HAL
open science

LatticeNN – Deep Learning and Formal Concept Analysis

Esteban Marquer

To cite this version:

Esteban Marquer. LatticeNN – Deep Learning and Formal Concept Analysis. Artificial Intelligence [cs.AI]. 2020. hal-02954843

HAL Id: hal-02954843

https://inria.hal.science/hal-02954843v1

Submitted on 1 Oct 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE LORRAINE
IDMC

MASTER THESIS

LatticeNN
Deep Learning and Formal Concept Analysis

Author:
Esteban MARQUER

Supervisors:
Pr. Miguel COUCEIRO
Ajinkya KULKARNI

A thesis submitted in fulfillment of the requirements for the degree of Master of Natural Language Processing

in the

Orpailleur team
Loria

February 20 – July 20, 2020

Declaration of Authorship

I, Esteban MARQUER, declare that this thesis titled, "LatticeNN
Deep Learning and Formal Concept Analysis" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

UNIVERSITÉ DE LORRAINE

Abstract

IDMC
Loria

Master of Natural Language Processing

LatticeNN
Deep Learning and Formal Concept Analysis

by Esteban MARQUER

In recent years there has been an increasing interest in approaches to combine formal knowledge and *artificial neural networks* (NNs), called *neuro-symbolic* approaches.*Formal concept analysis* (FCA) is a powerful formal tool for understanding complex data called *formal context* (FC). FCA can be used to generate a structured view of the data, typically a hierarchy of *formal concepts* called *concept lattice* or an ontology. It can also discover implications between some aspects of the data and generate explainable formal rules grounded on the data, which can in turn be used to construct decision systems from the data.

In this thesis, we explore ways to solve the scalability problem inherent to FCA with the hope of revealing implicit information not expressed by FCA, by using deep learning to reproduce the processes of FCA. Recently, neural generative models for graphs have achieved great performance on the generation of specific kinds of graphs. Therefore, we explore an approach to reproduce the formal lattice graph of FCA using generative neural models for graphs, and in particular GraphRNN. Additionally, we develop a data agnostic embedding model for formal concepts, Bag of Attributes (BoA). Relying on the performance of BoA, we develop an approach to generate the *intents*, a description of the formal concepts. We report experimental results on generated and real-word data to support our conclusions.

Acknowledgements

First, I would like to express my gratitude to Miguel Couceiro and Ajinkya Kulkarni, my supervisors, for the time they have spent on my internship. I am particularly grateful for their patient advice and their ideas without which this project wouldn't have existed. I also wish to thank Alain Gely and Alexandre Bazin who lent their expertise in FCA and helped making BoA a reality. Many thanks to all of them for their friendliness which made this internship such a nice experience.

A very special gratitude to all the members of Orpailleur, Semagramme and Synalp, and to my fellow students of the NLP master, for the many discussions I had with them this year and the ones before, related or not to this project, because it's this atmosphere which led me into academia.

Finally, I would like to thank the Loria for hosting me, even for a short time, and for they masterful crisis management. I also to thank the Inria Project Lab (IPL) HyAIAI ("Hybrid Approaches for Interpretable AI") for funding my research internship. The experiments presented in my thesis were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

Contents

Declaration of Authorship iii

Abstract v

Acknowledgements vii

1 Project Frame 1
1.1 Introduction . . . 1
1.2 Work Environment . . . 2
1.2.1 LORIA, Orpailleur and Multispeech . . . 2
1.2.2 Inria Project Lab HyAIAI . . . 3
1.2.3 Tools, Repository and Testbed . . . 3
1.3 Basic Background in Formal Concept Analysis . . . 3
1.3.1 Formal Contexts and Formal Concepts . . . 4
1.3.2 Formal Concept Lattices . . . 4
1.3.3 Formal Concept Lattices Generation Algorithms . . . 5
1.4 Basic Background in Deep Learning . . . 5
1.4.1 Neural Networks . . . 5
1.4.2 Deep Learning Algorithms . . . 6
1.4.3 Usual Loss Functions . . . 6
1.4.4 Binary Encoding and Softmax . . . 7
1.4.5 Major Neural Networks Architectures . . . 7
1.4.6 Auto-Encoders and Embeddings . . . 9
1.4.7 Metric Learning . . . 10
1.5 Problem Statement . . . 11

2 Initial Approach: Graph Generation 13
2.1 State of the Art of Graph Modeling . . . 13
2.1.1 Node-Centered Approaches . . . 14
2.1.2 Whole Graph Approaches . . . 14
2.1.3 Advantages of GraphRNN for Lattice Generation . . . 15
2.2 Transforming Lattices into Matrices . . . 15
2.2.1 Data Generation and Dataset . . . 16
2.2.2 From Breath-First Search to Level Ordering . . . 17
2.2.3 Encoding the Lattice Adjacency . . . 18
2.2.4 Encoding the Concepts . . . 19
2.3 Using GraphRNN for Lattices Modeling . . . 20
2.3.1 GraphRNN . . . 20
2.3.2 GraphRNN Constrained VAE . . . 21
2.4 Preliminary Experiments and Change of Approach . . . 22
2.4.1 Reconstruction Performance Without Features . . . 23
2.4.2 Formal Concept Representation and Change of Approach . . . 23

3 Bag of Attributes: Embeddings for Formal Contexts **25**
3.1 State of the Art: FCA2VEC . . . 25
3.1.1 Object2vec and Attribute2vec . . . 26
3.1.2 Closure2vec . . . 27
3.1.3 Evaluation . . . 27
3.2 Bag of Attributes Model . . . 29
3.2.1 Architecture . . . 29
3.2.2 Training Objective . . . 31
3.3 Training . . . 32
3.3.1 Training Process . . . 32
3.3.2 Training Dataset . . . 33
3.3.3 Data Augmentation . . . 33
3.3.4 Issues With KL Divergence . . . 34
3.4 Experiments . . . 34
3.4.1 Reconstruction Performance . . . 34
3.4.2 Metric Learning Performance . . . 36
3.4.3 Experiments on Real-World Datasets . . . 37

4 Second Approach: Intents Generation **39**
4.1 Pilot Experiments . . . 40
4.1.1 Variational Auto-Encoder Architecture . . . 40
4.1.2 Convolutional Neural Network Architecture . . . 40
4.1.3 Recurrent Neural Network Architecture and Attention Mechanisms . . . 42
4.1.4 Conclusions on the Tested Architectures . . . 44
4.2 Concept Number Upper Bound Prediction . . . 44
4.3 Intents, Cover and Order Relation Generation . . . 47
4.4 Training and First Experiments . . . 48

5 Conclusion and Discussion **51**

Bibliography **53**

List of Figures

1.1 Example of a formal context of geometrical shapes. 4
1.2 Lattice of the example context from Figure 1.1. 5
1.3 Multi-layer perceptron architecture. From https://www.researchgate.net/profile/Ayhan_Erdem2/publication/319309006. 6
1.4 Example of CNN architecture. From Wikipedia. 8
1.5 Structure of the major RNN architectures, the input at step t is x_t, the hidden state h_t, and the output o_t. From Wikipedia. 8
1.6 Example of attention mechanism for translation from English to French. From https://blog.floydhub.com/attention-mechanism/ 9
1.7 Example of DAN, Figure 1 from [23]. 9

2.1 Lattice from Figure 1.2 organized in levels and the corresponding level-based and BFS-based orders. 17
2.2 Diagram of a batch of 2 samples encoded using the 6 values NO EDGE (0) and EDGE (1) for the adjacency, SOS, MOS, EOS, and PAD. 19
2.3 Narrow encoding of the example lattice of Figure 1.2. 20
2.4 GraphRNN at inference time. Figure 1 from [51]. 21
2.5 Adapted GraphRNN Constrained VAE. 22
2.6 Details of the adapted GraphRNN, with edge and intent generation. . 22
2.7 Example of lattice reconstruction in the early stages. 23

3.1 Continuous Bag of Words (CBoW) and skip-gram architectures from word2vec. Figure 1 from [38]. 26
3.2 Schematic representation of the BoA architecture. Blue blocks correspond to tensors, the orange to neural components and green blocks to non-neural computations. Arrows joining blocks represent concatenation of tensors. 30
3.3 Schematic representation of the BoA architecture training process. . . . 32
3.4 Reconstruction performance on random contexts. The error bars and the shaded area correspond to the standard deviation. 35
3.5 Predicted metrics against the actual values, for the 200 samples with 20 objects and attributes from the test set. 36

4.1 Block diagrams of the VAE and CNN architectures. 41
4.2 Examples of results of the VAE and CNN architectures. In the images, a blue pixel corresponds to a 0 and a yellow one to 1. The first row correspond to the prediction by the model and the second row is the actual intent matrix. Each column correspond to a different sample in the same batch. 41
4.3 Example of a sequence to sequence RNN, figure from [24]. 42
4.4 Block diagram of the recurrent architectures. 43
4.5 Hierarchy of the tested recurrent architecture variants 43

4.6 Extract of results on the test set with the various variants of the LSTM model. In the images, a blue pixel corresponds to a 0 and a yellow one to 1. The first row correspond to the prediction by the model and the second row is the actual intent matrix. Each column correspond to a different sample in the same batch. 45
4.7 Upper bounds predicted by the model on all the samples of the evaluation set. 46
4.8 Schematic representation of the intent generation architecture. Blue blocks correspond to tensors and orange to neural components. 47
4.9 Schematic representation of the intent model training process. 49
4.10 Batch of 2 samples (one per row) of the evaluation set and the predictions of our model. The first three columns correspond to the sample itself, and the last three to the model's prediction. 50

List of Tables

2.1 Descriptive statistics on the dataset of randomly generated contexts. . 16
2.2 Prediction performance of the GraphRNN VAE on the development set, for the adjacency matrices of the graphs of $\prec$ and $\leq$. 23

3.1 Descriptive statistics on ICFCA* and wiki44k, the datasets used to evaluate object2vec and attribute2vec. Table 1 from [9]. 27
3.2 Rate of intra-cluster implication with the attribute2vec skip-gram variant (*a2v-SG*) and the naive and random baselines, for embeddings of size 2 and 3, for $k = 2$, 5 and 10 clusters, for 20 repetitions of the evaluation experiment (mean $\pm$ std.). Table 3 from [9]. 28
3.3 Performance of the two *object2vec* variants and the *node2vec* baseline, for embeddings of size 2 and 3, for 30 repetitions of the evaluation experiment (mean $\pm$ std.). *o2v-SG* stands for the skip-gram and *o2v-CBoW* for the CBoW variant. Table 2 from [9]. 29
3.4 Descriptive statistics on the dataset of randomly generated contexts. . 33
3.5 Descriptive statistics on SPECT heart. 37
3.6 Performance on the link prediction task (mean $\pm$ std.). 38
3.7 Performance on the attribute clustering task with 2, 5 and 10 clusters (mean $\pm$ std.). 38

4.1 Performance of the intent model on the evaluation set. 49

List of Abbreviations

AUC ROC	**A**rea **U**nder the **ROC C**urve
BCE	**B**inary **C**ross-**E**ntropy
BFS	**B**readth-**F**irst **S**earch
BLSTM	**B**idirectional **LSTM**
BoA	**B**ag **o**f **A**ttributes
CBoW	**C**ontinuous **B**ag **o**f **W**ords
CHD	**C**losure **H**amming **D**isatnce
CNN	**C**onvolutional **N**eural **N**etwork
CNRS	*Centre National de la Recherche Scientifique*
DAN	**D**eep **A**veraging **N**etwork
EOS	**E**nd **O**f **S**equence value
et al.	***et al****ii,* and others
FCA	**F**ormal **C**oncept **A**nalysis
FCA4AI	What can **FCA** do **for A**rtificial **I**ntelligence
FC	**F**ormal **C**ontext
GRU	**G**ated **R**ecurrent **Unit** RNN
HyAIAI	**Hy**brid **A**pproaches for **I**nterpretable **AI**
i.e.	***i****d* ***e****st,* that is
IPL	**I**nria **P**roject **L**ab
KL divergence	**K**ullback-**L**eibler divergence
LSTM	**L**ong **S**hort-**T**erm **M**emory RNN
MLP	**M**ulti-**L**ayer **P**erceptron
MOS	**M**iddle **O**f **S**equence value
MSE	**M**ean **S**quared **E**rror
NLP	**N**atural **L**anguage **P**rocessing
NN	**N**eural **N**etwork
PAD	**PAD**ding value
PCA	**P**rincipal **C**omponent **A**nalysis
ReLU	**Re**ctified **L**inear **U**nit
RNN	**R**ecurrent **N**eural **N**etwork
ROC	**R**eciever **O**perating **C**aracteristic
SG	**S**kip-**G**ram
SOS	**S**tart **O**f **S**equence value
Std.	**St**andard **d**eviation
TSNE	**T**-distributed **S**tochastic **N**eighbor **E**mbedding
UL	***U****niversité de* ***L****orraine*
UMR	***U****nité* ***M****ixte de* ***R****echerche*
VAE	**V**ariational **A**uto-**E**ncoder
w.r.t.	**w**ith **r**egard **t**o

List of Symbols

o	Object ($o \in O$)
O	Finite set of objects
a	Attribute ($a \in A$)
A	Finite set of attributes
$\mathbf{I}$	Incidence relation ($\mathbf{I} \subseteq A \times O$)
e	Extent ($e \in E, e \subseteq O$)
E	Finite set of extents
i	Intent ($i \in I, i \subseteq A$)
I	Finite set of intents
C	Binary matrix of the formal context
$\top$	Top concept
$\bot$	Bottom concept
L	Levels
$\mathcal{O}^L$	Order based on the levels L
$\leq$	Order relation
$\prec$	Cover relation
$\mathcal{L}$	Adjacency matrix of a relation
μ	Mean vector
σ	Standard deviation vector
$\cdot'$	Closure operator
$\lvert \cdot \rvert$	Size / norm of $\cdot$

Chapter 1

Project Frame

1.1 Introduction

In recent years there has been an increasing interest in approaches to combine formal knowledge and *artificial neural networks* (NNs), called *neuro-symbolic* approaches [3, 14]. *Formal concept analysis* (FCA) is a powerful formal tool for understanding complex data called *formal context* (FC) (see Subsection 1.3.1). FCA can be used to generate a structured view of the data, typically a hierarchy of *formal concepts* called *concept lattice* (see Subsection 1.3.2) or an ontology. It can also discover implications between some aspects of the data and generate explainable formal rules grounded on the data. This can be used to construct decision systems from the data.

Replicating FCA's mechanisms using NNs could help processing complex and large datasets [9, 34] by tackling FCA's scalability issues [33]. It could also help integrate FCA into connectionist pipelines and allow us to discover new aspects that the standard FCA process doesn't explore. Following this idea, we want to reproduce the general discovery process of FCA with NN architectures. This asks for a general framework capable of generating concept lattices using exclusively the FC.

To our knowledge, there are only a few neuro-symbolic approaches involving FCA. Rudolph *et al.* [41] show how to encode closure operators with simple feed-forward NNs. Gaume *et al.* [16] make the parallel between FCA and bipartite graph analysis, by considering the FC as a bipartite graph. In [34], Kuznetsov *et al.* present an approach to construct NNs from the results of FCA. More recently, Dürrschnabel *et al.* [9] present FCA2VEC, a framework to represent FCs by encoding FCA's closure operators, using the results of [41]. However, those approaches do not offer a complete neural framework to approximate FCA.

To overcome these limitations, we explore two complementary approaches to the generation of concept lattices using NNs, respectively the generation of lattices as labeled graphs and the generation of concept intents. Our work provides a framework for the task of reproducing FCA using NNs, supported by experimental results that serve as a baseline. We also provide a representation framework for FCs called *Bag of Attributes* (BoA) that we detail in [37].

This report is organized as follows. Chapter 1 introduces the subject and the background of the project. In Section 1.2 we present the Inria Project Lab HyAIAI and the LORIA laboratory, as well as our main development tools. In Section 1.3 and Section 1.4 we introduce basic notions respectively of FCA and deep learning. Finally, we explain the goals of the project in Section 1.5. We describe the two approaches we explored respectively in Chapter 2 and Chapter 4, by presenting a detailed overview of the literature, a description of the NN architectures we tested and the experiments we ran. Chapter 2 contains an overview of the state-of-the-art of graph generation, a description of our data representation and generation process,

and the details of the architecture we developed in our initial approach based on graphs. The challenges encountered when designing this architecture led us to create BoA, an embedding framework to represent FCs. BoA is detailed in Chapter 3 together with FCA2VEC [9]. Chapter 4 is the description of the second approach we explored, a very modular approach to the problem of lattice generation focused on intents. Finally, we discuss the results of the project and explain some plans to further expand our approach in Chapter 5.

1.2 Work Environment

The internship project is a collaboration of the Orpailleur and Multispeech research teams from LORIA, within the frame of the Inria Project Lab HyAIAI. The internship took place within the LORIA research lab, under the supervision of Miguel Couceiro and Ajinkya Kulkarni. In this section, we briefly describe the LORIA lab (Subsection 1.2.1) as well as the Inria Project Lab HyAIAI (Subsection 1.2.2). We also present the tools we used in our experiments (Subsection 1.2.3).

1.2.1 LORIA, Orpailleur and Multispeech

LORIA[1] is a French mixed research unit (*Unité Mixte de Recherche*, UMR 7503). In other words, it is a research lab shared by three institutions: the French national center for scientific research (*Centre National de la Recherche Scientifique*, CNRS)[2], the *Université de Lorraine* (UL)[3] and the Inria[4], the national institute for research in digital science and technology. The LORIA was created in 1997 and focuses on both fundamental and applied research in computer sciences. The lab is directed by Jean Yves Marion together with the direction team. They are helped by a scientific council, a lab council, and an assembly of the researcher responsible for each team.

The research in the lab is organized in 30 research teams, each focusing on specific thematic or goals. The teams are grouped in 5 research department depending on the main direction of the team's research:

- the "Algorithms, Computation, Image and Geometry" department focuses on geometry and symbolic computation and its algorithmic problems;
- the "Formal Methods" department focuses on "analyzing, verifying and developing safe and secure software-based systems"[1] using formal methods;
- the "Networks, Systems and Services" department focuses on large networks as well as parallel and distributed systems;
- the "Natural Language Processing and Knowledge Discovery" department focuses on processing and modeling language and knowledge;
- finally, research in the "Complex Systems, Artificial Intelligence and Robotics" department focuses on artificial intelligence and robotics.

[1] https://www.loria.fr/en
[2] www.cnrs.fr/en
[3] http://welcome.univ-lorraine.fr/en
[4] https://www.inria.fr/en

Orpailleur and Multispeech are two teams of the "Natural Language Processing and Knowledge Discovery" department. Orpailleur[5] groups researchers are interested in knowledge discovery and engineering while Multispeech[6] focuses on processing speech. Miguel Couceiro is the head of Orpailleur and Ajinkya Kulkarni is a Ph.D. student in Multispeech.

1.2.2 Inria Project Lab HyAIAI

Current and efficient machine learning approaches rely on complex numerical models, and the decisions which are proposed may be accurate but cannot be easily explained to the layman. That is a problem especially in some cases where complex and human-oriented decisions should be made, *e.g.*, to get a loan or not, to obtain a chosen enrollment at university.

HyAIAI[7] ("Hybrid Approaches for Interpretable Artificial Intelligence") is an Inria Project Lab (IPL) about the design of novel, interpretable approaches for artificial intelligence. The objectives of the IPL HyAIAI are to study the problem of making machine learning methods interpretable, by designing hybrid ML approaches that combine state-of-the-art, numerical models (*e.g.* neural networks) with explainable symbolic models (*e.g.* pattern mining). Our goal of creating a neuro-symbolic framework for FCA is the first step towards integrating FCA into NNs.

The IPL HyAIAI project involves seven Inria Teams, namely Lacodam in Rennes (project leader), Magnet and SequeL in Lille, Multispeech and Orpailleur in Nancy, and TAU in Saclay.

1.2.3 Tools, Repository and Testbed

Our project required several processing scripts and the implementation and training of the proposed NNs architectures. We stored our code on Gitlab. We used an Anaconda[8] environment with Python 3.8, the deep learning library PyTorch[9], as well as major data science libraries (*e.g.*, Pandas, Seaborn, ScikitLearn). The extensive list of packages used is available on our Gitlab repository[10].

Our experiments were run on Grid5000[11], a platform for experimentation supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations. Grid5000 provides computational clusters equipped with powerful *graphic processing units* (GPUs) which are necessary to train NNs in reasonable time. In particular, we used the *grue*, *graffiti* and *grele* clusters, whose description is available on Grid5000's website[12].

