Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Learning Complex Natural Language Inferences with Relational Neural Models

Boris Rakovan

Supervisor: Ing. Gustav Šír, Ph.D.
Field of study: Open Informatics
Subfield: Artificial Intelligence and Computer Science
May 2022

Acknowledgements

I would like to thank my supervisor, Ing. Gustav Šír, Ph.D., for his help, guidance and invaluable advice during my work on this thesis.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 19. 5. 2022

Abstract

Large language models based on neural networks have recently achieved remarkable results across a wide range of NLP tasks. However, they still struggle with deeper understanding of semantics and reasoning about entities and their relationships. To address these shortcomings, in this thesis we examine alternative deep machine learning architectures with the intent of testing their logical reasoning and systematic generalization capabilities. We propose and implement several deep learning models, mainly from the deep relational learning category, and evaluate the proposed models on a textual benchmark selected from the NLU domain. In the empirical part, we manage to demonstrate the substantial performance gap between the standard NLU models that work with unstructured text data and more advanced models, mainly graph and recurrent neural networks, that allow to process more structured inputs. Finally, we propose suitable relational model biases to address the particular forms of relational reasoning in the selected benchmark and manage to achieve results comparable to state-of-the-art.

Keywords: natural language processing, natural language understanding, deep relational learning, graph neural networks, text classification

Supervisor: Ing. Gustav Šír, Ph.D.
Praha
Resslova 307/9
Faculty of Electrical Engineering

Abstrakt

Velké jazykové modely založené na neuronových sítích dosáhly v nedávné době pozoruhodných výsledků napříč mnoha úkoly z oblasti zpracování přirozeného jazyka (NLP). Přesto se ukázalo, že je pro tyto modely problematické vyvozovat z textu logické závěry vyžadující jeho hlubší porozumění. V této bakalářské práci se pokusíme adresovat tyto nedostatky tím, že prozkoumáme různé architektury z oblasti hlubokého strojového učení se speciálním zaměřením na jejich schopnost logické dedukce a přenášení naučených poznatků. Navrhneme a implementujeme několik neuronových modelů, včetně architektur z kategorie hlubokého relačního učení, a všechny následně otestujeme na vybraném úkolu z oblasti porozumění přirozeného jazyka (NLU). Dosažené výsledky nám pomohou demonstrovat významné rozdíly ve výkonu standardních NLP modelů, které na vstupu pracují s čistým textem, a pokročilejších modelů – obzvláště grafových a rekurentních neuronových sítí – které dokáží zpracovat vstupní data ve vhodnější strukturované formě. Kromě toho implementujeme vhodné formy předzpracování vstupních dat, což nám pomůže dosáhnout výsledků srovnatelných se state-of-the-art na daném datasetu.

Klíčová slova: zpracování přirozeného jazyka, porozumění přirozeného jazyka, hluboké relační učení, grafové neurónové sítě, klasifikace textu

Překlad názvu: Učení složitých závislostí v textu pomocí relačních neurálních modelů

Contents

Figures

Tables

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Rakovan Boris**	Personal ID number:	**483500**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence and Computer Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Learning Complex Natural Language Inferences with Relational Neural Models

Bachelor’s thesis title in Czech:

Učení složitých závislostí v textu pomocí relačních neurálních modelů

Guidelines:

Large language models based on neural networks have recently achieved remarkable performances across a wide range of NLP tasks. However, they still struggle with deeper understanding of semantics and reasoning about entities and their relationships [1], which has been traditionally addressed by the symbolic AI approaches [2]. Recently, structured neural models utilizing inductive priors to address graph-structured [3] and relational [4] data have emerged to address similar problems in other domains. The subject of this thesis is to explore their use on a sample benchmark from the NLP domain.
1) Review prior art on:
a) Graph Neural Networks and (Deep) Relational learning,
b) Natural language processing, understanding, and inference
- with a special focus on (semantic) parsing into relational (graph) representations [7,8].
2) Review existing NLP challenges and find a suitable benchmark for relational reasoning (e.g. [5]).
3) Get acquainted with building (deep) NLP pipelines [6] and the target neural frameworks
- esp. GNNs [3] and LRNNs [4].
4) Propose suitable relational model biases to address the particular forms of relational reasoning present in the textual benchmark(s).
5) Compare your solution against standard (deep) NLP models without the relational prior.
6) Discuss your findings.

Bibliography / sources:

[1] Workshop on Enormous Language Models @ ICLR 2021: https://welmworkshop.github.io/
[2] Raedt L. (2011) Inductive Logic Programming. In: Encyclopedia of Machine Learning. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-30164-8_396
[3] Zhou, Jie, et al. "Graph neural networks: A review of methods and applications." AI Open 1 (2020).
[4] Šourek, G., Železný, F. & Kuželka, . Beyond graph neural networks with lifted relational neural networks. Mach Learn 110, 1695–1738 (2021).
[5] Sinha, Koustuv, et al. "CLUTRR: A diagnostic benchmark for inductive reasoning from text." arXiv preprint arXiv:1908.06177 (2019).
[6] Stanford NLP stack: https://github.com/stanfordnlp/CoreNLP
[7] Semantic parsing https://en.wikipedia.org/wiki/Semantic_parsing
[8] Krynsky, Daniel: Relational Learning with Neural Networks for Machine Translation, CTU theses, 2018.

Name and workplace of bachelor’s thesis supervisor:

Ing. Gustav Šír, Ph.D.   Department of Computer Science FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **07.01.2022**   Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

Ing. Gustav Šír, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt   Student’s signature

Chapter 1

Introduction

In the first chapter, we outline the overall structure of the bachelor thesis and provide a detailed definition of the problem that this work will be concerned with. We also introduce the motivation behind the topic and briefly describe the approaches will be taken to achieve the laid out goals.

1.1 Motivation

In recent years, the advances in natural language processing (NLP) discipline have sparked a wave of interest in its applications across a wide range of tasks. At the core of many of these tasks there is the issue of natural language understanding (NLU) - a branch of NLP that deals with machine reading comprehension. Thanks to the emergence of large-scale datasets and deep learning, many reading comprehension models have already surpassed the human performance on various benchmarks [1]. However, there are concerns regarding the ability of state-of-the-art language models to understand deeper semantics and reason about the entities in text and their relations. To this date, several approaches were proposed to address this issue, including variations of neural models that use inductive priors to accommodate graph-structured and relational data. In this work, we will explore several such deep learning architectures, with a special focus on their use in the NLU domain.

1.2 Problem definition

The aim of this thesis is to explore the application of different neural architectures, especially from the deep relational learning domain, on a specific natural language understanding problem. The carefully selected dataset introduced in later sections - CLUTRR - will serve as a benchmark that will be used to test the relational reasoning capabilities of different deep learning models. Throughout the thesis, we explain the necessary theoretical aspects and describe the practical considerations that motivated the decisions taken in the experimental part. We will seek to utilize various inductive priors that correctly address the particular structure of the input data, with the aim to maximize the accuracy of models on the given text classification task.

1.3 Thesis outline

In Chapter 2, we introduce the fundamental theoretical concepts that will be built upon in the following chapters. More specifically, we get acquainted with basic topics from the natural language processing and deep learning domains. In this chapter, we devote special attention to the different neural architectures that were chosen to be implemented and evaluated in the experimental part of this work.

We follow this in Chapter 3 by reviewing the relevant work that has been done on the subject to date and analyzing existing datasets that were considered to be used in the experimental part. Then, we enumerate the different criteria that drove the selection process and present in detail the final selected benchmark.

In Chapter 4, we proceed to experiment with different deep learning models and input preprocessing methods. In total, 12 different experiments will be described and carried out, followed by an analysis of the obtained results.

We conclude this work in Chapter 5 by reflecting on the achieved results and pinpointing the future research directions in this area.

Chapter 2

Theoretical foundations

This chapter introduces several fundamental areas that compose the theoretical background of this thesis. Mainly, we introduce the natural language processing, standard deep learning and deep relational learning disciplines, with special focus on the deep neural architectures that will be used in the forthcoming experiments. The goal of this part is to provide the reader with a basic knowledge of certain topics that will be referenced in the following chapters of this thesis.

2.1 Natural language processing

Natural language processing (NLP) refers to a branch of artificial intelligence (AI) that explores how computers can be used to achieve human-like language processing abilities. The historical development of NLP is rooted in a range of disciplines, including computer science, linguistics, mathematics, electrical engineering and robotics. Today, NLP is one of the fastest-growing AI research field, with vast array of innovative publications being released every year. It is used to solve a multitude of tasks in the modern world. Some of the more relevant to the purpose of this thesis include information extraction, text tokenization, part of speech tagging or dependency parsing. On a more general level, the applications of NLP include spam detection, machine translation, question answering, social media sentiment analysis and speech recognition.

2.1.1 Evolution

In the beginning, most of the NLP systems were based on complex sets of hand-coded rules. However, this was not sufficient to accommodate the increasing volumes of voice and text data that needed to be processed. Therefore, in the beginning of the 21st century, hand-coded rules were replaced by new statistical methods powered by more sophisticated computer algorithms. The emerging statistical methods could, nevertheless, still not equip the machines with significant language understanding capabilities. The major breakthrough arrived in the 2010s with the increase of computational power and subsequent rapid spike in the popularity of neural networks in machine learning.

In present, large deep learning models and architectures based on neural networks achieve state-of-the-art results in many natural language problems. Thanks to the advances in deep learning, we are able to efficiently process huge amounts of raw, unstructured text and voice data, extracting accurate meaning representations to be used on downstream tasks. Today, the models of choice in many NLP applications are large, pre-trained language models called transformers that will be introduced in Section 2.2.3. Other popular techniques used in the NLP pipeline include word embeddings introduced in Section 2.1.2.

2.1.2 Topics in NLP

This section presents a selection of topics from the NLP domain that were deemed essential for the experimental part of this thesis.

Dependency parsing

Dependency parsing, along with similar techniques such as constituency parsing, refers to a process of extracting the dependencies among words in a sentence. The result of the dependency parsing process is a dependency tree, where the nodes correspond to words and the edges represent binary syntactical relations between them. The edge types are drawn from a fixed set of grammatical relations such as nominal subject, direct object, adjectival modifier, and others. For a complete listing of possible relations along with their explanation, we refer the reader to chapter 14 of the comprehensive

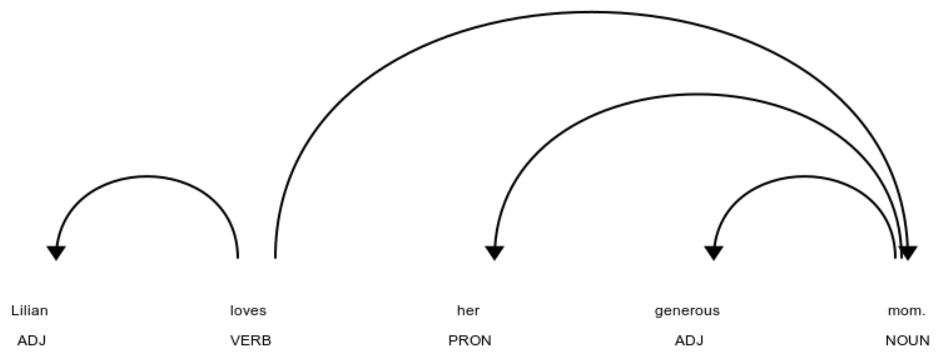


Figure 2.1: Dependency graph of an example sentence *Lilian loves her generous mom.* Image was generated using *spacy* python library.

book written on the subject of NLP - *Speech and Language Processing*[1]. In simple terms, a directed edge in the dependency tree of a sentence that starts in w_1 (head word) and ends in w_2 (dependent word) means that w_2 directly modifies w_1. Figure 2.1 illustrates one such dependency graph of an example sentence.

