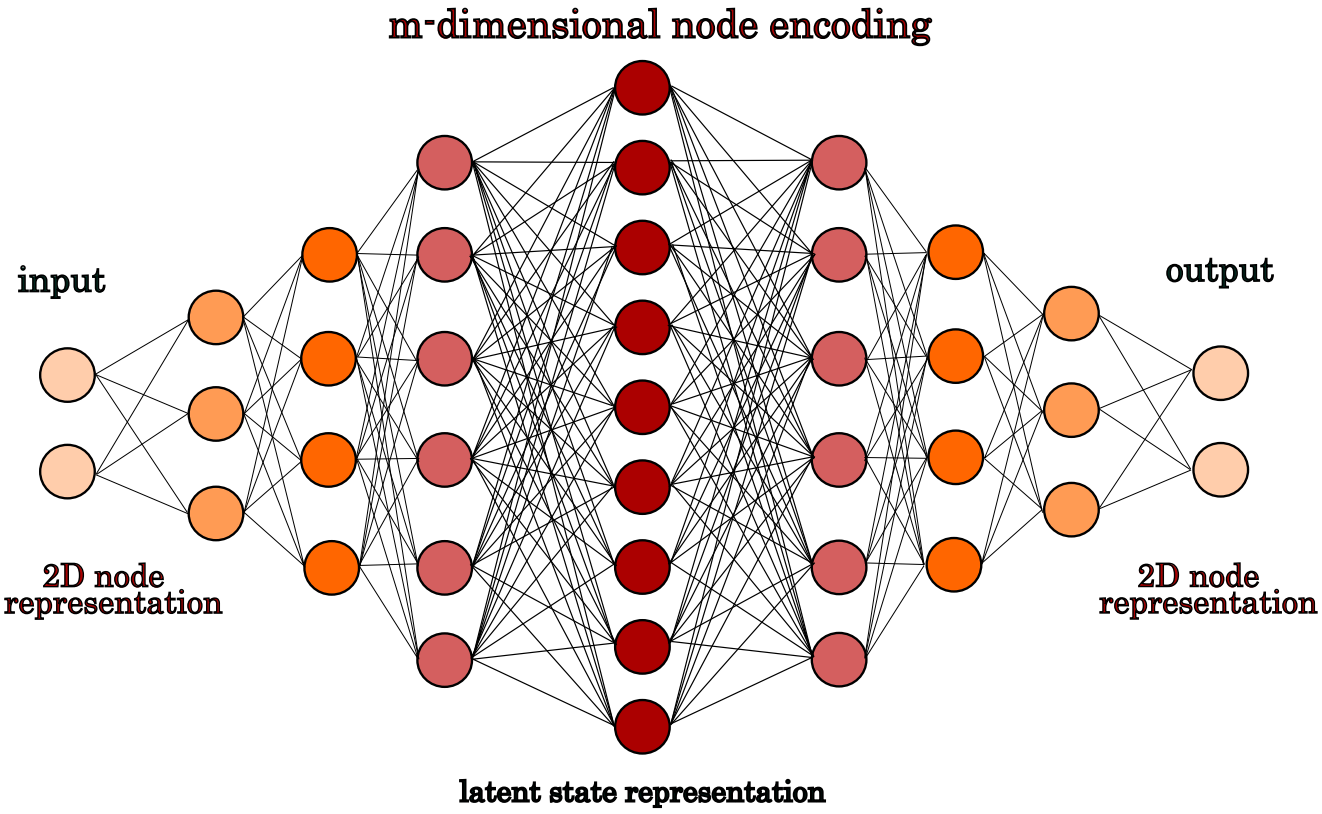



Figure 12: An intuition of the autoencoder approach used for TSP

# 7 Experiments

## 7.1 Preface

During the development of the document, the objective was to organically incorporate the theory principles acquired with practical experiments, to have tangible results and have a hands-on experience with the different approaches.

Some *Reinforcement Learning* and *Deep Learning* concepts may be more difficult to grasp without some form of substantiation via code.

## 7.2 Value-Iteration for TSP

The first approach taken was *Value-Iteration*. This is a relevant step to first comprehend *Markov's principle* and *Bellman's equations*.

### 7.2.1 Tools

The environment,tools and literature chosen to support this experiment were

- ***OpenAi GYM***: "*an open source Python library for developing and comparing reinforcement learning algorithms by providing a standard API to communicate between learning algorithms and environments, as well as a standard set of environments compliant with that API.*"
- ***Deep Reinforcement Learning Hands-On***, by *Maxim Lapan* [8] : this publication offers both theoretical and practical examples to gently introduce RL principles and concepts.

### 7.2.2 Environment

- **Observation** : The observation visible to the agent is a (*numCities + 1*) length integer array, where
  - ***0*** $\longrightarrow$ city (at that index position) is yet to be visited
  - ***1*** $\longrightarrow$ city has been visited
  - The first element is the index of the current city.

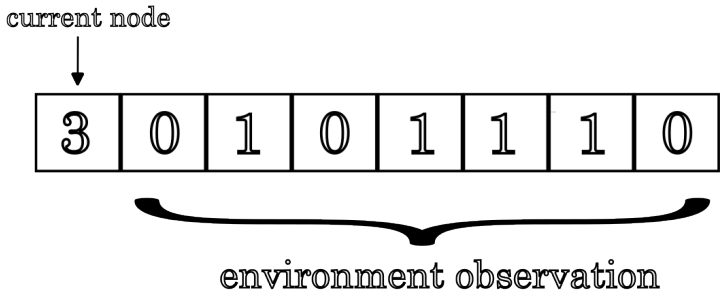



Figure 13: Visual representation of the observation

- **Reward**
  - a visit to an already visited node results in a large negative reward, to discourage the agent to perform a similar action in the future. It is also possible to mask the visited nodes and speed up the process, but a simpler implementation will highlight the reward system workings.
  - a visit to a new node instead will reward the agent with the cost of the trip between nodes. Depending on the chosen policy, it is also possible to calculate the reward as follows $reward = 1/(tripcost)$, so that shorter connections will have more weight.
- **Action Space** : the agent can only take actions in a discrete space, which can also be described as [ *0, numCities* ). Action $x$ will take the agent to node $x$.

### 7.2.3 Agent

The agent's main structural components, are the three following dictionaries

```
class AgentTSP:
    def __init__(self):
        self.env = TSPDistCost() #set the environment
        self.state = self.env.reset() #reset the state
        self.rewards = collections.defaultdict(int) # rewards table
        self.transits = collections.defaultdict() # transitions table
        self.values = collections.defaultdict(float)  # value table
```

- `self.rewards` $\longrightarrow$ **key** : $\rightarrow$ tuple $<$ *state, action, state'* $>$
  **value** $\rightarrow$ reward
  While exploring the environment, will memorize what reward to expect from an action when in a specific state.
- `self.transits` $\longrightarrow$ **key** : $\rightarrow$ tuple $<$ *state, action* $>$
  **value** $\rightarrow$ *state'*
  The *transits* dictionary is more relavant in an environment where action choice is not deterministic but stochastic. The agent may choose $action_1$, but there is a probability $p$, generally unknown, that it will perform a different action $a_2$. To keep track of this non-determinism, transits dictionary enable the agent to approximate the probability and calculate state and action values more accurately.
  A valid example is proposed in the *FrozenLake* environment ([4]). The agent may choose to go **right**, but since it is a *frozen lake*, it may slide elsewhere. The distribution of this probability can then be extracted with a sufficient amount of environment exploration.
  Despite this specific TSP environment being deterministic, it was still included because realistic contexts are usually affected by a stochastic component.
- `self.values` $\longrightarrow$ **key** $\rightarrow$ *state*
  **value** $\rightarrow$ *state value*
  The *values* dictionary will be updated during the ***value iteration*** process, with the aid of *Bellman's equations* discussed in 2.3.4.