1.3 Basic Background in Formal Concept Analysis

In this section, we briefly recall some basic background in FCA. We focus on the finite case of FCA (with a finite set of objects and attributes) because we want to apply FCA to analyze finite data. For further details see, *e.g.*, [13, 22].

[5] https://orpailleur.loria.fr/
[6] https://team.inria.fr/multispeech/
[7] https://project.inria.fr/hyaiai/
[8] https://www.anaconda.com/
[9] https://pytorch.org/
[10] https://gitlab.inria.fr/emarquer/fcat
[11] https://www.grid5000.fr
[12] https://www.grid5000.fr/w/Nancy:Hardware

	4 sides	3 sides	Regular	Isosceles
Square	×		×	
Rectangle	×			
Rectangle Triangle		×		
Isosceles Triangle		×		×
Equilateral Triangle		×	×	×

FIGURE 1.1: Example of a formal context of geometrical shapes.

1.3.1 Formal Contexts and Formal Concepts

A *formal context* (FC) is a triple $\langle A, O, \mathbf{I} \rangle$, where A is a finite set of attributes, O is a finite set of objects, and $\mathbf{I} \subseteq A \times O$ is an incidence relation between A and O. A formal context can be represented by binary table C with objects as rows C_o and attributes as columns C_a, for $o \in O$ and $a \in A$. The entry of C corresponding to o and a is defined by $C_{o,a} = 1$ if $(o, a) \in \mathbf{I}$, and 0 otherwise. In FCA, entries equal to 1 are also called *crosses*. Figure 1.1 is an example of a formal context of shapes and their geometrical properties.

It is well known [13] that every formal context $\langle A, O, \mathbf{I} \rangle$ induces a Galois connection between objects and attributes, called *closure operator*: for $X \subseteq O$ and $Y \subseteq A$, defined by: $X' = \{y \in A \mid (x, y) \in \mathbf{I} \text{ for all } x \in X\}$ and $Y' = \{x \in O \mid (x, y) \in \mathbf{I} \text{ for all } y \in Y\}$. A *formal concept* is then a pair (X, Y) such that $X' = Y$ and $Y' = X$, called respectively the *intent* and the *extent*. It should be noticed that both X and Y are closed sets, i.e., $X = X''$ and $Y = Y''$. We denote by $I \subseteq 2^A$ the set of intents, $E \subseteq 2^O$ the set of extents and $C \subseteq I \times E$ the set of concepts. In simple terms, a formal concept is a "rectangle" of crosses in a formal context, relating the attributes shared by a set of objects and the objects containing the set of attributes.

1.3.2 Formal Concept Lattices

The set of all formal concepts can be ordered by inclusion of the extents or, dually, by the reversed inclusion of the intents. The order relation $\leq$ on concepts is defined as $\langle i_1, e_1 \rangle \leq \langle i_2, e_2 \rangle \iff e_1 \subseteq e_2$. As $\leq$ is a partial order relation, the set of formal concepts together $\leq$ form a *partially ordered set*. More specifically, they form a lattice L called a *formal concept lattice*.

In lattices, every pair of elements (formal concepts in our case) have a unique *supremum* (or *join*), which is the lowest element which is greater than or equal to the elements in the pair. As such, it is also called *least upper bound*. Similarly, every pair of elements has a unique *infimum* (or *meet* or *greatest lower bound*). Additionally, in the finite case that interests us, there exist two special elements in every lattice: *top* (written $\top$) and *bottom* ($\bot$). $\top$ is the *global maximum* and $\bot$ the *global minimum* of the lattice. As such, for every concept $c \in C$ in the lattice, we have $\bot \leq c \leq \top$. Typically, the intent of $\top$ is empty and its extent is O, while the extent of $\bot$ is empty and its intent is A.

The *strict order relation* $<$ of the order relation $\leq$ is defined by $x < y$ if $x \leq y$ and $x \neq y$. The *cover relation* $\prec$ is defined by $x \prec y$ if $x < y$ and there is no z such that $x < z < y$. The *Hasse diagram*, the graph of this $\prec$ relation, is a standard representation for formal concept lattices. It is an *acyclic directed graph*. Figure 1.2 is the Hasse diagram of the example lattice from Figure 1.1.

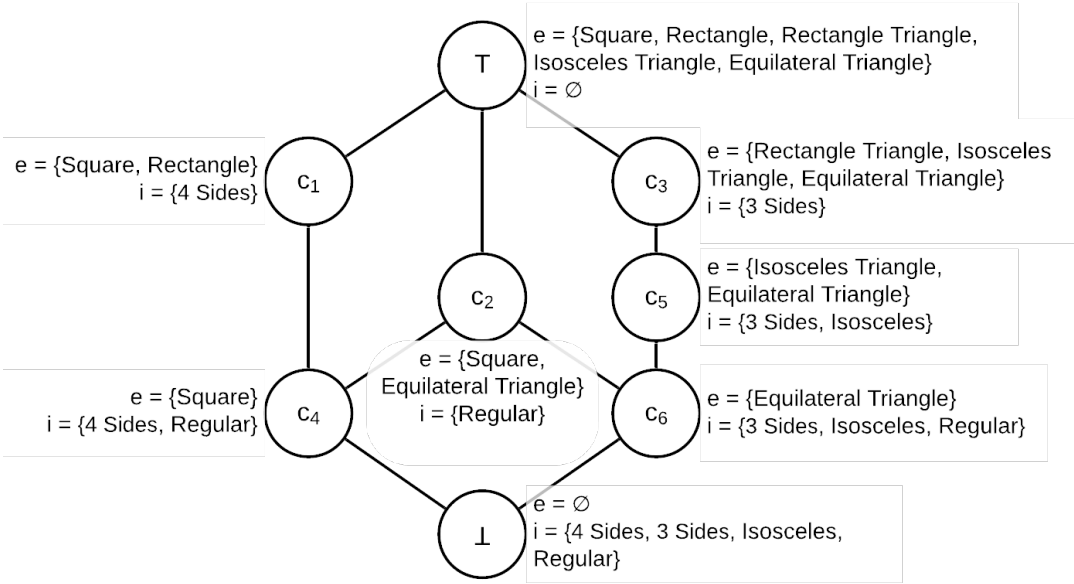


FIGURE 1.2: Lattice of the example context from Figure 1.1.

1.3.3 Formal Concept Lattices Generation Algorithms

A naive way to generate concept lattices is to follow the definition of formal concepts. The first step is to compute either the intents or the extents, and use it to build the set of formal concepts. To achieve this, we can construct the set of extents as $E = \{Y'' \text{ for all } Y \in 2^O\}$ and the set of concepts $\{\langle e', e\rangle \text{ for all } e \in E\}$. The set of intents would be $I = \{X'' \text{ for all } X \in 2^A\}$ and the set of concepts $\{\langle i, i'\rangle \text{ for all } i \in I\}$. The number of possible extents and intents are respectively $|2^O| = 2^{|O|}$ and $|2^A| = 2^{|A|}$. The size of the set of concepts is thus, in the worst case, $2^{min(|O|,|A|)}$. Therefore, assuming the time complexity of $\cdot'$ and $\cdot''$ is in $O(1)$, computing E first is in $O(2^{|O|})$ and computing I first is in $O(2^{|A|})$. In either case, there can not be a polynomial algorithm to build the set of concepts, because the size of the output (the set of formal concepts) is exponential to the size of the input (the FC). The second step of our naive algorithm is to compute the order by comparing each pair of extents with $\subseteq$.

There exist a wide variety of algorithms to generate concept lattices or at least the set of concepts. For an overview of those algorithms see, *e.g.*, [33].

1.4 Basic Background in Deep Learning

In this section, we recall some basic background in *deep learning*. For further detail on NN architectures see, *e.g.*, [42].

1.4.1 Neural Networks

Neural networks (NNs) are a class of statistical connectionist models trained using the *backpropagation* algorithm. The training is done by processing inputs with the model and evaluating the quality of the output using a *loss function*. By minimizing this loss with an optimization algorithm, the model learns to approximate the expected output. In this subsection, we detail several kinds of model designs (or *architectures*) frequently used in deep learning.

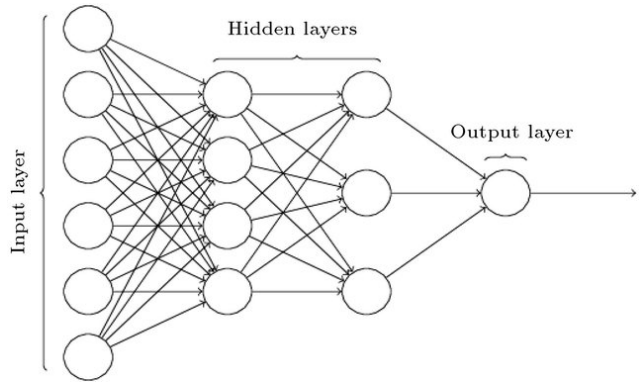


FIGURE 1.3: Multi-layer perceptron architecture. From https://www.researchgate.net/profile/Ayhan_Erdem2/publication/319309006.

A multi-layer perceptron (MLP) is the simplest architecture of NN. It is composed of multiple layers, called *feed-forward layers*. Each output of a feed-forward layer is a linear composition of all the layer's inputs. Between layers, the values are transformed by an *activation function* to increase the expressive capacity of the model. The *rectified linear unit* (ReLU) and *sigmoid* are among the most common activation functions. Figure 1.3 contains a simple diagram of the MLP.

1.4.2 Deep Learning Algorithms

To train NNs we use specific algorithms called *optimizers* to update the *parameters* of the NN model based on the value of a loss function. The parameters are the numerical values involved in the NN computations.

Training is performed by iteratively updating the *parameters* of the NN, by presenting it with *batches* (groups) of input samples and applying the optimizer based on the loss. An *epoch* is when all the data available is processed once. The process is usually repeated for multiple epochs until the model *converges*, in other words, until the performance of the model stops improving.

There exist a variety of optimizers, but the most recommended one [40] currently is Adam [27]. For an overview of existing optimizers see, *e.g.*, [40]. In our experiments, we use Adam. This particular algorithm has low computational and memory usage compared to other algorithms and is well suited for complex optimization problems.

1.4.3 Usual Loss Functions

The loss function to use depends on the kind of problems handled. When predicting specific values within a set of known values, it is standard to make the NN output probabilities of being in each of the possible values. Typical examples are classification problems and language modeling, where the model has to predict the most likely character or word among known ones. For those problems, we usually use the *cross-entropy* loss to make the predicted probabilities closer to the actual ones. In the binary case (only 2 possible values), we use the *binary cross-entropy* (BCE) which only requires the probabilities of one of the two classes. Equation (1.1) is the equation of the cross-entropy with X the set of all possible values, p the target distribution, q the predicted distribution, and $p(x)$ (or $q(x)$) the probability to have the value x according to p (respectively q). Equation (1.2) is the equation of the BCE with two values

0 and 1, but computed only using 1. Another common kind of problem is predicting real-valued data or integers, *e.g.*, image generation. In those cases, we typically use *mean squared error* (MSE) to reduce the squared distance between the predicted value and the expected one. Equation (1.3) is the equation of the MSE with p and q the target and the prediction. Other loss functions are widespread, *e.g.*, the cosine similarity, but we mainly use cross-entropy, BCE, MSE, and loss functions derived from those three.

$$H(p,q) = -\sum_{x \in X} p(x) log(q(x)) \tag{1.1}$$

$$BCE(p,q) = -p(1) \times log(q(1)) - (1 - p(1)) \times log(1 - q(1)) \tag{1.2}$$

$$MSE(p,q) = \sum_{i=1}^{|p|} (p_i - q_i)^2 \tag{1.3}$$

1.4.4 Binary Encoding and Softmax

In classification problems the possible values are usually indexed, and a specific label is represented by a binary vector. We speak of the *binary encoding* of a set of values and *one-hot encoding* of a value. Those encodings are computed with regards to the indexed set of all possible values. In our setup, the binary encoding of a set of attributes $X \subseteq A$ is the vector of size $|A|$, such that the position k of the vector contains 1 if $a_k \in X$ and 0 otherwise. The one-hot encoding of an attribute a_k is the binary encoding of $\{a_k\}$, in other words the vector of size $|O|$ full of 0 except at position k which is 1.

When performing a single-label classification, we usually generates the probablility for each value to be the correct one. To obtain this set of probabilities we usually use the *softmax* function, which transforms any finite set of numbers into a probability distribution. In other words, it transforms the values into probabilities (between 0 and 1) in such a way that the sum of all values is equal to 1 and that the proportions between the values are preserved.

1.4.5 Major Neural Networks Architectures

A *convolutional neural network* (CNN) is a NN that apply *convolutions* on an input. For an overview of CNN see, *e.g.*, [47]. Following the principle of *convolutions*, for a given input element, a CNN produces an output based on a learned *kernel* and the neighborhood of the input. It is possible to compute the size of the output and the size of the neighborhood taken into account by the CNN. A CNN can handle inputs of varying sizes, and the output size is proportional to the ones of the input. Additionally, CNNs maintain the relations between neighboring elements and is invariant to translation. In other words, a CNN will produce the same output for a given neighborhood at different locations in the input. They are widely used in image processing due to this property allowing it to learn filters to detect features independently of the position in the input. A basic CNN is shown in Figure 1.4.

The famous *long short-term memory recurrent NN* (LSTM) [17] and *gated recurrent unit NN* (GRU) [7] are architectures of a family called *recurrent NNs* (RNNs). This family of models handles sequences of inputs of variable length, and are designed to learn dependencies within the input sequence. By transmitting and updating a vector called *hidden state* from one step of the sequence to the following one, each output depends on the current input as well as the previous ones. A variant of

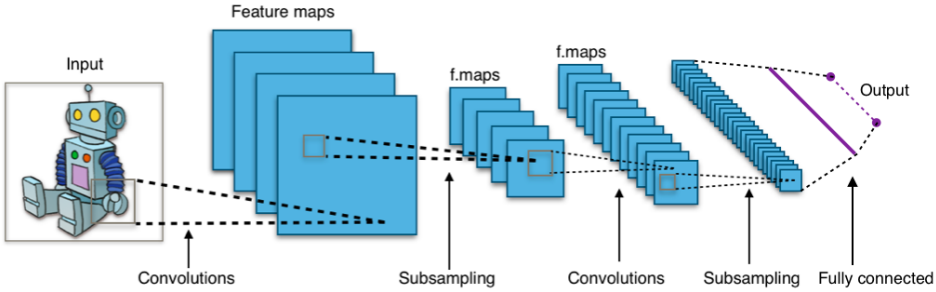
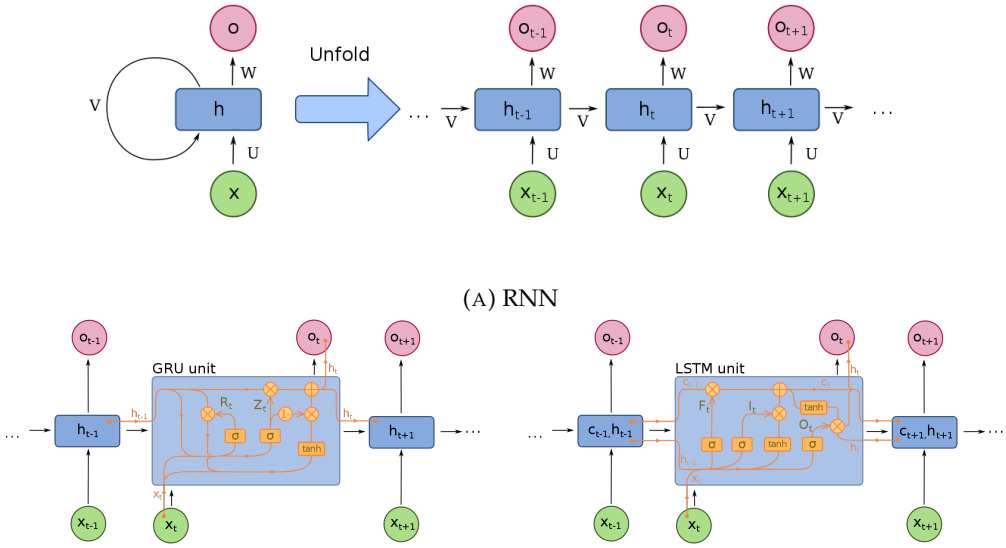


FIGURE 1.4: Example of CNN architecture. From Wikipedia.



(A) RNN

(B) GRU cell

(C) LSTM cell

FIGURE 1.5: Structure of the major RNN architectures, the input at step t is x_t, the hidden state h_t, and the output o_t. From Wikipedia.

the LSTM, called *bi-directional LSTM* (BLSTM) [17], uses a second LSTM to process the input sequence in the reverse direction. It can thus handle both forward and backward dependencies in the sequence. A major flaw of all RNNs is their inability to model long-range dependencies, with various variants like LSTM and GRU trying to solve this *memory* issue to some extent. A block diagram representing how RNN handle sequences is shown in Figure 1.5a, and the inner workings of a a GRU and an LSTM are shown respectively in Figure 1.5b, and Figure 1.5c.

Attention mechanisms [1], which have been developed to handle this issue, consider a full sequence and attribute *attention weights* to each element. The attention weights are usually computed with a *dot product* between a *query* and the elements in the sequence, though there are numerous variants of attention. The attention weights are usually used to weight the sequence directly or to compute a weighted average of the sequence. The summaries produced by the attention mechanisms are called *context*. Attention is cheap to compute (usually), and the analysis of the attention weights allows to determine the implication of each element in the final result. Due to this property, attention can be used to make an RNN's result interpretable. Attention is powerful enough to be used alone, like the *transformer network* [44] and more recently the *reformer network* [30]. An example of attention for neural machine translation is shown in Figure 1.6.

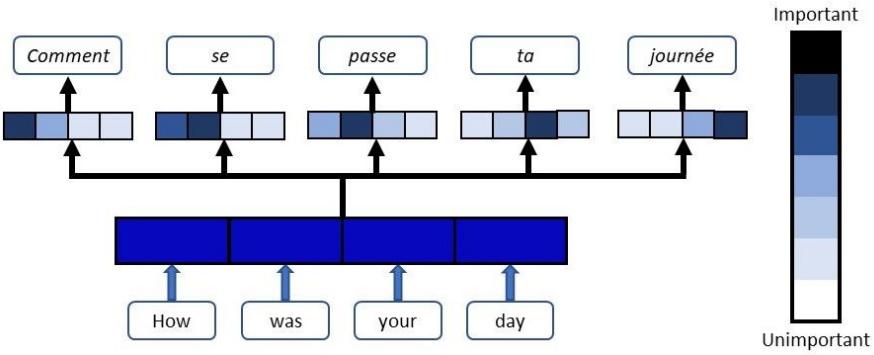


FIGURE 1.6: Example of attention mechanism for translation from English to French. From https://blog.floydhub.com/attention-mechanism/

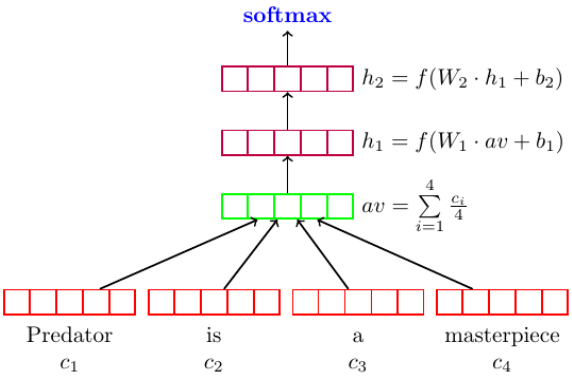


FIGURE 1.7: Example of DAN, Figure 1 from [23].

We call *unordered composition functions* operations that do not take into account the order of the input elements and can accommodate any number of input elements. Typical examples for vectors are the element-wise min, max, and average (also respectively called min-, max-, and average-pooling). Unordered composition-based models combining an unordered composition of the inputs with an MLP are called *deep averaging networks* (DANs). They have proven their effectiveness in a variety of tasks, for instance, sentence embedding [23], sentiment classification [6], and feature classification [15]. On the one hand, this family of architectures allows for varying sizes of input to be processed at a relatively low computational cost, by opposition to recurrent models like LSTM. On the other hand, the information related to the order of the input elements is lost. The DAN for sentence embedding from [23] is shown in Figure 1.7.

To summarize, CNNs, RNNs, attention mechanisms, and DANs are various architectures able to handle inputs of varying sizes, each with their advantages and drawbacks.

