

University of Applied Science Ulm

Rose Hulman Institute of Technology

Faculty Electrical Engineering and Information Technology

Dual Degree Program Systems Engineering and Management - International

Masterthesis

Multi-hop Reasoning in Knowledge Graphs using Reinforcement Learning

First Advisor:	Prof. Dr.-Ing. Dirk Bank (THU)
Second Advisor:	Prof. Dr.-Ing. Dan Moore (RHIT)
Supervisor:	Dipl.-Ing. Tilo Himmelsbach (TUB)
	Prof. Dr.-Ing. Sebastian Möller (TUB)
Author:	Daniel Julian Patzer
Student ID.:	3128398
E-Mail:	dpatzer@mail.hs-ulm.de
Date:	22.05.2020

Version May 22, 2020

Declaration

I hereby declare that I wrote the master thesis independently and used no other aids than those cited. In each individual case, I have clearly identified the source of the passages that are taken word for word or paraphrased from other works.

Daniel Julian Patzer

Abstract

The demand for clinical decision support systems in medicine and self-diagnostic symptom checkers has substantially increased in recent years. They depend on large biomedical databases, such as knowledge graphs (KG), to derive inferences. KGs express medical concepts as connections between multiple nodes in a graph. Most biomedical KGs are incomplete and miss relationships between nodes.
This work employed a deep medical reasoning system to infer missing relationships from a biomedical KG by synthesizing existing information. A question answering (QA) framework is conducted to explicitly infer relationships that are not directly linked in the KG. An embedding-based model was trained to answer questions based on semantic similarities. We were able to achieve high accuracy scores using the statistic metrics Hits@k and MRR. Further investigations revealed limitations when facing more complex chains of reasoning and the lack of transparency in cases of misclassifications. Following the latest research, we were able to overcome these limitations by using a path-based, multi-hop reinforcement learning (RL) approach. We trained the RL agent and presented its ability to use the symbolic compositionality of relations in a qualitative analysis. Given a question, the agent 'walks' on the graph by choosing relations at each step. Thus, providing an explainable reasoning path. Such computational methods can help in a variety of downstream tasks, such as generate hypotheses of potential protein associations or predict explainable interactions in drug-symptom networks.

Contents

List of Figures

List of Tables

1. Introduction

Deep medical reasoning systems are automated tools to support medical diagnostic and are used by patients seeking information about their symptoms [7, 8], as well as by clinicians when faced with a difficult case or to avoid prematurely focusing on a small number of potential diagnoses [9]. These models have shown significant success in improving didactic practices [10] assisting with diagnosis, at times, even outperforming experienced doctors [11, 12, 11, 13].
Considerable effort has been put into building deep medical reasoning systems by decoding relevant information to drive their inference capabilities [14, 15]. Today, large amounts of biomedical information are available in digital but unstructured representations, such as text. Recent advances in natural language understanding and text mining allow processing text data and decode underlying semantics in a formal knowledge representation [16, 17, 18, 19]. An approach proven to be successful in information technology is to express knowledge in a graph-structured knowledge base, namely a Knowledge Graph (KG) [20]. It describes factual information in the form of relationships between entities, which enable machines to explore the data. These systems can benefit various important biomedical tasks, such as predicting potential drug indications based on drug-disease association graphs [21], detecting long non-coding RNA[1] (lncRNA) functions based on lncRNA–protein interaction networks [22], and assisting clinical decision making via disease-symptom graphs [23].
Whether these KGs are constructed automatically or by a group of medical experts, they cannot capture the underlying semantics in their entirety, they result incomplete. Research in automated reasoning has explored ways to predict new relationships in incomplete KGs by utilizing existing graph structure [5, 24, 25]. Neural graph embeddings are a promising unsupervised learning technique to reason in incomplete KGs. These models map the graph to a vector space and learn a true value scoring-function for any potential combination of facts. These models enable to go beyond look-up-style query answering and infer novel relationships between nodes. However, they can not grasp the symbolic compositionality of KG relations which limit their application in more complex deep medical reasoning tasks. An alternative, multi-hop approach synthesizes information from paths linking multiple nodes. For example, consider the following small biomedical KG in Figure 1.1. A fact, such as ⟨antibiotics, *affects*, cell_function⟩ is represented as two entities (nodes)

[1] Unfolding and decoding non-coding Ribonucleic acid (lncRNAs) are a growing focus of cancer genomics studies.

connected through a relationship (edge). A potential investigation, "Do antibiotics affect inorganic chemicals" can be analyzed by following multiple nodes on a reasoning path: antibiotics → *affects* → cell_function → *affects* → inorganic_chemical.

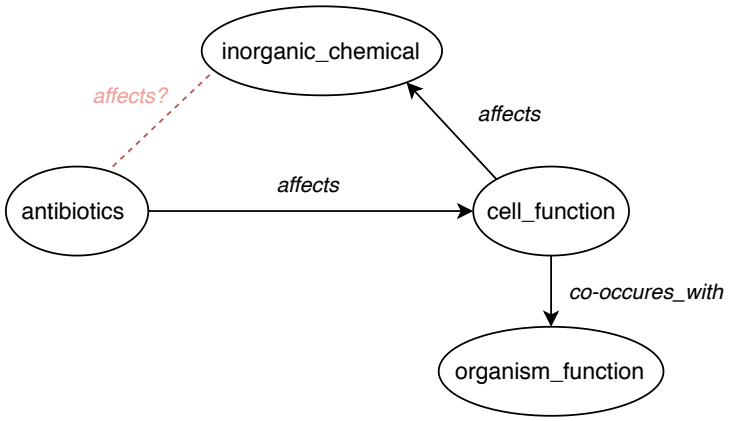

Figure 1.1.: Multi-hop reasoning in a small biomedical knowledge graph. Solid edges are observed and the dashed red edge is part of the query. The relation *affects* can be answered by traversing the graph via "logical" paths between entity 'antibiotics' and the corresponding answer.

Latest research models the path-based multi-hop reasoning approach as a Markov Decision Process (MDP) to leverage deep reinforcement learning (DRL) exploring the graph. The agent effectively searches for paths leading to the most promising answer [26, 27, 28, 29, 30]. The DRL agent starts from a given query node and learns individual probabilities to "walk" to the answer node by choosing a labeled edge at each step. The symbolic multi-hop approach has the immense advantage of being traceable.

1.1. Motivation

In the near future, Artificial intelligence (AI) in healthcare will, to an increasing degree, assist medical doctors in their decision making processes by emulating human cognition in the analysis of complicated medical data. Given the exponential power of computing, extended connectivity between humans and machines, accelerating speed of data production, aggregation, and new knowledge discovery, that results in massive data streams. This data can enable AI algorithms to derive inferences from the molecular and proteomic level to the organismic and population level, that is personalized for the patient. Deep Medial Reasoning Systems will be able to process patients' organismic data (e.g., genetics), wearable health technology data (e.g., heart rate blood pressure), and general health data by involving up-to-date biomedical research published in a specific field. The vision is to flawlessly integrate the latest research findings into the automated clinical decision process. This allows algorithms to evaluate potential conditions and suggest preventions before the human body is fully affected. The model extrapolates knowledge from the latest medical publications (i.e., articles, journal papers, or books) to steadily expand its inference capacities. However, we are not quite there yet!
A promising approach towards creating such systems is taking advantage of 'knowledge commons', which describe and interrelate a wide array of knowledge artifacts at multiple levels and allow inferences and decision support with emulated human analysis performance. An ontology is a set of controlled vocabularies that allow describing the meaning of data (its semantics) in a human and machine-readable way [31]. Especially a KG, the graphical representations of an ontology, is an increasingly popular tool to represent information extracted from biomedical research to downstream health assistant systems [32]. In recent years, we have seen an increasing number of open biomedical knowledge graphs. For example, Gene Ontology[2], UniPortKB[3], or UMLS[4]. Nevertheless, the quality of biomedical ontologies has hindered their applicability and subsequent adoption in real-world applications. In this work, we research latest advances in Deep Reinforcement Learning to effectively predict information in existing knowledge graphs, where the data quality is noisy and incomplete. This can be framed as an essential step towards general AI in biomedical decision making.

[2] `http://geneontology.org`
[3] `https://www.uniprot.org/help/uniprotkb`
[4] `https://www.nlm.nih.gov/research/umls/index.html`

1.2. Problem Statement and Research Aim

There are more than 28,000,000 million academic publications listed on PubMed[5] containing information on the intricate relationships of biological systems. On top of that, the number of publications is increasing exponentially with PubMed currently adding more than 1,000,000 citations per year [33]. Recent natural language processing (NLP) papers successfully applied large transformer-based architectures to automatically mine knowledge bases from text [34, 35, 19, 36, 37]. These constructed knowledge bases remain highly incomplete. Still, many valid facts can be inferred from the KG by synthesizing existing information. There is a high demand for such smart reasoning systems that can interactively retrieve biomedical information form KGs and predicting novel relations between, for example, protein, symptoms, or drugs, based on existing evidence [38].
The two state-of-the-art approaches in information technology we will research can be categorized into embedding-based[5] and path-based methods[39]. We comparatively evaluate the models' performances on deep medical link prediction, using a biomedical knowledge graph. Our primary research objective centers on the path-based approach because it implies potential advantages over neural embedding models. We seek to determine the following three aspects, addressed in distinct research questions:

- **Accuracy evaluation on link-prediction task**:
  Research question 1: What quantitative evaluation metric applies to the given problem statement, and how do both models comparably perform on biomedical link prediction?

- **Limitations**:
  Research question 2: What are the current limitations of neural graph embeddings, and DRL systems in terms of generalization performance?
  Can the models handle increasingly more data without performance dropping or hitting complexity limits?

- **Explainability**:
  Research question 3: Does the multi-hop deep reinforcement learning approach enhance the traceability of a predicted answer?

[5]PubMed is a platform which comprises more than 26 million citations for biomedical literature from MEDLINE, life science journals, and online books.

1.3. Thesis Structure

The Systes and Software Engineering Life Cycle Processes Model ISO/IEC/IEEE 15288:2015 [40] inspired the general research strategy and overall thesis structure as we followed the five generic stages:

- **Concept Definition**:
  We introduce the concept of semantic knowledge and research relevant open-source knowledge graph databases to operate a quantitative system evaluation study (Chapter 2).
  According to the research aims mentioned above, we define a set of requirements. Furthermore, we conduct a detailed literature review on state-of-the-art link prediction models and compare three fundamentally different methods (Chapter 3).

- **System Definition**:
  We choose two promising methods and perform a quantitative research study. Both systems are defined statistically through formal mathematical definitions (Chapter 3).

- **System Realization**:
  In Chapter 4, we provide brief introductions on relevant concepts in artificial intelligence and machine learning. This serves as a foundation for the subsequent system realization and integration (Chapter 5).

- **System Evaluation**: Both systems are comprehensively studied on the biomedical KG with systematic experiments and analyses. We introduce two evaluation protocols, which we use for benchmarking the models performance (Chapter 6).

- **System Validation**: In Chapter 7, we discussed the models performance for deep medical link predictions. Furthermore, we conducted a qualitative study and answered the research questions accordingly.

2. Semantic Knowledge

Semantic knowledge originates from the idea of framing things that are common knowledge, such as the names of colors, the sounds of letters, the capitals of countries, and other facts into formal symbolic representations. In AI, these concepts serve as a foundation to statistically model world knowledge. Providing machines with a logical understanding of the world. This idea has become an increasingly popular research direction towards cognition and human-level intelligence [41].
In this chapter we will cover the foundations of representing semantic knowlede in form of a graph i.e., a knowledge graph. Including discussions on data acquisition techniques to build graphical representations from text. We will then provide some examples on real-world knowledge-aware applications from industry. Followed by a detailed analysis of current graph data bases in biomedicine, which we will evaluate under constraints, such as, data quality, sparsity, density and size. Resulting in a knowledge graph data base that we will use to power our deep-medical reasoning system with explicit biomedical knowledge.

2.1. Knowledge Graph

A Knowledge graph represents knowledge in the form of interconnections between nodes and edges in a graph. It provides a structured representation of facts drawn from the relationships between entities. Where entities can be real-world objects or abstract concepts. Relationships represent the relation between them. The concept of representing knowledge in a graph initially gained traction in the web community by shifting from a 'web of documents' towards a semantic web. This new form of web content is structured and tagged to provide a meaningful representation for machines to explore the data [42]. In order to regulate the way publishing and interlinking data on the web in relational manner works, the World Wide Web Consortium (W3C)[1] released a collection of international standards, such as the Resource Description Framework (RDF). It defines the representation of a fact in the form of a triple $\langle head,$ `relation`, $tail\rangle$, (h,r,t), or $\langle subject,$ `predicate`, $object\rangle$, (s,p,o). Appreciations are (h,r,t) and (s,p,o), respectively. A combination of multiple facts stored in a list, is also known as a knowledge base (KB). The term of knowledge base is synonymous with knowledge base with a minor difference. A knowledge graph can be viewed as a graph when considering its graph structure. When it involves formal semantics, it can be taken as a knowledge base for interpretation and inference over facts.

[1] `https://www.w3.org`

2.1.1. Construction

Knowledge base construction is the process of populating a knowledge base (KB) with facts (or assertions) extracted from data. As a concrete example, one may want to build a medical knowledge base from the latest Covid-19 investigations to use the data for downstream natural language processing (NLP) tasks. As the vast amount of knowledge is available in text, consider the following fictitious research publication example:

> *"Corona is a virus called COVID-19. COVID-19 was first discovered in Wuhan, China and is now declared as a global Pandemic. COVID-19 belongs to the family of influenca, similar to SARS."*

Following the RDF standard, knowledge (facts) can be codified in $\langle subject, \texttt{predicate}, object \rangle$ (SPO) triples form text. There are multiple construction techniques on mining facts from data. These methods can be classified into four main groups:

- Curated approaches, where triples are populated manually by a closed group of medical experts. Currently most of the large biomedical knowledge bases, that will be mentioned later, are labeled and constructed by doctors or close groups of medical experts.
- Collaborative approaches, where triples are composed manually by their community members. Examples of collaborative approaches to construct knowledge graphs are Wikipedia [43].
- Automated semi-structured approaches construct triples automatically from semi-structured text (e.g., electronic health records) via learned rules, handcrafted rules, or regular expressions. Such methods led to large, highly accurate knowledge graphs such as DBpedia[2]. However, semi-structured text still covers only a small fraction of available information on the web, therefore current research focuses on processing raw text data.
- Automated unstructured knowledge base construction leverages latest advances in machine learning and natural language processing (NLP) to extract triples automatically from unstructured data (i.e., books, news, research papers, or clinical reports). Recent breakthroughs utilize a technique called transfer learning, which describes the process of training a model on a large-scale corpus of text. These pre-trained model are then used to operate on closely related downstream task (i.e., fact mining). A recently popular model architecture has been introduced by Google in 2017 i.e., Bidirectional Encoder Representations from Transformers (BERT) [35]. Their model reached state-of-the-art performance for NLP tasks on a variety of benchmark data sets and is now widely

[2]DBpedia is a knowledge base, linking all Wikipedia articles [44]

adopted for mining RDF triples from text. For example, SCIBERT is pre-trained on scientific language and is used for large-scale knowledge extraction and machine reading of scientific publications [34].

The previously given example of Covid-19 uses a curated approach to extract RDF triples manually form text. The derived collection of facts is listed in Table 2.1. This list is usually called a knowledge base.

Subject	Predicate	Object
$\langle Corona$	`is_a`	Virus$\rangle$
$\langle Corona$	`called`	COVID-19$\rangle$
$\langle COVID-19$	`discovered_in`	Wuhan$\rangle$
$\langle Wuhan$	`located_in`	China$\rangle$
$\langle COVID-19$	`is_a`	Pandemic$\rangle$
$\langle COVID-19$	`is_similar_to`	SARS$\rangle$
$\langle COVID-19$	`family_of`	Influenca$\rangle$

Table 2.1.: A data set of factual triples, also called knowledge base (KB).

In formal semantics, it can be taken as a knowledge base for interpretation and inference over facts. The example Table 2.1, is a formal collection of facts (knowledge base) and can be represented as a graph in Figure 2.1. Where subjects and objects are entities and predicates are directed edges or relationships between entities. Knowledge graphs are collections of facts that offer additional features i.e type hierarchies (Wuhan is located in China, China is a state, which has a geometric location) and type constraints (e.g., a city can only be located in a state).

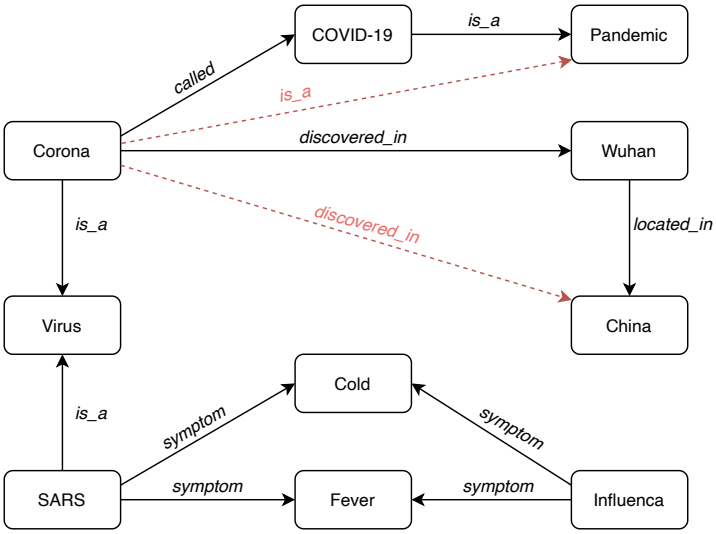

Figure 2.1.: Example knowledge graph constructed from factual SPO triples (black). Facts (red) are not directly observable in the graph, due to the incompleteness of the graph.

Whether knowledge bases were curated automatically or by a group of experts, they suffer from incompleteness [45, 41]. A fragmentary knowledge graph lacks relationships between facts to represent the original semantics, from the source data, adequately. However, missing facts can be inferred by existing ones. To illustrate this, we highlighted potential new inferences between nodes in Figure 2.1 with red dotted arrows. Our work primarily focuses on predicting certain new relationships based on existing evidence. We refer to these techniques as link prediction or knowledge base completion.

2.1.2. Knowledge-aware Applications

Knowledge graphs provide semantically structured information that machines can understand - this characteristic is key for building general-purpose artificial intelligence systems. As of today, KGs are already a driving force behind many technological advances. They power multiple "Big Data" applications in a variety of commercial and scientific domains. As already mentioned in the introduction, most large scale industry applications center around information networks with popular use-cases including Question Answer Systems (Chatbots, Symptom Checker[27, 23]), Dialog Systems [46, 47, 48], Semantic Search [49], and content-based Recommendation Engines [50]. The following provides discussions on four large scale industry cases that incorporate knowledge graphs as the essential component:

- In Recommendation Systems, knowledge graphs provide connectivity between

users and items. This provides richer and more complementary information on user-item interactions [50]. In the past, matrix factorization methods have mostly been used by e-commerce leaders like Amazon.com and Netflix for their recommendation engine[51]. Instead, recent research reveals the advantages of a richer semantic representation when using knowledge graphs. The German company Zalando, Europe's leading fashion retailer, uses a fashion KG. Their graph maps user intents with a broad range of articles. Imagine a user looking for *vegan* clothes. Not every article description might include the word `vegan`. The meaning of `vegan` is refusing to use any animal-based products. A knowledge graph can interpret the product data of wool, silk, or leather as unsuitable for vegans, and only offer products that are vegan by excluding those items with materials from animals.

- Social Graphs might be the largest real-world application of semantic graph-structured data representing social relations between entities (e.g., humans, institutions, or places). It interrelates individual information between likes, views, friends, places, and things you interact with as nodes and connects them through relationships (e.g., likes, interests, videos seen). Facebook popularized the concept in 2007. Today, their social graph is with approximately 500 million nodes, the largest social network on the internet [25].

- Semantic Search seeks to improve search accuracy by understanding a question's intent through contextual meaning. As a recent example, Causaly Inc., a London based, 2017 founded start-up, is currently building a large causality knowledge graph form scientific publications. Their engine extracts causal relationships between entities from raw text. Recently, this concept has proven to be very useful as it enables scientists to scan all latest publications on Corona with one search intent.

2.2. Data Acquisition

As of today, advanced research and commercial industry applications mostly center on social and information networks. Rather than comprehensively studied on biomedical networks under systematic experiments and analyses. This leaves room to research the tremendous potential of systems to perform information retrieval on biomedical knowledge graphs [23]. However, unlocking the true potential requires a detailed investigation of potential data sets. In most machine learning applications data quality has proven to be a bottleneck for training or convergence success. We evaluated five different biomedical knowledge bases. STRING database[3], GO[4],

[3] `http://string-db.org`
[4] `http://geneontology.org`

UMLS[5], CTD database[6], and Drugbank[7]. We followed recommendations for previous literature [27, 29] and searched for graphs with high density and rich interconnectivity of entities. We calculate the density ρ (Equ. 2.1[52]) for each graph in the following review section.

$$\rho = 2 \times \frac{|\mathcal{R}|}{|\mathcal{E}|^2} \tag{2.1}$$

2.2.1. Review

We considered the following datasets for training a deep-medical reasoning model:

- CTD knowledge graph: The Comparative Toxicogenomics Database[8] (CTD) is a premier public resource for literature-based, manually curated associations between chemicals, gene products, phenotypes, diseases, and environmental exposures. We extracted chemical-disease associations from database[53]. CTD offers two kinds of associations: curated (verified) and inferred. Finally, we obtained 92 813 edges between 12 765 nodes (9580 chemicals and 3185 diseases) in this graph.

- DrugBank knowledge graph: DrugBank is a comprehensive and freely accessible online database that combines detailed drug data (i.e., chemical, pharmacological and pharmaceutical) with comprehensive drug target (i.e., sequence, structure, and pathway) information. We collected verified DDIs from DrugBank [54] and obtained 242.027 DDIs between 2.191 drugs.

- STRING knowledge graph: The STRING database[9] aims to provide a critical assessment and integration of protein-protein interactions (PPI), including physical and functional associations. We restricted our research to only use Homo Sapiens PPIs from the database [55]. Each PPI is labeled with a confidence score. That indicates its possibility of being a true positive interaction. To reduce noise, we only collected PPI, whose confidence score is larger than 0.7, according to the guidelines of STRING database. We were able to obtain 359.776 interactions among 15.131 proteins.

- UMLS knowledge graph: The UMLS or universal medical language system network contains multiple concepts from molecular biology, genomics to med-

[5] `http://umlsks.nlm.nih.gov`
[6] `http://ctdbase.org/`
[7] `https://www.drugbank.ca`
[8] `http://ctdbase.org/`
[9] `http://string-db.org`

ical terminologies. These contain roughly $5,000$ facts including 104 entities and 95 relationships.

- GO ontology: The gene ontology GO incorporates knowledge regarding the functions of gene and gene products across all species. GO describes a complex biological phenomenon with a hierarchical structure and has little relationship types. Therefore, it is too sparse to accurately predict new inferences between entities. Following the previous investigations on graph structures, our proposed techniques are not recommended for graphs with a straight hierarchically structure [56]. Consequently, we did not do any further data analysis and did not include the statistics in Table 2.2.

We calculated the density values for each knowledge base, according to Equation 2.1. The datasets are sorted with increasing density levels in Table 2.2. The DrugBank dataset proved to have the highest density. However, after further investigations, we found that the Drug-Drug concepts stored in DrugBank are very specific and naturally hard to assess for a non-clinical audience. This would make a qualitative model evaluation unnecessarily tricky. We figured, interpretability especially during test-time, might be an essential factor in evaluating the models' prediction performance. For this reason, we are going to use the UMLS dataset, which provides a density of $1,7\%$. The interlinked concepts described in UMLS are in a broader biomedical context, enabling clearer understanding and enhanced explainability.