### 7.2.4 Training

- *Environment exploration*

```
def play_n_random_steps(self, count):
    for _ in range(count):
        #action is randomly sampled from the \textit{action space}
        action = self.env.action_space.sample()
        #current state (observation) is retrieved
        state = self.state.tolist()
        state_tuple = tuple(state)
        #action is taken and the environment will return
        #(1) the new target state
        #(2) the reward acquired
        #(3) a boolean "done" to signal whether the new state is final.
        new_state, reward, is_done, _ = self.env.step(action)
        new_state_tuple = tuple(new_state.tolist())
        #the agent will update its dictionaries and new state
        self.rewards[(state_tuple, action, new_state_tuple)] = reward
        self.transits[(state_tuple, action)] = new_state_tuple
        self.state = self.env.reset() if is_done else new_state
```

The agent must explore the environment to garner some form of knowledge of the context. For instance, for 7 nodes, it was observed that at least 10'000 random steps had to be taken for the agent to adequately populate the dictionaries.
As the number of nodes increases, the amount of necessary steps explodes exponentially, making it computationally unfeasible. Also, it should be noted that this is applied for a single graph instance, and is not able to generalize on random graphs.

- *Value Iteration algorithm*
It should be noted that the chosen state-value is the *minimum*, because of the reward design choice made in 7.2.2

```
def value_iteration(self):
    # loop over all explored states of the environment
    for state in self.transits.keys():
        #for every state we calculate the values of the states
        #reachable from the current state.
        for action in range(self.env.action_space.n):
            #calc_action_value will update the action value
            #according to the current values dictionary
            state_values = [self.calc_action_value(state, action)]
        #select non-negative min value
        min = 100000
        for val in state_values:
            if val<0:
                continue
            else:
                if val < min:
                    min = val
        self.values[state] = min
```

The `calc_action_value` function will compute the value of the action when in a certain state $s$ by exploiting the agent's dictionaries.

```
def calc_action_value(self, state, action):
    action_value = 0.0
    #target state is single, deterministic environment
    tgt_state = self.transits[state, action]
    #retrieve the reward for the current [s,a,s'] thruple
    reward = self.rewards[(state, action, tgt_state)]
    #calculate the updated action value with bellman equation
    #nb, Q(s) =  [immediate reward + discounted value for the target state]
    action_value += (reward + GAMMA * self.values[tgt_state])
    return action_value
```

When in an episode, the agent will choose the action that minimizes the *state-action* value, which translates to an overall shorter path.

### 7.2.5 Experiment results

The agent will then play episodes forever until it cannot find a better path with the acquired knowledge. In the following example, the agent has found two solutions before it becomes idle, because of lack of better results.

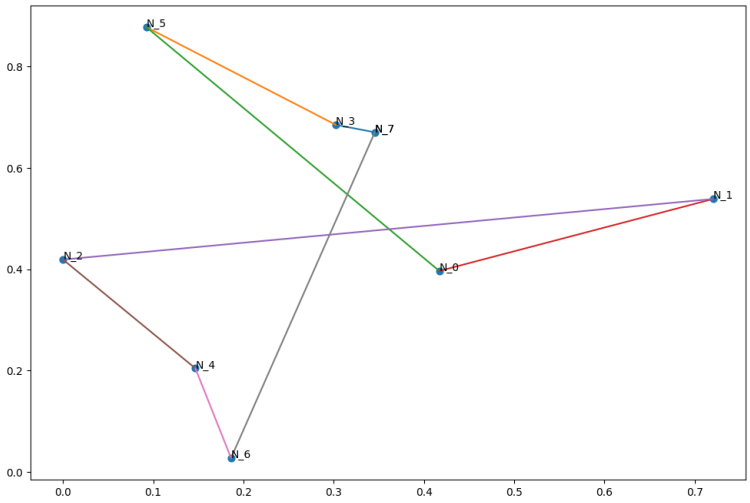



Figure 14: Path length : 3.3205104883664553 → [4, 0, 3, 7, 5, 2, 6, 1, 4]

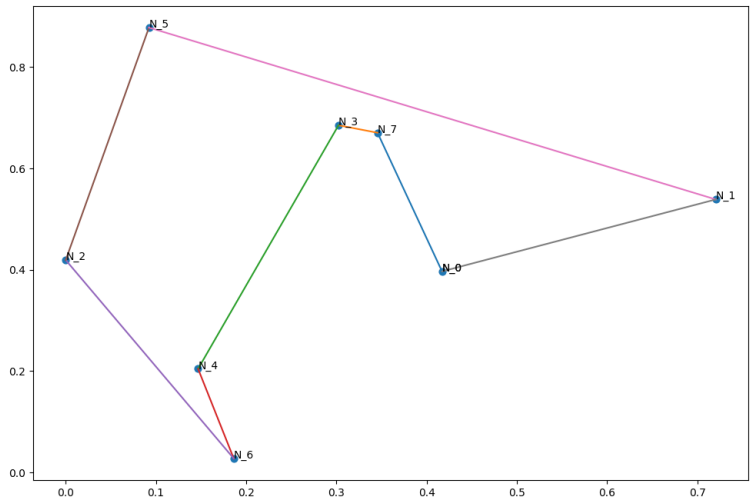



Figure 15: Path length : 2.9658626305269333 → [0, 7, 3, 4, 6, 2, 5, 1, 0]

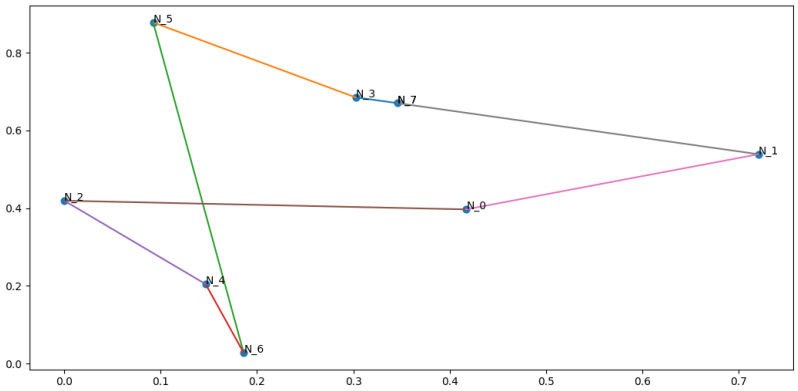



Figure 16: Path length : 2.7778151235560005 → [7, 3, 5, 6, 4, 2, 0, 1, 7]

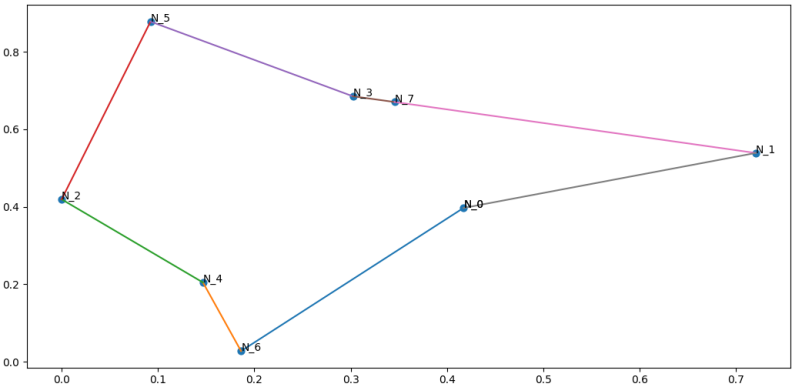



Figure 17: Path length : 2.408040871947452 → [0, 6, 4, 2, 5, 3, 7, 1, 0]

### 7.2.6 Experiment conclusions

The *Value Iteration* algorithm was here mostly implemented to confirm the feasibility of a TSP solver with Reinforcement Learning. The agent is able to solve TSP with nearly optimal solutions, but the node constraint is too binding and does not leave space for further development. The maximum number of nodes I was able to train the agent, on a single graph instance, is

8. Above this number the amount of exploration required to visit all possible states becomes unpractical and not a valid choice to solve TSP.

In this section it was decided to include the code, mostly because it is very simple to read and can give more immediate proof of the power of *Bellman's equations*.

## 7.3 Deep Q learning for TSP

### 7.3.1 Preface

As discussed several times, the problem with value iteration was that the potential ***observation space***, even when discrete, was generally too ample. *Deep Neural Networks* are a valid tool to tame the exploding numbers typical of *combinatorial* and *NP-hard* problems. They have been explored and described in 4.2.