1.4.6 Auto-Encoders and Embeddings

Auto-encoders are a class of deep learning models composed of: *(i)* an encoder, taking some x as an input and producing a latent representation z; *(ii)* a decoder, taking z as an input and reconstructing $\hat{x}$ a prediction of x. By training the model to match x and $\hat{x}$, the model learns to compress x into z. We call the training objective matching x and $\hat{x}$ the *reconstruction loss*. Auto-encoders are one of the methods to generate

representations of data as vectors. In that case, z is called the *embedding* of x, and the real-valued space in which z is defined is called the *embedding space*.

Unlike traditional auto-encoders, *variational auto-encoders* (VAEs) [26] encode a distribution for each value of z instead of the value itself. In practice, for each element of z, the encoder produces two values: a mean μ and standard deviation σ. When training the model, z is sampled from the normal distribution defined by μ and σ. Finally, the distribution defined by μ and σ is normalized by using an additional loss term called *Kullback–Leibler divergence* (KL divergence). To make this process differentiable and be able to train the model, a method called *reparametrization* [26] is used.

VAEs are known to provide better generalization capabilities and are easier to use to decode arbitrary embeddings, compared to classic auto-encoders. This property is useful for generation, as we can train a model generating embeddings then decode them with a pre-trained VAE. A typical example of VAE, is one trained on basic geometric shapes (circles, triangles, rectangles, *etc.*), allows us to decode arbitrary embeddings: the average of the embedding of a triangle and a rectangle would give us a trapezoid (a coherent mix of a triangle and a rectangle) even if none were seen during training. Thus, VAEs have been used in a wide variety of applications to improve the quality of embedding spaces: image [26], speech [31], and graph generation [29] for example.

A constrained VAE is a kind of VAE that feeds a *constraint* vector (or *condition*) to the decoder in addition to the embedding. Because the condition already contains part of the information, the model will learn to encode the rest in the embedding. This architecture allows to voluntarily exclude part of the information form the embedding or to constrain the decoding process. An example of constrained VAE is [31], which uses the condition to specify the speaker in emotional speech. The embedding contains "anonymized" speech information, and it becomes possible to transfer emotional speech from one speaker to another by changing the condition.

1.4.7 Metric Learning

Metric learning [35] is a process used to train embedding models, by making their embedding space have properties of a metric. To achieve this, a loss is used to reduce the distance between the embeddings of equal elements and increase the distance between embeddings of different elements. Multiple losses can achieve this, such as *pairwise loss* and *triplet loss*. Those losses consider the embeddings of three elements: an input x, some x' judged equal to x, and some y different from x. They are used to minimize the distance between equal elements and maximize the distance between different elements. In some approaches [31], a predictor (typically an MLP) is used to predict a distance between the embeddings, instead of applying a standard distance directly on the embeddings.

It is possible to learn metrics on different aspects of the elements, by splitting the embedding into different segments and learn a different metric on each segment [31].

Metric learning is usually used to approximate actual metrics. However, its process can be applied to learn other kinds of measures not fitting the definition of a metric. The current paper falls in this case.

1.5 Problem Statement

Our main objective for this project is to provide a grounded proof of concept, demonstrating the feasibility of concept lattice generation using NNs. We focus on providing a fully working and general framework, able to handle unseen FCs with no technical constraint on the size. The generation performance and the explainability are also taken into account, but they are secondary. Indeed, extensive training of the model can easily boost the performance once the framework is designed. Additionally, we orient our design choices to facilitate the integration of explainable methods in future work. Finally, we do not consider the computational optimization of the methods as a major concern.

In the previous section, we saw that a formal concept lattice is defined by a set of concepts and a partial order relation between the concepts. Given the relations between the order relation $\leq$, $<$, and $\prec$, a single one of them is enough to describe the partial order between the concepts. Also, the set of concepts can be completely described using either the set of intents or the set of extents. From those two aspects, we can apply the principle of "divide and conquer" to split the difficult task of lattice generation into simpler tasks: on the one hand, generating the order between the concepts, and on the other hand, defining the concepts themselves. Additionally, the order relation can be described by graphs (*e.g.*, the graphs of $<$ and $\prec$), with the concepts being the nodes of the graph, so state-of-the-art, methods in graph generation can be used.

Two approaches can be devised to solve the tasks of generating the concepts and the order between them: (*i*) generating the order between the concepts first and then defining the concepts, and (*ii*) generating the concepts, then computing the order relation between them. Each of these approaches focuses on a different aspect of the formal concept lattice. We explore the first approach in Chapter 2 by adapting state-of-the-art methods to generate the graphs of lattices. In Chapter 4, we focus on generating the intents using the information from the FC, thank to the BoA embedding model for FCs described in Chapter 3.

Our choice of working with the intents to define the concepts comes from the assumption that $|A| \leq |O|$ in most datasets, so focusing on intents should be less costly than with the extents as fewer elements are involved. In cases where this assumption is not verified ($|A| > |O|$), we can exchange the objects and the attributes. Indeed, if we exchange them the resulting lattice is the same, except $\leq$ and $\prec$ which respectively become $\geq$ and $\succ$, and the intents and extents which are swapped with one another. In practice, this exchange of A and O corresponds to using the transposed C instead of C itself. It is easy to swap back the intents and extents in the output and to reverse the order and cover relations to get the expected lattice. In simple words, whether we design our method by focusing on intents or extents, we can adapt it to the other case with simple transformations.

Chapter 2

Initial Approach: Graph Generation

In this chapter we explore the reconstruction of the lattice as a relation between formal concepts. As a relation corresponds to a graph, we can use the state-of-the-art graph-generation methods described in Section 2.1 to generate the lattice.

Our goal which is to learning a single model to generate all the lattices leads us to focus on models able to generate families of graphs. In particular, we selected GraphRNN [51], for the reasons explained in Subsection 2.1.3, and adapted it into a constrained VAE presented in Section 2.3. We also adapted their graph encoding method to represent lattices as we describe in Section 2.2.

We performed preliminary experiments with this method as described in Section 2.4. However, the challenge of how to use the FC as a condition for the VAE, together with the relatively poor generation performance, led us to put aside the graph generation approach and focus on generating the concepts first, as described in Chapter 4. The BoA embedding framework, described in Chapter 3, is the result of our tentative to make the FC usable as a condition for the VAE. As we did not finalize the approach, the corresponding code is not present in our GitLab repository.

2.1 State of the Art of Graph Modeling

In recent years, there has been an explosion of approaches to model graphs with NNs. Typical applications are knowledge graph processing and community graph detection. There exist a variety of approaches to graph modeling in the literature. In this section, we describe the main approaches of graph representation and generation, and their principal characteristics. For further details see, *e.g.*, [46] or the repository `https://github.com/dsgiitr/graph_nets`.

The various approaches for graph modeling using NNs vary along 2 major aspects: the data representation method and the kind of objects that the framework is capable of modeling, *e.g.*, individual nodes or whole graphs. Firstly, the representation and processing of the graph differ in how the two major aspects of the graph are represented. On the one hand, the structure of the graph is the adjacency between the nodes. It can be represented as an adjacency list, an adjacency matrix, a graph spectrum (see Subsection 2.1.2), or implicitly by considering the neighborhood of a node. On the other hand, nodes can be represented either implicitly by only considering the structure of the graph, by an arbitrary identifier like a random value, or by a set of features or labels. As a reminder, *features* of nodes in a graph are a set of properties contained in nodes, *e.g.*, the name, age, and email address in a social network. *Labels* are textual or numerical denominations of the nodes and edges, and can be seen as a specific kind of features. Secondly, graph NN can represent either

nodes or whole graphs. We detail those two kinds of graph NNs respectively in Subsection 2.1.1 and Subsection 2.1.2, along with major graph NNs approaches. We explain why we chose to focus on GraphRNN in Subsection 2.1.3.

2.1.1 Node-Centered Approaches

The first group of graph NNs focuses on representing nodes.This kind of NN can be trained on a specific graph to provide representations for this graph's nodes [38, 39]. It is used to generate local representations in particular for large and complex graphs, *e.g.*, generating entity embeddings for knowledge graphs or node embeddings for community graphs. Graphs NNs representing nodes can also be used to represent whole graphs, by combining the representation of the nodes. However, it is also possible to learn a more general model to represent nodes by training on multiple graphs [28].

A first method, proposed in DeepWalk [39], is to mimic natural language processing (NLP) methods for learning sentences. The neighborhood of a node is *linearized*, by taking a sequence of nodes obtained by randomly walking through the graph. The sequences of nodes are then processed as if they were sentences, with SkipGram [38] in the case of DeepWalk.

A second approach is *message passing*, in which a node representation is built iteratively. The nodes' initial representation can be the features of the nodes (represented as vectors), or randomly generated values if there are no given features for the nodes. GraphSAGE [20] and Graphite [19] are examples of message passing neural algorithms, with [19] including a variational version of the Graphite.

A third framework is *graph convolution*, an adaptation of CNNs from grid-like structures like images to the more irregular structure of graphs. In practice, the kernel is applied to the neighborhood of a node as defined in graphs, instead of the spatial neighborhood of a cell in a matrix. Graph Convolutional Network (GCN) [28] is the basic architecture implementing this approach, and Hypergraph Neural Network [11] is an adaptation of GCNs to hypergraphs. Graph Attention Network (GAT) [45], Masked GCN [48], SPAGAN [49], Hierarchical GAN [25] are improvements of GCNs with attention mechanisms and other similar techniques. Those improvements allow the NN to select neighboring nodes based on the current node and the content of the neighbors.

2.1.2 Whole Graph Approaches

Graph NNs representing whole graphs are used to represent graphs from a specific *family* (or class) of graphs. In other words, they are trained to handle graphs sharing some specific structural properties, *e.g.*, community graphs. This kind of graph NNs are used to perform tasks on a whole graph at once, *e.g.*, classification of a graph and generating graphs of a specific family.

A first method is *spectral convolution*, introduced in [5] and with a more recent implementation called ChebNet [43]. Spectral convolution is based on spectral graph theory, which represents a graph's adjacency using a spectrum. The spectrum of a graph provides a complete view of the graph's adjacency. A detailed introduction to spectral graph theory can be found, *e.g.*, in [8]. Spectral convolution is an application of CNNs on this spectrum of the graph.

A second framework is *hierarchical graph pooling*, presented in [50]. This method is an iterative simplification of the graph by grouping strongly related nodes using

clustering and pooling methods. Once a single group remains, the graph is fully simplified.

A third approach is to use a VAE directly on the adjacency matrix of a graph, as implemented in GraphVAE [29] and Constrained GraphVAE [36]. By flattening the adjacency matrix to a vector, it becomes possible to apply a VAE directly on the adjacency. This method performs well at generating graphs of specific families, despite being less involved than the other methods presented in this section. Indeed, it doesn't rely on specific properties of graphs to establish its architecture.

A fourth method is to use recurrent models on a linearization of the graph, as done in GraphRNN [51]. In this approach, the graph is first transformed into a sequence of nodes, and the nodes are then processed iteratively. GraphRNN processes each node of the sequence by computing its edges with the previous nodes in the sequence. They rely on a breadth-first search to linearize the graph to make connected nodes close in the sequence.

2.1.3 Advantages of GraphRNN for Lattice Generation

As a reminder, our goal is to learn a single neural model to generate lattices in general. We want the model to produce a whole lattice for each FC presented. From those constraints, we focus on graph NNs able to handle whole graphs at once. The architecture should also be able to generate graphs, and only GraphVAE, Constrained GraphVAE, and GraphRNN fit our criterion.

A drawback of GraphVAE and Constrained GraphVAE is that they are limited in the size of the graph they can handle. Indeed, VAE has a fixed-sized input and output, so the two approaches are unable to handle graphs larger than the graphs they are trained on. Conversely, the recurrent nature of GraphRNN makes it very flexible with regard to the size of the generated graphs.

A final argument in favor of GraphRNN is that it is easily extendable with node features. In our case, node features are the intent and extent of the concepts, and they are very important for our goal because they define the concept. In practice, when generating a new node, we would add a network to generate the intent or the extent to the link generation network of GraphRNN, as we explain in Subsection 2.3.2.

GraphRNN shows state-of-the-art performance when generating graphs of various families, and can handle any size of graphs. However, it is not designed to generate specific graphs on request, as we would like to do to generate lattices based on the FCs. Conversely, constrained VAEs (such as Constrained GraphVAE [36]) are generative models that can be constrained. Using such a model would allow us to generate lattices under the constraint of the FCs. To compensate for the limitations of GraphRNN with regards to our goal, we decided to adapt GraphRNN into such a constrained VAE. As described in Section 2.3, we first adapt GraphRNN into an auto-encoder, which is later adapted into a VAE and then further modified as a constrained VAE.

2.2 Transforming Lattices into Matrices

When training a NN on non purely numerical data, lattices in our case, it is necessary to first represent the data in a numerical form. The choice of the data, as well as the quality of the representation, noticeably affect the quality of the resulting model. Typically, a bad representation makes learning hard if not impossible, while a well-thought representation of the data is already half the job done. Also, the model will

	Size	# Concept	Density of the triangular adjacency Graph of $\prec$	Graph of $\leq$
Mean $\pm$ std.	All	27.35 ± 24.76	0.12 ± 0.07	0.22 ± 0.09
	5×5	7.28 ± 1.88	0.18 ± 0.02	0.30 ± 0.03
	10×10	29.49 ± 6.30	0.07 ± 0.01	0.17 ± 0.02
	10×20	66.00 ± 11.55	0.038 ± 0.005	0.11 ± 0.01
	20×10	64.68 ± 13.05	0.039 ± 0.006	0.11 ± 0.01
Range	All	1 to 117	0 to 0.25	0 to 0.39
	5×5	1 to 15	0 to 0.25	0 to 0.39
	10×10	15 to 64	0.04 to 0.11	0.12 to 0.29
	10×20	38 to 114	0.025 to 0.054	0.08 to 0.15
	20×10	37 to 117	0.024 to 0.058	0.08 to 0.16

TABLE 2.1: Descriptive statistics on the dataset of randomly generated contexts.

perform better and be better at generalizing when trained on more and more varied data.

We decide to learn on randomly generated FCs and the corresponding lattices. The main reason is that gathering and preparing large amounts of usable real-world FC is costly, while generating random data is cheap. We can afford to use randomly generated FCs because we want to learn the general process of FCA, which should behave the same no matter the source of the data.

In this section, we describe the lattice data we used for our experiments and how we encode it as matrices. First, we explain our generation process in Subsection 2.2.1. We then explain how we adapted the breadth-first search ordering method of GraphRNN in Subsection 2.2.2. We then discuss the encoding of the adjacency in Subsection 2.2.3 and the intents and extents in Subsection 2.2.4.

2.2.1 Data Generation and Dataset

The dataset used for training the GraphRNN models is composed of 4000 randomly generated FCs and the corresponding lattices computed using the Coron system[1]. To generate a context of $|O|$ objects and $|A|$ attributes we sample $|O| \times |A|$ values from a Poisson distribution and apply a threshold of 0.3. The values under this threshold correspond to 1 in the context, which leads to a density (the portion of entries equal to 1) around 0.3. Note that the random generation process may result in empty rows and columns, which will be dropped by Coron if they are at the extremities of the context. For this reason, the actual size of the generated context may be smaller than the requested one. For the training phase, a development set of 10% of the training set is randomly sampled from the training set. We generate different sizes of contexts: 2000 of 5×5, 1000 of 10×10, 500 of 10×20, and 500 of 20×10 ($|O| \times |A|$). The statistics of the generated dataset are reported in Table 2.1. The dataset itself and the generation process are available in a GitLab repository[2].

[1] http://coron.loria.fr/site/index.php
[2] https://gitlab.inria.fr/emarquer/random-lattice-dataset

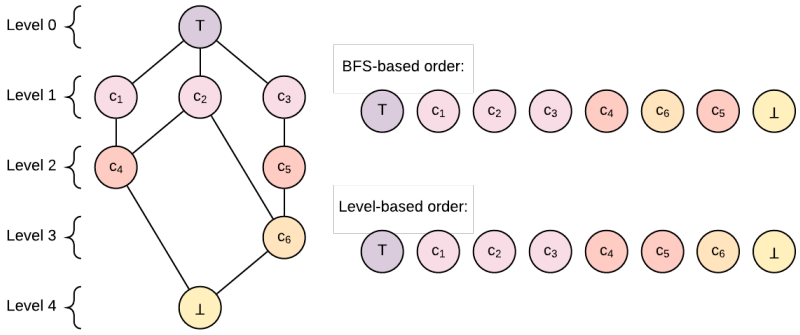


FIGURE 2.1: Lattice from Figure 1.2 organized in levels and the corresponding level-based and BFS-based orders.

2.2.2 From Breath-First Search to Level Ordering

In GraphRNN [51], a graph is represented by its adjacency. The graph is flattened into a sequence with a *breadth-first search* (BFS), each element of the sequence describing a node by defining its adjacency with the previous nodes in the sequence. In practice, we have a sequence $S^\pi = (S_1^\pi, \ldots, S_n^\pi)$ for a graph of n nodes ordered by π, with $S_i^\pi \in \{0,1\}^{i-1}$ an adjacency vector for the node i with the nodes 1 to $i-1$. The breadth-first search order π ensures that the distance between the two adjacent nodes in the sequence is small, as demonstrated in [51]. This method is designed for undirected graphs but also works for directed graphs, under the condition of using $S_i^\pi \in \{-1,0,1\}^{i-1}$ to represent both incoming and outgoing edges.

We adapt the methodology of [51] to lattices by defining an ordering of concepts based on what we call *levels*. First, we define levels in Definition 1.

Definition 1 *We iteratively define L, the indexed set of levels and a partition of C, such as a concept $c_1 \in C$ belongs to the level $l_i \in L$ if and only if the highest level of every other concept $c_2 \in C, c_2 \prec c_1$ belongs to is $i-1$.*

For a given lattice, L can be demonstrated to be unique. Due to the way levels are defined, two concepts within the same level are not comparable with the partial order $\leq$. Additionally, if a concept c_1 belongs to a lower level than a concept c_2, either c_1 and c_2 are not comparable or $c_2 \leq c_1$. Finally, $\perp$ is the only concept in level 0 and $\top$ is the only concept in the highest level. We can now define the *level-based order* $\mathcal{O}^L = (c_0, \ldots, c_n)$ an ordering of C based on L. This order is constructed by concatenating the levels from l_0 to $l_{|L|}$. The order of the concepts within each level does not matter and is thus arbitrarily chosen. The levels of the example from Figure 1.2 and the corresponding level-based order are shown in Figure 2.1.

Using this new order $\mathcal{O}^L$ with the method of [51] maintains most of the properties demonstrated in [51]. However, the distance between related nodes (concepts in our case) is less constrained, as concepts of the last layers can be related by $\prec$ with concepts of first layers. Conversely, using level-based ordering instead of BFS allows the adjacency vectors to contain only outgoing edges and no incoming ones. Indeed, for all $c_1, c_2 \in C$ such that $c_1 \prec c_2$, if $c_1 \in l_i$ then $c_2 \in l_j$ with $i < j$. In other words, a concept is only related to previous elements in the order.

In our example, BFS and level-based ordering produce different results. Indeed, c_6 appears before c_5 with BFS, so we need to represent the relation of c_6 and c_5 with an incoming edge from c_6 in the adjacency list of c_5. However, in the level-based ordering c_6 appears after c_5 and there is no need for incoming edges.

2.2.3 Encoding the Lattice Adjacency

The lattice adjacency can be encoded in multiple ways. A first method is to directly consider the sequence of adjacency vectors of different sizes as in [51]. A second method consists in concatenating the adjacency vectors, and a third method is to use the adjacency matrix as described in the following paragraph. For practical purposes, we tend to represent the data as vectors or matrices, so we focus on the last two representations. The concatenation of the adjacency vectors is the most optimal in terms of used space, while the adjacency matrix separates the adjacency of each node across one of the dimensions. The adjacency matrix provides additional benefits, like an easier visualization and simpler manipulation of the adjacency. However, the adjacency matrix requires twice the space of the concatenated adjacency vectors, as the upper triangular matrix is not used.

Basic Adjacency Matrix

We define a matrix $\mathbf{L}$ such that the entry at row i and column j, $\mathbf{L}_{i,j} = S^{\mathcal{O}L}_{i,j}$, with $S^{\mathcal{O}L}_{i,j}$ the element at position j in $S^{\mathcal{O}L}_{i}$. Where $S^{\mathcal{O}L}_{i,j}$ is not defined ($i \leq j$), $\mathbf{L}_{i,j} = 0$. In other words, we fill a lower triangular matrix with the adjacency vectors. The resulting lower triangular matrix is the adjacency matrix in the case of $\prec$ and $<$. For $\leq$, the diagonal ($i = j$) of the matrix of $<$ must be set to 1.