Word embeddings

An embedding is a contextual vector representation of a word (or other unit of speech) that captures its meaning. The idea of encoding words as fixed-size vectors in some finite linear space has brought a revolution to the field of NLP. Since then, several algorithms to create such vector representations were invented, most notably *word2vec* [3] and *Glove* [4]. In practice, machine learning practitioners often utilize word embeddings that are already trained and are available for download on the internet. These pre-trained embeddings usually vary in two main aspects - the dimensionality of the word vectors, that typically ranges from 50 to 500, and the size of the vocabulary for which the embeddings are available. One of the desired properties of word embeddings is that vectors corresponding to similar words are *close* to each other with respect to a metric defined on the linear space. This property is illustrated for a group of selected words on Figure 2.2.

The embeddings usually represent individual words, however, one can easily derive a vector representation of longer passages of text, such as sentences

[1]https://web.stanford.edu/~jurafsky/slp3/

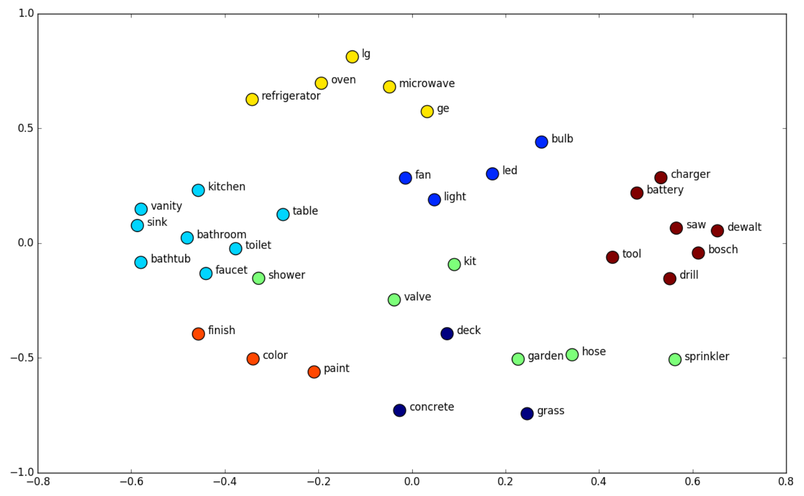


Figure 2.2: Selected word embeddings projected to 2D space using a dimensionality reduction technique. Image was taken from [2].

or an entire article. This can be achieved either by summing or averaging the embeddings of individual tokens from the sequence. This dense vector representation of longer text passages often replaces less effective techniques used in past, such as simple one-hot or bag-of-words representations that produce sparse vectors (vectors with large number of zeros). Later, we will describe how to obtain a numeric representation of a sequence of tokens in an even more effective way - using recurrent neural networks.

2.2 Standard deep learning

Deep learning is a special type of machine learning that aspires to imitate the way humans learn knowledge and gain intelligence. It differs from traditional machine learning methods in two significant ways. Firstly, it learns the representation of the raw input progressively, in multiple layers stacked on top of each other. For example in image recognition, the lower layers might learn to identify subtle features of the input, such as edges, while the higher layers might take up representations that are close to how humans discern images, such as digits, letters or even faces. Secondly, it removes the need for manually engineering input features. This is a significant leap from previous machine learning methods where the input for the model often had to be hand-crafted by experts with extensive domain knowledge - a process that could take significant amounts of time. In contrast, deep neural networks can process any kind of raw data, as long as it is preprocessed to numeric format.

This includes images, speech, or texts. The neural network is then able to learn the important hidden features of the input automatically.

In the rest of the chapter, we introduce several specific deep learning architectures that will be referenced in the experimental part of the work. We expect the reader to be familiar with basic concepts that govern the functioning of artificial neural networks, such as neuron layers, activation and loss functions, back-propagation, dot product, regularization and others. Detailed explanations of these concepts will be omitted, and we will rather focus on the more advanced topics that are directly relevant to the problem that we will be solving. Reader without prior knowledge of these topics is encouraged to study appropriate literature, for example a book titled Deep Learning[2] written by Goodfellow, Bengio and Courville, in order to get a full understanding of the rest of this chapter.

2.2.1 Feed-forward neural networks

A feed-forward neural network is the first invented and simplest kind of artificial neural network. In feed-forward networks, the information moves only in the forward direction as the network learns a hidden representation of the input through a series of hidden layers. Typically, the input to these networks is constrained to fixed-size numeric tensors.

As any other kind of neural network, the feed-forward network is composed of layers that are in turn composed of computational units called neurons. As the inputs from previous layer enter the next layer, they are multiplied by the weights associated with each neuron of the new layer. The results of this multiplication are then summed, incremented by a scalar value called bias, and finally, passed to a non-linear activation function. The activation functions are what allows the network to approximate all kinds of complex, non-linear functions. The network parameters - neuron weights and biases - are then optimized during the training process using the backpropagation algorithm.

In the simplest case, the neural networks consists only of a single layer of output nodes. This simple architecture is known as a single-layer perceptron and is usually not used in deep learning, as it is only capable of learning linearly separable patterns. In practice, feed-forward networks consist of multiple stacked layers of neurons where the neurons between each two successive layers are connected together.

[2]https://www.deeplearningbook.org/

The computation that takes place at the first neuron in one layer of a feed-forward network can be described using the following formula:

$$y_1 = \sigma\left(w_{1,0}x_0 + w_{1,1}x_1 + \ldots + w_{1,n}x_n + b_1\right) \; = \sigma\left(\sum_{i=1}^{n} w_{1,i}x_i + b_1\right)$$

where σ is an activation function (usually sigmoid, tanh or ReLU), $w_{1,i}$ are the scalar weights of the first neuron and x_i are the activations (outputs) of previous layer.

The computation that takes place in an entire feed-forward layer can be described using a matrix form of the previous formula:

$$\boldsymbol{y} = \sigma\left(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}\right)$$

2.2.2 Recurrent neural networks

Recurrent neural network (RNN) is a descendant of feed forward network where the connections between certain neurons form loops (also referred to as cycles or feedback connections in other literature). The rise of RNNs was driven by the shortcomings of existing neural architectures, mainly by their inability to process variable-length inputs. The core benefit that comes with RNNs is that they allow us to operate on sequences - sequences as the input, as the output, or both. In the recent years, different variations of recurrent networks were used to achieve state-of-the-art results in a variety of disciplines, including speech recognition, language modeling, image captioning or translation.

Essentially, RNNs operate similarly to feed-forward networks introduced in the previous section - they accept an input vector $\boldsymbol{x}$ and produce an output vector $\boldsymbol{y}$. The crucial difference is that the output vector is also influenced by all the inputs that were passed to the network prior to it. This is achieved by maintaining an internal hidden state that is updated progressively as new inputs enter the recurrent network, and then using this hidden state to compute the output given the input vector.

Internally, a simple recurrent neural network stores its learnable parameters in three matrices $\boldsymbol{W}_{hh}$, $\boldsymbol{W}_{xh}$ and $\boldsymbol{W}_{hy}$, and its hidden state in vector $\boldsymbol{h}$. One step of computation that the network performs can be characterized using the following formulas:

$$\boldsymbol{h}_t = \tanh(\boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{xh}\boldsymbol{x}_t)$$
$$\boldsymbol{y}_t = \boldsymbol{W}_{hy}\boldsymbol{x}_t$$

Where $\boldsymbol{x}_t$, $\boldsymbol{h}_t$ and $\boldsymbol{y}_t$ are the input, hidden state and output of the network at step t, respectively, and the tanh non-linearity is applied element-wise.

In theory, there is no limit on the length of the sequence that a RNN can process. In practice, however, the performance of classic RNNs tends to decrease with increasing length of the input. Similarly to standard neural networks, RNNs are also trained using the back-propagation algorithm, which is based on partial derivatives and gradient computation. Applying partial derivation to the parameters of the memory cells further from the end of the network causes the gradient values to be too small and thus have little effect on the parameter updates, which in turn diminishes the learning capacity of the network. This is mainly caused by the back-propagation dynamics - as the gradients *flow* back from the output layers, they get multiplied with gradients of the non-liner activation functions, whose outputs are relatively close to zero. This problem is termed the *vanishing gradient problem* and is the main reason why networks such as LSTM [5], described in the following section, were devised.

Long short-term memory networks

Simple recurrent networks work well for short input sequences, but there are often cases where the gaps between the pieces of relevant information and the point at which it is needed become large. Long short-term memory network is a more complex kind RNN architecture that is capable of learning long-term dependencies in the input. This structure allows it to solve the vanishing gradient problem that arises with longer input sequences.

The control flow in a LSTM network is similar to that of basic RNNs - it processes data sequentially and stores a hidden state as it propagates forward. The difference is in the operations that take place within the LSTM cells and that allow the network to selectively keep or discard information. To achieve this, LSTM networks use the concept of gates - small and usually shallow feed-forward layers that decide what information is relevant and thus should added or removed from the cell state. There are three different gates that regulate the information flow - input gate, forget gate and output gate. Following is a very simplistic explanation of the internal LSTM dynamics governed by the three gates:

- forget gate - decides what information to discard from the previous hidden state
- input gate - decides what new information should be stored in the hidden state
- output gate - decides what information the network is going to output

The specific update equations that take place in a LSTM cell are rather complicated and not necessary to grasp a basic understanding of the internal workings of the network. Therefore, they will be omitted from this section.

2.2.3 Transformers

A transformer is a deep learning model that is designed, similarly to recurrent neural networks, to handle sequential input data such as text or speech. Transformer models are large language models trained on vast amounts of data in an unsupervised fashion and can be fine-tuned to achieve state-of-the-art performance on lots of popular NLP benchmarks. The rise of transformers has led to development of enormous language models such as Generative Pre-trained Transformer (GPT) and Bidirectional Encoder Representations from Transformers (BERT) [7]. In contrast to recurrent neural networks, transformers do not contain any recurrent connections. Instead, they make use of a mechanism called attention to selectively attend to all parts of the input sequence, contrary to recurrent networks that process the sequences in their original order.

Attention mechanism

Attention is relatively new and powerful concept that was introduced by Vaswani et al. in a very influential paper titled *Attention Is All You Need* [6].