Dataset	Entities $\mathcal{E}$	Relationships $\mathcal{R}$	Density ρ
DrugBank	2.191	242.027	$10,08\%$
UMLS	104	95	$1,7\%$
STRING	15.131	359.776	$0,31\%$
CTD	12.765	92.813	$0,11\%$

Table 2.2.: Statistics of the experimental datasets, sorted by increasing sparsity level.

2.2.2. Universal Medical Language Systems

Therefore, we will use the Unified Medical Language System knowledge base for our experiments. The ontology combines various kinds of biomedical knowledge developed by the US National Library of Medicine and is free of charge for research and academic purposes. It unifies multiple sub-ontologies presented in Figure 2.2. This includes SNOMED-CT (Systematized Nomenclature of Medicine Clinical Terms), OMIM (Online Mendelian Inheritance in Man), ICD (International Classification of Diseases), and MeSH (Medical Subject Heading). The underlying data structure of UMLS can be divided into three parts; Metathesaurus, Semantic Network, and Specialist Lexicon. Metathesaurus is the major component as it is a repository of

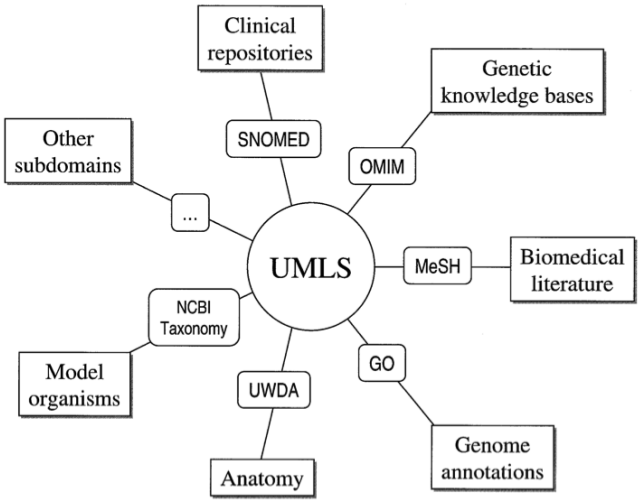

Figure 2.2.: The UMLS Metathesaurus not only serves as a link between the vocabularies, but also the subdomains they represent [1].

interrelated biomedical concepts. The two other knowledge sources in the UMLS are the Semantic Network, which provides high-level categories to structure every Metathesaurus concept and lexical resources. Each concept in the Metathesaurus is assigned to at least one category interlinked with each other through relationships. The Semantic Network is a catalog of 127 semantic types and 54 relationships. The majority of semantic types are organisms, chemicals, anatomical structures, events, biologic function, and physical objects. The links among semantic types define the structure of the network and highlight important relationships between groupings and concepts. The primary link between semantic types is the "isa" relation. Hence, the network also has 53 relationship types. For example, Figure 2.3 presents `malaria`, as a example entity and it provides all associated relationships, where the relationship type is constraint to `may_be_treated_by`. From the graph we retrieved `Malaria` → `may_be_treated_by` → `Sulfadiazine 500 MG Oral Tablet`.

Figure 2.3.: Entity `Malaria`(green) with relations `may_be_treated_by` (blue).

3. Related Work

Nowadays, networks are ubiquitous and many real-world applications need to mine the information within these networks for downstream NLP reasoning tasks, such as link predictions. These processes are formally known as knowledge representation learning (KRL), multi-relation learning, and statistical relational learning in literature. A widely used technique in KRL are knowledge graph embeddings, which use deep neural networks to map entities and relations into low-dimensional vector space while capturing their semantic meanings. Embedding based approaches have limitations in more complex reasoning tasks. An alternative solution for KG reasoning is to infer missing facts by synthesizing information from paths in the knowledge graph. In this chapter, we will analyze both approaches and present state-of-the-art models.
As an introduction to this chapter, we will study the current challenges KRL is facing and derive formal statistical definitions. These are serving as a foundation as we build upon state-of-the-art embedding-based and path-based systems.

3.1. Knowledge Representation Learning

Knowledge representation learning mainly focus on the process of learning knowledge graph embeddings, while keeping semantic similarities. Generally speaking, most concepts can be divided into two subsystems (Fig. 3.1). System one uses latent representations to decode entities and relations in low dimensional vector space. System two, the scoring component, measures the plausibility of facts, to adopt the semantic similarities between embeddings. Here, we can differentiate between similarity-based scoring functions and distance-based scoring functions, which will be further investigated in the following section.

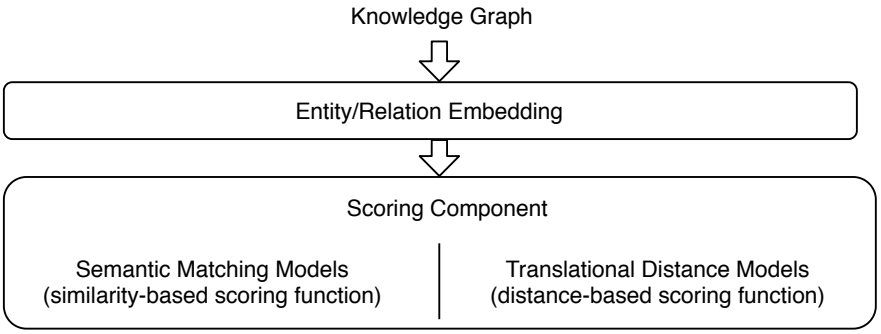

Figure 3.1.: The main components of knowledge representation learning

3.2. Challenges

The process of learning sophisticated network representations needs to be wrapped in a statistical framework. This defines concrete constraints which will be modeled and optimized throughout this chapter:

- **High non-linearity**: Underlying structure of real data is often highly non-linear. Thus, cannot be accurately approximated by linear manifolds. Designing a model to capture the highly non-linear structure is relatively difficult.
- **Structure-preserving**: A popular approach for modeling incomplete KGs is through embeddings. They map both entities and relations in the KG to a vector space and learn a truth value function for any potential triple in the KG. Quantitative network analysis through network embeddings requires to preserve the very complex network structure [57].
- **Sparsity**: Many real-world networks are often so sparse that only utilizing the minimal observed links is not enough to reach a satisfactory performance [28].
- **Scalability**: Most real-world networks contain millions of nodes and edges. For example, the Gene ontology contains roughly 500 million nodes [58]. Current computational capacities would run out of memory, when operating on the raw knowledge graph data directly. Consequently, there is a need for embedding methods that dimensionally reduce the network size by only extract relevant features. An essential issue when embedding knowledge graphs is how to deal with the vast amount of possible relationships (high-dimensionality), while efficiently incorporating the sparsity. Defining a scalable model can be challenging, especially when the model aims to preserve the network's global properties.

- **Incompleteness**: Knowledge graphs are intrinsically incomplete, which refers to missing links, between nodes, in the graph. This thesis follows the open-world assumption (OWA) to model incomplete knowledge graphs accurately. According to the OWA, a non-existing triple is treated as unknown, which means the corresponding relationship can be either be true or false.

3.3. Definitions

As we identified the general challenges, to model these knowledge graphs statistically requires some formal mathematical definitions. Following the RDF standard, let $\mathcal{E} = \{e_1, ..., e_n\}$ be the set of all entities (subjects and objects) and $\mathcal{R} = \{r_1, ..., r_m\}$ be the set of all relations (predicates) in the knowledge graph $\mathcal{G}$. Each potential triple $x_{ijk} = \{e_i, r_j, e_k\}$ over this set of entities and relations can be represented in a binary random variable $y_{ijk} = 0 \quad \vee \quad y_{i_{jk}} = 1$. The variable y_{ijk} is set to 1 indicating the existence of the triple in the KG, 0 otherwise. All existing interactions $\mathcal{E} \times \mathcal{R} \times \mathcal{E}$ will be stored in a third order adjacency tensor $\overline{\mathbf{Y}}$ (Fig. 3.2). We can

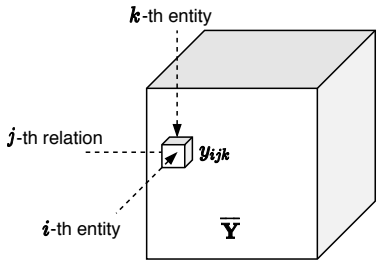

Figure 3.2.: Adjacency tensor representing existing interactions $\mathcal{E} \times \mathcal{R} \times \mathcal{E} \in \mathcal{G}$.

model patterns that repetitively occur in a knowledge graph (e.g., if Corona was first discovered in Wuhan, and Wuhan is located in China, then we can infer that Corona was first discovered in China), by assuming all y_{ijk} are conditionally independent given latent features. Where features can, for example, be the number of entities N_e and relations N_r together with additional parameters θ. This enables the model to predict the existence of a triple x_{ijk} via a score function $f(x_{ijk}; \theta)$. The function outputs a value representing the models' confidence in the existence of a certain triple, given the parameters θ. There are a variety of different methods defining $f(x_{ijk}; \theta)$. The following sections will introduce the commonly utilized modeling approaches. Here, the conditional independence assumptions allow the probabilistic

model to be written as follows:

$$\mathrm{P}(\overline{\mathbf{Y}} \mid \theta) = \prod_{i=1}^{n} \prod_{j=1}^{m} \prod_{k=1}^{n} \mathrm{BER}(y_{ijk} \mid \sigma(f(x_{ijk}; \theta))) \tag{3.1}$$

where σ is the sigmoid function and BER is the Bernoulli distribution. In addition to probabilistic models, which model the similarity among triples. We will also discuss models, which optimize f under other criteria. For instance, models that maximize the distance between existing and non-existing triples. We will refer to such models as distance score-based models [59].

3.4. Graph Embedding Methods

Graph embeddings or latent feature representations study the problem of embedding extensive information networks into low-dimensional vector spaces, which have proved extremely useful as feature inputs for a wide variety of prediction and graph analysis tasks [60, 61, 62, 63, 64]. This thesis will focus on neural link prediction, also known as knowledge graph completion. Besides that, classification and clustering are two strongly related tasks that are worth mentioning. A typical node classification predicts the most probable labels of nodes in a network. For example, in a social network we might want to predict users' interests, or in a protein-protein interaction network, we might be interested in predicting proteins' functional labels [65]. In contrast to link prediction, where we are interested in predicting the existence (or probability of correctness) for an interconnected pair of nodes in a network. Deep-medical link prediction is useful in a wide variety of domains; for instance, in genomics, it helps us discover novel interactions between genes or explore drug-drug interaction.
We will begin elaborating this concept by giving an intuitive example of typical link prediction task on our toy KG. Followed by an evaluation of different approaches and various neural link prediction models.
Consider the knowledge graph (Fig. 2.1), which was constructed in the previous chapter. New inferences on potential links (red dashed lines) between entities can be made by incorporating neighbor relationships (solid lines). This is a typical link prediction task and will be the primary research concern in this thesis. In the first step, we embed triples via latent features of entities: a possible explanation for the fact that *Corona* is a *Virus* is, the association to the family of influenza that spread rapidly amongst humans. This explanation uses latent features of entities (e.g., part of the influenza family) to explain observable facts. The concept of latent feature representation is not very intuitive because it is not directly observable in the data. Figure 3.3 provides a simple illustration of the concept. Here, we apply two features representing each entity. The latent feature representation of entity

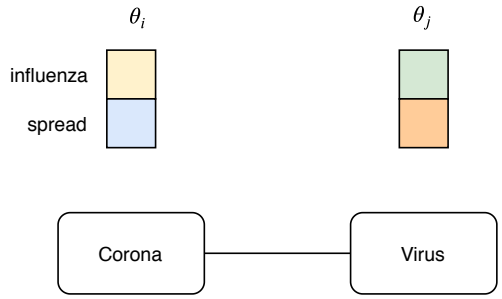

Figure 3.3.: Latent feature representation of entity *Corona* and *Virus* as a two dimensional feature vector θ.

Corona can be represented by the two dimensional vector θ_i. Note, in practice, the feature dimension is around 200 and, unlike this example, usually inferred by the latent feature models, which make them generally impossible to translate or interpret. The basic intuition of relational latent feature models is, that the relationships between entities can be interpreted from mathematical interactions of their latent features. A variety of possible ways to model these interactions are available and several methods to derive the existence of a relationship from them. Some of which we will explore in the second step. Until now, we exclusively extracted related features; this can be seen as an initial step (Fig. 3.1). In a second step we evaluate the existence of a relationship by calculating the scoring function between individual features $f(\theta_s, \theta_p, \theta_o)$. This produces a score per fact which represents the model confidence, whether a triple is valid. Internally, the model tries to maximize the score of $f(\theta_s, \theta_p, \theta_o)$ for any $x_{ijk} \in \mathcal{G}$ and minimize it for $x_{ijk} \notin \mathcal{G}$. In a neural link prediction model, both steps are comprised of a multi-layer neural network consisting of encoding layers and multiple scoring components. The central difference between individual models, present in the following, is in their choice of the scoring component. According to the scoring functions used, the models can be categorized into two groups: translational distance models and similarity-based model models.

3.4.1. Translational Distance Models

Translational distance models exploits distance-based scoring functions. The first model was introduced in 2013, namely TransE [59] and, since then, extended to multiple different versions; TransR [66], TransH [67], STransE [68], TransSparse [69], and TransD [70]. What these models have in common, they measures the plausibility of a fact as the distance between the two entities, usually after a translation carried out by the relation. Given a fact (s, p, o) the relation is interpreted as a translation vector θ_p, so that the embedded entities θ_s and θ_o can be connected by relation embedding θ_p with low error, i.e., $\theta_s + \theta_p \approx \theta_o$ when $y_{\{s,o,p\}} = 1$ holds. Equation 3.2

defines the scoring function as the negative distance between $\theta_s + \theta_p$ and θ_o.

$$f(s, p, o) = -\|\theta_s + \theta_p - \theta_o\|_{\frac{1}{2}} \tag{3.2}$$

3.4.2. Semantic Matching Models

RESCAL [2] is a widely adopted similarity-based model. The model executes a tensor factorisation of the adjancency tensor $\overline{\mathbf{Y}}$. RESCAL scores triples through

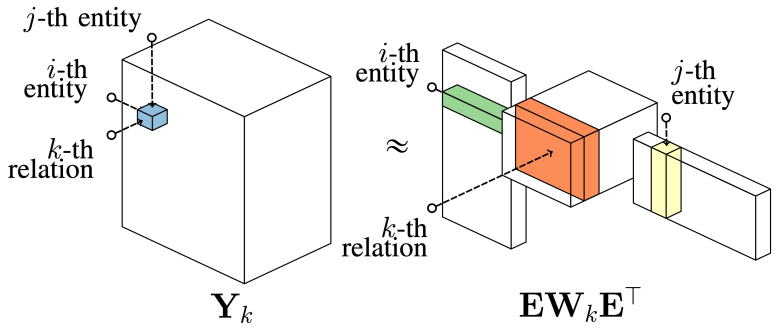

Figure 3.4.: RESCAL a three-way tensor factorisation model of the adjacency tensor $\overline{\mathbf{Y}}$ [2].

pairwise interactions of their feature vector of the entities θ_s and θ_o. W is a matrix associated with the relation.

$$f = \theta_s^{\mathrm{T}} W \theta_o \tag{3.3}$$

During training, we jointly learn the latent representations of entities and how the latent features interact for particular relation types by minimizing the log-loss using gradient-based methods, such as stochastic gradient descent. Since all parameters are learned jointly, these shared representations allow propagating information between triples via the latent representations of entities and the weights of relations. This permits the model to capture global dependencies in the data. Recent extensions to RESCAL are DISMULT [71], SimpleE [72], HolE [73], and TuckER [74]. In recent years neural networks for encoding semantic matchings have yielded remarkable predictive performance in recent studies [41]. Individually triples are converted into a vector obtained from concatenating subject, predicate, and object embedding, then fed into a multi-layer neural network. Extensions with CNNs architectures for learning deep expressive features result in models, such as ConvE [5] or ConvKG [75]. ConvE uses 2D convolution over embeddings and multiple layers of non-linear features and achieves state-of-the-art performance on common benchmark datasets

for knowledge graph link prediction. The embeddings θ_s and θ_p are reshaped, concatenated, and then fed in the convolution layer. Convolutional filters of $n \times n$ are used to output feature maps across different dimensional embedding entries. Features are optimized using the scoring following function:

$$f(s, p, o) = \sigma(\text{vec}(\sigma(\text{concat}(\overline{\theta_s}, \overline{\theta_p}) * \omega))\text{W})\theta_o \tag{3.4}$$

3.5. Path-based Reasoning

Learning embeddings of entities and relations using tensor factorization or neural methods to optimize a predefined scoring function between facts in the KG, achieves outstanding results on standard benchmark datasets [41]. However, these methods cannot capture chains of reasoning expressed by paths in the KG and cannot capture the full underlying semantics [27]. Path-based models address these problems by scoring facts based either on random walks over the KBs (path ranking) or by encoding entire paths in a vector space (path encoding).

3.5.1. Path Ranking

The Path-Ranking algorithm (PRA) [76], first proposed by Lao and Cohen in 2010, uses bounded-depth random walk with restarts to obtain paths. The key takeaway of PRA is the explicit usage of relational paths. That connect two entities as features to predict potential relations by using random walks of some predefined length. A relation path is a sequence of relations $\tau_r = (r_1, r_2, ..., r_n)$ that links two entities. Starting at entity $\mathcal{E}_i = e_0$, is defined as $P(\mathcal{E}_i \rightarrow \mathcal{E}_j|R)$. The probability of reaching $\mathcal{E}_j$ from $\mathcal{E}_i$ by a random walk that instantiates τ_r. A key idea for path ranking algorithm is that $P(\mathcal{E}_h \rightarrow \mathcal{E}_t|R)$ can be used as a feature for the task of prediction whether the triple $x_{ijk} = \{e_i, r_k, e_j\}$ is true. However, a deficiency of PRA is that it operates on symbols instead of embedding representations, which constraints good generalization performance to a small number of paths. Gardner et al. [77] extend PRA to include vector representations of verbs to overcome this limitation. Whereas, the verb representations are obtained during pre-training with PCA on a matrix of co-occurrence of verbs and subject-object tuples assembled from a large dependency-parsing construct. Later, these representations are used for clustering relations, thus avoiding an explosion of path features in prior PRA work while improving generalizations.

3.5.2. Path Encoding

While Gardner et al., (2013) [77] introduced vector representations for PRA, these representations are not trained end-to-end from task data but instead pre-trained on an external system. This implies that relation representations cannot be adapted

during training a KG. Neelakantan et al., (2015) [78] followed this research direction and instead came up with an approach, where embeddings of entire paths are encoded with RNNs. The RNNs take trainable relation representations as input, in the form of factual relations between two entities, connected through multiple steps in the graph. For the target relation, the RNN is trained to output path encodings, such that the dot product of the encoding and the relation can be maximized.
Das et al., (2016) [79] discovered the following three limitations of the work by Neelakantan et al., first, they did not implement parameter sharing in the RNNs design that encodes different paths for particular target relations. Second, there is no aggregation of information from multiple path encodings. There is no use of entity information along the path, as only relation representations are inputs to the RNN. Das et al. suggested the following solution to overcome the limitations. First, they used a single RNN, which allows the internal parameters to be shared across the network. Second, Das et al. trained an aggregation function over the encodings of multiple paths connecting two entities. Third, to obtain entity representations that are fed into the RNN alongside relation representations, they summed up learned vector representations.

3.5.3. RL-based Path Finding

Xiong et al., (2017) [29] argues that these methods still rely on first learning the PRA paths, which only operate in discrete space. Thus is does not scale. They were the first to propose Deep Reinforcement Learning (DRL) for multi-hop reasoning by formulating path-finding between entity pairs as sequential decision making, modeled as a Markov Decision Process (MDP). The policy-based RL agent learns to find a relation for each step to extend the reasoning paths via the interaction between the knowledge graph environment. The policy gradient algorithm (REINFORCE [80]) is used to training the RL agents. Furthermore, Xiong et al. developed a novel reward function to improve accuracy, path diversity, and path efficiency. It encodes states in the continuous space via a translational distance method and takes the relation space as its action space. However, they restrict their RL agent to simple tasks whereas predicting if a fact is true or not. Das et al., (2017) [27] built upon their work and introduced an RL agent designed to search the graph to find answer-proving paths efficiently. With source, query and current entity denoted as e_s, e_q and e_t , and query relation denoted as r_q, the MDP environment and policy networks of following current methods are summarized in Table 3.1.

Method	State s_t	Action a_t	Reward R	Policy Network
Multi-Hop [39]	$(e_t, (e_s, r_q))$	$\{(r', e') \vert (e_t, r', e') \in \mathcal{G}\}$	$R_b(s_t) + (1 - R_b(s_t)) f(e_s, r_q, e_T))$	$\texttt{LSTM}(\mathbf{h}_{t-1}, \mathbf{a}_{t-1})$
AttnPath [26]	$[\mathrm{e}_{t\perp}; \mathbf{e}_{s\perp} - \mathbf{e}_{t\perp}]$	$\{(r', e') \vert (e_t, r', e') \in \mathcal{G}\}$	$R_b(s_t) + (1 - R_b(s_t)) f(e_s, r_q, e_T))$ Diversity: $-\frac{1}{\vert F \vert} \sum_{i=1}^{\vert F \vert} \cos(\mathbf{p}, \mathbf{p}_i)$	$\texttt{LSTM}(\mathbf{h}_{t-1}, \mathbf{s}_t)$
MINERVA [27]	(e_t, e_s, r_q, e_q)	$\{(\mathrm{e}_t, r, v)\}$	$\mathbb{I}\{e_t = e_q\}$	$\texttt{LSTM}(\mathbf{h}_{t-1}, [\mathbf{a}_{t-1}; \mathbf{o_t}])$
DeepPath [29]	$(\mathbf{e}_t, \mathbf{e}_q - \mathbf{e}_t)$	$\{r\}$	Global 1if$e_t = e_q$else-1 Efficiency: $\frac{1}{length(p)}$ Diversity: $-\frac{1}{\vert F \vert} \sum_{i=1}^{\vert F \vert} \cos(\mathbf{p}, \mathbf{p}_i)$	Fully-connected NN

Table 3.1.: Comparison of reinforcement learning path-based link prediction models.

4. Background: Artificial Intelligence and Machine Learning

This chapter will present central ideas of artificial intelligence (AI) and machine learning (ML), which serves as a foundation for the subsequent model implementation chapter. Starting with a brief introduction of theoretical concepts in ML. That can be structured into, supervised learning, unsupervised learning, deep learning, reinforcement learning, and deep reinforcement learning. Relevant deep learning architectures are then discussed, such as fully-connected, convolutional, and recurrent. These architectures form the basic building blocks used in the both methods. After covering relevant deep learning architectures and important optimization algorithms, we present advanced deep reinforcement learning (DRL) techniques. DRL is a combination of reinforcement learning and deep learning. We will address the differences in value-based optimization and policy gradient and analyze the advantages of policy gradient. In particular the REINFORCE algorithm [80], which is used in the agent implementation.