When using a model to predict this matrix, we are in a binary classification problem between two classes: NO EDGE (0) and EDGE (1). The model can thus be trained using BCE.

Adjacency Matrix *à la* Sequence Modeling

GraphRNN is an architecture designed to process a sequence of nodes, each represented by a sequence of edges. Traditionally when using RNNs to process sequences we use special values to mark boundaries in the sequence, *e.g.*, the start and end of the sequence. A special value is also dedicated to padding the sequence. This padding value is used to make the sequence of a batch have the same size.

We also propose to make use of the empty space in the upper triangular matrix. The transpose of the adjacency matrices of $\prec$, $<$, and $\leq$ are respectively the adjacency matrices of $\succ$, $>$, and $\geq$. For our triangular matrix containing only outgoing edges from the nodes, the transposed matrix represents the incoming edges. This transposed matrix is upper triangular, and we use it to fill the unused space of the lower triangular matrix. Using this construction, we have redundancy between the lower and upper triangular matrices. This redundancy can prove beneficial for our task, as we can use the upper triangular matrix to check the results of the lower one.

For each node of the graph, we have a sequence of values as follows:

1. a *start of sequence* (SOS) value;
2. the sequence of outgoing edges, either 1 if there is an edge or 0 otherwise;
3. a *middle of sequence* (MOS) value;
4. the sequence of incoming edges, either 1 if there is an edge or 0 otherwise;
5. an *end of sequence* (EOS) value;
6. as many *padding* (PAD) values as necessary to reach the size of the largest element in the batch.

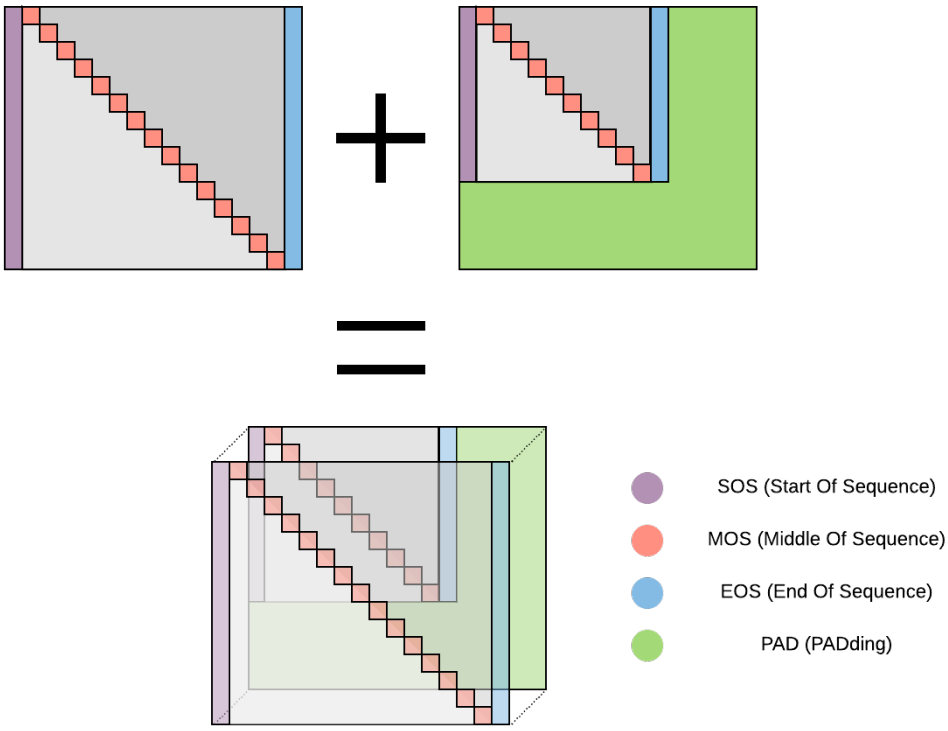


FIGURE 2.2: Diagram of a batch of 2 samples encoded using the 6 values NO EDGE (0) and EDGE (1) for the adjacency, SOS, MOS, EOS, and PAD.

We also add sequences full of PAD in the sequence of nodes to reach the size of the largest element in the batch. The resulting data takes the form of a matrix containing one of 6 values in each entry: NO EDGE (0), EDGE (1), SOS, MOS, EOS, and PAD. An example of a batch encoded using this process is shown in Figure 2.2.

When using a model to predict this matrix, we are in a multi-label classification problem between 6 classes corresponding to the possible values. The model can thus be trained using cross-entropy.

2.2.4 Encoding the Concepts

Concepts can be represented using either their intents, their extents, or both. In FCA, there are two ways to encode the intents and the extents: the *full encoding* and the *narrow encoding*. Using the full encoding, all the attributes (or objects) of the intent (respectively extent) are used to represent a concept. The narrow encoding however only use the attributes (or object) that "appear" in the concept, in other words, the attributes (respectively objects) that are present in the concept's intent (respectively extent) but not in the ones lower (respectively higher) according to the partial order $\leq$. To reconstruct the full intent (respectively extent) of a concept from the narrow encoding, we take the set of all the attributes (respectively objects) in the narrow encoding of the concept and the ones of concepts lower (respectively higher) according to $\leq$. An interesting property of this narrow representation is that each attribute (and object) appear only once in the lattice. Figure 2.3 shows the narrow encoding of the example lattice from Figure 1.2.

We design two numerical representations for the concepts. The first one is a matrix based on the full encoding. The entry at row i and column j is 1 if the attribute

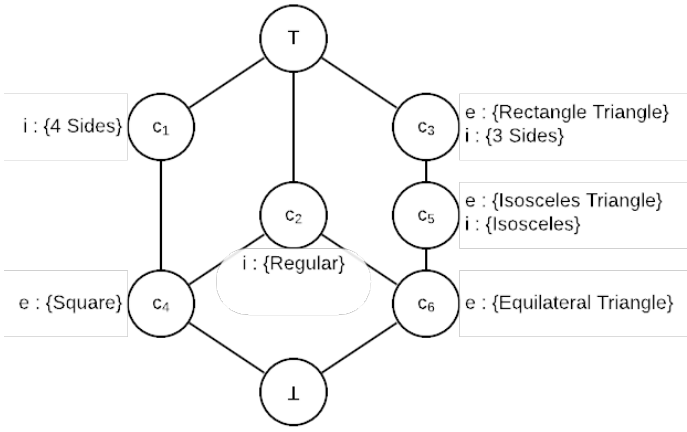


FIGURE 2.3: Narrow encoding of the example lattice of Figure 1.2.

a_j (or object o_j) is present in the intent (respectively extent) of the concept indexed i in $\mathcal{O}^L$, and 0 otherwise. The second representation is based on the narrow encoding. It takes the form of a vector, with each entry corresponding to an attribute (or object). The value of an entry is the index in $\mathcal{O}^L$ in which the corresponding attribute (respectively object) "appears".

2.3 Using GraphRNN for Lattices Modeling

The architecture we propose to model lattices as graphs is an implementation of GraphRNN as a constrained VAE. We first detail the workings of the original GraphRNN in Subsection 2.3.1 and then present our adapted architecture in Subsection 2.3.2.

2.3.1 GraphRNN

This subsection is a quick overview of the inner workings of GraphRNN. For a detailed explanation, see [51].

GraphRNN is a model for graph modeling composed of 2 components: a *graph-level* GRU to generate nodes, and a *node-level* GRU to generate edges. For each step of the graph-level model, the node-level model generates the edges with the previous nodes in the sequence. In other words, the node-level model predicts the adjacency vector S_i^π described in Subsection 2.2.2. The adjacency vector predicted by the node-level model is taken into account for the next step of the graph-level model. The model can be considered in two phases: training and inference. They respectively correspond to training the model and using the model to generate new graphs.

In practice, the graph-level model generates a representation of a node's adjacency at each step, as a fixed-sized vector serving to initialize the node-level model. This node representation is used as the first hidden state of the node-level RNN. At each step, the node-level model predicts the probability of an edge existing between the currently processed node and one of the previous nodes. During inference, this probability is used to randomly sample the edge, which is then used as the input of the next step of the node-level model. During training the true previously edge

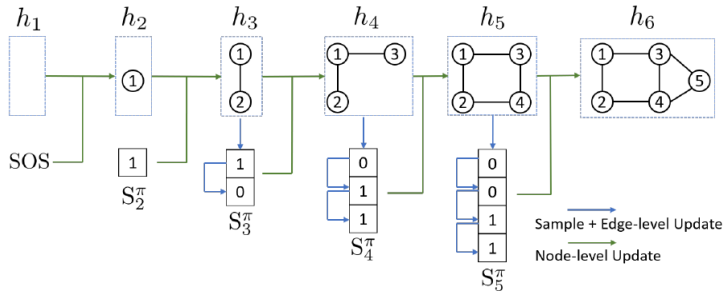


FIGURE 2.4: GraphRNN at inference time. Figure 1 from [51].

is used instead. Figure 2.4 is an example of how the model unfolds when during inference.

The TensorFlow (an alternative to PyTorch) implementation of GraphRNN is available in the Snap Stanford repository[3].

2.3.2 GraphRNN Constrained VAE

To have a model able to generate lattices from FC based on the graph structure of the lattice, we decide to split the problem. The first half of the problem is to learn a way to accurately represent lattices in a way that can be used to decode them. For that, we use an auto-encoder architecture to learn an embedding of lattices, and in particular a VAE. Indeed, as mentioned in Subsection 1.4.6, VAEs are more suited to generation problems like ours than traditional auto-encoders because of the properties of the embedding space. Once we have a representation of lattices, we can tackle the second half of the problem: to use an FC together with the decoder half of the VAE to generate the lattice corresponding to the FC. In practice, we have 2 main methods to achieve this. On the one hand, we can use an additional model to generate an embedding of the FC in the same embedding space as the lattices. On the other hand, we can use a constrained VAE with the FC as the condition. With this second option, the condition can contain other information in addition to the one from the FC, and the FC embedding is not necessarily in the lattices' embedding space, making it a more favorable option. However, the core of the problem stays the same: we need an embedding of the FC. The whole pipeline is schematized in Figure 2.5.

The development plan is in 3 steps: *(i)* building a VAE for lattices using the original GraphRNN design, *(ii)* integrating the intent generation to the VAE, by extending the original GraphRNN design and *(iii)* integrate the condition system. The first two steps of the approach allow our model to first learn the representation of lattices in an auto-supervised manner, thanks to the auto-encoding principle. At the issue of those two steps, the model should have learned "what is a lattice". The third step is for developing the embedding of the FC. This development process allows us to adapt the approach if one of the steps do not provide acceptable results. Indeed, if the model does not manage to capture the basic structural properties of lattices (without the intents), it is unlikely that it will be able to handle the full complexity of the structure of lattices with the intents. Similarly, if the model has trouble handling the full complexity of lattices, we won't be able to use it to generate lattices from FCs.

[3] https://github.com/snap-stanford/GraphRNN

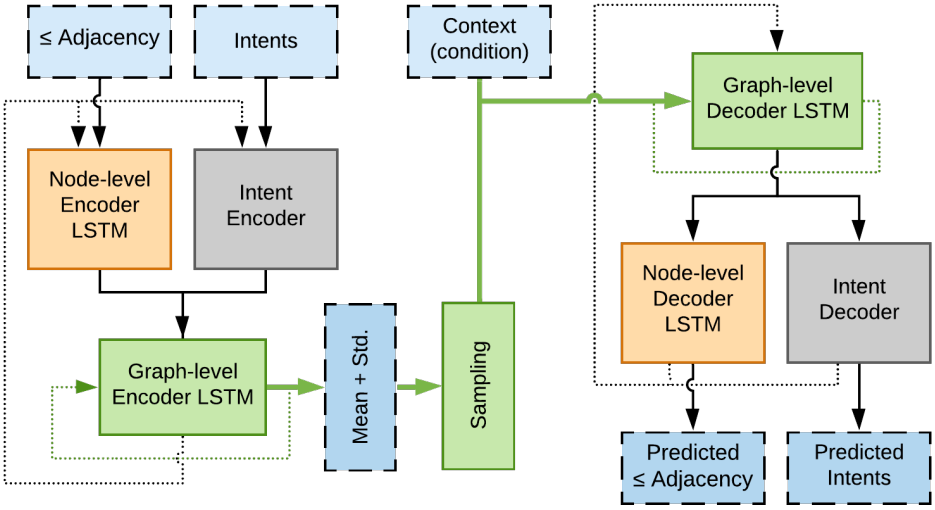
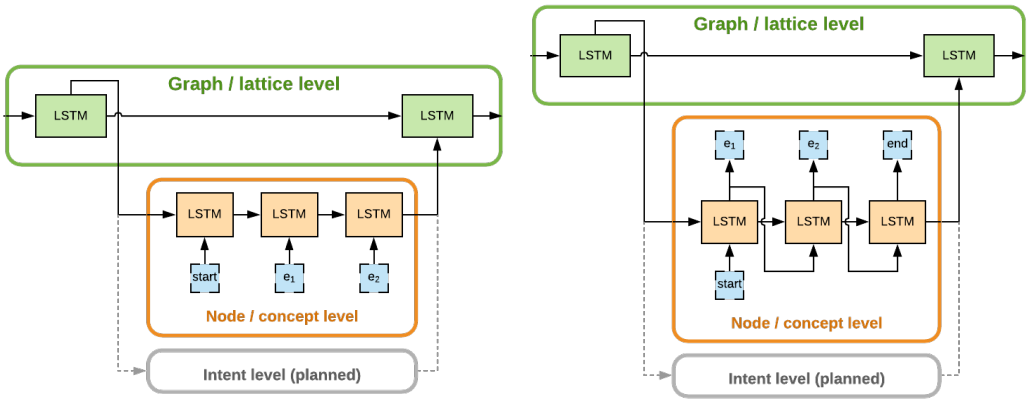


FIGURE 2.5: Adapted GraphRNN Constrained VAE.



(A) Encoder (B) Decoder

FIGURE 2.6: Details of the adapted GraphRNN, with edge and intent generation.

Our auto-encoder is composed of two GraphRNNs, one for the encoder and the other for the decoder. The encoder GraphRNN reads the graph, and the last hidden state of the graph-level RNN is used to build the embedding. The decoder GraphRNN is initialized with the embedding as the first hidden state of the graph-level RNN and generates. To handle the concept intents, an intent model is added alongside the node-level model. Similarly to the node-level, this second model takes the output of the graph-level model as its input. Block diagrams of the folded encoder and decoder are shown in Figure 2.6.

2.4 Preliminary Experiments and Change of Approach

To determine the feasibility of the proposed architecture, we first experimented with the reconstruction performance of our VAE without node features, in other words without intent.

$$\begin{bmatrix} 2 & 3 & 1 & 1 & 0 & 0 & 0 & 0 & 4 \\ 2 & 1 & 3 & 0 & 1 & 0 & 1 & 0 & 4 \\ 2 & 1 & 0 & 3 & 0 & 1 & 0 & 0 & 4 \\ 2 & 0 & 0 & 1 & 3 & 1 & 0 & 0 & 4 \\ 2 & 0 & 0 & 1 & 0 & 3 & 0 & 1 & 4 \\ 2 & 0 & 1 & 0 & 1 & 0 & 3 & 1 & 4 \\ 2 & 0 & 0 & 0 & 0 & 1 & 1 & 3 & 4 \end{bmatrix} \quad \begin{bmatrix} 2 & 3 & 1 & 1 & 0 & 0 & 0 & 0 & 4 \\ 2 & 1 & 3 & 0 & 0 & 0 & 1 & 0 & 4 \\ 2 & 1 & 0 & 3 & 1 & 1 & 0 & 0 & 4 \\ 2 & 0 & 0 & 1 & 3 & 0 & 1 & 0 & 4 \\ 2 & 0 & 0 & 1 & 0 & 3 & 0 & 1 & 4 \\ 2 & 0 & 1 & 0 & 1 & 0 & 3 & 1 & 4 \\ 2 & 0 & 0 & 0 & 0 & 1 & 1 & 3 & 4 \end{bmatrix} \quad \begin{array}{ll} 0: & \text{NO EDGE} \\ 1: & \text{EDGE} \\ 2: & \text{SOS} \\ 3: & \text{MOS} \\ 4: & \text{EOS} \\ 5: & \text{PAD} \end{array}$$

(A) Prediction (B) Gold (expected output) (C) Legend

FIGURE 2.7: Example of lattice reconstruction in the early stages.

Relation	Accuracy	Precision	Recall	F1
$\prec$	0.80 ± 0.05	0.50 ± 0.55	0.078 ± 0.089	0.13 ± 0.15
$\leq$	0.72 ± 0.03	0.50 ± 0.55	0.046 ± 0.055	0.084 ± 0.100

TABLE 2.2: Prediction performance of the GraphRNN auto-encoder, for adjacency matrices of the graphs of $\prec$ and $\leq$. The prediction performance for entries is reported (mean $\pm$ std.).

2.4.1 Reconstruction Performance Without Features

On a few samples of small size, the reconstruction performance of the VAE was satisfying. Examples of reconstructions an early stage proof of concept model are shown in Figure 2.7. This proof of concept model was trained on less than 20 samples smaller than 5 objects and attributes. However, the prediction performance on our larger dataset described in Subsection 2.2.1 is not as good. We experimented with various variants of the model, but without major improvement in the performance. Those changes include, but are not limitted to, predicting $\leq$ or $\prec$, completely sharing the parameters of the decoder and the encoder, and pretraining. The performance on the development set of our best model, detailed in Table 2.2, stays relatively low, with an average F1 score of 0.13 at best when reconstructing the graph of $\prec$.

It is hard to compare those results with those presented in [51] as we do not use the same evaluation tools. Indeed, GraphRNN is evaluated in [51] by comparing structural statistics of predicted graphs with those of true graphs of the modeled family. Conversely, because our goal is to correctly reconstruct a specific lattice and not any lattice-like graph, we use measures of prediction performance, *e.g.*, the F1 score. However, given the performance of GraphRNN demonstrated in [51], we expected higher reconstruction performance, with an F1 score above 0.7 at least.

2.4.2 Formal Concept Representation and Change of Approach

To generate the lattice from the FC we decided to add a condition to our VAE. Ideally, this condition should be the FC in some form. Because the usual constrained VAE architecture requires fixed-sized condition vectors, the representation of the FC has to be of fixed size for any FC. We could then generate the concept lattice using exclusively the FC as the input, by using some default embedding instead of the output of the encoder. This process can be seen as being able to generate any lattice, and using the condition to specify which lattice we want: the one corresponding to our FC. Another option would have been to generate lattice embedding from the FC, in the embedding space defined by our GraphRNN VAE.

Obtaining a detailed enough representation of the complexity of the FC is challenging, because the information defining the lattice is not directly accessible. Additionally, we have constraints on the size of the representation due to our needs. Focusing on this problem led us to design BoA presented in Chapter 3. The results of the BoA model together with the poor performance of the basic auto-encoder led us to modify our lattice approach. The new approach is detailed in Chapter 4.

Chapter 3

Bag of Attributes: Embeddings for Formal Contexts

It is essential to represent FCs, to reproduce FCA using neural networks. Ideally, we want to have a general embedding framework for FCs capable of handling data of arbitrary dimensions while encoding much of the contextual information.

FCA2VEC [9] is, to our knowledge, the only framework to generate embeddings of objects and attributes from an FC. This approach, which we explain in Section 3.1, has several limitations. Firstly, the embeddings for objects and attributes are not defined in the same embedding space, which can be problematic when processing objects and attributes together. Secondly, the embedding models need to be trained separately on every processed FC, which is costly. Thirdly, there is no guaranty that the resulting embeddings can be used to generalize across FCs, which is blocking for our goal of developing a single model able to handle FCs.

To overcome these limitations, we propose an embedding framework for FCs. As mentioned in Section 1.5, we focus on attributes and intents. To design this framework, we asked ourselves what attributes are, and in particular, which aspects of the attributes should interest us to reproduce FCA. We decided to focus on how attributes interact with each other, and more precisely, which attributes appear together and how often. This answer is based on the following observation: attributes that always appear together in the same dataset appear in the same intents, in the same concepts. However, we are not interested in the order of the attributes, because changing said order in an FC will not change the resulting lattice.