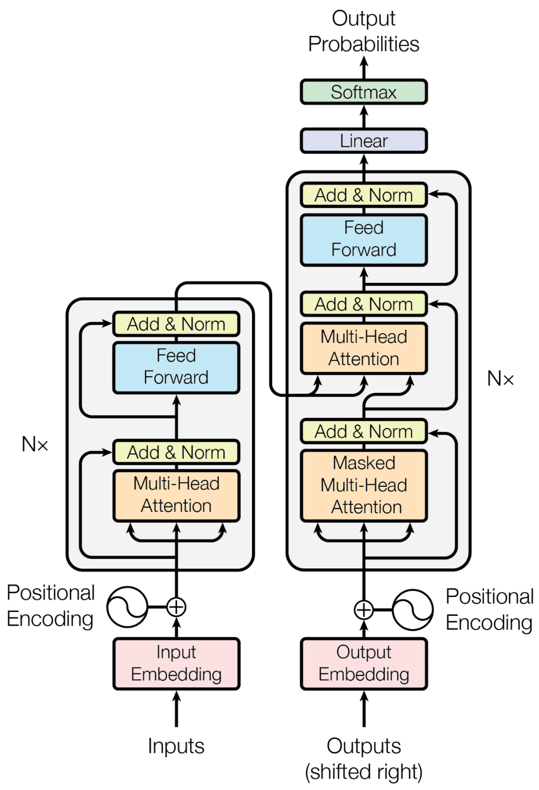


Figure 2.3: Transformer model architecture with the encoder on the left and decoder on the right. Image was taken from [6].

The attention mechanism has revolutionised the way many NLP tasks such as translation, text summarization or question answering are solved today.

In essence, attention is a mechanism that is able to capture the relative interdependence between all pairs of tokens from the input sequence. While recurrent networks store their hidden state in a single fixed-size vector and thus have limited access to the information provided in different parts of input sequence, the attention mechanism allows the model to draw richer information about any single part of the input sequence at any point. Transformers make use of multiple attention components at different stages. Mainly, they benefit from a specific kind of attention called self-attention. While generic attention is often applied to transfer information between different components of the architecture, self-attention is used to model dependencies of different parts of the sequence within the same layer.

In general, there are three different learnable weight matrices used to capture the state in an attention layer of a model. They are usually called key ($\boldsymbol{K}$), value ($\boldsymbol{V}$) and query ($\boldsymbol{Q}$). The computations that take place at the attention layers can be described using the following formula:

$$\text{attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}}\right)\boldsymbol{V}$$

The concept of attention was further built upon to create a multi-head attention mechanism. The multi-head attention with size h linearly projects the query, key and value h times, each time using a different learnable projection $\boldsymbol{W}$. Then, it applies the basic attention mechanism to each of these projections, concatenates the results and computes the final attention using another learnable projection $\boldsymbol{W}_O$:

$$\text{multihead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}_O$$
$$\text{head}_i = \text{attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V)$$

Architecture overview

Generally, transformer models are built from two components - encoder and decoder. The first component - encoder - takes a sequence of variable length (for example a sentence in English) and iteratively feeds it to a number of layers of different kinds to transform it to a hidden representation of a fixed shape. The second component - decoder - then receives the output of the encoder and maps it to a new variable-length sequence (for example the Spanish translation of the input sentence). Each of the two components is itself composed of multiple layers of different types and purposes. The graphical overview of the entire transformer architecture is displayed on figure 2.3.

While the full encoder-decoder architecture is indispensable for a number of tasks, there are also many tasks that only require either the encoder or the decoder part to be present. The decoder block is crucial mainly for tasks that involve generation of any kind, such as generative question answering or language translation. On the opposite side, encoder-only models are suitable for tasks such as extractive text summarization or text classification. As we will deal explicitly with the text classification task in this work, we will further shift our focus specifically toward the encoder-only architecture. Note that we intentionally omit some complex specifics of the transformer architecture to reduce the overall complexity and length of the text that would otherwise be required.

Internally, the encoder block is composed of the following modules:

1. Embedding layer - The input to the encoder block is first passed through a learnable embedding layer, which embeds the input tokens to a linear space of certain dimension. The dimensionality of the input is a hyper-parameter and is selected with respect to the task at hand.
2. Positional encoding - Next, there is a positional encoding layer that enriches the input embeddings with information about the relative positions of the tokens in the input sentence. Note that this step is crucial because unlike recurrent networks, transformers process the entire input sequence at the same time and thus inherently lose the ability to capture information about the positions of the individual tokens. Internally, the positional encoding layer increments the input by a vector computed using sine and cosine functions of different frequencies for different positions of the tokens.
3. Encoder layers - The encoder block, as presented originally, consists of a stack of several identical layers. Each layer is composed of two main components - multi-head self-attention and and a fully connected feed-forward neural network with ReLU activations.
4. Normalization layer - After each of the layers, there is a normalization layer used to stabilize the hidden state dynamics using various computational tricks.
5. Transformer head - Usually, the output of the previously described layers is fed as an input for the decoder block. However, in the case of encoder-only model, the decoder is not used and the final step depends on the specific task. In case of classification, it is usually a small number of feed-forward layers that project the output to the desired dimension followed by a softmax layer to generate a probability distribution over the classes.

The last step described above - head layers - are trained using the standard back-propagation algorithm and are essential for making the model work on the specific down-stream tasks. Optionally, one can also decide whether he wants to fine-tune the pre-trained parameters of the transformer itself. This is usually done using much finer learning rate than the one used for the head layers and it usually augments the performance of the model on the down-stream task even more.

In the forthcoming experiments, we will be working with a special kind of encoder-only transformer known as BERT (more precisely, its distilled version called DistilBERT).

2.3 Deep relational learning

Traditional deep learning methods work on grid-like tensor data such as images and in many cases, this can be a sufficiently expressive representation. However, lots of learning tasks require dealing with input data that is generated from non-Euclidean domains and contains complex relationships and interdependencies between entities [8]. Extending deep neural models to irregular domains has been an emerging research area.

Recently, more structured neural architectures that use inductive priors to accommodate the extra relational information in such inputs have emerged. These architectures are believed to augment the expressive capabilities of a neural network by exploiting the connections between different entities in the input [9].

Examples of these include Graph neural networks and Lifted relational neural networks. The following sections will introduce both while the rest of the work, including the experimental part, will mainly focus on the former.

2.3.1 Graph neural networks

In computer science, a **graph** is an abstract data type that consists of a set of vertices (nodes) and a set of edges between them. More formally, a graph is defined as an ordered pair $G = (V, E)$, comprising of a set of vertices V and a set of vertex pairs $E = \{(u_1, v_1), ..., (u_n, v_n)\}, u_i, v_i \in V$ called edges. Graph G is said to be **directed** if the pairs in E are ordered. A graph may also assign a numerical attribute or a symbolic label to each edge, in which case it is referred to as a **weighted** graph.

A **complete graph** is a graph in which any two distinct vertices are connected by a unique edge. The number of edges in a complete graph is thus equal to $\frac{|V|*(|V|-1)}{2}$.

A **tree** is a special kind of graph in which any two vertices are connected by exactly one path. The number of edges in a tree is equal to $|V| - 1$.

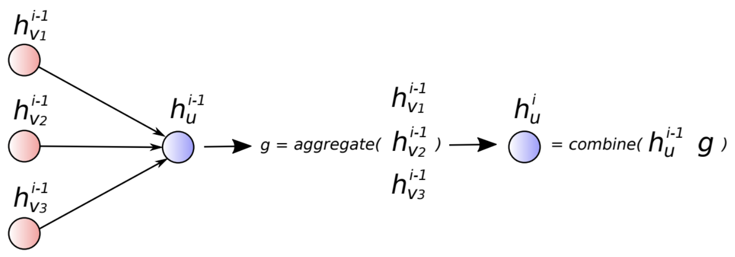
Graphs provide a natural way of representing irregular structures that appear everywhere around us, whether it is chemistry, biology, or social networks. The structure of real-world graph datasets can vary widely in terms of number of nodes and edges. The complexity and heterogeneous nature of these structures has proven to be challenging for traditional machine learning algorithms. As a response to this setback, a specific kind of neural network architecture was introduced.

Graph neural network (GNN) is a modification of standard neural network that operates on graph-structured data and captures the relational information via a neighborhood aggregation mechanism. In the recent years, GNNs have become very useful for a range of task from the machine learning domain. The applications of GNNs include antibacterial discovery, traffic prediction, physics simulations, fake news detection, complex recommendation systems and many more. Most of the problems that GNNs can solve can be classified into one of the following categories:

- node-level classification
- graph-level classification
- edge-level classification
- graph visualization
- graph clustering

As an input, the GNN requires the representation to be permutation invariant, which in other words means that the representation should not depend on the order of nodes. Therefore, one cannot simply use the adjacency matrix of a graph, since we can find a whole group of adjacency matrices can encode the same connectivity. In practice, the most common way of representing a graph is to map individual graph attributes (nodes, edges) to their fixed-size real-valued vector representations called embeddings. If the nodes correspond for example to words, one can use word embeddings introduced in the previous sections and set them fixed. However, it is more often that the pre-trained embeddings are not available, in which case the embeddings are initialized randomly and optimized during the training process jointly with the rest of the network.

During the training process, GNNs perform a differentiable transformation of the input graph called **message passing**. This operation is the backbone of all graph neural architectures today. Many different variations of message passing exist, however, they all conform to some general constraints. Mainly, the message passing operation must preserve the graph permutation invariance



h_{v1}^{i-1}
h_{v2}^{i-1}
h_{v3}^{i-1}
h_{u}^{i-1}
g = aggregate(h_{v1}^{i-1} h_{v2}^{i-1} h_{v3}^{i-1})
h_{u}^{i}
= combine(h_{u}^{i-1} g)

Figure 2.4: Illustration of the node representation update during the message passing. It shows how the new representation h_i^u of node on the right is computed aggregating the representations of its neighbours. Image was taken from [10].

and the connectivity of its edges. In each message passing step k, a new representation $\boldsymbol{x}_i^{(k)} \in \boldsymbol{R}^m$ is computed for each node i from its previous representation $\boldsymbol{x}_i^{(k-1)}$ and all its neighboring edge features $\boldsymbol{e}_{j,i} \in \boldsymbol{R}^d$ using the following formula

$$\boldsymbol{x}_i^{(k)} = \gamma^{(k)} \left(\boldsymbol{x}_i^{(k-1)}, \varphi_{j \in \mathcal{N}(i)} \, \phi^{(k)} \left(\boldsymbol{x}_i^{(k-1)}, \boldsymbol{x}_j^{(k-1)}, \boldsymbol{e}_{j,i} \right) \right),$$

where $\mathcal{N}(i)$ is the set of nodes adjacent to node i, γ and ϕ denote some learnable functions (e.g. multi-layer perceptrons), φ denotes an aggregation function (usually mean or sum), and m and d are the dimensionalities of node embeddings and edge embeddings, respectively. In essence, the message passing works in three steps:

1. for each node, gather the embeddings of its neighbors
2. use an aggregation function to aggregate these embeddings
3. pass the result to the update function that performs the final computation

For a visual illustration of a node representation update we refer the reader to figure 2.4.

As with standard neural architectures, we can stack an arbitrary number of message passing layers on top of each other. An overview of a simple graph neural network is shown on figure 2.5. Naturally, after performing n steps of message passing, each node contains information from all nodes at most n steps away from it. In practice, there are different types of message passing rules, distinguished by the specific functions γ, ϕ, φ that are used.