### 7.3.2 Tools

- ***OpenAi GYM***
- **TensorFlow** framework

### 7.3.3 Environment

The environment, state, observation and action space are identical to the Value Iteration model. The number of nodes has been increased to 15 nodes, to confirm whether Deep Learning can be exploited to overcome *Value iteration*'s limits and constraints. I reckon it is relevant to highlight again the reward system, already touched in 7.2.2.

- *Invalid action cost* : -100 : a visit to an already seen node will be highly discouraged in the future
- The reward from city to city is $(1/distance)$. Smaller connections will be rewarded more than longer trips. This will be also more clear in the training plots.

### 7.3.4 Agent

The model is equipped with two neural networks, for the reasons mentioned in 4.2. TensorFlow is equipped with a predefined Deep Q-Network class, which comes in extremely handy for this experiment. The model structure is the following.

```
from rl.agents import DQNAgent
from rl.policy import BoltzmannQPolicy
from rl.memory import SequentialMemorydef

def build_model(states, actions):
    model = tensorflow.keras.Sequential()
    model.add(Flatten(input_shape=(1,states)))
    model.add(Dense(256, activation='relu'))
    model.add(Dense(256, activation='relu'))
    model.add(Dense(actions, activation='linear'))
    return model
```

```
def build_agent(model, actions):
    policy = BoltzmannQPolicy()
    memory = SequentialMemory(limit=50000, window_length=1)
    dqn = DQNAgent(model=model, memory=memory, policy=policy,
                   nb_actions=actions, nb_steps_warmup=2000, target_model_update=1e-2)
    return dqn
```

### 7.3.5 Training

The algorithm chosen was the basic *Deep Q-Learning with experience replay*. A visual more immediate representation is available at 5. Here follows the pseudocode, taken from [11].

**Deep Q-Learning with experience replay**

Initialize replay memory buffer **RB** to length N
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with random weights $\hat{\theta} = \theta$
**for** episode = 1, M
  **for** $t = 1, T$:
    with *decaying* probability $\epsilon$
      Select random action $a_t$
      Select $a_t = argmax_a Q(s, a, \theta)$
    Perform step in environment, execute action $a_t$
    Observe $s_{t+1}$ , $r_t$, *done*
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in buffer **RB**
    Sample batch of transitions from **RB**.
    $$y_t = \begin{cases} r_t & \text{if } s_{t+1} \text{ is terminal} \\ r_t + \gamma max_{a'} \hat{Q}(s_{t+1}, a', \hat{\theta}) & \text{otherwise} \end{cases}$$
    Perform gradient descent on the error $(y_t - Q(s_t, a_t, \theta))^2$
    with respect to network parameters $\theta$
    Every C steps copy weights to target network $\hat{Q} = Q$
  **end for**
**end for**

### 7.3.6 Experiment results

Both the plots highlight the agent's predicted behaviour. It performs random actions at the beginning, discovering the intricacies of the environment and when the $\epsilon$ goes to a low enough value it will start performing future-conscious decisions based on its growing knowledge.

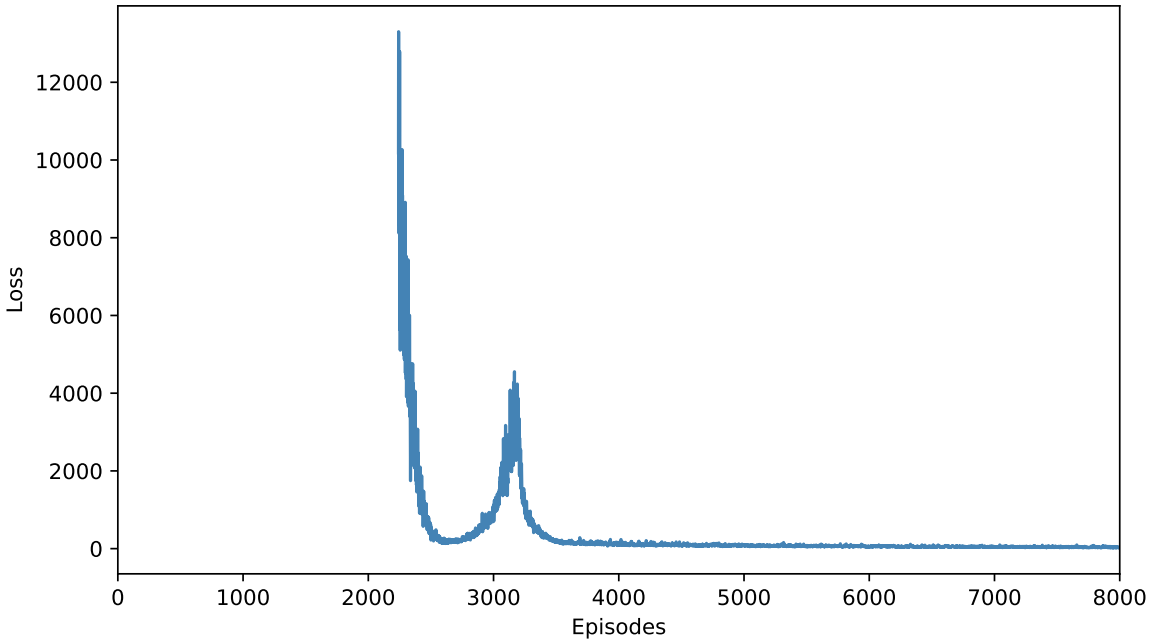



Figure 18: Loss over episodes

In the first 2000 episodes the gradient diverges and therefore cannot be represented. It start converging from around 3500 episodes played.