4.1. Introduction

AI refers to the simulation of human intelligence in machines. Whereas machine learning describes the process in which a computer can learn on its own without being explicitly programmed. It is an application of AI that provides systems the ability to learn and improve from experience automatically. ML can be classified into three main areas: supervised-, unsupervised-, and reinforcement learning. Where in the first two methods, we are given access to some table of data where the row's index examples and the columns index features (or attributes) of the data. In unsupervised learning, the goal is to summarize the patterns. Each row has a list of attributes X, and the goal is to create an encoded lower-dimensional representation $\mathbf{E}$ for each instance that summarizes the salient information in X. Knowledge graph embedding methods, for example, are a unsupervised learning techniques. The model uses the raw KG data as input and outputs a lower-dimensional representation.
On the other hand, in supervised learning the general task is to predict new features. Here, the objective is to forecast Y from X such that on new data, you are accurately predicting Y. Supervised learning is the most common discipline of ML and involves classification and regression tasks. For example, a cancer classification system will be trained on millions of labeled images of different cancer types to replicate its experience and classify unseen images.
In recent years, especially deep learning (DL), which is another sub-discipline within

machine learning, is making major advances in solving problems that have resisted the best attempts of the artificial intelligence community for many years. It turned out to be precise at discovering intricate structures in high-dimensional data and is applicable to many domains of science and business. The success story of deep learning has three accelerators; *computational power/ infrastructure*, *vast amount of data*, and *algorithms with a tremendous community*, which nowadays scale exponentially. Deep learning has demonstrated significant improvements in computer vision[81], natural language processing [82], recommendation [83], potential drug molecules prediction [84], analyzing particle accelerator data [85, 86], reconstructing brain circuits [87], and predicting the effects of mutations in non-coding DNA on gene expression and disease [88, 89].
DL architectures contain multiple layers of artificial neural networks (ANN) combined with simple non-linear modules as universal function approximators. These networks have parameters θ, which can be iteratively adjusted through training on real data. Deep learning can both be used in an unsupervised and supervised learning setup. So far, several concepts have already been mentioned. For a better general understanding of the relations amongst these, we assembled a conceptual representation in Figure 4.1. It hierarchically structures, techniques mentioned in the broader scope of AI and ML.

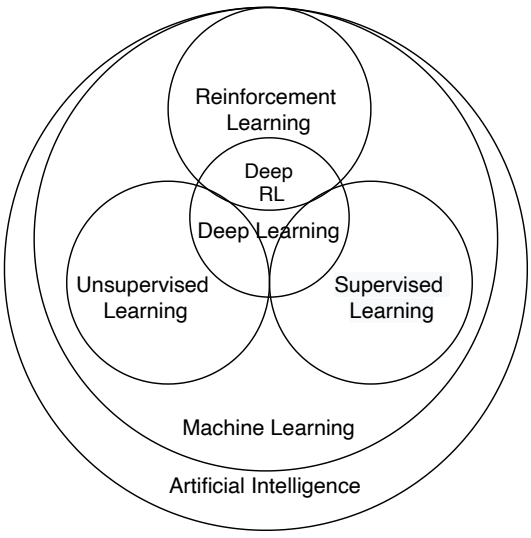

Figure 4.1.: Relations amongst several concepts under the umbrella of Artificial Intelligence (AI) and Machine Learning (ML) with Deep Reinforcement Learning (DRL), Deep Learning (DL), Supervised Learning, Unsupervised Learning, and Reinforcement Learning (RL)

The third area within machine learning is reinforcement learning (RL). Here, an artificial agent acquires experience through try and error interactions with an environment. Especially deep reinforcement learning (DRL), which blends deep learning techniques with RL has very recently shown groundbreaking achievement. For example, in 2019 OpenAI[1] solved a Rubik's Cube with a real-world human-like robot hand using DRL[90]. With AlphaGO, Deepmind[2] created the first ever RL algorithm, to defeat a human world champion in the game of GO.

4.2. Convolutional Neural Networks

The following sections will cover fundamental neural architectures, which we use as building blocks in our final implementation. We begin by introducing convolutional neural networks (CNNs) [91], which are a particular kind of neural network for processing data that has a grid-like topology (i.e., images). As the name convolutional neural networks already imply, they use mathematical convolution operations. The intuition behind CNNs compounds two things. First, a general understanding of standard artificial neural networks (ANN) and second, an extension with convolution operations. We will start by introducing ANNs plus convolutions on grid-like data and combine both concepts. Note, convolutional neural networks will be an initial step in the embedding model implementation. The KG data will be compound into grid-like data arrays to apply a convolutional neural network.

Neural Networks

A neural network is a composition of operations that can be expressed by nodes of a directed graph. Each node, representing an artificial neuron, also called Perception [92] and is illustrated in Figure 4.2. Each input to an artificial neuron x_n is multiplied with the neuron weights w_n and summed up concurrently with an extra bias b. The summation output converts through a non-linear activation function $f = \sigma(\mathrm{wx} + \mathrm{b})$. Nonlinear activation functions are preferred as they allow the network to learn more complex structures in the data. Traditionally, a widely used nonlinear activation functions is the sigmoid function $\sigma(x) = 1/(1 + e^x)$, also called the logistic function. The input to the function is mapped into a range between 0.0 and 1.0. Inputs that are exceeding 1.0 are transformed to the value 1.0. Similarly, values much smaller than 0.0 are bounded to 0.0.
A network containing multiple layers of nonlinear activation functions can fail to receive useful gradient information. The error, which is backpropagated through the network and used to update the weights, decreases dramatically with every ad-

[1] OpenAI is an American AI research laboratory.
[2] DeepMind Technologies is a UK artificial intelligence company founded in September 2010 and acquired by Google in 2014.

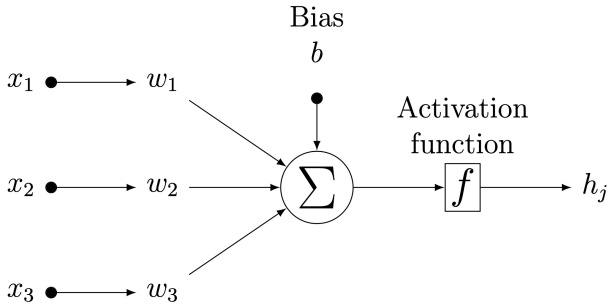

Figure 4.2.: Graphical representation of one artificial neuron

ditional layer. This dilemma is due to the small derivations of sigmoid activation functions and widely known as the vanishing gradient problem, which prevents deep (multi-layered) networks from learning effectively [93]. Therefore, a proposed advance in terms of the activation function is the usage of the Rectified Linear Unit (Equ. 4.1). They are wildly used for deep networks as a replacement for sigmoid functions.

$$f_{\texttt{ReLU}}(x) = \begin{cases} x, & \text{if } x < 0. \\ 0, & \text{otherwise.} \end{cases} \tag{4.1}$$

A multi-layered network, or feed-forward network, is the concatenation of multiple artificial neurons into one input-, various hidden-, and one output-layer. Figure 4.3 illustrates an example for a fully connected neural network with one input layer containing 36 artificial neurons, two hidden layers with six neurons each, and one artificial neuron as the output layer. Convolutional neural networks (CNN) share similar architectures. The main difference is in replacing the fully connected zone within layers by convolution blocks and some additional differentiable operations.

Convolution Operation

In signal processing, the convolution operation between the signal f and g results in a new signal with a modified response. f can be thought of as an 1D, 2D or N-D signal that is convolved with the filter g of respective dimensions, but different size. A convolution between the two signals is defined as:

$$f * g = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{4.2}$$

where the signal g, also known as filter or kernel, is reversed and shifted. The model composes the parameters of each input layer through a convolution. This

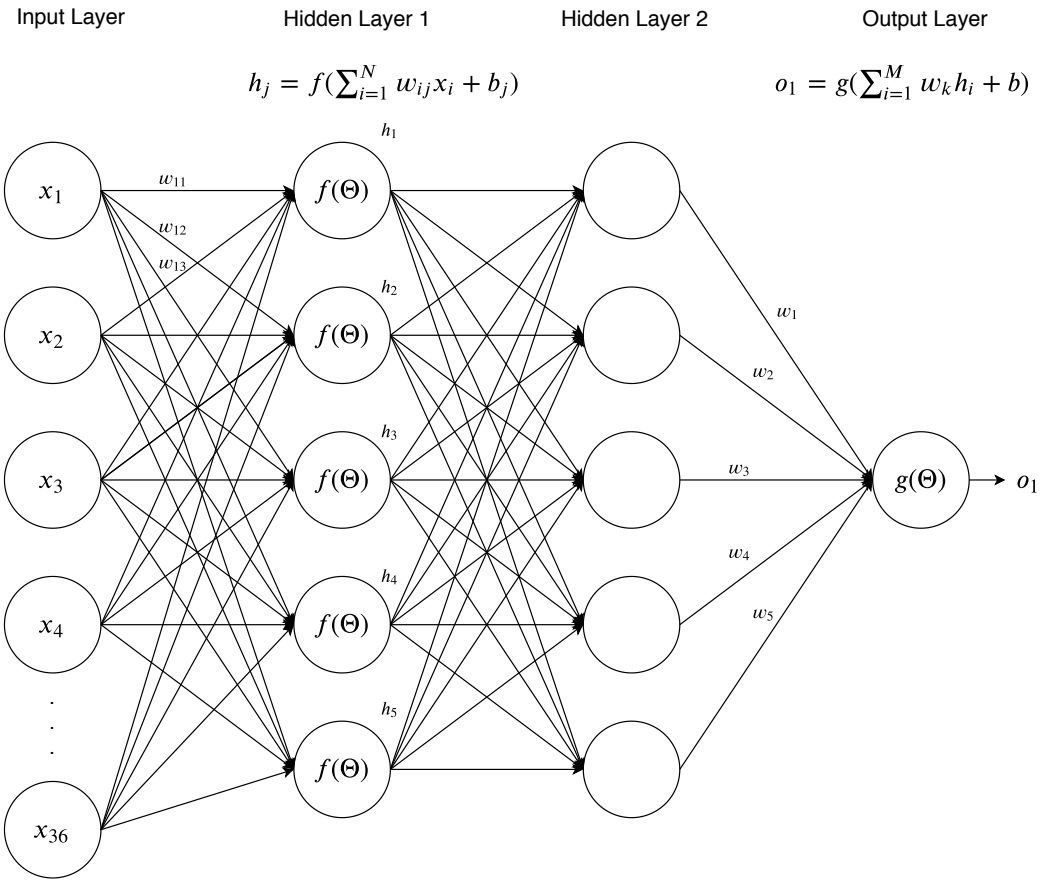

Figure 4.3.: A fully connected multi-layered artificial neural network

particular kind of convolution is also known as spatial convolution. It convolves 2D grid-like data, such as digital gray-scale images or, in our specific case, concatenated embedding vectors. Figure 4.4 underlines the concept of 2D spatial convolutions graphically. As input, we use a 6×6 concatenated entity and relation representation as a two-dimensional tensor.

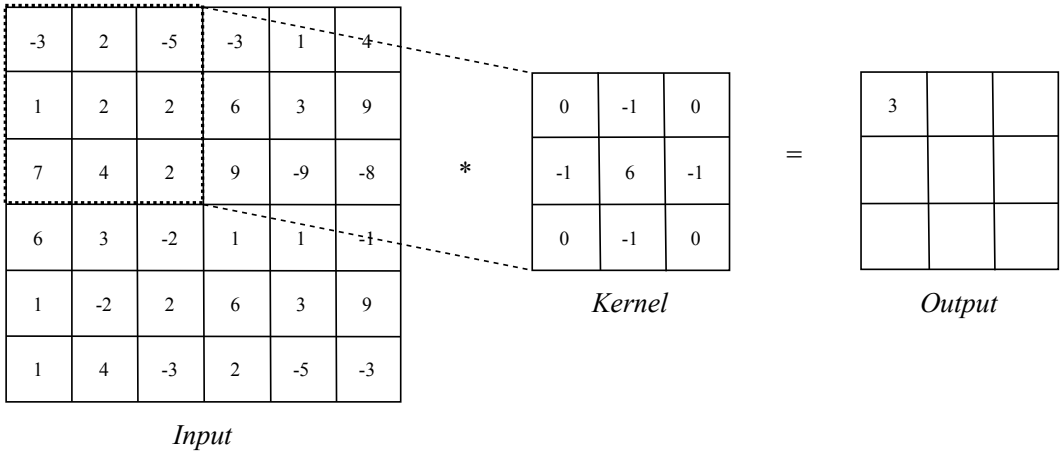

Figure 4.4.: An example of 2-D spatial convolution. The entity and relation *Input* tensor gets convolved by *Kernel* g resulting in the fist feature map *Output* value.

For a 2D discrete signals, the integral in equation 4.2 can be replaced by a summation. The filter g is convolved with the signal f by sliding the kernel over the signal f. At each position, the filter values are multiplied with the signal values and then summed up. Note, the sliding number is called stride. In this particular example, the stride is set to one. The first output value can be calculated in the following way:

$$(-1) \times 2 + (-1) \times 2 + (-1) \times 4 + (-1) \times 1 + 2 \times 6 = 3$$

In convolutional network terminology, the first argument (function f) to the convolution is often referred to as the input and the second argument (function g) as the kernel. The output is sometimes referred to as the feature map. Usually, multiple kernels are applied to the same input tensor, resulting in multiple feature maps. This is useful for extracting different features (or shapes in images) with each kernel.
Note, in example 4.4, we do not flip the channel. In theory, the only reason to flip the kernel is to obtain the commutative property. While the commutative property is useful for writing proofs, it is not usually an important property of neural network implementations. Instead, many neural network libraries implement a closely related operation called the cross-correlation, which is a similar operation, without flipping

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	-3	2	-5	-3	1	4	0	0
0	0	1	2	2	6	3	9	0	0
0	0	7	4	2	9	-9	-8	0	0
0	0	6	3	-2	1	1	-1	0	0
0	0	1	-2	2	6	3	9	0	0
0	0	1	4	-3	2	-5	-3	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Figure 4.5.: Padding on input tensor to retrieve the input dimensions after convolution operator

the kernel. Further investigations regarding the convolution example (Fig. 4.4) reveal a shrinking in dimensions, form a 7×7 original image into a 3×3. However, in most cases, the spatial dimensions should be the same during a convolution. Therefore, padding is used to retain the data dimensions after the convolution by adding two rows of constants (zeros) on the top, bottom, and both sides of the input layer, respectively. The concept is illustrated in Figure 4.5 we padded the output tensor with zeros, also known as zero-padding. Note, the stride is variable and treated as a hyperparameter. In contrast to normal parameters, a hyperparameter in machine learning is not a learn-able parameter and is set before training. The three hyperparameters padding, striding, and filter are important for defining a convolutional layer.

Architecture

A convolutional neural network is a composition of convolution operations of multiple filters, combined with activation functions and other differentiable operations. For the sake of completeness, an essential differentiable operation used for large input dimensions is pooling (or subsampling). Pooling layers perform a downsampling

operation and reduce the input dimensions. The idea is to progressively reduce the spatial representation size and reduce the number of parameters/computation in the network. This technique prevents the model from overfitting. There are several types of pooling layers: max-pooling, average-pooling. However, max-pooling is mostly used and has the best performance [94].
Our input dimensions are comparably low-dimensional and we are not using max-pooling operations. The underlying architecture used is comprised of an input layer, multiple convolutional operations with non-linear activation functions, a fully-connected layer, and one output layer (see Fig., 4.6).

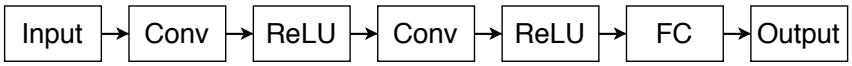

Figure 4.6.: Underlying convolutional neural network architecture

4.3. Long Short Term Memory

The neural network architectures described so far are limited to process a grid of values X from a single time step. However, data can also be coming from a sequence of values $x_1, ..., x_\tau$. In the real-world, this could be continuous measurements (e.g., accelerometer, gyroscope, magnetometer), stock market, audio, or text. In our specific case, we have a sequence of history nodes that the agent visited at each time step. The novel network architecture that can encode sequential data is a recurrent neural network or RNN [95]. To go from multi-layer network to recurrent network, we need to take advantage of one of the early ideas found in machine learning and statistical models: sharing parameters across multiple parts of the model. Parameter sharing allows the model to process large sequences of input samples that would require too many parameters in a standard feed-forward constellation. Also, parameter sharing makes it possible to extend and apply the model to examples of different lengths and generalize across them. In a RNN architectures this is achieved with an inside loop, allowing information to persist. In the signal flow diagram (Fig. 4.7), a chunk of a neural network, h, observes some input x_t and outputs a value y_t. The loop allows information to be passed from one step of the network to the next. An RNN can be thought of as multiple copies of the same network, each passing data to a successor. The following illustration (Fig. 4.8) unrolls the network for each time step $t_0, ..., t_n$. In recent years, there have been multiple successful applications of recurrent neural networks in a variety of problems, including speech recognition, language modeling, or translation. Unfortunately, standard recurrent cells can only access a limited range of contextual information. The influence of a given input on the hidden layer either decays or blows up exponentially when the data flows through

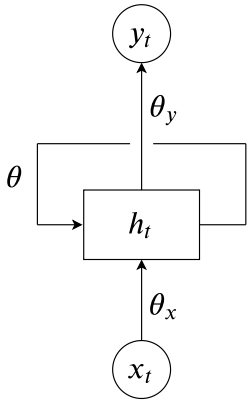

Figure 4.7.: Recurrent neural networks have loops which allows the information to persist.

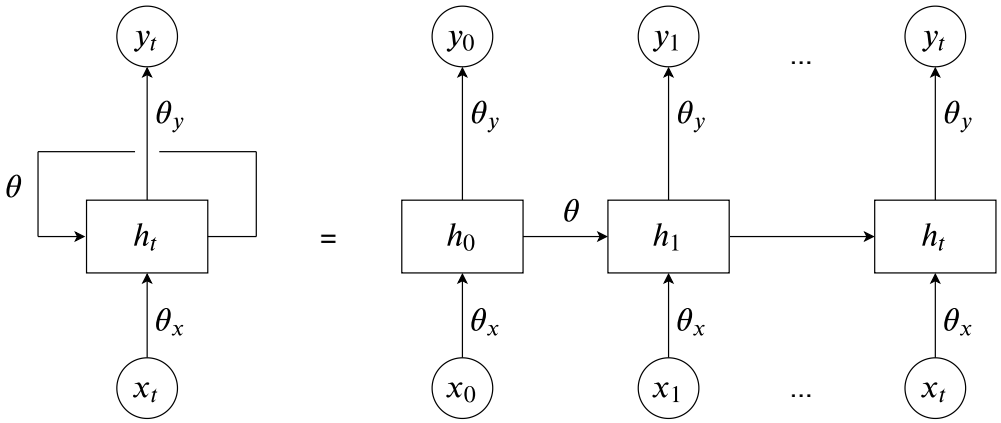

Figure 4.8.: Unfolded computational graph of an RNN for three time steps. Input vectors are labels with x and output vectors with y, respectively. Bias weights are omitted for clarity.

the network. This issue is referred to as the vanishing gradient problem [96, 93]. To address this problem Hochreiter et. al. [96] proposed the Long Short-Term Memory (LSTM) cell as a replacement of the standard recurrent cell, which involves a set of recurrently connected subnets, known as memory blocks. According to Figure 4.9,

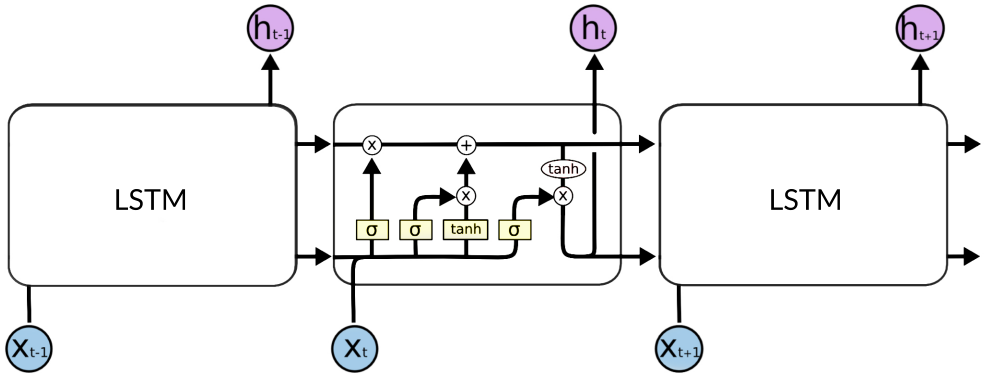

Figure 4.9.: The repeating module in an LSTM network contains four interacting layers marked in yellow.

LSTMs share the same chain structure, but the individual memory blocks where extended. Instead of having a single parameterized neural network loop, there are four, interacting with each other. Each represented by a σ box, i.e gate. A gate is a linear layer with sigmoid activation. The result of the activation is multiplied element-wise with the data vector. Which allows to select pieces of data which shall be forgotten or remembered. The gates can be divided into forget gate, input gate, and output gate. The forget gate is the first σ box, where the neural network learns which information of the cell state shall be forgotten by setting the elements in f_t to 0 or 1. Where, 0 corresponds to completely forget, and 1 means to keep information completely. Values between 0 and 1 correspond to partially retain information. f_t is calculated the following way: $f_t = \sigma(W_t[h_{t-1}, x_t] + b_f)$. The second σ box is called the input gate, indicating which information of the input signal should be stored in the cell. It contains two individual parts. First, it decides about the values of the cell state which should be updated, given by: $i_t = \sigma(W_t[h_{t-1}, x_t] + b_i)$. Next, a tanh layer creates a vector of new candidate values, C_t, that can be added to the state. C_t is given by: $C_t = \tanh(W_t[h_{t-1}, x_t] + b_C)$. Next, the cell state C_t is updated by applying the multiplication of the forget gate and adding the gated new data. This can be written as: $C_t = f_t \cdot C_{t-1} + i_t \cdot C_t$. Finally, the output gate decides which parts of the cell state is used for the new hidden state h_t, based on the previous hidden state h_{t-1} and the input x_t. It is given by: $\sigma_t = \sigma(W_\sigma[h_{t-1}, x_t] + b_\sigma)$. A tanh activation is applied on the cell state C_t before applying the output gate. This is to normalize the values to be between -1 and 1. Lastly, the output of the hidden

state h_t is computed based on the the cell state and the result of the output gate, given by $h_t = \sigma_t \cdot \tanh(C_t)$. The hidden state h_t will go to the next LSTM cell or the output layer.
During training, RNNs apply Backpropagation Through Time (BPTT) to calculate the derivative between the loss function and the network weights. Like standard backpropagation, BPTT consists of a repeated application of the chain rule. The difference is that the loss function depends on the hidden layer not only through its influence on the output layer but also through its impact on subsequent hidden layers. Note, we will not go into details on the backpropagation algorithm, instead we refer to [6, p.200] for intuitive explanations. In the next section we will discuss relevant optimization techniques.