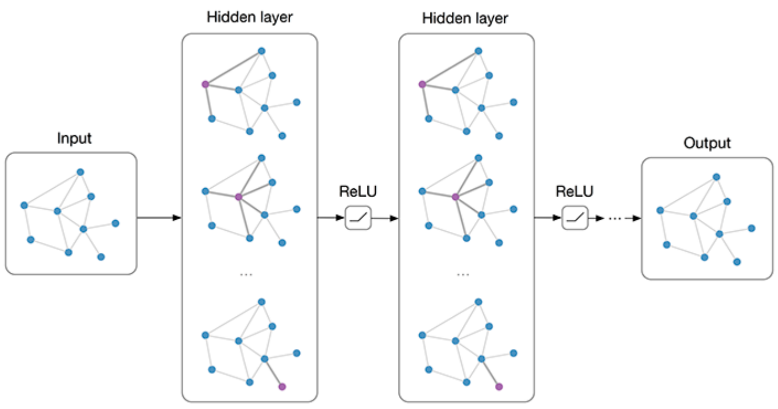


Figure 2.5: Overview of a simple graph neural network. Image was taken from [11].

This, together with the number of layers and the dimensionality of input embeddings, are the key choices to make when training a graph neural network.

There are additional steps needed after obtaining the latent representations of graph attributes in order to perform the final classification task. For node-level classification, we simply apply a linear classifier as the output layer, mapping each node to the dimension corresponding to the number of classes. In graph-level prediction task, we need a way of aggregating the node-level information before applying the linear transformation. This aggregation is usually achieved by summing or averaging the learned node embeddings.

Since the rise of deep learning on graphs, different variants of GNNs were established in the deep learning community. These variants are mainly distinguished by different forms of the general message passing rule used to update the node representations. We will consider two concrete modifications, namely the graph convolutional networks and graph attention networks, as in the recent years they have both outperformed many of the state-of-the-art models on various tasks.

Graph convolutional network (GCN) was first introduced in a paper by Bruna et al. [12] where the authors explore possible generalizations of convolutional neural networks to inputs coming from more general domains. The key idea behind this variant of GNN is that the message passing rule can be viewed as a generalization of the operation performed by convolutional neural networks. In the essence, both are operations that aggregate the neighboring nodes' information and update the current element's value according to some rule. The important distinction is that in images, the number of neighbors is based on spatial locality and is always constant whereas in graphs, the number of neighbors varies and is based on the topology of the graph. During the update step, GCNs update the representation of the current node by convolving the representations of its direct neighbors. The exact convolutional operator is, however, not defined explicitly and can take on many different forms.

Graph attention network (GAT) further builds upon the GCN architecture. It uses a special kind of convolutional operator that includes the self-attention mechanism already described in Section 2.2.3. Thanks to the self-attentional layers, the GAT is able to implicitly learn different weights for different nodes in a neighborhood of the currently updated node. This is believed to address the shortcomings of prior methods based on graph convolutions without the attention mechanism [13].

An interesting analogy can be drawn between graph attention networks and transformer models introduced in Section 2.2.3. As explained earlier, transformers use the attention mechanism to figure out how important are all the other words with respect to each word in the sequence. This is similar to the operation that takes place during message passing in a GAT layer, where each node gathers the information from its neighboring nodes and applies the learnt pair-wises attention to update its representation. Transformers can thus be seen as graph attention networks that operate on a complete graph where the nodes correspond to all words in the input sequence.

2.3.2 Lifted relational neural networks

In Lifted relational neural network (LRNN) introduced by Sourek et al. [14], various problems can be encoded as parametrized logic programs. They are devised to boost the traditional neural learning by combining it with the interpretability and expressive power of relational logic. In their framework, clauses in first-order logic, either hand-crafted by domain experts or learned using inductive logic programming, are used to describe the structure of the task at hand. Then, several different neural networks are constructed, each corresponding to a training or testing example. The weights are shared among the networks and trained using the standard stochastic gradient descent algorithm.

LRRNs draw inspiration from the field of inductive logic programming, in a sense that they use rules and facts to define a problem. In practice, a user of a LRNN framework first creates a template, which is a sequence of clauses (rules) written in the first-order logic extended with numeric parameters.

Loosely speaking, we can say that these templates define the architecture of a model. The user then provides a set of input examples that are also viewed as a set of weighted clauses (facts). These inputs analogical to standard training samples to any deep learning model, with the exception that in LRNNs, they are not *inputted* to the model, but rather merged with the set of rules. The resulting set of logical representations is then unfolded into a *grounded* model by obtaining all the valid groundings of the set of rules in the template - a step that is achieved using an inference engine such as Prolog. This grounded logical model is then mapped to a neural model which can be trained using a stochastic gradient descent algorithm, optimizing the parameters that were associated with the rules during the creation of a template.

2.4 Generalization and robustness of deep learning models

Contemporary deep learning models have undergone a series of developments and successes, however, they still suffer from a number of shortcomings. Garnello and Shanahan [15] emphasize the following main drawbacks:

1. Data inefficiency - contemporary neural networks require large amounts of data to be successful. For example, the training process of large language models that emerged in recent past, such as GPT-2 [16], required enormous volumes of data and computational resources.
2. Poor generalisation - today's neural networks are very prone to data distribution shifts - they usually perform very poor when exposed to data that come from distribution different to what they have been trained on. This is where humans excel - we are able to re-use the intelligence and expertise acquired on some task and transfer it onto challenges that we have not seen before. In contrast, even small and invisible changes to the inputs can significantly derail the predictions of a deep neural model.
3. Lack of interpretability - the computations and reasoning steps performed at different layers generally lack human-interpretable semantics. This *black-box* nature of today's deep neural networks have become one of the largest obstacles in their wide-scale adoption in various mission-critical applications, such as medicine.

Later in this work, we will try to address primarily the second mentioned issue - poor generalization. The ability of NLU models to generalize systematically and robustly is being questioned by many researchers in past years.

The research has shown that large language models tend to exploit spurious correlations and other shortcuts in the data and thus exhibit poor robustness and systematic generalization capabilities [17]. It appears that many deep networks capture the wrong patterns in data and fail to understand the true content. Oftentimes, this also renders the deep learning models very exploitable and susceptible to adversarial attacks.

2.5 Graph neural networks and NLP

For some years, deep learning has been a dominant approach for solving various tasks in the NLP domain. In these tasks, we often need to work with data in the form of text. Since text is a sequence of tokens, the natural choice when it comes to deep learning are models that work well with sequential inputs, such as recurrent neural networks or transformers. However, recently there has been an increasing interest in applying GNNs to a large number of problems from the NLP domain, including text classification and relation extraction, or generative tasks like machine translation and question answering. The intersection of these two research areas - NLP and deep relational learning - has triggered interesting developments on both sides. In deep relational learning, special variants of GNNs are being developed to better accommodate the nature of textual inputs. On the other hand, the ability to process graphs as an input has allowed to incorporate task-specific knowledge and augment the original textual data in many ways, for example by building dependency or constituency parse trees for the input sentences. In the following sections, we will briefly introduce the most common ways of creating a graph structure from text.

2.5.1 Text to graph construction

The most simple way of representing a natural language is a bag of words or one-hot encoded vector. With these approaches, however, one completely loses the positional information about individual tokens, since the same initial sequence could be sorted in any order and still produce the same vector. Therefore, these representations are rarely used as inputs to deep learning models. In deep learning, the text is typically represented as a sequence of words or, more generally, tokens. But there is also another alternative that can be used to encode text - graph structures. A variety of NLP problems can benefit from being represented as a graph. Compared to the previous two approaches, graph representations often allow for capturing richer information from the input text. Naturally, being able to construct a graph from text is necessary to be able to use GNNs on NLP problems. This challenge has been profoundly studied in past and many fundamentally different ways have been proposed to solve it. In this section, we will look at the most effective ones, with special focus on those that will be directly used later in this thesis.

Essentially, graph construction techniques can be classified into two main categories - static and dynamic.

Static construction methods usually construct the graph during the preprocessing phase. Typically, they do so by using existing relation parsing tools or other manually constructed static rules. Some results of static graph construction are dependency graphs, constituency graphs or knowledge graphs. For example in dependency graph construction, one first need to obtain the dependency parse tree of the input sentence. This is done using the dependency parsing technique which was already introduced in previous sections. The relations in the dependency tree are then extracted and edges in the graph are formed between the head and the dependent word of a relation. Additionally, it is a common practice to add also a second group of edges between pairs of successive words in order to preserve the positional information from the sentence. Another naive approach that is feasible only for relatively short texts is to connect all pairs of two words with an edge, forming a complete graph. One common disadvantage of these methods is that they require extensive domain expertise in order to construct an effective graph topology. Another problem is that the graph construction step is disconnected from the rest of the training and cannot be optimized with the rest of the architecture. It is therefore crucial to choose the optimal construction technique for the particular downstream task at hand.

Contrary to static construction, **dynamic construction** methods usually require only minimal manual human effort or domain expertise. Instead, they try to learn the graph structure dynamically, and this step is often optimized in an end-to-end fashion with the rest of the graph network. While a lot of the methods in this category are useful in practice, they are more suited to longer passages of text. Since we will be working on relatively short text samples, we will utilize mainly the static construction approaches in our experiments.

Chapter 3

Benchmark selection

In this chapter, we will provide some context for the problem that the rest of the thesis will be concerned with. First, we present an overview of relevant benchmarks that were considered to be used in the experimental part. We also explain the key factors that drove the final decision and provide a detailed description of the selected dataset.

3.1 Relevant benchmarks

In the past years, many publications concerned with the issue of machine reading comprehension have emerged. However, only a small subset of them deals with the ability of models to exhibit *true* reasoning and generalization capabilities. These were the focus of research while searching for a benchmark eligible to test the ability of a model to reason about deeper semantics and relations in text.

Below is a comprehensive list of candidate datasets that were considered, along with a short description of each of them.

- **bAbi** - Weston et al. presented a set of prerequisite tasks used to test the ability of models to answer questions via chaining facts, induction, deduction and more [18]. The questions in these task are generated synthetically using different templates. Further research pointed out that these tasks might not be as sufficient of a measure of AI-complete question answering as it was initially perceived [19].

- **CLUTRR** - Compositional Language Understanding and Text-based Relational Reasoning is a synthetically generated benchmark suite proposed to address some of the key issues related to the robustness and systematicity of NLU systems [20]. The goal is to predict kinship relations between pairs of entities whose relationships are indirectly described in the short generated stories.
- **BoolQ** - In their paper, Clark et al. have introduced a reading comprehension QA dataset containing only yes/no questions [21]. Answering these questions should require difficult entailment-like inferences and thus provide a new challenge for state-of-the-art models.
- **SNLI** - The Stanford Natural Language Inference corpus tries to address the lack of large-scale benchmarks in the natural language inference domain. It is a collection of over 500 thousand labeled sentence pairs that can be used as a testing ground for the ability of a model to develop semantic representations of text [22].
- **ReClor** - Reading Comprehension dataset requiring logical reasoning aims to enhance the logical reasoning ability of current models by providing a more challenging benchmark [23]. In this benchmark, the authors identify biased data points and based on their findings separate the dataset into an easy and hard part.
- **FewRel** - In parallel to previous works, in Few-Shot Relation Classification Dataset the authors test the reasoning ability of models on the relation classification task. The whole benchmark consists of 70K sentences and 100 relations derived from Wikipedia [24].
- **ROPES** - In Reasoning Over Paragraph Effects in Situations, the aim is to apply knowledge from reading paragraphs [25]. The models are challenged to utilize the implications of a passage of text in order to answer questions about novel situations.
- **QuaRel** - The QuaRel benchmark consists of over 2500 questions involving 19 different types of quantitative relationship. Many of the questions come from the field of science, economics and medicine [26].
- **LogiQA** - LogiQA is a comprehensive dataset intended to test the logical reasoning capabilities of models. It consists of questions written by experts with the aim to test human deductive reasoning [27].
- **Rule reasoning dataset** - In a paper titled Transformers as Soft Reasoners over Language [28], the authors explore the ability of various transformer-based models to draw conclusions from provided rules in a question-answering task. To do this, they introduce a synthetically

created dataset, where each sample contains a set of rules, a set of facts and a query, all transformed to natural language narratives.

