UCLouvain

École polytechnique de Louvain

Combinatorial Optimization using Recurrent Neural Network and Reinforcement Learning

Author: **Quentin Deffense**
Supervisor: **Pierre Schaus**
Readers: **Guillaume Derval, Siegfried Nijssen**
Academic year 2019–2020
Master [120] in Computer Science and Engineering

Abstract

The Combinatorial optimization problems are traditionally solved with handcrafted heuristics. Finding these heuristics for an unseen or a poorly solved problem can be a challenge and the engineer cost can be high. On the other hand, Machine Learning models yield incredible results in many fields like text categorization, image processing, speech recognition and many others. The recent idea of using Machine Learning models to learn heuristics for Combinatorial optimization problem can save human engineer cost.

In this thesis, we explain in more depth the framework introduced by Bello et al. in *Neural Combinatorial Optimization with Reinforcement Learning*. Their work uses Recurrent Neural Networks and Reinforcement Learning to learn heuristics. The model uses LSTM encoder and decoder, along with Neural Attention mechanisms. The model is trained using the REINFORCE algorithm with a Critic Network as baseline. To obtain better results, some search strategies are also used. Since their paper is made for experienced people, we give a more complete description of the components of the model, such that someone without knowledge about Recurrent Neural Network could understand it. We also try to reproduce their results on the famous Traveling Salesman Problem but with three simple modifications of their model and one more search strategy, 2opt. This strategy is used to improve the performances since the model doesn't necessarily produce local optima solutions. As Bello et al., we compare our results against the OR Tools and Concorde solvers but we also add simpler baselines with Nearest Neighbor and 2opt. We also test if the model is able to generalize to variable problem sizes.

Acknowledgements

I would like to thank my supervisor Professor Pierre Schaus for his relevant remarks and his advises throughout my thesis.

I would also like to thank Yerzat Dulat (Higgsfield) for allowing me to use his code.

Finally, I would like to thank my family and friends for their endless support, especially Emy Scheyvaerts.

Contents

1 Introduction

Motivation : Machine learning has helped to reduce human engineer cost by replacing handcrafted features in many fields such as speech recognition, machine translation, image processing and many others. Combinatorial Optimization is a fundamental problem in computer science and is traditionally handled by two different approaches. The first one, the exact methods, guarantees finding the optimal solution. The second one, the approximate methods, trades off optimality for computational cost. Both of them use carefully handcrafted heuristics to search among the space of feasible solutions in an efficient manner. The challenge is the difficulty in applying well-known handcrafted heuristics to unseen problems or to new variants of a problem (see No free lunch theorem [25]).

In this thesis, we try to address this problem by building the framework introduced by Bello et al. [3] using Neural Networks and Reinforcement Learning. This model, based on policy gradient, learns a policy that replaces the handcrafted heuristics. The model's architecture is based on the Pointer Network [20] which is a sequence to sequence [18] architecture using an encoder LSTM [8] and a decoder LSTM. The Pointer Network also uses a mechanism of neural attention [2] to select a member of the input as output. The model uses an additional computation step [19], named glimpse, to complement the attention mechanism. We just make three simple changes to the original model, as proposed in Higgsfield's code [7]. To train the model (as done by Bello et al. [3]), we use the REINFORCE algorithm with two different baseline functions. The first one is a simple exponential moving average while the other is a parametric baseline that acts as a critic network. While the network can be trained on a train set and then used the pretrained model to optimize a test set, it can also be trained directly on a single test instance which is called Active Search. Another option is to use a combination of both ways and apply the Active Search on a pretrained model. Sampling candidates outputted by the trained model can also improve the results. These search strategies proposed by Bello et al. [3] are efficient but do not necessarily reach local optima. To reach it, we add a 2opt local search strategy.

To check the performances of this framework we focus on the Traveling Salesman Problem (TSP). The TSP is a NP-hard problem in Combinatorial optimization with the following statement [23] : " given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city? ". The TSP can be applied to many fields (such as logistics or planning) in its pure form and to much more if slightly modified. Since it is one of the most studied problem, we can compare our results to a panel of different methods.

In this document, we will first describe the TSP and the baselines that we will use to compare the performances of our model. Then, we will explain all the models needed to build it. In the fourth chapter, we will give a complete description of our model's architecture and the differences between the model of Bello et al. [3] and ours. We will later see why and how to use Reinforcement Learning for training. We will also talk about the search strategies to reach better performances. Finally, we will show the results and analyze them. We also test if the model trained on a certain size is able to generalize to other sizes.

2 TSP

2.1 Formal description

For this thesis, we focus on the 2D Euclidian TSP. Given a sequence s of n cities, in a two dimensional space $s = \{x_i\}_{i=1}^{n}$ where $x_i \in \mathbb{R}^2$, the goal is to find a permutation of the cities π, named tour, that visits each city only once and has a length as low as possible. We define the length of a tour π of a given set s as

$$L(\pi|s) = \sum_{i=1}^{n-1} ||x_{\pi(i)} - x_{\pi(i+1)}||_2 + ||x_{\pi(n)} - x_{\pi(1)}||_2$$

where $x_{\pi(i)}$ is the ith city of the tour π and $||\dots||_2$ is the l_2 norm.

2.2 Baselines description

In this section, we will briefly describe the other solvers that we use as baselines to compare our results. While OR Tools and Concorde were also used by Bello et al. [3], we also use the Nearest Neighbor and the 2opt algorithms to have simpler baselines.

2.2.1 Nearest Neighbor

The Nearest Neighbor algorithm [22] is one of the most simple way to construct a tour but it can miss easy shorter routes. In the Nearest Neighbor (see Algorithm 1), we start with an arbitrary city and select the nearest city to add to the tour. Then we select the city that is the nearest from the last city selected until all the cities are in the tour.

Algorithm 1: Nearest Neighbor

input : s, a sequence of n cities
output: π, a solution tour

$current \longleftarrow SelectRandomIn(s)$
$Rest \longleftarrow s - \{current\}$
$\pi \longleftarrow \{current\}$
while $Rest \neq \emptyset$ **do**
  $next \longleftarrow InRestSelectNearestFrom(current)$
  $\pi \longleftarrow \pi + \{next\}$
  $Rest \longleftarrow Rest - \{next\}$
  $current \longleftarrow next$

2.2.2 2-opt

The 2-opt[5] is a local search algorithm used to improve an initial solution. Therefore, it can be paired with another algorithm that would create that initial solution. The search is based on a swapping mechanism (see Algorithm 2). The swapping mechanism consists of swapping two cities and to reverse the order of all the cities between them. At each step of the search, we try to find a swap that improves the length of the tour. If one is found, we swap the cities and, if none are found, we stop the search. A time limit can also be implemented to stop the search earlier.

Algorithm 2: 2-opt

input : π, a solution tour of n cities
output: π^*, an optimized solution tour

```
π* ⟵ π
improvement ⟵ True
while improvement is True do
    improvement ⟵ False
    for i ← 0 to n − 1 do
        for j ← i + 1 to n do
            if length if swap i and j < current length then
                π* ⟵ SwapCities(i, j)
                improvement ⟵ True
```

2.2.3 Concorde

Concorde [1] is widely accepted as one of the best exact TSP solvers[1]. It makes use of cutting plane and branch-and-bound algorithms. The cutting plane algorithm will solve a relaxation of the TSP by using linear programming. Without losing optimality, it will iteratively add inequalities to find a feasible solution. The branch-and-bound algorithm will compute upper bound of the length of the optimal solution and the lower bound of solutions within parts of the search space. If a part of the search space has a lower bound higher than the upper bound of the optimal solution, the optimal solution is not in it. It can then safely prune that part to have a smaller search space.

2.2.4 OR Tools

The vehicle routing solver from OR-Tools [6] gets a solution using Christofides' algorithm[4] and improves that solution with simple local search operators including 2-opt and the Lin-Kernighan heuristic[14], stopping when it reaches a local minimum[2]. To escape poor local optima, it can use different metaheuristics, such as simulated annealing, tabu search or guided local search. Even if not state-of-the-art for the TSP, OR-Tools can tackle a superset of the TSP and is a good trade-off between the simplicity of the basic local search operators and the complexity of the best solvers.

Christofides' algorithm obtains a solution in polynomial time and is guaranteed to be within a 1.5 ratio compared to the optimal solution. Let G be the graph induced from a TSP instance. It creates T, the minimum spanning tree of G, and then takes the set of cities O with odd degree in T. It finds M, a minimum weight perfect matching in the subgraph of G given by O. The solution is obtained by combining the edges of M and T and skipping the cities visited more than once.

As explained above, the 2-opt heuristic tries to swap two cities to optimize the solution tour. The Lin-Kernighan heuristic tries k successive swap and check if the final result improves the solution. With k, a parameter that is adapted.

[1]We tested our test instances thanks to https://github.com/jvkersch/pyconcorde
[2]We tested our test instances thanks to https://github.com/google/or-tools

3 From Feedforward Networks to Pointer Networks

The model of Bello et al. [3] is based upon the Pointer Network. But, to completely understand how the Pointer Network works and why it is useful, we need to take a step back and explain all the bricks needed to construct such a neural network. In this chapter, we explain all the modifications needed to the simplest neural network to have a Pointer Network.

In the first section, we will do a review of Feedforward Networks. We'll explain the model and then illustrate how it works with a small example. Then, with another example, we will highlight some of its limitations. In the next section, we show how to adjust the network to have a Recurrent Neural Network which will meet these limitations. We will describe this network and demonstrate its usefulness on the problematic example. By introducing a harder task, we will show that the Recurrent Network is still not enough and that another network has to be introduced to handle task with long-term dependencies. That will lead us to the Long Short-Term Memory (LSTM), described in the third section.

The LSTM is the main component of the Pointer Network. But, unfortunately, it can not handle sequence-to-sequence problems without a framework. This framework, called *seq2seq*, uses two LSTM, one to process (encode) the data in the input and one to produce the output.

All of these sections will allow us to tackle sequence-to-sequence problems but not necessarily in a optimized way, mostly on problem with long sequences. That's why we will need the Attention Model to "focus" on some parts of the input.

The last section of this chapter will explain how to put it all together to obtain the Pointer Network.

3.1 Review of Feedforward Networks

The Feedforward Network, also called Multi-Layer Perceptron (MLP), is the most basic neural network. Even if it is the simplest, the MLP is very useful in many situations and shows good performances. In this section, we will explain the general architecture of the MLP and a specific example of architecture. Then, we'll show how it works in practice to categorize 2x2 images. Finally, we will try to build a Feedforward Network to tackle a time series problem and we will see that the MLP is not suited for such a task.

3.1.1 The MLP Model

The MLP takes an input vector X and makes it go through a certain number of layers of weights to get the output vector Y. The first layer of weights takes the input vector X, multiplies it by a matrix of weights, adds a bias and uses an activation function to get its own output vector. The second layer uses this vector as input and produces its own output vector similarly. And so on until the last layer produces the final output vector Y.

More formally,

$$h_i = activation(W_i h_{i-1} + b_i) \qquad i \in \{1, ..., N\}$$

with $h_0 = X$ and, with N the number of layer, $h_N = Y$. *activation* is the activation function, see Figure 1 for examples of nonlinear activation functions. W_i, the weight matrix, and b_i, the bias vector, are parameters that are refined during training.