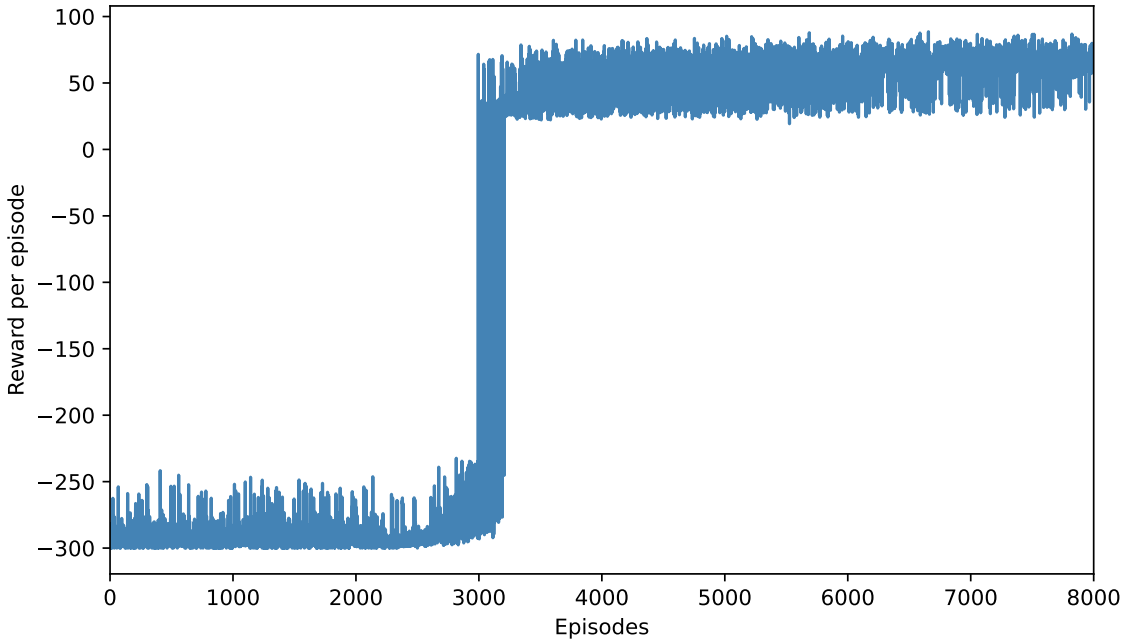



Figure 19: Average total reward per episode

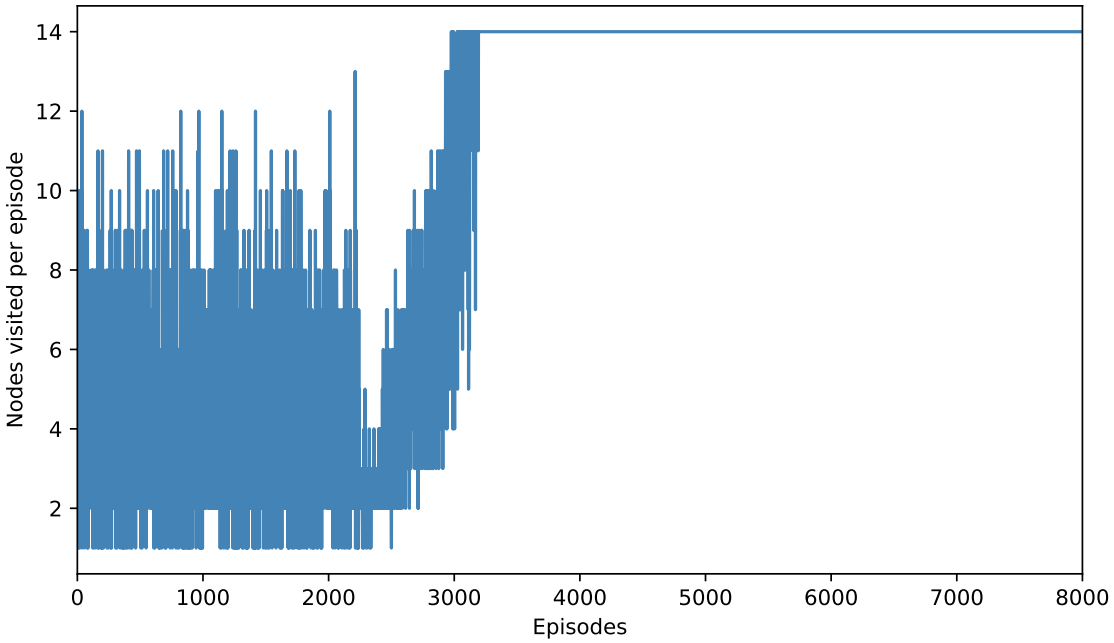



Figure 20: Nodes visited per episode

### 7.3.7 Experiment conclusions

This model has confirmed, as predicted, that Deep Neural Networks can be exploited to scale up the problem coming from a purely tabular application of Reinforcement Learning principles. The experiment here was done on 15 nodes, so it is also possible to confirm the fact this is an improvement over the *Value iteration* algorithm.

The next step is trying to generalize the problem on any random graph instance with a more sophisticated agent and neural network model, whose architecture and reasoning has been explained in 5. Considering that the state observable by the agent is the same one as the *Value Iteration* experiment, it is remarkable it can converge with such little information.

## 7.4 Variation on Bello's model

### 7.4.1 Preface

Bello's paper and implementation of a neural network have been dissected in depth in 5 and in particular in 5.2, therefore I will skip directly to the variation here implemented.

### 7.4.2 Tools

- The *Github* repository with an implementation of Bello's model, available at the appendix A.2 .
- The computing power was part of the resources offered to Politecnico di Torino students. More info about the cluster is available at the appendix A.2. It was possible to run multiple parallel instances, but the lack of GPUs (as of June 2022) made certain processes more cumbersome and lengthy.
- The code implementation for the *autoencoder* is available also in the appendix A.1.2.

### 7.4.3 Autoencoder Variation

The main difference has been introduced in 6. Instead of feeding the neural network only with the node coordinates, a multidimensional representation of the nodes, generated with *autoencoders* is the new input. The dimensionalities chosen were

- 32 dimensions per node for 20,50 node instances
- 64 dimensions per node for 100 node instances.

Comparison tests were run on almost identical hyperparameters and configuration, in order to be able to determine the magnitude of the effect of autoencoders in such a problem.

It is also possible that the new input is inadvertedly feeding the net with better initial weights, since the loss behaviour is mostly the same from a macroscopic point of view, with a speed inflection in the first 1500 steps.

Running on the same machine, with no other processes running, the results are the following.