Benchmark	Task	Size	Used models	Best accuracy
bAbi	QA	20 tasks, 2K questions per task	STM, LSTM	100% (STM)
CLUTRR	classification	unlimited (synthetic)	Bi-LSTM,RN, BERT, GAT, CPTs	95% (CTP)
BoolQ	binary classification (T/F)	16K questions	RNNs, BERT, T5 Transformer	91% (T5)
SNLI	NLI	570K pairs	Transformers	90% (DeBERTa)
ReClor	QA	6K questions	RoBERTa, XL-Net, transformers	78% (Knowledge model)
QuaRel	QA	2.7K questions	Bi-LSTM, Neural Semantic Parser	75% (Neural Semantic Parser)
LogiQA	QA	8K questions	Rule-based, BERT, RoBERTa	37% (RoBERTa)
Rule reasoning	binary classification (T/F)	unlimited (synthetic)	RoBERTa, BERT, LSTM	99% (RoBERTa)

Table 3.1: Overview of different properties of the examined benchmarks. QA stands for Question answering and NLI for natural language inference.

3.2 Selection process

All of the above benchmarks were carefully examined to select the most appropriate one on which our experiments will be performed. Ideally, we wanted to find benchmarks that contain rich relational information so that we can test the ability of deep learning models to make inferences using these relations. The following criteria have been considered during the selection, with decreasing level of importance:

- level of difficulty of the relational inference that the benchmark requires
- overall size of the benchmark
- the best achieved performance on the benchmark so far

- type of the best-performing model

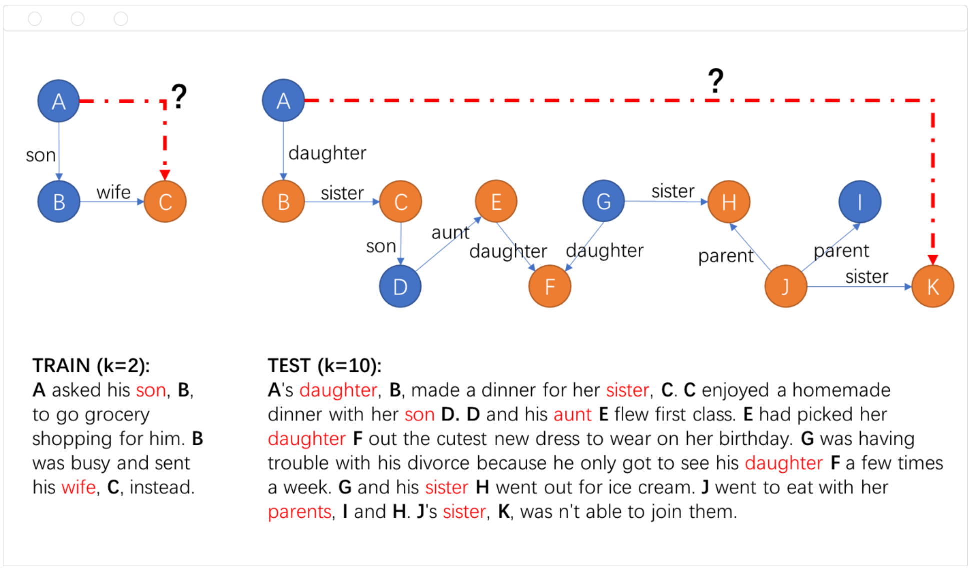
Following the above criteria, we selected one dataset from the above candidates - CLUTRR. It will be described more in detail in the following section.

3.3 CLUTRR dataset

Relation	Index
aunt	0
brother	1
brother in law	2
daughter	3
daughter in law	4
father	5
father in law	6
granddaughter	7
grandfather	8
grandmother	9
grandson	10
husband	11
mother	12
mother in law	13
nephew	14
niece	15
sister	16
sister in law	17
son	18
son in law	19
uncle	20
wife	21

Table 3.2: List of CLUTRR relations along with their corresponding indices.

After careful consideration, we decided to select the Compositional Language Understanding and Text-based Relational Reasoning (CLUTRR) benchmark suite. CLUTRR is a synthetically generated dataset in English language intended to test the systematic generalization and inductive reasoning capabilities of NLU models. Given a text story describing hypothetical family relationships of different entities, the goal is to classify the kinship relation of two people whose relationship is not explicitly mentioned in the story. It is



TRAIN (k=2):
A asked his son, **B**, to go grocery shopping for him. **B** was busy and sent his wife, **C**, instead.

TEST (k=10):
A's daughter, **B**, made a dinner for her sister, **C**. **C** enjoyed a homemade dinner with her son **D. D** and his aunt **E** flew first class. **E** had picked her daughter **F** out the cutest new dress to wear on her birthday. **G** was having trouble with his divorce because he only got to see his daughter **F** a few times a week. **G** and his sister **H** went out for ice cream. **J** went to eat with her parents, **I** and **H**. **J**'s sister, **K**, was n't able to join them.

Figure 3.1: Example of CLUTRR train and test instances. Train instance (left) is of length $k = 2$ and test (right) is of length $k = 10$. Image was taken from [29].

thus an R-way supervised classification task, where R is the size of the set of known relations, such as *parent* or *grandchild*. To predict the correct relation, the model must perform reasoning over the implicit knowledge graph that represents the underlying story. Each sample in the CLUTRR dataset consists of a few sentences (story) and a query pair (target). As an illustration, given the sentences *Marry is John's sister* and *John is Monica's father* and query *(Marry, Monica)*, the model should correctly infer the relation as *aunt*, as Marry would be Monica's aunt in this example. In total, there are $R = 22$ relations that will serve as labels in our classification task. Their names, along with the corresponding indices that will be used to refer to the them from now, are depicted in Table 3.2.

This benchmark is also devised to test the systematic generalization and inductive reasoning capability of machine learning models. It achieves this by including in the test set stories that contain previously unseen combinations of kinship relations. Naturally, the model has to induce the logical regularities that govern the kinship relations (e.g. the parent of a sibling is a parent), and learn to compose these induced rules. The difficulty of this task is further augmented by training the model only on samples with certain number of necessary reasoning steps - k_{train} - and testing it on samples that require more reasoning steps $k_{test} > k_{train}$. In other words, k represents the number of reasoning steps that the model must make in order to be able to answer the target query. This number corresponds to the number of edges in the kinship graph of the input story that it has to traverse in order to induce the true relationship of the two entities. Furthermore, CLUTRR can also test the

Dataset file	Reasoning steps	Number of samples	Target relations
1.2,1.3,1.4_train.csv	2, 3, 4	15083	None
1.2_test.csv	2	38	7, 10
1.3_test.csv	3	107	1, 5, 7, 12, 13, 16, 18
1.4_test.csv	4	77	1, 3, 5, 6, 7, 8, 9, 10, 12, 13, 16, 18
1.5_test.csv	5	185	0, 1, 3, 4, 5, 7, 8, 9, 10, 12, 13, 14, 15, 16, 18, 19, 20
1.6_test.csv	6	105	0, 1, 3, 4, 5, 7, 8, 9, 10, 12, 14, 15, 16, 19, 20
1.7_test.csv	7	155	0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 18, 20
1.8_test.csv	8	135	0, 1, 3, 5, 7, 8, 9, 10, 12, 14, 15, 16, 18, 20
1.9_test.csv	9	124	0, 1, 3, 5, 7, 8, 9, 10, 12, 14, 15, 16, 18, 20
1.10_test.csv	10	122	0, 1, 3, 5, 7, 8, 9, 10, 14, 15, 16, 18, 20

Table 3.3: Overview of different properties of the train and a set of test CLUTRR datasets. The last column contains the indices of all the target relations (labels) contained in the test set.

robustness of different models by including noisy or unrelated relationships in the generated story. In this case, the difficulty of the task naturally increases, as the model must select the relationships that are relevant to the given query and disregard the rest.

Since CLUTRR is a synthetic dataset, one could in theory generate datasets with samples of arbitrary length by passing different hyper-parameters to the script developed by Sinha et al. [20]. In our experiments, we will work with a set of already generated datasets, which consists of one dataset used to train the model and 9 different test datasets to evaluate the model's performance. The different properties of the used datasets are summarized in Table 3.3.

In summary, we will train our models on dataset with reasoning depth $k \in \{2, 3, 4\}$ and test on multiple datasets with their reasoning depths in the range $k \in [4, 10]$. The graph structure of one such train and test instance is illustrated on Figure 3.1. This setup will allow us to evaluate not only the learning capacity of the models, but also their generalization abilities and robustness.

The highly relational nature of this dataset, together with its other properties, makes it suitable for the forthcoming experimental part.

Chapter 4

Experiments

The following chapter will be devoted to the experimental part of the thesis. We begin this chapter by briefly summarizing the problem that we will be solving. Then, we proceed to describe the programming language and accompanying tools and libraries that were used to carry out the experiments. Next, we describe the proposed data preprocessing methods and deep learning models with special focus on their adaptations to our task. Finally, we conclude the chapter by summarizing and discussing the empirical results that were achieved.

4.1 Problem definition

The CLUTRR dataset was already introduced in detail in Chapter 3. We will be using it to perform a single-label multi-class supervised classification task using different neural architectures from the deep learning spectrum. In multi-class classification task, the goal is to classify given samples into one of R predefined classes. A special kind of multi-class classification is binary classification, where the number of classes R is equal to two. In the following experiments, we will be classifying the short stories into one of $R = 22$ classes according to the relation of the two queried entities. The goal is to train machine learning models to correctly predict the class based on the given input features. The ability of a model to succeed in this task is usually measured using the accuracy metric, characterized as a ratio of the number of correctly predicted samples to the total number of samples in the given dataset.

4.2 Languages and tools

The whole code base is written in the Python programming language, which is a standard choice among many machine learning practitioners, mainly due to its relative simplicity and a wide range of libraries and frameworks available. To carry out most of the experiments, we used the Pytorch framework and its Pytorch Geometric (PyG) extension.

Pytorch is an open-source machine learning framework written in Python with core parts written in Cython (language for writing Python extensions in C programming language). It is used by engineers from all over the world to build end-to-end machine learning pipelines, from prototyping to actual production deployment. It offers many excellent materials and tutorials and a large community willing to help with any inquiries. It also has a well-developed ecosystem of other tools and extensions, which made it the framework of choice for the experimental part of our work. Pytorch Geometric is a library built on top of Pytorch that allows to easily transform unstructured data to graphs and use them to train various kinds of graph neural networks.

We also utilized the Hugging Face transformers library, which provided us with a programming interface to easily download and use state-of-the-art pre-trained transformer models.

For natural language related tasks, such as tokenization, normalization, dependency parsing, and relation extraction, we used the Natural Language Toolkit (NLTK), spacy and Stanford Open Information Extraction (OpenIE) libraries.