Sometimes, in the literature, a N-layer network is defined as a network with N layer of vectors (or *neurons*). Since X and Y are counting as layers of vector, a network with N layers of vectors will have

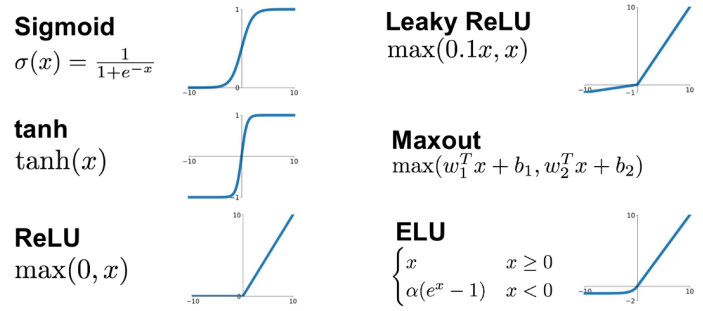


Figure 1: Examples of nonlinear activation functions [10]

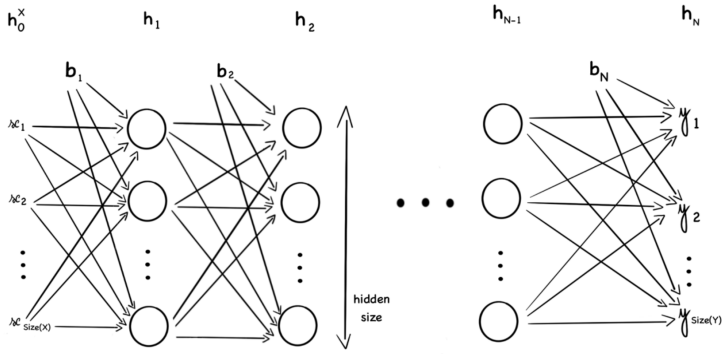


Figure 2: Feedforward Neural Network with N layers of same hidden size

N-1 layers of weight. But, in this thesis, we use the convention that a N-layer network is a network with N layers of weights.

In the most simple case, each layer of neurons has the same size, named *hidden size*, and the same activation function. The architecture of this neural network is represented in Figure 2.

For example, a 3-layer sigmoid neural network (see Figure 3) computes the output vector with the following formula :

$$Y(X) = \sigma\Big(W_3\ \sigma\Big(W_2\ \sigma(W_1\ X + b_1) + b_2\Big) + b_3\Big)$$

where

- σ is the sigmoid activation function
- X is the input vector, $X \in \mathbb{R}^{input\ size}$
- Y is the output vector, $Y \in \mathbb{R}^{output\ size}$
- W_i is the weights matrix of the ith layer of weights

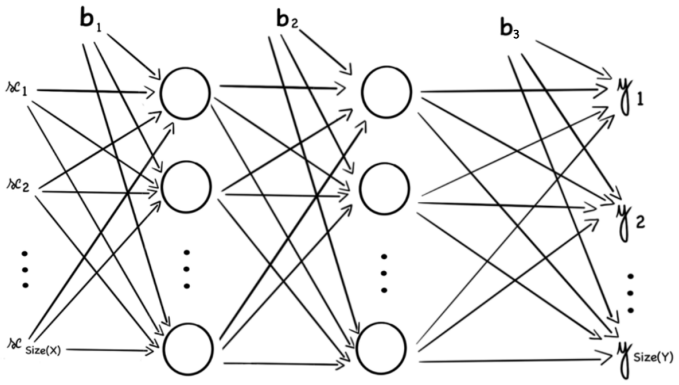


Figure 3: Feedforward Network with 3 layer of weights

- $W_1 \in \mathbb{R}^{hidden\ size\ X\ input\ size}$
- $W_2 \in \mathbb{R}^{hidden\ size\ X\ hidden\ size}$
- $W_3 \in \mathbb{R}^{output\ size\ X\ hidden\ size}$
- b_i is the bias vector of the ith layer of weights
- $b_1 \in \mathbb{R}^{hidden\ size}$
- $b_2 \in \mathbb{R}^{hidden\ size}$
- $b_3 \in \mathbb{R}^{output\ size}$

3.1.2 Example 1 : Categorizing 2x2 images

To show how the Feedforward neural network works in practice, we introduce a simple example of image pattern recognition presented in [17] (The two next examples also come from [17]). The goal is to classify a picture of 4 pixels into one of these 4 categories : solid, vertical line, diagonal line and horizontal line. For simplicity, the pictures are in black and white. The darkness of each pixel is represented by a number from -1 for black to 1 for white. We train a neural network with the following characteristics :

- 4 layers of weights
- the first two layers of weights use a sigmoid squashing function as activation function, the third uses a ReLU function and the last layer doesn't use any activation function
- no bias
- the size of the two first hidden layer of neurons is 4 and the size of the third is 8

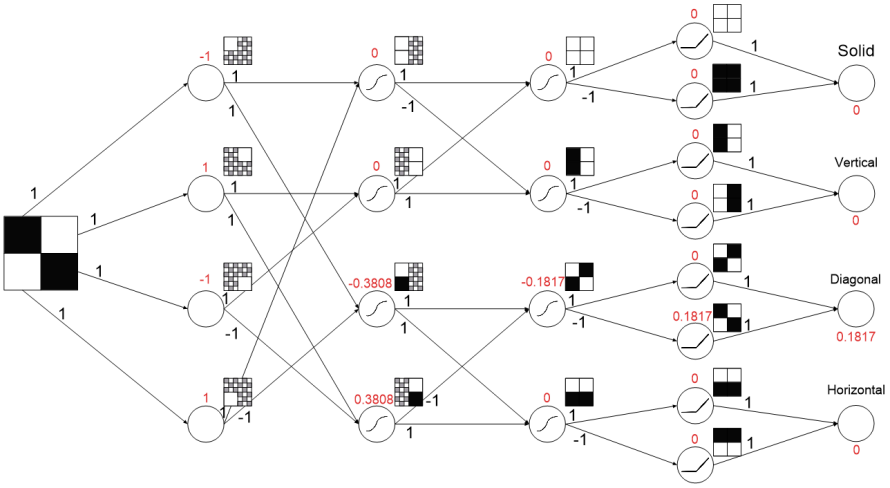


Figure 4: Categorizing a diagonal line with a Feedforward Neural Network

- the output vector is 4 real numbers representing respectively the solid form, the vertical line, the diagonal line and the horizontal line

Let's say that after the model was trained, the weights have these values :

$$W_1 = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix} \qquad W_2 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$$W_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \end{bmatrix} \qquad W_4 = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

In Figure 4 and 5, we show how this network works on two examples. The first is a diagonal line in only black and white. As shown bellow, the output vector only has value for the diagonal line pattern. The second example is a horizontal line in black, grey and white and the network outputs a more mitigated vector but the maximal value is clearly for the horizontal line pattern.

$$Input\ 1 \quad \begin{bmatrix} -1 & 1 & -1 & 1 \end{bmatrix} \qquad Input\ 2 \quad \begin{bmatrix} -1 & -1 & 0 & 1 \end{bmatrix}$$

$$Output\ 1 \quad \begin{bmatrix} 0 & 0 & 0.1817 & 0 \end{bmatrix} \qquad Output\ 2 \quad \begin{bmatrix} 0.0575 & 0.0575 & 0.0374 & 0.1484 \end{bmatrix}$$

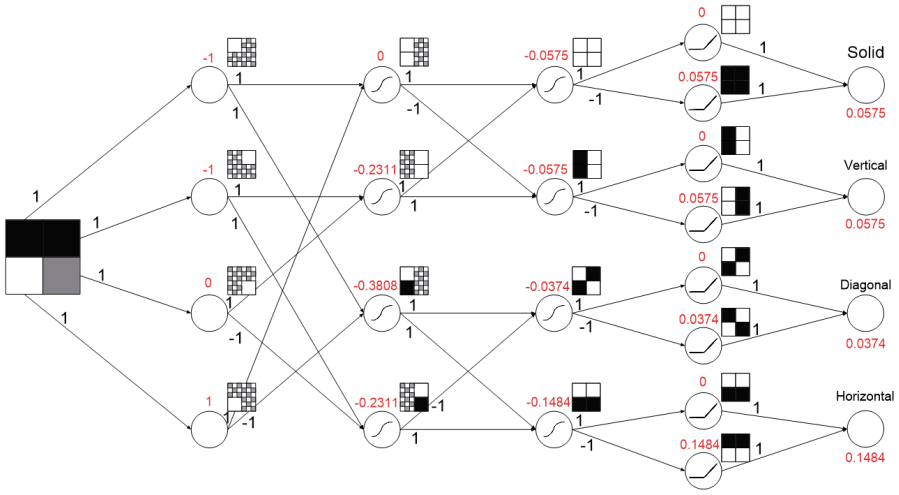


Figure 5: Categorizing a horizontal line with a Feedforward Neural Network

3.1.3 Example 2 : What's for dinner?

In this section, we introduce an other simple example to show the limitations of the Feedforward network. Let's imagine you live in an apartment with a flatmate and he loves to cook. The goal is to build a Feedforward neural network that will guess what's for dinner tonight. Let's say that your flatmate cook either pizza, sushi or burger. If the input vector is composed with the day of the week, the month, the year and today's weather, the network never guesses what's for dinner tonight with good accuracy, whatever the number of layers, the hidden size or the activation functions used. You then realize that your flatmate always cook in a regular cycle : pizza, then sushi, then burger, then pizza again, ... You build a neural network that uses an input vector that, this time, contains what was for dinner yesterday. Now, you have a network that perfectly predict the dinner.

Let's now imagine that you don't have access to the all the data needed because you weren't there yesterday. Your network is then unable to predict anything. It is to address that problem that, in the next chapter, we introduce another neural network, called recurrent neural network.

3.2 Basic Recurrent Neural Network

The Recurrent Neural Network (RNN) is derived from Feedforward network to be able to handle temporal dynamic behavior. RNN uses its internal state to process variable length sequences of inputs. This class of neural network is used in many fields such as time series prediction but also handwriting recognition or speech recognition. In this section, the basic RNN model will be explained and then illustrated with the time series introduced in the previous section. To close this section, we will introduce another task that will demonstrate that, even if basic RNNs are powerful, it is not enough for harder tasks.

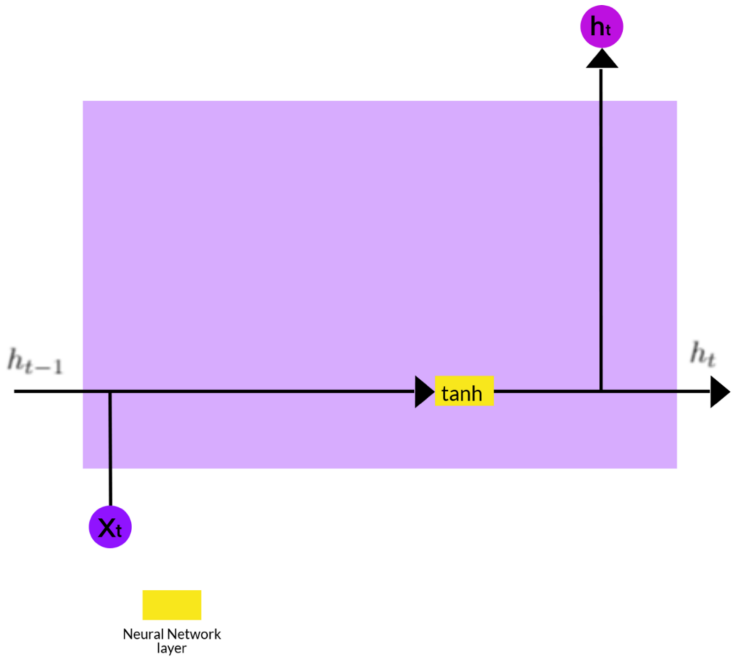


Figure 6: Basic Recurrent Network

3.2.1 Basic RNN model

A basic RNN is a neural network that make use of previous predictions to make the current prediction (see Figure 6). The goal of this network is to add a 'memory' to the Feedforward neural network by adding a feedback loop connected to its past decisions. We give a mathematical description of the process for time t :

$$h_t = activation(WX_t + Uh_{t-1} + b)$$

where :

- h_t is the hidden state at time t
- X_t is the input vector at time t
- W is the weights matrix (like the weight matrices used in the Feedforward network)
- h_{t-1} is hidden state at the previous time step
- U is a weights matrix used for hidden-state-to-hidden-state
- b is the bias vector
- *activation* is the activation function, usually either a logistic sigmoid function or tanh

As in the previous model, the weights matrix W is used as filter to determine how much importance to give to each part of the input vector. In the same manner, the U matrix determines the importance of each part of the hidden state from the past. The current hidden state, h_t, is based on h_{t-1} which is based on h_{t-2} and so on. Therefore, each hidden state contains traces of not only the previous hidden state, but also of all those before. That is why, in theory, RNN can also handle long-term dependencies. But, in practice, it handles only short-term dependencies really well.