# 20 nodes

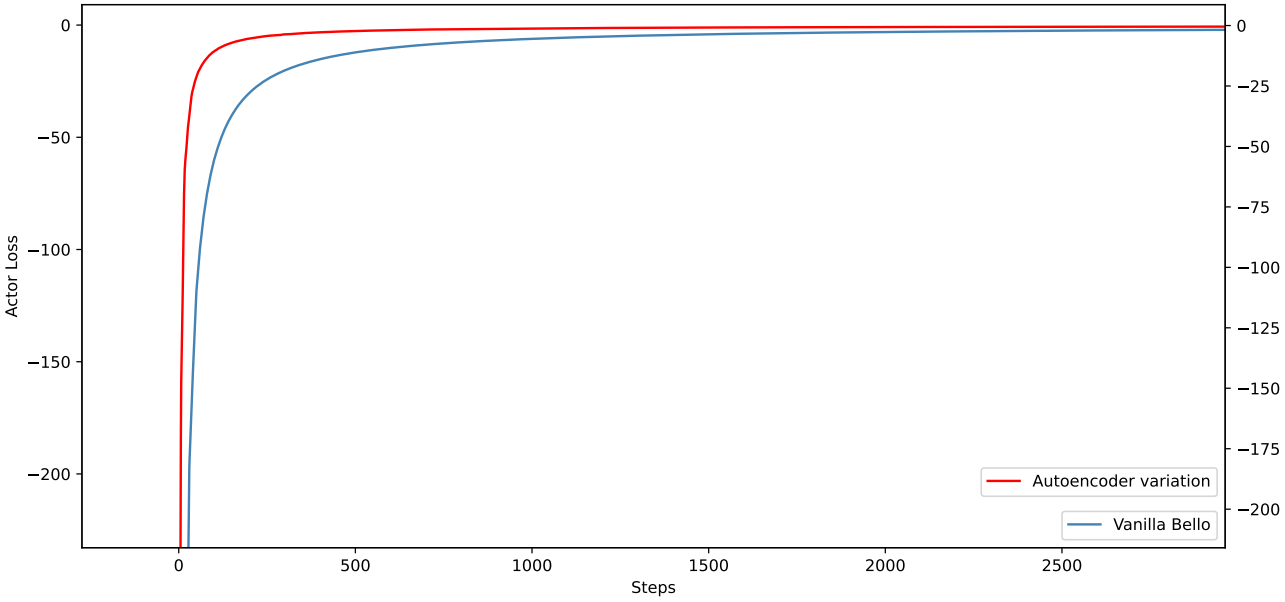



Figure 21: Actor Loss comparison

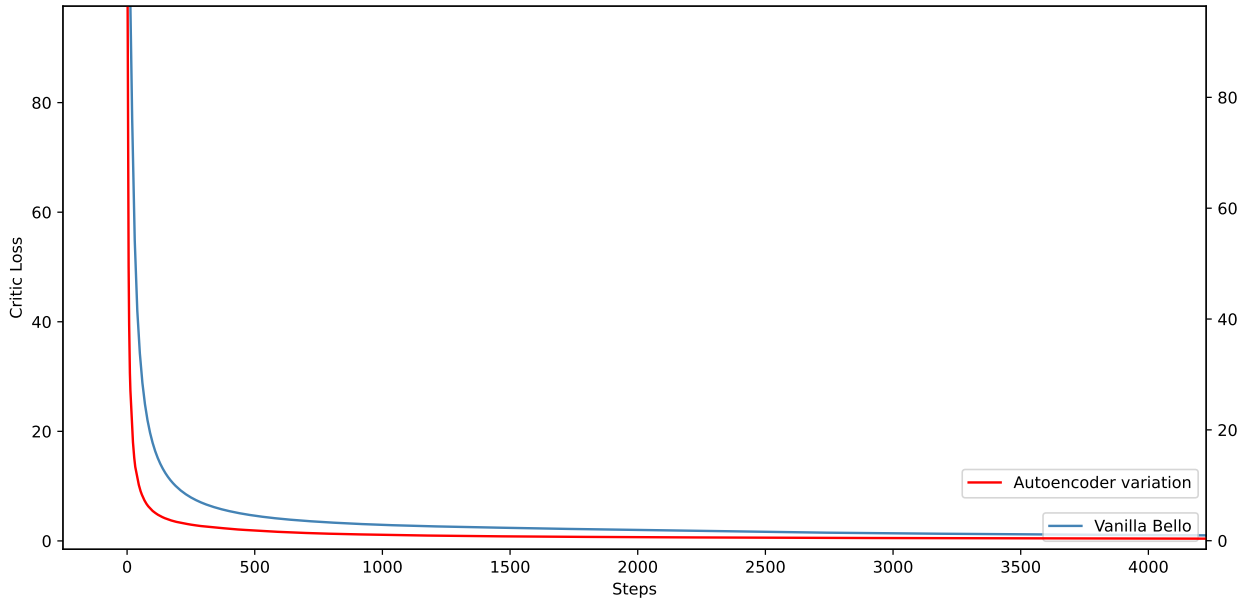



Figure 22: Critic Loss comparison

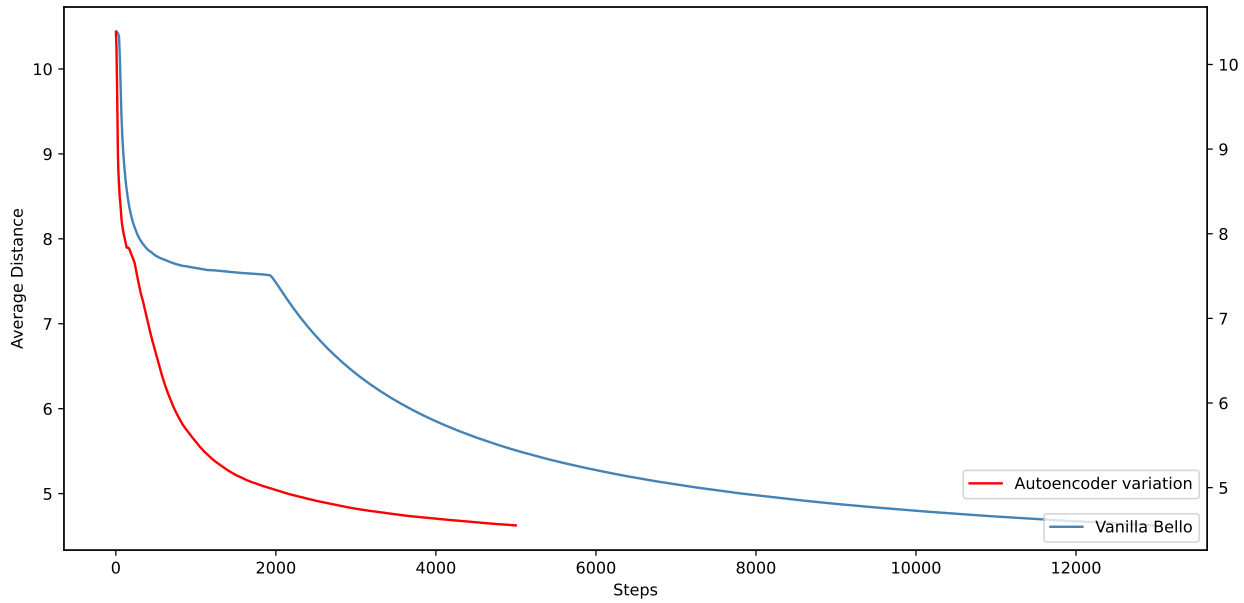
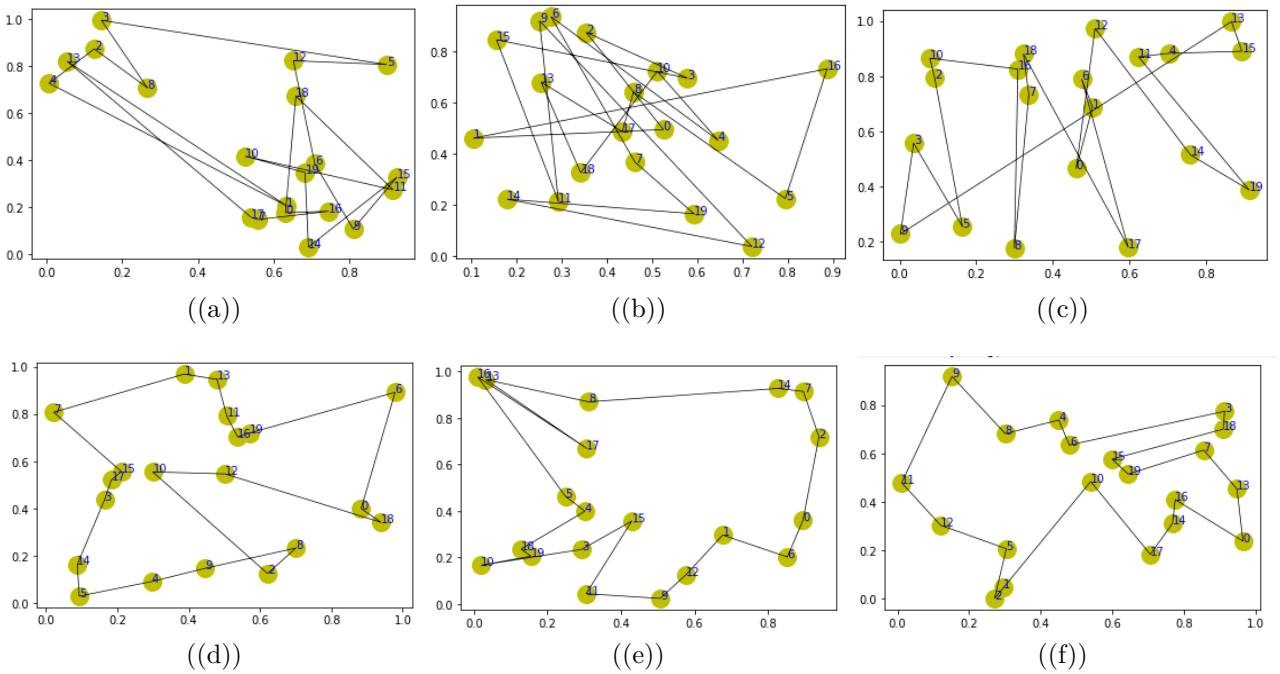



Figure 23: Average trip length comparison

# Model Progress



((a)) ((b)) ((c))

((d)) ((e)) ((f))