All the experiments, except the one using BERT transformer, were conducted in a local development environment and the models were trained using CPU only. Since most of the neural architectures used throughout the experiments were fairly lightweight as to the number of layers and their weights, the computational power of CPU was sufficient to handle the training process in reasonable time and no GPU support was necessary. The only exception - training BERT transformer - has shown to be very computationally intensive, mainly due to the amount of hidden layers and parameters that had to be fine-tuned. In this experiment, we therefore resorted to using a Google Colab notebook and fine-tuning the model in a remote environment with access to the GPUs provided free of charge by Google.

4.3 Proposed graph construction methods

The most common approaches to text-to-graph conversion were already presented in Chapter 2. In this work, we developed three different graph construction methods that will be described more in detail below. The input to each of the methods is a CLUTRR instance consisting of a $(story, target)$ pair and the output is a tuple of size two. The first entry in the output tuple is a PyG Data object that holds information about the created graph corresponding to the input story. The second entry is an encoded query pair (e_1, e_2) where e_1 and e_2 are integer indices used to identify which nodes in the graph correspond to the two entities from the query.

In addition to the methods listed below, we also experimented with one dynamic graph construction method - using attention-based edge selection on complete graph, where the graph construction step would be optimized jointly with the rest of the network. However, this experiment was not successful due to implementation obstacles that were not overcome, so we decided not to include it here for the sake of brevity.

Complete graph. The first and simplest way to construct a graph from an input story is to create a complete graph on all of its words. In this approach, we consider each word as a node in the graph and connect all possible pairs of nodes with a unique edge. Undoubtedly, this method is not feasible for larger passages of texts, since it produces graphs with quadratic number of edges. However, stories in CLUTRR are relatively short and thus we can utilize this approach as a very simple baseline for the more elaborate methods that follow. We also managed to reduce the overall graph size by removing stop-words from the input sentence. A list of most common English stop-words was imported from the NLTK library.

To make this method a bit more robust, we embedded the words to 300-dimensional vectors using pre-trained Glove embeddings that were used as node features in the graph. Alternatively, we could use a trainable embedding layer to learn the embeddings jointly with the model parameters, but we decided to take advantage of the fact that the nodes correspond directly to words for which we already have existing embeddings. Edge features were not used in this part. We utilized the spacy python library to handle tokenization of the raw input stories, as it produces much more accurate results than naively splitting the input string by spaces. This library was also used to normalize the words - for example transforming the plural forms to singular, transforming verbs to their infinitive forms and handling other syntactical specifics of English.

Dependency graph. The second proposed construction method uses dependency parsing introduced in Section 2.1.2 to parse the input story into its dependency tree. It uses the dependency parser provided by the spacy library, which is based on the non-monotonic arc-eager transition-system described by Honnibal and et al. [30]. We refer the reader to the original paper in case he wants to learn more details about this algorithm. With the help of the library, we constructed a graph where the nodes are represented by token IDs and there is a directed edge, annotated with the type of dependency relation, between all *(head, dependent)* pairs from the output of the dependency parser.

Kinship graph. The most advanced method that will prove to reflect the nature of the dataset the best is inspired by the CLUTRR generation process. During the dataset generation, developers of CLUTRR first generate triplets or facts that always contain two entities and the relationship between them. Only then they paraphrase these facts into a textual narrative using different natural language templates developed by Amazon Mechanical Turk crowdworkers. As an illustration, we will use the following input story: *Ashley's daughter, Lillian, asked her mom to read her a story. Nicholas's sister Lillian asked him for some help planting her garden.*. The original triplets for this sentence would look as follows:

(Ashley, daughter, Lillian), (Nicholas, sister, Lilian)

Essentially, we want to reverse-engineer this process to obtain the triplets from the text of the story and then use them to construct a kinship graph, where the nodes will correspond to entities in text and the directed edges, annotated with the type of relation, will connect the entities that are on the two sides of the original relation. It is worth noting that the triplets from which the stories were generated are included as part of all CLUTRR datasets for the users' convenience. Nevertheless, using the original triplets directly could negatively affect the objectivity of the results when comparing the performance of models, since it could benefit graph models that use this special form of input at the expense of others that work directly on the raw text. Therefore, we deemed necessary to ensure that it was possible to reconstruct these triplets from the original story. We managed to accomplish this adapting an already existing script found in a GitHub repository [1]. This script uses an annotator module implemented in the OpenIE library, which extracts open-domain relation triples that consist of a subject, a relation, and an object of the relation. Using this approach combined with some heuristics, it was possible to successfully reconstruct the original ground-truth triplets and thus their use in the forthcoming experiments is justified.

[1]https://github.com/Wesley12138/clutrr-baselines

4.4 Proposed model architectures

This section provides the reader with an enumeration of all the different models that were proposed to solve the classification task. It explains in detail how the architectures were adapted to be used on the CLUTRR dataset, taking into account the input pre-processing and all the modifications, improvements and specific parameters that were introduced to accommodate the model to our problem. The theory behind the models and details of their inner architecture were already introduced in Chapter 2, so we will only focus on the parts specific to the experiments here.

The actual experiments differ along two dimensions - the input pre-processing method and the actual model architecture that was used. We devised multiple different variations of input pre-processing, using various NLP tools and graph construction methods, most of which are relevant only for a certain subset of the proposed models. For example, graph-based architectures expect the input to be a graph, while the sequence-based architectures expect a sequence of encoded tokens. Therefore, we divide the models into three parts roughly corresponding to the input they can process - text-based models, sequence-based models and graph-based models.

4.4.1 Text-based models

The text-based models process the entire input stories as a raw text. The texts are then tokenized and encoded to their numeric representations suitable for the model. In this part, we evaluated two different variants of a feed-forward neural network and a BERT transformer.

Basic feed-forward network

In the first variant, we constructed a simple feed-forward module with one input layer followed by a ReLU activation and one hidden layer with 32 neurons. We used a very simple approach to construct the numeric input for this network - we encoded the input story as a *bag of relations* contained in the story. This technique, commonly referred to in NLP as *bag of tokens*, builds a representation of the input sequences in two steps:

1. First, it builds a vocabulary of size N from all known tokens and creates a mapping to their unique IDs. In our case, this vocabulary is not formed from all words, but limited only to all possible 22 relation words that can occur in a story.
2. Then, it constructs a vector $\boldsymbol{v}^s \in \mathbb{R}^N$ for each story s, where v_i^s is equal to the number of occurrences of the token with ID i in s.

The *bag of relations* vectors constructed for each sample using the described approach are then used as an input for the network. The main drawback of this method is that it does not take into account the relative positional information of the individual relations in the story. Therefore, we do not expect it to perform particularly well - its purpose is rather to serve as a baseline for other, more advanced architectures introduced in the following sections.

In the following text, we will refer to the this experiment using the notation *text-ff-base*.

Feed-forward network with embeddings and positional encoding

In the second variation, we tried to partially alleviate the lack of positional information in the previous experiment. To do this, we altered the input pre-processing method and introduced two special modules on top of the layers described in the previous section:

- embedding layer - works as a *lookup table* that stores unique learnable embeddings of a fixed size for all tokens in the vocabulary.
- positional encoding layer - injects information about the relative positions of the tokens in the input. This concept is adapted from the transformer architecture and was explained more in detail in Section 2.2.3.

We set the same number of dimensions $d = 40$ for both the embeddings and positional encoding. The ordered sequences of tokens IDs corresponding to relation words are passed to the embedding layer. The output embeddings are then summed with the respective positional embeddings to form the final input vector for the subsequent feed-forward layers.

We will assign the notation *text-ff-pos* to this experiment.

DistilBERT Transformer

DistilBERT is a special version of the well-known BERT transformer. Both BERT and DistilBERT are special cases of encoder-only transformer architecture that was introduced in Section 2.2.3. We opted for DistilBERT because it is a smaller, faster and more light-weight transformer model trained by distillation of the base BERT transformer. Its performance is very close to the performance of the much larger BERT model, but it has 40% less parameters and has proven to run 60% faster, which allowed us to iterate the training and hyper-parameter fine-tuning process much more efficiently. In this experiment, we will use the pre-trained version of the transformer, since training such a large model from scratch would require significant computational power and long training time. In practice, it is usually sufficient to attach just few additional *head* layers that transform the output of the pre-trained encoder module to the desired dimension.

In this experiment, we will be optimizing both, the DistilBERT's pre-trained parameters and two additional hidden layers that were used as the classification head to transform the 768-dimensional output of the encoder to the desired dimension equal to the number of CLUTRR classes. The custom transformer head uses one extra hidden layer of size 512 before the final output layer. We will use two different values of learning rate - $lr_1 = 1 * 10^{-3}$ to optimize the transformer head and $lr_2 = 1 * 10^{-5}$ to fine-tune the pre-trained BERT parameters. Since all BERT models come with its own specific tokenizer that is used during their training, we will use it to tokenize the raw input stories into tokens.

Furthermore, we also need a way to provide the model with information about the query pair for a given story. While this can be done in multiple ways, we decided to employ a special *[SEP]* token that BERT uses internally to separate different logical parts of the input sequence (for example, it is used to separate a question and answer pair in question answering tasks). Below is an example of one such input sequence for story *'Ashley's daughter, Lillian, asked her mom to read her a story. Nicholas's sister Lillian asked him for some help planting her garden.'* and query pair *(Ashley, Nicholas)* before encoding the tokens to their IDs and padding the sequence with special *[PAD]* tokens. Note that *[CLS]* and *[PAD]* are another special tokens that BERT uses internally to store different information.

[[CLS], ashley, nicholas, [SEP], ashley, ', s, daughter, , ', lillian, , ', asked, her, mom, to, read, her, a, story, ., nicholas, ', s, sister, lillian, asked, him, for, some, help, planting, her, garden, .]

We will further refer to this experiment using the notation *text-bert*.

4.4.2 Graph-based models

In this part, we will describe in detail the graph neural architectures that we designed to solve the CLUTRR prediction task.

Throughout these experiments, we made heavy use of the already mentioned PyG library that implements many different GNN layers. It also provides toolkit for transforming all kinds of unstructured data to graphs and the further manipulation of these graphs. A single graph in PyG is described by an instance of *Data* class that holds the list of nodes (more precisely, their indices), node and optional edge features, information about edge connectivity and target labels used for training. PyG also allows us to conveniently create batches of Data objects and feed them to different GNN modules.

Before we proceed to describe the actual architectures, there is one part of architecture that is shared among all graph-based models. While the models in this category differ in the way they process the input graphs during the message passing phase, the output of this phase is always still a graph. The node representations in this graph are updated as a result of progressively applying the message passing rule, however, at this point we still need a way to somehow include also the information about the actual query pair corresponding to this graph. Otherwise, the model would simply not know what question it is trying to answer and would not be able to learn anything useful. There are multiple different ways of incorporating the query into the prediction. In our graph-based experiments, we chose the following approach - first, we retrieve the updated representation of the nodes originally corresponding to the two query entities, resulting in two distinct vectors of size n. Then, we compute an aggregated representation of the whole graph by averaging the updated feature vectors of all its nodes. Finally, these three vectors are concatenated to form a vector of size $3 * n$, and this aggregated vector is passed through a learnable feed-forward layer whose output dimension is equal to the number of classes in the prediction problem. We will refer to these final aggregation step as *decoder layer*. The graph encoding part that actually distinguishes the models will be described in the following subsections.