3.2.2 Vanishing and exploding gradients

In this section, we will talk about the exploding and vanishing gradients problems. We didn't cover yet the training of our models but we will talk about it in more depth in chapter 5. For now, all we have to know is that, for training, we use gradients. The gradient expresses the change in all weights with regard to the change in error. The gradient is needed to refine the weights in the direction that will decrease the error. Without gradient, the network doesn't learn.

Any quantity multiplied frequently by a number slightly greater than one can become immensely large. The opposite is also true : a number multiplied frequently by a number slightly lower than one can quickly become very small. The problem is that, in neural network, the layers and time steps relate to each other through multiplication. The gradient can then become too large (Exploding gradient) or too small (Vanishing gradient).

The exploding gradient problem can easily be addressed. For example, with gradient clipping, the gradient is re-scaled when the gradient exceed a certain threshold. The vanishing gradient problem is harder to handle. We will talk later how it can be addressed with the Long Short-Term Memory cell.

3.2.3 Example 2 : What's for dinner?

With RNN, it becomes possible to handle this example even if some data are missing. As input, we use what was for dinner yesterday and our network prediction for yesterday as shown in Figure 7a. If we have all the data the network will predict easily like seen before. But, if some data are missing, we just need to unwrap in time the network (see Figure 7b) until we find actual data and then play it forward to predict the dinner for tonight. The model with RNN can then perfectly predict as long as we had data at least one time in the past.

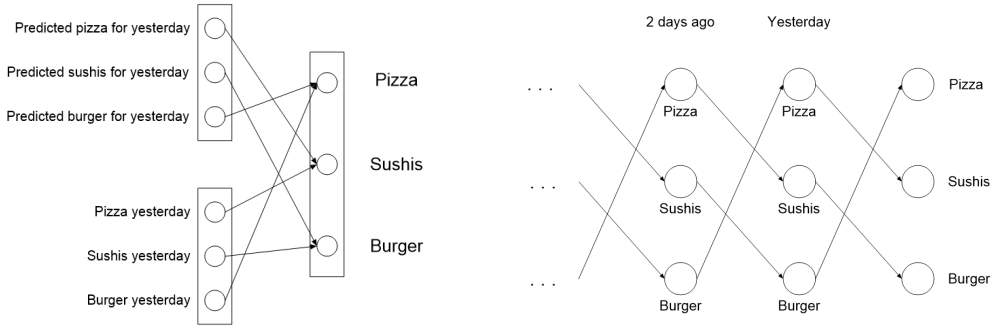
3.2.4 Example 3 : Children's Book

To show that Recurrent Neural Network is not sufficient to meet all our needs, we introduce another toy example. The goal is to write a children's book that only contains sentences of the format "Leonard saw Sheldon." . So our dictionary will only contains names, "saw" and "." . An example of such a book is :

Leonard saw Penny.
Penny saw Amy.
Sheldon saw Leonard.
Amy saw Sheldon.

In this case our dictionary is { Leonard, Sheldon, Penny, Amy, saw, . } .

The input vector of our RNN is the previous word in a one hot encoding vector. For example, the input would be [0,1,0,0,0,0] for Sheldon and [0,0,0,0,1,0] for saw. The output is the prediction of the next word. With training, the network will easily learn that, after a name, comes "saw" or "." and that, after "saw" or ".", comes a name.



(a) Recurrent Neural Network to predict 'What's for dinner?'

(b) Unwrapped predictions for 'What's for dinner?'

Figure 7

But, since a RNN only uses the last prediction, it has a really short-term memory and will make mistakes[3]. Here are three examples of mistakes that such a recurrent network will produce :

Leonard saw Leonard.

Leonard saw Sheldon saw Penny saw Amy.

Leonard. Penny. Sheldon.

To avoid this type of mistakes, we need a memory that can handle longer dependencies. This will be addressed in the next section with the Long Short-Term Memory.

3.3 Long Short-Term Memory

Long Short-Term Memory (LSTM) was proposed in 1997 [8] to deal with the vanishing and exploding problems. LSTM is a recurrent neural network that doesn't only use the last prediction but that also uses an intern memory cell and gates to handle long-term dependencies. The memory cell and gates mechanisms will be described in depth in this section and tested on the previous task.

3.3.1 LSTM model

To control the effects of its memory cell (also called Context), the LSTM uses three gates (see Figure 8) [21]. The forget gate decides what information the cell should forget, the input gate decides what information of the input should be stored in the cell and the output gate decides what parts of the cell is used for the production of the output. The input, the last predictions, the forget gate and the input gate are used for updating the cell. The output gate and the updated cell are used to produce the current prediction. Since the gates act like filters, a logistic sigmoid squashing function is used at each gate to have a vector of numbers between 0 and 1.

The forget gate : The forget gate is a separate neural network that filters the memory cell. It learns when to forget and what parts of the cell to forget. This gate allows the LSTM to quickly forget

[3]Actually, since it is a really simple exercise, the basic RNN should be able to handle it but, as we will see later, the LSTM is more suited

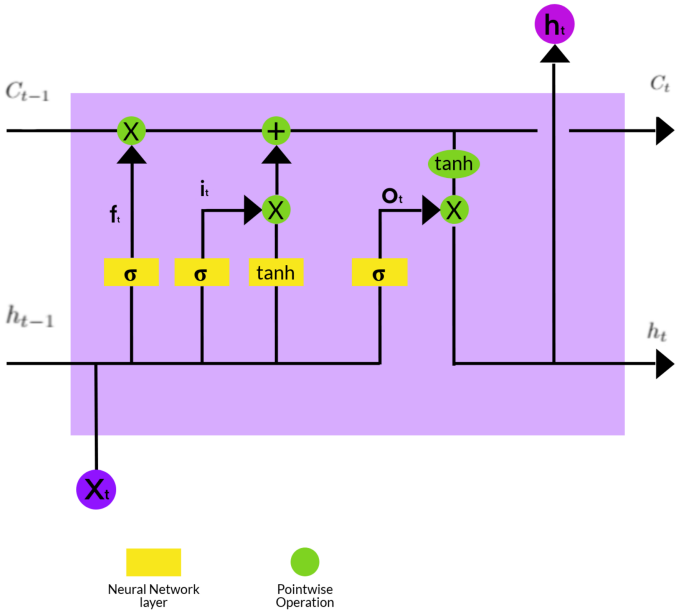


Figure 8: Long Short-Term Memory

the data that has only a short-term dependency while keeping the long-term dependencies. Here are the math behind the forget gate :

$$f_t = \sigma(W_{xf}X_t + W_{hf}h_{t-1} + b_f)$$

where :

- f_t is the forget gate's output vector at time t
- σ is the logistic sigmoid squashing function
- X_t is the input vector at time t
- h_{t-1} is the hidden state vector at time $t-1$ (last predictions)
- W_{xf} and W_{hf} are the forget gate's weights matrices used respectively for the input vector and the hidden state vector
- b_f is the forget gate's bias vector

The input gate : Since it is not necessarily useful to keep all information contained in the input vector, the LSTM uses the input gate to filter the input vector (and the last predictions vector). It's a separate neural network that acts like an attention mechanism to ignore the parts that are not immediately relevant :

$$i_t = \sigma(W_{xi}X_t + W_{hi}h_{t-1} + b_i)$$

where :

- i_t is the input gate's output vector at time t
- σ is the logistic sigmoid squashing function
- X_t is the input vector at time t
- h_{t-1} is the hidden state vector at time $t-1$ (last predictions)
- W_{xi} and W_{hi} are the input gate's weights matrices used respectively for the input vector and the hidden state vector
- b_i is the input gate's bias vector

The cell update : With the forget and input gates, the memory cell can be updated. Firstly, the forget gate is used to forget information, then information is added to the result. This extra information comes from the input gate combined with another neural network that processes the input like in the RNN. This neural network uses a *tanh* squashing function as activation function which keeps its output between -1 and 1 to avoid the exploding and vanishing gradient problem. Note that this cell update addresses the vanishing gradient problem since it is done by an addition and not a multiplication. More formally, the memory cell is updated with the following :

$$C_t = f_t \circ C_{t-1} + i_t \circ tanh(W_{xc}X_t + W_{hc}h_{t-1} + b_c)$$

where :

- $\circ$ is the Hadamard product, the element-wise (or pointwise) product
- C_{t-1} is the memory cell vector at time $t-1$ (before the update)
- C_t is the updated memory cell vector (or context vector) for time t
- f_t is the forget gate's output vector at time t
- i_t is the input gate's output vector at time t
- X_t is the input vector at time t
- h_{t-1} is the hidden state vector at time $t-1$ (last predictions)
- W_{xc} and W_{hc} are the cell update's weights matrices used respectively for the input vector and the hidden state vector
- b_i is the cell update's bias vector

The output gate : While all data contained in the updated memory cell are useful to keep, some of them are not necessary for the current predictions. That's why we need another gate to filter the cell in order to make predictions. Again, the output gate has its own neural network that learns to choose the relevant information of the cell. It works in a similar way to the other gates :

$$o_t = \sigma(W_{xo}X_t + W_{ho}h_{t-1} + b_o)$$

where :

- o_t is the output gate's output vector at time t
- σ is the logistic sigmoid squashing function
- X_t is the input vector at time t
- h_{t-1} is the hidden state vector at time $t-1$ (last predictions)
- W_{xo} and W_{ho} are the output gate's weights matrices used respectively for the input vector and the hidden state vector
- b_o is the output gate's bias vector

Producing the predictions : With the updated cell and the output gate, we finally can build our predictions. Since the cell is updated by an addition, another $tanh$ function is used to be sure that the numbers are between -1 and 1.

$$h_t = o_t \circ tanh(C_t)$$

where :

- $\circ$ is the element-wise product
- h_t is the hidden state at time t (the current predictions)
- o_t is the output gate's output vector at time t
- C_t is the updated memory cell vector at time t

Note : The initial states for C and h are also parameters that will be refined during training.

3.3.2 Example 3: Children's Book

To illustrate how the LSTM cell works and why it is more suited than the basic Recurrent Neural Network for certain tasks, we use an LSTM cell to write the children's book seen in the previous section. Let's say that, after training, the LSTM network has the following behavior :
If the last prediction (h_{t-1}) is a '.' :

- The forget gate will reset all the weights of the memory cell to zero.
- The input gate will be all ones (it doesn't filter anything).
- The cell update will put the weight for the '.' to a strong negative weight so it is not predicted soon. All the other weights will be updated to a strong positive weight.
- The output gate will have zeros for 'saw' and '.' and ones for the names since the first prediction of the sentence should be a name.
- The prediction will be a name and all the names will have the same probabilities to be chosen.

If the last prediction is a name :

- The forget gate will have zeros for the names.

- The input gate will have zeros for the 'saw' and '.' weights because they have to keep their information untouched. For the names, we will have ones so their weights can be updated.
- The weights for the names will be updated : strong negative for the name that as been previously predicted and strong positive for the others. This will avoid the problem of having sentences like 'Leonard saw Leonard.'.
- The output gate will have zeros for the names and ones for 'saw' and '.' because it shouldn't predict a name after a name.
- The prediction will be either 'saw' or '.' depending on the weights of the memory cell. If it is the beginning of the sentence, the weight for 'saw' should be a strong positive and the weight for '.' should be a strong negative. If it is the end of the sentence it should be the opposite.

If the last prediction is 'saw' :

- The forget gate will reset the weights for 'saw' and '.'. The weights for the names will stay untouched to keep track of the name already used.
- The input gate will only allow the weights for 'saw' and '.' to be changed.
- The memory updated will set the weight of 'saw' to strong negative and the weight of '.' to a strong positive to ensure that '.' will be predicted before 'saw'.
- The output gate will have zeros for 'saw' and '.' and ones for the names.
- The prediction will be a name but not the same name as the first of the sentence since its weight will still be strong negative.

With this behavior, this LSTM network will perfectly handle this task. To start with a correct prediction, the initial state of h is '.'.