4.4. Optimization

This section focuses on one particular case of optimization: finding the parameters θ of a neural network that significantly reduces a cost function $J(\theta)$, which typically can be written as an average over the training set, such as:

$$J(\theta) = \mathbb{E}_{(x,y)\sim\hat{p}_{data}} \mathcal{L}(f(x;\theta), y) \tag{4.3}$$

where $\mathcal{L}$ is the per-example loss function, $f(x;\theta)$ is the predicted output when the input is x, and $\hat{p}_{data}$ the empirical distribution. Gradient-descent [6] is the main optimization algorithm used to minimize some Loss function $\mathcal{L}$ by iteratively updating the network parameters.
Popular loss functions are cross entropy-, L1-, and L2-Loss. Where cross-entropy measures the distance between two probability distributions. One probability distribution corresponds to the softmax output of the NN and the other to the ground-truth. L1-Loss function minimizes the absolute differences between the estimated values and the existing target values. It can be written as: $\mathcal{L} = \sum_i^K |y_i - x_i|$. L2-Loss function minimizes the squared difference between the prediction and target. It can be written as: $\mathcal{L} = \sum_i^K (y_i - x_i)^2$. After determining a beneficial loss function, back-propagation is used to calculate the gradients of the parameters $\nabla\theta$, and gradient descent for updating the parameters. We will not go into detail on how back-propagation works since this algorithm is a standardized approach to calculate the individual gradients of the network parameters. It is already implemented in most modern ML libraries. For example, PyTorch[3] uses `autodiff()` or `autograd()` for automatic differentiation. After calculating the gradient for each parameter, we apply gradient descent to update the parameters, respectively. The Gradient descent algorithm can optimize the loss function by starting from an initial search point, i.g, set of parameter values for θ, and take a step opposite to the direction of

[3] https://pytorch.org

the gradient to update the parameters accordingly. The learning rate η regulates the influence of the gradient after each step. This is an iterative process and tends to converge very well to the global minimum of the loss function, or the local minimum when the loss is non-convex.
An intuitive example, could discribe the process as follows. We start with a forward pass through the network to calculate the resulting loss (error). Then use back-propagation to compute the gradient of the loss with respect to each variable and use gradient descent to update the weights, respectively. The process is repeated in a loop until we reach convergence, or certain criteria are fulfilled. There are three variant of gradient descent:

- **Batch gradient decent**, where the gradient of the loss $\mathcal{L}(\theta)$ with respect to the parameters θ, is computed on the whole training set. It is defined as:

$$\theta_{t+1} = \theta_t - \nabla_\theta \mathcal{L}(f(x; \theta_t), y) \tag{4.4}$$

- **Stochastic gradient descent**, where the the parameter update is performed for each training sample separately. It is defined as:

$$\theta_{t+1} = \theta_t - \nabla_\theta \mathcal{L}(x_i, y_i, \theta_t)) \tag{4.5}$$

- **Mini-batch gradient descent**, peforms a gradient update for every mini-batch of n samples. It is defined as:

$$\theta_{t+1} = \theta_t - \nabla_\theta \mathcal{L}(x_i, y_i, \theta_t)) \tag{4.6}$$

Stochastic Gradient Descent

Batch methods, such as batch gradient descent, use the full training set to compute the next update. This technique tends to converge very well to local optima. In practice, computing the loss with respect to the gradient over the entire training set can be prolonged and sometimes intractable on a single computer when the dataset is too large to fit the memory. Another issue with batch optimization methods is that they do not give an easy way to incorporate new training data in an 'online' setting. These problems were addressed by Stochastic Gradient Descent (SGD)[97]. An SGD algorithm is computing the gradient of the parameters using only a single training example. However, this generates a high variance due to the lack of gradient averaging during the update step. Most stochastic gradient descent algorithms share the idea of mini-batch gradient descent to compensate for this problem. They perform a gradient update for every mini-batch of n samples. The best trade-off between efficient and stable training. Later, when referring to SGD, it will imply the mini-batch version defined in Equation 4.6.

Momentum

In practice choosing the right learning rate is not trivial. Low learning rates can lead to slow convergence, whereas high learning rates can hamper the convergence, and the loss function can fluctuate around the minima or even diverge. An adjustment of the learning rate during training is useful. A larger learning rate is only necessary during the first few episodes. Momentum [98] is used as an updating rule for the learning rate. The aim of the momentum is to increase over time and add more velocity u to the parameter update, symbolized in Equation 4.7. The parameter update remains the same and is given by: $\theta_{t+1} = \theta_t + u$.

$$u = \alpha u - \eta \nabla_\theta (\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(f(x^i; \theta_t), y^i)) \tag{4.7}$$

Adaptive Learning Rate

Momentum provides a robust solution for a variable learning rate during gradient update. However, in practice, we encounter situations where some parameters are more sensitive to the gradient update than others. Avoiding performance drops and convergence difficulties means the gradient influence should not be the same for all parameters. This is the case in adaptive learning rate algorithms. The learning rate is adapted component-wise for individual parameters by incorporating knowledge of past observations. It performs larger updates (i.e., high learning rates) for those parameters that are infrequently updated and smaller updates (i.e., low learning rates) for frequently updated parameters. The current gradient values are scaled by the reciprocal of the square root from the sum over all historical squared gradient values. This specific algorithm is called Adagrad [99] and comprises the following three steps:

1. Compute the gradient g:

$$g = \nabla_\theta (\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(f(x^i; \theta_t), y^i))$$

2. Accumulate squared gradient:

$$\mathrm{G} = \mathrm{G} + g \otimes g$$

   where G is a diagonal matrix and the operation $\otimes$ corresponds to the outer product.

3. Compute and apply gradient:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\varepsilon I + diag(G_t)}} \cdot g_t$$

$G_t \in \mathbb{R}^{d \times d}$ here is a diagonal matrix where each diagonal element i, is the sum of the squares of the gradients w.r.t. θ_i. An additional ϵ is a smoothing term that avoids division by zero.

A central benefit of using Adagrad[99] is the elimination of manually tuning the learning rate. However, this comes with a significant drawback, such as the accumulation of gradients in the denominator. All terms added are positive, which causes the sum to get very large. This causes the learning rate to get infinitely small during training. The algorithm is no longer able to train appropriately with a learning rate close to zero.
Recently proposed algorithms like Root Mean Square Propagation (RMSProbp) do also maintain per-parameter learning rates. However, they are adapted on the average of current magnitudes from individual gradients (e.g., how quickly they change). Meaning the algorithm does well on online and non-stationary problems (e.g., noisy). Adam optimizer [3] blends the benefits of both AdaGrad and RMSProp. Rather than adapting the parameter learning rates based on the average first moment as in RMSProp, Adam makes further use of the second-order gradient moments. The algorithm calculates an exponential moving average of the gradient and the squared gradient, parameterized by $beta1$ and $beta2$.
Summarizing the various optimization algorithms, we must admit that the choice of optimization algorithm strongly depends on the underlying data. Following current optimization evaluations, the Adam optimizer is currently recommended as the default algorithm to use [100, 101]. Figure 4.10 compares the performance of various optimization algorithms mentioned. It indicates that Adam ultimately converges considerably faster. Following recent work, we will use Adam as an optimizer to train our models' knowledge graph data [5].

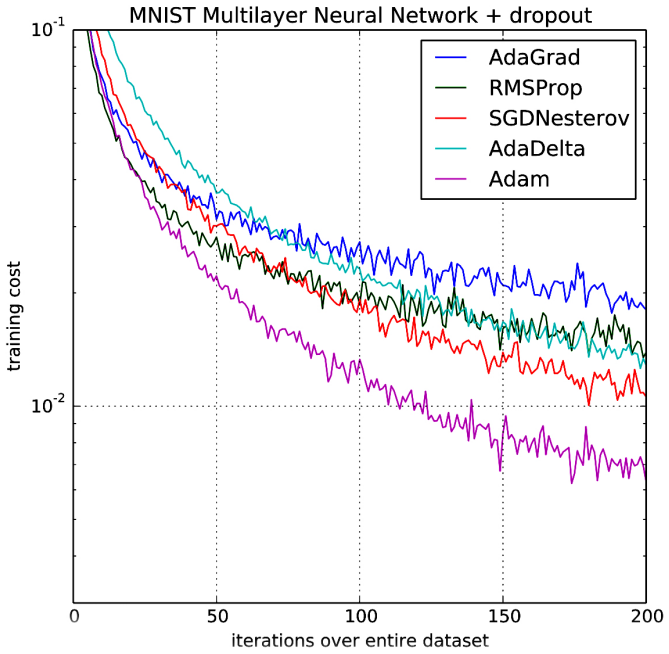

Figure 4.10.: Performance comparison of various optimization algorithms used for training a multi-layer feed-forward network on the MNIST digits classification data set. In practice, Adam is currently recommended as the optimization algorithm of choice [3] since it uses adaptive learning rates to update individual parameters.

4.5. Reinforcement Learning

Until now, we defined relevant architectures and optimization algorithms, which will now expand on concepts in reinforcement learning. In reinforcement learning, an artificial agent interacts with a predefined environment, where it is executing actions to maximize some given objective. This means the agent creates its training samples by trial-and-error interactions with the environment. Given a state $S_t \in S$, the agent can take one of the available actions $A_t \in A(s)$. The environment responds with a new state S_{t+1} and a reward R_{t+1}. The reward is a numeric representation that allows the agent to evaluate the success of the action taken. The agent seeks to maximize the rewards over time through its choice of actions. This Agent-

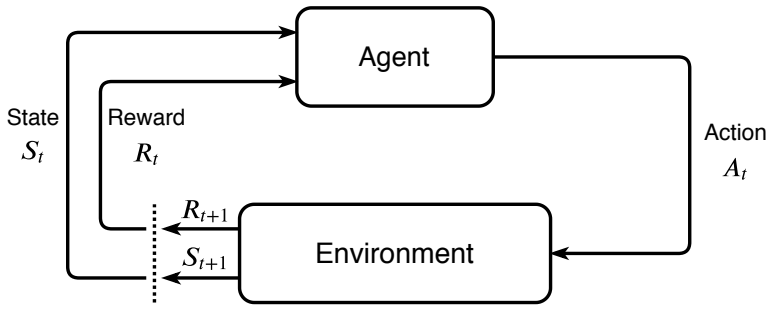

Figure 4.11.: Agent-Environment interaction as a general reinforcement framework. At each time stamp t the agent is in a state $s_t \in S$ and takes one of the possible available actions $a_t \in A$. The environment responds with a new state s_{t+1} and a reward R_{t+1} [4].

Environment interface (Fig. 4.11) is a sequential decision making problem and can mathematically formulated as a finite Markov decision process, or finite MDP. The MDP and agent together thereby give rise to a sequence or trajectory τ that begins like the following:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \tag{4.8}$$

4.5.1. Markov Decision Process

In a finite MDP, the sets of states, actions, and rewards (S, A, and R) all have a finite number of elements. Finite, because in an episodic task, the agent will terminate after a predefined number of steps. In this case, the random variables R_t and S_t have well defined discrete probability distributions dependent only on the preceding state and action (Markov assumption). That is, for particular values of these random variables, $s' \in S$ and $r \in R$. There is a probability of those values occurring at time t, given particular values of the preceding state and action. The function p is defined as the dynamic or transition model for each action specified in

the equation 4.9.

$$p(s' \mid s, a) = \Pr(S_t = s' \mid S_{t-1} = s, A_{t-1} = a) \tag{4.9}$$

Furthermore, in Equation 4.10 the reward function is defined as an expected value given the state and action.

$$r(s, a) = \mathbb{E}(R \mid S_t = s, A_t = a) \tag{4.10}$$

These conclude the mathematical formulations necessary to set up a finite MDP. The agents' global objective is to maximize the expected cumulative reward. The aggregated reward at each time step t can be written as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{n=1}^{T} \gamma^n R_{t+n+1} \tag{4.11}$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the discount rate. The discount rate weights the present value of future rewards: a reward received n time steps in the future is worth only γ^{n-1} times what it would be worth if it were received immediately. For the sake of simplicity, we will break the agent-environment interaction down into sequences of separate episodes. Note, in a continuous task, the agent has to learn how to choose the best actions and simultaneously interact with the environment. The former case is mathematically easier because each action affects only a finite number of rewards received during the episode.

4.5.2. Policy and Value Functions

To evaluate how "good" the current position for the agent is we use a value function $v_\pi(s)$. The notion of "how good" here is defined in terms of future rewards that can be expected. The reward the agent can expect to receive in the future depends profoundly on whatever actions it is going to take. Therefore, value functions are defined concerning particular strategies of environment interactions, namely a policy π. Mathematically, a policy $\pi(a|s)$ is a stochastic rule by which the agent selects actions as function of state $a = \pi(s)$. The value of a state s following a policy π, denoted $v_\pi(s)$, is the expected future return when following π. We call $v_\pi(s)$ the state-value function for policy π (Equ. 4.13).

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{n=1}^{T} \gamma R_{t+n+1} \right], \forall s \in S \tag{4.12}$$

Similarly, we define the value of taking action a in state s under a policy π, denoted $q(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$q(s, a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right], \ \forall s \in S, \ \forall a \in A \tag{4.13}$$

The state-value function v_π and action-value function q_π can be learned from experience. In our case, by taking paths in the knowledge graph. While following a policy π, the agent sequentially selects an outgoing edge and traverses to a new entity until it arrives at the target note. The agent encounters, for each state, an actual return that has followed that state, then the average will convert to the state's value, $v_\pi(s)$. If separate averages are kept for each action taken in each state, then these averages will similarly converge to the action values, $q_\pi(s, a)$. We call estimation methods of this kind Monte Carlo methods because they involve averaging over many random samples of actual returns. In most real-world scenarios, the state and action space is usually very high. In this case, it may not be practical to keep separate averages for each state individually. Instead, the agent would have to approximate v_π and q_π as parameterized functions. This is where ANN, as non-linear function approximator, is applied. When using deep learning models to approximate the value functions, or the policy π directly, is denoted as deep reinforcement learning.

4.6. Deep Reinforcement Learning

Solving a model-free reinforcement learning task means, roughly, finding a policy that achieves a lot of reward over the long run. Two ways to approach this problem: using value-based methods or policy gradient methods. Value-based methods learn the values of actions and then select actions based on the highest estimated action values. Their policies would not even exist without the action-value estimate.
Policy gradient methods learn a parameterized policy that can select actions without consulting a value function. Note, a value function may still be used to learn the policy parameter but is not obligatory for action selection.

4.6.1. Value-based Methods

To find the optimal action-value function q^* for each state, we incorporate the recursive relationship between the value of s and the value of its possible successor states s'. This relationship is modeled by the following Equation 4.14, which is called the Bellman equation [4]. The Bellman equation forms the basis of a number of ways to compute, approximate, and learn q_π^*

$$q^*(s, a) = \sum_{s', r} p(s', r|s, a) \left[R(s, a) + \gamma \max_{a'} q_*(s', a') \right] \tag{4.14}$$

4.6.2. Policy Gradient Methods

So far only implicit deterministic policies have been considered. The agent is given a state and selects a action according to largest q value. In contrast, methods that

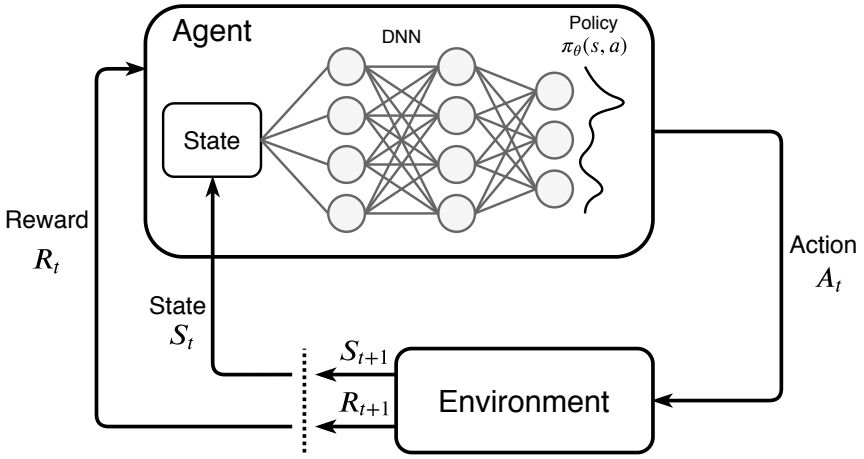

Figure 4.12.: Deep Reinforcement learning leverages a ANN as non-linear function approximator parameterized by θ in order to learn a policy function π_θ, from where the next action at is taken.

learn a parameterized policy can select actions without consulting a prior value function. This methods are called policy gradient methods. Where the policy function $\pi(a|s,\theta) = \Pr(A_t = a \mid S_t = s, \theta_t = \theta)$ outputs a probability disrtibution of actions and is parameterized by $\theta \in \mathbb{R}^{d'}$. The goal is to learn a policy function $\pi(\theta)$ with parameters θ by maximizing an performance measure $J(\theta)$ (Eq. 4.15).

$$\theta^* = \arg\max_\theta J(\theta) \tag{4.15}$$

The parameters θ are maximized with gradient ascent. This updates θ_{t+1} in the positive direction of the gradient of the policy performance measure $\nabla_\theta J(\theta)$ with the learning rate α (Eq. 4.16). $\nabla_\theta J(\theta)$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its parameter θ_t.

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta) \tag{4.16}$$

Policy-gradient methods are capable of learning stochastic policies, which is necessary when the environment is uncertain. Another reason for using policy-based methods over value-based are the better convergence properties. The problem with value-based methods is their significant oscillation while training. This occurs because the choice of action may change dramatically for an arbitrarily small change in the estimated action values. Additionally, policy gradients are more effective in high dimensional action spaces. A knowledge graph is considered to be a high dimensional space. It can quickly scale up to 10^6 nodes, each node incorporating up to 100 relations leading up to 10^{10} possible actions.

Using policy gradients, we need to encounter two main drawbacks. Firstly, they converge most of the time to a local minimum rather than a global one. Secondly, they are very data inefficient. Reinforcement learning algorithms, in general, are very data inefficient, but policy gradients tend to converge even slower compared to value-based methods. This results in longer training times [4].
Evaluating advantages and disadvantages lead to the conclusion, considering the complex environment, and state-action space, we are going to follow previous work [27, 29] and use policy gradient methods for policy optimization. The following section introduces the necessary theory behind REINFORCE, a policy gradient algorithm that will be used to approximate and learn the policy π.

REINFORCE

The REINFORCE algorithm was introduced by Williams in 1992 [80], and also known as Monte-Carlo policy gradient. It relies on an estimated return by Monte-Carlo methods using episode samples to update the policy parameter θ. REINFORCE works because the expectation of the sample gradient is equal to the actual gradient. Given an episodic case the objective $J(\theta)$ is defined as the maximal expected future reward over trajectory (eq. 4.8) evaluated on the policy π.

$$J(\theta) = \mathbb{E}_{\pi}[G(\tau)] \tag{4.17}$$

The solution will be derived by using the Policy Gradient Theorem. It provides an analytic expression for the gradient $\nabla J(\theta)$ with respect to policy π that does not involve the differentiation of the state distribution.

$$\begin{aligned}
\nabla_{\theta} J(\theta) &= \nabla_{\theta} \sum_{\tau} \pi(\tau, \theta) G(\tau) \\
&= \sum_{\tau} \nabla_{\theta} \pi(\tau, \theta) G(\tau) \\
&= \sum_{\tau} \frac{\pi(\tau, \theta)}{\pi(\tau, \theta)} \nabla_{\theta} \pi(\tau, \theta) G(\tau) \\
&= \sum_{\tau} \pi(\tau, \theta) \nabla_{\theta} \ln \pi_{\theta}(\tau, \theta)
\end{aligned} \tag{4.18}$$

5. Methodology

The previous chapter introduced fundamental concepts in unsupervised learning and deep reinforcement learning, which now serves as a foundation to elaborate on specific concepts. We will leverage fundamental concepts of deep reinforcement learning and policy gradient methods to model our given deep-medical reasoning system. Here, the task is predicting new links between nodes in the UMLS knowledge graph, based on proven existing evidence. By following previous work [39], we leverage an RL-based framework for multi-hop relation reasoning, aiming to explore and predict paths between entity pairs, not existing in the KG, due to graph incompleteness. This multi-hop pathfinding problem is formulated as a sequential decision-making problem that can be solved with an RL agent. The environment will be framed as a Markov Decision Process (MDP) and a policy-based RL agent will learn a stochastic policy to pick promising reasoning paths.
The chapter is structured in the following way. The first two sections will cover the reinforcement learning formulation, including the various MDP components and the policy network. For training the system in the third section, we oblige a state-of-the-art neural embedding model to overcome the sparse reward problem, which we will then introduce. The embedding model can be seen as an essential subsystem since it delivers crucial reward information to the RL-agent. We will start by training the embedding-based model using unsupervised learning techniques, followed by optimizing the RL agent.

5.1. Framework

The framework is based on a classical two-component reinforcement learning scenario (Figure 5.1), where the artificial agent interacts with an environment by choosing action to maximize the future reward. This process is modeled as a Markov Decision Process (MDP). The MDP is defined as a tuple $\{\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}\}$, where $\mathcal{S}$ is the continuous state space, $\mathcal{A}$ is the set of available actions depending on the current state, $\mathcal{T}$ is the transition probability matrix, and $\mathcal{R}$ is the reward function.
The second substantial part of the system, is the RL agent represented as a parameterized policy network $\pi_\theta(s, a)$. The agent starts at a initial query $(e_s, r_q, ?)$, where e_s is the source entity and r_q is the unknown relationship of interest and performes an efficient search. This is done by hopping over the KG and evaluating the set of possible answers, which are not directly connected with the source entity, due to graph incompleteness. The following list will define the exact mathematical formulation of the MDP-Model, which serves as a framework to leverage reinforcement

learning optimization techniques. We decided to use the definition offered in [39] rather then [27]:

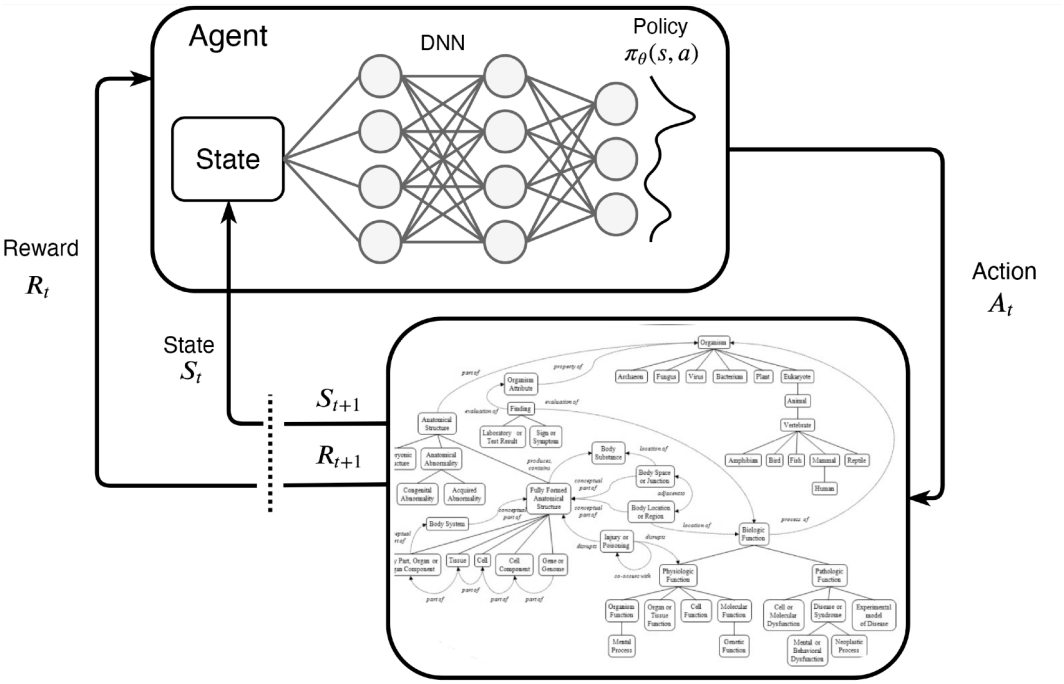

Figure 5.1.: The framework based on a classical two component RL scenario modeled as a Markov Decision Process.
Lower part: The environment incorporates to the whole KG.
Upper part: A representation of the policy network, which outputs the probabilities for choosing the next action.
At each step t, the agent interacts with the environment and learns selecting a relation link to extend the reasoning path.