Figure 24

# 50 nodes

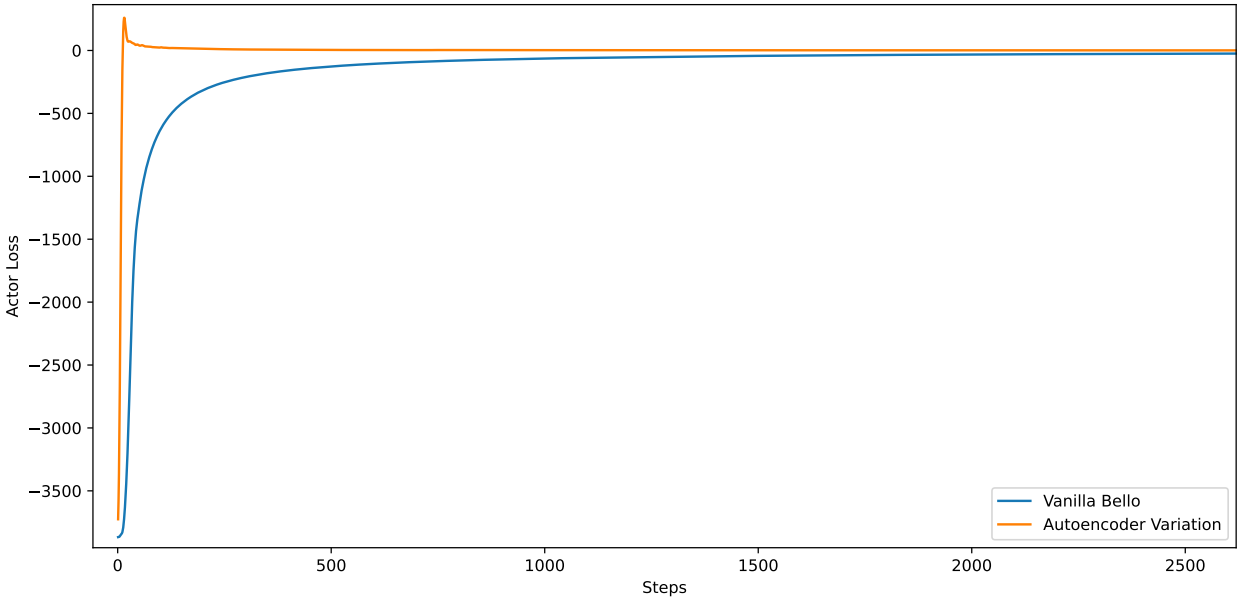



Figure 25: Actor Loss comparison

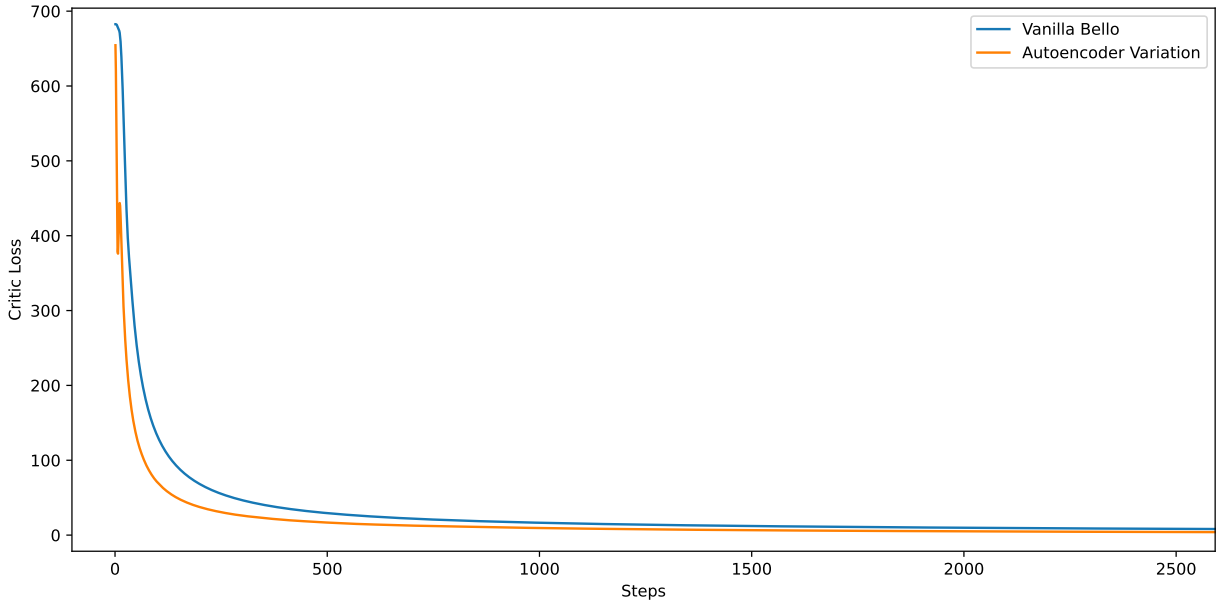



Figure 26: Critic Loss comparison

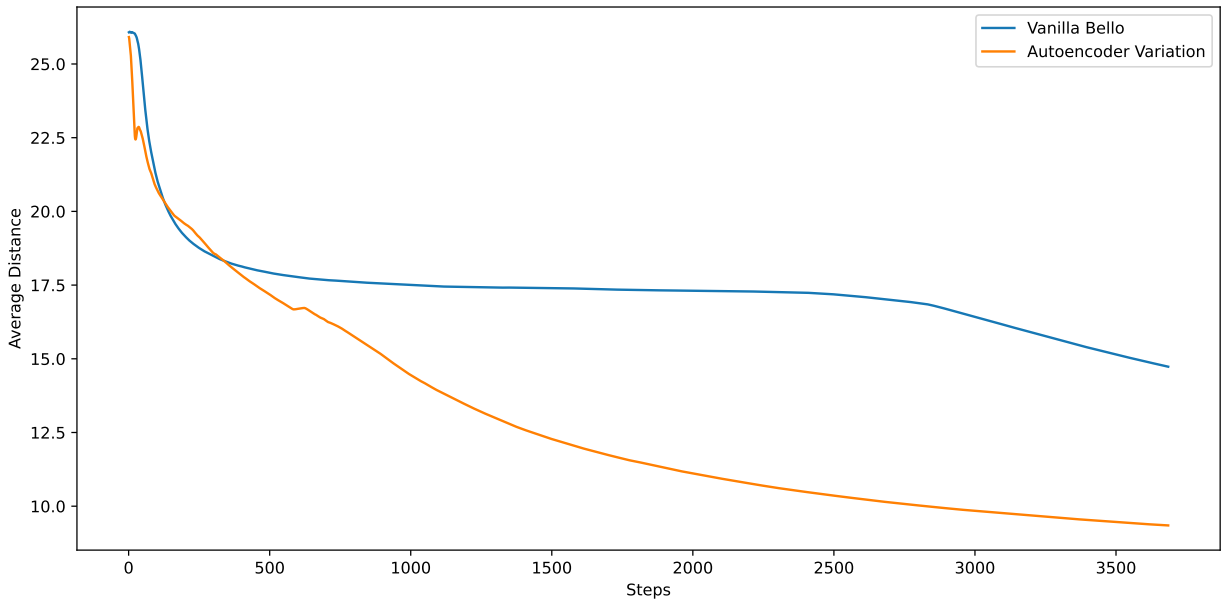
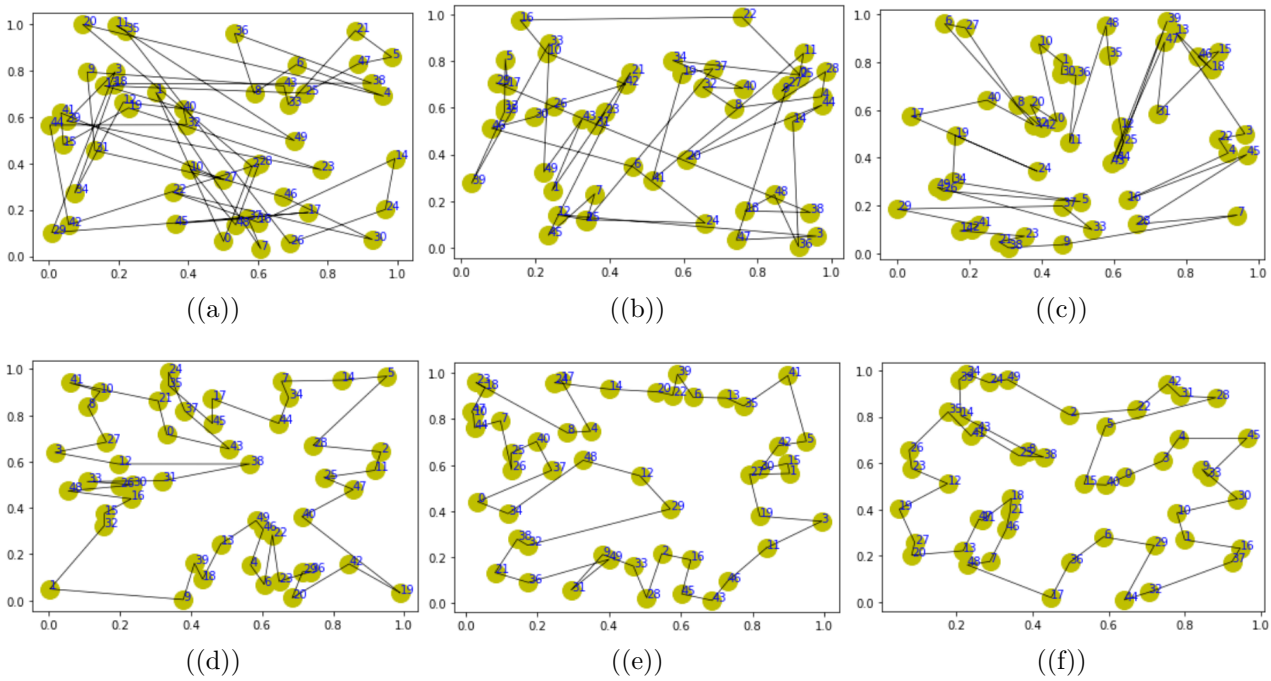