3.4 Sequence-to-Sequence

The LSTM works well to produce a sequence of predictions as we've seen in the last section. It can also process an input that is a sequence and produce an output either for classification or regression. For example, an LSTM can be used to classify movie reviews into bad review and good review. The words of the review would be passed one by one into the LSTM cell and the last state of the memory would be the predicted classification. On the other hand, one LSTM alone is not able to deal with Sequence-to-Sequence problems. A Sequence-to-Sequence problem is a task where the input is a sequence and has to be processed to produce an output that is also a sequence. The problem we want to tackle in this thesis, the TSP, is a Sequence-to-Sequence task. Indeed, the input is a sequence of cities and the output is a permutation of the cities (which is also a sequence).

In 2014, Google introduced a model, named Seq2Seq [18], to handle these problems. Seq2Seq uses an LSTM encoder to process the input and an LSTM decoder that uses the last hidden state and last state of the memory cell (Context Vector) of the encoder as input to produce the output. Note that both the encoder and the decoder can be multi-layer LSTM's.

If we use this model on the TSP (see Figure 9), here is how it will work :

$$h_i^e \ , \ C_i^e = LSTM^e(h_{i-1}^e, C_{i-1}^e, X_i) \qquad for \quad i = 1, ..., n$$

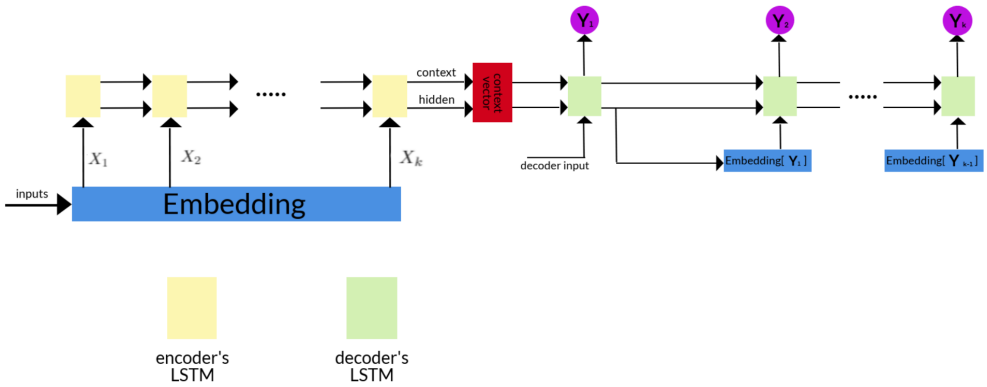


Figure 9: Sequence-to-Sequence for a TSP with k cities

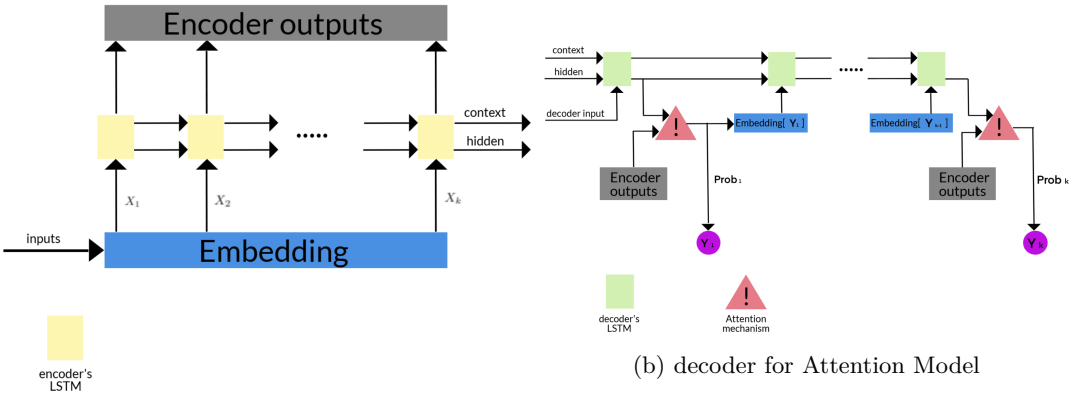
$$h_i^d \ , \ C_i^d = LSTM^d(h_{i-1}^d, C_{i-1}^d, Y_{i-1}) \qquad for \quad i = 1, ..., n$$

where :

- n is the number of cities.
- X_i is the (embedded) coordinates of the ith city.
- $LSTM^e$ is the LSTM network of the encoder.
- h_i^e is the hidden state of the encoder's LSTM after it has processed X_i.
- C_i^e is the state of the memory cell of the encoder's LSTM after it has processed X_i.
- $LSTM^d$ is the LSTM network of the decoder.
- h_i^d is the hidden state of the decoder's LSTM after its ith step. It is a vector representing the probability of each city to be chosen as the ith city of the permutation.
- Y_i is the (embedded) coordinates of ith city of the permutation (chosen according to h_i^d)
- C_i^d is the state of the memory cell of the decoder's LSTM after its ith step.
- the concatenation of C_n^e and h_n^e is the Context Vector, so $C_0^d = C_n^e$ and $h_0^d = h_n^e$
- h_0^e, C_0^e and Y_0 are parameters that are refined during training.

Intuitively, here is how it works : the encoder analyses the input graph by looking at one city at a time. While doing that, it keeps in memory what it has learned from the cities already seen. Once it has analysed all the cities, it has built its own representation of the input graph and it forwards it to the decoder. The decoder does the opposite job. Starting from the given representation (the context vector), it produces the output tour one city at a time. Again, while doing that, it keeps in memory what it has produced and changes the context vector accordingly.

As you can see in Figure 9, the 2D cities are embedded in a higher dimensional space before being passed through the encoder. This embedding is obtained by a linear transformation and gives us more



(a) encoder for Attention Model

(b) decoder for Attention Model

Figure 10: Sequence-to-Sequence with Attention

information about the city and its relations with the other cities. The parameters of this transformation can also be learned during training.

While this model is able to tackle the TSP, it presents a major disadvantage : the Context Vector has a fixed size. Indeed, during encoding, the encoder extracts all useful information from the input and store all the data in the Context Vector. This behavior works well on small input sequence because the Context Vector would be large enough to contain every useful data. But, if the input sequence is very long, the Context Vector would be too small and some data necessary to the production of a relevant output would be lost. A naive way to fix this is to use a really large hidden size but this would highly increase the computational cost during both training and testing.

3.5 Attention Model

To fix the problem of the fixed-sized Context Vector, the Attention Model [2] uses an additional neural network that will simulate a search through the encoder's hidden state during decoding. The goal of this network is to fetch information contained in the input sequence while focusing its attention on certain parts of the input sequence (see Figure 10).

For simplicity, we will name e_i the encoder's hidden state at step i ($h_{e,i}$ in last section) and d_i the decoder's hidden state at step i ($h_{d,i}$ in last section). e_i and d_i are also called respectively reference and query because d_i 'request' the input components as 'references'. The encoder has the same behavior as in the vanilla sequence-to-sequence model but the decoder has the attention mechanism that, for each step i, has the following behavior :

$$u_j^i = v^T tanh(W_{ref} e_j + W_q d_i) \qquad for \quad j = 1, ..., n$$
$$a_j^i = softmax(u^i)_j \qquad for \quad j = 1, ..., n$$
$$d'_i = \sum_{j=1}^{n} a_j^i e_j$$

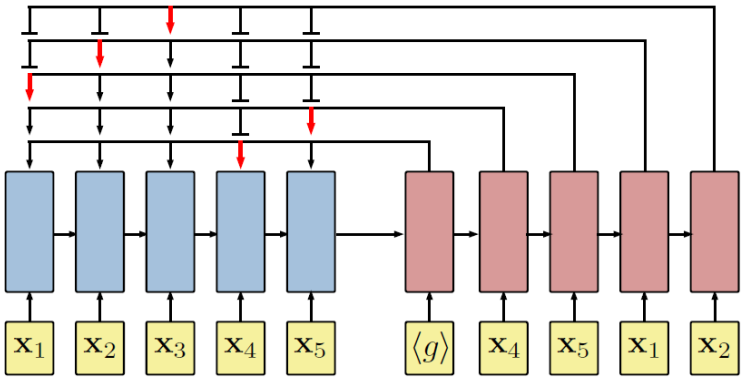


Figure 11: Pointer Network as presented by Vinyals et al. [20]

where:

- v, W_{ref} and W_q are learnable parameters that are refined at training. Usually, the same hidden size is used for the encoder and the decoder so v is a vector and W_{ref} and W_q are square matrices.
- n is the size of the input vector. In the TSP case, it will be the number of cities.
- the softmax function is a function that takes a vector as input and return a vector of the same size. Each component of this vector will be between 0 and 1 and their sum is equal to 1 so they can be interpreted as probabilities. For a vector u of size K : $softmax(u)_j = \frac{e^{u_j}}{\sum_{k=1}^{K} e^{u_k}}$

a^i will act as an 'attention' mask over the encoder's hidden states. Then, d'_i and d_i are concatenated to be used as the hidden states from which the predictions are made and which we feed to the next step of the decoder.

This model can process much longer input sequence without losing important information but it is not applicable to problems where the output dictionary depends on the input dictionary (like in the TSP). Fortunately, the Pointer Network can solve this flaw by doing a simple reduction of the Attention Model.

3.6 Pointer Network

The Pointer Network, introduced in 2015 by Vinyals et al. [20], is a very simple modification of the Attention Model describe in last section. While the latter uses the encoder's states to extract extra information, the Pointer Network uses the vector u^i as pointers to the input components (see Figure 11).

$$u^i_j = v^T tanh(W_{ref} e_j + W_q d_i) \qquad for \quad j = 1, ..., n$$
$$Prob^i = softmax(u^i)$$

Where $Prob_j^i$ is seen as the probability for the input j to be chosen as output at the step i of the decoder. That means that, the higher a component of u^i is, the more chance the city i has to be chosen. The components of the vector u^i are also called the Logits. Again, v, W_{ref} and W_q are learnable parameters.

This simple modification allows us to use the attention mechanism on problems such as the TSP.

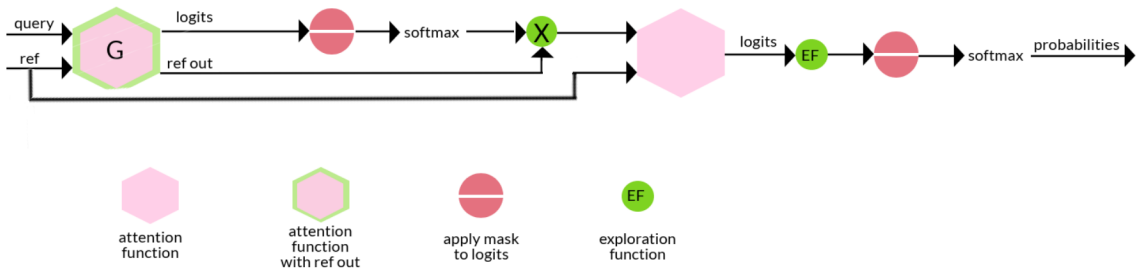


Figure 12: Attention Mechanism with mask to logits, exploration function and one glimpse

4 Our Model's Architecture

Our model uses the Pointer Network architecture but with three extensions used by Bello et al. [3] in 2016 (see Figure 12). The first extension is used to fit the TSP constraints. It consists of a mask on the Logits vector to ensure that the model doesn't choose the same city twice. The second extension, called glimpse, is used to decrease the consequences of how the input sequence is ordered. The last extension propose two different exploration functions to control the distribution of the Logits vector. In this section, we will describe these extensions, give an intuitive description of how it works and show a complete mathematical description of the model's architecture. At the end of the section, we also discuss the differences between our model and the one described by Bello et al. [3].