We detail the resulting architecture, *Bag of Attributes* (BoA), in Section 3.2, and present experimental results in Section 3.4. BoA is the object of an article [37] published in the 8th FCA4AI ("What can FCA do for Artificial Intelligence?") workshop[1]. A significant portion of the content of this section is a reformulation of the article [37].

3.1 State of the Art: FCA2VEC

Binary FCs are binary tables. There exist a wide variety of *rank lowering* methods to represent such tables, like *latent semantic analysis* in NLP. The resulting representation is usually a pair of sets of vectors, one for the rows and one for the columns of the table. However, such methods do not take into account the properties manipulated by FCA, such as the closure operator or the formal concepts.

To our knowledge, the only embedding framework specialized for FCs and based on FCA is FCA2VEC [9] by Dürrschnabel *et al.*. We explain the major aspects of their approach in this section. For further detail, see the original article.

[1] https://fca4ai.hse.ru/2020/

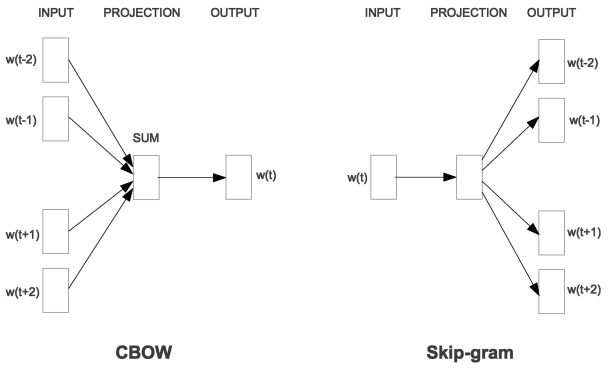


FIGURE 3.1: Continuous Bag of Words (CBoW) and skip-gram architectures from word2vec. Figure 1 from [38].

In Subsection 3.1.1 and Subsection 3.1.2, we describe the embedding architectures based on FCA's closure operator proposed in [9], namely *object2vec*, *attribute2vec*, and *closure2vec*. Finally, we describe in Subsection 3.1.3 the methods used in the article to evaluate object2vec and attribute2vec. Note that all FCA2VEC models are designed to build embeddings of small dimensions (2 or 3 elements only).

3.1.1 Object2vec and Attribute2vec

Object2vec and *attribute2vec* are proposed in FCA2VEC to provide embeddings of objects and attributes using respectively the extents and the intents. They are based on the idea "to interpret two objects to be more close to each other if they are included in more concept extents together." [9]

Based on this principle, the authors adapt the *word2vec* [38] distributional word embedding framework. Word2vec learns to associate a word with the words it appears with often, by learning to predict either the context of a word given the word (*skip-gram* variant) or a word based on its context (*CBoW* variant). This principle is schematized in Figure 3.1. By interpreting the objects as words and the extents as sentences and applying word2vec, an object will be associated with objects often appearing in the same extent. The resulting model is called object2vec. Similarly, attribute2vec is obtained by using attributes as words and intents as sentences.

The samples used to train object2vec are generated by taking, for each extent e, for each object $o_1 \in e$, either:

- all the pairs $\langle o_1, o_2 \rangle$ for all the other objects o_2 in the extent ($o_2 \in e/o_1$[2]), for the skip-gram variant;
- the pair $\langle e/o_1, o_1 \rangle$, for the CBoW variant.

In practice, o_1 and o_2 are represented by their respective one-hot encoding, and e/o_1 is represented by the element-wise average vector of the one-hot encoding of its elements.

It is interesting to note that for an extent of size $|e| = 5$, 5 samples are generated for CBoW, while for skip-gram 5×4 (2 among 5) samples are generated. For FCs with large amounts of objects and concepts, the number of samples used to train the skip-gram variant is explosively large using this method.

[2] As a reminder, e/o_1 stands for the set e without the element o_1.

Dataset	# Objects	# Attributes	Density	# Concepts
ICFCA*	263	8442	0.005	680
Wiki44k	45021	101	0.04	21923

TABLE 3.1: Descriptive statistics on ICFCA* and wiki44k, the datasets used to evaluate object2vec and attribute2vec. Table 1 from [9].

3.1.2 Closure2vec

Closure2vec is proposed in FCA2VEC based on the result of [41]. Rudolph demonstrates that for any closure operator on two sets X and Y, there exists an MLP able to perfectly encode said closure operator. The MLP of the closure operator from X to X has an input size of $|X|$, a first layer with $|Y|$ neurons and a second one with $|X|$ neurons. This general result is applicable to FCA's closure operator, *e.g.*, $\cdot''$ on the set of attributes can be encoded with a 2 layer MLP with an input size of $|A|$, a first layer with $|O|$ neurons and a second one with $|A|$ neurons. Such a model, once defined, can be learned using deep learning algorithms.

Closure2vec is designed to take two sets of attributes (for example, two objects) and compute an embedding for each of them. It is then trained to match the *closure Hamming distance* (see Definition 2) between the two sets of attributes to the distance between the corresponding embeddings. The distance between the embeddings is computed using either the Euclidian or the cosine distance. In practice, closure2vec uses a 2 layer MLP as defined in [41], with an additional feed-forward layer that computes a 2- or 3-dimensional embedding vector.

Definition 2 *The* closure Hamming distance *(CHD) between two sets of attributes is the* Hamming distance *between the binary representation of respective closures of the two sets of attributes. In other words, it is the number of modifications necessary to go from one closure to the other. Two attribute sets are called* equivalent *if they have the same closure, so if their CHD is 0.*

Closure2vec is trained on randomly sampled pairs of sets of attributes, with the sets in a pair differing by exactly one attribute.

3.1.3 Evaluation

In this subsection, we describe the evaluation processes used to evaluate attribute2vec and object2vec. For the evaluation, the authors use real-world datasets and tasks using the embeddings.

Attribute Clustering for Attribute2vec

The performance of *attribute2vec* is evaluated through an attribute clustering task. The *wiki44k* dataset[3] is used for this experiment. Wiki44k is a knowledge graph (KG) where *entities* (the nodes) are put in relation using *statements* (directed labeled edges) labeled with *properties* (edge labels). The corresponding FC uses the entities as objects and property names as attributes. An object o_i and an attribute a_j are related by the incidence relation if entity o_i appears in a statement labeled with property a_j. As the authors of [9] mention, wiki44k is sparse for an FC, with a density of 0.04.

[3] Available at `http://people.mpi-inf.mpg.de/~gadelrab/RuLES/`.

Emb. size	Model	$k = 2$	$k = 5$	$k = 10$
	Naive	0.0158 ± 0.0000	0.0055 ± 0.0042	0.0035 ± 0.0002
2	Random	0.0534 ± 0.0412	0.0084 ± 0.0088	0.0010 ± 0.0007
	a2v-SG	0.1608 ± 0.0031	0.0703 ± 0.0122	0.0069 ± 0.0004
3	Random	0.0219 ± 0.0107	0.0036 ± 0.0027	0.0007 ± 0.0004
	a2v-SG	0.3217 ± 0.0005	0.1038 ± 0.0218	0.0080 ± 0.0001

TABLE 3.2: Rate of intra-cluster implication with the attribute2vec skip-gram variant (*a2v-SG*) and the naive and random baselines, for embeddings of size 2 and 3, for $k = 2$, 5 and 10 clusters, for 20 repetitions of the evaluation experiment (mean $\pm$ std.). Table 3 from [9].

The canonical base of wiki44k is used for the attribute clustering task of [9]. A canonical base can be seen as a minimal set of implications (of the form $X \rightarrow Y$, with $X, Y \in A$) which is sufficient to describe all the valid implications of a lattice. An implication $X \rightarrow Y$ is valid if Y appears every time X does, *i.e.*, if we have X, we always have Y. Hanika, one of the authors of [9], explains in [21] the interest of studying the canonical base of the FC of wiki44k. For further detail on the canonical base see, *e.g.*, [2, 21].

The attribute clustering task consists in clustering attribute embeddings using a simple clustering algorithm (k-means). The performance is measured by taking the rate of *intra-cluster* implications, *i.e.*, implications from the canonical base which are "completely contained in one cluster" [9]. An implication $X \rightarrow Y$ is called *intra-cluster* if there is a cluster K such that $X \cup Y \in K$. Two baselines are used: *(i)* a random clustering, obtained by generating random clusters with similar sizes as ones obtained by attribute2vec, and *(ii)* a naive clustering, obtained by clustering a naive representation of the attributes. For an attribute a, this naive representation is the binary encoding of a', the set of objects having the attribute a.

The performance reported in [9] is shown in Table 3.2. The performance of attribute2vec's CBow variant is lower than the random baseline, so the corresponding results are not reported by [9]. Attribute2vec skip-gram outperforms the two baselines, which could be expected given the training objective of attribute2vec. Indeed, attribute2vec is trained to bring together attributes often appearing in the same intents, and implications describe sets of attributes appearing together in the dataset.

Link Prediction for Object2vec

A link prediction task is used to evaluate *object2vec*. The *object2vec* model is applied on ICFCA⋆[4], a dataset of co-authorship in the FCA community. Descriptive statistics on the dataset are reported in Table 3.1. It is interesting to note that the dataset used is very sparse for an FC, with a density of 0.005, almost 10 times lower than the already sparse of wiki44k. In the co-authorship graph, two authors have an edge between them if they are co-authors of an article. The FC of this dataset uses authors as objects, papers as attributes, and the incidence relation is the authorship of a paper by an author. The dataset is split in two: all the papers published strictly before 2016 are used to train the embedding model (training set), and all the papers from January 2016 to August 2019 are used to evaluate the performance of the model (evaluation

[4] ICFCA⋆ is available at the *conexp-clj* repository: `https://github.com/tomhanika/conexp-clj`.

Emb. size	Model	Recall	Precision	F1
2	node2vec	0.56 ± 0.14	0.60 ± 0.07	0.57 ± 0.09
	o2v-SG	0.66 ± 0.08	0.65 ± 0.03	0.65 ± 0.05
	o2v-CBoW	0.68 ± 0.09	0.64 ± 0.04	0.66 ± 0.06
3	node2vec	0.60 ± 0.15	0.56 ± 0.08	0.58 ± 0.10
	o2v-SG	0.70 ± 0.08	0.62 ± 0.04	0.66 ± 0.06
	o2v-CBoW	0.73 ± 0.07	0.65 ± 0.06	0.69 ± 0.06

TABLE 3.3: Performance of the two *object2vec* variants and the *node2vec* baseline, for embeddings of size 2 and 3, for 30 repetitions of the evaluation experiment (mean $\pm$ std.). *o2v-SG* stands for the skip-gram and *o2v-CBoW* for the CBoW variant. Table 2 from [9].

set). Only the articles written by authors appearing in the training set are kept for the evaluation set.

To evaluate the performance, the author (or object) embeddings are generated on the whole dataset. Co-authorship embeddings are then computed by taking the element-wise product of two randomly sampled author embeddings. Half of the generated embeddings correspond to actual co-authorship in the dataset and are called positive samples. The other half is made of negative samples, in other words, pairs of authors who are never co-authors in the dataset. A basic classifier (logistic regression) is trained on the training set to predict if the co-authorship relation described by the corresponding embedding is true or not. The performance of this classifier is then evaluated on the test set. As a baseline, the authors of [9] use the node embeddings computed on the co-authorship graph by *node2vec* [18], a graph embedding method similar to DeepWalk (see Subsection 2.1.1).

The performance reported in [9] is shown in Table 3.3. For this task, object2vec outperforms node2vec by at least 5% on each score, for both sizes of embedding. The CBoW variant slightly outperforms the skip-gram one.

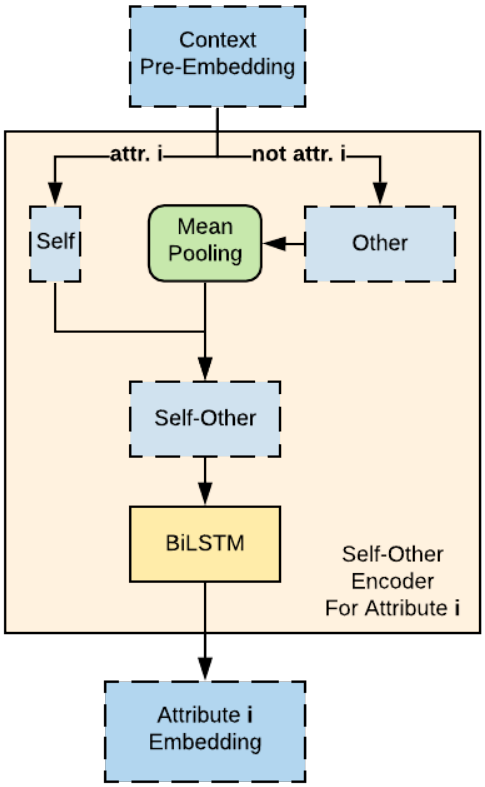
3.2 Bag of Attributes Model

In this section, we define the proposed FC embedding architecture and the objectives used during training.

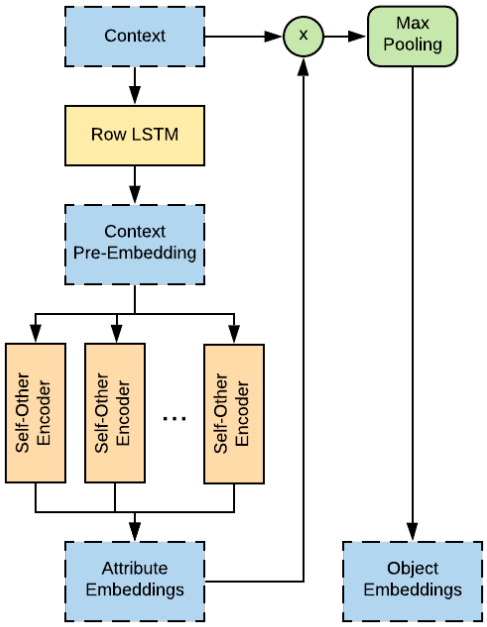
3.2.1 Architecture

Bag of Attributes (BoA) takes a formal context as input and produces embeddings for its attributes. Then, object embeddings are computed using the embeddings of the attributes and the formal context. BoA has four main components: a pre-embedding generator, an attribute encoder called *self-other encoder* to compute the attribute embedding, an object encoder, and a decoder. The order of the attributes in the FC does not matter for FCA, because intents are unordered sets of attributes. Therefore, in BoA, the order of the attributes is ignored by design. BoA considers the attributes as an unordered set to produce the object and attribute embeddings. The name is an homage to *Bag of Words* (BoW) [38], which consider sentences as unordered bags of words.

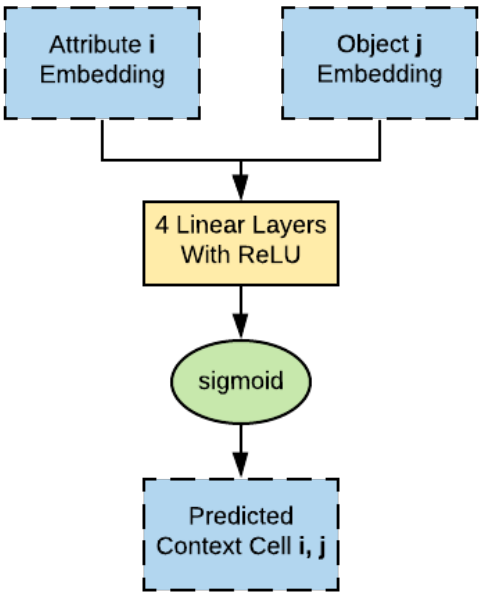
To capture the absence of order between the attributes, they are processed in a similar manner. Each attribute is compared to all the other attributes, for each



(A) Self-other attribute encoder for an attribute.



(B) BoA encoder architecture.



(C) BoA decoder architecture.

FIGURE 3.2: Schematic representation of the BoA architecture. Blue blocks correspond to tensors, the orange to neural components and green blocks to non-neural computations. Arrows joining blocks represent concatenation of tensors.

object of the FC. In practice, the column of an attribute (*self*) is compared to an unordered composition (average-pooling) of all the other attributes (*other*). *Self* and *other* are then concatenated and processed by a BLSTM, with the object dimension as the sequence dimension. The last hidden state of the BLSTM is processed with a feed-forward layer into an embedding that represents the attribute. The structure of the attribute encoder is presented in Figure 3.2a. A μ and a σ vector are produced for each attribute because BoA is trained as a VAE, and the actual embeddings are samples from the normal distributions defined by μ and σ. Finally, the object embeddings are computed by applying max-pooling on the embeddings of the attributes present in the object's intent. The structure of the encoder is schematized in Figure 3.2b.

This encoder architecture successfully ignores the order of the attributes. However, we are not able to differentiate between the attributes anymore either, which is a problem to model FCA. Indeed, we still need to differentiate the attributes to know which ones belong in which intent, even if the order of the attribute does not matter. Using unordered composition directly on the FC will prevent the model from differentiating, *e.g.*, when an attribute a_1 is present and a_2 is not from when a_2 is present and not a_1: the average of the list $(0, 1)$ is 0.5, the same as the average of the list $(1, 0)$. The same kind of problem arises with standard embedding methods using a learned vector to represent each possible value. In our previous example, if we replace in C every 1 with the same embedding emb_1 and every 0 by emb_0, the model is still unable to determine which one between a_1 and a_2 is present. To avoid this issue, we apply an LSTM on each row of C before the *self-other encoder*. This LSTM can produce different embeddings for each attribute despite the same input, so it allows the model to identify the attributes when the unordered composition is applied. Indeed, RNNs produce different outputs from the same input if their hidden state is different.

The decoder is an MLP predicting if an object has an attribute or not (1 or 0, respectively). Its input is the concatenation of the object and the attribute embeddings. A sigmoid function applied to the output ensures it is in $[0, 1]$. It is schematized in Figure 3.2c.

3.2.2 Training Objective

We train BoA using KL divergence on the attribute embeddings exclusively because the sampling happens before the computation of the object embeddings. We use the BCE loss for reconstruction because the model predicts between two classes (1 and 0). To improve the quality of the embeddings, we use metric learning the *co-intent similarity* (defined in the next paragraph) and the number of concepts [13], with *mean square error* (MSE) as the loss function. We use MLPs to predict the co-intent similarity and number of concepts. The training setup is schematized in Figure 3.3.

On the one hand, we need both "equal" and "different" attributes to use metric learning losses on attribute embeddings. Nonetheless, even if we consider equivalent attributes (*i.e.* with the same extent) as "equal", they are usually rare within a given context. We define co-intent similarity to compare attributes and avoid this issue. Given two attributes a_1 and a_2 we define their *pairwise co-intent similarity* as:

$$\text{co-intent}(a_1, a_2) = \begin{cases} 1 & \text{if } |\{i \in I | a_1 \in i\}| + |\{i \in I | a_2 \in i\}| = 0 \\ \dfrac{2 \times |\{i \in I | a_1 \in i, a_2 \in i\}|}{|\{i \in I | a_1 \in i\}| + |\{i \in I | a_2 \in i\}|} & \text{otherwise.} \end{cases} \tag{3.1}$$

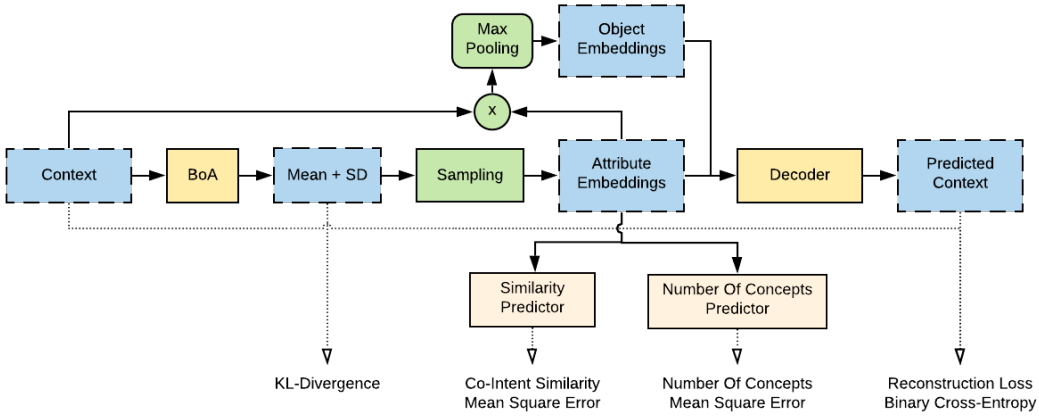


FIGURE 3.3: Schematic representation of the BoA architecture training process.