Environment: The environment refers to the entire KG. The graph is formally represented as $\mathcal{G} = (\mathcal{E}, \mathcal{R})$, where the $\mathcal{E}$ is the set of entities and $\mathcal{R}$ is the set of relations. Each node (entity), is connected through a specific relationship. A true triple in the KG (e_h, r, e_t) defined as $x_{ijk} \in \mathcal{G}$. The environment remains consistent throughout the training process.

States: The state space $\mathcal{S}$ consists of all valid combinations in $\mathcal{E} \times \mathcal{R} \times \mathcal{E}$. A specific state needs to encode the query (e_s, r_q), and the current location of the RL agent during exploration e_t. Intuitively e_t can be interpreted as the state-dependent information while (e_s, r_q) is the global context shared by all states. The state s_t is defined in the following:

$$s_t = (e_t, (e_s, r_q)) \tag{5.1}$$

Actions: The set of possible actions $A_t \in \mathcal{A}$ at every time step t contains the outgoing edges of the current vertex e_t in $\mathcal{G}$ and is formally defined in Equation 5.2. At each state, the agent has the option to choose one of the possible outgoing relations, by incorporating all state-dependent outgoing relationships r' and potential destination nodes e'. During implementation, the computation graph will be unrolled to a fixed number of time steps T. An additional self-loop is added to every A_t, providing the agent with the option to terminate a search. This is important, given questions that are easier to answer, and require fewer steps of reasoning than others. This is especially helpful, when the agent has managed to reach a correct answer at a time step $t \leq T$ and can continue to stay at the 'answer node' for the rest of the time steps, the self-loop acts similarly to a 'stop' action.

$$A_t = \{(r', e' | (e_t, r', e') \in \mathcal{G}\} \tag{5.2}$$

Transition: A transition function $\delta : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ can be defined according to Equation 5.3. Since the environment is deterministic we simply update the state to the new vertex incident to the edge selected by the agent. The query and answer remains the same.

$$\delta(S, A) = \delta(e_t, (e_s, r_q), A_t) \tag{5.3}$$

Reward: Reward is the feedback, the agent receives depending on a series of actions lead to the ground truth tail entity e_T, in a predefined amount of time steps T. In this case, the agent receives a binary reward 1 when stopped at the correct answer node $R_b(s_T) = \mathbb{I}\{(e_s, r_q, e_T)$. However, the knowledge graph $\mathcal{G}$ is intrinsically incomplete, having only one binary reward based on the agent's success in reaching a correct answer, would penalize the false-negative search attempts identically to true negatives. In order to mitigate this problem, a soft reward shaping technique is adopted from [39]. This contains the state-of-the-art embedding-based model ConvE, described in chapter 3.4.2. The model estimates a soft reward for target entities whose correctness is unknown, which we will then use as a reward to our agent. The embedding-based models map $\mathcal{E}$ and $\mathcal{R}$ in vector space, and estimate the likelihood of each fact $x_{ijk} \in \mathcal{G}$ using a scoring function $f(x_{ijk})$. This allows the model to capture link semantics well, and unobserved but correct answers would

receive a higher reward compared to a true negative entity. We use the following (Eq. 5.5) reward shaping strategy [102].

$$R(s_t) = R_b(s_t) + (1 - R_b(s_t))f(e_s, r_q, e_T) \tag{5.4}$$
$$\tag{5.5}$$

Where $R_b(s_T) = \mathbb{I}\{(e_s, r_q, e_T)$. If the agent stops at the correct answer destination e_T according to G, it receives a reward 1. Otherwise, the agent receives a fact score estimated by the scoring function $f(e_s, r_q, e_T)$, which is required to be trained in advance.

5.2. Policy Network

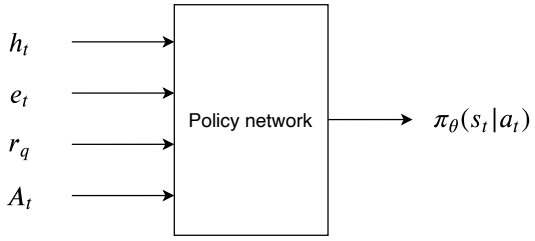

Figure 5.2.: The input–process–output (IPO) diagram presents the policy network interface. Inputs starting from the top are LSTM encoded history, current entity, initial query relationship, and action space.

In the following section, we will introduce the policy network or the agents 'brain' that determines what action to choose at a given state. To solve the finite horizon deterministic MDP described earlier, a non-stationary history-dependent policy is learned. The policy is parameterized using state information and global context, plus the search history. Every entity and relation in $\mathcal{G}$ is assigned to a dense vector embedding $\mathbf{e} \in \mathbb{R}^d$ and $\mathbf{r} \in \mathbb{R}^d$. A specific action $a_t = (r_{t+1}, e_{t+1}) \in A_t$ can be represented as a concatenation between relation embedding $\mathbf{r}$ and the end node embedding $\mathbf{e'_t}$, as $\mathbf{a}_t = [\mathbf{r}; \mathbf{e'_t}]$. The search history $h_t = (r_0, e_s, r_1, e_1, r_2, e_2, \ldots, r_t, e_t)$ encodes the actions taken up to step t, by using an LSTM [96]. A graphical representation is provided in Figure 5.3. We start our explanations from the bottom layer (white circles). Here, the raw history h_t contains previously visited nodes $e_t = (e_s, e_1, e_2, \ldots e_t)$, and actions (relations between nodes) $r_t = (r_0, r_1, r_2, \ldots, r_t)$ taken by the agent. One layer above, actions and nodes will be embedded into a dense vector and concatenated (i.e stacked) in the following way:

$$\left(\begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} ; \begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} \right) = \begin{bmatrix} e_1 & e_2 & e_3 & r_1 & r_2 & r_3 \end{bmatrix}^T$$

For simplicity the embedding dimension in the example is set to $d = 3$. In practise more features are used. Our history encoding layer will contain 400 features.
The third layer encodes the concatenated vector representation recursively using an LSTM in the following way (Equ. 5.6 and 5.7):

$$\mathbf{h}_0 = \texttt{LSTM}(\mathbf{0}, [\mathbf{r}_0; \mathbf{e}_s]) \tag{5.6}$$

$$\mathbf{h}_1 = \texttt{LSTM}(\mathbf{h}_{t-1}, \mathbf{a}_{t-1}), \ \forall t \leq 0 \tag{5.7}$$

Note in Equation 5.6, r_0 refers to an exceptional start relation used to form a start action with e_s, after the initial step the history will be encoded by following Equation 5.7.
Moving one layer up, the encoded history representation $\mathbf{h_t}$ is concatenated with the embedding representation of the current state embedding $\mathbf{e_t}$ and query relationship embedding $\mathbf{r_q}$, respectively. Based on the history embedding $\mathbf{h_t}$, the policy network chooses an action from all available actions A_t conditioned on the query relation. Recall, each possible action represents an outgoing edge, including relevant information about the relationship and subsequent destination node. The embedding for each $A_t \in \mathcal{A}$ is $[\mathbf{r}_0; \mathbf{e}_s]$, and stacking embeddings for all outgoing edges we obtain the action space matrix $\mathbf{A_t} \in \mathbb{R}^{|A_t| \times 2d}$. The matrix is then scalar multiplied with the history encoder. The history encoder contains the global context (i.e., current state $\mathbf{e}_t$, and initial query relationship $\mathbf{r}_q$), plus the search history $\mathbf{h}_t$ through a two-layer feed-forward network with a nonlinear ReLU activation function. It outputs the probability distribution, which will be assigned to each available action in $\mathcal{A}_t$ through scalar multiplication. The final output by the policy network is a probability distribution over possible actions from which a discrete action, with the highest probability, is sampled in the following way $A_t \sim \texttt{Categorical}(\pi_\theta(a_t|s_t))$.
The policy network can be mathematically expressed as follows:

$$\pi_\theta(a_t|s_t) = \sigma(\mathbf{A}_t \times W_2 \texttt{ReLU}(W_1[\mathbf{e}_t; \mathbf{h}_t; \mathbf{r}_q])) \tag{5.8}$$

where W_1, W_2, correspond to the LSTM parameters. The embedding matrices represent the parameters θ of the entire policy network.
The next section will introduce the optimization technique to update the parameters of the stochastic policy during training. However, in order to train the agent appropriately and leverage the soft rewards skim, we need to train the embedding-based ConvE first on the knowledge graph training set.

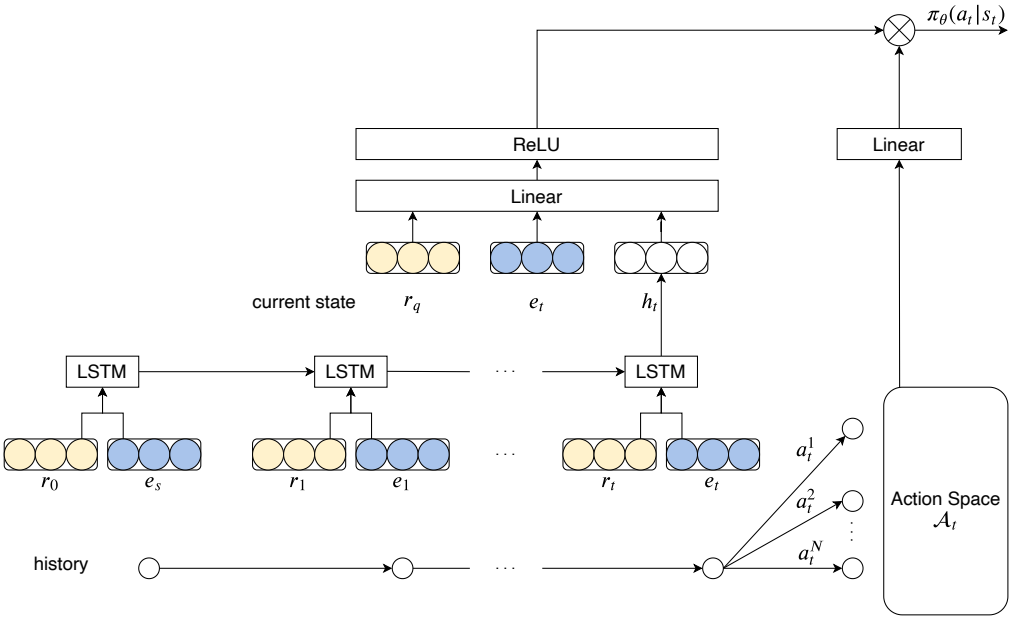

Figure 5.3.: The policy network outputs a probability distribution over possible actions. For the sake of simplicity, the model can be divided into two parts. A path encoder (left) and the Action Space $\mathbf{A}_t$ (right), which in the end gets combined using scalar multiplication. The path encoder contains global context and the search history. Wheres $\mathbf{A}_t$ is a matrix containing all available actions. The scalar product combines both and outputs a probability distribution over all available actions. Where we then use a categorical distribution to select a specific action A_t

5.3. Training

To train the policy network efficiently, we are following previous work [39] and compute a soft reward with the ConvE model. Recall, this step is necessary to overcome the sparse reward problem. Real-world KGs are intrinsically incomplete, which leads to an arrival at the correct answer whose link is not in the training set. In this case, we will use the output provided by the ConvE model as a reward to the agent, to prevent sparse feedback. The process is broken down into the following three steps. First, we acquire the UMLS knowledge graph data and run prepossessing computations. In a second step, we describe the essential parts to train the knowledge graph embedding model on the UMLS data set. Third, we explain the policy network optimization using the REINFORCE [80] algorithm.

5.3.1. Data Prepossessing

The knowledge base used to train the models and evaluate the system is the Universal Medical Language System (UMLS). It incorporates biomedical and general medical concepts. The data can be downloaded from the official National Library of Medicine Website[1] and is free for academic and research purposes. From now on, as we are going to refer a lot to the actual graph structure, it is appropriate to adjust the notation from (Subject, Object, Predicate) to more intuitive and relational enhancing notation (head entity, relation, tail entity). Although both variants can be used interchangeably throughout the paper.
After unpacking, we construct the data into triples (e_h, r, e_t). Where e_h refers to the head entity, r is the relation, and e_t is the tail entity. Every KG link is treated as bidirectional and the reversed (e_t, r^{-1}, e_h) links were augmented to the graph [27]. The inverse triples enable the agent to step backward in the KG. In total we accumulate 6.529 triples in the data set which we split into three buckets: train, dev, and test. Where 80% forms the training set and 20% the test and dev set. The train set, with 5.216 triples, will be used to train the model. Whereas the dev set, with 652 facts, is for additional hyperparameter tuning. The test set, with 610 facts, is exclusively used for performance evaluations. It is necessary to always keep a good separation between test- and training-set [103]. Therefore, we excluded any link from the dev in the test set. Plus its reversed link from the train set. Otherwise, the model could replicate what it has learned on the test set relations without any generalization necessary.

5.3.2. Embedding-based Model Optimization

The model used to predict the soft reward to our agent is the embedding model ConvE. Referring to various benchmarks, the model has shown state-of-the-art per-

[1] `https://www.nlm.nih.gov/research/umls/index.html`

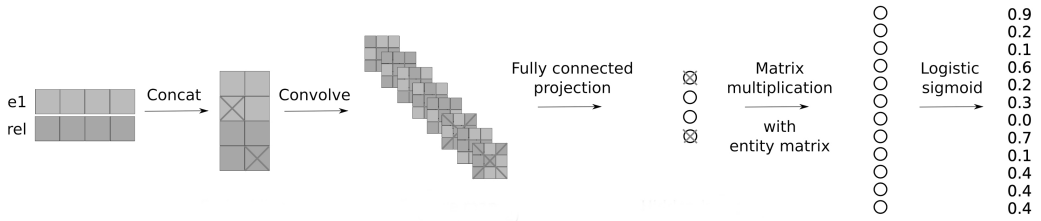

Figure 5.4.: The ConvE model architecture, reshaped and concatenated the entity and relation embeddings in the first two steps; the resulting matrix is then used as input to a convolutional layer (step 3); the following feature map tensor is vectorised and projected into a k-dimensional space (step 4) and matched with all candidate object embeddings (step 5) [5].

formance on comparable data sets [5, 24]. The source code is released under the three-clause BSD open-source license on Github[2] published by Tim Dettmers (2018). The architecture is summarised in Figure 5.4. Here, the first operation, the model performs during a feed-forward pass, is a vector look-up on both embedding matrices; $\mathbf{E}^{|\mathcal{E}|\times 200}$ and $\mathbf{R}^{|\mathcal{R}|\times 200}$ for the entity-, and relation-embedding. Note, $|\mathcal{E}|$ and $|\mathcal{R}|$ denote the number of entities and relations with an embedding dimension set to 200 for most of the experiments. The model then concatenates $\mathbf{e_s}$ and $\mathbf{r}$ to use it as an input to the 2D convolutional layer. The convolution uses three filters, each 3×3 in kernel size. The layer returns a feature tensor $\mathcal{T} \in \mathbb{R}^{3\times 3\times 3}$. The tensor $\mathcal{T}$ is then reshaped into a vector $\text{vec}(\mathcal{T})$, and projected into a k-dimensional space using a linear transformation. The transformation is parametrized by the matrix $\mathbf{W}$ and matched with the tail entity $\mathbf{e}_t$ by an inner product. Algorithm 1 provides insides on the various stages to train the model on the UMLS training data in an unsupervised learning fashion. Note, for weight initialization Xavier Initialization[104] was used. The benefit of Xavier initialization over random initialization in neural networks is to prevent the neurons activation functions to not start in saturated or dead regions. Additionally, the training data was shuffled at every training epoch to help escape saddle-points, we found this is helpful since the gradients are then different in every epoch [103].
In order to optimize the models' performance to unseen data, with respect to robust generalization capabilities and prediction accuracy, precise hyperparameter tuning is required. We propose the following list of hyperparameters, which allows the model to fit the UMLS data accurately. We give explanations to all relevant hyperparameter required for model adjustment. Each section includes a short description, background information, plus theoretical aspects from literature:

[2]https://github.com/TimDettmers/ConvE

- **Embedding dimensions**: The embedding dimensions define the n-dimensional space in which the knowledge graph will be encoded, where n is directly related, the number of features (rows) in the embedding vectors for entities $\mathbf{e}$ and relations $\mathbf{r}$. For choosing the right embedding size, we need to think about the density of the points in the embedding space. Since we want to create a reasonably good separation between nodes that are semantically correlated and those not, we generally need a greater embedding space. We followed the general rule-of-thumb to determining the number of embedding dimension as follows [105]:

$$\texttt{embedding\_dimension} = \texttt{number\_of\_categories}^{0.25} \tag{5.9}$$

  However, there is no quantitative research that proofs a general rule for optimal embedding dimensions yet exists. According to research published on related topics, such as Word2Vec. Where a large corpus of words is embedded into low-dimensional space. They argue that the quality of word embedding increases with higher dimensionality, but it reaches a point where the marginal gain will diminish. They suggested that in practice, the dimensionality of the vectors should be set between 100 and $1,000$ [106]. According to the rule-of-thumb, our embedding dimension should be around 100, since we can increase the quality with a larger embedding space we use 200.

- **Epoch**: An epoch corresponds to iterating over the entire training dataset one time. This means that each sample in the training dataset has had an opportunity to update the internal model parameters. An epoch is comprised of multiple batches.

- **Mini-batch size**: The ConvE model is trained on mini-batches using stochastic gradient descent. The batch size controls the number of training samples to work through before the models' internal parameters are updated. A variety of factors generally drives Mini-batch sizes, and determining the ideal batch size for our model is a trade-off between performance, computation time, convergence, generalization, and hardware restrictions. The following analyses the most relevant aspects we considered to investigate an optimal batch size. On the one hand, large batch sizes provide an accurate estimate of the gradient. On the other hand, small batches can offer a regularizing effect due to the noise they add to the learning process. Generalization error, which describes the gap between the models' performance on the test,- and dev dataset, is often best at a batch size of 1 [107]. Training with such a small batch size might require a small learning rate to maintain stability and smooth out the high variance in the gradient estimates. In retrospect, a small learning rate can cause slow convergence properties, which leads to a large total runtime. To find a good trade-off between accuracy and generalization performance, we used a batch

size of 64. Additionally, we took the fact that GPUs achieve better runtime for batch sizes of power two, and according to literature, a typical batch sizes range from 32 to 256, with 16 sometimes being attempted for large models [6, 272].

- **Optimization parameter**: According to the review, on various optimization algorithms we conducted in chapter 4.4, we use Adam as an optimization algorithm. Adam belongs to the group of adaptive learning rate algorithms. The algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters beta1 and beta2 control the decay rates of these moving averages. A fairly low `learning rate` of $\alpha = 0.003$ was chosen, since this does not significantly affect computation time, but shows positive results in smooth gradient updates. The values for the `learning rate decay`, `beta1`, and `beta2` were adopted from the original paper, and are 1.0, $0,999$, and 0.9, respectively [3].

- **Regularization parameter**: The general idea behind regularization is to prevent the neural network to over-fit the training data. Regularization combines certain techniques that make slight modifications to the learning algorithm and allow the model to generalizes more reliable. This, in turn, improves the models' performance when evaluated on new data, e.g., test dataset. One technique that is widely adopted is called Dropout [108]. The idea of Dropout is to randomly shut off neurons, i.e., activation functions and their connections during training. It can be used with most types of layers, such as dense fully-connected layers, convolutional layers, and recurrent layers. It prevents units from co-adapting by creating randomly different network structures. Following previous work, Dropout was used in several stages during training [5]. In particular on the embeddings, the feature maps after the convolution operation, and the hidden units after the fully connected layer. The individual Dropout operations are marked in Figure 3.4.2 with crossed-out symptoms. Another regularization technique we adopted is label smoothing ϵ. It is a loss function modification that has proven to be very effective for training deep learning networks. It helps to prevent the model from predicting the training examples too confidently [109].

- **Convolution parameter**: We have two parameters that adjust the convolution properties, the kernel size, and num_out_channels. The kernel size refers to the filter size of the convolution layer (Chapter 4.2 for details). The filter dimensions depend on the input size and the number of features we want to capture in each convolution layer. The concatenated input matrix is relatively small, so a standard 3×3 filter is applied. The num_out_channels set the number of output channels for the convolution layer. Here we use the default

num_out_channels of 32 [27].