4.1 Apply Mask to Logits

In Combinatorial optimization problems, respecting the constraints is essential. That is why we need a way to enforce the constraints in our model. If we are unable to find a trick that can eliminate every constraint violations possible, we can discourage it during the training by giving a bad reward when the output is not a feasible solution. Luckily, the TSP case doesn't have any constraints that are hard to enforce. The only constraint is that each city has to be selected once and only once. We can easily ask our model to produce an output of the same size as the input by controlling the number of step at decoding time. With that, the only thing that must be ensured is that we never select twice the same city. This can be done by setting the logits of the cities already selected to $-\infty$. The logits vector then become for each decoder's step i :

$$u_j^i = \begin{cases} v^T tanh(W_{ref} e_j + W_q d_i) & \text{if } j \neq \pi(k) \text{ for all } k < j \\ -\infty & \text{otherwise} \end{cases} \qquad for \quad j = 1, ..., n$$

This mask on the logits is sufficient since, with the softmax function, the probability of selecting a city already in the tour will be zero.

4.2 Glimpse

As discussed by Vinyals et al. in *Order matters: Sequence to sequence for sets* [19], the order of the input sequence matters. Some orders may imply better or worst quality of predictions. To avoid this fluctuation and get the best out of the model, we use an additional computation step, named glimpse. The glimpse will search into the input sequence and will aggregate the contributions of its different

parts. It will compute a linear combination of the encoder's hidden states weighted by the attention probabilities. The whole attention mechanism for each decoder's step i becomes then :

$$
\begin{aligned}
refout_j &= W^g_{ref} e_j \qquad for \quad j = 1, ..., n \\
u^{g,i}_j &= \begin{cases} v^{g,T} tanh(refout_j + W^g_q d_i) & \text{if } j \neq \pi(k) \text{ for all } k < j \\ -\infty & \text{otherwise} \end{cases} \qquad for \quad j = 1, ..., n \\
Prob^{g,i} &= softmax(u^{g,i}) \\
g &= \sum_{k=1}^{n} refout_k Prob^{g,i}_k \\
u^i_j &= \begin{cases} v^T tanh(W_{ref} e_j + W_q g) & \text{if } j \neq \pi(k) \text{ for all } k < j \\ -\infty & \text{otherwise} \end{cases} \qquad for \quad j = 1, ..., n \\
Prob^i &= softmax(u^i)
\end{aligned}
$$

where v^g, $W^g_r ef$ and W^g_q are learnable parameters. The glimpsing mechanism could be applied multiple times using the same parameters but Bello et al. [3] observed that glimpsing more than once made the model less likely to learn with an insignificant improvement of the results.

4.3 Exploration Functions

The last extension we use is the exploration functions to control the distribution of the probabilities. We use two different functions : softmax temperature and logit clipping.

Softmax temperature: We modify the u^i vector described in last section as follow :

$$u^i = u^i / T$$

Where T is a temperature hyperparameter set to $T = 1$ during training. The higher T is, the less steep the probabilities are. This is hence preventing the model from being overconfident.

Logit clipping: We modify the u^i vector from last section as follow :

$$u^i = C \ tanh(u^i)$$

Where C is a hyperparameter that controls the range of the logits and hence the entropy of the probabilities.

4.4 Intuitive description

In this section, we will describe intuitively what the trained model does to produce a solution for a TSP instance. The model is still divided in two parts : the encoder and the decoder. As you can see in Algorithm 3, the encoder is the most simple and only uses an LSTM. Before being sent to the encoder, the coordinates of the cities are embedded in a higher dimensional space to get more information about their relative position compared to each other. The encoder analyses each city at a time, building progressively its own representation of the graph. In addition to that, it keeps its analyse

of each city as a reference to that city. When all the cities has been analysed, the representation of the entire graph is built and sent to the decoder.

Helped by the encoder's graph representation and set of references, the decoder tries to construct a good solution tour. At each step, it selects one of the remaining city to add to the tour. First, the decoder's LSTM reads the representation, sends a query to the Attention mechanism and updates the representation. This query can be interpreted as 'Based on what I know, I want to select a city with the following characteristics'. The Attention mechanism tries to estimate which reference (from the set of references) will fit the best this query. The first part of this mechanism is the glimpse. Its main objective is to transform the query such that it doesn't suffer from the order of the references. To achieve this goal, the glimpse searches through the references and creates a new vector of logits. Then, this vector is processed twice : one time with a mask to ignore the cities already selected and another time to highlight the greatest value thanks to the softmax function. The resulting vector is then combined linearly with the references to form the new query. The second part of the Attention mechanism is the Attention function. It takes the transformed query and searches in the set of references to see which cities fit the query. The vector of logits resulting has a high value for the cities that fit well and a low value for the ones that do not correspond to the query. To control the distribution of this vector, a exploration function is used. Then, like in the glimpse part, the vector is processed by a mask and a softmax function. This gives a final vector representing the estimate probabilities that each city has to be chosen. The selection of the city depends on the search strategy (for example, with a greedy strategy, the city with the highest probability is chosen) but, once selected, it is added to the solution tour and sent to the next step of the decoder such that the LSTM knows what was the answer of its last query. The decoder does this until all the cities has been selected to have a complete solution tour.

4.5 Complete mathematical description

In this section, we will group all the mechanisms stated that compose our model and put them together to give a complete mathematical description.

4.5.1 Encoder

With i going from 1 to n, the number of cities, here is the description of the ith step of the encoder.

$$
\begin{aligned}
f_i^e &= \sigma(W_{xf}^e X_i + W_{hf}^e e_{i-1} + b_f^e) \\
i_i^e &= \sigma(W_{xi}^e X_i + W_{hi}^e e_{i-1} + b_i^e) \\
C_i^e &= f_i^e \circ C_{i-1}^e + i_i^e \circ tanh(W_{xc}^e X_i + W_{hc}^e e_{i-1} + b_c^e) \\
o_i^e &= \sigma(W_{xo}^e X_i + W_{ho}^e e_{i-1} + b_o^e) \\
e_i &= o_i^e \circ tanh(C_i^e)
\end{aligned}
$$

Where :

- W_{xf}^e, W_{hf}^e, W_{xi}^e, W_{hi}^e, W_{xc}^e, W_{hc}^e, W_{xo}^e, W_{ho}^e, b_f^e, b_i^e, b_c^e, b_o^e, e_0 and C_0^e are all learnable parameters refined during training
- X_i is the embedded coordinates of the ith city of the input sequence

Algorithm 3: Resolution of an instance with a trained model

input : test instance s of n cities, trained parameters : e_0, C_0^e, Y_0
output: tour π

$X \longleftarrow Embedding(s)$
for $i \leftarrow 1$ **to** n **do**
  $e_i, C_i^e \longleftarrow LSTM^e(e_{i-1}, C_{i-1}^e, X_i)$
$d_0 \longleftarrow e_n$
$C_0^d \longleftarrow C_n^e$
for $i \leftarrow 1$ **to** n **do**
  $d_i, C_i^d \longleftarrow LSTM^d(d_{i-1}, C_{i-1}, Y_{i-1})$
  $logits, refout \longleftarrow Glimpse(d_i, e)$
  $logits \longleftarrow ApplyMask(logits)$
  $Probs = softmax(logits)$
  $g \longleftarrow \sum_{k=1}^{n} refout_k Probs$
  $logits \longleftarrow Attention(g, e)$
  $logits \longleftarrow Exploration(logits)$
  $logits \longleftarrow ApplyMask(logits)$
  $Probs = softmax(logits)$
  $j \longleftarrow selectCity(Probs)$
  $\pi \longleftarrow \pi + \{j\}$
  $Y_i \longleftarrow X_j$

4.5.2 Decoder

With i going from 1 to n, the number of cities, here is the description of the ith step of the decoder using a logit clipping function and one glimpse :

$$
\begin{aligned}
f_i^d &= \sigma(W_{xf}^d Y_{i-1} + W_{hf}^d d_{i-1} + b_f^d) \\
i_i^d &= \sigma(W_{xi}^d Y_{i-1} + W_{hi}^d d_{i-1} + b_i^d) \\
C_i^d &= f_i^d \circ C_{i-1}^d + i_i^d \circ tanh(W_{xc}^d Y_{i-1} + W_{hc}^d d_{i-1} + b_c^d) \\
o_i^d &= \sigma(W_{xo}^d Y_{i-1} + W_{ho}^d d_{i-1} + b_o^d) \\
d_i &= o_i^d \circ tanh(C_i^d) \\
refout_j &= W_{ref}^g e_j \qquad\qquad for \quad j = 1, ..., n \\
u_j^{g,i} &= \begin{cases} v^{g,T} tanh(refout_j + W_q^g d_i) & \text{if } j \neq \pi(k) \text{ for all } k < j \\ -\infty & \text{otherwise} \end{cases} \qquad for \quad j = 1, ..., n \\
Prob^{g,i} &= softmax(u^{g,i}) \\
g_i &= \sum_{k=1}^{n} refout_k Prob_k^{g,i} \\
u_j^i &= v^T tanh(W_{ref} e_j + W_q g_i) \qquad for \quad j = 1, ..., n \\
u^i &= C \ tanh(u^i) \\
u_j^i &= \begin{cases} u_j^i & \text{if } j \neq \pi(k) \text{ for all } k < j \\ -\infty & \text{otherwise} \end{cases} \qquad for \quad j = 1, ..., n \\
Prob^i &= softmax(u^i)
\end{aligned}
$$

Where :

- W_{xf}^d, W_{hf}^d, W_{xi}^d, W_{hi}^d, W_{xc}^d, W_{hc}^d, W_{xo}^d, W_{ho}^d, b_f^d, b_i^d, b_c^d, b_o^d, W_{ref}^g, W_q^g, v^g, W_{ref}, W_q, v and Y_0 are all learnable parameters that are refined during training
- C is a hyperparemeter
- Y_i is the embedded coordinates of the ith city of the output permutation chosen according to $Prob^i$
- $d_0 = e_n$ and C_0^d=C_n^e

Note that the masking process cannot take place before the logit clipping because the probability of choosing a city already in the permutation wouldn't be zero anymore.

4.5.3 Remarks

Our model has four hyperparameters so far : C for the logit clipping, T for the softmax temperature (of course, choosing what exploration function to use is also a hyperparameter), the hidden size which is the same for both the encoder and decoder and the embedding size. Note that we also have to set initial values for all the learnable parameters.

The number of parameters is depending on the hyperparameters but can be computed with the following formula :

$$\begin{aligned} 12 \times hiddensize^2 + 8 \times hiddensize \times embeddingsize \\ +12 \times hiddensize + embeddingsize \end{aligned}$$

While this number can become easily huge, keep in mind that the goal in this thesis is to replace handcrafted heuristics. Therefore, the computational cost of training is less important than the computational cost at inference or the quality of the output solution.

Our model is very similar to the model proposed by Bello et al. [3] but has three differences proposed in Higgsfield's code [7]. The first one is that we use the exploration function only once while Bello et al. [3] also apply it in the glimpse mechanism. Also in the glimpse, they do a linear combination of the query by the set of references while we use the *refout* described in section 4.2. We noticed that using these two differences can slightly speed up the training without any additional computing cost. It's even a little bit more efficient since we don't compute the exploration function twice. The last difference, but the most important, is that we apply the mask to the logits after the exploration function. The problem with applying it before is that the probability of choosing an already selected city would not be zero anymore (with the logit clipping function). With this problem, the model can choose twice the same city, making its output tour an invalid solution.

5 Training

Now that we have the architecture of the model, we need a way to refine its parameters to get the best predictions possible. If we were to focus only on the TSP, we could use Supervised Learning to train the model. Since this Combinatorial problem is one of the most studied, it is easy to find an algorithm that proved its efficiency and we could use its results as objective for training. We could use either an exact algorithm or an approximate algorithm but they both have drawbacks. An exact algorithm would give us the optimal solution for training but it requires a high computational cost. On the contrary, an approximate algorithm is cheaper but we would have to train our model on a lower quality solution. So, if we were to choose Supervised Learning, we would have to compromise between computational cost and quality of the solution. In addition to that, the goal of this model is to be able to generalize to find competitive solution on unseen or on poorly solved problems where there is no efficient heuristics found yet. Therefore, it would be expensive to get solutions to train on.