We will append the following suffixes to the names of different graph-based experiments depending on which graph construction method they use - *-com* for complete, *-dep* for dependency *-kin* for kinship.

Graph convolutional network

In the beginning, we developed a baseline GCN which only considers node features and disregards the edge features. Due to the highly relational nature of the CLUTRR dataset, the information carried in edges of certain graphs might be important and its absence will prove to be disadvantageous. During the forward pass, we perform three message passing rounds that progressively transform the graph representation, where in each round, the inputs pass through two GCN layers with hidden size $n = 100$. We use ReLU activation between the two layers. After the message passing, we end up with an updated graph representation where the dimensionality of the node embeddings is equal to the hidden size n of the last GCN layer. If the input is one-dimensional, as it is the case with some graph construction methods, it additionally passes the input to a learnable embedding layer of 100 before performing the message passing.

This model will be evaluated with all three graph construction methods and the individual experiments will be titled as *graph-gcn-com*, *graph-gcn-dep*, and *graph-gcn-kin*.

Graph edge convolutional network

Next, we developed a more sophisticated variant of the GCN that takes into account also the edge features of the input graphs. It uses a special convolutional operator that was inspired by the one used in the experiments [2] conducted on CLUTRR by Minervini et al. in their recent paper [31]. The first step in the forward pass consists of two distinct embedding layers - the first projects the input nodes into dimension $n = 100$ and the second projects the input edges to a dimension $e = 20$. Similarly to the baseline GCN, the model then performs three rounds of message passing. In each round, following operations are applied in order:

1. dropout layer with probability $p = 0.6$
2. first convolutional layer of size n
3. ReLU activation
4. dropout layer with probability $p = 0.6$

[2] https://github.com/uclnlp/ctp/blob/master/ctp/geometric/gcn.py

5. second convolutional layer of size n

The dropout layers were used as a means of regularization and have proven to be very beneficial for the overall model's performance. After the message passing, we pass the graph to the *decoder* layer to obtain the final embeddings.

We evaluated the model using two different graph construction methods - dependency graph and kinship graph. The experiments will be denoted as *graph-egcn-dep* and *graph-egcn-kin*.

Graph edge attention network

Following the previous experiments, we developed another advanced variant of the generic graph neural network - graph attention network with edge embeddings. Again, the graph attention layers that are used during the three message passing steps are adapted from the already existing experiments [3]. This architecture is very similar to the previous edge GCN variant and it differs mainly in the exact form of the convolutional operator used. In this case, it uses the attention mechanism described in Section 2.2.3 that allows each node to gradually learn which of its neighbors are important and should have larger effect in the computation of the updated node representation. The steps in the message passing phase are very similar to the previous model:

1. dropout layer with probability $p = 0.6$
2. first graph attention layer of size n
3. ReLU activation
4. dropout layer with probability $p = 0.6$
5. second graph attention layer of size n

The experiments conducted using this architecture will be denoted as *graph-egat-dep* and *graph-egat-kin*.

[3] https://github.com/uclnlp/ctp/blob/master/ctp/geometric/gat.py

4.4.3 Sequence-based models

In this part we selected an approach that is very specific to the nature of the CLUTRR dataset and therefore expected to perform well. Unlike the previous part, where the input to the model contains the sequence of all words from the original story, here we employed a more sophistical approach and only considered words that are actually relevant to the classification task - we extracted only words corresponding to the 22 kinship relations from the text. To achieve this, we used the logical graph structure underlying the stories described in the previous section. Since the order of the relations is important in this case (inference using ordered edges *mother* - *sister* - *daughter* yields different result than *mother* - *daughter* - *sister*), we ensured that the relations are ordered according to the path from the first entity of the query edge to the second entity of the query edge. In some of the CLUTRR datasets, this is guaranteed as one of the properties of the dataset generation process. The sequence of relation tokens was then encoded to their unique IDs.

In our experiments, we used both the basic RNN and an LSTM modules implemented in the Pytorch library and evaluated them on the CLUTRR datasets. After experimenting with different dimensionalities of the hidden states, we selected value $d = 64$ for both the networks. We will refer to the basic recurrent network and the more advanced LSTM variant as *seq-rnn* and *seq-lstm*, respectively.

4.5 Training process

This section will be concerned with the remaining implementation details, mainly the model training and evaluation process.

In all cases, the models were trained and evaluated using 1 train and 9 test datasets described in chapter 3. Furthermore, all the models were trained in a batch-wise fashion, as it allows for better parallelization of the matrix multiplications that take place during forward and backward passes and therefore speeds up the overall training process.

The objective function that was optimized during the training is the cross entropy loss. Cross entropy first applies softmax function on the output vector of the model in order to get a probability distribution over the classes and then computes the final loss using negative log likelihood between the predicted

Category	Experiment	Batch size	Epochs	Learning rate
text-based	text-ff-base	100	64	0.001
text-based	text-ff-pos	60	64	0.001
text-based	text-bert	64	20	0.001
graph-based	graph-gcn-com	100	64	0.001
graph-based	graph-gcn-dep	100	64	0.001
graph-based	graph-gcn-kin	100	64	0.001
graph-based	graph-egcn-dep	100	40	0.001
graph-based	graph-egcn-kin	100	64	0.001
graph-based	graph-egat-dep	100	32	0.0015
graph-based	graph-egat-kin	100	64	0.001
sequence-based	seq-rnn	100	40	0.001
sequence-based	seq-lstm	100	64	0.001

Table 4.1: List of used hyper-parameter values for all the experiments.

and the actual probability distributions. The model parameter updates were performed using the Adaptive Moment Estimation (Adam) optimization algorithm. Adam is an extension to the standard stochastic gradient descent that was recently adopted as the primary choice by many deep learning and NLP practitioners. We also experimented with different optimization algorithms but they did not seem to yield any significant performance boost.

The hyper-parameters for all the experiments, namely the size of a batch, the number of epochs and the learning rate, were chosen using an initial grid search. The selected values have proven to work relatively well for most of the models, which is the reason why they are roughly the same across most of the experiments. The exact selected values are listed in Table 4.1.

In the evaluation part, we focused specifically on the accuracy metric that can be expressed as the ratio between the number of samples that the model has predicted correctly and the total number of samples. This is also the metric that will be reported for all the trained models in the results section.

4.6 Results and discussion

This section will summarize the empirical results and provide a short discussion, highlighting the strengths and weaknesses of individual models.

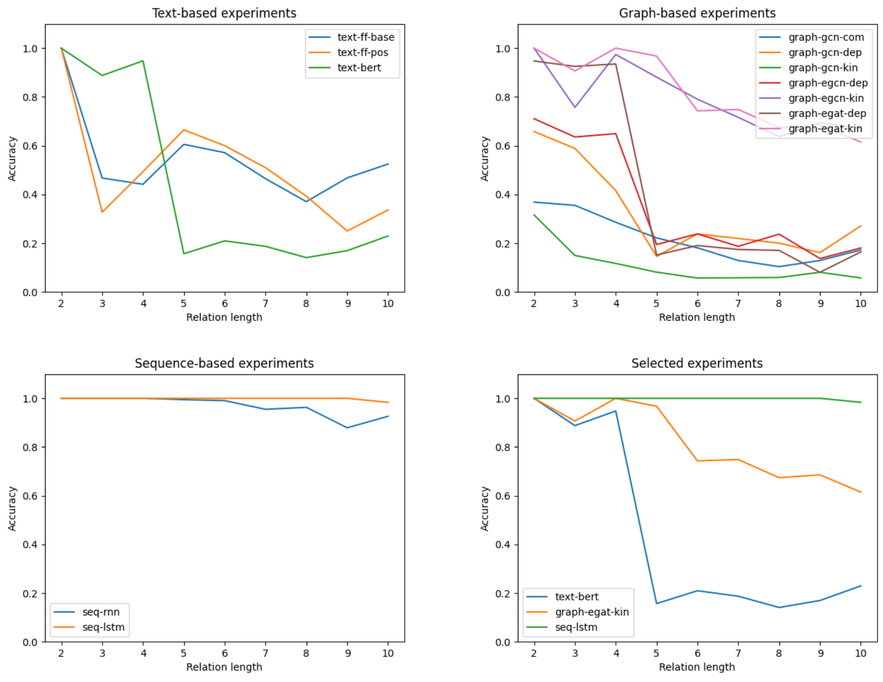


Figure 4.1: Plots of accuracy on all test sets for all model categories - text-based, graph-based and sequence-based. The values on the x axis correspond to the relation lengths in individual test datasets. The fourth plot on the bottom-right shows additional comparison of one selected experiment from each category.

Experiment	1.2,1.3,1.4	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10
text-ff-base	0.71	1.00	0.47	0.44	0.61	0.57	0.46	0.37	0.47	0.52
text-ff-pos	0.71	1.00	0.33	0.49	0.66	0.60	0.51	0.39	0.25	0.34
text-bert	1.00	1.00	0.89	0.95	0.16	0.21	0.19	0.14	0.17	0.23
graph-gcn-com	0.41	0.37	0.36	0.29	0.22	0.18	0.13	0.10	0.13	0.17
graph-gcn-dep	0.93	0.66	0.59	0.42	0.15	0.24	0.22	0.20	0.16	0.27
graph-gcn-kin	0.14	0.32	0.15	0.12	0.08	0.06	0.06	0.06	0.08	0.06
graph-egcn-dep	0.68	0.71	0.64	0.65	0.19	0.24	0.19	0.24	0.14	0.18
graph-egcn-kin	0.94	1.00	0.76	0.97	0.88	0.79	0.72	0.64	0.69	0.67
graph-egat-dep	0.86	0.95	0.93	0.94	0.15	0.19	0.17	0.17	0.08	0.16
graph-egat-kin	0.99	1.00	0.91	1.00	0.97	0.74	0.75	0.67	0.69	0.61
seq-rnn	1.00	1.00	1.00	1.00	0.99	0.99	0.95	0.96	0.88	0.93
seq-lstm	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98

Table 4.2: Final accuracy on the one train and nine test datasets achieved in the experiments. The dataset columns are annotated using the prefix in the dataset file name (1.x where x stands for the number of necessary reasoning steps for the given dataset).

In total, we conducted 12 different experiments on the CLUTRR dataset. The overall results are shown in multiple plots on Figure 4.1. Table 4.2 then shows these results in a tabular format, and contains also the final training accuracy of the models.

If we inspect these outputs, we can clearly observe a decreasing trend in the accuracy of most of the models as the number of necessary reasoning steps increases. Naturally, the more reasoning steps the model is required to perform on the test samples, the higher is the overall difficulty of the test set. In almost all cases, the models perform best on the easiest set, (*1.2_test.csv*), and worst on the most difficult one (*1.10_test.csv*). There are special cases when a certain model performs slightly better on some test set with reasoning depth k_1 than on some other test set with reasoning depth $k_2 > k_1$. These fluctuations might be attributed to the ability of a model to predict some types of relationships better than others, since not all datasets contain all relation classes.