Figure 27: Average trip length comparison

# Model Progress



((a)) ((b)) ((c))

((d)) ((e)) ((f))

Figure 28

# 100 nodes

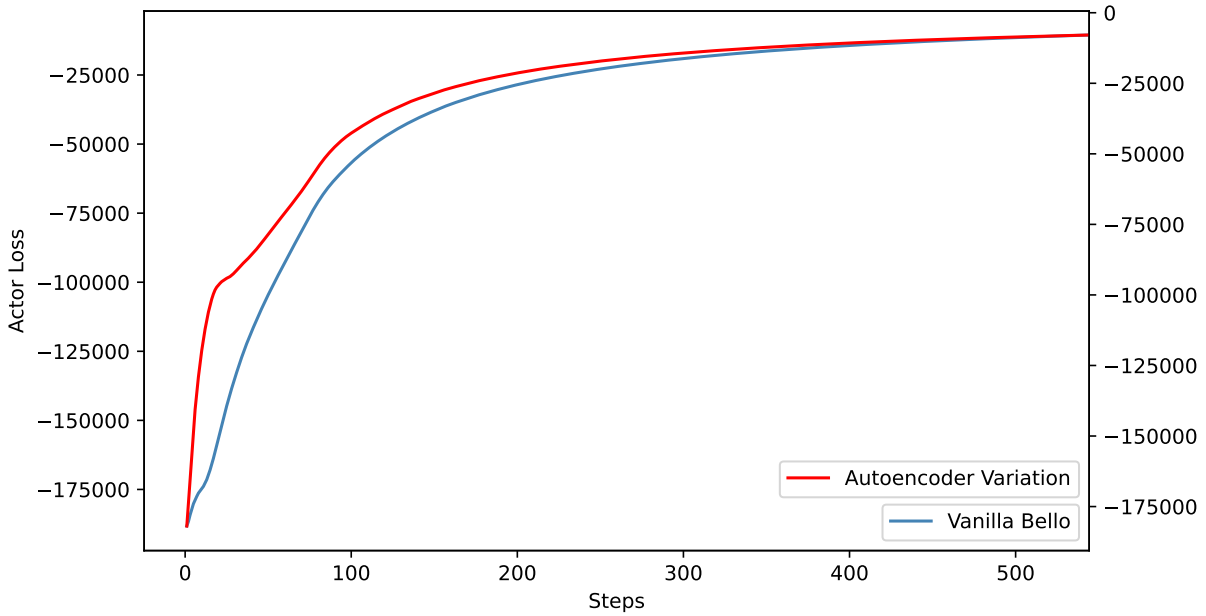



Figure 29: Actor Loss comparison

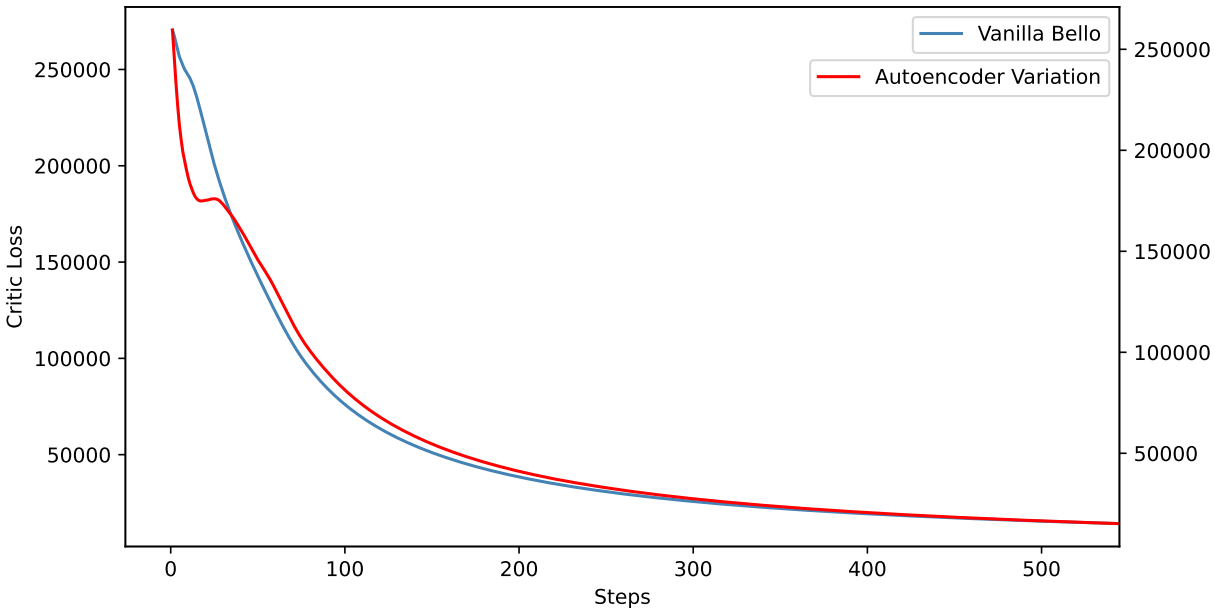



Figure 30: Critic Loss comparison

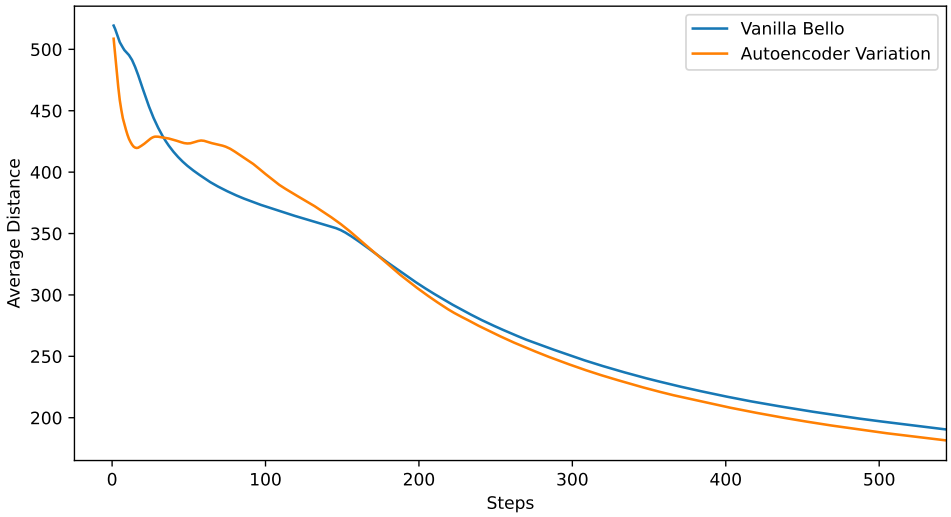
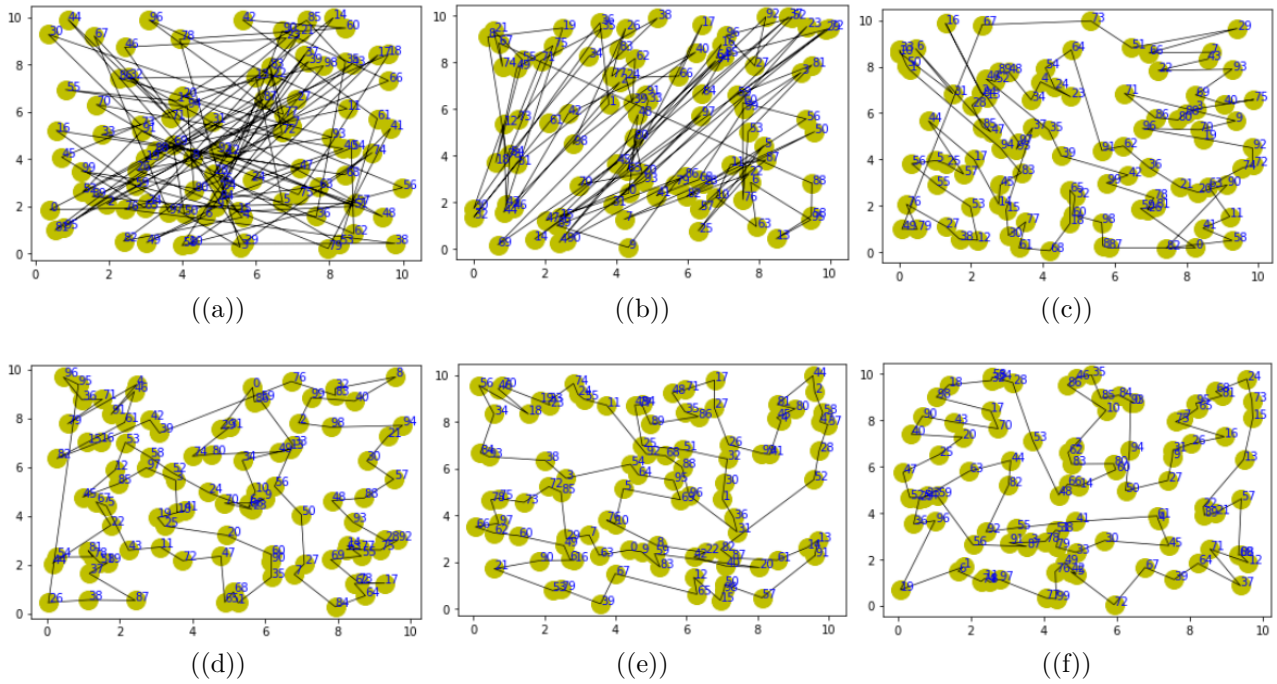