Algorithm 1: Training ConvE

Input: Training set $S = (e_h, r, e_t)$, hyper-parameter list
Output: trained model in .tar format

Initialize: $\mathbf{E}^{|\mathcal{E}|\times 200}$ and $\mathbf{R}^{|\mathcal{R}|\times 200} \leftarrow$ xavier initialization[104]
for *epochs* **do**
  $S_{batch} \leftarrow sample(S, 32)$ //sample a mini-batch of size 32
  $T_{batch} \leftarrow \emptyset$ //initialize the set of pairs of triplet
  for $(e_h, r, e_t) \in S_t$ **do**
    $(e_h, r, e_t) \leftarrow sample(S'_{(e_h,r,e_t)})$ *//sample a corrupted triplet*
    $T_{batch} \leftarrow T_{batch} \bigcup \Big((e_h, r), (e_h, r, e'_t) \Big)$ *//1-N scoring*
  Update weights w.r.t:

$$\sum_i \nabla(t \cdot log(p) + (1-t) \cdot log(1-p)), \ with$$

$$p = \texttt{ReLU}(vec(\texttt{ReLU}(concat(\overline{\mathbf{e}_h}, \overline{\mathbf{e}_t}) * \omega))\, W)\theta_o$$

–

5.3.3. Policy gradient optimization

We described the training of the reward shaping model successfully and focus on the RL-agent training throughout the following section. The general training process in RL is fundamentally different from the aforementioned unsupervised learning case. In reinforcement learning, the artificial agent explores the environment during training and maximizes the accumulative future reward. The environment responds with a positive reward of +1, whenever the agent stops at the correct answer node. In case the agent stops at an entity whos correctness is unknown, we use a soft reward computed via the pre-trained ConvE embedding model. This process is visualized in the lower half of Figure 5.5. The general objective is incrementally updating the parameters θ to maximize the expected reward over all queries in $\mathcal{G}$. This is defined by the following objective function $J(\theta)$:

$$J(\theta) = \mathbb{E}_{(e_h,r,e_t)\sim\mathcal{D}}[\mathbb{E}_{a_1,\ldots,a_T\sim\pi_\theta}[R(s_T|e_h, r)]] \tag{5.10}$$

The optimization is done using REINFORCE [80], which relies on an estimated

return by Monte-Carlo methods using episode role-outs to update the policy parameter θ. We iterate through all (e_h, r, e_t) triples in $\mathcal{G}$ and update the policy network parameters θ accordingly:

$$\theta_{new} \leftarrow \theta_{old} + \nabla J(\theta)$$

We optimize the objective function (Equ. 5.10) by assuming there is a true underlying distribution $(e_h, r, e_t) \sim \mathcal{D}$. This lets us replace the first expectation with an empirical average over the dataset. Thus, we approximate the second expectation by running multiple rollouts for each training sample. This holds, since the expectation of a random variable can mathematically be approximated by the arithmetic mean of equiprobable outcomes. Therefore, we perform 20 equiprobable rollouts for each training example. We than take the derivation of $J(\theta)$ (Equ. 4.18) to update θ. Considering the agent does not have access to any oracle path, it is possible to arrive at a correct answer via a path irrelevant to the query relation. This is a highly complex problem in which we need to discriminate paths of different qualities as the agent largely relies on the terminal reward to bias the search. In a conventional KG setup, there are usually more factitious paths than correct ones. Spurious paths are often found first, and the agent can progressively bias towards them. To compensate for this problem, we adopt the action dropout technique [39]. Action dropout is an effective regularization technique usually used in deep learning for reducing overfitting in neural networks by preventing complex co-adaptations on training data. We adopt this technique in a way to drop-out actions in the policy network. It uses a mask $\mathbf{m}$ to vanish some outgoing edges for the agent during the sampling step of REINFORCE. The mask $\mathbf{m}$ is a vector of binary variables $\mathbf{m} \in \{0, 1\}^{|A_t|}$ samples from a Bernoulli distribution. The resulting corrupted stochastic REINFORCE policy is denoted as $\tilde{\pi}_\theta(s_t|a_t)$, and is modeled in the upper half of Figure 5.5.
During one training batch, the action space $\mathcal{A}_t$ of different states (i.e., the set of outgoing edges of nodes $\mathbf{e}_t$) is arranged into a list of 10 actions (one bucket), based on their sizes to save the memory consumption caused by paddings. For example, certain nodes in the graph may have thousands of outgoing edges, while a long tail of nodes only contains a small number of outgoing edges. If a batch contains a node with 500 outgoing relations while the rest of the nodes have a maximum of five outgoing relations, we need to pad the action spaces of all nodes to 500, which causes a high memory consumption. With the bucketing approach, each bucket is padded separately. In this case, the node with 500 outgoing edges will be in its own bucket, and the rest of the nodes will not be affected by padding the action space to five. Once the action spaces are grouped in buckets, the policy network computation is carried out for every bucket iteratively. Once all computations are performed, we concatenate the results of all buckets and reconstruct their original order in the batch. The computation outside the policy network module is thus unchanged. The overall training approach is given in Algorithm 2. The essential hyperparameters, used for training the model, are grouped into Network Architecture, Optimization,

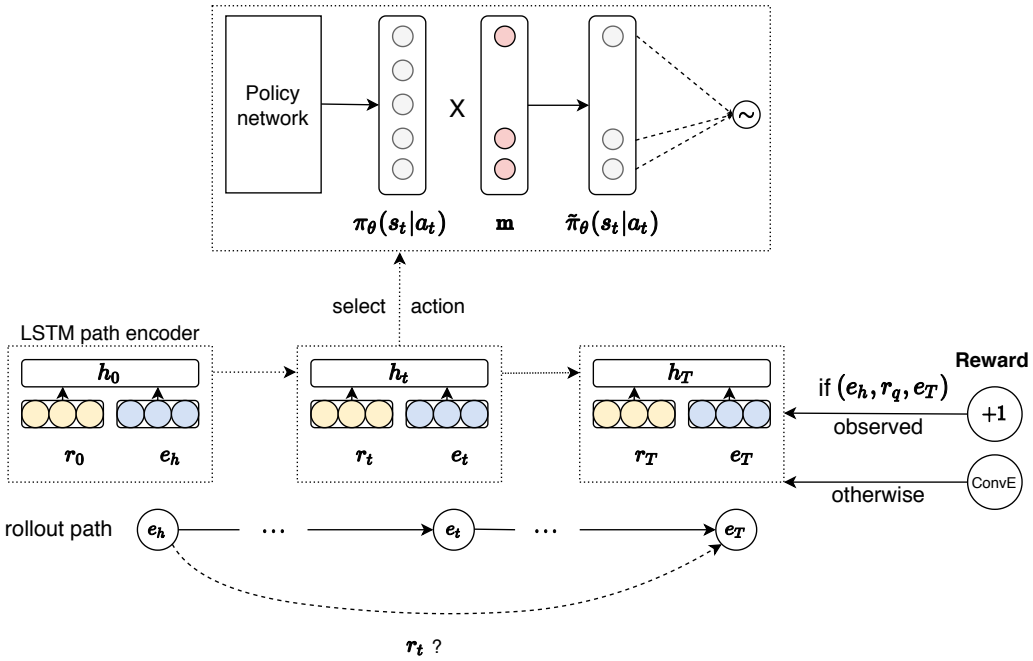

Figure 5.5.: A model of the overall training approach. At each time step t, the agent samples an outgoing link according to $\tilde{\pi}_\theta(a_t|s_t)$, which is the stochastic REINFORCE policy $\pi(a_t|s_t)$ perturbed by a random binary mask $\mathbf{m}$. The agent receives reward $+1$ if stopped at an observed answer of the query $(e_h, r_q, ?)$. Otherwise, it receives reward $f(e_h, r_q, e_T)$ estimated by the embedding-based subsystem.

and Reinforcement Learning. For the specific choices in hyperparameter, we used previous work [27, 26, 39] as a baseline and optimized the parameters for our needs, resulting in the following list:

Network Architecture Parameter

- `ff_dropout_rate` = 0.3: Dropout masks out connections in the fully connected feed-forward layer. This emulates a training with different architectures and prevents over-fitting.
- `entity_dimension` = 200: The entity vector has 200 features.
- `relation_dimension` = 200: The relation vector has 200 features.
- `history_dimension` = 200: The history vector has 200 features.
- `history_num_layers` = 3: In our model, we utilize a three-layer LSTM, enabling the agent to memorize and learn from the actions taken before.
- `action_dropout_rate` = 0.1: Dropout rate for randomly masking out knowledge graph edges.
- `action_dropout_anneal_factor` = 0.95: Decrease the action dropout rate once the dev set results stopped increase.

Optimization Parameter

- `epochs` = 500: Is the number of loops over the entire training set. Since reinforcement learning algorithms generally are very sample inefficient, resulting in a demand for a large number of interactions with the environment.
- `train_batch_size` = 128: The agent is optimized using Adam, a stochastic gradient decent algorithm, which trains the model in batches [3]. At each training step the agent is exposed to a set of training queries with a size of 128.
- `learning_rate` = 0.001: The learning rate is set fairly low maintain a slow and steady convergence property since we have a large number of training circles.
- `learning_rate_decay` = 1 : Together with the following two parameters, it is used to parameterize the Adam optimizer. With learning rate decay, the learning rate is calculated after each update (e.g., end of each mini-batch).
- `adam_beta1` = 0.9: Determines how much the first order gradient estimate affects the individual learning rate adaption.

- `adam_beta2` = 0.999: Determines how much the first order gradient estimate affects the individual learning rate adaption.
- `grad_norm` = 1000: This parameter sets the threshold for gradient clipping. Which is a technique to prevent exploding gradients, the gradient norms exceeding the value 1000 are scaled down to match the norm.

Reinforcement Learning Parameter

- `num_rollouts` = 20: The agent performs 20 equiprobable rollouts (random walks) for each training sample to update its policy function.
- `num_rollout_steps` = 2: We restrict the agent to perform a maximal number of two multi-hops for answering a query. Since we are in a biomedical link prediction setup, we found that the system is more robust when restricting to the local neighborhood of an entity instead of exploring a larger path of five steps.
- `bandwidth` = 300: Maximum number of outgoing edges to explore at each step.
- `beta` = 0.1: Is the entropy regularization weight and relates directly to the unpredictability of the actions an agent takes given the policy $\tilde{\pi}_\theta(a_t|s_t)$. The higher the entropy, the more randomness is in the action selection process. Beta tries to model the relation between exploitation and exploration, meaning the agent chooses a path that has already been determined through previous steps, or it chooses a completely unique path.
- `gamma` = 1: Is the discount factor for the accumulative reward. The larger the gamma, the smaller the discount. That indicates that the learning agent cares more about the long term reward. In retrospect, the lower the gamma, the higher the discount, meaning our agent cares more about the short term reward (the closes answer).

Algorithm 2: Training RL-agent

Input: Training set $S = (e_h, r, e_t)$, hyper-parameter list
for *episode* **do**
  Initialize LSTM path encoder
  Initialize $num\_steps = 0$
  Initialize state vector
  while $num\_steps < max\_steps$ **do**
    Randomly sample action $a \sim \tilde{\pi}_\theta(a_t|s_t)$
    Observe reward $\mathcal{R}_t$, next state s_{t+1}
    if $\mathcal{R}_t = -1$ **then**
      add $< s_t, a >$ to $\mathcal{M}_{neg}$
    else
      add $< s_t, a >$ to $\mathcal{M}_{pos}$
    if $success \vee num\_steps = max\_steps$ **then**
      break
    Increment num_steps
  Update θ using
  $g \propto \nabla_\theta \sum_{\mathcal{M}_{neg}} \log \pi(a = r_t|s_t; \theta)(-1)$
  if *success* **then**
    Update θ using
    $g \propto \nabla_\theta \sum_{\mathcal{M}_{pos}} \log \pi(a = r_t|s_t; \theta)(R_{total})$
  else
    $g \propto \nabla_\theta \sum_{\mathcal{M}_{pos}} \log \pi(a = r_t|s_t; \theta)(R_{ConvE})$

6. Experimental Evaluation

The previous chapter introduced the general framework, containing the RL model and the embedding model. Both systems can be seen as semi-independent during training. Although, the RL model relies on the soft reward predictions, estimated from the embedding model, to train accurately. Therefore, the overall system performance strongly relies on both individual elements. While training both models is a crucial step, evaluating model performance is equally important. This chapter introduces statistical measures for evaluating any process that produces a list of possible responses to a sample of queries, ordered by the probability of correctness. Hits@N Accuracy and Mean Reciprocal Rank two commonly used statistical measures evaluate the models' prediction accuracy, as well as generalization performance, to unseen data. Both ratios are calculated during training, which allows continuous evaluations after each episode. In the following, we first evaluate the performance of the ConvE embedding-based model on the UMLS KG. Subsequentially, we will train and assess the RL-based model and provide a detailed performance analysis.

6.1. Model Evaluation Protocol

Model evaluation metrics are obliged to quantify the systems' query answering capacities and measures how well the model captures the underlying semantics. We evaluated both techniques by performing link prediction on the UMLS dataset. We desire to answer queries in the form $(e_h, r_q, ?)$, where e_h is the source entity and r_q is the relation of interest. We convert each triple (e_h, r, e_t) from the test set into a query by removing e_t and compute ranking-based evaluation metrics. The models take (e_h, r) as input, and output a list of possible answers sorted by highest probabilities of correctness $E_t = [e^1, \ldots, e^n]$, where $(e_h, r_q, e^{1-n}) \notin \mathcal{G}$ due to incompleteness. To reliably evaluate the model prediction capabilities, two metrics are introduced. The Hit@1,3,10 accuracy and Mean Reciprocal Rank (MRR), which are proven metrics for KG completion [27, 110, 59].

6.1.1. Hits@N Accuracy

Hits@N accuracy is also known as top-N accuracy, which is a metrics commonly used in many classification tasks and recommendation engines. It measures how often the predicted class falls in the top-N values of the output (softmax distribution) and can

formally be defined in Equation 6.1.

$$\text{Hits@N} = \sum_{i,j,k=1}^{|\mathcal{G}|} 1 \text{ if } rank_{x_{ijk}} \leq N \tag{6.1}$$

where $\mathcal{G}$ is a set of test triples in the KB and facts are defined as $x_{ijk} \in \mathcal{G}$.
Let us consider the following example in Table 6.1. Given an input query in the form (e_h, r) the ConvE model outputs a list $E_t = [e^1, .., e^n]$ of possible answers each assigned with a confidence estimate. Answers with the highest probability are ranked first. Please note, the first relevant answers are considered, possible further fitting answers are restricted to five for simplicity.

Input (e_h, r)	Output E_t	Confidence Score	Rank
genetic_function, `affects`, ?	**organ_function**	**0.789**	**1**
	human	0.752	2
	bacterium	0.675	3
	biologic_function	0.454	4
	individual_behavior	0.223	5
diagnostic_procedure, `measures`, ?	cell_function	0.789	1
	clinical_attribute	0.752	2
	molecular_function	**0.675**	**3**
	anatomical_abnormality	0.454	4
	sign_or_symptom	0.223	5
cell, `part_of`, ?	bacterium	0.921	1
	genetic_function	0.345	2
	vitamin	0.288	3
	carbohydrate	0.201	4
	organism	**0.156**	**5**

Table 6.1.: Example predict of ConvE model on the UMLS test set givens three input queries (e_h, r). The model assigns score to all possible output entities. **Bold** indicates the test triple's true tail entity and *italics* other true tail entities present in the training set.

The Hits@1 accuracy can be compared to precision in a classification supervised learning setup, as it represents the models' prediction capacities against the true label. In our example, the first triple ranks 1st, the second ranks 3rd, and the third triple ranks 5th (against their respective synthetic negatives). We then count how many positives occur in the top-1, top-3 or top-5 position and divide by the number

of triples in the test set (in this example includes three triples). This results in a Hits@1= $\frac{1}{3} = 33\%$, Hits@3= $\frac{2}{3} = 75\%$, and Hits@5= $\frac{3}{3} = 100\%$, respectively. Note, an accuracy of 33% might seem unsatisfying. However, considering this is a unsupervised learning setup, where the true underlying tail-entity might not exist in the test set since we are more focused on generalization to unseen relationships. A high Hits@1 score might demonstrate that model has learned the training set, but it does not indicate a good generalization performance. Therefore, Hits@3,5,10 are equally relevant to evaluate overall performance. It provides a confidence interval within the true tail-entity and indicates the models' generalization performance [111].

6.1.2. Mean Reciprocal Rank

The reciprocal rank of a query response is the multiplicative inverse of the rank of the first correct tail-entity; assigning 1 for first place, $\frac{1}{2}$ for second place, $\frac{1}{3}$ for third place, respectively. The mean reciprocal rank is the average of the reciprocal ranks of results for a sample of queries x_{ijk} and can formally be defined by Equation 6.2:

$$\text{MRR} = \frac{1}{|\mathcal{G}|} \sum_{i,j,k=1}^{\mathcal{G}} \frac{1}{rank_{x_{ijk}}} \tag{6.2}$$

where $\mathcal{G}$ is a test set of triples and $x_{ijk} \in \mathcal{G}$ is a triple.
Referring back to the three sample queries in Table 6.1. In each case, the system makes three guesses, with the first being the one it thinks is most likely correct. We take the reciprocal rank of the true tail-entity for each guess, add them all up and divide by the total number of queries: $(\frac{1}{1} + \frac{1}{3} + \frac{1}{5})/3 = 0.51$. This results in the mean reciprocal rank of 51%. Normally, the higher the score, the better the model learned the knowledge graph. However, this is always a trade-off with generalization performance.

6.2. Hyperparameter Search

Both previously introduced statistical evaluation matrices, Hits@k, and MRR are flawlessly integrated in the training process. The model is trained for 150 epochs in n mini-batches (size 32). We save the individual loss after every mini-batch to calculate the average loss at the end of the epoch. The average loss is then used to update the models' parameter accordingly. Hence, we run the evaluation protocol to calculate the MRR by using the dev set after every second epoch. Figure 6.1 presents the training process with an MRR of 95.1% after 150 epochs. We see the loss function decreasing exponentially throughout the early training epochs. This characteristic is common when using Adam Optimizer. Adam leverages the power of adaptive learning rate methods to find individual learning rates for each parameter and allows large gradient steps during early parameter updates. In contrast to the

loss function, the MRR performance, is steadily increasing until it converges to 95%. This result demonstrates an accurate prediction performance achieved when the model is evaluated on the dev set.
After every second episode, we evaluate the Hits@1,3,5,10, appearing in Figure A.1, which we included in the appendix section. We reached a Hits@10 accuracy of 99% and a Hits@5 of 98%. These measurements conclude in an especially tight confidence interval. The model scores the true tail-entity 98% of all instances, within the first five output probabilities with the highest confidence scores. Furthermore, in 97% of the text cases, the true tail entity is even within the first three predicted answers. We summarized the hyperparameters used for our experiments in Table 6.3, we used the parameters proposed in the original ConvE paper[5] as a baseline.

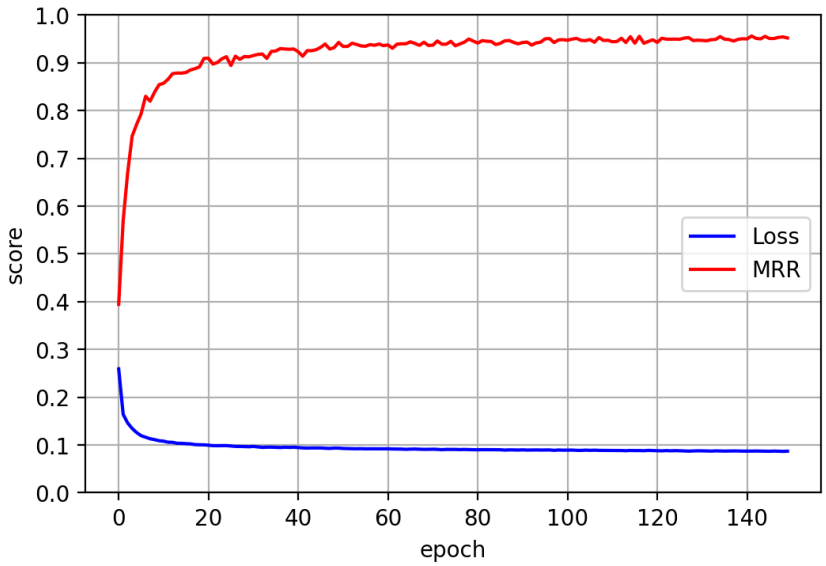

Figure 6.1.: Performance evaluations for 150 training epochs. The average loss (blue), calculated at the end of each epoch, shows fast convergence properties during the early updates. The MRR (red) calculated on the test set is improving rapidly during the first 20 epochs, from there on linearly with a small slope until it reaches the 95.2% within the last epoch.

Hyperparameter	Values
Evaluation Batch Size	64
Relation Dimensions	200
Training Batch Size	32
Embedding Dropout	0.3
Entity Dimension	200
Label Smoothing	0.1
Learning Rate	0.003
Hidden Dropout	0.3
Kernel Size	3
Beta1	0.9
Beta2	0.999

Table 6.2.: Summary of Hyperparameters used for the following experiments to investigate an optimal training process with fast convergence properties. Note, a detailed description on each individual parameter can be found in Chapter 5.3.2.

6.2.1. Embedding Dimensions

The following subsection will investigate potential relationships between the embedding dimensions for the entities and relationships with respect to the accuracy, training speed, and scalability. Note, this is a crucial investigation concerning real-world applications where new concepts are regularly added to the knowledge graph. This task is time-consuming plus directly relates to computational cost, which we seek to evaluate for our specific model. The UMLS knowledge graph utilized for our experiments is a considerably small KG compared with millions of entities in the GO Ontology, therefore scaling is a relevant constraint for further investigations. The results are represented in Table 6.3. The model was trained for 150 epochs throughout the experiment.
According to the models' space complexity $\mathcal{O}(n_e k + n_r k)$, where n_e is the number of entities and n_r the number of relations, the computation clock time scales almost linearly with the embedding dimension and amount of parameters, respectively. However, we noticed a performance drop in the MRR evaluation, which might indicate the model to overfit. This happens when a model learns details and noise in the training data to such an extent that it negatively impacts the models' performance on unseen data. This means that the noise or random fluctuations in the training data is picked up and learned as features by the model. The problem is that these concepts do not apply to new data and negatively impact the models' ability to generalize.
We could elevate this problem by increasing the regularization parameters, such as

Embedding dim.	num. parameters	Time in s	MRR in %
200	2.1M	38	94.6
300	4.9M	43	95.6
400	8.8M	52	94.8
500	14.4M	67	93.8
600	20.5M	81	92.5
700	28.2M	103	91.1
800	37.1M	125	90.7

Table 6.3.: Experimental results to evaluate the correlations between the embedding dimension, number of parameters, computation clock time, and MRR. According to the thesis in [6]; if the embedding dimension is increased, the model can capture more relevant features, which should result in an accuracy gain. However, we investigated this holds only partly true for our model. We notice a linear MRR increase until we reach some threshold; in our experiments, the threshold is between $500 - 600$. Everything above 600 feature dimensions does not result in increased accuracy.

the dropout intensity throughout multiple layers. Another factor that could prevent the model from overfitting is a decreased learning rate. We tried different variations, throughout our experiment, and concluded in an optimal embedding dimension of 250.

6.3. Policy Gradient

For evaluating the policy gradient, we follow the same metrics mentioned earlier. Note, it is not trivial to assess the two models under similar criteria since we compare two fundamentally different methods. Both are having benefits and limitations, which will be further analyzed in the next chapter. This section will present the evaluation results that we later use to drive further investigations.
Following literature [39], we trained the RL agent for 500 epochs with a batch size of 128 and 20 rollouts for each query (e_h, r, e_t). The agent performs 20 random walks to search for the true tail-entity. We limited the agent to perform two multi-hops to obtain an answer. We found this is helpful since we interact with a biomedical knowledge graph we do not want the agent to explore in a context where inferences might be too abstract or simply not traceable. It is beneficial to focus the agents' attention on the close head-entity e_h, r neighborhood. For regularization, we used beta and dropout techniques, proposed by [39]. The beta term scales the cost function (Equ. 4.3) by 0.1. This encourages diversity in the paths sampled by the policy [27]. Dropout techniques used for the linear fully-connected layer is set to 0.1,

Hyperparameter	Values
Epoch	500
Bandwidth	400
Entity dimensions	200
Relation dimensions	200
History dimension	200
Num. rolloutos	20
Multi-hop steps	2
Batch size	128
Train batch size	32
Learning rate	0.001
Learning rate decay	1
Action dropout rate	0.3
Ff dropout rate	0.2
Beta	0.01
Adam beta1	0.9
Adam beta2	0.999
Grad norm	1000
Gamma	1
Beam size	128

Table 6.4.: Hyperparameters used for training the RL agent

and the action dropout rate is set to 0.1. Further hyperparameters used to adjust the training are summarized in Table 6.4. Please refer to Chapter 5.3.3 for further hyperparameter descriptions .
The evaluation result on Hits@k and MRR are presented in Figure A.2, included in the appendix. The RL agent reaches a Hits@1$\approx$ 80%, Hits@3$\approx$ 92, 7%, and Hits@10 close to 97%. For the mean reciprocal rank, it scored approximately 86% after 500 training epochs.

7. Discussion

So far we trained both models independently and simultaneously evaluated the model performance on the dev set, calculated for every second episode. In this chapter we will further investigate the deep-medical reasoning system in respect to our initial research questions.

7.1. Overview Experimental Results

In the previous chapter, we trained the embedding-based model ConvE, that uses 2D convolution over embeddings and multiple layers of non-linear features to model the knowledge graphs. As well as a path-based reinforcement learning agent on the UMLS graph, respectively. We evaluated both models on a deep-medical reasoning task, which comprises a link predictions on the UMLS KG. We convert each triple (e_h, r, e_t) in the test set into a query and the models take e_h, r as the input to output a list of candidate answers $E_o = [e_1, ..., e_L]$ ranked in decreasing order of confidence score. The models performances were evaluated using two statistical measures, mean reciprocal rank (MRR) and Hits@1,3,5,10, which are standard metricise for recommendation systems. The evaluation scores mean and standard deviation across multiple runs are presented in Table 7.1. We trained the ConvE model for 150 epochs and 500 epochs for the RL agent.

7.2. Comparison Analysis

In this analysis section, we will primarily focus on answering the first research question in its full spectrum. We will compare the models' performance under two main aspects, the general accuracy, which provides insights on the models' ability to cap-

Measures	ConvE	RL agent
MRR	95 % ± 2 %	84.3 % ± 3.1 %
Hits@1	92 % ± 0.9 %	78 % ± 2.9 %
Hits@10	99 % ± 0.09 %	96.4 % ± 0.03 %

Table 7.1.: Accuracy evaluation metrics mean and standard deviation across multiple runs for the ConvE model (150 epochs each run) and for the RL agent (1000 epochs each run).

ture semantic similarities in the graph as well as the computational cost at training and inference time. The accuracy measure is essential to conclude how efficient the models learn facts and captures the underlying structure between entities in the graph. Additionally, we evaluate the training and inference time, which are two relevant dimensions for industry implementations.