In other words, it is the ratio of intents containing both attributes over the intents containing a_1 or a_2[5]. In cases where no intent contains the attributes (both attributes are empty or padding columns), the similarity is set to 1. Co-intent similarity ranges from 0, for attributes never appearing in the same intents, to 1, for attributes always appearing together or for identical attributes. To predict the co-intent similarity between two attributes a_1 and a_2, the input of the MLP predictor is the concatenated embeddings of a_1 and a_2, and a sigmoid output function is added to ensure the predicted similarity is in $[0, 1]$.

On the other hand, predicting the number of concepts from the context without actually computing the intents, helps when generating the set of concepts using neural models. Indeed, knowing how many elements to generate beforehand facilitates the generation process. Note that counting the number of concepts of a context is #P-complete (and so, is a "hard" task to achieve) [32]. We apply a max-pooling over the attribute embeddings before predicting the number of concepts with the MLP predictor, which corresponds to a *deep averaging network* (DAN) [23].

3.3 Training

In this section, we explain how we train BoA. We describe the training process in Subsection 3.3.1 and the randomly generated dataset in Subsection 3.3.2. In Subsection 3.3.3, we define a data augmentation pipeline used to improve the generalization capacity of BoA and compensate for some of the biases of the random generation. Finally, we explain in Subsection 3.3.4 a problem we encountered with the variational aspect of BoA.

3.3.1 Training Process

We train BoA in two phases of 5000 epochs each. In the first phase, we apply the reconstruction loss and the KL divergence only. Then, we gradually introduce the prediction of the co-intent similarity and the number of concepts. When using metric learning with multiple distances, a common approach is to split the embedding space and to learn one distance per sub-part of the embedding space [31]. We apply

[5]Observe that this is essentially the Jaccard index on the set of intents.

	Dataset	# Object	# Attribute	# Concept	Density of C
Mean $\pm$ std.	Train	12.83 ± 6.11	12.98 ± 6.03	77.93 ± 78.39	0.329 ± 0.057
	Test	12.83 ± 6.13	12.97 ± 6.04	78.12 ± 77.27	0.332 ± 0.057
Range	Train	1 to 20	2 to 20	1 to 401	0 to 0.56
	Test	2 to 20	3 to 20	2 to 401	0 to 0.49

TABLE 3.4: Descriptive statistics on the dataset of randomly generated contexts.

the same principle and use 50% of the embedding space to predict the co-intent similarity and 25% for the number of concepts. The exact embedding dimension of BoA is 128, with a pre-embedding size of 64. The LSTM and the BLSTM have two layers each. The decoder MLP has four layers, and the MLPs used for distance prediction both have two layers. We use a ReLU activation function between all the layers of the model.

3.3.2 Training Dataset

The dataset used for training BoA is composed of 6000 randomly generated formal contexts and the corresponding lattices split into training and validation. We generate a training set of 5000 contexts and a test set of 1000 samples using the generation principle described in Subsection 2.2.1. For the training phase, a development set of 10% of the training set is randomly sampled from the training set. For each set, we generate different sizes of contexts, 20% of each: 5×5, 10×10, 10×20, 20×10, and 20×20 contexts ($|O| \times |A|$). We report statistics of the generated datasets in Table 3.4.

3.3.3 Data Augmentation

We rely on plain random generation for the formal contexts, and not on more involved generation processes as discussed in [12, 10], so the random data is biased. We introduce a simple way to compensate for some of those biases while improving the generalization capability of the model using *data augmentation*. In deep learning and machine learning in general, data augmentation is the process of artificially augmenting the amount of training data, typically by modifying existing training samples. We implement the following data augmentation pipeline: *(i)* duplicating of objects and attributes, *(ii)* inverting (dropping) the value of entries, and *(iii)* shuffling objects and attributes. With this process, we simulate identical (duplication) and nearly identical (duplication + drop) objects or attributes that appear in real-world datasets.

Objects and attributes have a probability p of being duplicated. If duplicated, they have the same probability p of being duplicated again. From this definition, the number of copies of an object (or attribute) follows a geometric law with a probability of success p. Consequently, the exact number of objects and attributes seen during training does not match the ones reported in Table 3.4. Nonetheless, the duplication follows a geometric law so we can estimate the number amount of objects and attributes seen as $number/(1-p)$. Inverting some randomly selected values in the formal context is our adaptation of dropout, a common technique in deep learning. The shuffling after duplication avoids the model's reliance on the order of the objects and the attributes. We set the duplication probability to $p = 0.1$ and the drop

probability to 0.01. In this setting, the estimated average object and attribute numbers are respectively 14.25 and 14.42, for both the training and development sets.

When co-intent similarity is used, duplication and shuffling are reproduced on the intents. However, drops in a formal context alter the corresponding lattice, so they are not applied when using the intents to compute co-intent similarity. This precaution avoids making the model insensitive to small variations in the input.

3.3.4 Issues With KL Divergence

When adding the KL divergence to the prototype of BoA (initially a simple auto-encoder) the performance of the model was greatly impaired. The analysis of the predictions revealed the model was going for "low hanging fruits" and ignored the embeddings themselves, as described in [4]. To solve this issue we apply *annealing* [4] and multiply the KL divergence by a lambda that we set to 10^{-3}. This reduces the impact of the KL divergence on the training and allows the model to learn some features before the KL divergence comes into effect. However, it reduces the benefits we get from using a VAE.

3.4 Experiments

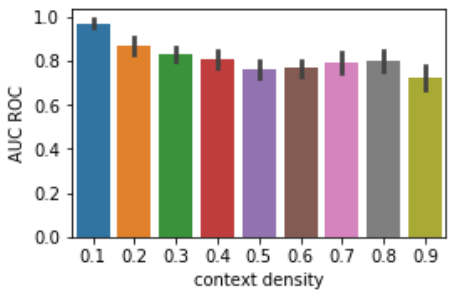
In Subsection 3.4.1 and Subsection 3.4.2, we explore the limits of BoA *w.r.t.* input data. In Subsection 3.4.3, we reproduce the experiments from [9] on real-world data. All the experiments described in Subsection 3.4.1 and Subsection 3.4.2 are performed on randomly generated data to control the of the evaluation process.

3.4.1 Reconstruction Performance

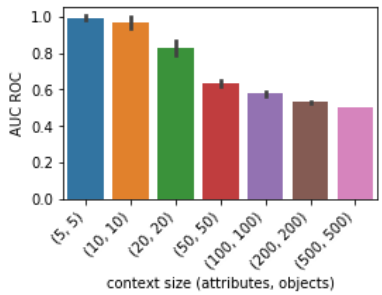
To assess the reconstruction performance of the BoA auto-encoder, we use the *area under the receiver operating characteristic curve* (AUC ROC). It allows us to determine whether the BoA has a good predictive capacity and, similarly to the F1 measure, AUC ROC gives a general account of performance. To determine if the results are significantly different, we use Student t-test on means. The results are presented in Figure 3.4.

We first evaluate the impact of the density on the reconstruction by comparing the performance on random contexts with densities from 0.1 to 0.9. We use 100 samples per density with a fixed size of 20 objects and attributes. Student's t-test shows significant differences between the performance with the various densities: all the p-values are under 0.01 except between 0.4 and 0.8 (0.24), 0.5 and 0.6 (0.39), and 0.7 and 0.8 (0.19). However, the model performance stays overall stable across the densities, while slightly better with smaller densities. We suspect this tendency is due to the composition process of the object embeddings: the higher the density, the more attributes are present for an object, so more attribute embeddings are involved in the composition of the object embeddings, making it more complex to decode.

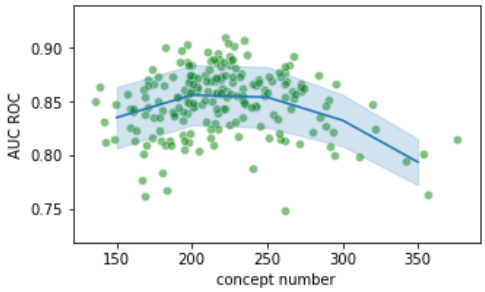
We also examine the effect of the size on the AUC ROC. Square random contexts ($|O| = |A|$) of sizes in $\{5, 10, 20, 50, 100, 200, 500\}$ and a fixed density of 0.3 are used for this experiment, with 20 samples per size. The model performs very well for seen data sizes with a slight drop to 0.83 for 20 objects and attributes. As expected, the performance drops when manipulating larger contexts. For 50 objects and attributes (2.5 times the maximum seen size), the AUC ROC is above 0.63, but from 100 objects and attributes onward, it drops under 0.6. Finally, with 500 objects and attributes (25 times the largest seen data size and 4 times the embedding size), the reconstruction



(A) Impact of the density, from 0.1 to 0.9, 100 samples per density.

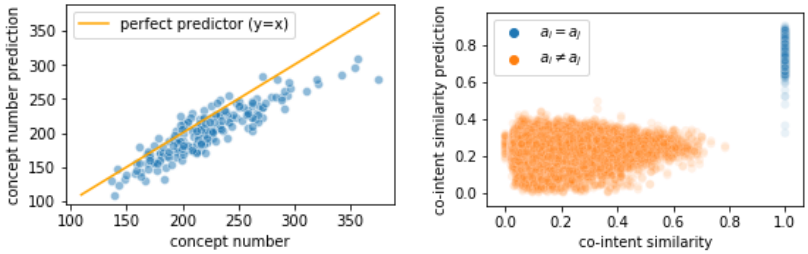


(B) Impact of the size, from 5 to 500 objects and attributes, 20 samples per size.



(C) Impact of the concept number, for 200 sample with 20 objects and attributes. The blue line is the general tendency when rounding the concept number to 50.

FIGURE 3.4: Reconstruction performance on random contexts. The error bars and the shaded area correspond to the standard deviation.



(A) Concept number. (B) Co-intent similarity.

FIGURE 3.5: Predicted metrics against the actual values, for the 200 samples with 20 objects and attributes from the test set.

AUC ROC falls to 0.50 on average. This is the limit of reconstruction performance with the current training process.

Finally, we examine the impact of the number of concepts on the performance of the model. We use contexts of fixed size (20 objects and attributes) from the test set, totaling 200 contexts. We consider the concept number as an indicator of the variety of attributes and objects in the context. Indeed, if the concept number is high for a given size of context, we can expect the context to be close to the clarified context (context with no equivalent objects or attributes). This implies a lower amount of duplicate objects and attributes. Besides, we can expect the model to have a harder time encoding and decoding irregular contexts than repetitive ones. Consequently, the drop of the AUC ROC for higher concept numbers is not surprising. However, we also observe a lower performance around 150 concepts. This second decrease requires further investigation.

3.4.2 Metric Learning Performance

We evaluate the performance of BoA for the co-intent similarity and number of concepts' prediction, by computing the attribute embeddings and applying the predictors trained together with the BoA model. We use the 200 contexts of 20 objects and attributes from the test set. The prediction results are reported in Figure 3.5.

The Pearson correlation coefficient is 0.9 between the actual concept number and the prediction, indicating a strong correlation. We can notice in Figure 3.5a the tendency of the model to under-evaluate the concept number. We mention in Subsection 1.3.3 a naive upper bound of the number of concept $2^{min(|O|,|A|)}$. In [32, p. 2], Kuznetsov mentions a more involved upper bound defined by Schütt:

$$3/2 \times 2^{\sqrt{|\mathbf{I}|+1}} - 1$$

The density d of the FC is such that $d = \frac{|\mathbf{I}|}{|O| \times |A|}$, so we can write $|\mathbf{I}| = d \times |O| \times |A|$. The predictions performed by our model are much smaller than the theoretical upper bounds of the number of concepts. The prediction performance of the model is very helpful for our final goal, the generation of concepts, as mentioned in Subsection 3.2.2.

When predicting the co-intent similarity, BoA manages to differentiate a_i and a_j when $a_i = a_j$. However, the predictions in the other cases are not clear: for similarities between 0 and 0.8, they seem randomly picked between 0 and 0.4. The analysis of the training process reveals a small difference between the MSE of the first and the last training epochs: from 0.17 to 0.05. Using MSE on values between 0 and 1

Dataset	# Objects	# Attributes	Density	# Concepts
SPECT heart	68	23	0.23	911

TABLE 3.5: Descriptive statistics on SPECT heart.

seems to cause the problem: an MSE of 0.05 corresponds to an actual distance of around 0.22, so 10% of the interval of definition of co-intent similarity. We envision several solutions, like changing the loss to *mean absolute error* (MAE) or normalizing the similarity.

3.4.3 Experiments on Real-World Datasets

To evaluate the performance of BoA on real-world datasets, we reproduce the link prediction and attribute clustering of [9]. We use the same ICFCA dataset as [9] for link prediction. However, for attribute clustering, we use SPECT heart[6] as it is smaller than wiki44k, with dimensions closer to the training data: 68 objects and 23 attributes. Additionally, SPECT heart is much denser than wiki44k with a density of 0.23. Descriptive statistics on SPECT heart are reported in Table 3.5.

We train the CBoW and SG variants of FCA2VEC models using the same settings as in [9], with 20 random iterations of each model and an embedding size of 3. To obtain comparable results, we reduce the embeddings produced by BoA to 3 dimensions by applying two standard dimensionality reduction techniques: *principal component analysis* (PCA) and *t-distributed stochastic neighbor embedding* (TSNE). We use Student's t-test on means to determine if the results are significant.

We report the link prediction performance in Table 3.6. The three BoA variants show a significantly different performance from o2v SG, with all the p-values lower than 0.005. We found that the classifier based on BoA, the one with the best F1 score, systematically answers positive. Additionally, we fail to reproduce the performance of [9] (F1 score of 0.69 for o2v CBoW, 0.66 for o2v SG) Finally, the ICFCA context is very sparse: it has a density of 0.003 on the train and 0.005 on the test set. Due to this, the task may not be representative of the performance of the object embeddings on most datasets. These results hint that the task needs to be adapted to get proper insights into the object embedding performance. The attribute clustering performance is reported in Table 3.7. In this experiment, we find that the CBoW variant performs significantly better than the SG (all t-test p-values under 0.0005). This is the opposite of the result found by [9] for attribute clustering. However, this result may be due to using a different dataset. Interestingly, the BoA PCA variant performs equally to the full BoA. The performance of BoA (and BoA PCA) is significantly better than a2v CBoW for 2 and 5 clusters (p-values under 10^{-14}). For 10 clusters however, a2v CBoW performs significantly better (p-value under 0.001). The model improves the performance of a2v CBoW by 4% for 2 clusters and 8% for 5 clusters.

[6] https://archive.ics.uci.edu/ml/datasets/SPECT+Heart

TABLE 3.6: Performance on the link prediction task (mean $\pm$ std.).

Model	Precision	Recall	F1
o2v-CBoW	0.63 ± 0.05	0.46 ± 0.05	0.53 ± 0.05
o2v-SG	0.70 ± 0.04	0.49 ± 0.03	0.57 ± 0.02
BoA PCA 3d	0.65	0.42	0.51
BoA TSNE 3d	0.58	0.67	0.62
BoA	0.50	1.00	0.67

TABLE 3.7: Performance on the attribute clustering task with 2, 5 and 10 clusters (mean $\pm$ std.).

Model	k = 2	k = 5	k = 10
a2v-SG	0.35 ± 0.13	0.11 ± 0.03	0.042 ± 0.010
a2v-CBoW	0.66 ± 0.00	0.14 ± 0.02	0.063 ± 0.013
BoA TSNE 3d	0.30	0.29	0.044
BoA PCA 3d	0.70	0.22	0.051
BoA	0.70	0.22	0.051

Chapter 4

Second Approach: Intents Generation

BoA allows us to generate embeddings of attributes and objects, and to reconstruct the FC from those embeddings. As we have a representation of objects, which can be seen as sets of attributes, we can use this same representation for intents, which are also sets of attributes. The reconstruction performance of BoA should allow us to generate intent embeddings and to decode them.

Based on those results and the results of our preliminary experiments described in Section 4.1, we designed a 4 phase approach to build an intent generation model:

1. train a BoA auto-encoder;
2. train a simple model to predict an upper bound of the number of concepts from the attribute embeddings, based on the predictor learned with BoA;
3. design a model to predict the intents and the cover and order relations, using LSTM and the attention mechanisms tested in the preliminary experiments;
4. as we expect the previous model to produce imperfect results given the performance of the preliminary models, an additional can be designed to "refine" the results.

Once the whole system is finished, a final fine-tuning of the whole system should be performed. This approach allows us to split the task into simpler problems. Each phase produces a layer of the final model, and allows to train and evaluate them independently. Consequently, it is easier to determine the weak-points of the architecture and improve the architecture accordingly. Additionally, the first phase is already solved and the second one should be trivial. This second approach is closer to the naive concept generation algorithm than the graph-based approach, because we first generate the set of intents and then compute the order between them.

We successfully implemented this approach, with the BoA model, a DAN upper bound predictor, and an attention-based LSTM lattice generator, respectively described in Chapter 3, Section 4.2, and Section 4.3. However, due to time constraints, we focused on improving the results of phase 3 instead of designing and training the "refiner" of phase 4. Such modification of the approach do not prevent us form providing results for the rest of the architecture, which is one of the advantages of our modular approach. We describe our training process and the performance of our current model in Section 4.4.

4.1 Pilot Experiments

In this section, we explain our preliminary experiments which led to the design of our intent generation model. The goal of the pilot experiments presented in this section is to determine if a specific architecture could benefit our task, so we consider simple variants of major architecture families which could fit our goal. Also, the architectures are not trained to their maximum, and even if the results are not conclusive, it does not mean that an architecture in itself has no potential. However it hints a lower potential improvement of the performance for our task. We consider 3 families of architectures for the task, each requiring some preliminary assumption on the size of the data:

- the VAE architecture requires to have a fixed input and output size, so all the dimensions ($|O|$, $|A|$, and $|I|$) must be defined when creating a model; it is presented in Subsection 4.1.1;
- the CNN architecture only requires to have a fixed $|I|$; it is presented in Subsection 4.1.2;
- the LSTM architectures only require to have a fixed $|A|$; it is presented in Subsection 4.1.3.

We explain those limitations in the following subsections. In each case, a model will be able to handle smaller inputs and outputs, but nothing larger than the values given creating the model.

4.1.1 Variational Auto-Encoder Architecture

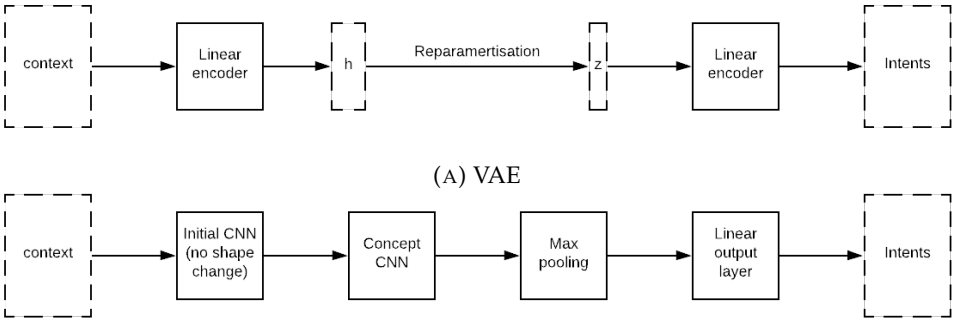
The VAE architecture is structurally the simplest of all the tested architecture. It takes as an input a flattened FC at once and the output is reshaped into an intent matrix. The encoder and the decoder are two MLPs. A block diagram of the architecture is shown in Figure 4.1a.

Using MLPs directly forces the input and output to always have the same size. In this setup, we need to define $|O|$, $|A|$, and $|I|$ when we create the model. Typically, those should be the maximal values in dataset of $|O|$, $|A|$, and $|I|$. Using padding will allow us to process samples with smaller dimensions, but the model will be unable to process larger samples without loosing information.

This architecture is very simple and performs well, as shown in Figure 4.2a. Indeed, the VAE seems to produce the expected shape and spread of the attributes across the contexts: few attributes for the intents close to $\top$, and more and more attributes until we reach $\perp$. However, the fixed input and output size is a major drawback of this approach.