By contrast, Reinforcement Learning is more suited to our case. Indeed, it will make our model independent of any existing handcrafted heuristic. The reward mechanism is moreover simple to compute for Combinatorial optimization problem. In this chapter, we will explain how Reinforcement Learning works and, in particular, how we can express the REINFORCE algorithm [24] to our problem. We will then describe an Actor-Critic training proposed by Bello et al. [3] that uses a parametric network as critic. Finally, we will describe the optimizer used for stochastic gradient descent.

5.1 Reinforcement Learning

Reinforcement Learning is based on 'trial and error'. The *agent* observes the *environment* and decides what *action* to take. Then, based on this action, the environment sends back a *reward* to the agent. If the reward is good (lower is better), the agent reinforces its behavior, if it is bad, the agent discourages it. See Figure 13.

For our case, our model (the agent) observes the graph (the environment), takes an action by choosing a permutation of the cities and receives a reward which is the length of this permutation. If the reward is good, the parameters of the model would be adapted to encourage this choice and, if the reward is bad, they would be refined to discourage it.

5.2 REINFORCE algorithm

To optimize the parameters of our model, we use model-free policy-based Reinforcement Learning using policy gradient methods and stochastic gradient descent. The training objective, J, is the tour length. Then, for an input graph s, it is defined as

$$J(\theta|s) = L(\pi|s)$$

Where θ is the parameters of the model and $L(\pi|s)$ is the length of the tour π for the graph s, as defined in chapter two.

We estimate the gradient of this objective by using the famous REINFORCE algorithm. This algorithm states that the gradient is proportional to the sum, over all actions, of the quality of an action weighted by the gradient of the probability of taking that action given the current parameters and the state. We formulate the gradient of the objective for the graph s as :

$$\nabla_\theta J(\theta|s) \approx L(\pi|s)\, \nabla_\theta \; log \; p_\theta(\pi|s)$$

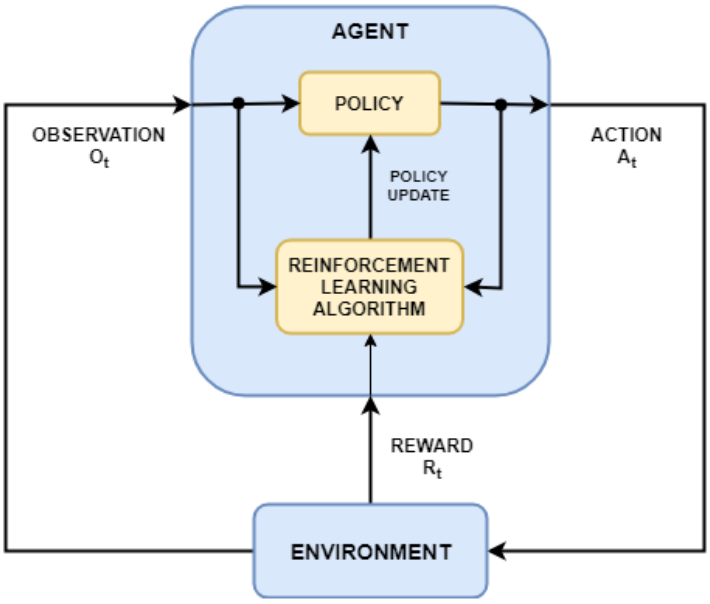


Figure 13: Reinforcement Learning [15]

Where $p_\theta(\pi|s)$, the policy, is the probability that our model select π as the tour for the graph s, given its parameters θ.

To speed up the training process, we use batches. For each batch, we sample B graphs (s_1, s_2, ... , s_B), pass them through the model and then compute the gradient for the batch using the following formula :

$$\nabla_\theta J(\theta) \approx \frac{1}{B} \sum_{i=1}^{B} L(\pi_i|s_i)\, \nabla_\theta \; log \; p_\theta(\pi_i|s_i)$$

In the TSP, even with the same number of city, the different graphs can have very different optimal tour length. Hence, the gradients will have high variance. To reduce this variance, we use a baseline function, *b(s)*, that does not depend on π and estimate the expected tour length. The gradient then becomes :

$$\nabla_\theta J(\theta) \approx \frac{1}{B} \sum_{i=1}^{B} \big(L(\pi_i|s_i) - b(s_i)\big)\, \nabla_\theta \; log \; p_\theta(\pi_i|s_i)$$

A commonly used way of computing the baseline function is a simple exponential moving average of the rewards obtained by the model. This baseline will take into account the fact that the network improves with training. Here is how it is computed for the ith batch of training :

$$b_i = \beta \; b_{i-1} + (1-\beta)\frac{1}{B} \sum_{i=1}^{B} L(\pi_i|s_i)$$

Where β is a hyperparameter (between 0 and 1). Although this is a simple and relatively efficient baseline, it suffers from two disadvantages : It does depend on π and it is shared across all instances of the batch. Because of the later, a good (or even optimal) prediction of a difficult graph may be still discouraged because it can be greater than the baseline.

5.3 Critic network

As proposed by Bello et al. [3], another way of estimating the expected tour length is to use a parametric baseline. This additional network will act as a *critic* while our model described in Chapter 4 will act as *actor*. The critic network is composed of three neural modules :

- An LSTM encoder that behaves exactly like the actor's encoder.
- An LSTM process block that takes the context vector from the encoder and performs P steps of computation to update it. At each processing state, the LSTM produces a hidden state that is updated by a Glimpse function and uses the output of this function as input of the next processing step.
- a two layers ReLU Feedforward neural network that will take as input the last hidden state of the process block and output a single scalar that will be the baseline prediction.

The critic network has its own parameters, denoted θ_c, and thus its own optimizer. We train the critic with stochastic gradient descent using a mean square error objective between its baseline predictions and the length of the tour predicted by the actor. We formulate this additional objective's gradient as :

$$\nabla_{\theta_c} J_c(\theta_c) \approx \frac{1}{B} \sum_{i=1}^{B} \left(b_{\theta_c}(s_i) - L(\pi_i|s_i) \right)^2$$

The full training algorithm with the critic network is describe in Algorithm 4

Algorithm 4: Actor-critic training [3]

input : set of training graphs S, number of training step T, batch size B
output: refined actor network's parameters θ

Initialize actor network's parameter θ
Initialize critic network's parameters θ_c
for $t \leftarrow 1$ **to** T **do**
  $s_i \longleftarrow SampleInput(S)$ for $i \in \{1, ..., B\}$
  $\pi_i \longleftarrow SampleSolution(p_\theta(.|s_i))$ for $i \in \{1, ..., B\}$
  $b_i \longleftarrow b_{\theta_c}(s_i)$ for $i \in \{1, ..., B\}$
  $g_\theta \longleftarrow \frac{1}{B} \sum_{i=1}^{B} \left(L(\pi_i|s_i) - b_i \right) \nabla_\theta \; log \; p_\theta(\pi_i|s_i)$
  $g_{\theta_c} \longleftarrow \frac{1}{B} \sum_{i=1}^{B} \left(b_i - L(\pi_i|s_i) \right)^2$
  $\theta \longleftarrow Optimizer(\theta, g_\theta)$
  $\theta_c \longleftarrow Optimizer(\theta_c, g_{\theta_c})$

5.4 Optimizer

To update our parameters, we use an optimizer that is widely used for models with large number of parameters, the Adam optimizer [12]. Introduced in 2015, it is an optimization algorithm used rather than the classical stochastic gradient descent procedure. At each time step t, it computes an exponential moving average of the gradient and another exponential moving average for the squared gradient :

$$m_t = \beta_1 \; m_{t-1} \; + \; (1 - \beta_1) \; g_{\theta,t}$$
$$v_t = \beta_2 \; v_{t-1} \; + \; (1 - \beta_2) \; g^2_{\theta,t}$$

Where :

- β_1 and β_2 are hyperparamters between 0 and 1 that represent the exponential decay rates. They are set by default at $\beta_1 = 0.9$ and $\beta_2 = 0.999$.
- m_t and v_t are the exponential moving averages respectively for the gradient and the squared gradient.
- $m_0 = 0$ and $v_0 = 0$

Due to their initialization, these averages lead to estimates that are biased toward zero. To counteract this initialization bias, bias-corrected estimates, $\hat{m_t}$ and $\hat{v_t}$ are used :

$$\hat{m_t} = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v_t} = \frac{v_t}{1 - \beta_2^t}$$

Where β_1^t and β_2^t denote β_1 and β_2 to the power t. With this estimate, we can now update the parameters :

$$\theta_t = \theta_{t-1} - \alpha \; \frac{\hat{m_t}}{\sqrt{\hat{v_t}} + \epsilon}$$

Where α is a hyperparameter representing the stepsize (set by default at $\alpha = 0.001$) and ϵ is a hyperparamter used to prevent any division by zero (by default, $\epsilon = 10^{-8}$)

6 Search Strategies

Once the model is trained, predicting a tour is relatively fast and the evaluation of a tour is also inexpensive. This allows us to use search procedures in combination of our model. We can use usual local search algorithm like 2opt search because our model does not necessarily produce local optima. But we can also implement search based on our model. A simple way to improve the results, proposed by Bello et al. [3], is to use several trained models and to select the best tour among their predicted tour for each test instance. But we also considered two other search strategies, also proposed by Bello et al., Sampling and Active Search.

6.1 Sampling

In greedy decoding, the city selected at each decoder's step is the one with largest probability. In sampling strategy, multiple candidate tours are sampled for each test instance and the shortest one is selected. Some of the candidate tours will be the same because we don't enforce the model to sample different tours. However, the softmax temperature exploration function, described in section 4.3, allows us to control the diversity of the sampled tours.

6.2 Active Search

In greedy and Sampling strategies, the reward is ignored at inference time and, thus, the parameters are not refined. On the contrary, the idea of the Active Search is to refine its parameters for each single test instance, specializing the model for that particular instance. We can use this strategy with a pretrained model or even from scratch with an untrained model. To increase the stochasticity of the sampling procedure, we shuffle the test instance sequence before feeding it to the model. With B, the batch size, B solutions are sampled with the model's policy. Then the gradient is computed using a similar formula than in Algorithm 4 :

$$\nabla_\theta J(\theta) = \frac{1}{B} \sum_{i=1}^{B} \big(L(\pi_i|s_i) - b\big) \nabla_\theta \; log \; p_\theta(\pi_i|s_i)$$

The first difference is that all the graph s_i are the same graph but with a different order. Also, the baseline b used in the Active Search is a exponential moving average as describe in section 5.2. The reason we use a moving average instead of a critic network is the fact that we don't need to differentiate the input graphs anymore since there are all the same. Thus, there is no risk of discouraging a harder graph's prediction. Note that the pretraining of our model is dependent of the distribution of training graphs but the Active Search is distribution independent. The algorithm of the Active Search is presented in Algorithm 5.

Algorithm 5: Active Search [3]

input : test instance s, number of training step T, batch size B, β
output: best tour π

Initialize moving average b
$\pi \longleftarrow RandomTour()$
$L_\pi \longleftarrow L(\pi|s)$
for $t \leftarrow 1$ **to** T **do**
  $s_i \longleftarrow Shuffle(s)$ for $i \in \{1, ..., B\}$
  $\pi_i \longleftarrow SampleSolution(p_\theta(.|s_i))$ for $i \in \{1, ..., B\}$
  $j \longleftarrow Argmin(L(\pi_1|s_1),\ ...\ , L(\pi_B|s_B))$
  $L_j \longleftarrow L(\pi_j|s_j)$
  if $L_j < L_\pi$ **then**
    $\pi \longleftarrow \pi_j$
    $L_\pi \longleftarrow L_j$
  $g_\theta \longleftarrow \frac{1}{B} \sum_{i=1}^{B} \big(L(\pi_i|s_i) - b\big) \nabla_\theta\ log\ p_\theta(\pi_i|s_i)$
  $\theta \longleftarrow Optimizer(\theta, g_\theta)$
  $b \longleftarrow \beta\ b + (1 - \beta) \frac{1}{B} \sum_{i=1}^{B} L(\pi_i|s_i)$

7 Experiment and Analyses

In this chapter, we will give a description of the code. Then, we will give all the details about the methods used to experiment our model. We will finally show the results and analyze them.