Accuracy: The neural embedding-based approach achieves promising results on biomedical prediction tasks. The mean and standard deviation for MRR is 95 % ± 2 % symbolizes the model ranks the predicted answer in the test set precisely to the correct answer. Also, a Hits@1 of 92 % ± 0.9 % states that the model was able to score the right answer with the highest probability in 92 % ± 0.9 % of all test queries. However, in close comparison to the RL agent, we notice a significant performance gap of approximately −10%. The agent achieved, after 500 training epochs, a maximal MRR of 84.3 % ± 3.1 % and a Hits@1 of 78 % ± 2.9 %. Only the Hits@10 performances are equally good.

Training time: We noticed that the computational cost of ConvE is comparably low. We used an embedding size of 200 for entities and relations, which results in 5×10^6 total parameters. In comparison, the RL agent policy network is slightly lighter, with $3,7 \times 10^6$ parameters. In terms of total computation time on an NVIDIA Geforce-GTX-1080 GPU with 10 Gigabyte storage, it took 101s to compute 500 ConvE training epochs and 3.4 hours to compute 500 epochs on the RL agent. Note, our experiments investigated with the right hyperparameter tuning it is unnecessary to train the ConvE model for such a large number of epochs. We were able to achieve slightly better mean reciprocal rank results with 150 training epochs, compared to [5, 27]. The 101s of total computation time is used for equal comparisons, and demonstrate the computation overhead on the RL approach since the training of 500 epochs took 3.4 hours, with is 121 times more.

Inference time: At test time, embedding-based model ConvE ranks all entities in the graph, resulting in a test time proportional to $\mathcal{O}(|\mathcal{G}|)$ where $\mathcal{G}$ denotes the set of entities plus relationships in the graph. On the other hand, the RL approach is efficient at inference time since it has to search for answer entities in its local neighborhood mainly. The main cost of inference time for the agent is to compute probabilities for all outgoing edges along the path, depending only on the degree of outgoing relationships of nodes in the close neighborhood region. This number is still considerably below calculating probabilities for all entities in the graph. The wall-clock inference time on the UMLS test set with a GPU implementation of ConvE is 130s, whereas for multi-hop RL is 35s, which is 3.7 times less.

Concluding on the first research question, we investigated that the performance

gap between embedding-based approaches and multi-hop RL is still considerably large ($\approx 10\%$). The ConvE model proved excellent results on deep-medical reasoning tasks. However, we will investigate some limitations that need to be encountered when using a neural embedding approach for deep medical reasoning. Furthermore, we argue that the symbolic multi-hop approach is the right research direction towards explainable AI when dealing with complex real-world link-prediction tasks. We start to elaborate on our argumentation by pointing out some specific limitations and constraints in embedding-based learning. We then discuss how the symbolic RL approach helps to overcome some of these issues.

7.3. Limitations Graph Embedding

This section will investigate the limitations of neural embedding-based link prediction models regarding the third research question. Current neural graph reasoning methods encounter four challenges as follows:

(1) Their performance is often sensitive to the sparsity and completeness of the graph—missing edges (i.e., potential false positives) make it harder to find evidential paths reaching target entities. As we investigated in the data acquisition (Section 2.2), the density of the graph is recommended to be $\geq 3\%$. In addition, hierarchically structured KGs with a high degree of 1-Many relationships for each node, are not ideal. For learning sophisticated graph representations, current methods rely on contrasting positive instances with negative ones. Unfortunately, the model is only exposed to positive triplets from the training data set. For calculating the knowledge embedding we need negative samples to minimize a margin-based ranking loss. During training, we corrupt facts by uniformly replacing tail-entities to augment the data. To then minimize a margin-based ranking loss, separating the positive triplets' scores from the generated negative ones by a fixed margin. In critical healthcare applications, this could lead to significant accuracy drawbacks, because to some extent, we generate false positives. That are in theory valid facts but due to graph incompleteness not included in the training data. The model would explicitly adjust the scoring function to minimize the probability of false positives. For illustration, consider the example as follows; we want to train the model a fact ⟨kidney, is_part_of, body⟩. To compute a margin-based ranking loss, we augment the triple by randomly changing the tail entity with other entities in the graph. This could result in the following negative training items: ⟨kidney, is_part_of, brain⟩, ⟨kidney, is_part_of, gene⟩, ⟨kidney, is_part_of, arm⟩, ⟨kidney, is_part_of, human⟩. Here the last fact is certainly a false positive because kidneys are indeed a part of humans. But during the training process, the model would treat the augmented triples as equally negative examples and learns the scoring function to maximize the margin between true triple ⟨kidney, is_part_of, body⟩ and all negative examples in-

cluding the false positive ⟨kidney, is_part_of, human⟩. The probabilities for hitting a false positive are considerably low and is dependent on the negative sampling strategy used [112]. To avoid such adversarial embedding solutions, the set of negative samples, are required to be curated or supervised by clinicians and professional doctors. Considering that maximize accuracy is demanded in healthcare applications, and misclassifications are unacceptable.

(2) The model assumes the graph is static and provides no online settings to adapt to dynamically enriched graphs where emerging new facts are constantly added. Each update to the graph requires retraining the model. Depending on the size of the graph this leads to continuous computational cost.

(3) In terms of scalability, the complexity of the model scales linearly with the number of nodes and edges in a knowledge graph ($\mathcal{O}(\mathcal{G})$). For example, a model with two billion nodes and 100 embedding parameters per node (expressed in float numbers) would require 800GB of memory just to store its parameters, thus several standard embedding methods exceed the memory capacity of typical server infrastructure. Recently published work by Facebook released an open-source framework to partition the whole graph into buckets and learn each bucked individually with distributed execution [25]. That way embedding-based methods can scale to large graphs

(4) Neural embedding models can be seen as a black-box model. Without sophisticated possibilities to explain why it provides good results or ways to apply modifications in the case of misclassification. Despite some recent work on visualizing high-level features by using the weight filters in a CNN [113], the neural embedding approach is often not interpretable. The ConvE embedding model learns vector representations to encode the latent meaning of the entity/relationship in the graph. One way to inspect at least the learned embeddings is by visualizing the embeddings projected in the vector space. Similar entities tend to be close in the vector space. But, final predictions are still challenging to explain because the result comes from the combination of latent features in a 200 dimensional vector space.

Concluding on the restrictions mentioned so far. They heavily limits the applicability of decision-making systems without a human in the loop. Particularly in healthcare applications, predictions made by a deep-medical system are required to be interpretable, and misclassifications need to be individually corrected. Nowadays, the only way to fix a misclassification in a deep learning model is by retraining the model on dozens of counterexamples and hoping for the best. Whereas, creating these large counterexamples sets might, in some cases, not even be possible or

at least very expensive since clinic professionals would manually label the training data. This lack of interpretability limits embedding-based approaches, especially in healthcare. Therefore, we argue that the neural-symbolic methods which directly predict links based on the existing graph structure offer huge potentials to future healthcare reasoning. These models directly use the knowledge graph structure as a ground trough and combine logical and symbolic concepts to reason. The deep-reinforcement learning approach can be seen as a neural-symbolic method under some constraints since it combines neural networks and fundamental ideas in symbolic AI. Current embedding methods fall into the category of narrow-AI. They are good at representation learning, which solves graph complexity by embedding large graphs into low dimensional space and train a simple scoring function to evaluate triples. As we showed earlier, the embedding model has proven high accuracy at this narrow task. However, these models are reaching a saturation point because they are not applicable to a broader context. The latest research published on neural graph embeddings is mostly centered on designing different scoring functions to match triples with a richer semantically representation [41] or explore new negative sampling techniques to augment negative triples resulting in a richer training set [114]. But what all these methods have in common, they do not solve the essential problem, which is the general limit to a domain-specific objective. To allow these techniques to transform from narrow-AI into broad-AI, we need to incorporate symbolic concepts. We argue that this is an essential step to build comprehensive multi-modal reasoning systems. In the following section, we explain that the reinforcement learning approach provides a basis to shift to broad-AI.

7.4. Multi-Hop Reasoning and Explainability

In this section, we aim to answer the second research question by further investigating the characteristics of multi-hop RL reasoning as a step towards hybrid neural-symbolic reasoning, which ultimately can enhance the explainability of the reasoning process. For qualitative results on the agents' path-finding performance, we use the pre-trained agent to retrieve correct answers from the test set. The agent is given a dev set query in the form $(e_h, r, ?)$ and performs five roll-outs to find the answer node. In a first execution we aim to answer the query $\langle$`antibiotics`, `affects`, ?$\rangle$. The agent starts at the given head-entity `antibiotic` and chooses paths in the graph according to the state-action probability distribution $\tilde{\pi}_\theta$. We restricted the agent to paths with a maximum length of three hops. The first five paths sampled by the agent are presented in Figure A.3, which can be found in the appendix section. The diagram shows the paths the agent took to reach multiple tail-entity predictions. Thurs, we printed the models' confidence score (blue) at the end of each run. The confidence score represents the likelihood in which the proposed entity answers the initial query. Note, the true tail-entity

`cell_or_molecular_dysfunction` (red dot) is exclusively found in the dev set. The agent took the following path to answer the query: `neoplastic_process` $\xrightarrow{\texttt{complicates}}$ `cell_or_molecular_dysfunction` $\xrightarrow{\texttt{<null>}}$ `cell_or_molecular_dysfunction`. This example demonstrates that the agent makes use of the self-loop (null pointer). Here the correct answer required only one step of reasoning. The agent reaches the answer within the first step and remained at the correct answer node.
In a second example A.4, which is also included in the appendix, we tried to answer the query (`neoplastic_process`, `process_of`, ?). The agent sampled the path: `neoplastic_process` $\xrightarrow{\texttt{process\_of}}$ `cell_or_molecular_dysfunction` $\xrightarrow{\texttt{process\_of}}$ `molecular_function`, which leads to the true underlying tail entity with a 0.74 coincidence score. In this example we are interested in the closely related paths. One closely related path, with a confidence score of 0.83, is the following: `antibiotic` $\xrightarrow{\texttt{process\_of}}$ `cell_or_molecular_dysfunction` $\xrightarrow{\texttt{process\_of}}$ `genetic_function`. This reasoning path is interesting to analyze because it derives a close related answer which is very likely to be correct. This case has shown that the agent can learn general rules. Case (b) in Table 7.2 provides examples on rules learned by the agent.

(a) **The agent can circulate on answer nodes:**
antibiotic $\xrightarrow{part\_of}$ cell or molecular dysfunction $\xrightarrow{Null}$ cell or molecular dysfunction
(b) **The agent learn general rules:**
process_of(X,Y) $\leftarrow$ process_of(X,Z) & process_of(Z,Y) process_of(X,Y) $\leftarrow$ process_of(X,Z) & affects(Z,Y)

Table 7.2.: Two example of paths found by the agent on the UMLS knowledge graph. The agent can learn general rules (example (b)) and learns shorter paths if necessary (example (a)) by using the self-loop

Overall, the experiments provide insights on the agents' ability to derive an answer. Specifically, the path it has chosen to retrieve the tail-entity. This is a significant advantage over embedding approaches, where no kind of teachable insights are derivable. However, we are still limited to black-box probability estimations through the policy network. The network is comprised of LSTM cells and multiple feed-forward components. Again, these deep-learning typologies do not support information mining on specific factors that lead to a particular action a_t in a state s_t, not to mention that it is still an open research question to change the probability

distributions for specific parameters encapsulated. Given the tedious process of negative sampling. We can conclude on the second research question, the path-based approach does increase the explainability because answers can directly be inferred on the graph structure. However, we see similar issues, compared with neural embedding methods, in cases of missclassification.

8. Conclusion and Future Directions

8.1. Summary

Two novel approaches for automated reasoning in biomedical knowledge graphs, a neural embedding approach [5] and a path-based reinforcement learning method [39] were deployed. Both systems were trained on the UMLS knowledge graph. A deep-medical reasoning task was set up to evaluate the models performance. The neural graph embedding model, achieved high accuracy scores using standard link prediction evaluation protocols. However, two significant drawbacks where investigated. During training, most embedding models minimize a margin-based ranking loss by randomly corrupting triples in the KG, which can lead to false-positive training examples. Ultimately, this results in random misclassifications and limit the applicability in critical health care domains. Counterexamples, used to compensate errors, is a labour intensive process and must be augmented by clinic professionals. We found that the path-based multi-hop approach overcomes these limitations in a way, such as providing tractability in the decision it makes. Predicted answers can be directly investigated by evaluating the policy function over the agents' history nodes. We conducted a qualitative analysis where the agent predicts possible answers by 'walking' on the graph structure. This can contribute to future health care implementations. Where current neural methods stagnated due to the lack of transparency in the decisions they make. The agent directly navigates on the underlying graph structure and can be seen as a step towards enabling explainable AI [115].

8.2. Future Directions

Reinforcement learning has great potential to be applied in a broader, more complex clinical decision-making scenario. This branch of research is relatively new to the KG question answering field and we are excited to see further development in path-based reasoning. The RL setup offers a flexible framework and can be seen as a step towards explainable AI. This is a crucial factor for real-world health care adoptions. The fact that biomedical knowledge graphs are becoming an increasingly popular tool to represent information extracted from biomedical research, results in great potentials for reasoning systems in the future. As a fundamental step towards personalized automated reasoning systems, which will, to an increasing degree, support medical diagnostic practices in the long run. Nevertheless, we believe that an algorithm of any kind will not entirely supplant the power of human reasoning, nor do we think a computer can replace the intimate and essential relationship between a doctor and their patient. Instead, we argue that the kind of deep-medical link prediction systems, investigated in this thesis, can be applied to use-cases, such as predicting links between individual health data and potential disease. This provides assumptions on possible diseases to allocate actions in advance. That can prevent the patient from getting sick. Further, these systems can play an essential role when dealing with rare diseases since they are not biased to population distributions.

A. Appendix

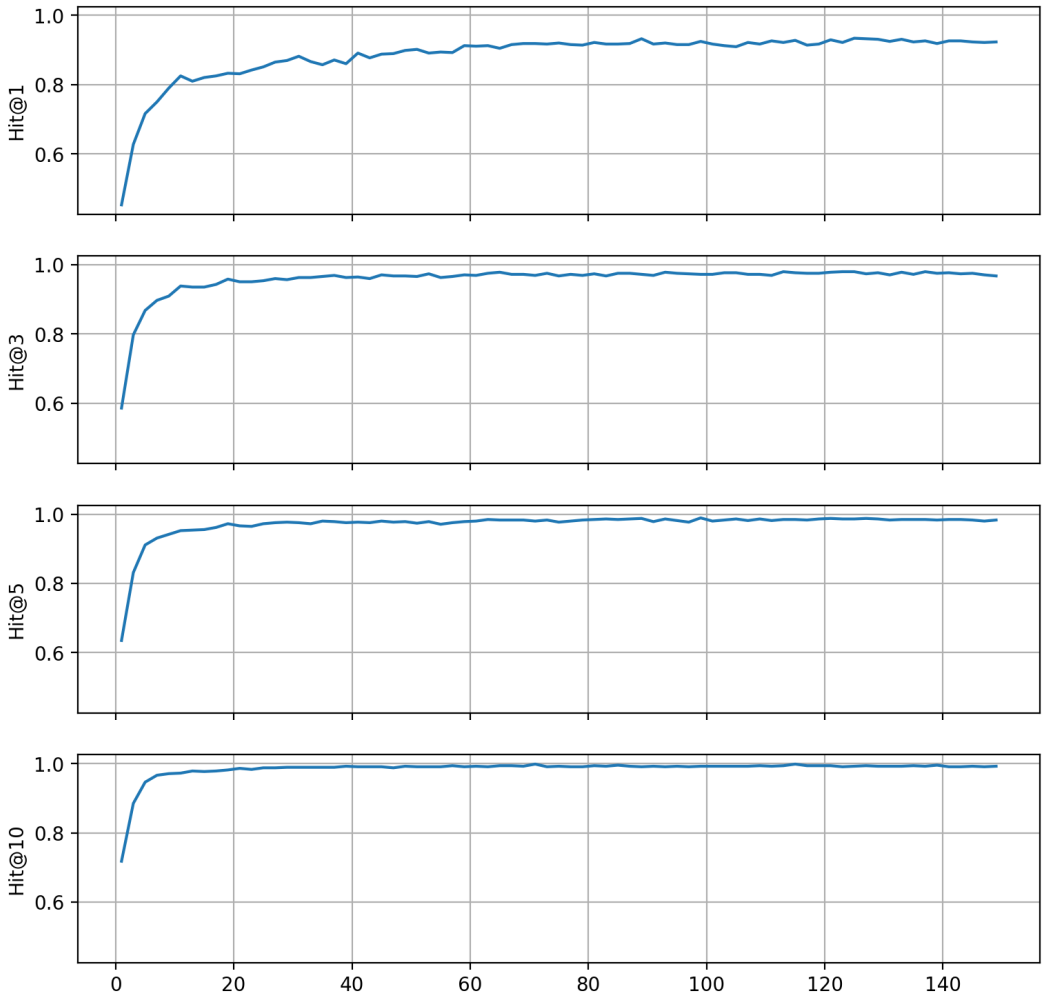

Figure A.1.: Model query answering evaluation after 150 training epochs.

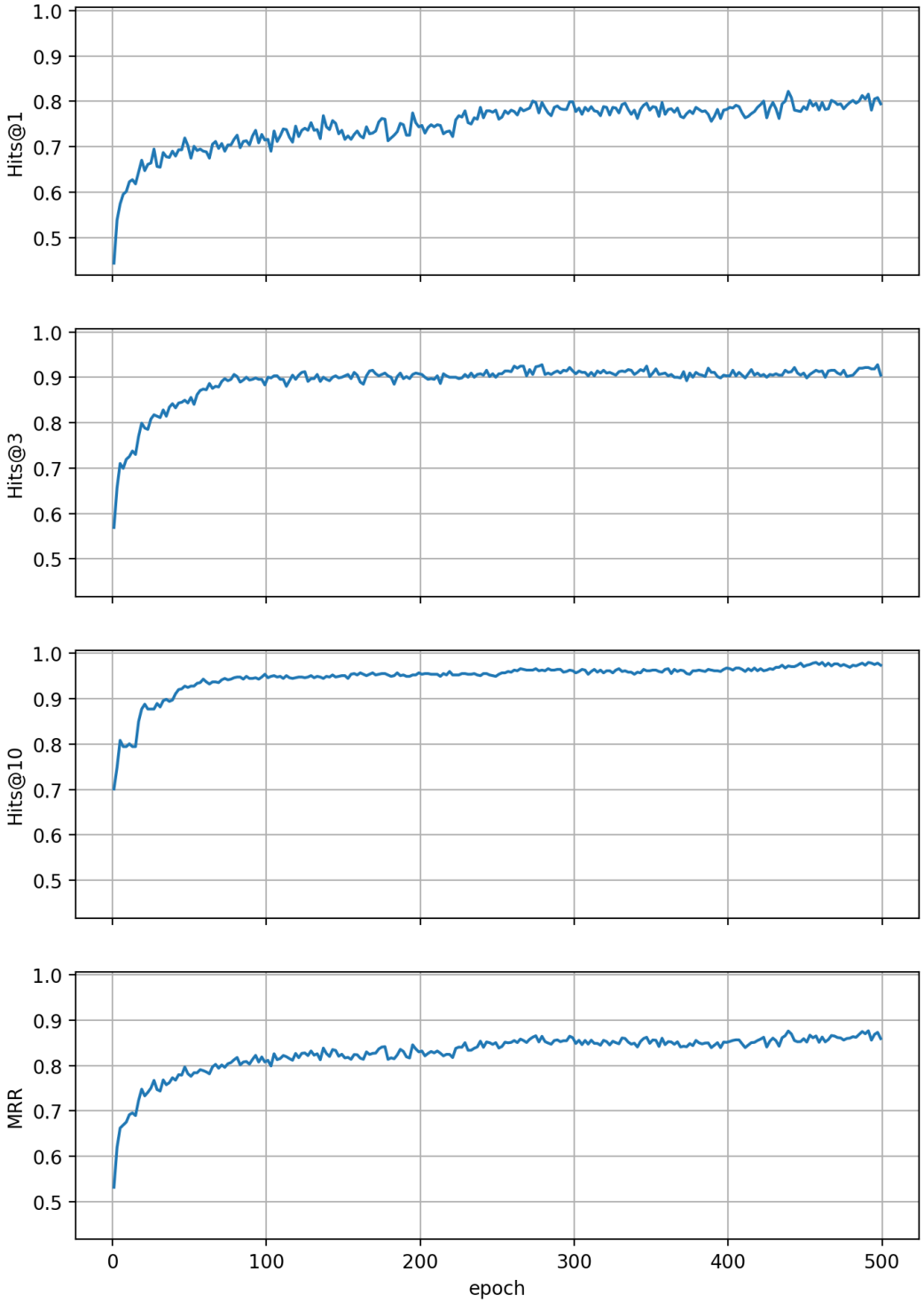

Figure A.2.: Evaluation results for the reinforcement learning path-based reasoner after 500 epochs, respectively.

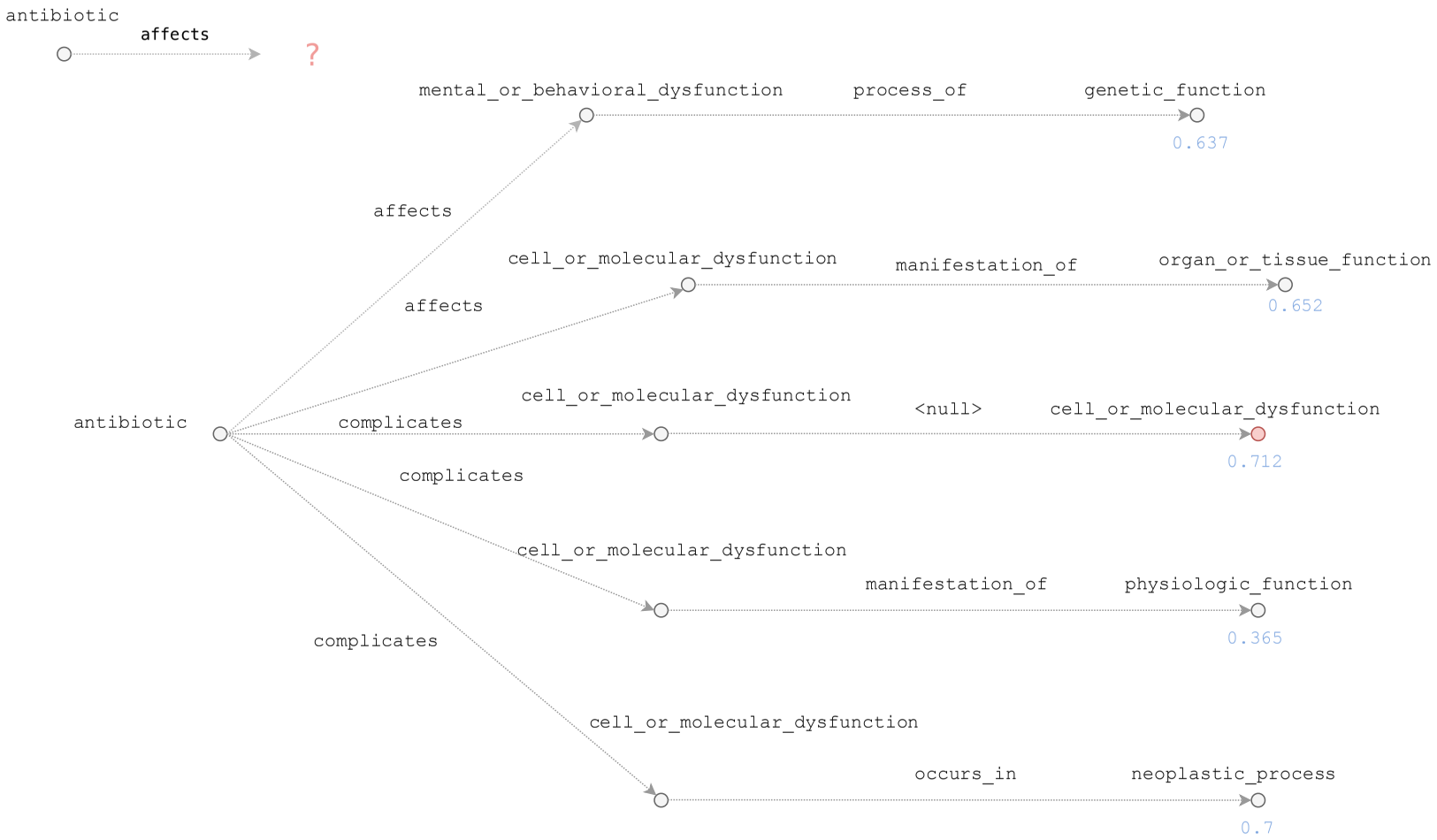

Figure A.3.: Five Policy roll-outs providing answers to the query (**antibiotic**, **affects**,?) executed on the test set.