4.1.2 Convolutional Neural Network Architecture

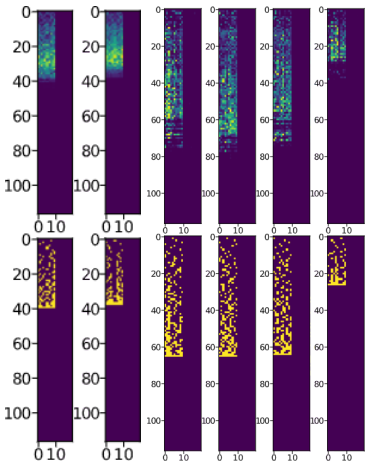
The CNN architecture offers flexibility in the size of the input and output. However, the size of the output of a CNN is proportional to the size of its input (due to the convolution principle). In our case, it would mean that the number of predicted concepts would be directly proportional to the number of objects, which seems quite strange especially since that the theoretical upper bounds (see Subsection 3.2.2) make the number of concept an exponential of the size of the FC. With a standard CNN, the relation between the number of input objects and the number



(A) VAE

(B) CNN

FIGURE 4.1: Block diagrams of the VAE and CNN architectures.



(A) VAE (B) CNN

FIGURE 4.2: Examples of results of the VAE and CNN architectures. In the images, a blue pixel corresponds to a 0 and a yellow one to 1. The first row correspond to the prediction by the model and the second row is the actual intent matrix. Each column correspond to a different sample in the same batch.

of output intents would have to be defined when designing the architecture, which seems unreasonable due to the explosive nature of the number of potential concepts.

To avoid the issue described above, we use one output *channels* of the CNN for each concept, leading to the architecture schematized in Figure 4.1b. Consequently, we do not have a relation between the number of objects and the number of concepts nor any constraint on the number of objects, even if we still require the maximum $|I|$. Interestingly, there is a full "view" on the data dedicated to each concept because each channel serves as a parallel and independent "view" on the FC. The tested CNN requires a lot of parameters to compensate for removing the constraint on the object number. This large amount of parameters and its internal structure gives it a greater learning capability than the VAE.

As shown in Figure 4.2b, the shape of the output is close to the expected one and the spread of the attributes matches the expected output, similarly to the results of the VAE. The results of the CNN look a little more random than the ones of the VAE, even if the "texture" (the sharpness of the result) of the predictions by the CNN is closer tho the expected output than the one produced by the VAE. This approach seem to perform comparably to VAE, but with only a constraint on $|I|$.

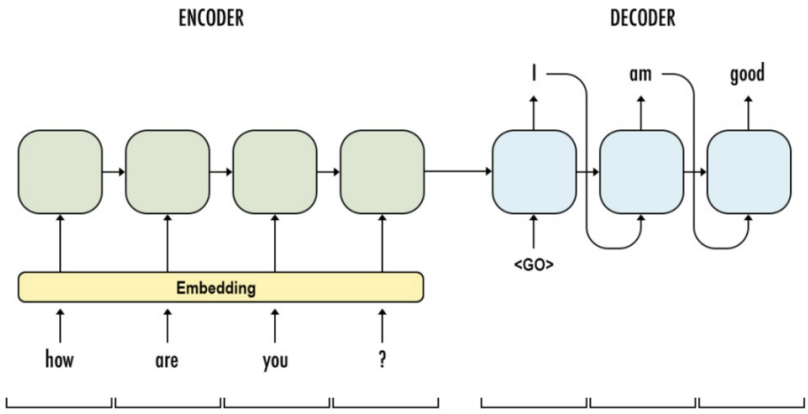


FIGURE 4.3: Example of a sequence to sequence RNN, figure from [24].

4.1.3 Recurrent Neural Network Architecture and Attention Mechanisms

We use LSTM as a sequence to sequence model (*seq2seq*), a common way to use the RNN to produce an output which does not have the same size as the input, *e.g.*, Figure 4.3. Indeed, in our case, the number of concepts is different from the number of objects. We use the attribute dimension as the input and output size, and the object and concept dimension as the sequence dimension. Due to this, the model is constrained the maximum $|A|$. We experiments with a few variants of attention mechanisms:

- *soft-attention*, which compute a summary of all the inputs with respect to the current output, and use this summary to improve the current output;
- *self-attention*, which compute a summary of the previous outputs with respect to the current output, and use this summary to improve the current output;
- a variant for our case of self-attention: *link-attention*, which compute the similarity of the previous outputs with respect to the current output, and this similarity (between 0 and 1) serves as a link prediction; more precisely, the similarity between previous output i and current output j represent the probability to have a link between concept i and concept j.

We use a simple version of attention, with $\cdot$ the dot product, q the *query* (the current LSTM output in our case), s the sequence. We first compute the attention weight for a value $v \in s$ as $w_{q,v} = q \cdot v$, and apply a softmax on all the weights. Because after the softmax the sum of the weights is 1, the attention context $a_{q,s} = \sum_{v \in s} w_{q,v}$ also corresponds to the weighted average of the input sequence. The global structure of the recurrent architecture is presented in Figure 4.4, with the three attention mechanisms in different colors. Because we want to check whether using a specific attention mechanism improves the output, we test 5 variants of the architecture listed bellow, and compare the results along the arrows on Figure 4.5.

The *basic* model uses no attention, and takes the FC as a sequence of objects in input, and produces a sequence of intents in output. Also, a second output is produced: the stop vector. It serves to determine when to stop generating intents during real-case usage. In this specific implementation, the stop vector contains 1 if the corresponding row is a context intent, 0 if it is just padding. The *soft-attention* model extends the basic model with the soft-attention mechanism, which should

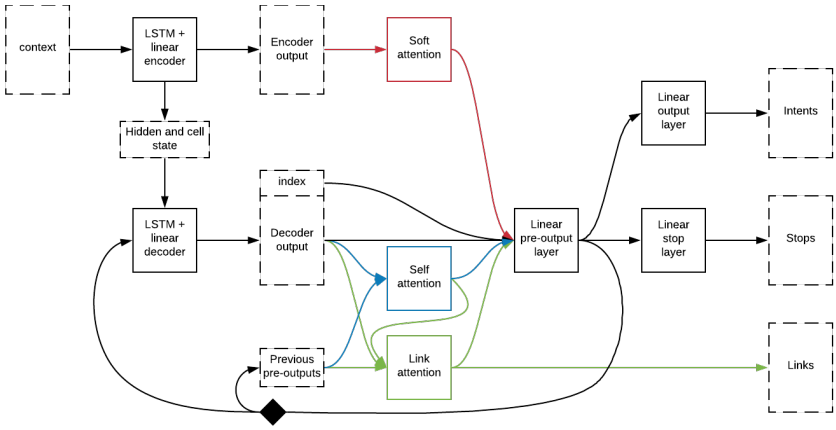


FIGURE 4.4: Block diagram of the recurrent architectures.

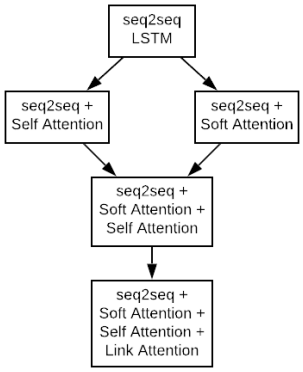


FIGURE 4.5: Hierarchy of the tested recurrent architecture variants

improve the relevance of the output *w.r.t.* the input because the model has a better access to the input information. The *self-attention* model extends the basic model with self-attention, which should improve the internal coherence of the output, as the model has a better access to the previously generated intents. We should obtain, *e.g.*, a gradient from the empty set to the full set of attributes and a better prediction of the stop vector. The two attention mechanisms are used in the *bi-attention* model, so we can expect this model to bring the improvements of the self- and soft- attention together. Finally, the *tri-attention* model extends the bi-attention model with a link-attention as predicted above. It generates an additional output: the links between the concepts, in other words the adjacency between the concepts. In our case we predict $\leq$, as we expect it to be easier to learn then $\prec$. Indeed, for $\leq$, a generic subset relation is enough, while for $\prec$ an additional step is required.

The basic model performs quite bad (see Figure 4.6a), which is to be expected given the complexity of the data and the relative simplicity of the model. All the produced intents look similar, and are most likely a sort of average of all the intents in the training set, and the stop vector is not at all predicted. Not much impact of the input can be seen, which is to be expected: it is too "far" from the output for a simple LSTM to handle. The soft-attention improves the model in a noticeable, and expected, manner: the output now has the same number of attributes as the input (see Figure 4.6b). The self-attention also fulfills our expectations: the output has a

nicer gradient (compared to the basic model) and the stop vector begins to be predicted correctly. An unexpected improvement is that the output now also has the same number of attributes as the input. In Figure 4.6c display those improvement, and one can also notice an interesting "bar" around the end of the prediction, which seem to correspond to $\perp$, hinting a large potential improvement of the stop prediction in case of further training. The bi-attention produces the expected results too: the output has the correct number of attributes, a the stop vector is closer to the expected one, and the length of the output matches quite well the expected one (as we can see on Figure 4.6d, with sample 2 being predicted slightly longer than sample 3, which matches the expected output).

However, the tri-attention produces both expected and unexpected effects. As expected, the links in Figure 4.6e look close to the expected output which hints that link attention is beneficial and can be used to predict the relations between concepts. A first noteworthy point is that the performance of the model varies from good (see Figure 4.6e) to bad, depending on the random initialization. We suppose that training all 3 outputs (intents, stops and links) at the same time from scratch, is too much for the model, and that training one aspect alone before introducing the others could avoid this type of issue. The negative impact of those training issues on the performance cannot be neglected. The second interesting point is that the stops in Figure 4.6e are close to the expected output, and the intents and links are clearly impacted by the stops. Finally, we don't have too much of a degradation of the performance from the bi-attention model despite the training difficulties, even if the number of attributes is less well-predicted.

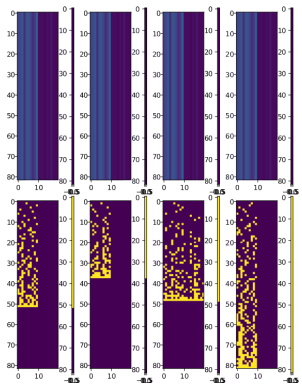
4.1.4 Conclusions on the Tested Architectures

In conclusion, the VAE has heavy constraints on the dimension of the data which are not compatible with our goal, even if the architecture is simple, the training fast and the results among the best of all the studied architectures. The CNN is not very accurate. However, it manages to capture the shape of the expected data, and thus could be used to reffine the more "blurry" output of the recurrent models.

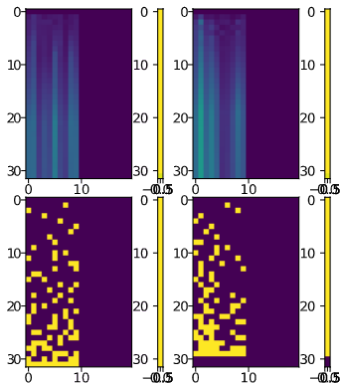
The LSTM architectures are flexible enough for practical use, and the various attention mechanisms improve the performance by a notable margin in addition to providing with tools for the interpretability of the results of the model. Also, the link attention process seem to have potential, but the way it interacts with the rest of the output and the training process both have to be revised. Note that the LSTM architectures are applied directly on the FC. Applying this approach on the embeddings generated by BoA should allow us to improve the performance. More importantly, by replacing the attribute dimension by the embedding dimension, the resulting architecture can be applied in virtually any size of FC.

4.2 Concept Number Upper Bound Prediction

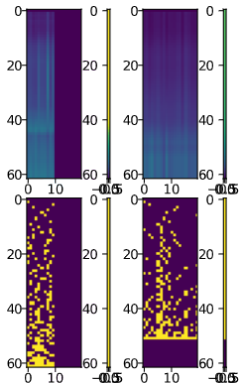
In sequence generation, there are two approaches on how to generate the correct number of elements. The first option is to interrupt the generation based on a value generated at each step by the model, *e.g.*, a stop value equal to 1 when generating the last element. The alternative is to determine the number of elements beforehand and generate exactly this number of elements. To generate the correct number of intents, we decided to use the second option, given the performance of the concept



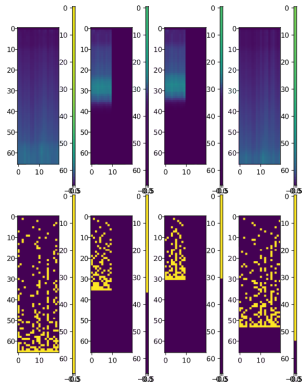
(A) Extract of a result with the the recurrent model.



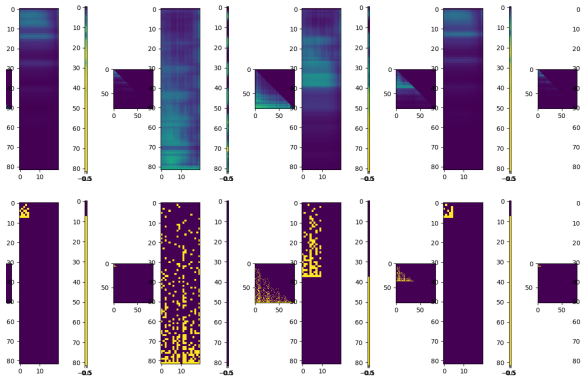
(B) Extract of a result with the soft-attention model.



(C) Extract of a result with the self-attention model.



(D) Extract of a result with the bi-attention model.



(E) Extract of a result with the 2nd iteration of the tri-attention model.

FIGURE 4.6: Extract of results on the test set with the various variants of the LSTM model. In the images, a blue pixel corresponds to a 0 and a yellow one to 1. The first row correspond to the prediction by the model and the second row is the actual intent matrix. Each column correspond to a different sample in the same batch.

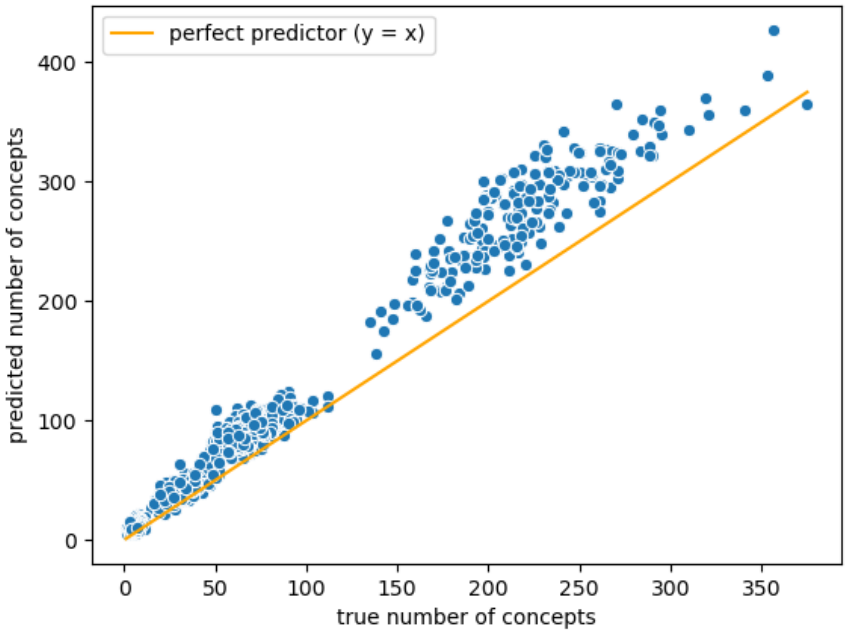


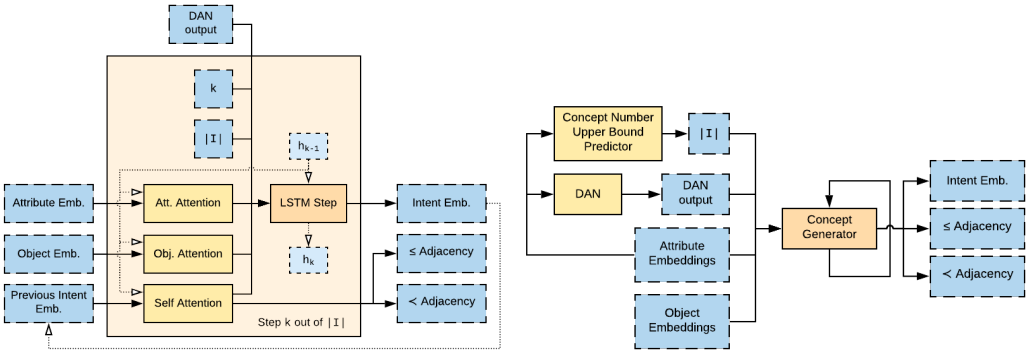
FIGURE 4.7: Upper bounds predicted by the model on all the samples of the evaluation set.

number predictor learned with BoA. Indeed, the number of intents is equal to the number of concepts in the lattice.

In our case, we have a margin of error in the predicted number of concepts $|\hat{I}|$. Indeed, the generative model (phase 3 of the approach) will be able to compensate by filling the excess space with empty intents if $|\hat{I}|$ is higher than the actual number of concepts. However, if $|\hat{I}|$ is too low, the generative model will not have enough space to generate the intents. In short, we need an upper bound of the number of concept rather than the number of concept itself. The additional freedom makes it easier to design a model fitting our needs.

We could use a theoretical upper bound, but those are rather large and increase exponentially with the size of the data (see Subsection 3.4.2). Instead, we train a MLP to predict an upper bound of the concept number, from the prediction of the model learned with BoA and an average-pooling of the attribute embeddings. We train this model using an adapted MSE described in Equation (4.1), with p the target and q the predicted number of concepts. This new loss function first shifts the prediction target 10% higher than the actual concept number, and penalizes predictions under the target by multiplying the squared error by 100 (an arbitrary number which provided good results).

We show in Figure 4.7 the upper bounds predicted on the validation set. Of the 1000 samples tested, less than 1.2% of the predictions are under the actual concept number. The remaining prediction are close to the actual numbers, in average around 125% of the true concept number.



(A) Concept generator at step k (B) Full architecture

FIGURE 4.8: Schematic representation of the intent generation architecture. Blue blocks correspond to tensors and orange to neural components.

$$
\begin{aligned}
ShiftedError(p,q) &= q - (p \times 1.1) \\
UpperBoundError(p,q) &= ShiftedError(p,q)^2 \quad \text{if } ShiftedError(p,q) > 0 \\
&= 100 \times ShiftedError(p,q)^2 \quad \text{otherwise}
\end{aligned}
\tag{4.1}
$$

4.3 Intents, Cover and Order Relation Generation

Our initial design for the intent prediction was split into 3 modules:

1. the first module would create a skeleton for the intents, from the concept number upper bound and the attribute embeddings;
2. the second module would predict the intents, from the skeleton and the object and attribute embeddings;
3. the last module would use the intents and the attribute embeddings to predict the links; this last module wouldn't share its gradient with the first two.

Experimental results led us to simplify the model by merging the 3 modules into a simpler model described in Figure 4.8. The resulting architecture relies on a DAN, an LSTM, and attention mechanisms described in the next paragraphs. The DAN takes the attribute embeddings as input, and generates a single output per FC which is then used at every step of the generation process, as shown in Figure 4.8. We expect it to learn general information about the FC similarly to how the number of concept prediction works. In addition to the output of the DAN and the attention contexts, the LSTM is fed the number of concept upper bound and the index of the current intent.

For intent generation we use a basic version of attention, equipped with multiple attention heads. Attention heads can be seen as reading heads scanning the same sequence, but each head performing a different scan. Our attention mechanism works by comparing a query q with sequence S of elements. The query at step k of the generation is obtained by flattening the hidden state generated at step $k-1$.

Let φ be a feed-forward layer with input and output size $|s|$ for $s \in S$. Let ψ_i be a feed-forward layer associated with head i with input size $|q|$ and output size

$|s|$. Finally, let $\cdot$ be the dot product. The attention weight computed by head i for element s can be expressed as $weight_i(s,q) = \varphi(s) \cdot \psi_i(q)$. A softmax is applied to all the weights of an head, which are then used to compute the attention summary $summary_i(S,q) = \sum_{s \in S} softmaxedWeight_i(s,q) \times \varphi(s)$. We also use a sigmoid variant of our attention for which the weights are $weight_i(s,q) = sigmoid(\varphi(s) \cdot \psi_i(q))$.

In our model we use 3 attentions, each equipped with 4 heads, for a total of 12 attention heads:

- an *object attention*, an attention applied on the set of object embeddings;
- an *attribute attention*, an attention applied on the set of attribute embeddings;
- a *self attention*, an attention applied on the previously generated intents.

Two of the self attention heads are of the sigmoid variant, and are used to predict the relations between the currently seen intent (or concept) and the attended intent (or concept). The sigmoid attention weight serve as a probability of there being an edge between the current concept and the previous one, *à la* GraphRNN. One of the heads is dedicated to $\leq$ and the other to $\prec$.

Compared to our preliminary experiments, the soft attention is split in two because there are two input sequences, and the link attention correspond to the two sigmoid attention heads of the self attention.