7.1 The code

We started with the code of Higgsfield [7] and modified it to fit our model. This Python notebook is an implementation of the model presented by Bello et al. [3] using PyTorch [16]. It has different classes and functions :

- The TSPDataset class : This class creates num_sample TSP instances where num_nodes cities are drawn uniformly at random in the unit square $[0, 1]^2$. The random_seed parameter is used to manually set the random seed.
- The reward function : This function takes as input a number of tours equal to the batch size and computes their respective length.
- The Attention class : This class implements a PyTorch module for the Attention Mechanism (either for the glimpse or the attention function). The init function initiates the W_{ref} and W_q matrices and the v vector. The forward function takes the references set and the query as input and computes the attention function and the exploration function outputting the $refout$ and the $logits$ (the $refout$ is only used for the glimpse and the exploration function is only used for the attention function).
- The GraphEmbedding class : It implements a PyTorch module for the Embedding mechanism. Its forward function takes the batch of input graphs and returns a batch of graphs with embedded coordinates

- The PointerNet class : It implements a PyTorch module for the model. It initiates an instance of the GraphEmbedding class, the LSTM encoder module, the LSTM decoder module, an instance of the Attention class for the glimpse and another one for the attention function. It also initiates the Y_0 vector. The apply_mask_to_logits function updates the mask and apply it on the logits. The forward function takes the batch of input graphs and makes them pass through the model outputting the indexes of the tours selected and the probabilities given by the attention mechanism at each decoder step.

- The CombinatorialRL class : It also implements a PyTorch module and initiates an instance of the PointerNet class. Its forward function makes the inputs go through that instance and computes the rewards of the resulting tours.

- The TrainModel class : This class initiates the Adam optimizer and has a function called train_and_validate. This function optimizes the parameters during *n_epochs* epochs. For each epoch, it loads the batches from the train set. For each of these batches, it makes it go through the model and uses the rewards to optimize the parameters thanks to a Exponential Moving Average baseline. It often validates the current parameters on the validation set. Also, it regularly plots the rewards (of the training and validation sets) thanks to the Matplotlib library [9].

This code supports GPU thanks to the PyTorch cuda() method. To use it on CPU, the variable USE_CUDA has to be set on *False*.

We started from this code and made modifications. For debugging, we were helped by the useful discussions on Stack Overflow[4] and on PyTorch[5]. We began by doing some fixes such that the code is compatible with Python 3 and the recent versions of PyTorch. We changed the TSPDataset class so the default value (-1) would not set manually any random seed. Then, we added a greedy function in the PointerNet and CombinatorialRL classes. This function acts like their respective forward function but, at each decoder step, selects the city with the highest probability. We also moved the exploration function from the Attention class to the PointerNet class. This modification allows us to choose between using the exploration function before applying the mask or after. We also added the softmax temperature exploration function.

To train the model with a critic baseline, we added a CriticNet class that implements a PyTorch module. Its init function initiates an instance of the GraphEmbedding class, an LSTM module for the encoder, an LSTM module for the process block, an instance of the Attention class for the glimpse, two linear layers and a ReLU function. The forward function takes the batch of inputs and makes it pass through the encoder, the processing block and the two layers with ReLU to return a baseline prediction. To use that class, we updated the TrainModel class. Now, in its init function, it also has to initiate an Adam optimizer for the critic's parameters. The train_and_validate function uses the critic as baseline instead of the Exponential Moving Average. It also uses a mean square error to optimize the critic's parameters.

Another modification that we made in the TrainModel class is a learning rate decay. To do so, we used the LambdaLR function with a scheduler[6] and we set the number of training steps by epoch at 5000. To allow the fact that the training set is different at each epoch, we used a vector of random seeds as input of the train_and_validate function with each seed corresponding to one epoch. We also save the model state after each epoch using the torch.save() function[7].

[4] https://stackoverflow.com/
[5] https://discuss.pytorch.org/
[6] as described in https://pytorch.org/docs/stable/optim.html
[7] Thanks to this tutorial : https://pytorch.org/tutorials/beginner/saving_loading_models

To implement the Sampling search strategy, we added a Sampling class. The sample function takes the test set and, for each instance, creates a dataset composed of replications of this instance. This dataset's size is equal to the number of candidates that have to be sampled. Then, the dataset is passed in the model and the best reward is kept during the process. The sample function outputs a vector containing the best rewards corresponding at each test instance and another vector with their respective solution tour.

The last class we added is the Active_Search class. Its init function initiates an Adam optimizer and a checkpoint for the actor's parameters. The act_search function handles the test set by taking one instance at a time. For each instance, it creates a dataset containing replications and permutations of the instance. Again, its size is equal to the number of candidates that have to be sampled. Before using this new dataset, the model state is set to the checkpoint so the model has the same starting point for every test instances. Then, the dataset is passed through the model and it keeps track of the best reward found. After each batch, the model's parameters are optimised thanks to a Exponential Moving Average baseline.

To save and load the test sets and the solution tours, we used the Pickle library[8]

Our code of the model is available[9] on GitHub in the Neural_Combinatorial_Optimization.py file, along with the test instances used and an example of trained model for each size of TSP. The Python file contains all the classes described above and, at the end of the file, there are some examples of how to use them.

7.2 Experiment details

We consider three different sizes of the Euclidean TSP : 20, 50 and 100 cities. To test the performances for each size, we use a testing set of 1000 instances having all their cities drawn uniformly at random in the unit square $[0, 1]^2$. In all of our models described below, we use a hidden size and a embedding size of 128. We use the Adam optimizer with an initial learning rate of 10^{-3} for the TSP20 and TSP50 and 10^{-4} for the TSP100. We decay this learning rate every 5000 training steps by a factor of 0.96. To avoid an exploding gradient, we clip the norm of our gradients to 1.0. We initialize our parameters uniformly at random within $[-\frac{1.}{\sqrt{128}}, \frac{1.}{\sqrt{128}}]$. We use one attention glimpse as described in chapter 4.

To Pretrain the models, we use the Critic Network with 3 processing steps and we use the logit clipping exploration function with $C = 10$, clipping them to $[-10, 10]$. We train our models with up to 200000 training steps for TSP20 and TSP50 and up to 250000 for TSP100.

Greedy The simplest search strategy is greedy decoding from a pretrained model. At each decoding step, we select the city with the largest probability. Since the time to solve one instance must be unbiased, we have to use a batch size of 1.

Greedy 8 With greedy 8, we use 8 pretrained models. For each test instances, each model collects greedily a tour and the shortest one is kept. Again, we use a batch size of 1.

Greedy + 2opt For this search strategy, we greedily decode from a pretrained model and use the predictions as initial value for the 2opt search algorithm.

[8]https://docs.python.org/3/library/pickle.html

[9]code available at https://github.com/qdeffense/np-hard-deep-reinforcement-learning

Method	$TSP20$		$TSP50$		$TSP100$	
	tour length	time	tour length	time	tour length	time
Nearest Neighbor	4.4868	0.0098s	7.0080	0.0543s	9.6885	0.2828s
2 opt	3.9473	0.0319s	6.1302	0.2481s	8.5140	1.1382s
Nearest Neighbor + 2 opt	3.9211	0.0358s	6.0143	0.2546s	8.2977	1.1057s
Concorde	3.8207	0.0134s	5.7001	0.0587s	7.7690	0.3139s
OR Tools	3.8517	0.0067s	5.8541	0.0488s	8.0538	0.2271s
Greedy	3.9259	0.0442s	6.2177	0.1443s	8.7305	0.3872s
Greedy 8	3.8503	0.3441s	5.9884	1.1360s	8.4867	3.1137s
Greedy + 2 opt	3.8697	0.0594s	5.9708	0.2733s	8.3074	1.0404s
Sampling	3.8450	0.9481s	6.0284	4.1461s	8.3017	50.8806s
Trained + Active Search	3.8430	15.3028s	6.0110	47.2664s	8.2755	163.7373s
Untrained + Active Search	6.2242	170.3564s	-	-	-	-

Table 1: Results of our model against the baselines, running time is on the Google Colab's Nvidia Tesla K80 GPU

Sampling For each test instance, we sample 12800 candidates from a pretrained model while keeping track of the shortest tour. The batches consist of replication of the test instance. Since the parameters are not refined during searching, we use a larger batch size of 1024. While Bello et al. [3] used the softmax temperature exploration function to sample, we experimented better results with the logit clipping function. Even with a small search grid on the temperature, the best result for TSP20 with the temperature was 3.8571 with $T = 2.5$ but, with the clipping, we obtained a result of 3.8450.

Trained + Active search This Active Search strategy starts with the parameters of a pretrained model and refines them for each test instance. The learning rate is set to a hundredth of the initial learning rate of the pretrained model (thus 10^{-5} for TSP20 and TSP50 and 10^{-6} for TSP100). We run Active Search for 100 training steps for each test instance and keep track of the shortest tour. Thus, with a batch size of 128, we sample 12800 candidates. Here, the batches consist of replications or permutations of the test instance. The baseline used to train with Active Search is an exponential moving average with a decay set to $\beta = 0.99$

Untrained + Active Search This strategy does the same as last strategy but starts from an untrained model for each test instance. Since it is not trained, we let it run 1000 training steps for each test instance. Thus, with a batch size of 128, it samples 128000 candidates. Since this method is not supposed to produce good results and is not cheap, we show its results only on the TSP20.

7.3 Results and analyses

In Table 1, we compare our results against the baselines described in section 2.2. The two first baselines are the Nearest Neighbor and the 2-opt algorithms. Then, we use the 2-opt search with, as initial solution, the tours found by the Nearest Neighbor. The next baseline is the Concorde solver which gives the optimal solution. The last one is the vehicle routing solver from OR Tools which, even if not state-of-the-art, gives a good approximate solution. An example of resolutions of a test instance is also presented in Appendix B.

	$TSP20$		$TSP50$		$TSP100$	
Method	tour length	time	tour length	time	tour length	time
Greedy TSP20	3.9259	0.0442s	6.7409	0.1531s	11.7004	0.4159s
Greedy TSP50	4.0448	0.0497s	6.2177	0.1443s	8.9189	0.4159s
Greedy TSP100	4.5369	0.0497s	6.6246	0.1542s	8.7305	0.3872s

Table 2: Results of our models against variable problem sizes, running time is on the Google Colab's Nvidia Tesla K80 GPU

We observed that, for the TSP20, our model can outperformed the OR Tools solver with Greedy 8, Sampling and Trained Active Search. These three methods achieve close to optimal but are the most expensive strategies. The Trained Active Search is the slowest, it is even more than a thousand time slower than the Concorde algorithm. The Greedy and Greedy 2 opt still managed to beat the two easy baselines (Nearest Neighbor and 2opt).

For the TSP50, none of the methods could outperform the OR Tools solver but Greedy 8, Greedy 2opt and Trained Active Search had better performances than the Nearest Neighbor + 2opt. The Sampling was better than the two simplest baselines but Greedy only managed to beat the Nearest Neighbor. Surprisingly, the Sampling and Trained Active Search were outperformed by Greedy 8 and Greedy 2opt. The results of the Greedy 2opt show that upgrading the model's solution to the local optimum may be really efficient.

The TSP100 Trained Active Search was the only method that outperformed the Nearest Neighbor + 2opt and the Greedy was the only one that's not beating the 2opt. The rest of the methods had performances between the 2opt and the Nearest Neighbor + 2opt.

In general, the best method is the Trained Active Search but it is the most expensive. The sampling method is a good trade-off between performances and computational cost. It often achieves results that are close from the Active Search. For these two methods, the performances and the cost are tied to the number of candidates sampled. Using 1,280,000 candidates would give better results like shown by Bello et al. [3] but we believed that using 12,800 candidates has already a high enough computational cost. We observed that our results are a little less competitive but still close to the results obtained by Bello et al.

The Greedy method is not really competitive but its two extensions (Greedy 8 and Greedy 2opt) achieve good performances for an attractive computational cost.

As expected, the Untrained Active Search give bad results for a high computational cost. But it still proves that Active Search can learn from scratch and, with more sampled candidates, it could reach better performances.

The main problem is that all the methods of our model are less competitive as the length of the instance increases.