The empirical results also provide an overview of how the individual categories of models (text-based, sequence-based, graph-based) compare against each other. We can observe that the best overall performance was achieved by the sequence-based models - recurrent neural network and its LSTM variant - that achieved nearly 100% on all test datasets. The sequence-based models clearly benefit from their ability to process an arbitrary sequence as an input. In our case, this allowed them to process the CLUTRR instances in their cleanest possible form - as sequences of the relation words. Since the ordered sequence of relation words unambiguously determines the target relationship, it was informative enough for the model to learn to generalize the rules learned on sequences of lengths $k \in \{2, 3, 4\}$ to longer sequences up to length $k = 10$. Arguably, this is a relatively easy task for a model with enough learnable parameters and given enough training time.

On the other hand, the text-based models that treat the input as raw text and do not have any inductive bias exhibited the lowest overall performance. We can observe that models that work on natural language stories have difficulty learning a robust mapping from the narratives to the underlying logical facts. While they work relatively well on test sets with reasoning steps $k \leq 4$, there is a significant drop in their accuracy on samples with previously unseen values of k. Specifically, it is interesting to study the sudden performance drop of the *text-bert* transformer model. The results of this experiment support the hypothesis from the beginning of this thesis, stating that the large pre-trained language models do not *really* learn to understand the content and are not capable of generalizing to previously unseen combinations in a systematic and robust way.

The same sudden decrease in performance was detected in some of the graph-based experiments, mainly *graph-gcn-com*, *graph-gcn-dep*, *graph-egcn-dep*, *graph-egat-dep*. However, in this case, the low performance of models is most likely caused by the two graph construction methods - dependency and complete graph construction - whose output might not be informative enough for the model to be able to identify the correct signal. On the other hand, the poor results in the *graph-gcn-kin* experiment are almost certainly caused by the inability of the baseline GCN architecture to consider edge features. Since graphs created using kinship graph construction store the important relational information in edges, without the edge features, the model simply does not have the necessary signal to learn anything useful.

In contrast, the other two experiments that use the kinship graph construction were more successful. This was expected, since unlike the previous two methods, the kinship graph very closely reflects the underlying nature of the CLUTRR dataset. The more advanced architectures from the deep relational learning domain that consider edge features - graph convolutional network and graph attention network - outperformed almost all the other graph and text-based models by a large margin. Their success can be ascribed mainly to the underlying relational graph structure to which they had access. The rich relational information stored in the kinship graphs of stories have proven to be very helpful for these advanced variants of GNN. It allowed them to not only to learn to induce the logical regularities that govern the kinship relations, but also to compose and generalize these rules to much more difficult stories with previously unseen combinations of relations. For instance, the accuracy of the edge GAT network dropped only by 39% (from 100% to 61%) between the easiest and the most difficult test dataset.

All in all, the empirical results highlight the gap between both recurrent and graph models that work on structured symbolic inputs and those that work on unstructured texts. While the recurrent neural networks that worked on sequences of relations undeniably outperformed all other models, the price was paid in the simplifying assumptions that were made while building the input for their architecture. Undoubtedly, the methods used to preprocess the input in these two experiments were highly specific to the CLUTRR dataset and could only hardly be generalized to real-world applications. On the other hand, the GNN-based models have exhibited lower, but still comparable reasoning capabilities without overly sacrificing their practical reusability. We can see a clear trade-off between the amount of inductive assumptions that the model makes about its data and its ability to generalize beyond the training scenarios.

Chapter 5

Conclusion

In this bachelor thesis, we have presented the state of affairs in the NLU discipline, including the problems that it is currently facing and the solutions that were proposed to address them. We have conducted a detailed survey of the relevant literature and identified over 10 existing benchmarks that test the relational reasoning ability of machine learning models. From this set, we have selected the most appropriate benchmark - CLUTRR - for the experimental part.

In the experimental part, we have devised 12 distinct experiments, distinguished by both the data preprocessing methods and the specific deep learning models that were used. We trained and evaluated the proposed models on a text classification task in 9 testing scenarios with increasing level of difficulty. Along the line, we proposed suitable inductive biases to address the highly relational nature of the dataset, which helped us to train models of increasing expressiveness and power that achieve state-of-the-art performance on part of the CLUTRR benchmark suite.

Finally, we compared the models with relational priors against the transformer architecture and demonstrated the disparity in their reasoning abilities. We found that large NLU language models, such as BERT, might indeed exhibit poor generalization and inductive reasoning capability when compared graph and recurrent neural networks that work directly on symbolic inputs.

5.1 Future work

At last, let us identify some future research directions.

Firstly, we could further study the interplay between different hyper-parameters used to train the models and fine-tune them to achieve even better results. Another research direction would be to develop more advanced dynamic graph construction methods and optimize the construction step jointly with the rest of the network. There is also space for a more detailed analysis of the reasons behind the poor generalization capabilities of large language models.

Naturally, another compelling next step would be to evaluate the successful models on different benchmarks from the NLU domain and test their ability to extrapolate beyond the CLUTRR dataset.

In spite of all the remarkable progress in deep learning made in recent years, there is still a lot of work to be made by those of us who aspire to build *truly* intelligent systems.

Appendix A

Bibliography

[1] Changchang Zeng et al. *A Survey on Machine Reading Comprehension: Tasks, Evaluation Metrics and Benchmark Datasets*. 2020. arXiv: 2006.11880 [cs.CL].

[2] Suriyadeepan Ramamoorthy. "Chatbots with Seq2Seq". In: (2016). URL: https://suriyadeepan.github.io/2016-06-28-easy-seq2seq/.

[3] Tomas Mikolov et al. *Distributed Representations of Words and Phrases and their Compositionality*. 2013. DOI: 10.48550/ARXIV.1310.4546. URL: https://arxiv.org/abs/1310.4546.

[4] Matthew E. Peters et al. *Deep contextualized word representations*. 2018. DOI: 10.48550/ARXIV.1802.05365. URL: https://arxiv.org/abs/1802.05365.

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

[6] Ashish Vaswani et al. *Attention Is All You Need*. 2017. DOI: 10.48550/ARXIV.1706.03762. URL: https://arxiv.org/abs/1706.03762.

[7] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2018. DOI: 10.48550/ARXIV.1810.04805. URL: https://arxiv.org/abs/1810.04805.

[8] Benjamin Sanchez-Lengeling et al. "A Gentle Introduction to Graph Neural Networks". In: *Distill* (2021). https://distill.pub/2021/gnn-intro. DOI: 10.23915/distill.00033.

[9] Ameya Daigavane, Balaraman Ravindran, and Gaurav Aggarwal. "Understanding Convolutions on Graphs". In: *Distill* (2021). https://distill.pub/2021/understanc gnns. DOI: 10.23915/distill.00032.

[10] Lukas Zahradnik. "Extending Graph Neural Networks with Relational Logic". In: (2021). URL: https://dspace.cvut.cz/handle/10467/97065.

[11] Thomas N Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *arXiv preprint arXiv:1609.02907* (2016).

[12] Joan Bruna et al. *Spectral Networks and Locally Connected Networks on Graphs*. 2013. DOI: 10.48550/ARXIV.1312.6203. URL: https://arxiv.org/abs/1312.6203.

[13] Petar Veličković et al. *Graph Attention Networks*. 2017. DOI: 10.48550/ARXIV.1710.10903. URL: https://arxiv.org/abs/1710.10903.

[14] Gustav Sourek et al. "Lifted Relational Neural Networks: Efficient Learning of Latent Relational Structures". In: (2018).

[15] Marta Garnelo and Murray Shanahan. "Reconciling deep learning with symbolic artificial intelligence: representing objects and relations". In: *Current Opinion in Behavioral Sciences* 29 (2019). Artificial Intelligence, pp. 17–23. ISSN: 2352-1546. DOI: https://doi.org/10.1016/j.cobeha.2018.12.010. URL: https://www.sciencedirect.com/science/article/pii/S2352154618301943.

[16] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". In: (2019).

[17] Suchin Gururangan et al. *Annotation Artifacts in Natural Language Inference Data*. 2018. arXiv: 1803.02324 [cs.CL].

[18] Jason Weston et al. *Towards AI-Complete Question Answering: A Set of Prerequisite Toy Tasks*. 2015. arXiv: 1502.05698 [cs.AI].

[19] Moontae Lee et al. *Reasoning in Vector Space: An Exploratory Study of Question Answering*. 2016. arXiv: 1511.06426 [cs.CL].

[20] Koustuv Sinha et al. *CLUTRR: A Diagnostic Benchmark for Inductive Reasoning from Text*. 2019. arXiv: 1908.06177 [cs.LG].

[21] Christopher Clark et al. *BoolQ: Exploring the Surprising Difficulty of Natural Yes/No Questions*. 2019. arXiv: 1905.10044 [cs.CL].

[22] Samuel R. Bowman et al. *A large annotated corpus for learning natural language inference*. 2015. arXiv: 1508.05326 [cs.CL].

[23] Weihao Yu et al. *ReClor: A Reading Comprehension Dataset Requiring Logical Reasoning*. 2020. arXiv: 2002.04326 [cs.CL].

[24] Xu Han et al. *FewRel: A Large-Scale Supervised Few-Shot Relation Classification Dataset with State-of-the-Art Evaluation*. 2018. arXiv: 1810.10147 [cs.LG].

[25] Kevin Lin et al. *Reasoning Over Paragraph Effects in Situations.* 2019. arXiv: `1908.05852 [cs.CL]`.

[26] Oyvind Tafjord et al. *QuaRel: A Dataset and Models for Answering Questions about Qualitative Relationships.* 2018. arXiv: `1811.08048 [cs.CL]`.

[27] Jian Liu et al. *LogiQA: A Challenge Dataset for Machine Reading Comprehension with Logical Reasoning.* 2020. arXiv: `2007.08124 [cs.CL]`.

[28] Peter Clark, Oyvind Tafjord, and Kyle Richardson. *Transformers as Soft Reasoners over Language.* 2020. arXiv: `2002.05867 [cs.CL]`.

[29] Wanshui Li and Pasquale Minervini. *Differentiable Reasoning over Long Stories – Assessing Systematic Generalisation in Neural Models.* 2022. DOI: `10.48550/ARXIV.2203.10620`. URL: `https://arxiv.org/abs/2203.10620`.

[30] Matthew Honnibal and Mark Johnson. "An Improved Non-monotonic Transition System for Dependency Parsing". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing.* Lisbon, Portugal: Association for Computational Linguistics, Sept. 2015, pp. 1373–1378. DOI: `10.18653/v1/D15-1162`. URL: `https://aclanthology.org/D15-1162`.

[31] Pasquale Minervini et al. *Learning Reasoning Strategies in End-to-End Differentiable Proving.* 2020. DOI: `10.48550/ARXIV.2007.06477`. URL: `https://arxiv.org/abs/2007.06477`.

Appendix B

Attachments

Name	Description
clutrr/	code related to the preprocessing of the CLUTRR dataset
experiments/	model training and evaluation of experiments
graph/	implementation of different graph construction methods
models/	implementation of different deep neural networks
nlp/	NLP-related utilities such as tokenization and word embeddings
notebooks/	a collection of notebooks mainly exported from Google Colab
outputs/	generated figures and plots used in the thesis
scripts/	a miscellaneous collection of scripts for different purposes
main.py	the entrypoint for running experiments
utils.py	miscellaneous utility functions

Table B.1: Thesis source code repository structure.

This thesis has one attachment - a compressed archive that contains the source code for all of the experiments. Table B.1 lists the directories and files in the root directory of the archive along with their short description. Dataset files and pretrained word embeddings were not included in submission due to their excessive size.