4.4 Training and First Experiments

We use the training process schematized in Figure 4.9 to train our intent generation model on the dataset used for BoA. For details on the data, refer back to Subsection 3.3.2. We train the intent model with BCE on the intent matrix and the cover and order relation. We also use MSE between the generated intent embeddings and the expected intent embeddings, produced by applying the object embedding process of BoA on the intent matrix. Using MSE on the expected intent embedding has proven, experimentally, to improve the performance and convergence speed compared to using only the BCE on the intent matrix. We also confirmed that the intent prediction performance was not hindered by the BCE for the order and cover relations. Finally, retraining the BoA decoder with the rest of the model converges faster but performs worse than training only the intent model.

We rely on two sets of measures to evaluate the performance of our model for intent generation. First, we determine the predictive performance at the scale of each component of the intent matrix using AUC ROC. Then, we apply a threshold of 0.5 on the intent matrix. We transform the resulting matrix into a set of predicted intents $\hat{I}$, so we remove all duplicate predicted intents. After removing the duplicates from the set versions of the intents, we compute the intent precision as $|\hat{I} \cap I|/|\hat{I}|$ and the recall as $|\hat{I} \cap I|/|I|$. From the precision and recall we can compute the F1 score. We also use the AUC ROC to measure the prediction performance on the relation prediction by the attention, for both $\leq$ and $\prec$. We apply it on the adjacency vector of the relations, as described in Subsection 2.2.3.

We report the performance on our evaluation set in Table 4.1, and show the prediction result for a few samples of the training set in Figure 4.10. The model presented was trained for 12h which corresponds to 50 epochs. The AUC ROC for the two relations $\leq$ and $\prec$ is close to 0.85 which is not perfect but not bad either. However, the AUC ROC on the intent matrix is 0.72, which is still above a random model but is not good enough for our application. Indeed, the precision on the intents

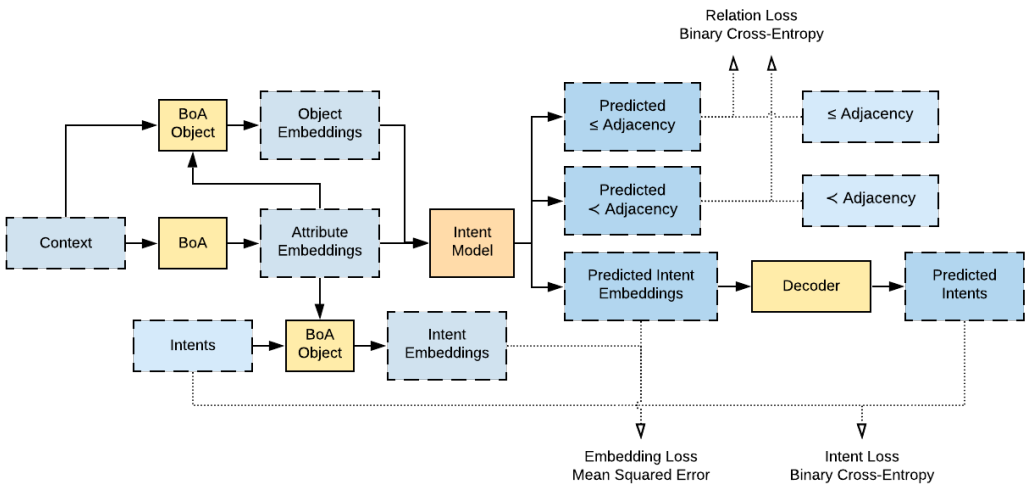


FIGURE 4.9: Schematic representation of the intent model training process.

Measure	Mean ± std.
Intent precision	0.217 ± 0.224
Intent recall	0.071 ± 0.092
Intent F1	0.046 ± 0.054
Intent matrix AUC ROC	0.720 ± 0.062
$\leq$ AUC ROC	0.852 ± 0.071
$\prec$ AUC ROC	0.846 ± 0.072

TABLE 4.1: Performance of the intent model on the evaluation set.

is under 25% in average, and the recall and F1 are under 10%. The qualitative results displayed in Figure 4.10 are better than the ones obtained in the preliminary experiments. It is interesting to note that for the first of the two samples, the first 50 predicted concepts look similar and are processed similarly by the attention generating the order relation, resulting in a slightly denser column between 0 and 50 in the adjacency of $\leq$ (6th column of the picture. This indicates that the intent and the relations are related within the model.

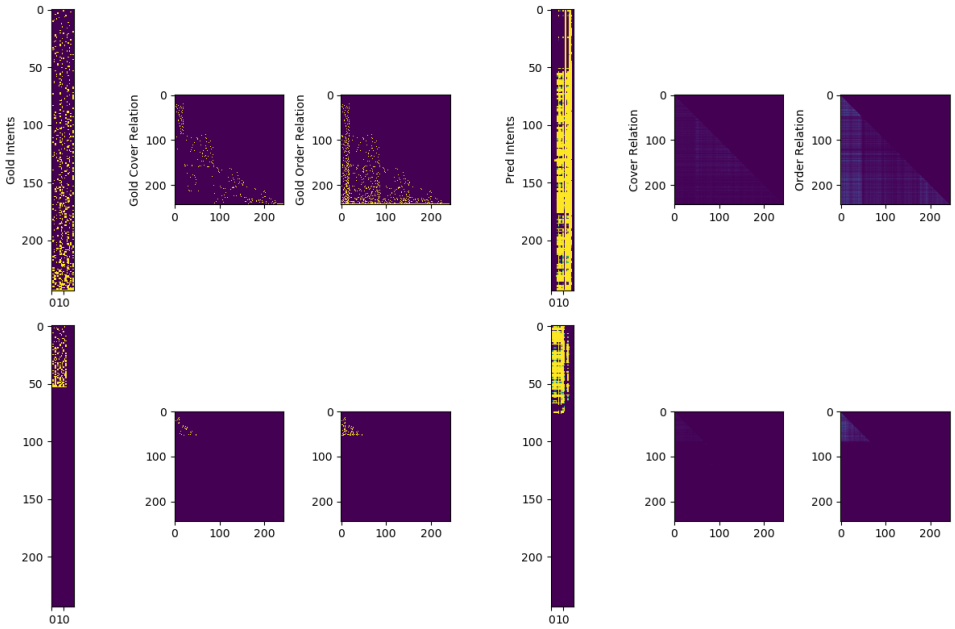


FIGURE 4.10: Batch of 2 samples (one per row) of the evaluation set and the predictions of our model. The first three columns correspond to the sample itself, and the last three to the model's prediction.

Chapter 5

Conclusion and Discussion

In this internship, we explored methods to reproduce FCA using NNs by focusing on the generation of concept lattices. The task we tackled is not much documented. To bridge this gap in the literature, we conduct a very exploratory work.

Our initial approach relied on GraphRNN, and did not produce the intended results. However, it allowed us to tackle multiple challenges. First, we developed a data format to represent lattices in a way usable in deep learning. To achieve this, we designed a concept ordering based on what we call levels, in replacement for the breadth-first search used in GraphRNN. Then, handling GraphRNN led us to explore multiple methods to represent and manipulate adjacency matrices for $\leq$ and $\prec$.

Finally, we addressed the problem of the representation of FCs of any size. To handle this challenging task, we developed BoA, an embedding architecture for FCs. This embedding method is flexible and data agnostic, and is published in the workshop FCA4AI of the conference ECAI with the article [37]. Most of the properties of BoA were designed towards intent generation. For instance, to train BoA, we designed a similarity measure for attributes based on their co-appearance in the intents, the co-intent similarity.

We evolved our approach to make the most of the properties of the new BoA model by centering the architecture on intent generation. Along the way, we designed a simple and effective model to predict a reasonable upper bound of the number of concepts. Our intent model uses a variety of attention mechanisms along with an LSTM to predict intent embeddings. This design decision was led by preliminary tests on multiple common architectures. It is interesting to note that we reproduce the inner process of GraphRNN with our self attention mechanism. Indeed, we iteratively add intents and predict their relation with previous intents, similarly to how GraphRNN handles adjacency.

The performance of the intent model is heavily dependent on the quality of the BoA model. As observed in [37], the current BoA can be improved in several ways. For instance, the co-intent similarity is not correctly predicted. We expect that embeddings already containing this notion of intent would greatly improve the performance of the generative model. Due to time constraints, we were unable to finalize the approach. However, we designed a first end-to-end intent model which provides baseline results for intent generation. We have 3 main tracks to improve our current approach, that will be explored and published during the beginning of our Ph.D. Firstly, improving BoA, the foundation of our approach, would positively impact the performance of the whole framework. We are thinking of working on the variational aspect of the embedding model, in addition to rectifying the co-intent similarity prediction. Secondly, adding the planned refiner and further training the full intent model would improve the performance. Thirdly, we have plans to explore applications of BoA on other fields than FCA, *e.g.*, pattern mining, sentiment

analysis, and information retrieval.

Finally, we used a wide variety of models, from node and graph neural frameworks to complex generative models, passing by attention mechanism and embedding models based on FCA. Most of these models come from or are extensively developed in natural language processing (NLP). Firstly, recurrent architectures like LSTM are extensively used in language modeling. Secondly, attention models first appeared for alignment automatic translation [1]. They were further developed in the transformer and reformer models [30, 44], both designed for translation too. Thirdly, word2vec [38] is one of the most famous word embedding architecture which was adapted in a wide variety of domains, including FCA with FCA2VEC [9] and node embedding with node2vec [18]. In a nutshell, a large portion of the work presented in this thesis relies on NNs used in NLP. Additionally, FCA is well known to go from row data to knowledge and supports many semantic web tasks such as ontology building, *e.g.*, extracting properties of pharmaceutical products from medical articles to build an ontology of pharmaceutical substances and their effects. The knowledge can then be fed back into NLP systems, *e.g.*, a chatbot for medical monitoring.

At the beginning of the internship, we lacked mathematical bases in lattice theory to make the most of the FCA process. The fundamentals in FCA, ontologies and data mining that are taught in the NLP master of *Université de Lorraine* greatly helped us to fill the blanks. Furthermore, our accumulated expertise in a wide variety of models used in NLP helped us in manipulating the NN architectures we encountered. It also helped us in developing a meaningful approach to FC embedding and intent generation.

Bibliography

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1409.0473.

[2] Karell Bertet et al. "Lattices, closures systems and implication bases: A survey of structural aspects and algorithms". In: *Theoretical Computer Science* (Nov. 2016). DOI: 10.1016/j.tcs.2016.11.021.

[3] Tarek R. Besold et al. "Neural-Symbolic Learning and Reasoning: A Survey and Interpretation". In: *CoRR* abs/1711.03902 (2017). arXiv: 1711.03902. URL: http://arxiv.org/abs/1711.03902.

[4] Samuel R. Bowman et al. "Generating Sentences from a Continuous Space". In: *CoRR* abs/1511.06349 (2015). arXiv: 1511.06349. URL: http://arxiv.org/abs/1511.06349.

[5] Joan Bruna et al. "Spectral Networks and Locally Connected Networks on Graphs". In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2014. URL: http://arxiv.org/abs/1312.6203.

[6] Xilun Chen et al. "Adversarial Deep Averaging Networks for Cross-Lingual Sentiment Classification". In: *CoRR* abs/1606.01614 (2016). arXiv: 1606.01614. URL: http://arxiv.org/abs/1606.01614.

[7] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *CoRR* abs/1406.1078 (2014). arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078.

[8] Fan RK Chung. "Lectures on spectral graph theory". In: ().

[9] Dominik Dürrschnabel, Tom Hanika, and Maximilian Stubbemann. "FCA2VEC: Embedding Techniques for Formal Concept Analysis". In: (2019). arXiv: 1911.11496 [cs.LG]. URL: https://arxiv.org/abs/1911.11496v1.

[10] Maximilian Felde and Tom Hanika. "Formal Context Generation Using Dirichlet Distributions". In: *Graph-Based Representation and Reasoning*. Ed. by Dominik Endres, Mehwish Alam, and Diana Şotropa. Cham: Springer International Publishing, 2019, pp. 57–71. ISBN: 978-3-030-23182-8.

[11] Yifan Feng et al. "Hypergraph Neural Networks". In: *CoRR* abs/1809.09401 (2018). arXiv: 1809.09401. URL: http://arxiv.org/abs/1809.09401.

[12] Bernhard Ganter. "Random Extents and Random Closure Systems". In: *CLA*. 2011.

[13] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.

[14] Artur S. d'Avila Garcez et al. "Neural-Symbolic Computing: An Effective Methodology for Principled Integration of Machine Learning and Reasoning". In: *CoRR* abs/1905.06088 (2019). arXiv: 1905.06088. URL: http://arxiv.org/abs/1905.06088.

[15] Andrew Gardner et al. "Classifying Unordered Feature Sets with Convolutional Deep Averaging Networks". In: *CoRR* abs/1709.03019 (2017). arXiv: 1709.03019. URL: http://arxiv.org/abs/1709.03019.

[16] Bruno Gaume, Emmanuel Navarro, and Henri Prade. "Clustering bipartite graphs in terms of approximate formal concepts and sub-contexts". In: *International Journal of Computational Intelligence Systems* vol. 6.n° 6 (2013), pp. 1125–1142. DOI: 10.1080/18756891.2013.819179. URL: https://hal.archives-ouvertes.fr/hal-01127983.

[17] Alex Graves and Jürgen Schmidhuber. "Framewise phoneme classification with bidirectional LSTM and other neural network architectures". In: *Neural networks* 18.5-6 (2005), pp. 602–610.

[18] Aditya Grover and Jure Leskovec. "node2vec: Scalable Feature Learning for Networks". In: *CoRR* abs/1607.00653 (2016). arXiv: 1607.00653. URL: http://arxiv.org/abs/1607.00653.

[19] Aditya Grover, Aaron Zweig, and Stefano Ermon. "Graphite: Iterative Generative Modeling of Graphs". In: (2018). arXiv: 1803.10459 [stat.ML]. URL: http://arxiv.org/abs/1803.10459v4.

[20] William L. Hamilton, Rex Ying, and Jure Leskovec. "Inductive Representation Learning on Large Graphs". In: *CoRR* abs/1706.02216 (2017). arXiv: 1706.02216. URL: http://arxiv.org/abs/1706.02216.

[21] Tom Hanika, Maximilian Marx, and Gerd Stumme. "Discovering Implicational Knowledge in Wikidata". In: *CoRR* abs/1902.00916 (2019). arXiv: 1902.00916. URL: http://arxiv.org/abs/1902.00916.

[22] Dmitry I. Ignatov. "Introduction to Formal Concept Analysis and Its Applications in Information Retrieval and Related Fields". In: *CoRR* abs/1703.02819 (2017). arXiv: 1703.02819. URL: http://arxiv.org/abs/1703.02819.

[23] Mohit Iyyer et al. "Deep Unordered Composition Rivals Syntactic Methods for Text Classification". In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Beijing, China: Association for Computational Linguistics, July 2015, pp. 1681–1691. DOI: 10.3115/v1/P15-1162. URL: https://www.aclweb.org/anthology/P15-1162.

[24] Aditya Jain, Gandhar Kulkarni, and Vraj Shah. "Natural Language Processing". In: *International Journal of Computer Sciences and Engineering* 6 (Jan. 2018), pp. 161–167. DOI: 10.26438/ijcse/v6i1.161167.

[25] Raehyun Kim et al. "HATS: A Hierarchical Graph Attention Network for Stock Movement Prediction". In: (2019). arXiv: 1908.07999 [q-fin.ST].

[26] Diederik Kingma and Max Welling. "Auto-Encoding Variational Bayes". In: *ICLR* (Dec. 2013).

[27] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[28] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *CoRR* abs/1609.02907 (2016). arXiv: 1609.02907. URL: http://arxiv.org/abs/1609.02907.

[29] Thomas N. Kipf and Max Welling. "Variational Graph Auto-Encoders". In: (2016). arXiv: 1611.07308.

[30] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. *Reformer: The Efficient Transformer*. 2020. arXiv: 2001.04451 [cs.LG].

[31] Ajinkya Kulkarni, Vincent Colotte, and Denis Jouvet. "Deep Variational Metric Learning For Transfer Of Expressivity In Multispeaker Text To Speech". working paper or preprint. May 2020. URL: https://hal.inria.fr/hal-02573885.

[32] Sergei Kuznetsov. "On Computing the Size of a Lattice and Related Decision Problems". In: *Order* 18 (Dec. 2001), pp. 313–321. DOI: 10.1023/A:1013970520933.

[33] Sergei Kuznetsov and Sergei Obiedkov. "Comparing performance of algorithms for generating concept lattices". In: *J. Exp. Theor. Artif. Intell.* 14 (Apr. 2002), pp. 189–216. DOI: 10.1080/09528130210164170.

[34] Sergei O. Kuznetsov, Nurtas Makhazhanov, and Maxim Ushakov. "On Neural Network Architecture Based on Concept Lattices". In: *Foundations of Intelligent Systems*. Ed. by Marzena Kryszkiewicz et al. Cham: Springer International Publishing, 2017, pp. 653–663. ISBN: 978-3-319-60438-1.

[35] Xudong Lin et al. "Deep Variational Metric Learning". In: *Computer Vision – ECCV 2018*. Ed. by Vittorio Ferrari et al. Cham: Springer International Publishing, 2018, pp. 714–729. ISBN: 978-3-030-01267-0.

[36] Tengfei Ma, Jie Chen, and Cao Xiao. "Constrained Generation of Semantically Valid Graphs via Regularizing Variational Autoencoders". In: *CoRR* abs/1809.02630 (2018). arXiv: 1809.02630. URL: http://arxiv.org/abs/1809.02630.

[37] Esteban Marquer, Ajinkya Kulkarni, and Miguel Couceiro. "Embedding Formal Contexts Using Unordered Composition". In: *FCA4AI - 8th International Workshop "What can FCA do for Artificial Intelligence?" (colocated wit ECAI2020)*. Santiago de Compostela, Spain, Aug. 2020. URL: https://hal.archives-ouvertes.fr/hal-02912874.

[38] Tomas Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". In: Jan. 2013, pp. 1–12.

[39] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. "DeepWalk: Online Learning of Social Representations". In: *CoRR* abs/1403.6652 (2014). arXiv: 1403.6652. URL: http://arxiv.org/abs/1403.6652.

[40] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *CoRR* abs/1609.04747 (2016). arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747.

[41] Sebastian Rudolph. "Encoding closure operators into neural networks". In: *Proceedings of the 3rd International Conference on Neural-Symbolic Learning and Reasoning-Volume 230*. CEUR-WS. org. 2007, pp. 40–45.

[42] Ralf C. Staudemeyer and Eric Rothstein Morris. *Understanding LSTM – a tutorial into Long Short-Term Memory Recurrent Neural Networks*. 2019. arXiv: 1909.09586 [cs.NE].

[43] Shanshan Tang, Bo Li, and Haijun Yu. "ChebNet: Efficient and Stable Constructions of Deep Neural Networks with Rectified Power Units using Chebyshev Approximations". In: (2019). arXiv: 1911.05467 [cs.LG].

[44] Ashish Vaswani et al. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[45] Petar Veličković et al. "Graph Attention Networks". In: (2017). arXiv: 1710.10903.

[46] Zonghan Wu et al. "A Comprehensive Survey on Graph Neural Networks". In: *CoRR* abs/1901.00596 (2019). arXiv: 1901.00596. URL: http://arxiv.org/abs/1901.00596.

[47] Rikiya Yamashita et al. "Convolutional neural networks: an overview and application in radiology". In: *Insights into imaging* 9.4 (2018), pp. 611–629.

[48] Liang Yang et al. "Masked Graph Convolutional Network". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, July 2019, pp. 4070–4077. DOI: 10.24963/ijcai.2019/565. URL: https://doi.org/10.24963/ijcai.2019/565.

[49] Yiding Yang et al. "SPAGAN: Shortest Path Graph Attention Network". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, July 2019, pp. 4099–4105. DOI: 10.24963/ijcai.2019/569. URL: https://doi.org/10.24963/ijcai.2019/569.

[50] Rex Ying et al. "Hierarchical Graph Representation Learning with Differentiable Pooling". In: *CoRR* abs/1806.08804 (2018). arXiv: 1806.08804. URL: http://arxiv.org/abs/1806.08804.

[51] Jiaxuan You et al. "GraphRNN: A Deep Generative Model for Graphs". In: *CoRR* abs/1802.08773 (2018). arXiv: 1802.08773. URL: http://arxiv.org/abs/1802.08773.