An important property of our model is to be able to generalize to variable problem sizes. In table 2, we can see that the generalization is not good for a long range. Indeed, the TSP20 model is not good for solving TSP100 instances and the TSP100 is not good for solving TSP20. On the other hand, at a short range, the models are still efficient. Thus, the model is able to generalize to a certain extend.

8 Further work

We tested our model on the TSP but the goal is to be able to handle unseen Combinatorial problems and a lot of them are very different from the TSP. To announce that a model is ready to tackle all these problems, it has to be tested on a large variety of Combinatorial problems.

A cheap way to estimate if the model is able to generalize to a certain problem is to use the Untrained Active Search on one or several instances of this problem. The averages of the rewards during the Active Search would show if the model is improving or not. If it does, the model would most likely be able to learn heuristics for this problem.

On another hand, our model obtained good results but it is still far from the state-of-the-art hand-crafted heuristics. A first way to achieve better performances is a deeper tuning of the hyperparameters. A downside of this is that the model may become overconfident and may lose its generality for other problems. If it is the case, replacing heuristics made by hand by hyperparameters tuned by hand is pointless.

During training, there is a moment where our model stopped learning. Improving the training would give a better trained model which would give better results. To improve the training, another baseline may be more suited. For example, Kool et al. [13] proposed to use a deterministic greedy rollout[10]. Another way may be to use an hybrid training using both Reinforcement and Supervised Learning. First, the model would be trained with Reinforcement Learning and, then, the training would be completed with Supervised Learning on optimal solution. As discussed, in chapter 5, the optimal solution for a new problem or a poorly solved problem would be costly to gather. But, if it is only used at the end of the training, this training set would not need to be large and, therefore, would not be too expensive.

The main problem is to scale up to large test instances. As discussed by Joshi et al. in [11], one possible way to fix that is to also use a hybrid approach to improve the model. The model would first be trained using Supervised Learning on small (thus cheap) instances and then trained on large instances using Reinforcement learning.

Another issue with our problem are the LSTMs. The LSTMs are efficient but also expensive in computational cost. A model without them would then be cheaper. For example, Kool et al. [13] introduced a model using Attention mechanisms without LSTMs and giving good performances. Another example is the model proposed by Joshi et al. [11] that uses deep Graph Convolutional Networks with Beam Search.

[10] the rollout uses the best model so far to compute the baseline

9 Conclusion

We have explained in more depth the model presented by Bello et al. [3] and presented our slightly different model. We have built the model using Recurrent Neural Network with Attention Mechanisms to learn heuristics to solve Combinatorial problems. We have focused on Traveling Salesman Problem (TSP) and we have trained our model with Reinforcement Learning using a Critic Network. Thanks to different search strategies, especially the Trained Active Search and the Greedy 2opt, our model gave good solutions for the TSP but seemed unable to scale up to large instances. The results of the Greedy 2opt proves that the model does not often output a local optimum solution. Also, the model was able to generalize to variable problem sizes within a certain range.

References

[1] David Applegate et al. “Concorde TSP solver”. In: (2006). URL: http://www.math.uwaterloo.ca/tsp/concorde/.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate”. In: *arXiv e-prints*, arXiv:1409.0473 (Sept. 2014), arXiv:1409.0473. arXiv: 1409.0473 [cs.CL].

[3] Irwan Bello et al. “Neural Combinatorial Optimization with Reinforcement Learning”. In: *arXiv e-prints*, arXiv:1611.09940 (Nov. 2016), arXiv:1611.09940. arXiv: 1611.09940 [cs.AI].

[4] Nicos Christofides. *Worst-case analysis of a new heuristic for the travelling salesman problem.* Technical Report 388. Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

[5] G. A. Croes. “A Method for Solving Traveling-Salesman Problems”. In: *Operations Research* 6.6 (1958), pp. 791–812. URL: https://EconPapers.repec.org/RePEc:inm:oropre:v:6:y:1958:i:6:p:791-812.

[6] Google. “OR Tools, google optimization tools”. In: (2016). URL: https://developers.google.com/optimization.

[7] Higgsfield. “np-hard-deep-reinforcement-learning”. In: (Dec. 2017). URL: https://github.com/higgsfield/np-hard-deep-reinforcement-learning.

[8] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[9] J. D. Hunter. “Matplotlib: A 2D graphics environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[10] Pawan Jain. “Complete Guide of Activation Functions”. In: (June 2019). URL: https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044.

[11] Chaitanya K. Joshi, Thomas Laurent, and Xavier Bresson. “An Efficient Graph Convolutional Network Technique for the Travelling Salesman Problem”. In: *arXiv e-prints*, arXiv:1906.01227 (June 2019), arXiv:1906.01227. arXiv: 1906.01227 [cs.LG].

[12] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *arXiv e-prints*, arXiv:1412.6980 (Dec. 2014), arXiv:1412.6980. arXiv: 1412.6980 [cs.LG].

[13] Wouter Kool, Herke van Hoof, and Max Welling. “Attention, Learn to Solve Routing Problems!” In: *arXiv e-prints*, arXiv:1803.08475 (Mar. 2018), arXiv:1803.08475. arXiv: 1803.08475 [stat.ML].

[14] S. Lin and W. Kernighan. “An effective heuristic algorithm for the traveling-salesman problem”. In: *Operation Research* 21.2 (1973), pp. 498–516.

[15] MathWorks. In: (). URL: https://es.mathworks.com/help///reinforcement-learning/ug/what-is-reinforcement-learning.html.

[16] Adam Paszke et al. “Automatic differentiation in PyTorch”. In: (2017).

[17] Brandon Rohrer. “Recurent Neural Network (RNN) and Long Short-Term Memory (LSTM)”. In: (June 2017). URL: https://www.youtube.com/watch?v=WCUNPb-5EYI&t.

[18] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. "Sequence to Sequence Learning with Neural Networks". In: *arXiv e-prints*, arXiv:1409.3215 (Sept. 2014), arXiv:1409.3215. arXiv: 1409.3215 [cs.CL].

[19] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. "Order Matters: Sequence to sequence for sets". In: *arXiv e-prints*, arXiv:1511.06391 (Nov. 2015), arXiv:1511.06391. arXiv: 1511.06391 [stat.ML].

[20] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. "Pointer Networks". In: *arXiv e-prints*, arXiv:1506.03134 (June 2015), arXiv:1506.03134. arXiv: 1506.03134 [stat.ML].

[21] Wikipedia contributors. *Long short-term memory — Wikipedia, The Free Encyclopedia*. [Online; accessed 15-August-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Long_short-term_memory&oldid=965461224.

[22] Wikipedia contributors. *Nearest neighbour algorithm — Wikipedia, The Free Encyclopedia*. [Online; accessed 15-August-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Nearest_neighbour_algorithm&oldid=944937402.

[23] Wikipedia contributors. *Travelling salesman problem — Wikipedia, The Free Encyclopedia*. [Online; accessed 16-August-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Travelling_salesman_problem&oldid=971259137.

[24] Ronald J. Williams. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Mach. Learn.* 8.3–4 (May 1992), pp. 229–256. ISSN: 0885-6125. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696.

[25] D. H. Wolpert and W. G. Macready. "No Free Lunch Theorems for Optimization". In: *Trans. Evol. Comp* 1.1 (Apr. 1997), pp. 67–82. ISSN: 1089-778X. DOI: 10.1109/4235.585893. URL: https://doi.org/10.1109/4235.585893.

Appendices

A Critic Network Description

A.1 LSTM Encoder

With i going from 1 to n, the number of cities, here is the description of the ith step of the encoder.

$$
\begin{aligned}
f_i^e &= \sigma(W_{xf}^e X_i + W_{hf}^e e_{i-1} + b_f^e) \\
i_i^e &= \sigma(W_{xi}^e X_i + W_{hi}^e e_{i-1} + b_i^e) \\
C_i^e &= f_i^e \circ C_{i-1}^e + i_i^e \circ tanh(W_{xc}^e X_i + W_{hc}^e e_{i-1} + b_c^e) \\
o_i^e &= \sigma(W_{xo}^e X_i + W_{ho}^e e_{i-1} + b_o^e) \\
e_i &= o_i^e \circ tanh(C_i^e)
\end{aligned}
$$

Where :

- W_{xf}^e, W_{hf}^e, W_{xi}^e, W_{hi}^e, W_{xc}^e, W_{hc}^e, W_{xo}^e, W_{ho}^e, b_f^e, b_i^e, b_c^e, b_o^e, e_0 and C_0^e are all learnable parameters refined during training
- X_i is the embedded coordinates of the ith city of the input sequence

Note that this is exactly the same behavior as the actor's encoder.

A.2 LSTM Process Block

With i going from 1 to P, the number of processing steps, here is the description of the ith step of the decoder using one glimpse :

$$
\begin{aligned}
f_i^d &= \sigma(W_{xf}^d g_{i-1} + W_{hf}^d d_{i-1} + b_f^d) \\
i_i^d &= \sigma(W_{xi}^d g_{i-1} + W_{hi}^d d_{i-1} + b_i^d) \\
C_i^d &= f_i^d \circ C_{i-1}^d + i_i^d \circ tanh(W_{xc}^d g_{i-1} + W_{hc}^d d_{i-1} + b_c^d) \\
o_i^d &= \sigma(W_{xo}^d g_{i-1} + W_{ho}^d d_{i-1} + b_o^d) \\
d_i &= o_i^d \circ tanh(C_i^d) \\
refout_j &= W_{ref}^g e_j \qquad for \quad j = 1, ..., n \\
u_j^{g,i} &= v^{g,T} tanh(refout_j + W_q^g d_i) \qquad for \quad j = 1, ..., n \\
Prob^{g,i} &= softmax(u^{g,i}) \\
g_i &= \sum_{k=1}^{n} e_k Prob_k^{g,i}
\end{aligned}
$$

Where :

- W_{xf}^d, W_{hf}^d, W_{xi}^d, W_{hi}^d, W_{xc}^d, W_{hc}^d, W_{xo}^d, W_{ho}^d, b_f^d, b_i^d, b_c^d, b_o^d, W_{ref}^g, W_q^g, v^g, and g_0 are all learnable parameters that are refined during training
- $d_0 = e_n$ and C_0^d=C_n^e

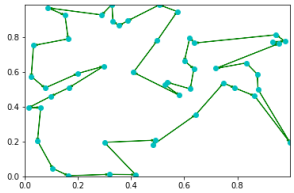
A.3 2-layers ReLU decoder

$$b(s) = W_{c2}\ ReLU(W_{c1}\ g_P + b_{c1}) + b_{c2}$$

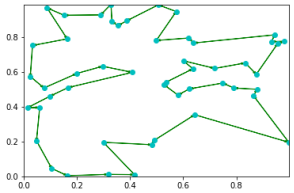
Where :

- ReLU(x) = max(0,x).
- W_{c1}, W_{c2}, b_{c1} and b_{c2} are learnable parameters refined during training.
- $b(s)$ is the prediction of the baseline function for the graph s.

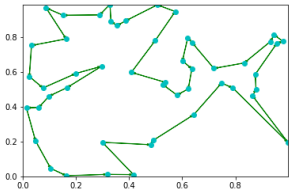
B Example of resolutions of a test instance



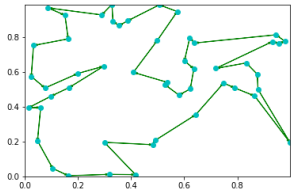
(a) Greedy 1 (6.4177)



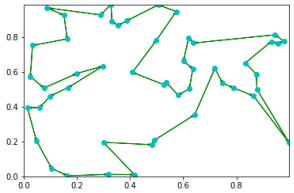
(b) Greedy 8 (6.0818)



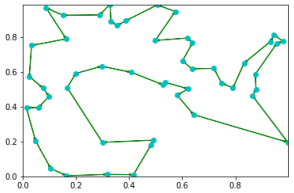
(c) Greedy + 2opt (6.0727)



(d) Sampling (6.2332)



(e) Trained Active Search (6.1394)



(f) Optimal by Concorde (5.7867)

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**