Figure 31: Average trip length comparison

# Model Progress



((a)) ((b)) ((c))

((d)) ((e)) ((f))

Figure 32

### 7.4.4 Experiment conclusions

Tangible results are more noticeable in the 20 and 50 node instances, the results are not groundbreaking, but with enough optimization and possibly a more elaborated structure to represent it is possible the benefits could be more appreciable and noteworthy.

Different encoding techniques can be tested in the future to determine the most information-rich representation that can be extracted.

## 7.5 Overall experiments conclusions

- Practical tests were fundamental to translate theoretical principles to working models. The main objective was to confirm that TSP is a problem that can be approached with *Reinforcement* and *Deep Learning* techniques. I reckon it is relevant to show the growing proficiency and evolution of the models and the rationale behind choices and how this design approach can be implemented for other projects.
- It is clear that the *Value Iteration* (7.2) and the *Deep Q-Network* (7.3) experiment were not optimized; moreover, their plots may seem inferior in quality compared to Bello's model, but this should only be more proof that a more specialized and thought out architecture can result in immensely better results.

# 8 Conclusions

- The development of this document was made possible with the precious input from the supervising professors, who directed me towards a structured and organic progress in the model building procedure.
- This document's aim is to organically describe a Reinforcement Learning model development, from the theoretical principles to the practical code, and offer a possible source of later advancement.
- I personally found the process captivating because it made the model building procedure understandable and it highlighted the inner workings of neural networks, which can sometimes just feel like an incomprehensible black box. A more in-depth look at the models analyzed in these months instead brought more sense and reasoning to the design choices cited from the references.
- It should be reminded that the *Travelling Salesman Problem* has been analyzed for decades and there are several heuristic algorithms able to solve it. I find the Deep Learning approach fascinating because the potential for the models to be more improved and sophisticated is visible. A variation of this model could be used for a large scale *Vehicle Routing Problem* (**VRP**) with dynamically changing weights and priorities, or for the optimization of industrial sites with moving machines.
- The fact that an NP-hard problem such as TSP can be now solved with nearly optimal results with machine learning techniques is remarkable. It shows how problems once considered unfeasible if scaled up can now be tamed and could require less energy and time to be solved with the rise of Neural Network system-on-chip's.

# A Appendix

## A.1 Code Implementation Appendix

*Here less relevant code snippets are made available for a more comprehensive view. Full code is available on the **Github** link at the end of the document.*

### A.1.1 Value Iteration

**Environment**

```
class TSPEnv(gym.Env):
    def __init__(self, *args, **kwargs):
        self.N = 7
        self.invalid_action_cost = -100
        self.mask = False
        utils.assign_env_config(self, kwargs)
        self.nodes = np.arange(self.N)
        self.coords = self._generate_coordinates()
        self.distance_matrix = self._get_distance_matrix()
        self.obs_dim = 1 + self.N
        obs_space = spaces.Box(-1, self.N, shape=(self.obs_dim,), dtype=np.int32)
        self.observation_space = obs_space
        self.action_space = spaces.Discrete(self.N)

        self.reset()
```

### A.1.2 Autoencoder

```
import torch
import torch.nn as nn
import torch.optim as optim

class AutoEncoder(nn.Module):
    def __init__(self, **kwargs):
        super().__init__()
        self.encoder_hidden_layer = nn.Linear(
            in_features=kwargs["input_shape"], out_features=64)
        self.encoder_output_layer = nn.Linear(
            in_features=64, out_features=64)
        self.decoder_hidden_layer = nn.Linear(
            in_features=64, out_features=64)
        self.decoder_output_layer = nn.Linear(in_features=64,
            out_features=kwargs["input_shape"])
    def encode(self, x):
        activation = self.encoder_hidden_layer(x)
        activation = torch.relu(activation)
        code = self.encoder_output_layer(activation)
        code = torch.relu(code)
        return code
```

```
def decode(self, code):
    activation = self.decoder_hidden_layer(code)
    activation = torch.relu(activation)
    activation = self.decoder_output_layer(activation)
    reconstructed = torch.relu(activation)
    return reconstructed

def forward(self, features):
    encoded = self.encode(features)
    reconstructed = self.decode(tmp)
    return reconstructed
```

In this appendix were included portions of code deemed relevant to better understand certain passages. The repository associated to this document with all of the coding implementation can be found at the following link

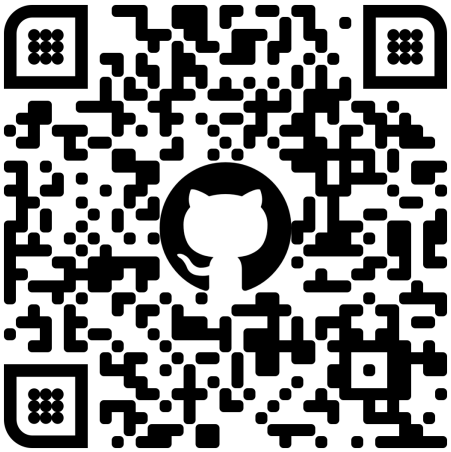


Figure 33: github.com/arya-h/DL_RL_TSP_AH

## A.2 Resources and repositories

- Github repository with implementation of [3] : Repository link
- Cluster available to PoliTo students. PoliTo's Cluster information link

# References


[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2014.

[2] Bram Bakker. Reinforcement learning with long short-term memory. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14. MIT Press, 2001.

[3] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016.

[4] Paolo Dell'Aversana. The frozen lake problem. an example of optimization policy, 12 2021.

[5] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. *CoRR*, abs/1607.00653, 2016.

[6] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1999.

[7] Vijay R. Konda and John N. Tsitsiklis. Actor-critic algorithms. pages 1008–1014. 2000.

[8] Maxim Lapan. *Deep Reinforcement Learning Hands-On*. Packt Publishing, Birmingham, UK, 2018.

[9] Michael Mccloskey and Neil J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. *The Psychology of Learning and Motivation*, 24:104–169, 1989.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 2013. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[12] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. *CoRR*, abs/1403.6652, 2014.

[13] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[15] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks, 2015.

[16] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.