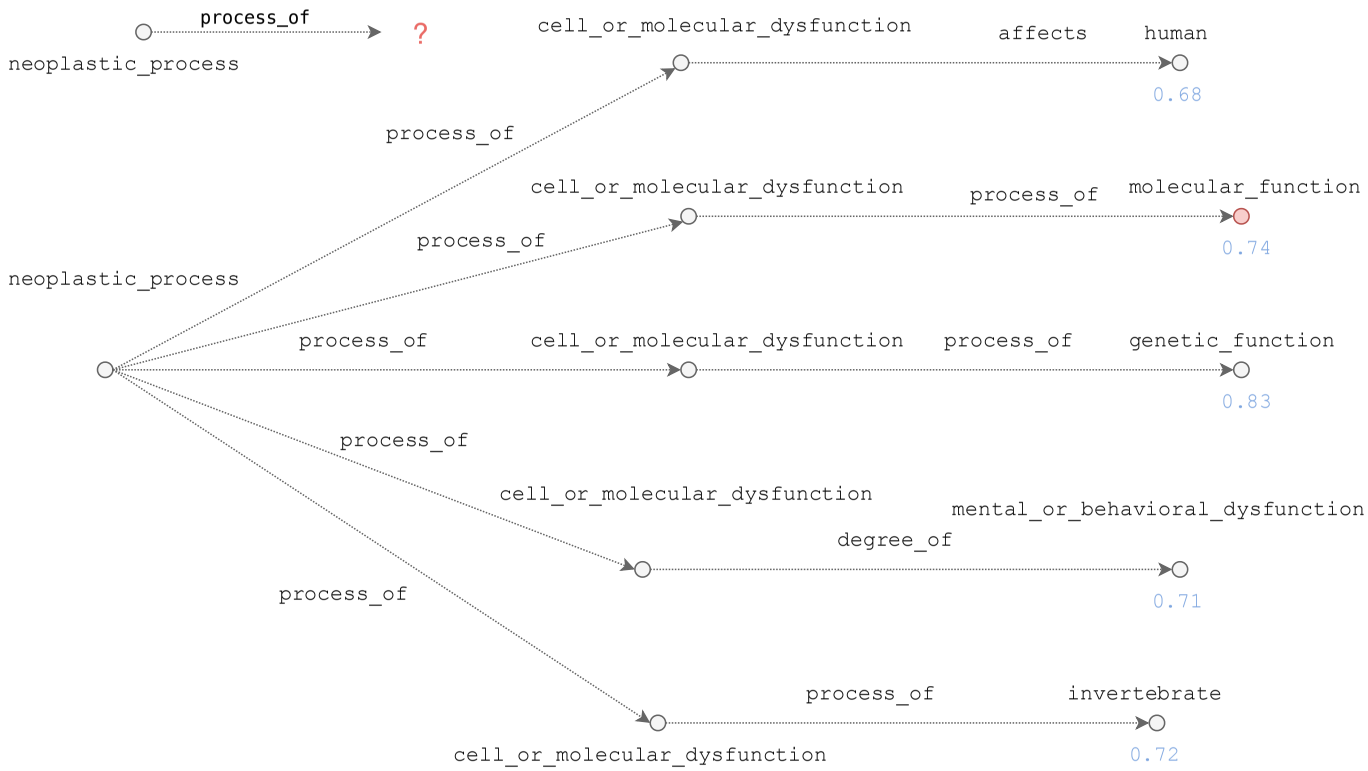

Figure A.4.: Five Policy roll-outs providing answers to the query (`neoplastic_process` , `process_of`,?) executed on the test set.

Bibliography

[1] Olivier Bodenreider. The unified medical language system (umls): integrating biomedical terminology. *Nucleic acids research*, 32:D267–D270, 2004.

[2] Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A three-way model for collective learning on multi-relational data. In *Icml*, volume 11, pages 809–816, 2011.

[3] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980*, 2014.

[4] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[5] Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. Convolutional 2d knowledge graph embeddings. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[7] John Paparrizos, Ryen W White, and Eric Horvitz. Screening for pancreatic adenocarcinoma using signals from web search logs: Feasibility study and results. *Journal of Oncology Practice*, 12(8):737–744, 2016.

[8] Ryen W White and Eric Horvitz. Cyberchondria: studies of the escalation of medical concerns in web search. *ACM Transactions on Information Systems (TOIS)*, 27(4):1–37, 2009.

[9] Jerome Groopman. *How doctors think*. Houghton Mifflin Harcourt, 2008.

[10] RA Miller and FE Masarie. Use of the quick medical reference (qmr) program as a tool for medical education. *Methods of information in medicine*, 28(04):340–345, 1989.

[11] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542(7639):115–118, 2017.

[12] Pranav Rajpurkar, Jeremy Irvin, Kaylie Zhu, Brandon Yang, Hershel Mehta, Tony Duan, Daisy Ding, Aarti Bagul, Curtis Langlotz, Katie Shpanskaya, et al. Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning. *arXiv:1711.05225*, 2017.

[13] Daniel Shu Wei Ting, Carol Yim-Lui Cheung, Gilbert Lim, Gavin Siew Wei Tan, Nguyen D Quang, Alfred Gan, Haslina Hamzah, Renata Garcia-Franco, Ian Yew San Yeo, Shu Yen Lee, et al. Development and validation of a deep learning system for diabetic retinopathy and related eye diseases using retinal images from multiethnic populations with diabetes. *Jama*, 318(22):2211–2223, 2017.

[14] Adam Lally, Sugato Bagchi, Michael A Barborak, David W Buchanan, Jennifer Chu-Carroll, David A Ferrucci, Michael R Glass, Aditya Kalyanpur, Erik T Mueller, J William Murdock, et al. Watsonpaths: scenario-based question answering and inference over unstructured information. *AI Magazine*, 38(2):59–76, 2017.

[15] Leslie J Bisson, Jorden T Komm, Geoffrey A Bernas, Marc S Fineberg, John M Marzo, Michael A Rauh, Robert J Smolinski, and William M Wind. Accuracy of a computer-based diagnostic program for ambulatory patients with knee pain. *The American journal of sports medicine*, 42(10):2371–2376, 2014.

[16] Yijia Zhang, Hongfei Lin, Zhihao Yang, Jian Wang, Shaowu Zhang, Yuanyuan Sun, and Liang Yang. A hybrid model based on neural networks for biomedical relation extraction. *Journal of biomedical informatics*, 81:83–92, 2018.

[17] Marc V Singleton, Stephen L Guthery, Karl V Voelkerding, Karin Chen, Brett Kennedy, Rebecca L Margraf, Jacob Durtschi, Karen Eilbeck, Martin G Reese, Lynn B Jorde, et al. Phevor combines multiple biomedical ontologies for accurate identification of disease-causing alleles in single individuals and small nuclear families. *The American Journal of Human Genetics*, 94(4):599–610, 2014.

[18] Manolis Kyriakakis, Ion Androutsopoulos, Artur Saudabayev, et al. Transfer learning for causal sentence detection. *arXiv:1906.07544*, 2019.

[19] Manolis Kyriakakis, Ion Androutsopoulos, Artur Saudabayev, et al. Transfer learning for causal sentence detection. *arXiv:1906.07544*, 2019.

[20] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and Philip S Yu. A survey on knowledge graphs: Representation, acquisition and applications. *arXiv preprint arXiv:2002.00388*, 2020.

[21] Assaf Gottlieb, Gideon Y Stein, Eytan Ruppin, and Roded Sharan. Predict: a method for inferring novel drug indications with application to personalized medicine. *Molecular systems biology*, 7(1), 2011.

[22] Wen Zhang, Xiang Yue, Guifeng Tang, Wenjian Wu, Feng Huang, and Xining Zhang. Sfpel-lpi: sequence-based feature projection ensemble learning for predicting lncrna-protein interactions. *PLoS computational biology*, 14(12):e1006616, 2018.

[23] Maya Rotmensch, Yoni Halpern, Abdulhakim Tlimat, Steven Horng, and David Sontag. Learning a health knowledge graph from electronic medical records. *Scientific reports*, 7(1):1–11, 2017.

[24] Dat Quoc Nguyen. An overview of embedding models of entities and relationships for knowledge base completion. *arXiv preprint arXiv:1703.08098*, 2017.

[25] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. Pytorch-biggraph: A large-scale graph embedding system. *arXiv:1903.12287*, 2019.

[26] Heng Wang, Shuangyin Li, Rong Pan, and Mingzhi Mao. Incorporating graph attention mechanism into knowledge graph reasoning based on deep reinforcement learning. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2623–2631, 2019.

[27] Rajarshi Das, Shehzaad Dhuliawala, Manzil Zaheer, Luke Vilnis, Ishan Durugkar, Akshay Krishnamurthy, Alex Smola, and Andrew McCallum. Go for a walk and arrive at the answer: Reasoning over paths in knowledge bases using reinforcement learning. *arXiv:1711.05851*, 2017.

[28] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

[29] Wenhan Xiong, Thien Hoang, and William Yang Wang. Deeppath: A reinforcement learning method for knowledge graph reasoning. *arXiv:1707.06690*, 2017.

[30] Yelong Shen, Jianshu Chen, Po-Sen Huang, Yuqing Guo, and Jianfeng Gao. Reinforcewalk: Learning to walk in graph with monte carlo tree search. 2018.

[31] Manuel Carlos Díaz-Galiano, María Teresa Martín-Valdivia, and LA Ureña-López. Query expansion with a medical ontology to improve a multimodal information retrieval system. *Computers in biology and medicine*, 39(4):396–403, 2009.

[32] Bogumil M Konopka. Biomedical ontologies—a review. *Biocybernetics and Biomedical Engineering*, 35(2):75–86, 2015.

[33] Andrea Manca, David Moher, Lucia Cugusi, Zeevi Dvir, and Franca Deriu. How predatory journals leak into pubmed. *CMAJ*, 190(35):E1042–E1045, 2018.

[34] Iz Beltagy, Kyle Lo, and Arman Cohan. Scibert: A pretrained language model for scientific text. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3606–3611, 2019.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[36] Yixin Cao, Lei Hou, Juanzi Li, and Zhiyuan Liu. Neural collective entity linking. *arXiv:1811.08603*, 2018.

[37] Chenwei Zhang, Yaliang Li, Nan Du, Wei Fan, and Philip S Yu. On the generative discovery of structured medical knowledge. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2720–2728, 2018.

[38] Thomas Davenport and Ravi Kalakota. The potential for artificial intelligence in healthcare. *Future healthcare journal*, 6(2):94, 2019.

[39] Xi Victoria Lin, Richard Socher, and Caiming Xiong. Multi-hop knowledge graph reasoning with reward shaping. *arXiv:1808.10568*, 2018.

[40] ISO/IEC/IEEE 15288: 2015. Systems and software engineering–system life cycle processes. 2015.

[41] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and Philip S Yu. A survey on knowledge graphs: Representation, acquisition and applications. *arXiv preprint arXiv:2002.00388*, 2020.

[42] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific american*, 284(5):34–43, 2001.

[43] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1247–1250, 2008.

[44] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *The semantic web*, pages 722–735. Springer, 2007.

[45] D. Zhang, J. Yin, X. Zhu, and C. Zhang. Network representation learning: A survey. *IEEE Transactions on Big Data*, 6(1):3–28, 2020.

[46] Xisen Jin, Wenqiang Lei, Zhaochun Ren, Hongshen Chen, Shangsong Liang, Yihong Zhao, and Dawei Yin. Explicit state tracking with semi-supervisionfor neural dialogue generation. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 1403–1412, 2018.

[47] He He, Anusha Balakrishnan, Mihail Eric, and Percy Liang. Learning symmetric collaborative dialogue agents with dynamic knowledge graph embeddings. *arXiv preprint arXiv:1704.07130*, 2017.

[48] Hao Zhou, Tom Young, Minlie Huang, Haizhou Zhao, Jingfang Xu, and Xiaoyan Zhu. Commonsense knowledge aware conversation generation with graph attention. In *IJCAI*, pages 4623–4629, 2018.

[49] Introducing the knowledge graph: things, not strings. `https://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html`. Accessed: 2020-03-15.

[50] Xiang Wang, Dingxian Wang, Canran Xu, Xiangnan He, Yixin Cao, and Tat-Seng Chua. Explainable reasoning over knowledge graphs for recommendation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 5329–5336, 2019.

[51] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[52] Xiang Yue, Zhen Wang, Jingong Huang, Srinivasan Parthasarathy, Soheil Moosavinasab, Yungui Huang, Simon M Lin, Wen Zhang, Ping Zhang, and Huan Sun. Graph embedding on biomedical networks: methods, applications and evaluations. *Bioinformatics*, 36(4):1241–1251, 2020.

[53] Allan Peter Davis, Cynthia J Grondin, Robin J Johnson, Daniela Sciaky, Roy McMorran, Jolene Wiegers, Thomas C Wiegers, and Carolyn J Mattingly. The

comparative toxicogenomics database: update 2019. *Nucleic acids research*, 47(D1):D948–D954, 2019.

[54] David S Wishart, Yannick D Feunang, An C Guo, Elvis J Lo, Ana Marcu, Jason R Grant, Tanvir Sajed, Daniel Johnson, Carin Li, Zinat Sayeeda, et al. Drugbank 5.0: a major update to the drugbank database for 2018. *Nucleic acids research*, 46(D1):D1074–D1082, 2018.

[55] Damian Szklarczyk, Andrea Franceschini, Stefan Wyder, Kristoffer Forslund, Davide Heller, Jaime Huerta-Cepas, Milan Simonovic, Alexander Roth, Alberto Santos, Kalliopi P Tsafou, et al. String v10: protein–protein interaction networks, integrated over the tree of life. *Nucleic acids research*, 43(D1):D447–D452, 2015.

[56] Antoine Bordes, Jason Weston, Ronan Collobert, and Yoshua Bengio. Learning structured embeddings of knowledge bases. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[57] Blake Shaw and Tony Jebara. Structure preserving embedding. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 937–944, 2009.

[58] Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 601–610, 2014.

[59] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Advances in neural information processing systems*, pages 2787–2795, 2013.

[60] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM international on conference on information and knowledge management*, pages 891–900, 2015.

[61] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[62] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD interna-*

tional conference on Knowledge discovery and data mining, pages 701–710, 2014.

[63] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web*, 2015.

[64] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1225–1234, 2016.

[65] Predrag Radivojac, Wyatt T Clark, Tal Ronnen Oron, Alexandra M Schnoes, Tobias Wittkop, Artem Sokolov, Kiley Graim, Christopher Funk, Karin Verspoor, Asa Ben-Hur, et al. A large-scale evaluation of computational protein function prediction. *Nature methods*, 10(3):221–227, 2013.

[66] Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. Learning entity and relation embeddings for knowledge graph completion. In *Twenty-ninth AAAI conference on artificial intelligence*, 2015.

[67] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge graph embedding by translating on hyperplanes. In *Twenty-Eighth AAAI conference on artificial intelligence*, 2014.

[68] Dat Quoc Nguyen, Kairit Sirts, Lizhen Qu, and Mark Johnson. Stranse: a novel embedding model of entities and relationships in knowledge bases. *arXiv preprint arXiv:1606.08140*, 2016.

[69] Guoliang Ji, Kang Liu, Shizhu He, and Jun Zhao. Knowledge graph completion with adaptive sparse transfer matrix. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[70] Guoliang Ji, Shizhu He, Liheng Xu, Kang Liu, and Jun Zhao. Knowledge graph embedding via dynamic mapping matrix. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 687–696, 2015.

[71] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. *arXiv:1412.6575*, 2014.

[72] Seyed Mehran Kazemi and David Poole. Simple embedding for link prediction in knowledge graphs. In *Advances in neural information processing systems*, pages 4284–4295, 2018.

[73] Maximilian Nickel, Lorenzo Rosasco, and Tomaso Poggio. Holographic embeddings of knowledge graphs. In *Thirtieth Aaai conference on artificial intelligence*, 2016.

[74] Ivana Balažević, Carl Allen, and Timothy M Hospedales. Tucker: Tensor factorization for knowledge graph completion. *arXiv preprint arXiv:1901.09590*, 2019.

[75] Dai Quoc Nguyen, Tu Dinh Nguyen, Dat Quoc Nguyen, and Dinh Phung. A novel embedding model for knowledge base completion based on convolutional neural network. *arXiv preprint arXiv:1712.02121*, 2017.

[76] Ni Lao and William W Cohen. Relational retrieval using a combination of path-constrained random walks. *Machine learning*, 81(1):53–67, 2010.

[77] Matt Gardner, Partha Talukdar, Bryan Kisiel, and Tom Mitchell. Improving learning and inference in a large knowledge-base using latent syntactic cues. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 833–838, 2013.

[78] Arvind Neelakantan, Benjamin Roth, and Andrew McCallum. Compositional vector space models for knowledge base completion. *arXiv:1504.06662*, 2015.

[79] Rajarshi Das, Arvind Neelakantan, David Belanger, and Andrew McCallum. Chains of reasoning over entities, relations, and text using recurrent neural networks. *arXiv:1607.01426*, 2016.

[80] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[81] Zhongyi Han, Benzheng Wei, Yuanjie Zheng, Yilong Yin, Kejian Li, and Shuo Li. Breast cancer multi-classification from histopathological images with structured deep learning model. *Scientific reports*, 7(1):1–10, 2017.

[82] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[83] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys (CSUR)*, 52(1):1–38, 2019.

[84] Junshui Ma, Robert P Sheridan, Andy Liaw, George E Dahl, and Vladimir Svetnik. Deep neural nets as a method for quantitative structure–activity relationships. *Journal of chemical information and modeling*, 55(2):263–274, 2015.

[85] T Ciodaro, D Deva, JM De Seixas, and D Damazio. Online particle detection with neural networks based on topological calorimetry information. In *Journal of physics: conference series*, volume 368, page 012030. IOP Publishing, 2012.

[86] Claire Adam-Bourdarios, Glen Cowan, Cécile Germain, Isabelle Guyon, Balázs Kégl, and David Rousseau. The higgs boson machine learning challenge. In *NIPS 2014 Workshop on High-energy Physics and Machine Learning*, pages 19–55, 2015.

[87] Moritz Helmstaedter, Kevin L Briggman, Srinivas C Turaga, Viren Jain, H Sebastian Seung, and Winfried Denk. Connectomic reconstruction of the inner plexiform layer in the mouse retina. *Nature*, 500(7461):168–174, 2013.

[88] Michael KK Leung, Hui Yuan Xiong, Leo J Lee, and Brendan J Frey. Deep learning of the tissue-regulated splicing code. *Bioinformatics*, 30(12):i121–i129, 2014.

[89] Hui Y Xiong, Babak Alipanahi, Leo J Lee, Hannes Bretschneider, Daniele Merico, Ryan KC Yuen, Yimin Hua, Serge Gueroussov, Hamed S Najafabadi, Timothy R Hughes, et al. The human splicing code reveals new insights into the genetic determinants of disease. *Science*, 347(6218):1254806, 2015.

[90] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.

[91] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[92] Sankar K Pal and Sushmita Mitra. Multilayer perceptron, fuzzy sets, classifiaction. 1992.

[93] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[94] Waseem Rawat and Zenghui Wang. Deep convolutional neural networks for image classification: A comprehensive review. *Neural computation*, 29(9):2352–2449, 2017.

[95] David E Rumelhart, Paul Smolensky, James L McClelland, and G Hinton. Sequential thought processes in pdp models. *Parallel distributed processing: explorations in the microstructures of cognition*, 2:3–57, 1986.

[96] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[97] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[98] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[99] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(Jul):2121–2159, 2011.

[100] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv:1609.04747*, 2016.

[101] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra. Draw: A recurrent neural network for image generation. *arXiv:1502.04623*, 2015.

[102] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287, 1999.

[103] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Advances in neural information processing systems*, pages 2787–2795, 2013.

[104] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706, 2007.

[105] Zi Yin and Yuanyuan Shen. On the dimensionality of word embedding. In *Advances in Neural Information Processing Systems*, pages 887–898, 2018.

[106] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv:1301.3781*, 2013.

[107] D Randall Wilson and Tony R Martinez. The general inefficiency of batch training for gradient descent learning. *Neural networks*, 16(10):1429–1451, 2003.

[108] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[109] Rafael Müller, Simon Kornblith, and Geoffrey E Hinton. When does label smoothing help? In *Advances in Neural Information Processing Systems*, page 4705, 2019.

[110] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33, 2015.

[111] Dragomir R Radev, Hong Qi, Harris Wu, and Weiguo Fan. Evaluating web-based question answering systems. In *LREC*, 2002.

[112] Bhushan Kotnis and Vivi Nastase. Analysis of the impact of negative sampling on link prediction in knowledge graphs. *arXiv preprint arXiv:1708.06816*, 2017.

[113] W. Samek, A. Binder, G. Montavon, S. Lapuschkin, and K. Müller. Evaluating the visualization of what a deep neural network has learned. *IEEE Transactions on Neural Networks and Learning Systems*, 28(11):2660–2673, 2017.

[114] Yongqi Zhang, Quanming Yao, Yingxia Shao, and Lei Chen. Nscaching: Simple and efficient negative sampling for knowledge graph embedding. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 614–625. IEEE, 2019.

[115] Wojciech Samek. *Explainable AI: interpreting, explaining and visualizing deep learning*, volume 11700. Springer Nature, 2019.

Abbreviations

KG	Knowledge Graph
$UMLS$	Universal Medical Language System
MDP	Markov Decision Process
AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
ANN	Artificial Neural Network
RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
MRR	Mean Reciprocal Rank
NLP	Natural Language Processing
CPU	Central Processing Unit
GPU	Graphics Processing Unit
CNN	Convolutional Neural Network
$LSTM$	Long Short-Term Memory
RNN	Recurrent Neural Network
PRA	Path-Ranking Algorithm
PCA	Principal Component Analysis
SGD	Stochastic Gradient Descent